Numerical Analysis

A Learning Guide

Gustav Delius Eric Sullivan

2026-02-23

Table of contents

Introduction **1**
- What Is Numerical Analysis? 1
  - Example from Algebra 1
  - Example from Calculus 2
  - Example from Differential Equations 3
  - Reasons to study Numerical Analysis 3
- How this module works 5
- Assessment 7
- Textbooks 8
- Your jobs 8
- Python 9
- Google Colab 9
  - The use of AI 10
- About you 11
- References 12

1. Numbers **13**
- 1.1. Binary Numbers 14
- 1.2. Floating Point Numbers 18
- 1.3. Rounding Errors 22
- 1.4. Loss of Significant Digits 23
- 1.5. Exam-style question 25
- 1.6. Problems 26

2. Functions **29**
- 2.1. Polynomial Approximations 29
  - 2.1.1. Approximating the exponential function 31
  - 2.1.2. Taylor Series 34
- 2.2. Truncation Error 38
- 2.3. Exam-style question 41
- 2.4. Neural Networks 42
  - 2.4.1. A Single Neuron 42
  - 2.4.2. Combining Neurons into a Layer 44
  - 2.4.3. A Two-Layer Neural Network 45
  - 2.4.4. Deep Neural Networks 48
  - 2.4.5. Why Depth Matters: The Sawtooth Example 49

2.5. Problems 55

3. Roots 1 **59**
3.1. Introduction to Numerical Root Finding 59
3.2. The Bisection Method 61
3.2.1. Intuition 61
3.2.2. Implementation 64
3.2.3. Analysis 66
3.3. Fixed Point Iteration 73

4. Roots 2 **79**

5. Derivatives, Integrals **81**

6. Optimisation **83**

7. ODEs 1 **85**

8. ODEs 2 **87**

9. PDEs 1 **89**

10. PDEs 2 **91**

Appendices **93**

A. Python **93**
A.1. Why Python? 93
A.2. Python Programming Basics 94
A.2.1. Variables 94
A.2.2. Indexing and Lists 96
A.2.3. List Operations 98
A.2.4. Tuples 100
A.2.5. Control Flow: Loops and If Statements 100
A.2.6. Functions 107
A.2.7. Lambda Functions 110
A.2.8. Packages 111
A.3. Numerical Python with NumPy 113
A.3.1. Numpy Arrays, Array Operations, and Matrix Operations 115
A.3.2. `arange, linspace, zeros, ones`, and `meshgrid` 120
A.4. Plotting with Matplotlib 122
A.4.1. Basics with `plt.plot()` 122
A.4.2. Subplots 125
A.4.3. Logarithmic Scaling with `semilogy`, `semilogx`, and `loglog` 126

A.5. Dataframes with Pandas . . . 128
A.6. Problems . . . 129

B. Exam solutions 133
B.1. Numbers . . . 133

C. Linear Algebra 135
C.1. Intro to Numerical Linear Algebra . . . 135
C.2. Notation . . . 136
C.3. Vectors and Matrices in Python . . . 137
C.4. Matrix and Vector Operations . . . 141
C.4.1. The Dot Product . . . 141
C.4.2. Matrix Multiplication . . . 143
C.5. The LU Factorization . . . 148
C.5.1. A Recap of Row Reduction . . . 148
C.5.2. The LU Decomposition . . . 151
C.5.3. Solving Triangular Systems . . . 157
C.5.4. Solving Systems with LU Decomposition . . . 162
C.6. The QR Factorization . . . 163
C.7. Over-determined Systems and Curve Fitting . . . 173
C.8. The Eigenvalue-Eigenvector Problem . . . 176
C.9. Algorithm Summaries . . . 182
C.10. Problems . . . 183
C.11. Projects . . . 193
C.11.1. The Google Page Rank Algorithm . . . 193
C.11.2. Alternative Methods For Solving $Ax = b$ . . . 200

Introduction

What I cannot create, I do not understand.
–Richard P. Feynman

Mathematics is not just an abstract pursuit; it is an essential tool that powers a vast array of applications. From weather forecasting to black hole simulations, from urban planning to medical research, from ecology to epidemiology, the application of mathematics has become indispensable. Central to this applied force is Numerical Analysis.

What Is Numerical Analysis?

Numerical Analysis is the discipline that bridges continuous mathematical theories with their concrete implementation on digital computers. These computers, by design, work with discrete quantities, and translating continuous problems into this discrete realm is not always straightforward.

In this module, we will explore some key techniques, algorithms, and principles of Numerical Analysis that enable us to translate mathematical problems into computational solutions. We will delve into the challenges that arise in this translation, the strategies to overcome them, and the interaction of theory and practice.

Many mathematical problems cannot be solved analytically in closed form. In Numerical Analysis, we aim to find *approximation algorithms* for mathematical problems, i.e., schemes that allow us to compute the solution approximately. These algorithms use only elementary operations that computers know how to do $(+, -, \times, /)$, but often a long sequence of them, so that in practice they need to be run on computers.

Example from Algebra

Solve the equation

$$\log(x) = \sin(x)$$

for x in the interval $x \in (0, \pi)$. Stop and try using all of the algebra that you ever learned to find x. You will quickly realize that there are no by-hand techniques that can solve this problem! A numerical approximation, however, is not so hard to come by. The following graph shows that there is a solution to this equation somewhere between $x = 2$ and $x = 2.5$.

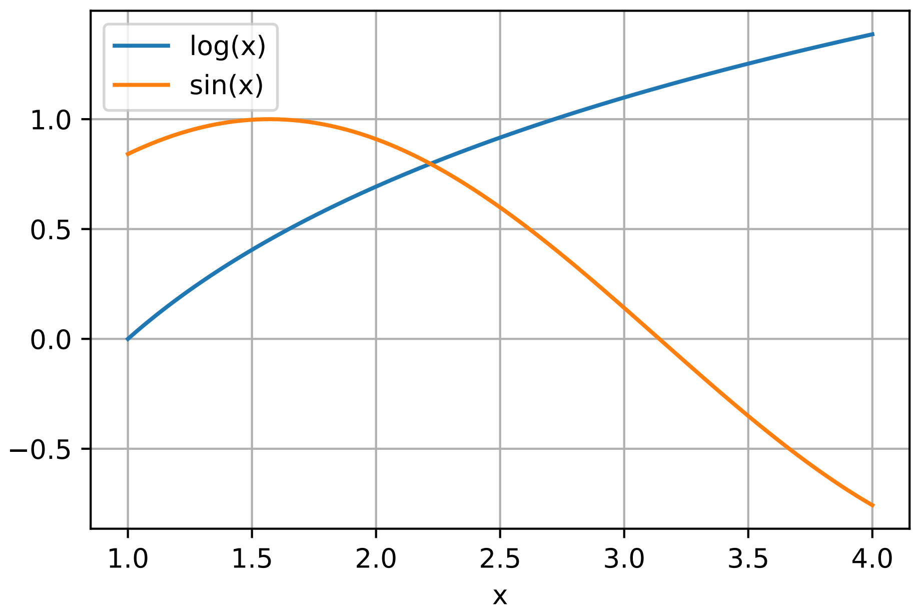


Figure 1.: The graphs of the real functions $\log(x)$ and $\sin(x)$ intersect at exactly one point, giving the solution to the equation $\log(x) = \sin(x)$.

Example from Calculus

What if we want to evaluate

$$\int_0^\pi \sin(x^2)dx?$$

Again, trying to use any of the possible techniques for using the Fundamental Theorem of Calculus, and hence finding an antiderivative, on the function $\sin(x^2)$ is completely hopeless. Substitution, integration by parts, and all of the other techniques that you know will all fail. Again, a numerical approximation is not so difficult and is very fast and gives the value

```
0.7726517138019184
```

By the way, this integral (called the Fresnel Sine Integral) actually shows up naturally in the field of optics and electromagnetism, so it is not just some arbitrary integral that was cooked up just for fun.

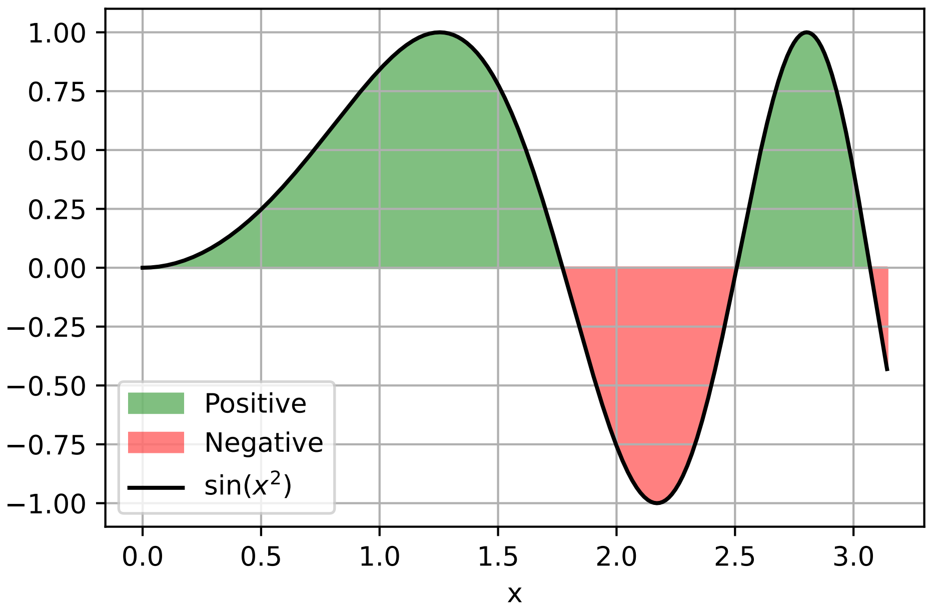


Figure 2.: Visual representation of the integral of $\sin(x^2)$ from 0 to π.

Example from Differential Equations

Say we needed to solve the differential equation

$$\frac{dy}{dt} = \sin(y^2) + t.$$

The nonlinear nature of the problem precludes us from using most of the typical techniques (e.g. separation of variables, undetermined coefficients, Laplace Transforms, etc). However, computational methods that result in a plot of an approximate solution can be made very quickly. Here is a plot of the solution up to time $t = 2.5$ with initial condition $y(0) = 0.1$:

This was an artificial example, but differential equations are central to modelling the real world in order to predict the future. They are the closest thing we have to a crystal ball. Here is a plot of a numerical solution of the SIR model of the evolution of an epidemic over time:

Reasons to study Numerical Analysis

So why should you want to venture into Numerical Analysis rather than just use the computer as a black box?

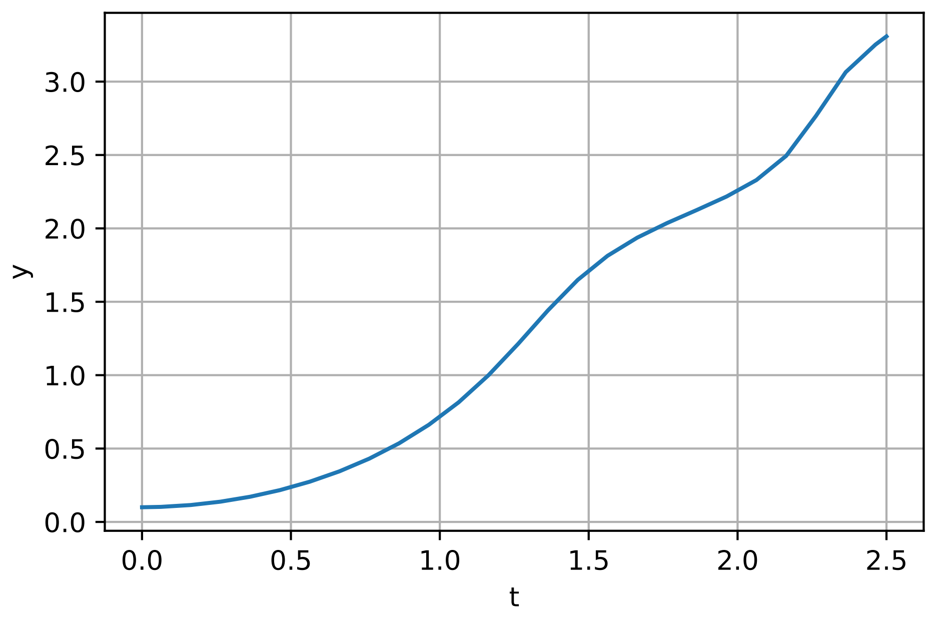


Figure 3.: Plot of numerical solution of $dy/dt = \sin(y^2) + t$ with $y(0) = 0.1$.

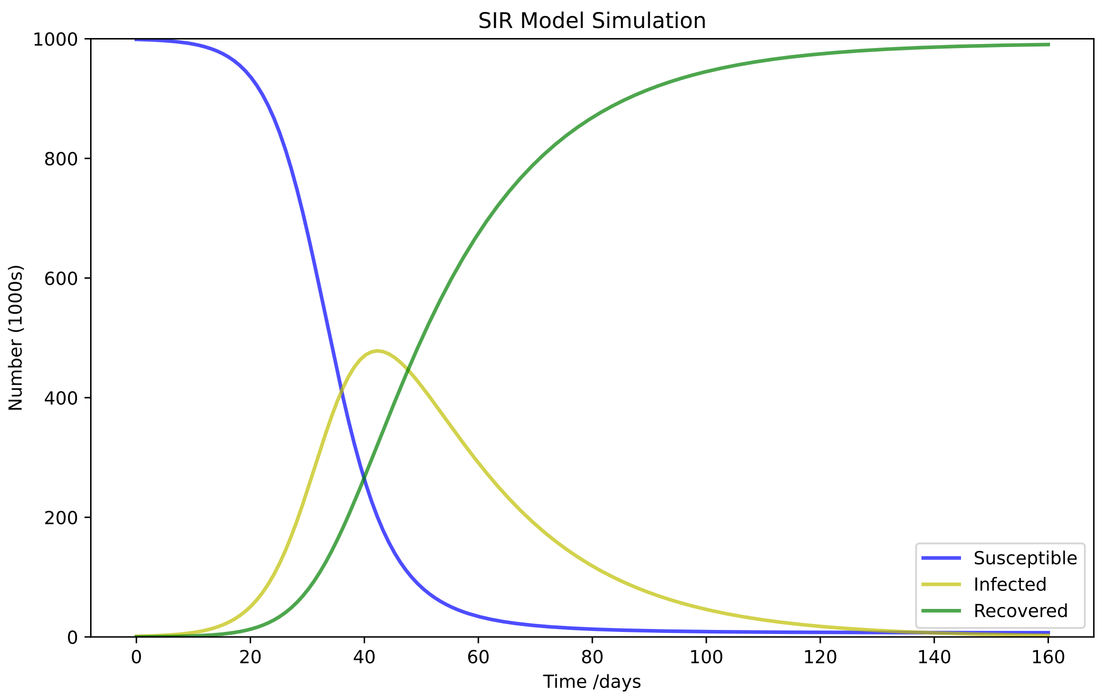


Figure 4.: Plot of a numerical solution of the SIR model

1. **Precision and Stability:** Computers, despite their power, can introduce significant errors if mathematical problems are implemented without care. Numerical Analysis offers techniques to ensure we obtain results that are both accurate and stable.
2. **Efficiency:** Real-world applications often demand not just correctness, but efficiency. By grasping the methods of Numerical Analysis, we can design algorithms that are both accurate and resource-efficient.
3. **Broad Applications:** Whether your interest lies in physics, engineering, biology, finance, or many other scientific fields, Numerical Analysis provides the computational tools to tackle complex problems in these areas.
4. **Basis for Modern Technologies:** Core principles of Numerical Analysis are foundational in emerging fields such as artificial intelligence, quantum computing, and data science.

The prerequisites for this material include a firm understanding of calculus and linear algebra and a good understanding of the basics of differential equations.

By the end of this module, you will not merely understand the methods of Numerical Analysis; you will be equipped to apply them efficiently and effectively in diverse scenarios: you will be able to tackle problems in physics, engineering, biology, finance, and many other fields; you will be able to design algorithms that are both accurate and resource-efficient; you will be able to ensure that your computational solutions are both accurate and stable; you will be able to leverage the power of computers to solve complex problems.

How this module works

There are 4 one-hour **whole-class sessions** every week. Three of these are listed on your timetable as "Lecture" and one as "Computer Practical". However, in all these sessions you, the student, are the one that is doing the work; discovering methods, writing code, working problems, leading discussions, and pushing the pace. I, the lecturer, will act as a guide who only steps in to redirect conversations or to provide necessary insight. You will use the whole-class sessions to share and discuss your work with the other members of your group. There will also be some whole-class discussions moderated by me.

You will find that this text is not a set of lecture notes. Instead it mostly just contains collections of exercises with minimal interweaving exposition. It is expected that you **do every one of the exercises** in the main body of each chapter and use the sequencing of the exercises to guide your learning and understanding.

Therefore the whole-class sessions form only a very small part of your work on this module. For each hour of whole-class work you should timetable yourself about two and a half hours of **work outside class** for working through the exercises on your own. I strongly recommend that you put those two and a half hours (ten hours spread throughout the week) into your timetable.

In order to enable you to get immediate feedback on your work also in between class sessions, I have made **feedback quizzes** where you can test your understanding of the material and your results from some of the exercises. Exercises that have an associated question in the feedback quiz are marked with a .

There are no traditional problem sheets in this module. In this module you will be working on exercises continuously throughout the week rather than working through a problem sheet only every other week.

At the end of each chapter there is a section entitled "**Problems**" that contains additional exercises aimed at consolidating your new understanding and skills. These are optional. Many of the chapters also have a section entitled "**Projects**". These projects are more open-ended investigations, designed to encourage creative mathematics, to push your coding skills and to require you to write and communicate mathematics. These projects are entirely optional and perhaps you will like to return to one of these even after the module has finished. If you do work on one of the projects, be sure to share your work with me at gustav.delius@york.ac.uk because will be very interested, also after the end of the module.

If you notice any mistakes or unclear things in the learning guide, please point them out to me in the class sessions or at gustav.delius@york.ac.uk. Many thanks go to Ben Mason and Toby Cheshire for the corrections they had sent in for a previous version of this guide.

You will need two **notebooks** for working through the exercises in this guide: one in **paper** form and one **electronic**. Some of the exercises are pen-and-paper exercises (and often these are marked with a pen icon) while others are coding exercises (often marked with a computer icon) and some require both writing or sketching and coding. You can keep your two notebooks linked through the numbering of the exercises. There are also some exercises that are marked with a Discussion icon . These are exercises that you should be discussing with other members of your group.

You will keep your coding notebooks in **Google Colab**, which we will discuss below. Most students find it easiest to have one dedicated Colab notebook per section, but some students will want to have one per chapter. You are highly encouraged to write explanatory text into your Google Colab notebooks as you go so that future-you can tell what it is that you were doing, which problem(s) you were solving, and what your thought processes were.

In the end, your collection of notebooks will contain solutions to every exercise in the guide and can serve as a reference manual for future numerical analysis problems. At the end of each of your notebooks you may also want to add a summary of what you have learned, which will both consolidate your learning and make it easier for you to remind yourself of your new skills later.

One request: do not share your notebooks publicly on the internet because that would create temptation for future students to not put in the work themselves, thereby robbing them of the learning experience.

If you have a **computer**, bring it along to the class sessions. However this is not a requirement. I will bring along some spare machines to make sure that every group has at least one computer to use during every session. The only requirements for a computer to be useful for this module is that it can connect to the campus WiFi, can run a web browser, and has a physical keyboard (typing code on virtual keyboards is too slow). The "Computer Practical" takes place in a PC classroom, so there will of course be plenty of machines available then.

Assessment

Unfortunately, your learning in the module also needs to be assessed. The final mark will be made up of 40% coursework and 60% final exam.

The **40% coursework** mark will come from **10 short quizzes** that will take place during the "Computer practical" in weeks 2 to 11. Answering each quiz should take less than 5 minutes but you will be given 16 minutes each to complete the quizzes in order to give you a large safety margin and remove stress. Late answers will not be accepted. The quizzes will be based on exercises that you will already have worked through and for which you will have had time to discuss them in class, so they will be really easy if you have engaged with the exercises as intended. Each quiz will be worth 5 points. There will be a practice quiz in the computer practical in week 1.

While working on the assessment quizzes you can already check your answers. If one of your answers is incorrect you can correct it and submit it again. However this will attract a penalty of 30% of the total mark for that answer. For multiple choice questions the penalty for a wrong answer may be even higher to discourage guessing. So it pays to be careful.

During the assessment quizzes you will be required to work exclusively on one of the PCs in the **computer practical room** rather than your own machine. This means in particular that you need to be physically present for the assessment. While working on the quiz you are only allowed to use a web browser, and the only pages you are allowed to have open are this guide, the quiz page on Moodle and any of your notebooks on Google Colab, with the AI features switched off. You are not allowed to use any AI assistants or other web pages. Besides your digital notebooks on Google Colab you may also use any hand-written notes as long as you have written them yourself.

To allow for the fact that there may be weeks in which you are ill or otherwise prevented from performing to your best in the assessment quizzes, your final coursework mark will be calculated as the average over your 8 best marks. If exceptional circumstances affect more than two of the 10 quizzes then you would need to submit an exceptional circumstances claim.

The **60% final exam** will be a 2 hour exam of the usual closed-book form in an exam room during the exam period. To prepare yourself for the final exam, there will be an exam style question at the end of each chapter, and I will make a practice exam available at the end of the module.

Textbooks

In this module we will only scratch the surface of the vast subject that is Numerical Analysis. The aim is for you at the end of this module to be familiar with some key ideas and to have the confidence to engage with new methods when they become relevant to you.

There are many textbooks on Numerical Analysis. Standard textbooks are (Burden and Faires 2010) and (Kincaid and Cheney 2009). They contain much of the material from this module. A less structured and more opinionated account can be found in (Acton 1990). Another well known reference that researchers often turn to for solutions to specific tasks is (Press et al. 2007). You will find many others in the library. They may go also under alternative names like “Numerical Methods” or “Scientific Computing”.

You may also want to look at textbooks for specific topics covered in this module, like for example (Butcher 2016) for methods for ordinary differential equations.

Your jobs

You have the following jobs as a student in this module:

1. **Fight!** You will have to fight hard to work through this material. The fight is exactly what we are after since it is ultimately what leads to innovative thinking.
2. **Screw Up!** More accurately, do not be afraid to screw up. You should write code, work problems, and develop methods, then be completely unafraid to scrap what you have done and redo it from scratch.
3. **Collaborate!** You should collaborate with your peers, both within your group and across the whole class. Discuss exercises, ask questions, help others.
4. **Enjoy!** Part of the fun of inquiry-based learning is that you get to experience what it is like to think like a true mathematician / scientist. It takes hard work but ultimately this should be fun!

Python

To properly understand numerical analysis, you will need to write code in order to experiment with the methods we discuss. You will be using Python for this purpose. or most of you, coding in Python will not be new. For example, you have probably seen it in your first year "Mathematical Programming & Skills" module. But perhaps you have never seen Python before. You will have some notion of what a programming language "is" and "does", but you may never have written any code. That is all right. You will pick it up as you go along.

Appendix A covers some of the basics of Python programming that we will use in this module. If you are new to Python, don't feel that you need to work through this appendix in one go. Instead, spread the work over the first two weeks of the course and interlace it with your work on the first two chapters.

Google Colab

Every computer is its own unique flower with its own unique requirements. Hence, we will not spend time here giving you all of the ways that you can install Python and all of the associated packages necessary for this module. Instead I ask that you use the Google Colab notebook tool for writing and running your Python code: https://colab.research.google.com.

Google Colab allows you to keep all of your Python code on your Google Drive. The Colab environment is a free and collaborative version of the popular Jupyter notebook project. Jupyter notebooks allow you to write and test code as well as to mix writing (including LaTeX formatting) in along with your code and your output. I recommend that if you are new to Google Colab, you start by watching the brief introductory video.

Now the time has come for the first two exercises of this module.

Exercise 0.1. Spend a bit of time poking around in Colab. Figure out how to

- Create new Colab notebooks.
- Add and delete code cells.
- Type a simple calculation like `1+1` into a code cell and evaluate it.
- Add and delete text cells.
- Add an equation to a text cell using LaTeX notation.
- Save a notebook to your Google Drive.
- Open a notebook from Google Drive.

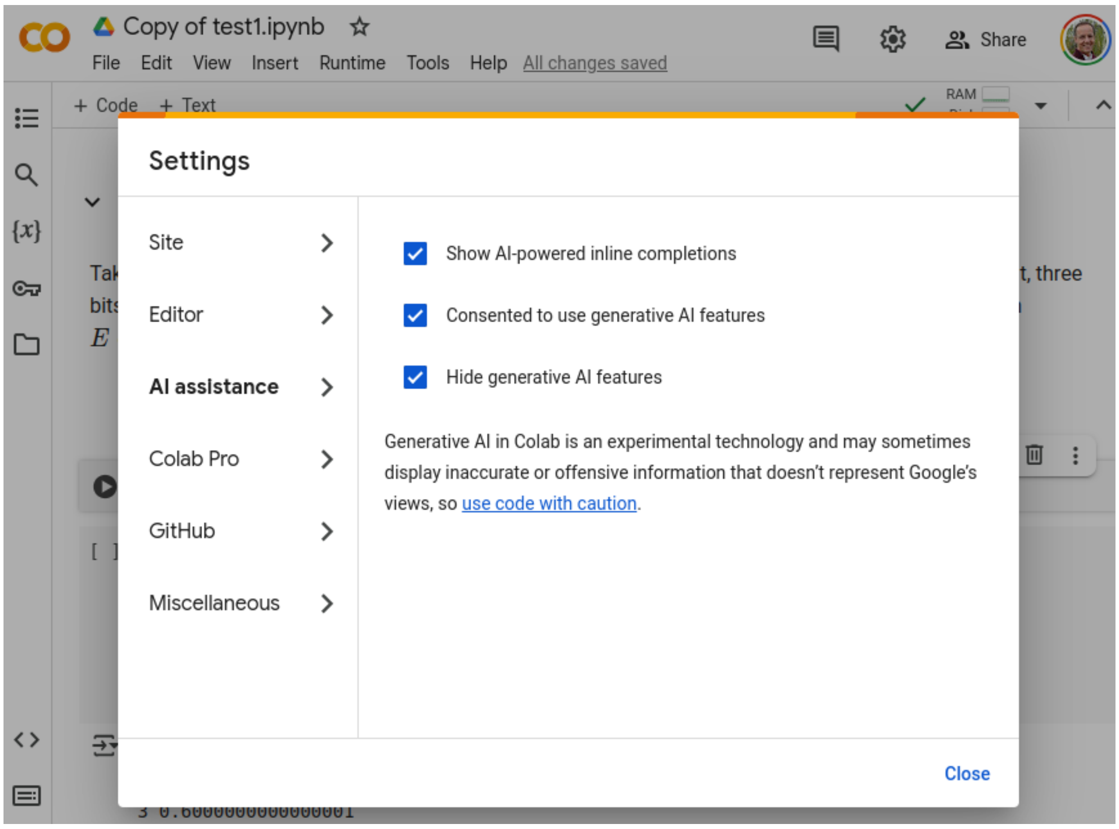
- Share a notebook with other members of your group and see if you can collaboratively edit it.

Exercise 0.2. Click on this link to a Colab notebook. It should open it in Colab. Then save a **copy** of it to your Google Drive. You need a copy because you will not have permission to edit the original. Follow the instructions in the notebook.

The use of AI

You will have gathered from the previous exercise that in this module you are not only allowed to use AI, **you are encouraged to use AI**. However you have probably already discovered that you get more out of an AI if you are already familiar with the basic concepts of a subject. You will need to be able to understand and check any answer an AI gives you. If there is something in an AI answer that is not totally clear or not obviously correct, always ask the AI to explain the details of its answer and ask follow-on questions until everything is crystal-clear.

During the 10 assessment quizzes you will not be allowed to use any AI. In particular you will be required to switch off the AI features in Google Colab. It is thus a good idea when working on practice exercises to also switch off the AI features to make sure you know what you are doing even when there is no AI assistance. To switch off the AI features you should tick the "Hide generative AI" checkbox on the "AI assistance" tab of the "Settings" page in Google Colab.



Settings

- [✓] Show AI-powered inline completions
- [✓] Consented to use generative AI features
- [✓] Hide generative AI features

Generative AI in Colab is an experimental technology and may sometimes display inaccurate or offensive information that doesn't represent Google's views, so use code with caution.

About you

Exercise 0.3. I am interested to learn more about you to help me make this module work well for you. It would be helpful if you would answer the questionnaire at https://forms.gle/YHHExaAnVzcLtBcU7. Of course you are free to share as much or as little as you like.

References

1. Numbers

We think in generalities, but we live in details.
–Alfred North Whitehead

Have you ever wondered how computers, which operate in a realm of zeros and ones, manage to perform mathematical calculations with real numbers? The secret lies in approximation.

In this chapter and the next we will investigate the foundations that allow a computer to do mathematical calculations at all. How can it store real numbers? How can it calculate the values of mathematical functions? We will understand that the computer can do these things only approximately and will thus always make errors. Numerical Analysis is all about keeping these errors as small as possible while still being able to do efficient calculations.

We will meet the two kinds of errors that a computer makes: **rounding errors** and **truncation errors**. Rounding errors arise from the way the computer needs to approximate real numbers by binary floating point numbers, which are the numbers it know how to add, subtract, multiply and divide. We'll discuss this in this chapter. Truncation errors arise from the way the computer needs to reduce all calculations to a finite number of these four basic arithmetic operations. We will see that for the first time in Chapter 2 when we discuss how computers approximate functions by power series and then have to truncate these at some finite order.

Let's start with a striking example of how bad computers actually are at doing even simple calculations:

Exercise 1.1. By hand (no computers!) compute the first few terms of the sequence

$$x_{n+1} = \begin{cases} 2x_n, & x_n \in [0, \frac{1}{2}) \\ 2x_n - 1, & x_n \in [\frac{1}{2}, 1] \end{cases} \tag{1.1}$$

with the initial condition $x_0 = 1/10$. Calculate enough terms so that you can see a pattern.

Exercise 1.2. Now use a spreadsheet to do the computations. Do you get the same answers?

Exercise 1.3. Finally, solve this problem with Python. Some starter code is given to you below.

```python
x = 1.0/10
for n in range(50):
    if x< 0.5:
        # put the correct assignment here
    else:
        # put the correct assigment here
    print(x)
```

(Even if you don't know Python, you should be able to do this exercise after having read up to Section A.2.1 in the chapter on Essential Python.)

In the previous exercise it seems like the computer has failed you! What do you think happened on the computer and why did it give you a different answer? What, do you suppose, is the cautionary tale hiding behind the scenes with this problem?

Exercise 1.4. Now calculate the sequence that you get when starting with $x_0 = 1/8$? Do you again get a different answer from the computer than when doing the calculations by hand?

1.1. Binary Numbers

A computer circuit knows two states: on and off. As such, anything saved in computer memory is stored using base-2 numbers. This is called a binary number system. To fully understand a binary number system it is worthwhile to pause and reflect on our base-10 number system for a few moments.

What do the digits in the number "735" really mean? The position of each digit tells us something particular about the magnitude of the overall number. The number 735 can be represented as a sum of powers of 10 as

$$735 = 700 + 30 + 5 = 7 \times 10^2 + 3 \times 10^1 + 5 \times 10^0 \tag{1.2}$$

and we can read this number as 7 hundreds, 3 tens, and 5 ones.

Now let us switch to the number system used by computers: the binary number system. In a binary number system the base is 2 so the only allowable digits are 0 and 1 (just like in base-10 the allowable digits were 0 through 9). In binary (base-2), the number "101,101" can be interpreted as

$$101,101_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \tag{1.3}$$

(where the subscript "2" indicates the base). If we put this back into base 10, so that we can read it more comfortably, we get

$$101,101_2 = 32 + 0 + 8 + 4 + 0 + 1 = 45_{10}.$$

(The commas in the numbers are only to allow for greater readability – we can easily see groups of three digits and mentally keep track of what we are reading.)

Exercise 1.5. Express the following binary numbers in base-10.

1. 111_2
2. $10,101_2$
3. $1,111,111,111_2$

For the last one you can save yourself some work by noticing that $1,111,111,111_2 = 10,000,000,000_2 - 1$. This is a generally useful trick that will help you again in some later exercises.

Exercise 1.6. Explain the joke: *There are 10 types of people. Those who understand binary and those who do not.*

Exercise 1.7. Discussion: With your group, discuss how you would convert a base-10 number into its binary representation, without using a calculator or computer. Once you have a proposed method put it into action on the number 237_{10} to show that the base-2 expression is $11,101,101_2$.

Exercise 1.8. By hand, using the methods you developed above, convert the following numbers from base 10 to base 2 or visa versa.

- Write 12_{10} in binary
- Write 11_{10} in binary
- Write 23_{10} in binary
- Write 11_2 in base 10
- What is 100101_2 in base 10?

The icons indicate that you should enter your answers into the feedback quiz. This gives you feedback on whether or not your answer is correct, but just as importantly, it lets your lecturer know how you are progressing with the material. The feedback quizzes are not used for assessment purposes, solely for feedback.

Exercise 1.9. Write down an explanation of the technique that your group has come up with to do the conversion from base 10 to base 2.

Next we will work with fractions and decimals.

Example 1.1. Let us take the base 10 number 5.341_{10} and expand it out to get

$$5.341_{10} = 5 + \frac{3}{10} + \frac{4}{100} + \frac{1}{1000} = 5 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} + 1 \times 10^{-3}.$$

The position to the right of the decimal point is the negative power of 10 for the given position.

We can do a similar thing with binary decimals.

Exercise 1.10. The base-2 number $1,101.01_2$ can be expanded in powers of 2. Fill in the question marks below and observe the pattern in the powers.

$$1,101.01_2 = ? \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + ? \times 2^0 + 0 \times 2^? + 1 \times 2^{-2}.$$

Example 1.2. Convert 11.01011_2 to base 10.
Solution:

$$\begin{aligned} 11.01011_2 &= 2 + 1 + \frac{0}{2} + \frac{1}{4} + \frac{0}{8} + \frac{1}{16} + \frac{1}{32} \\ &= 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-5} \\ &= 3.34375_{10}. \end{aligned}$$

Exercise 1.11. Convert the base 10 decimal 0.15625_{10} to binary using the following steps.

1. Multiply 0.15625 by 2. The whole number part of the result is the first binary digit to the right of the decimal point.
2. Take the result of the previous multiplication and ignore the digit to the left of the decimal point. Multiply the remaining decimal by 2. The whole number part is the second binary decimal digit.
3. Repeat the previous step until you have nothing left.

Explain to each other in the group why each step gives the binary digit that it does.

Exercise 1.12. Convert the base 10 fraction 0.1 into binary. Use this to explain why the computer made errors when it calculated with this number in Exercise 1.3.

1.2. Floating Point Numbers

In this section we will discuss how a computer actually stores a number. More specifically, since computers only have finite memory, we would really like to know the full range of numbers that are possible to store in a computer. Clearly, given the uncountable nature of the real numbers, there will be gaps between the numbers that can be stored. We would like to know what gaps in our number system to expect when using a computer to store and do computations on numbers. For this it is important to know that computers store numbers in a way that is similar to how we write numbers in scientific notation.

Example 1.3. Let us start the discussion with a very concrete example. Consider the number $x = -123.15625$ (in base 10). As we have seen this number can be converted into binary. Indeed

$$x = -123.15625_{10} = -1111011.00101_2$$

(you should check this).

If a computer needs to store this number then first they put in the binary version of scientific notation. In this case this will be

$$x = -1.111011001 01_2 \times 2^6.$$

This is the **floating point representation** of the number.

Definition 1.1. For any non-zero base-2 number x the **floating point representation** is given by

$$x = (-1)^s \times (1 + m) \times 2^E$$

where $s \in \{0, 1\}$, m is a binary number such that $0 \leq m < 1$, and E is an integer.

The number $1 + m$ is called the **significand**, s is known as the **sign bit**, and E is known as the **exponent**. We will refer to m, the fractional part of the significand that actually contains the information, as the **mantissa**, but this use is not universal.

To allow for both very large and very small numbers, the exponent E can be positive or negative. However, inside the computer it is efficiently stored as an unsigned integer e called the **biased exponent**. The true exponent E is obtained by subtracting a fixed **bias** B from e:

$$E = e - B.$$

The bias is chosen to be roughly half of the maximum possible value of the stored exponent e.

Example 1.4. What are the mantissa, sign bit, and unbiased exponent for the numbers $7_{10}, -7_{10}$, and $(0.1)_{10}$?
Solution:

- For the number $7_{10} = 111_2 = 1.11 \times 2^2$ we have $s = 0, m = 0.11$ and $E = 2$.
- For the number $-7_{10} = 111_2 = -1.11 \times 2^2$ we have $s = 1, m = 0.11$ and $E = 2$.
- For the number $\frac{1}{10} = 0.000110011001100\cdots = 1.100110011\cdots \times 2^{-4}$ we have $s = 0, m = 0.100110011\cdots$, and $E = -4$.

In the last part of the previous example we saw that the number $(0.1)_{10}$ is actually a repeating decimal in base-2. This means that in order to completely represent the number $(0.1)_{10}$ in base-2 we need infinitely many decimal places. Obviously that cannot happen since we are dealing with computers with finite memory. Each number can only be allocated a finite number of bits. Thus the number needs to be rounded to the nearest number that can be represented with that number of bits. That leads to an error called the **rounding error** (sometimes also called *roundoff error*). We'll look into these in more detail in Section 1.3 below.

Definition 1.2. Machine precision is the gap between the number 1 and the next larger floating point number. Often it is represented by the symbol ϵ. To clarify: the number 1 can always be stored in a computer system exactly and if ϵ is machine precision for that computer then $1 + \epsilon$ is the next largest number that can be stored with that machine.

For all practical purposes the computer cannot tell the difference between two numbers if the relative difference is smaller than machine precision. It is important to remember this when you want to check the equality of two numbers in a computer.

Exercise 1.13. To make all of these ideas concrete let us play with a small computer system where each number is stored in the following format, using 6 bits:

$$s\, e_1\, e_2\, b_1\, b_2\, b_3$$

The first bit is for the sign ($0 = +$ and $1 = -$). The next two bits, e_1 and e_2 are for the biased exponent, and we will assume in this example that the bias is $B = 1$. The three bits on the right represent the significand of the number. Hence, every number in this number system takes the form

$$(-1)^s \times (1 + 0.b_1 b_2 b_3) \times 2^{e_1 e_2 - 1}$$

- What is the smallest positive number that can be represented in this form?
- What is the largest positive number that can be represented in this form?
- What is the machine precision in this number system?

Over the course of the past several decades there have been many systems developed to properly store numbers. The IEEE standard that we now use is the accumulated effort of many computer scientists, much trial and error, and deep scientific research. We now have two standard precisions for storing numbers on a computer: single and double precision. The double precision standard is what most of our modern computers use.

Definition 1.3. According to the IEEE 754 standard:

- A **single-precision** number consists of 32 bits, with 1 bit for the sign, 8 for the exponent, and 23 for the mantissa. The bias is $B = 127$.
- A **double-precision** number consists of 64 bits with 1 bit for the sign, 11 for the exponent, and 52 for the mantissa. The bias is $B = 1023$.

Exercise 1.14. What are the largest numbers that can be stored in single and double precision?

Exercise 1.15. What is machine precision for the single and double precision standard?

Exercise 1.16. What is the gap between 2^n and the next largest number that can be stored in double precision?

Exercise 1.17. Computers contain hardware that can perform the basic operations of addition, subtraction, multiplication, and division on floating point numbers. To get a feel for what goes on under the hood, figure out how to add the numbers $x = 1.010_2 \times 2^3$ and $y = 1.110_2 \times 2^1$. How do you deal with the different exponents? What do you do so that the result is in the correct floating point format?

Much more can be said about floating point numbers such as how we store infinity, how we store NaN, and how we store 0. The Wikipedia page for floating point arithmetic might be of interest for the curious reader. It is beyond the scope of this module to go into all of those details here.

The biggest takeaway points from this section and the previous are:

- Real numbers are stored with finite precision in a computer.
- Nice rational numbers like 0.1 are sometimes not machine representable in binary.
- Machine precision is the gap between 1 and the next largest number that can be stored.
- The gap between one number and the next grows in proportion to the number.

1.3. Rounding Errors

When the binary representation of a real number has too many binary digits to be represented faithfully by a floating point number, we need to round it to the nearest floating point number that can be represented. That introduces rounding errors. We have seen above that the gap between two consecutive floating point numbers grows in proportion to the number. This means that the relative error of rounding is bounded by $\epsilon/2$ where ϵ is the machine precision.

The rounding rule that is used is "**round to nearest, ties to even**", which means that if the number is exactly halfway between two numbers that can be represented then we round the mantissa to an even binary number, i.e., to a mantissa that ends in 0.

Example 1.5. If we want to store the number $1.625 = 1.101_2$ in a floating point number system where the mantissa has only 2 bits then we round to $1.10_2 = 1.5_{10}$ because 1.101_2 is exactly halfway between 1.100_2 and 1.110_2 and the rounding rule is "round to nearest, ties to even".

To dive a little deeper into what happened in Exercise 1.3, simplify the detailed analysis by working with only a 4 bit mantissa:

Exercise 1.18. Write down how the number 1/10 is represented in a floating point number system where the mantissa has only 4 bits. Then calculate the first 10 terms of the sequence

$$x_{n+1} = \begin{cases} 2x_n, & x_n \in [0, \frac{1}{2}] \\ 2x_n - 1, & x_n \in (\frac{1}{2}, 1] \end{cases} \quad \text{with} \quad x_0 = \frac{1}{10} \tag{1.4}$$

using this number system.

Exercise 1.19. (This problem is modified from (Greenbaum and Chartier 2012))

Sometimes floating point arithmetic does not work like we would expect (and hope) as compared to exact mathematics. In each of the following problems we have a mathematical problem that the computer gets wrong. Explain why the computer is getting these wrong.

1. Mathematically we know that $\sqrt{5}^2$ should just give us 5 back. In Python type `np.sqrt(5)**2 == 5`. What do you get and why do you get it?
2. Mathematically we know that $49 \cdot \left(\frac{1}{49}\right)$ should just be 1. In Python type `49*(1/49) == 1`. What do you get and why do you get it?

3. Mathematically we know that $e^{\log(3)}$ should just give us 3 back. In Python type `np.exp(np.log(3)) == 3`. What do you get and why do you get it?
4. Create your own example of where Python gets something incorrect because of floating point arithmetic.

1.4. Loss of Significant Digits

As we have discussed, when representing real numbers by floating point numbers in the computer, rounding errors will usually occur. When doing a calculation with double-precision floating point numbers then the rounding error is only a tiny fraction of the actual number, so one might think that they really don't matter. However, calculations usually involve a number of steps, and we saw in Exercise 1.3 that the rounding errors can accumulate and become quite noticeable after a large number of steps.

But the problem is even worse. If we are not careful, then the rounding errors can get magnified already after very few steps if we perform the steps in an unfortunate way. The following examples and exercises will illustrate this.

Example 1.6. Consider the expression

$$(10^{10} + 0.123456789) - 10^{10}.$$

Mathematically, this is strictly equal to

$$(10^{10} - 10^{10}) + 0.123456789 = 0.123456789.$$

However, let us evaluate this in Python:

```
0.12345695495605469
```

Only the first six digits after the decimal point were preserved, the other digits were replaced by something seemingly random. The reason should be clear. The computer makes a rounding error when it tries to store the 10000000000.123456789. This is known as the loss of significant digits. It occurs whenever you subtract two almost equal numbers from each other.

Exercise 1.20. Consider these two mathematically equivalent ways to compute the same thing:

1) $(a+b)-c$
2) $a+(b-c)$

a) Why might these give different results in floating-point arithmetic?
b) If a is very small compared to b and c, which form would you expect to be more accurate? Why?

Exercise 1.21. Consider the trigonometric identity

$$2\sin^2(x/2) = 1-\cos(x).$$

It gives us two different methods to calculate the same quantity. Ask Python to evaluate both sides of the identity when $x = 0.0001$. Hint: as described in Section A.2.8, use `import math` so that you can then use `math.cos()` and `math.sin()`. Also remember that exponentiation in Python is represented by `**`.

What do you observe? If you want to calculate $1-\cos(x)$ with the highest precision, which expression would you use? Discuss.

Exercise 1.22. You know how to find the solutions to the quadratic equation

$$ax^2+bx+c=0.$$

You know the quadratic formula. For the larger of the two solutions the formula is

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

Let's assume that the parameters are given as

$$a=1, \quad b=1000000, \quad c=1.$$

Use the quadratic formula to find the larger of the two solutions, by coding the formula up in Python. You should get a solution slightly smaller than -10^{-6}. Hint: use `math.sqrt()` to code up the square root.

Then check whether your value for x really does solve the quadratic equation by evaluating ax^2+bx+c with your value of x. You will notice that it does not work. Discuss the cause of the error.

Now, on a piece of paper, rearrange the quadratic formula for the larger solution by multiplying both the numerator and denominator by $-b-\sqrt{b^2-4ac}$ and then simplify by multiplying out the resulting numerator. This should give you the alternative formula

$$x = \frac{2c}{-b-\sqrt{b^2-4ac}}.$$

Can you see why this expression will work better for the given parameter values? Again evaluate x with Python and then check it by substituting into the quadratic expression. What do you find?

Exercise 1.23. Google the term "catastrophic cancellation" to find more examples of this phenomenon of loss of significant digits.

These exercises will give much material for in-class discussion. The aim is to make you sensitive to the issue of loss of significant figures and the fact that expressions that are mathematically equal are not always computationally equal.

1.5. Exam-style question

Consider a hypothetical "8-bit Mini-Float" system based on the IEEE 754 standard. The 8 bits are allocated as follows:

- **Sign bit (s):** 1 bit (Bit 7)
- **Exponent (e):** 3 bits (Bits 6-4), using a bias of 3.
- **Mantissa (m):** 4 bits (Bits 3-0), normalized with an implied leading 1.

The value of a number in this system is given by: $x = (-1)^s \times (1.m)_2 \times 2^{e-3}$.

(a) How is the **machine precision** ϵ defined. Give its value for this floating-point system. How is machine precision related to rounding errors? [3 marks]

(b) Convert the decimal number **13.5** into this 8-bit floating-point representation. Write your final answer as an 8-bit binary pattern (e.g., `0 101 1010`). [4 marks]

(c) Using this specific floating-point system, perform the addition of the number **13.5** (from part b) and **0.25**.

  - Write 0.25 in this floating point system.

- Perform the addition simulating the hardware: align exponents, add significands.
- Apply the rounding rule (“Round to Nearest, Ties to Even”) to determine the final stored bits.
- What is the final decimal value stored by the system, and what is the absolute error compared to the exact mathematical sum? [4 marks]

(d) A student attempts to calculate the function $f(x) = \sqrt{x^2+1} - x$ for a very large value $x = 10^8$.

- Explain why the result computed by a standard computer might be inaccurate (specifically naming the type of error).
- Propose an algebraically equivalent formula for $f(x)$ that avoids this error. [3 marks]

1.6. Problems

These problem exercises will let you consolidate what you have learned so far and combine it with your Python coding skills, see Appendix A.

Exercise 1.24. (This problem is modified from (Greenbaum and Chartier 2012))

In the 1999 film *Office Space*, a character creates a program that takes fractions of cents that are truncated in a bank’s transactions and deposits them to his own account. This idea has been attempted in the past and now banks look for this sort of thing. In this problem you will build a simulation of the program to see how long it takes to become a millionaire.

Assumptions:

- Assume that you have access to 50,000 bank accounts.
- Assume that the account balances are uniformly distributed between \$100 and \$100,000.
- Assume that the annual interest rate on the accounts is 5% and the interest is compounded daily and added to the accounts, except that fractions of cents are truncated.
- Assume that your illegal account initially has a \$0 balance.

Your Tasks:

1. Explain what the code below does.

```python
import numpy as np
accounts = 100 + (100000-100) * np.random.rand(50000,1);
accounts = np.floor(100*accounts)/100;
```

2. By hand (no computer) write the mathematical steps necessary to increase the accounts by (5/365)% per day, truncate the accounts to the nearest penny, and add the truncated amount into an account titled "illegal."
3. Write code to complete your plan from part 2.
4. Using a `while` loop, iterate over your code until the illegal account has accumulated \$1,000,000. How long does it take?

Exercise 1.25. (This problem is modified from (Greenbaum and Chartier 2012))

In the 1991 Gulf War, the Patriot missile defence system failed due to rounding error. The troubles stemmed from a computer that performed the tracking calculations with an internal clock whose integer values in tenths of a second were converted to seconds by multiplying by a 24-bit binary approximation to $\frac{1}{10}$:

$$0.1_{10} \approx 0.00011001100110011001100_2. \tag{1.5}$$

1. Convert the binary number above to a fraction by hand.
2. The approximation of $\frac{1}{10}$ given above is clearly not equal to $\frac{1}{10}$. What is the absolute error in this value?
3. What is the time error, in seconds, after 100 hours of operation?
4. During the 1991 war, a Scud missile travelled at approximately Mach 5 (3750 mph). Find the distance that the Scud missile would travel during the time error computed in part 3.

Exercise 1.26 (The Python Caret Operator)**.** Now that you're used to using Python to do some basic computations you are probably comfortable with the fact that the caret, ^, does NOT do exponentiation like it does in many other programming languages. But what does the caret operator do? That's what we explore here.

1. Consider the numbers 9 and 5. Write these numbers in binary representation. We are going to use four bits to represent each number (it is OK if the first bit happens to be zero).

$$\begin{aligned} 9 &= \underline{\quad}\,\underline{\quad}\,\underline{\quad}\,\underline{\quad} \\ 5 &= \underline{\quad}\,\underline{\quad}\,\underline{\quad}\,\underline{\quad} \end{aligned} \tag{1.6}$$

2. Now go to Python and evaluate the expression `9^5`. Convert Python's answer to a binary representation (again using four bits).
3. Make a conjecture: How do we go from the binary representations of a and b to the binary representation for Python's `a^b` for numbers a and b? Test and verify your conjecture on several different examples and then write a few sentences explaining what the caret operator does in Python.

2. Functions

How does a computer *understand* a function like $f(x) = e^x$ or $f(x) = \sin(x)$ or $f(x) = \log(x)$? What happens under the hood, so to speak, when you ask a computer to do a computation with one of these functions? A computer is good at arithmetic operations, but working with transcendental functions like these, or really any other sufficiently complicated functions for that matter, is not something that comes naturally to a computer. What is actually happening under the hood is that the computer only approximates the functions.

2.1. Polynomial Approximations

A class of functions that computers are very good at working with are polynomial functions. This is because to evaluate a polynomial function at any point we only need to addition and multiplication operations. In this section we will explore how we can use polynomial functions to approximate other functions.

Exercise 2.1. In this exercise you are going to make a bit of a wish list for all of the things that a computer will do when approximating a function. We are going to complete the following sentence:
If we are going to approximate a smooth function $f(x)$ near the point $x = x_0$ with a simpler function $g(x)$ then ...

(I will get us started with the first two things that seems natural to wish for. The rest of the wish list is for you to complete.)

- the functions $f(x)$ and $g(x)$ should agree at $x = x_0$. In other words, $f(x_0) = g(x_0)$
- the function $g(x)$ should only involve addition, subtraction, multiplication, division, and integer exponents since computer are very good at those sorts of operations.
- if $f(x)$ is increasing / decreasing near $x = x_0$ then $g(x)$ ...
- if $f(x)$ is concave up / down near $x = x_0$ then $g(x)$...
- if we zoom into plots of the functions $f(x)$ and $g(x)$ near $x = x_0$ then ...

- ... is there anything else that you would add?

Exercise 2.2. Discuss: Could a polynomial function with a high enough degree satisfy everything in the wish list from the previous problem? Explain your reasoning.

Exercise 2.3. Let us put some parts of the wish list into action. If $f(x)$ is a differentiable function at $x = x_0$ and if $g(x) = A + B(x - x_0) + C(x - x_0)^2 + D(x - x_0)^3$ then

1. What is the value of A such that $f(x_0) = g(x_0)$? *(Hint: substitute* $x = x_0$ into the $g(x)$ function)
2. What is the value of B such that at x_0 f and g have the same slope? In other words, what is the value of B such that $f'(x_0) = g'(x_0)$? *(Hint: Start by taking the derivative of* $g(x)$*)*
3. What is the value of C such that at x_0 f and g have the same concavity? In other words, what is the value of C such that $f''(x_0) = g''(x_0)$?
4. What is the value of D such that at x_0 f and g have the same third derivative? In other words, what is the value of D such that $f'''(x_0) = g'''(x_0)$?

In the previous 3 exercises you have built up some basic intuition for what we would want out of a mathematical operation that might build an approximation of a complicated function. What we have built is actually a way to get better and better approximations for functions out to pretty much any arbitrary accuracy that we like so long as we are near some anchor point (which we called x_0 in the previous exercises).

In the next several problems you will unpack the polynomial approximations of $f(x) = e^x$ and we will wrap the whole discussion with a little bit of formal mathematical language. Then we will examine other functions like $\sin(x)$ and $\log(x)$. One of the points of this whole discussion is to give you a little glimpse as to what is happening behind the scenes in scientific programming languages when you do computations with these functions. A bigger point is to start getting a feel for how we might go in reverse and approximate an unknown function out of much simpler parts. This last goal is one of the big takeaways from numerical analysis: *we can mathematically model highly complicated functions out of fairly simple pieces.*

2.1.1. Approximating the exponential function

Exercise 2.4. What is Euler's number e? You have been using this number often in Calculus and Differential Equations. Do you know the decimal approximation for this number? Moreover, is there a way that we could approximate something like $\sqrt{e} = e^{0.5}$ or e^{-1} without actually having access to the full decimal expansion?

For all of the questions below let us work with the function $f(x) = e^x$.

1. The function $g_0(x) = 1$ matches $f(x) = e^x$ exactly at the point $x = 0$ since $f(0) = e^0 = 1$. Furthermore if x is very very close to 0 then the functions $f(x)$ and $g_0(x)$ are really close to each other. Hence we could say that $g_0(x) = 1$ is an approximation of the function $f(x) = e^x$ for values of x very very close to $x = 0$. Admittedly, though, it is probably pretty clear that this is a horrible approximation for any x just a little bit away from $x = 0$.
2. Let us get a better approximation. What if we insist that our approximation $g_1(x)$ matches $f(x) = e^x$ exactly at $x = 0$ and ALSO has exactly the same first derivative as $f(x)$ at $x = 0$.
    1. What is the first derivative of $f(x)$?
    2. What is $f'(0)$?
    3. Use the point-slope form of a line to write the equation of the function $g_1(x)$ that goes through the point $(0, f(0))$ and has slope $f'(0)$. Recall from algebra that the point-slope form of a line is $y = f(x_0) + m(x - x_0)$. In this case we are taking $x_0 = 0$ so we are using the formula $g_1(x) = f(0) + f'(0)(x - 0)$ to get the equation of the line.
3. Write Python code to build a plot like Figure 2.1. This plot shows $f(x) = e^x$, our first approximation $g_0(x) = 1$ and our second approximation $g_1(x) = 1 + x$. You may want to look at Example A.43 in the Python chapter for a refresher on how to build plots containing the graphs of several functions. If you need a hint on how to plot the function $g_0(x)$ since it is a constant function, take a look at the `np.ones_like()` function in Example A.40.

Exercise 2.5. Let us extend the idea from the previous problem to much better approximations of the function $f(x) = e^x$.

1. Let us build a function $g_2(x)$ that matches $f(x)$ exactly at $x = 0$, has exactly the same first derivative as $f(x)$ at $x = 0$, AND has exactly the same second derivative as $f(x)$ at $x = 0$. To do this we will use a quadratic function. For a quadratic approximation of a function we just take a slight extension to the point-slope form of a line and use the equation

$$g_2(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2. \tag{2.1}$$

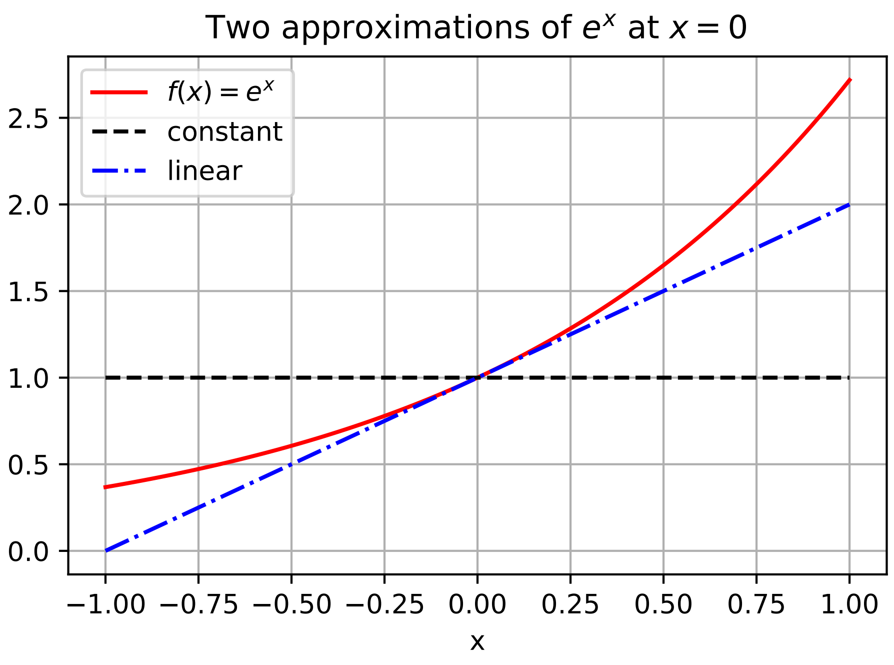


Figure 2.1.: The first two polynomial approximations of the exponential function.

In this case we are using $x_0 = 0$ so the quadratic approximation function looks like

$$g_2(x) = f(0) + f'(0)x + \frac{f''(0)}{2}x^2. \tag{2.2}$$

1. Find the quadratic approximation for $f(x) = e^x$.

2. Add your new function to the plot you created in the previous problem.

2. Let us keep going!! Next we will do a cubic approximation. A cubic approximation takes the form

$$g_3(x) = f(x_0) + f'(0)(x - x_0) + \frac{f''(0)}{2}(x - x_0)^2 + \frac{f'''(0)}{3!}(x - x_0)^3 \tag{2.3}$$

1. Find the cubic approximation for $f(x) = e^x$.

2. How do we know that this function matches the first, second, and third derivatives of $f(x)$ at $x = 0$? What's the deal with the 3! on the cubic term?

3. Add your function to the plot.

Exercise 2.6. Write a function that takes the arguments `x` and `n` and returns the `n`th order Taylor series approximation of $f(x) = e^x$. To remind yourself of how functions are defined in Python, you may want to look at Section A.2.6 in the Python chapter. You will also need a loop, see Section A.2.5. Please start from the following skeleton and put your code where it says "TODO".

```python
def exp_approx(x, n):
    """
    Computes the nth order Taylor series approximation of e^x at x=0.

    Parameters:
    x (float): The value at which to evaluate the approximation.
    n (int): The order of the Taylor series expansion.

    Returns:
    float: The nth order Taylor approximation of e^x.
    """
    if n < 0:
        raise ValueError("n must be at least 0")
    # Start with zero-order approximation
    approximation = 1.0
    # Add higher-order terms
    for i in range(1, n + 1):
        approximation += # TODO: calculate the i'th term of the Taylor series
    return approximation
```

Make sure you perfectly understand the code. Use the AI to explain everything to you in detail. You can select parts of the code and ask the AI questions like "Why do we need to use `range(1, n + 1)`?" and "What does `approximation +=` do?"

Exercise 2.7. Use the function `exp_approx` that you have built in Exercise 2.6 to approximate $\frac{1}{e} = e^{-1}$. Check the accuracy of your answer using `np.exp(-1)` in Python.

Exercise 2.8. Brainstorm within your group to see if you can make your function more efficient by writing it without using exponentiation and the factorial function, coding all multiplications and divisions explicitly. Try to minimise the number of arithmetic operations that you need to perform? Can you make it so that it only needs $n-1$ multiplications, $n-1$ divisions and n additions?

Use the `%timeit` magic command to measure how fast your `exp_approx` function is. In a new code cell, run

```
%timeit exp_approx(1.5, 100)
```

This will run the function multiple times and give you an estimate of the execution time.

2.1.2. Taylor Series

What we have been exploring so far in this section is the **Taylor Series** of a function.

Definition 2.1 (Taylor Series)**.** If $f(x)$ is an infinitely differentiable function at the point x_0 then

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + \cdots \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + \cdots \tag{2.4}$$

for any reasonably small interval around x_0. The infinite polynomial expansion is called the **Taylor Series** of the function $f(x)$. Taylor Series are named for the mathematician Brook Taylor.

The Taylor Series of a function is often written with summation notation as

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k. \tag{2.5}$$

Do not let the notation scare you. In a Taylor Series you are just saying: *give me a function that*

- *matches* $f(x)$ at $x = x_0$ exactly,
- *matches* $f'(x)$ at $x = x_0$ exactly,
- *matches* $f''(x)$ at $x = x_0$ exactly,
- *matches* $f'''(x)$ at $x = x_0$ exactly,
- etc.

(Take a moment and make sure that the summation notation makes sense to you.)

Moreover, Taylor Series are built out of the easiest types of functions: polynomials. Computers are rather good at doing computations with addition, subtraction, multiplication, division, and integer exponents, so Taylor Series are a natural way to express functions in a computer. The down side is that we can only get true equality in the Taylor Series if we have infinitely many terms in the series. A computer cannot do infinitely many computations. So, in practice, we **truncate** Taylor Series after many terms and think of the new polynomial function as being *close enough* to the actual function so far as we do not stray too far from the anchor x_0.

Exercise 2.9. Do all of the calculations to show that the Taylor Series centred at $x_0 = 0$ for the function $f(x) = \sin(x)$ is indeed

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots. \tag{2.6}$$

Exercise 2.10. Write a Pyton function `sin_approx(x, n)` that computes the nth order Taylor Series approximation of $\sin(x)$ centred at $x_0 = 0$. Test your function by comparing its output to `np.sin(x)` for a few values of x and n. Use it to make a plot of the first three approximations for x in the range $[-\pi, \pi]$ similar to the plots you made for approximations of e^x above.

Exercise 2.11. Let us compute a Taylor Series that is not centred at $x_0 = 0$. For example, let us approximate the function $f(x) = \log(x)$ near $x_0 = 1$. Near the point $x_0 = 1$, the Taylor Series approximation will take the form

$$f(x) = f(1) + f'(1)(x-1) + \frac{f''(1)}{2!}(x-1)^2 + \frac{f'''(1)}{3!}(x-1)^3 + \cdots \tag{2.7}$$

Write the first several terms of the Taylor Series for $f(x) = \log x$ centred at $x_0 = 1$ until you get a feel for the pattern.

Exercise 2.12. Write a Pyton function `log_approx(x, n)` that computes the nth order Taylor Series approximation of $\log(x)$ centred at $x_0 = 1$. Use it to build the plot below showing the approximations.

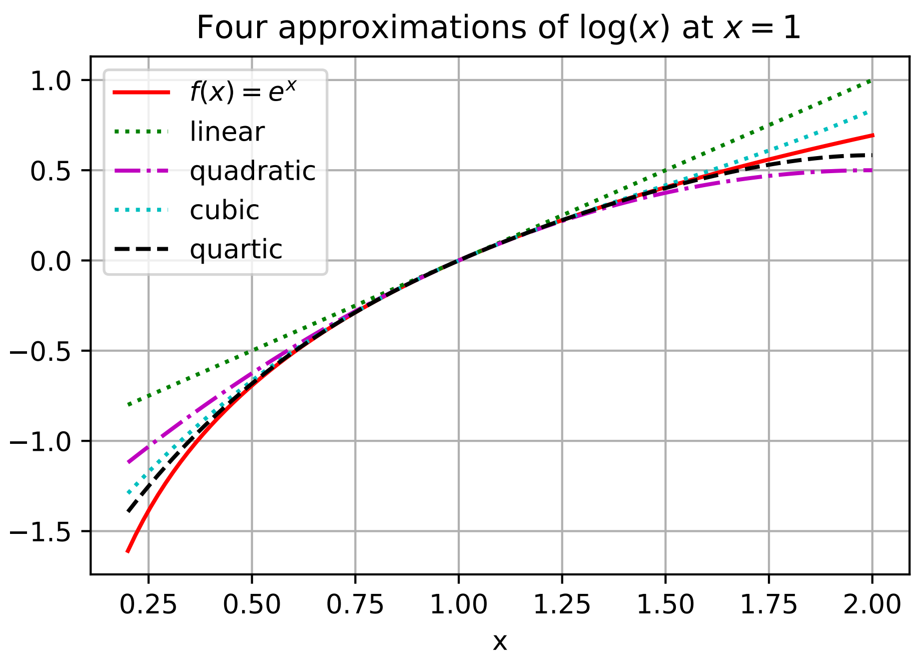


Figure 2.2.: Taylor series approximation of the logarithm.

Example 2.1. Let us conclude this brief section by examining an interesting example. Consider the function

$$f(x) = \frac{1}{1-x}. \tag{2.8}$$

If we build a Taylor Series centred at $x_0 = 0$ it is not too hard to show that we get

$$f(x) = 1 + x + x^2 + x^3 + x^4 + x^5 + \cdots \tag{2.9}$$

(you should stop now and verify this!). However, if we plot the function $f(x)$ along with several successive approximations for $f(x)$ we find that beyond $x = 1$ we do not get the correct behaviour of the function (see Figure 2.3). More specifically, we cannot get the Taylor Series to change behaviour across the vertical asymptote of the function at $x = 1$. This example is meant to point out the fact that a Taylor Series will only ever make sense *near* the point at which you centre the expansion. For the function $f(x) = \frac{1}{1-x}$ centred at $x_0 = 0$ we can only get good approximations within the interval $x \in (-1, 1)$ and no further.

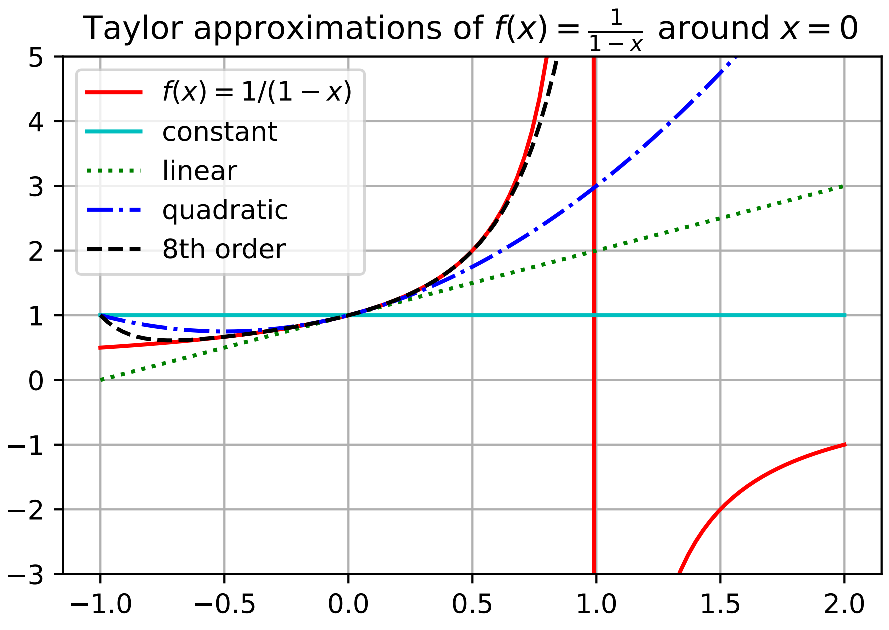


Figure 2.3.: Several Taylor Series approximations of the function $f(x) = 1/(1 - x)$.

In the previous example we saw that we cannot always get approximations from Taylor Series that are good everywhere. For every Taylor Series there is a **domain of convergence** where the Taylor Series actually makes sense and gives good approximations. It is beyond the scope of this section to give all of the details for finding the domain of convergence for a Taylor Series. You have done that in your first-year Calculus module. However a good heuristic is to observe that a Taylor Series will only give reasonable approximations of a function from the centre of the series to the nearest asymptote. The domain of convergence is typically symmetric about the centre as well. For example:

- If we were to build a Taylor Series approximation for the function $f(x) = \log(x)$ centred at the point $x_0 = 1$ then the domain of convergence should be $x \in (0, 2)$ since there is a vertical asymptote for the natural logarithm function at $x = 0$.
- If we were to build a Taylor Series approximation for the function $f(x) = \frac{5}{2x-3}$ centred at the point $x_0 = 4$ then the domain of convergence should be $x \in (1.5, 6.5)$ since there is a vertical asymptote at $x = 1.5$ and the distance from $x_0 = 4$ to $x = 1.5$ is 2.5 units.
- If we were to build a Taylor Series approximation for the function $f(x) = \frac{1}{1+x^2}$ centred at the point $x_0 = 0$ then the domain of convergence should be $x \in (-1, 1)$. This may seem quite odd (and perhaps quite surprising!) but let us think about where the nearest asymptote might be. To find the asymptote we need to solve $1 + x^2 = 0$ but this gives us the values $x = \pm i$. In the complex plane, the numbers i and $-i$ are 1 unit away from $x_0 = 0$, so the "asymptote" is not visible in a

real-valued plot but it is still only one unit away. Hence the domain of convergence is $x \in (-1, 1)$. You may want to pause now and build some plots to show yourself that this indeed appears to be true.

Of course you learned all this and more in your first-year Calculus but I hope it was fun to now rediscover these things yourself. In your Calculus module it was probably not stressed how fundamental Taylor series are to doing numerical computations.

2.2. Truncation Error

The great thing about Taylor Series is that they allow for the representation of potentially very complicated functions as polynomials – and polynomials are easily dealt with on a computer since they involve only addition, subtraction, multiplication, division, and integer powers. The down side is that the order of the polynomial is infinite. Hence, every time we use a Taylor series on a computer, what we are actually going to be using is a **Truncated Taylor Series** where we only take a finite number of terms. The idea here is simple in principle:

- If a function $f(x)$ has a Taylor Series representation it can be written as an infinite sum.
- Computers cannot do infinite sums.
- So stop the sum at some point n and throw away the rest of the infinite sum.
- Now $f(x)$ is approximated by some finite sum so long as you stay pretty close to $x = x_0$,
- and everything that we just chopped off of the end is called the **remainder** for the finite sum.

Let us be a bit more concrete about it. The Taylor Series for $f(x) = e^x$ centred at $x_0 = 0$ is

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots. \tag{2.10}$$

When we truncate this series at order n, we separate it into an approximation and a remainder:

$$e^x = \underbrace{1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}}_{n^{th}\text{ order approximation}} + \underbrace{\frac{x^{n+1}}{(n+1)!} + \frac{x^{n+2}}{(n+2)!} + \cdots}_{\text{remainder}}. \tag{2.11}$$

For small values of x near $x_0 = 0$, the largest term in the remainder is $\frac{x^{n+1}}{(n+1)!}$, since higher powers of x become progressively smaller. We use **Big-O notation** to express this:

$$e^x \approx 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \mathcal{O}(x^{n+1}), \tag{2.12}$$

where the notation $\mathcal{O}(x^{n+1})$ (read "Big-O of x^{n+1}") signifies that the error is bounded by $C|x|^{n+1}$ for some constant C as $x \to 0$. This indicates that the error scales like x^{n+1} for values of x near the center $x_0 = 0$.

For example:

- 0^{th} order: $e^x \approx 1 + \mathcal{O}(x)$
- 1^{st} order: $e^x \approx 1 + x + \mathcal{O}(x^2)$
- 2^{nd} order: $e^x \approx 1 + x + \frac{x^2}{2} + \mathcal{O}(x^3)$

Keep in mind that this sort of analysis is only good for values of x that are very close to the centre of the Taylor Series. If you are making approximations that are too far away then all bets are off.

Exercise 2.13. Now make the previous discussion a bit more concrete. You know the Taylor Series for $f(x) = e^x$ around $x = 0$ quite well at this point so use it to approximate the values of $f(0.1) = e^{0.1}$ and $f(0.2) = e^{0.2}$ by truncating the Taylor series at different orders. Because $x = 0.1$ and $x = 0.2$ are pretty close to the centre of the Taylor Series $x_0 = 0$, this sort of approximation is reasonable.

Then compare your approximate values to Python's values $f(0.1) = e^{0.1} \approx$ `np.exp(0.1)` $= 1.1051709180756477$ and $f(0.2) = e^{0.2} \approx$ `np.exp(0.2)` $= 1.2214027581601699$ to calculate the truncation errors $\epsilon_n(0.1) = |f(0.1) - f_n(0.1)|$ and $\epsilon_n(0.2) = |f(0.2) - f_n(0.2)|$.

Fill in the blanks in the table. If you like, you can copy and paste the code and extend it to fill in the missing rows. For a bit of explanation of the syntax of the print commands see Example A.20 but for more detailed information ask Gemini.

```
Order n   | f_n(0.1)         | _n(0.1)          | f_n(0.2)         | _n(0.2)
------------------------------------------------------------------------------
0         | 1                | 0.1051709181     | 1                | 0.2214027582
1         | 1.1              | 0.005170918076   | 1.2              | 0.02140275816
2         |                  |                  |                  |
3         |                  |                  |                  |
4         |                  |                  |                  |
5         |                  |                  |                  |
```

You will find that, as expected, the truncation errors $\epsilon_n(x)$ decrease with n but increase with x.

Exercise 2.14. To investigate the dependence of the truncation error $\epsilon_n(x)$ on n and x a bit more, add an extra column to the table from the previous exercise with the ratio $\epsilon_n(0.2)/\epsilon_n(0.1)$.

```
Order n  | _n(0.1)         | _n(0.2)         |  _n(0.2) / _n(0.1)
--------------------------------------------------------------------------------
0        | 0.1051709181    | 0.2214027582    | 2.105170918
1        | 0.005170918076  | 0.02140275816   | 4.139063479
2        |                 |                 |
3        |                 |                 |
4        |                 |                 |
5        |                 |                 |
```

Formulate a conjecture about how ϵ_n changes as x changes.

Exercise 2.15. To test your conjecture, examine the truncation error for the sine function near $x_0 = 0$. You know that the sine function has the Taylor Series centred at $x_0 = 0$ as

$$f(x) = \sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots. \tag{2.13}$$

So there are only approximations of odd order. Use the truncated Taylor series to approximate $f(0.1) = \sin(0.1)$ and $f(0.2) = \sin(0.2)$ and use Python's values `np.sin(0.1)` and `np.sin(0.2)` to calculate the truncation errors $\epsilon_n(0.1) = |f(0.1) - f_n(0.1)|$ and $\epsilon_n(0.2) = |f(0.2) - f_n(0.2)|$.

Complete the following table:

```
Order n  | _n(0.1)         | _n(0.2)         |  _n(0.2) / _n(0.1)
--------------------------------------------------------------------------------
1        | 0.0001665833532 | 0.001330669205  | 7.988008283
3        |                 |                 |
5        |                 |                 |
7        |                 |                 |
9        |                 |                 |
```

To learn how you can loop over only odd integers in Python, see Example A.7.

Did these results force you to revise your conjecture of how ϵ_n changes as x changes?

The entry in the last row of the table will almost certainly not agree with your conjecture. That is okay! That discrepancy has a different explanation. Can you figure out what it is? Hint: Think about the discussion of machine precision in Chapter 1.

Exercise 2.16. Perform another check of your conjecture by approximating $\log(1.02)$ and $\log(1.1)$ from truncations of the Taylor series around $x = 1$:

$$\log(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots.$$

Exercise 2.17. Write down your groups's observations about how the truncation error changes as x changes. Explain this in terms of the form of the remainder of the truncated Taylor series.

2.3. Exam-style question

You know the Taylor Series expansion for $f(x) = \sin(x)$ centred at $x_0 = 0$:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots.$$

(a) Explain why there is no x^2 term in this Taylor Series expansion. [2 marks]

(b) Complete the Python code to calculate the Taylor expansion for $f(x) = \sin(x)$ centred at $x_0 = 0$ up to order n. [4 marks]

```python
def sin_taylor(x, n):
    """
    Calculate the Taylor expansion for f(x) = sin(x) centred at x_0 = 0 up to order n.
    """
    ....
    for i in range(...):
        ....

    return result
```

(c) i) If we only keep terms up to x^3 in this Taylor Series expansion, what is the truncation error in Big-O notation for the approximation? [2 marks]

ii) If we use the same approximation to calculate $\sin(0.05)$ instead of $\sin(0.1)$, by what factor do we expect the truncation error to decrease? [2 marks]

2.4. Neural Networks

So far we have discussed approximating functions with polynomials, and in particular with Taylor Series. However there are other families of functions that can be used to approximate an arbitrary function. One of them you have already met in your first year: Fourier series. Another one that is usually discussed in a course on Numerical Analysis is splines. In this module we will not discuss these families of functions. Our aim in this module is not to be exhaustive, but to get the fundamental ideas across, so that you will be well equipped to acquire further knowledge on the topic later on.

There is however a family of functions that has recently become very popular in machine learning: **neural networks**. This section guides you through exercises to obtain a good intuitive understanding of neural networks.

It may well be that by the time you reach this point in the learning guide, you will not have much time left this week. For that reason the material on neural networks is not examinable. You are under no pressure to work through this material. However, if you do have the time and interest, I would encourage you to do so.

Similar to how a polynomial $p(x)$ is determined by giving the coefficients in front of the powers of x, a neural network is determined by giving a set of parameters, called weights and biases. In this section you will explore how the weights and biases determine the neural network function.

2.4.1. A Single Neuron

Let us start by building up the components of a neural network one piece at a time. The fundamental building block is called a **neuron**.

Exercise 2.18. Consider the simple function

$$f(x) = \max(0, x). \tag{2.14}$$

The function $g(x) = \max(0, x)$ is called the **Rectified Linear Unit** or **ReLU** for short.

1. By hand, sketch the graph of this function for $x \in [-3, 3]$.
2. What is the derivative of $f(x)$ for $x > 0$? What about for $x < 0$? What happens at $x = 0$?

The ReLU function is an example of what is called an **activation function** in neural networks. The idea is that the neuron "activates" (produces a non-zero output) only when the input exceeds a certain threshold.

Exercise 2.19. Now let us make our neuron a bit more interesting by allowing it to have adjustable parameters.

Consider the function

$$h(x) = \max(0, wx + b) \tag{2.15}$$

where w and b are parameters that we can choose. These are called the **weight** and **bias** respectively.

1. By hand, sketch the graph of $h(x)$ for $x \in [-3, 3]$ for each of the following parameter values:

   i. $w = 1, b = 0$
   ii. $w = 2, b = 0$
   iii. $w = 1, b = 1$
   iv. $w = 1, b = -1$
   v. $w = -1, b = 0$

2. Describe in words what the weight w controls about the function $h(x)$.
3. Describe in words what the bias b controls about the function $h(x)$.
4. For which values of x is the function "active" (i.e., non-zero) when $w = 1$ and $b = -1$?

Exercise 2.20. So far we have been working with a neuron that takes a single input x. In many applications, we want to work with functions of multiple variables.

Suppose we have two inputs x_1 and x_2, and we define a neuron as

$$h(x_1, x_2) = \max(0, w_1x_1 + w_2x_2 + b) \tag{2.16}$$

where w_1, w_2 are weights and b is a bias.

1. What is the output of this neuron when $x_1 = 1, x_2 = 2$, using the parameters $w_1 = 1, w_2 = -1, b = 0$?
2. The expression $w_1x_1 + w_2x_2 + b = 0$ defines a line in the (x_1, x_2) plane. For the parameters in part 1, sketch this line in the region $x_1 \in [-2, 2], x_2 \in [-2, 2]$.
3. On which side of this line is the neuron "active" (produces non-zero outputs)? Shade that area in your sketch from part 2.

2.4.2. Combining Neurons into a Layer

A single neuron can detect when the input crosses a particular threshold. But to approximate more complex functions, we need to combine multiple neurons together. Let us explore this idea.

Exercise 2.21. Suppose we have two neurons, both taking the same input x, but with different weights and biases:

$$h_1(x) = \max(0, w_1 x + b_1) \tag{2.17}$$

$$h_2(x) = \max(0, w_2 x + b_2) \tag{2.18}$$

Consider the specific case where $w_1 = 1, b_1 = -1$ and $w_2 = -1, b_2 = 1$.

1. For what values of x is $h_1(x)$ active (non-zero)?
2. For what values of x is $h_2(x)$ active (non-zero)?
3. By hand, sketch the graphs of both $h_1(x)$ and $h_2(x)$ on the same axes for $x \in [-2, 2]$.
4. Now consider the sum $h_1(x) + h_2(x)$. By hand, sketch the graph of this function. Describe its shape.
5. What is the minimum value of $h_1(x) + h_2(x)$? At what value of x does this minimum occur?

Exercise 2.22. Rather than just adding two ReLU neurons, consider the weighted combination:

$$f(x) = c_1 h_1(x) + c_2 h_2(x) \tag{2.19}$$

where c_1 and c_2 are **output weights** that can be positive or negative.

Let $h_1(x) = \max(0, x)$ and $h_2(x) = \max(0, x - 1)$.

- What is $f(x) = h_1(x) - 2h_2(x)$ for $x < 0$?
- What is $f(x) = h_1(x) - 2h_2(x)$ for $0 \leq x < 1$?
- What is $f(x) = h_1(x) - 2h_2(x)$ for $x \geq 1$?

By hand, sketch the graph of this function.

Exercise 2.23. In the previous exercise you created a function with a triangular bump but the function continued decreasing to negative values beyond $x = 2$. Now see if you can find a way to combine three ReLU neurons to create a triangular bump function that:

- is zero for $x < 0$ and $x > 2$
- equals x for $0 \leq x \leq 1$
- equals $2 - x$ for $1 < x \leq 2$

1. Write down the weights $w_1, b_1, w_2, b_2, w_3, b_3$ and output weights c_1, c_2 and c_3 that produce the desired function. Hint: You need h_3 to "cancel out" the negative values after $x = 2$.
2. Sketch the graph of your function to verify that it works.
3. How could you modify the output weights to make the bump twice as tall?
4. How could you modify the weights and biases to shift the bump so that it is centred at $x = 5$ instead of $x = 1$?

2.4.3. A Two-Layer Neural Network

In the previous exercises, you discovered that by combining multiple neurons (each with a ReLU activation), we can build flexible function approximations. Let us now formalise this into what is called a **two-layer neural network** or **single hidden layer network**.

The structure is as follows:

1. **Input**: We start with an input value x.
2. **Hidden Layer**: We apply n neurons to the input, creating n hidden values:

   $$h_i(x) = \max(0, w_i x + b_i) \quad \text{for } i = 1, 2, \dots, n \tag{2.20}$$

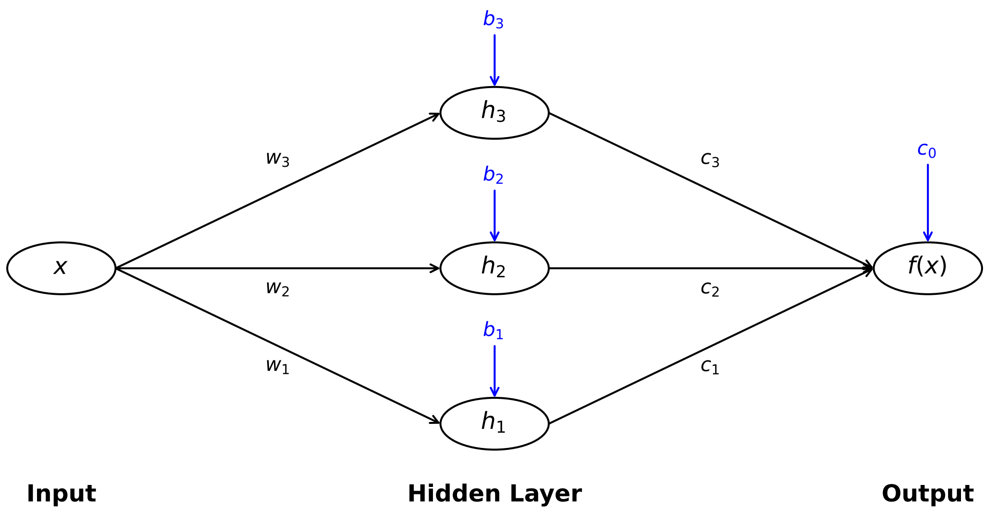
   The parameters w_i and b_i are called the **weights** and **biases** of the hidden layer.
3. **Output Layer**: We combine the hidden values using output weights:

   $$f(x) = c_1 h_1(x) + c_2 h_2(x) + \dots + c_n h_n(x) + c_0 \tag{2.21}$$

   where $c_0, c_1, \dots, c_n$ are the output layer parameters.

We can write this more compactly using summation notation:

$$f(x) = \sum_{i=1}^{n} c_i \max(0, w_i x + b_i) + c_0. \tag{2.22}$$



This function $f(x)$ is determined by the parameters:

- Hidden layer weights: $w_1, w_2, \dots, w_n$
- Hidden layer biases: $b_1, b_2, \dots, b_n$

- Output weights: $c_1, c_2, \dots, c_n$
- Output bias: c_0

That gives us a total of $3n + 1$ parameters to work with.

Note that, due to the simple linear nature of the ReLU activation function that we have chosen to use here, we can absorb the weights of the hidden layer neurons into a rescaling of the of the output weights and the biases. This would not be true for more general activation functions.

So we choose to set all hidden weights to 1 and thus work with

$$f_{nn}(x) = \sum_{i=1}^{n} c_i \max(0, x + b_i) + c_0 \tag{2.23}$$

The following Python code implements such a neural network. Make sure you understand the code.

Exercise 2.24. Explain in your own words why a two-layer neural network is flexible enough to approximate many different functions. How do the output weights control the shape of the approximation?

How is this similar to polynomial approximation? How is it different?

Exercise 2.25. In this exercise we will approximate a smooth function with a neural network using a simple but powerful idea: we can create a piecewise linear approximation by using a sum of ReLU neurons, each of which activates at a new point. Each ReLU neuron that activates at a new point changes the slope of our piecewise linear approximation.

Consider approximating $f(x) = \sin(x)$ on the interval $[0, \pi]$ using a two-layer neural network. We'll place ReLU neurons at several "knot points" along the curve, and each neuron will adjust the slope.

1. Understanding slope changes: Suppose we use knot points at $x_1 = -b_1 = 0, x_2 = -b_2 = \pi/2$.

   - For $x \in [0, \pi/2)$: What is the slope of $f_{nn}(x)$? (Hint: which neurons are active?)
   - For $x \in [\pi/2, \pi)$: What is the slope?

   Explain how the output weight c_2 controls the change in slope at $x = \pi/2$.

   - Suggest a choice for the output weights c_0, c_1, c_2 that you think will give some kind of crude approximation of $\sin(x)$ on the interval $[0, \pi]$. Use the Python function `plot_neural_network` provided below to make a plot of your approximation.

2. Use the Python function `plot_neural_network` with 4 knot points at $x = 0, \pi/4, \pi/2, 3\pi/4$. Try to find output weights that create a good approximation.
3. Calculate the maximum approximation error and discuss how it could be improved.
4. Experiment with more knot points (try 9 or 17 points evenly spaced). How does the approximation improve?

Python helper function:

After your experimentation, the following result will no longer seem so surprising:

Theorem 2.1 (Universal Approximation Theorem (informal version))**.** *A two-layer neural network with a sufficient number of neurons can approximate any continuous function on a bounded interval to arbitrary accuracy.*

More precisely: for any continuous function $f : [a, b] \to \mathbb{R}$ and any $\epsilon > 0$, there exists a two-layer neural network f_{nn} such that

$$\max_{x\in[a,b]} |f(x) - f_{nn}(x)| < \epsilon. \tag{2.24}$$

Exercise 2.26. Reflect on what you have learned about neural networks and compare them to Taylor series:

1. What are the advantages of using neural networks for function approximation compared to Taylor series?
2. What are the advantages of Taylor series compared to neural networks?
3. For each of the following functions, which approximation method (Taylor series or neural network) do you think would be more appropriate, and why?
   - $f(x) = e^x$ near $x = 0$
   - $f(x) = |x|$ near $x = 0$

   - $f(x) = \sin(100x)$ on $[0, 2\pi]$
   - A function defined by experimental data points
4. Neural networks are determined by their weights and biases. Taylor series are determined by the derivatives of the function at a point. Which do you think is easier to compute in practice, and why?

2.4.4. Deep Neural Networks

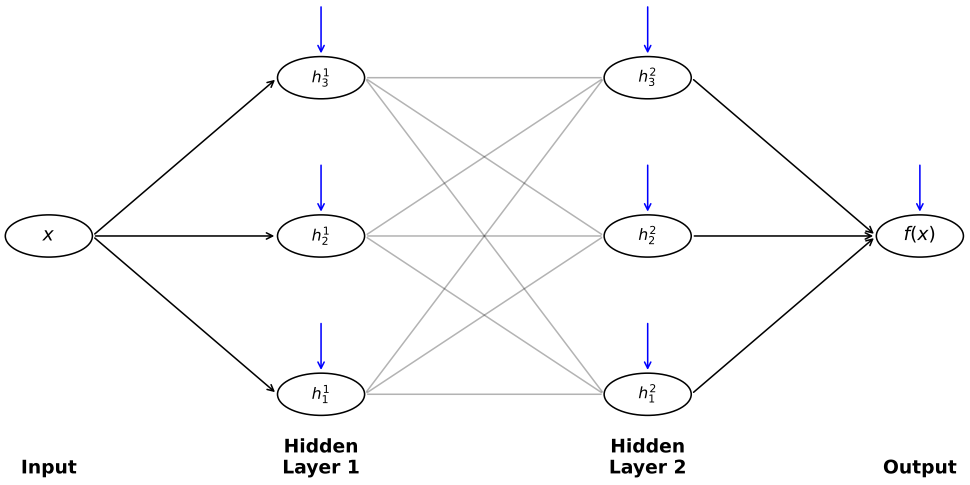
So far we have looked at **two-layer** neural networks (one hidden layer + one output layer). But we can also stack multiple layers on top of each other to create **deep neural networks**.

The idea is simple: instead of having the output layer directly combine the hidden neurons, we can feed the hidden neurons into another layer of neurons, and then another, and so on.

For example, a three-layer network would look like:

1. **Input**: x
2. **First Hidden Layer**:
$$h_i^{(1)}(x) = \max(0, w_i^{(1)}x + b_i^{(1)}) \quad \text{for } i = 1, \ldots, n_1 \tag{2.25}$$
3. **Second Hidden Layer**: Each neuron in this layer takes all outputs from the first hidden layer:
$$h_j^{(2)} = \max\left(0, \sum_{i=1}^{n_1} w_{ji}^{(2)} h_i^{(1)} + b_j^{(2)}\right) \quad \text{for } j = 1, \ldots, n_2 \tag{2.26}$$
4. **Output Layer**:
$$f(x) = \sum_{j=1}^{n_2} c_j h_j^{(2)} + c_0 \tag{2.27}$$

This is called a **deep neural network** because it has multiple hidden layers. The term "deep learning" comes from using such multi-layer architectures.

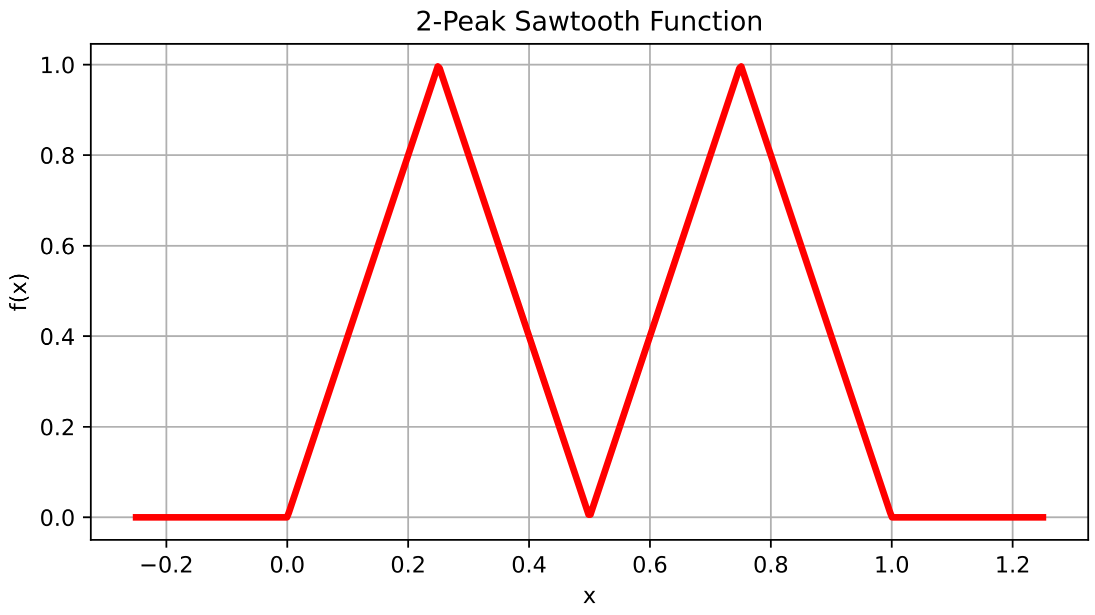


2.4.5. Why Depth Matters: The Sawtooth Example

Why are deep neural networks, with many layers, often more powerful than shallow networks with just one large hidden layer? In the following exercises, you will explore a classic example, the **sawtooth wave**, to discover a concrete answer. You'll build both a shallow and a deep network to represent the same function, and compare their efficiency as the complexity grows. Through this exploration, you'll see how deep networks can represent certain functions far more efficiently by exploiting their compositional structure.

We'll work with a sawtooth function that has 2 triangular peaks on the interval $[0, 1]$:

$$f(x) = \begin{cases} 4x & \text{if } 0 \leq x < 0.25 \\ 2 - 4x & \text{if } 0.25 \leq x < 0.5 \\ 4x - 2 & \text{if } 0.5 \leq x < 0.75 \\ 4 - 4x & \text{if } 0.75 \leq x \leq 1 \end{cases} \tag{2.28}$$



Exercise 2.27 (Building a Shallow Network for the Sawtooth)**.** You already know how to represent piecewise linear functions with a two-layer neural network. Each of the 5 "hinge points" (where the slope changes) requires a ReLU neuron.

a) For each hinge point at position x_i, you need a ReLU neuron with bias $b_i = -x_i$.

b) What are the output weights with which you need to combinae these neurons?

Use this to complete the code below and use the plot to check that the resulting function has the desired form.

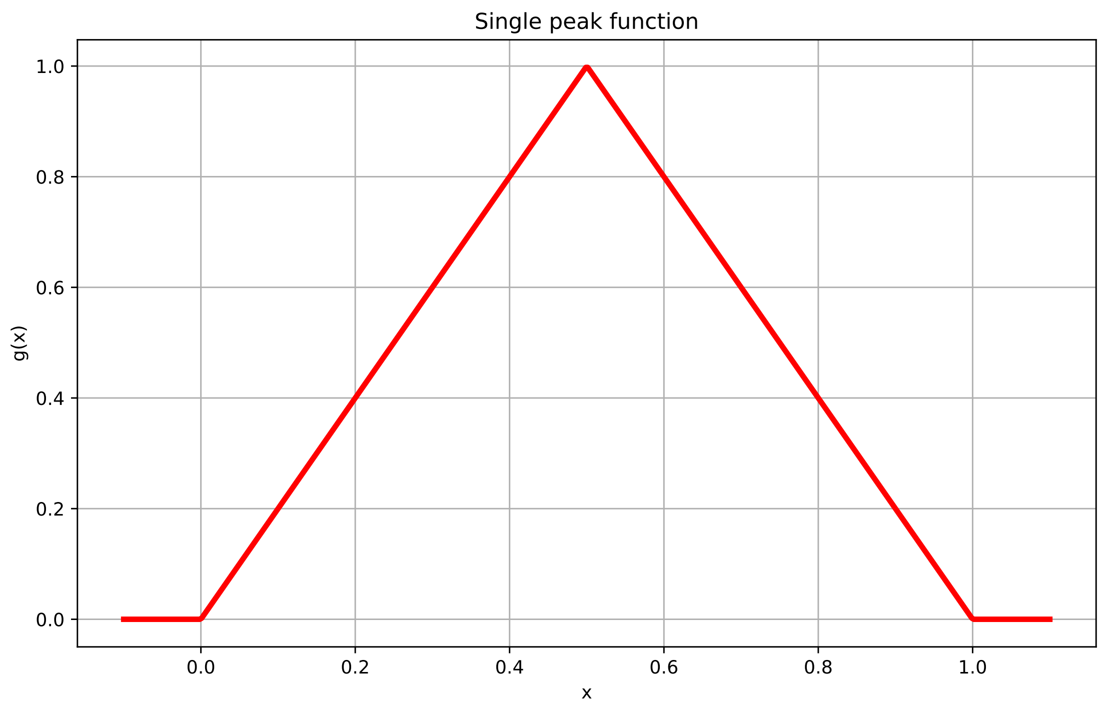
```python
def shallow_2_peak_network(x):
    biases = [.....]
    output_weights = [.....]
    return two_layer_network(x, biases, output_weights)

# Plot
plt.figure(figsize=(10, 6))
x = np.linspace(-0.1,1.1,500)
plt.plot(x, shallow_2_peak_network(x), 'r-', lw=3)
plt.grid()
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('2-Peak Sawtooth Function')
plt.show()
```

Exercise 2.28 (Building the Deep Network)**.** A deeper network takes a more elegant approach. Instead of directly creating all the peaks, it uses one layer to create a single peak and then another layer to duplicate this peak.

The key insight is to create a single-peak function

$$g(x) = \begin{cases} 2x & 0 \le x < 1/2 \\ 2(1-x) & 1/2 \le x \le 1 \\ 0 & x \notin [0,1] \end{cases}$$

that produces one triangular peak at $x = 0.5$ with height 1, returning to 0 at $z < 0$ and $z > 1$.

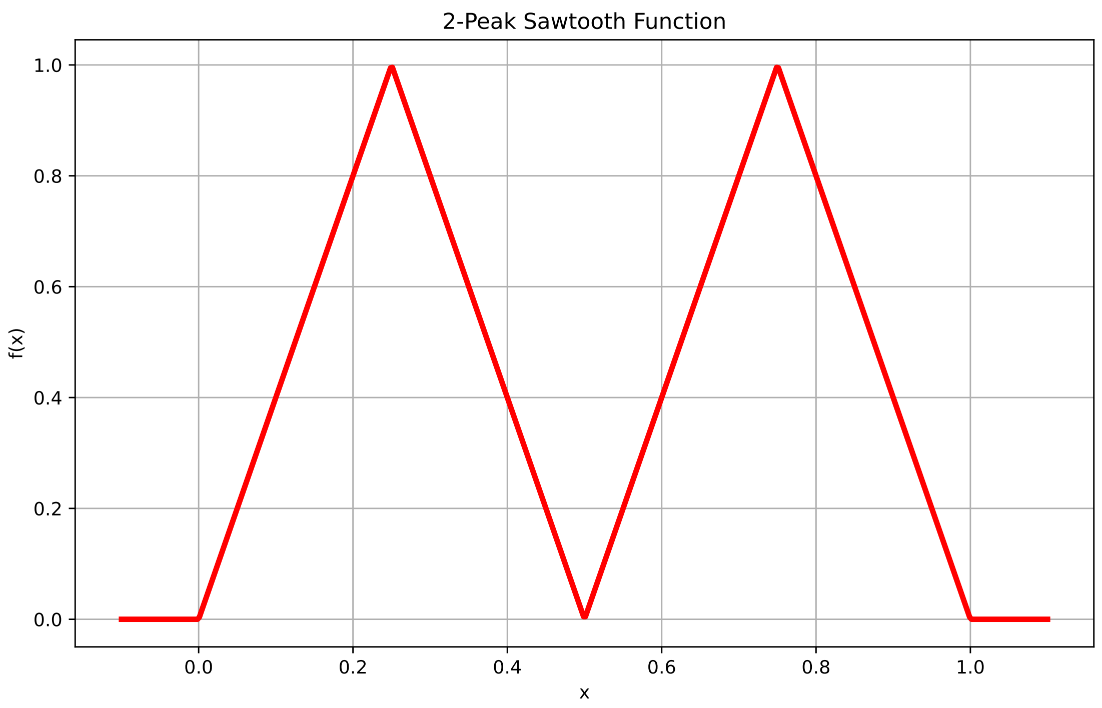


Then, our 2-peak sawtooth is simply $f(x) = g(g(x))$.

a) First, create a single-peak as a shallow network with three hidden neurons.

```python
def g(x):
    """ Single triangular peak """
    biases = [.....]
    output_weights = [.....]
    return two_layer_network(x, biases, output_weights)

# Plot
plt.figure(figsize=(10, 6))
x = np.linspace(-0.1,1.1,500)
plt.plot(x, g(x), 'r-', lw=3))
plt.grid()
plt.xlabel('x')
plt.ylabel('g(x)')
plt.title('Single peak function')
plt.show()
```

b) Now comes the magic! Pass the result of your network $g(x)$ through the same network again: compute $g(g(x))$. Plot the result and compare it to the target.



c) Explain in your own words: Why does applying g twice produce two peaks? Think about what happens as the first application of g creates a peak that rises from 0 to 1 and back to 0. What does the second application of g see as its input?

d) Create a 4-peak sawtooth by composing g three times: $g(g(g(x)))$. Plot it. How many hidden neurons does this use? Compare to the shallow network approach from the previous exercise.

Exercise 2.29 (Scaling Analysis: The Exponential Advantage)**.** Now let's compare how the two approaches scale as we increase the number of peaks.

a) Complete this table by implementing both shallow and deep networks for different numbers of peaks:

Number of Peaks	Shallow Network Neurons	Deep Network Neurons	Advantage (ratio)
2	5	6	0.83
4	?	?	?
8	?	?	?
16	?	?	?

b) Based on your table, write formulas for the number of neurons needed as a function of the number of peaks n for both approaches. What kind of growth do you observe (linear, logarithmic, exponential)?

Hint: For the deep network, if $n = 2^k$ peaks, how many layers do you need? How many neurons per layer?

c) Visualize the scaling behavior. Plot the number of neurons (y-axis) versus the number of peaks (x-axis) for both approaches:

```python
# TODO: Create arrays for different numbers of peaks
peaks = [2, 4, 8, 16, 32, 64]
shallow_neurons = [...]  # Fill in based on your formula
deep_neurons = [...]     # Fill in based on your formula

# Plot
plt.figure(figsize=(8, 5))
plt.plot(peaks, shallow_neurons, 'bo-', label='Shallow Network', lw=2)
plt.plot(peaks, deep_neurons, 'mo-', label='Deep Network', lw=2)
plt.xlabel('Number of Peaks')
plt.ylabel('Number of Hidden Neurons')
plt.title('Scaling: Shallow vs Deep Networks')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

d) Based on your analysis, explain why deep networks can be more efficient than shallow networks for certain types of functions. What specific property of the sawtooth function makes it particularly well-suited for a deep architecture?

e) The deep network achieves its efficiency by **reusing** the same learned "peak-making" module across layers through composition. Can you think of other real-world functions or patterns that might benefit from deep architectures in a similar way? Consider:

- Functions with repetitive structures at different scales (e.g., fractals, wavelets)
- Hierarchical patterns (e.g., language: letters $\rightarrow$ words $\rightarrow$ sentences)
- Functions that can be naturally expressed as compositions of simpler functions

f) What are the limitations of this advantage? Can you think of functions where a shallow network might be more efficient than a deep one? (Hint: What if the function has no repetitive or compositional structure?)

2.5. Problems

Exercise 2.30. Find the Taylor Series for $f(x) = \frac{1}{\log(x)}$ centred at the point $x_0 = e$. Then use the Taylor Series to approximate the number $\frac{1}{\log(3)}$ to 4 decimal places.

Exercise 2.31. In this problem we will use Taylor Series to build approximations for the irrational number π.

a. Write the Taylor series centred at $x_0 = 0$ for the function

$$f(x) = \frac{1}{1+x}. \tag{2.29}$$

b. Now we want to get the Taylor Series for the function $g(x) = \frac{1}{1+x^2}$. It would be quite time consuming to take all of the necessary derivatives to get this Taylor Series. Instead we will use our answer from part (a) of this problem to shortcut the whole process.

   1. Substitute x^2 for every x in the Taylor Series for $f(x) = \frac{1}{1+x}$.
   2. Make a few plots to verify that we indeed now have a Taylor Series for the function $g(x) = \frac{1}{1+x^2}$.

c. Recall from Calculus that

$$\int \frac{1}{1+x^2} dx = \arctan(x). \tag{2.30}$$

   Hence, if we integrate each term of the Taylor Series that results from part (2) we should have a Taylor Series for $\arctan(x)$.[1]

d. Now recall the following from Calculus:

   - $\tan(\pi/4) = 1$
   - so $\arctan(1) = \pi/4$
   - and therefore $\pi = 4\arctan(1)$.

[1]There are many reasons why integrating an infinite series term by term should give you a moment of pause. For the sake of this problem we are doing this operation a little blindly, but in reality we should have verified that the infinite series actually converges uniformly.

Let us use these facts along with the Taylor Series for $\arctan(x)$ to approximate π: we can just plug in $x = 1$ to the series, add up a bunch of terms, and then multiply by 4. Write a loop in Python that builds successively better and better approximations of π. Stop the loop when you have an approximation that is correct to 6 decimal places.

Exercise 2.32. In this problem we will prove the famous formula

$$e^{i\theta} = \cos(\theta) + i\sin(\theta). \tag{2.31}$$

This is known as Euler's formula after the famous mathematician Leonard Euler. Show all of your work for the following tasks.

1. Write the Taylor series for the functions e^x, $\sin(x)$, and $\cos(x)$.
2. Replace x with $i\theta$ in the Taylor expansion of e^x. Recall that $i = \sqrt{-1}$ so $i^2 = -1$, $i^3 = -i$, and $i^4 = 1$. Simplify all of the powers of $i\theta$ that arise in the Taylor expansion. I will get you started:

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots \\ e^{i\theta} &= 1 + (i\theta) + \frac{(i\theta)^2}{2!} + \frac{(i\theta)^3}{3!} + \frac{(i\theta)^4}{4!} + \frac{(i\theta)^5}{5!} + \cdots \\ &= 1 + i\theta + i^2\frac{\theta^2}{2!} + i^3\frac{\theta^3}{3!} + i^4\frac{\theta^4}{4!} + i^5\frac{\theta^5}{5!} + \cdots \\ &= \text{... keep simplifying ... ...} \end{aligned} \tag{2.32}$$

3. Gather all of the real terms and all of the imaginary terms together. Factor the i out of the imaginary terms. What do you notice?
4. Use your result from part (3) to prove that $e^{i\pi} + 1 = 0$.

Exercise 2.33. In physics, the *relativistic energy* of an object is defined as

$$E_{rel} = \gamma mc^2 \tag{2.33}$$

where

$$\gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}. \tag{2.34}$$

In these equations, m is the mass of the object, c is the speed of light ($c \approx 3 \times 10^8$m/s), and v is the velocity of the object. For an object of fixed mass (m) we can expand the Taylor Series centred at $v = 0$ for E_{rel} to get

$$E_{rel} = mc^2 + \frac{1}{2}mv^2 + \frac{3}{8}\frac{mv^4}{c^2} + \frac{5}{16}\frac{mv^6}{c^4} + \cdots . \tag{2.35}$$

1. What do we recover if we consider an object with zero velocity?
2. Why might it be completely reasonable to only use the quadratic approximation

$$E_{rel} = mc^2 + \frac{1}{2}mv^2 \tag{2.36}$$

   for the relativistic energy equation?[2]

3. (some physics knowledge required) What do you notice about the second term in the Taylor Series approximation of the relativistic energy function?
4. Show all of the work to derive the Taylor Series centred at $v = 0$ given above.

[2]This is something that people in physics and engineering do all the time – there is some complicated nonlinear relationship that they wish to use, but the first few terms of the Taylor Series captures almost all of the behaviour since the higher-order terms are very very small.

3. Roots 1

Success is the sum of small efforts, repeated day in and day out.
–Zeno of Elea

3.1. Introduction to Numerical Root Finding

In this chapter and the next we want to solve equations using a computer. Because it takes time to get used to doing numerical calculations we have split this material over two weeks.[1] In this chapter we will cover the bisection method and fixed point iteration. In the next chapter we will cover the Newton-Raphson and secant methods.

The goal of equation solving is to find the value of the independent variable which makes the equation true. These are the sorts of equations that you learned to solve at school. For a very simple example, *solve for x* if $x + 5 = 2x - 3$. Or, for another example, the equation $x^2 + x = 2x - 7$ is an equation that could be solved with the quadratic formula. The equation $\sin(x) = \frac{\sqrt{2}}{2}$ is an equation which can be solved using some knowledge of trigonometry. The topic of Numerical Root Finding really boils down to approximating the solutions to equations *without* using all of the by-hand techniques that you learned in high school. The down side to everything that we are about to do is that our answers are only ever going to be approximations.

The fact that we will only ever get approximate answers begs the question: *why would we want to do numerical algebra if by-hand techniques exist?* The answers are relatively simple:

- Most equations do not lend themselves to by-hand solutions. The reason you may not have noticed that is that we tend to show you only nice equations that arise in often very simplified situations. When equations arise naturally they are often not *nice.*
- By-hand algebra is often very challenging, quite time consuming, and error prone. You will find that the numerical techniques are quite elegant, work very quickly, and require very little overhead to actually implement and verify.

[1]You may find that we are going too slow for you and that you have the capacity and interest to learn more. In this case I can recommend the optional Appendix C on Numerical Linear Algebra.

Let us first take a look at equations in a more abstract way. Consider the equation $\ell(x) = r(x)$ where $\ell(x)$ and $r(x)$ stand for left-hand and right-hand expressions respectively. To begin solving this equation we can first rewrite it by subtracting the right-hand side from the left to get

$$\ell(x) - r(x) = 0. \tag{3.1}$$

Hence, we can define a function $f(x)$ as $f(x) = \ell(x) - r(x)$ and observe that **every** equation can be written as:

$$\text{Find } x \text{ such that } f(x) = 0. \tag{3.2}$$

This gives us a common language for which to frame all of our numerical algorithms. An x where $f(x) = 0$ is called a **root** of f and thus we have seen that solving an equation is always a root finding problem.

For example, if we want to solve the equation $3\sin(x) + 9 = x^2 - \cos(x)$ then this is the same as solving $(3\sin(x) + 9) - (x^2 - \cos(x)) = 0$. We illustrate this idea in Figure 3.1. You should pause and notice that there is no way that you are going to apply by-hand techniques from algebra to solve this equation … an approximate answer is pretty much our only hope.

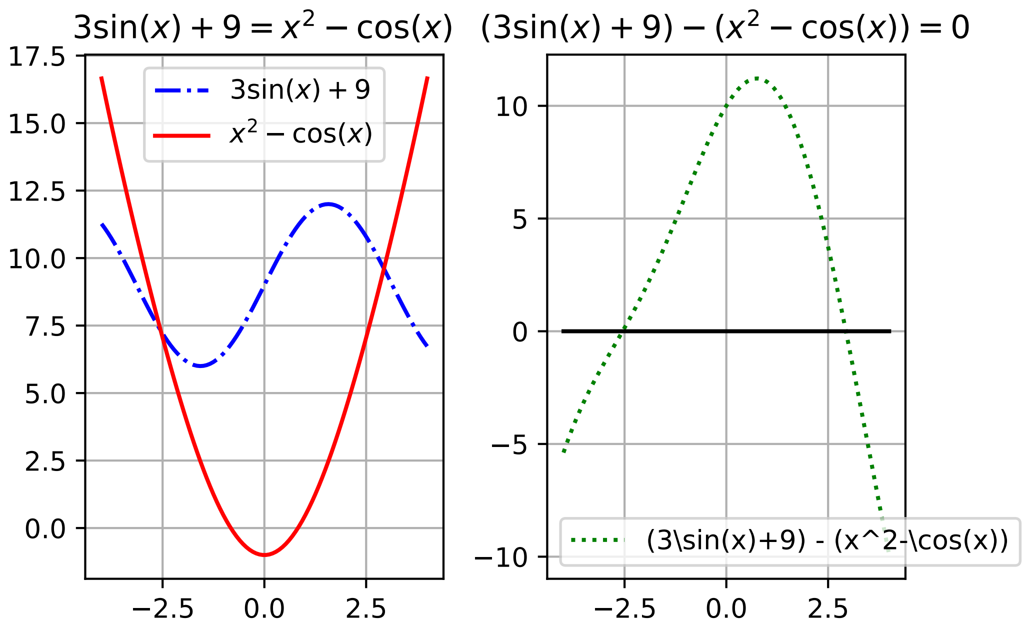


Figure 3.1.: Two ways to visualise the same root finding problem

On the left-hand side of Figure 3.1 we see the solutions as the intersections of the graph of $3\sin(x) + 9$ with the graph of $x^2 - \cos(x)$, and on the right-hand side we see the solutions as the intersections of the graph of $(3\sin(x) + 9) - (x^2 - \cos(x))$ with the x axis. From either plot we can read off the approximate solutions: $x_1 \approx -2.55$ and $x_2 \approx 2.88$.

Figure 3.1 should demonstrate what we mean when we say that solving equations of the form $\ell(x) = r(x)$ will give the same answer as finding the roots of $f(x) = \ell(x) - r(x)$.

We now have one way to view every equation-solving problem. As we will see in this chapter, if $f(x)$ has certain properties then different numerical techniques for solving the equation will apply – and some will be much faster and more accurate than others. In the following sections you will develop several different techniques for solving equations of the form $f(x) = 0$. You will start with the simplest techniques to implement and then move to the more powerful techniques that use some ideas from Calculus to understand and analyse. Throughout this chapter you will also work to quantify the amount of error that one makes when using these techniques.

This chapter is split over two weeks. In the first week we will cover the bisection method and fixed point iteration. In the second week we will cover the Newton-Raphson and secant methods.

3.2. The Bisection Method

3.2.1. Intuition

This section contains several discussion exercises to get you thinking about the bisection method. You may not have anyone around to discuss with, in which case you should just discuss the questions with yourself, i.e., think through the questions. When you next meet with your group you can then discuss them with your peers.

Exercise 3.1. A friend tells you that she is thinking of a number between 1 and 100. She will allow you multiple guesses with some feedback for where the mystery number falls. How do you systematically go about guessing the mystery number? Is there an optimal strategy?

For example, the conversation might go like this.

- Sally: I am thinking of a number between 1 and 100.
- Joe: Is it 35?
- Sally: No, 35 is too low.
- Joe: Is it 99?
- Sally: No, 99 is too high.
- ...

Exercise 3.2. Imagine that Sally likes to formulate her answer not in the form "x is too small" or "x is too large" but in the form "$f(x)$ is positive" or "$f(x)$ is negative". If she uses $f(x) = x - x_0$, where x_0 is Sally's chosen number then her new answers contain exactly the same information as her previous answers? Can you now explain how Sally's game is a root finding game?

Exercise 3.3. Now go and play the game with other functions $f(x)$. Choose someone from your group to be Sally and someone else to be Joe. Sally can choose a continuous function and Joe needs to guess its root. Does your strategy still allow Joe to find the root of $f(x)$ at least approximately? When should the game stop? Does Sally need to give Joe some extra information to give him a fighting chance?

Exercise 3.4. Was it necessary to say that Sally's function was continuous? Does your strategy work if the function is not continuous.

Now let us get to the maths. We will start the mathematical discussion with a theorem from Calculus.

Theorem 3.1 (The Intermediate Value Theorem (IVT))**.** *If $f(x)$ is a continuous function on the closed interval $[a, b]$ and y_* lies between $f(a)$ and $f(b)$, then there exists some point $x_* \in [a, b]$ such that $f(x_*) = y_*$.*

Exercise 3.5. Draw a picture of what the intermediate value theorem says graphically.

Exercise 3.6. If $y_* = 0$ the Intermediate Value Theorem gives us important information about solving equations. What does it tell us? Fill in the blanks in the following corollary.

Corollary 3.1. *If $f(x)$ is a continuous function on the closed interval $[a,b]$ and if $f(a)$ and $f(b)$ have opposite signs then from the Intermediate Value Theorem we know that there exists some point $x_* \in [a,b]$ such that ______.*

The Intermediate Value Theorem (IVT) and its corollary are *existence theorems* in the sense that they tell us that some point exists. The annoying thing about mathematical existence theorems is that they typically do not tell us *how* to find the point that is guaranteed to exist. The method that you developed in Exercise 3.1 to Exercise 3.3 gives one possible way to find the root.

In those exercises you likely came up with an algorithm such as this:

- Say we know that a continuous function has opposite signs at $x = a$ and $x = b$.
- Guess that the root is at the midpoint $m = \frac{a+b}{2}$.
- By using the signs of the function, narrow the interval that contains the root to either $[a,m]$ or $[m,b]$.
- Repeat until the interval is small enough.

Now we will turn this strategy into computer code that will simply play the game for us. But first we need to pay careful attention to some of the mathematical details.

Exercise 3.7. Where is the Intermediate Value Theorem used in the root-guessing strategy?

Exercise 3.8. Why was it important that the function $f(x)$ is continuous when playing this root-guessing game? Provide a few sketches to demonstrate your answer.

3.2.2. Implementation

Exercise 3.9 (The Bisection Method)**.** Goal: We want to solve the equation $f(x) = 0$ for x assuming that the solution x^* is in the interval $[a, b]$. To understand the Bisection Method, have someone in the group sketch a graph of the function $f(x)$ on the interval $[a, b]$. Then together, draw in the steps of the following algorithm:

The Algorithm: We assume that your group member has followed the instructions above and has sketched the graph of a function $f(x)$ that is continuous on the interval $[a, b]$. To make approximations of the solutions to the equation $f(x) = 0$, do the following:

1. Check to see if $f(a)$ and $f(b)$ have opposite signs. You can do this taking the product of $f(a)$ and $f(b)$.
    - If $f(a)$ and $f(b)$ have different signs then what does the IVT tell you?
    - If $f(a)$ and $f(b)$ have the same sign then what does the IVT not tell you? What should you do in this case?
    - Why does the product of $f(a)$ and $f(b)$ tell us something about the signs of the two numbers?
2. Compute the midpoint of the closed interval, $m = \frac{a+b}{2}$. Mark the result in your plot so that you can see the value of $f(m)$.
    - Will m always be a better guess of the root than a or b? Why?
    - What should you do here if $f(m)$ is really close to zero?
3. Compare the signs of $f(a)$ versus $f(m)$ and $f(b)$ versus $f(m)$.
    - What do you do if $f(a)$ and $f(m)$ have opposite signs?
    - What do you do if $f(m)$ and $f(b)$ have opposite signs?
4. Choose the new interval $[a, m]$ or $[m, b]$ and repeat steps 2 and 3 for this interval. Stop when the interval containing the root is small enough.

Exercise 3.10. We want to write a Python function for the Bisection Method. Instead of jumping straight into writing the code we should first come up with the structure of the code. It is often helpful to outline the structure as comments in your file. Use the template below and complete the comments. Note how the function starts with a so-called docstring that describes what the function does and explains the function parameters and its return value. This is standard practice and is how the help text is generated that you see when you hover over a function name in your code.

Don't write the code yet, just complete the comments. I recommend switching off the AI for this exercise because otherwise the AI will keep already suggesting the code while you write the comments.

```python
def bisection(f, a, b, tol=1e-5):
    """
    Find a root of f(x) in the interval [a, b] using the bisection method.

    Parameters:
        f   : function, the function for which we seek a root
        a   : float, left endpoint of the interval
        b   : float, right endpoint of the interval
        tol : float, stopping tolerance

    Returns:
        float: approximate root of f(x)
    """

    # check that a and b have opposite signs
    # if not, return an error and stop

    # calculate the midpoint m = (a+b)/2

    # start a while loop that runs while the interval is
    # larger than 2 * tol

        # if ...
        # elif ...
        # elif ...

        # Calculate midpoint of new interval

    # end the while loop
    # return the approximate root
```

Exercise 3.11. Now use the comments from Exercise 3.10 as structure to complete a Python function for the Bisection Method. Test your Bisection Method code on the following equations. For each equation make a plot of the function on the interval to verify your result.

1. $x^2 - 2 = 0$ on $x \in [0, 2]$

2. $\sin(x) + x^2 = 2\log(x) + 5$ on $x \in [1, 5]$
3. $3\sin(x) + 9 = x^2 + \cos(x)$ on $x \in [1, 5]$

3.2.3. Analysis

After we build any root finding algorithm we need to stop and think about how it will perform on new problems. The questions that we typically have for a root-finding algorithm are:

- Will the algorithm always converge to a solution?
- How fast will the algorithm converge to a solution?
- Are there any pitfalls that we should be aware of when using the algorithm?

Exercise 3.12. Discussion:

- What must be true in order to use the bisection method?
- Does the bisection method work if the Intermediate Value Theorem does not apply? (Hint: what does it mean for the IVT to "not apply?")
- If there is a root of a continuous function $f(x)$ between $x = a$ and $x = b$, will the bisection method always be able to find it? Why / why not?

Next we will focus on a deeper mathematical analysis that will allow us to determine exactly how fast the bisection method actually converges to within a pre-set tolerance. Work through the next problem to develop a formula that tells you exactly how many steps the bisection method needs to take before it gets close enough to the true solution.

Exercise 3.13. Let $f(x)$ be a continuous function on the interval $[a, b]$ and assume that $f(a) \cdot f(b) < 0$. A recurring theme in Numerical Analysis is to approximate some mathematical thing to within some tolerance. For example, if we want to approximate the solution to the equation $f(x) = 0$ to within ε with the bisection method, we should be able to figure out how many steps it will take to achieve that goal.

1. Let us say that $a = 3$ and $b = 8$ and $f(a) \cdot f(b) < 0$ for some continuous function $f(x)$. The width of this interval is 5, so if we guess that the root is $m = (3 + 8)/2 = 5.5$ then our error is less than $5/2$. In the more general setting, if there is a root of a continuous function in the interval $[a, b]$ then how far off could the midpoint approximation of the root be? In other words, what is the error in using $m = (a + b)/2$ as the approximation of the root?

2. The bisection method cuts the width of the interval down to a smaller size at every step. As such, the approximation error gets smaller at every step. Fill in the blanks in the following table to see the pattern in how the approximation error changes with each iteration.

Iteration	**Width of Interval**	**Maximal Error**
1	$\lvert b-a \rvert$	$\frac{\lvert b-a \rvert}{2}$
2	$\frac{\lvert b-a \rvert}{2}$	
3	$\frac{\lvert b-a \rvert}{2^2}$	
$\vdots$	$\vdots$	$\vdots$
n	$\frac{\lvert b-a \rvert}{2^{n-1}}$	

3. Now to the key question:
If we want to approximate the solution to the equation $f(x) = 0$ to within some tolerance ε then how many iterations of the bisection method do we need to take?
Hint: Set the n^{th} approximation error from the table equal to ε. What should you solve for from there?

In Exercise 3.13 you actually proved the following theorem.

Theorem 3.2 (Convergence Rate of the Bisection Method)**.** *If $f(x)$ is a continuous function with a root in the interval $[a, b]$ and if the bisection method is performed to find the root then:*

- *The error between the actual root and the approximate root will decrease by a factor of 2 at every iteration.*
- *If we want the approximate root found by the bisection method to be within a tolerance of ε then*

$$\frac{|b-a|}{2^n} = \varepsilon \tag{3.3}$$

where n is the number of iterations that it takes to achieve that tolerance.

Solving for the number n of iterations we get

$$n = \log_2 \left(\frac{|b-a|}{\varepsilon} \right). \tag{3.4}$$

Rounding the value of n up to the next integer gives the number of iterations necessary to approximate the root to a precision less than ε.

Exercise 3.14. Apply what you have learned in Theorem 3.2 to determine how many iterations the bisection method will take to approximate the solution of the equation

$$3\sin(x) + 9 = x^2 + \cos(x)$$

on $x \in [1, 5]$ to within a tolerance of 10^{-6}.

Now create a second version of your Python Bisection Method function that uses a `for` loop that takes the exact number of steps required to guarantee that the approximation to the root lies within a requested tolerance. This should be in contrast to your first version which likely used a `while` loop to decide when to stop. Is there an advantage to using one of these version of the Bisection Method over the other?

The final type of analysis that we should do on the bisection method is to make plots of the error between the approximate solution that the bisection method gives you and the exact solution to the equation. This is a bit of a funny thing! Stop and think about this for a second: *if you know the exact solution to the equation then why are you solving it numerically in the first place!?!?* However, whenever you build an algorithm you need to test it on problems where you actually do know the answer so that you can be somewhat sure that it is not giving you nonsense. Furthermore, analysis like this tells us how fast the algorithm is expected to perform.

From Theorem 3.2 you know that the bisection method cuts the interval in half at every iteration. You proved in Exercise 3.13 that the error given by the bisection method is therefore cut in half at every iteration as well. The following example demonstrate this theorem graphically.

Example 3.1. Let us solve the very simple equation $x^2 - 2 = 0$ for x to get the solution $x = \sqrt{2}$ with the bisection method. Since we know the exact answer we can compare the exact answer to the value of the midpoint given at each iteration and calculate an absolute error:

$$\text{Absolute Error} = |\text{Approximate Solution} - \text{Exact Solution}|. \tag{3.5}$$

Let us write a Python function that implements the bisection method and collects the absolute errors at each iteration into a list.

We can now use this function to see the absolute error at each iteration when solving the equation $x^2 - 2 = 0$ with the bisection method.

```
[np.float64(0.08578643762690485),
 np.float64(0.16421356237309515),
 np.float64(0.039213562373095145),
 np.float64(0.023286437626904855),
 np.float64(0.007963562373095145),
 np.float64(0.0076614376269048545),
 np.float64(0.00015106237309514547),
 np.float64(0.0037551876269048545),
 np.float64(0.0018020626269048545),
 np.float64(0.0008255001269048545),
 np.float64(0.0003372188769048545),
 np.float64(9.307825190485453e-05),
 np.float64(2.8992060595145475e-05),
 np.float64(3.2043095654854525e-05),
 np.float64(1.5255175298545254e-06),
 np.float64(1.3733271532645475e-05),
 np.float64(6.103877001395475e-06),
 np.float64(2.2891797357704746e-06),
 np.float64(3.818311029579746e-07),
 np.float64(5.718432134482754e-07),
 np.float64(9.500605524515038e-08),
 np.float64(1.4341252385641212e-07),
 np.float64(2.4203234305630872e-08)]
```

Next we write a function to plot the absolute error on the vertical axis and the iteration number on the horizontal axis. We get Figure 3.2. As expected, the absolute error follows an exponentially decreasing trend. Notice that it is not a completely smooth curve since we will have some jumps in the accuracy just due to the fact that sometimes the root will be near the midpoint of the interval and sometimes it will not be.

Without Theorem 3.2 it would be rather hard to tell what the exact behaviour is in the exponential plot above. We know from Theorem 3.2 that the error will divide by 2 at

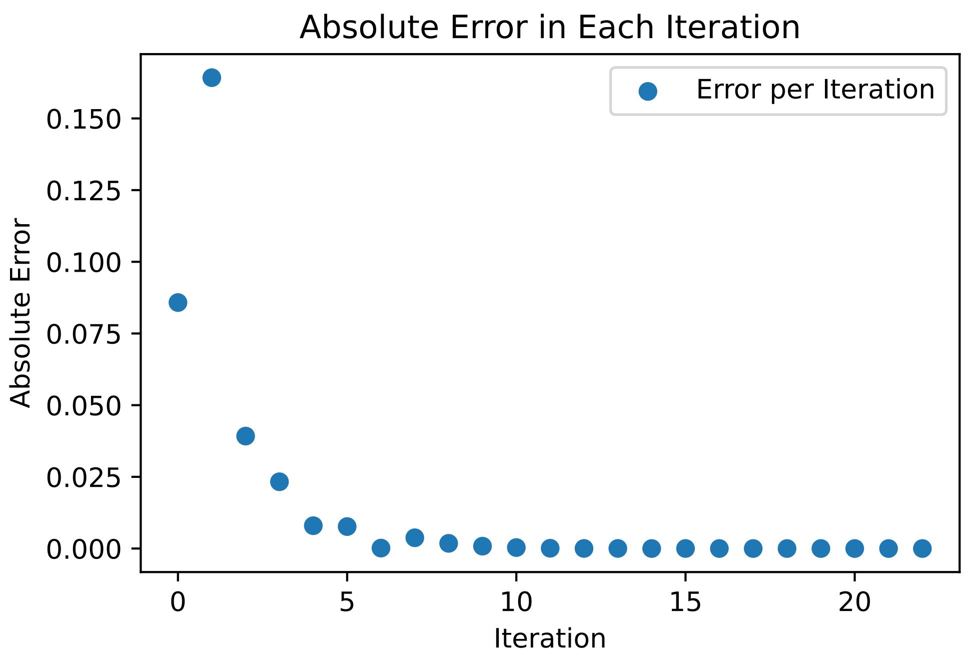


Figure 3.2.: The evolution of the absolute error when solving the equation $x^2 - 2 = 0$ with the bisection method.

every step, so if we instead plot the base-2 logarithm of the absolute error against the iteration number we should see a linear trend as shown in Figure 3.3.

There will be times later in this course where we will not have a nice theorem like Theorem 3.2 and instead we will need to deduce the relationship from plots like these.

1. The trend is linear since logarithms and exponential functions are inverses. Hence, applying a logarithm to an exponential will give a linear function.
2. The slope of the resulting linear function should be -1 in this case since we are dividing by a factor of 2 each iteration. You can visually verify that the slope in the plot above follows this trend (the red dashed line in the plot is shown to help you see the slope).

Exercise 3.15. Carefully read and understand all of the details of the previous example and plots. Then create plots similar to that example to solve the equation

$$e^{x-3} + \sqrt{x+6} = 4$$

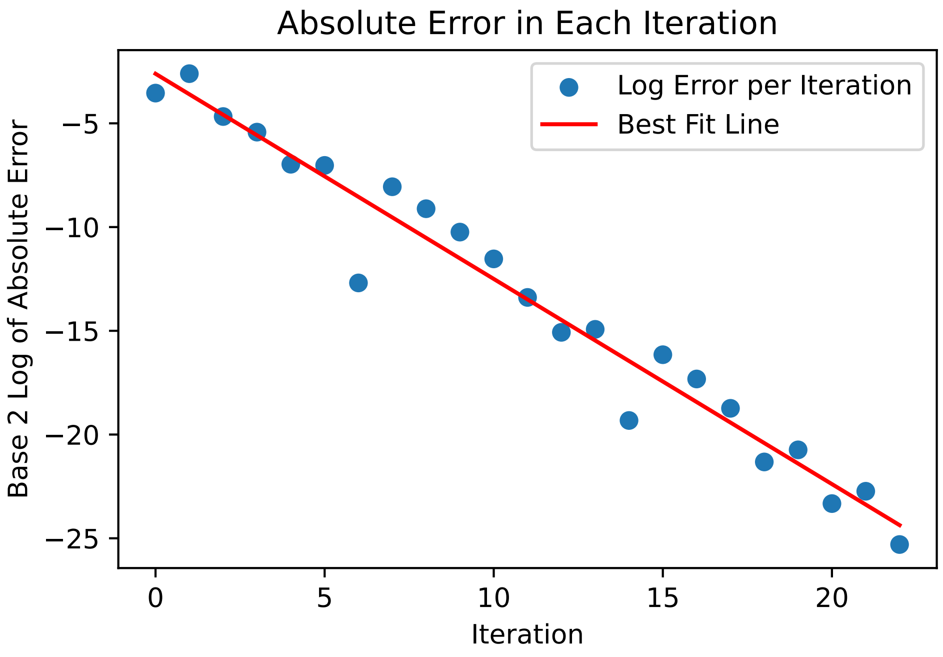


Figure 3.3.: Iteration number vs the base-2 logarithm of the absolute error. Notice the slope of -1 indicating that the error is divided by a factor of 2 at each step of the algorithm.

on the interval $x \in [0,5]$ to which the exact solution is $x = 3$. You should see the same basic behaviour based on the theorem that you proved in Exercise 3.13. If you do not see the same basic behaviour then something has gone wrong. Also, the plot will look much more regular than in the previous example. Can you explain why?

Example 3.2. Another plot that numerical analysts use quite frequently for determining how an algorithm is behaving as it progresses is shown in Figure 3.4. and is defined by the following axes:

- The horizontal axis is the absolute error at iteration n.
- The vertical axis is the absolute error at iteration $n+1$.

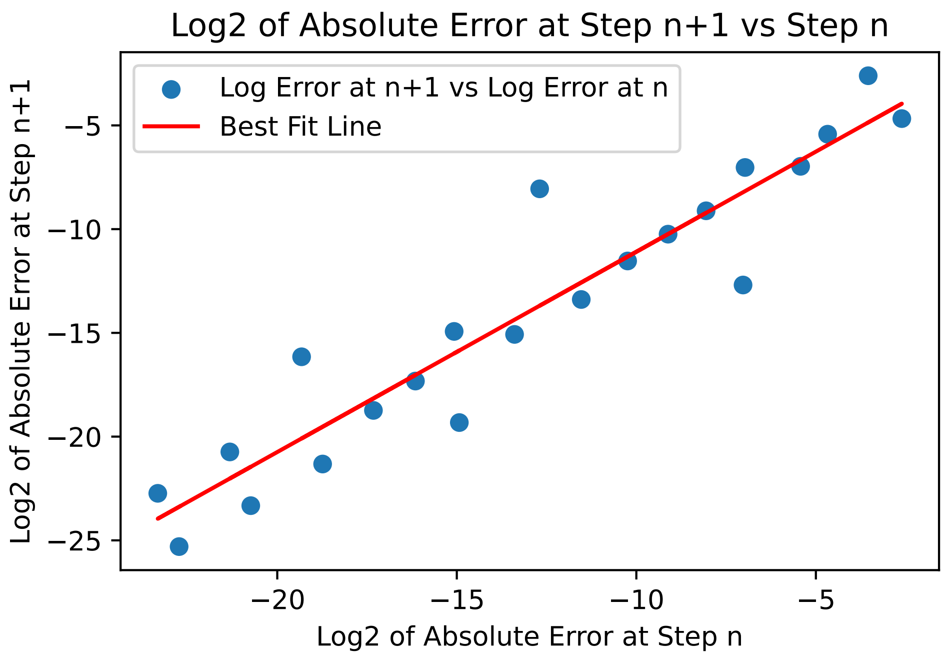


Figure 3.4.: The base-2 logarithm of the absolute error at iteration n vs the base-2 logarithm of the absolute error at iteration $n+1$.

This type of plot takes a bit of explaining the first time you see it. Each point in the plot corresponds to an iteration of the algorithm. The x-coordinate of each point is the base-2 logarithm of the absolute error at step n and the y-coordinate is the base-2 logarithm of the absolute error at step $n+1$. The initial interations are on the right-hand side of the plot where the error is the largest (this will be where the algorithm starts). As the iterations progress and the error decreases the points move to the left-hand side of the plot. Examining the slope of the trend line in this plot shows how we expect the error to progress from step to step. The slope appears to be about 1 in Figure 3.4 and the intercept appears to be about -1. In this case we used a base-2 logarithm for each axis so we have just empirically shown that

$$\log_2(\text{absolute error at step } n+1) \approx 1 \cdot \log_2(\text{absolute error at step } n) - 1. \tag{3.6}$$

Exponentiating both sides we see that this linear relationship turns into (You should stop now and do this algebra.) Rearranging a bit more we get

$$(\text{absolute error at step } n+1) = \frac{1}{2}(\text{absolute error at step } n), \tag{3.7}$$

exactly as expected!! Pause and ponder this result for a second – we just empirically verified the convergence rate for the bisection method just by examining Figure 3.4. That's what makes these types of plots so useful!

Exercise 3.16. Reproduce plots like that in the previous example but for the equation that you used in Exercise 3.15. Again check that the plots have the expected shape.

3.3. Fixed Point Iteration

We will now investigate a different problem that is closely related to root finding: the fixed point problem. Given a function g (of one real argument with real values), we look for a number p such that

$$g(p) = p.$$

This p is called a **fixed point** of g.

Any root finding problem $f(x) = 0$ can be reformulated as a fixed point problem, and this can be done in many (in fact, infinitely many) ways. For example, given f, we can define $g(x) := f(x) + x$; then

$$f(x) = 0 \quad \Leftrightarrow \quad g(x) = x.$$

Just as well, we could set $g(x) := \lambda f(x) + x$ with any $\lambda \in \mathbb{R}\backslash\{0\}$, and there are many other possibilities.

The heuristic idea for approximating a fixed point of a function g is quite simple. We take an initial approximation x_0 and calculate subsequent approximations using the formula

$$x_n := g(x_{n-1}).$$

A graphical representation of this sequence when $g = \cos$ and $x_0 = 2$ is shown in Figure 3.5.

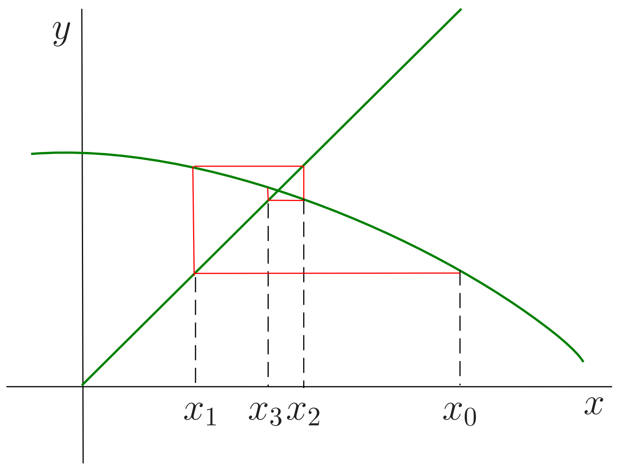


Figure 3.5.: Fixed point iteration

Exercise 3.17. The plot that emerges in Figure 3.5 is known as a cobweb diagram, for obvious reason. Explain to others in your group what is happening in the animation in Figure 3.5 and how that animation is related to the fixed point iteration $x_n = \cos(x_{n-1})$.

Exercise 3.18. The animation in Figure 3.5 is a graphical representation of the fixed point iteration $x_n = \cos(x_{n-1})$. Use Python to calculate the first 10 iterations of this sequence with $x_0 = 0.2$. Use that to get an estimate of the solution to the equation $\cos(x) - x = 0$.

Why is the sequence (x_n) expected to approximate a fixed point? Suppose for a moment that the sequence (x_n) converges to some number p, and that g is continuous. Then

$$p = \lim_{n\to\infty} x_n = \lim_{n\to\infty} g(x_{n-1}) = g\left(\lim_{n\to\infty} x_{n-1}\right) = g(p). \tag{3.8}$$

Thus, *if* the sequence converges, then it converges to a fixed point. However, this resolves the problem only partially. One would like to know:

- Under what conditions does the sequence (x_n) converge?
- How fast is the convergence, i.e., can one obtain an estimate for the approximation error?

So there is much for you to investigate!

Exercise 3.19. Copy the two plots in Figure 3.6 to a piece of paper and draw the first few iterations of the fixed point iteration $x_n = g(x_{n-1})$ on each of them. In the first plot start with $x_0 = 0.2$ and in the second plot start with $x_0 = 1.5$ and in the second plot start with $x = 0.9$ What do you observe about the convergence of the sequence in each case?

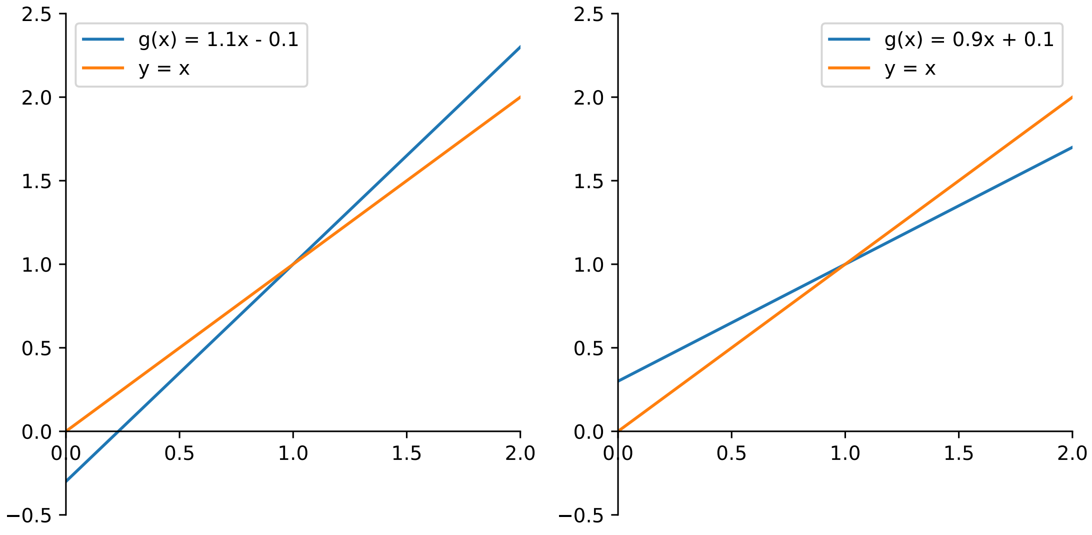


Figure 3.6.: Two plots for practicing your cobweb skills.

Can you make some conjectures about when the sequence (x_n) will converge to a fixed point and when it will not?

Exercise 3.20. Make similar plots as in the previous exercise but with different slopes of the blue line. Can you make some conjectures about how the speed of convergence is related to the slope of the blue line?

Now see if your observations are in agreement with the following theorem:

Theorem 3.3 (Fixed Point Theorem)**.** *Suppose that $g : [a, b] \to [a, b]$ is differentiable, and that there exists $0 < k < 1$ such that*

$$|g'(x)| \leq k \quad \textit{for all } x \in (a, b). \tag{3.9}$$

Then, g has a unique fixed point $p \in [a, b]$; and for any choice of $x_0 \in [a, b]$, the sequence defined by

$$x_n := g(x_{n-1}) \quad \textit{for all } n \geq 1 \tag{3.10}$$

converges to p. The following estimate holds:

$$|p - x_n| \leq k^n |p - x_0| \quad \textit{for all } n \geq 1. \tag{3.11}$$

Proof. The proof of this theorem is not difficult, but you can skip it and go directly to Exercise 3.21 if you feel that the theorem makes intuitive sense and you are not interested in proofs.

We first show that g has a fixed point p in $[a, b]$. If $g(a) = a$ or $g(b) = b$ then g has a fixed point at an endpoint. If not, then it must be true that $g(a) > a$ and $g(b) < b$. This means that the function $h(x) := g(x) - x$ satisfies

$$h(a) = g(a) - a > 0, \quad h(b) = g(b) - b < 0$$

and since h is continuous on $[a, b]$ the Intermediate Value Theorem guarantees the existence of $p \in (a, b)$ for which $h(p) = 0$, equivalently $g(p) = p$, so that p is a fixed point of g.

To show that the fixed point is unique, suppose that $q \neq p$ is a fixed point of g in $[a, b]$. The Mean Value Theorem implies the existence of a number $\xi \in (\min\{p, q\}, \max\{p, q\}) \subseteq (a, b)$ such that

$$\frac{g(p) - g(q)}{p - q} = g'(\xi).$$

Then

$$|p - q| = |g(p) - g(q)| = |(p - q)g'(\xi)| = |p - q||g'(\xi)| \leq k|p - q| < |p - q|,$$

where the inequalities follow from Eq. 3.9. This is a contradiction, which must have come from the assumption $p \neq q$. Thus $p = q$ and the fixed point is unique.

Since g maps $[a, b]$ onto itself, the sequence $\{x_n\}$ is well defined. For each $n \geq 0$ the Mean Value Theorem gives the existence of a $\xi \in (\min\{x_n, p\}, \max\{x_n, p\}) \subseteq (a, b)$ such that

$$\frac{g(x_n) - g(p)}{x_n - p} = g'(\xi).$$

Thus for each $n \geq 1$ by Eq. 3.9, Eq. 3.10

$$|x_n - p| = |g(x_{n-1}) - g(p)| = |(x_{n-1} - p)g'(\xi)| = |x_{n-1} - p||g'(\xi)| \leq k|x_{n-1} - p|.$$

Applying this inequality inductively, we obtain the error estimate Eq. 3.11. Moreover since $k < 1$ we have

$$\lim_{n\to\infty} |x_n - p| \leq \lim_{n\to\infty} k^n |x_0 - p| = 0,$$

which implies that (x_n) converges to p. □

Exercise 3.21. This exercise shows why the conditions of the Theorem 3.3 are important.

The equation

$$f(x) = x^2 - 2 = 0$$

has a unique root $\sqrt{2}$ in $[1, 2]$. There are many ways of writing this equation in the form $x = g(x)$; we consider two of them:

$$x = g(x) = x - (x^2 - 2), \quad x = h(x) = x - \frac{x^2 - 2}{3}.$$

Calculate the first terms in the sequences generated by the fixed point iteration procedures $x_n = g(x_{n-1})$ and $x_n = h(x_{n-1})$ with start value $x_0 = 1$. Which of these fixed point problems generate a rapidly converging sequence? Calculate the derivatives of g and h and check if the conditions of the fixed point theorem are satisfied.

The previous exercise illustrates that one needs to be careful in rewriting root finding problems as fixed point problems—there are many ways to do so, but not all lead to a good approximation. In the next section about Newton's method we will discover a very good choice.

Note at this point that Theorem 3.3 gives only sufficient conditions for convergence; in practice, convergence might occur even if the conditions are violated.

Exercise 3.22. In this exercise you will write a Python function to implement the fixed point iteration algorithm.

For implementing the fixed point method as a computer algorithm, there's one more complication to be taken into account: how many steps of the iteration should be taken, i.e., how large should n be chosen, in order to reach the desired precision? The error estimate in Eq. 3.11 is often difficult to use for this purpose because it involves estimates on the derivative of g which may not be known.

Instead, one uses a different **stopping condition** for the algorithm. Since the sequence is expected to converge rapidly, one uses the difference $|x_n - x_{n-1}|$ to measure the precision reached. If this difference is below a specified limit, say τ, the iteration is stopped. Since it is possible that the iteration does *not* converge—see the example above—one would also stop the iteration (with an error message) if a certain number of steps is exceeded, in order to avoid infinite loops. In pseudocode the fixed point iteration algorithm is then implemented as follows:

Fixed point iteration

```
 1 : function $FixedPoint(g, x_0, tol, N)$            ♯ function $g$, start point $x_0$,
 2 :   $x \leftarrow x_0$; $n \leftarrow 0$                      ♯ tolerance $tol$,
 3 :   for $i \leftarrow 1$ to $N$                          ♯ max. num. of iterations $N$
 4 :     $y \leftarrow x$; $x \leftarrow g(x)$
 5 :     if $|y - x| < tol$ then                    ♯ Desired tolerance reached
 6 :       return $x$
 7 :     end if
 8 :   end for
 9 :   exception(Iteration has not converged)
10 : end function
```

Implement this algorithm in Python. Use it to approximate the fixed point of the function $g(x) = \cos(x)$ with start value $x_0 = 2$ and tolerance $tol = 10^{-8}$.

Further reading: Section 2.2 of (Burden and Faires 2010).

---

4. Roots 2

In preparation.

5. Derivatives, Integrals

In preparation.

6. Optimisation

In preparation.

7. ODEs 1

In preparation.

8. ODEs 2

In preparation.

9. PDEs 1

In preparation.

10. PDEs 2

In preparation.

A. Python

Simple is better than complex.
–Guido van Rossum (creator of Python)

In this appendix we will walk through some of the basics of using Python — the popular general-purpose programming language that we will use throughout this module.

For most of you this material will not be new. For example, you have probably seen it in your first year "Mathematical Programming & Skills" module. But for some of you this may be entirely new. You will have some notion of what a programming language "is" and "does", but you may never have written any code. That is all right.

If you are new to Python, don't feel that you need to work through this appendix in one go. Instead, spread the work over the first two weeks of the course and interlace it with your work on the first two chapters.

There is a lot of material in this appendix. Do not feel that you need to learn it all by hard. The idea is just that you should have seen the various language constructs once. Your familiarity with them will come automatically later when you use them throughout the course.

A.1. Why Python?

We are going to be using Python since

- Python is free,
- Python is very widely used,
- Python is flexible,
- Python is relatively easy to learn,
- and Python is quite powerful.

It is important to keep in mind that Python is a general purpose language that we will be using for Scientific Computing. The purpose of Scientific Computing is *not* to build apps, build software, manage databases, or develop user interfaces. Instead, Scientific Computing is the use of a computer programming language (like Python) along with mathematics to solve scientific and mathematical problems. For this reason it is definitely not our purpose to write an all-encompassing guide for how to use Python. We will only cover what is necessary for our computing needs. You will learn more as the course progresses, so use this appendix just to get going with the language. To keep things as simple as possible, we will for example not use object oriented programming, so will not introduce classes and methods.

We are also definitely not saying that Python is the best language for scientific computing under all circumstances. The reason there are so many scientific programming languages coexisting is that each has particular strengths that make it the best option for particular applications. But we are saying that Python is so widely used that everyone should know Python.

There is an overwhelming abundance of information available about Python and the suite of tools that we will frequently use.

- Python https://www.python.org/,
- `numpy` (numerical Python) https://www.numpy.org/,
- `matplotlib` (a suite of plotting tools) https://matplotlib.org/,
- `scipy` (scientific Python) https://www.scipy.org/.

These tools together provide all of the computational power that we will need. And they are free!

A.2. Python Programming Basics

If you are already very practised in using Python then you can jump straight to Section A.6 with the coding exercises. But if you are new to Python or your Python skills are a bit rusty, then you will benefit from working through all the examples and exercises below, making sure you copy and paste all the code into your Colab notebook and run it there, and then critically evaluate and understand the output.

A.2.1. Variables

Variable names in Python can contain letters (lower case or capital), numbers 0-9, and some special characters such as the underscore. Variable names must start with a letter. There are a bunch of reserved words that you can not use for your variable names because they have a special meaning in the Python syntax. Python will let you know with a syntax error if you try to use a reserved word for a variable name.

You can do the typical things with variables. Assignment is with an equal sign (be careful R users, we will not be using the left-pointing arrow here!).

Warning: When defining numerical variables you do not always get floating point numbers. In some programming languages, if you write `x=1` then automatically `x` is saved as 1.0; a floating point number, not an integer. In Python however, if you assign `x=1` it is defined as an integer (with no decimal digits) but if you assign `x=1.0` it is assigned as a floating point number.

```
# assign some variables
x = 7 # integer assignment of the integer 7
y = 7.0 # floating point assignment of the decimal number 7.0
print("The variable x has the value", x, " and has type", type(x), ". \n")
print("The variable y has the value", y, " and has type", type(y), ". \n")
```

Remember to copy each code block to your own notebook, execute it and look at the output. To copy the code from this guide to your notebook you can use the "Copy to Clipboard" icon that pops up in the top right corner of a code block when you hover over that code block.

```
# multiplying by a float will convert an integer to a float
x = 7 # integer assignment of the integer 7
print("Multiplying x by 1.0 gives", 1.0*x)
print("The type of this value is", type(1.0*x), ". \n")
```

The allowed mathematical operations are:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Integer Division (modular division): `//` and `%`
- Exponents: `**`

That's right, the caret key, `^`, is NOT an exponent in Python (sigh). Instead we have to get used to `**` for exponents.

```
x = 7.0
y = x**2 # square the value in x
y
```

Exercise A.1. Write code to define positive integers a, b and c of your own choosing. Then calculate a^2, b^2 and c^2. When you have all three values computed, check to see if your three values form a Pythagorean Triple so that $a^2 + b^2 = c^2$. Have Python simply say True or False to verify that you do, or do not, have a Pythagorean Triple defined. **Hint:** You will need to use the `==` Boolean check just like in other programming languages.

A.2.2. Indexing and Lists

Lists are a key component to storing data in Python. Lists are exactly what the name says: lists of things (in our case, usually the entries are floating point numbers).

Warning: Python indexing starts at 0 whereas some other programming languages have indexing starting at 1. In other words, the first entry of a list has index 0, the second entry as index 1, and so on. We just have to keep this in mind.

We can extract a part of a list using the syntax `name[start:stop]` which extracts elements between index `start` and `stop-1`. Take note that Python stops reading at the second to last index. This often catches people off guard when they first start with Python.

Example A.1 (Lists and Indexing)**.** Let us look at a few examples of indexing from lists. In this example we will use the list of numbers 0 through 8. This list contains 9 numbers indexed from 0 to 8.

- Create the list of numbers 0 through 8

```
my_list = [0,1,2,3,4,5,6,7,8]
```

- Output the list

```
my_list
```

- Select only the element with index 0.

```
my_list[0]
```

- Select all elements up to, but not including, the third element of `my_list`.

```
my_list[:2]
```

- Select the last element of `my_list` (this is a handy trick!).

```
my_list[-1]
```

- Select the elements indexed 1 through 4. Beware! This is not the first through fifth element.

```
my_list[1:5]
```

- Select every other element in the list starting with the first.

```
my_list[0::2]
```

- Select the last three elements of `my_list`

```
my_list[-3:]
```

In Python, elements in a list do not need to be the same type. You can mix integers, floats, strings, lists, etc.

Example A.2. In this example we see a list of several items that have different data types: float, integer, string, and complex. Note that the imaginary number i is represented by $1j$ in Python. This use of j instead of i is common in some scientific disciplines and is just another thing that we Mathematicians will need to get used to in Python.

```
MixedList = [1.0, 7, 'Bob', 1-1j]
print(MixedList)
print(type(MixedList[0]))
print(type(MixedList[1]))
print(type(MixedList[2]))
print(type(MixedList[3]))
# Notice that we use 1j for the imaginary number "i".
```

Exercise A.2. In this exercise you will put your new list skills into practice.

1. Create the list of the first several Fibonacci numbers:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89. \tag{A.1}$$

2. Print the first four elements of the list.
3. Print every third element of the list starting from the first.
4. Print the last element of the list.
5. Print the list in reverse order.
6. Print the list starting at the last element and counting backward by every other element.

A.2.3. List Operations

Python is awesome about allowing you to do things like appending items to lists, removing items from lists, and inserting items into lists. Note in all of the examples below that we are using the code
`variable.method`
where you put the variable name, a dot, and the thing that you would like to do to that variable. For example, `my_list.append(7)` will append the number 7 to the list `my_list`. We say that `append` is a "method" of the list `my_list`. This is a common programming feature in Python and we will use it often.

Example A.3. The `.append` method can be used to append an element to the end of a list.

```python
my_list = [0,1,2,3]
print(my_list)
# Append the string 'a' to the end of the list
my_list.append('a')
print(my_list)
# Do it again ... just for fun
my_list.append('a')
print(my_list)
# Append the number 15 to the end of the list
my_list.append(15)
print(my_list)
```

Example A.4. The `.remove` method can be used to remove an element from a list.

```
# Let us remove the 3
my_list.remove(3)
my_list
```

Example A.5. The `.insert` method can be used to insert an element at a location in a list.

```
# insert the letter `A` at the 0-indexed spot
my_list.insert(0,'A')
# insert the letter `B` at the spot with index 3
my_list.insert(3,'B')
# remember that index 3 means the fourth spot in the list
my_list
```

Exercise A.3. In this exercise you will go a bit further with your list operation skills.

1. Create the list of the first several Lucas Numbers: $1, 3, 4, 7, 11, 18, 29, 47$.
2. Add the next three Lucas Numbers to the end of the list.
3. Remove the number 3 from the list.
4. Insert the 3 back into the list in the correct spot.
5. Print the list in reverse order.
6. Do a few other list operations to this list and report your findings.

A.2.4. Tuples

In Python, a “tuple” is like an ordered pair (or ordered triple, or ordered quadruple, ...) in mathematics. We will occasionally see tuples in our work in numerical analysis so for now let us just give a couple of code snippets showing how to store and read them.

We can define the tuple of numbers $(10, 20)$ in Python as follows:

Example A.6.

```
point = 10, 20
print(point, type(point))
```

We can also define a tuple with parenthesis if we like. Python does not care.

```
point = (10, 20) # now we define the tuple with parenthesis
print(point, type(point))
```

We can then unpack the tuple into components if we wish:

```
x, y = point
print("x = ", x)
print("y = ", y)
```

There are other important data structures in Python that we will not use in this module. These include dictionaries and sets. We will not cover these here because we are trying to keep things simple so that we can concentrate on Numerical Analysis instead. If you are interested in learning more about these data structures, you can find a lot of information about them in the Python documentation.

A.2.5. Control Flow: Loops and If Statements

Any time you need to do some repetitive task with a programming language you can use a loop. Just like in other programming languages, we can do loops and conditional statements in very easy ways in Python. The thing to keep in mind is that the Python language is very white-space-dependent. This means that your indentations need to be correct in order for a loop to work. You could get away with sloppy indention in other languages but not so in Python. Also, in some languages (like R and Java) you need to wrap your loops in curly braces. Again, not so in Python.

Caution: Be really careful of the white space in your code when you write loops.

A.2.5.1. `for` Loops

A `for` loop is designed to do a task a certain number of times and then stop. This is a great tool for automating repetitive tasks, but it also nice numerically for building sequences, summing series, or just checking lots of examples. The following are some examples of Python for loops.

Example A.7. Print the first 6 perfect squares.

```
for x in [1,2,3,4,5,6]:
    print(x**2)
```

Often instead of writing the list of integers explicitly one uses the `range()` function, so that this example would be written as

```
for x in range(1,7):
    print(x**2)
```

Note that `range(1,7)` produces the integers from 1 to 6, not from 1 to 7. This is another manifestation of Python's weird 0-based indexing. Of course it is only weird to people who are new to Python. For Pythonists it is perfectly natural.

You can also use `range()` to generate a sequence of numbers with a specific step size. For example, `range(1, 10, 2)` will generate the odd integers from 1 to 9.

```
for x in range(1, 10, 2):
    print(x)
```

Take careful note of the syntax for a for loop as it is the same as for other loops and conditional statements:

- a control statement,
- a colon, a new line,
- indent four spaces,
- some programming statements

When you are done with the loop, just back out of the indention. There is no need for an `end` command or a curly brace. All of the control statements in Python are white-space-dependent.

Example A.8. Print the names in a list.

```
NamesList = ['Alice','Billy','Charlie','Dom','Enrique','Francisco']
for name in NamesList:
    print(name)
```

In Python you can use a more compact notation for `for` loops sometimes. This takes a bit of getting used to, but is super slick!

Example A.9. Create a list of the perfect squares from 1 to 9.

```
# create a list of the perfect squares from 1 to 9
CoolList = [x**2 for x in range(1,10)]
print(CoolList)
# Then print the sum of this list
print("The sum of the first 9 perfect squares is", sum(CoolList))
```

`for` loops can also be used to build sequences, as can be seen in the next couple of examples.

Example A.10. In the following code we write a for loop that outputs a list of the first 7 iterations of the sequence $x_{n+1} = -0.5x_n + 1$ starting with $x_0 = 3$. Notice that we are using the command `x.append` instead of $x[n+1]$ to append the new term to the list. This allows us to grow the length of the list dynamically as the loop progresses.

```
x=[3.0]
for n in range(0,7):
    x.append(-0.5*x[n] + 1)
    print(x) # print the whole list x at each step of the loop
```

Example A.11. As an alternative to the code from the previous example we can pre-allocate the memory in a list of zeros. This is done with the clever code `x = [0] * 10`. Literally multiplying a list by some number, like 10, says to repeat that list 10 times.

Now we will build the sequence with pre-allocated memory.

```
x = [0] * 7
x[0] = 3.0
for n in range(0,6):
    x[n+1] = -0.5*x[n]+1
    print(x) # This print statement shows x at each iteration
```

Exercise A.4. We want to sum the first 100 perfect cubes. Let us do this in two ways.

1. Start off a variable called `total` at 0 and write a `for` loop that adds the next perfect cube to the running total.

2. Write a `for` loop that builds the sequence of the first 100 perfect cubes. After the list has been built find the sum with the `sum()` function.

The answer is: 25,502,500 so check your work.

Exercise A.5. Write a `for` loop that builds the first 20 terms of the sequence $x_{n+1} = 1 - x_n^2$ with $x_0 = 0.1$. Pre-allocate enough memory in your list and then fill it with the terms of the sequence. Only print the list after all of the computations have been completed.

A.2.5.2. `while` Loops

A `while` loop repeats some task (or sequence of tasks) while a logical condition is true. It stops when the logical condition turns from true to false. The structure in Python is the same as with `for` loops.

Example A.12. Print the numbers 0 through 4 and then the word "done." we will do this by starting a counter variable, `i`, at 0 and increment it every time we pass through the loop.

```python
i = 0
while i < 5:
    print(i)
    i += 1 # increment the counter
print("done")
```

Example A.13. Now let us use a while loop to build the sequence of Fibonacci numbers and stop when the newest number in the sequence is greater than 1000. Notice that we want to keep looping until the condition that the last term is greater than 1000 – this is the perfect task for a `while` loop, instead of a `for` loop, since we do not know how many steps it will take before we start the task

```python
Fib = [1,1]
while Fib[-1] <= 1000:
    Fib.append(Fib[-1] + Fib[-2])
print("The last few terms in the list are:\n",Fib[-3:])
```

Exercise A.6. Write a `while` loop that sums the terms in the Fibonacci sequence until the sum is larger than 1000

A.2.5.3. `if` Statements

Conditional (`if`) statements allow you to run a piece of code only under certain conditions. This is handy when you have different tasks to perform under different conditions.

Example A.14. Let us look at a simple example of an `if` statement in Python.

```
Name = "Alice"
if Name == "Alice":
    print("Hello, Alice.  Isn't it a lovely day to learn Python?")
else:
    print("You're not Alice.  Where is Alice?")
```

```
Name = "Billy"
if Name == "Alice":
    print("Hello, Alice.  Isn't it a lovely day to learn Python?")
else:
    print("You're not Alice.  Where is Alice?")
```

Example A.15. For another example, if we get a random number between 0 and 1 we could have Python print a different message depending on whether it was above or below 0.5. Run the code below several times and you will see different results each time.

Note: We have to import the `numpy` package to get the random number generator in Python. Do not worry about that for now. We will talk about packages in a moment.

```
import numpy as np
x = np.random.uniform() # get a random number between 0 and 1
if x < 0.5:
    print(x," is less than a half")
else:
    print(x, "is NOT less than a half")
```

(Take note that the output will change every time you run it.)

Example A.16. In many programming tasks it is handy to have several different choices between tasks instead of just two choices as in the previous examples. This is a job for the `elif` command.

This is the same code as last time except we will make the decision at 0.33 and 0.67.

```python
import numpy as np
x = np.random.rand(1,1) # get a random 1x1 matrix using numpy
x = x[0,0] # pull the entry from the first row and first column
if x < 0.33:
    print(x," < 1/3")
elif x < 0.67:
    print("1/3 <= ",x,"< 2/3")
else:
    print(x, ">= 2/3")
```

(Take note that the output will change every time you run it.)

Exercise A.7. Write code to give the Collatz Sequence

$$x_{n+1} = \begin{cases} x_n/2, & x_n \text{ is even} \\ 3x_n + 1, & \text{otherwise} \end{cases} \tag{A.2}$$

starting with a positive integer of your choosing. The sequence will converge[1] to 1 so your code should stop when the sequence reaches 1.

Hints: To test whether a number `x` is even you can test whether the remainder after dividing by 2 is zero with `(x % 2) == 0`. Also you will want to use the integer division `//` when calculating $x_n/2$.

---

[1]Actually, it is still an open mathematical question whether every integer seed will converge to 1. The Collatz sequence has been checked for many millions of initial seeds and they all converge to 1, but there is no mathematical proof that it will always happen. You will check the conjecture numerically in Exercise A.25

A.2.6. Functions

Mathematicians and programmers talk about functions in very similar ways, but they are not exactly the same. When we say "function" in a programming sense we are talking about a chunk of code that you can pass parameters and expect an output of some sort. This is not unlike the mathematician's version. But unlike a mathematical function, a Python function can also have side effects, like plotting a graph for example. So Python's definition of a function is a bit more flexible than that of a mathematician.

In Python, to define a function we start with `def`, followed by the function's name, any input variables in parenthesis, and a colon. The indented code after the colon is what defines the actions of the function.

Example A.17. The following code defines the polynomial $f(x) = x^3 + 3x^2 + 3x + 1$ and then evaluates the function at a point $x = 2.3$.

```
def f(x):
    return(x**3 + 3*x**2 + 3*x + 1)
f(2.3)
```

Take careful note of several things in the previous example:

- To define the function we cannot just type it like we would see it one paper. This is not how Python recognizes functions.
- Once we have the function defined we can call upon it just like we would on paper.
- We cannot pass symbols into this type of function.[2]

Exercise A.8. Define the function $g(n) = n^2 + n + 41$ as a Python function. Write a loop that gives the output for this function for integers from $n = 0$ to $n = 39$. Euler noticed that each of these outputs is a prime number (check this on your own). Will the function produce a prime for $n = 40$? For $n = 41$?

---

[2]There is the `sympy` package if you want to do symbolic computations, but we will not use that in this module.

Example A.18. One cool thing that you can do with functions is call them recursively. That is, you can call the same function from within the function itself. This turns out to be really handy in several mathematical situations.

Let us define a function for the factorial. This function is naturally going to be recursive in the sense that it calls on itself!

```python
def factorial(n):
    if n==0:
        return(1)
    else:
        return(n*factorial(n-1))
        # Note: we are calling the function recursively.
```

When you run this code there will be no output. You have just defined the function so you can use it later, as follows:

```python
factorial(12)
```

Example A.19. For this next example let us define a function to calculate the next element in the sequence

$$x_{n+1} = \begin{cases} 2x_n, & x_n \in [0, 0.5] \\ 2x_n - 1, & x_n \in (0.5, 1] \end{cases} \tag{A.3}$$

and then build a loop to find the first several elements of the sequence starting at any real number between 0 and 1.

```python
# Define the function
def my_seq(xn):
    if xn <= 0.5:
        return(2*xn)
    else:
        return(2*xn-1)
# Now build a sequence with this function
x = [0.125] # arbitrary starting point
for n in range(0,5): # Let us only build the first 5 terms
    x.append(my_seq(x[n]))
print(x)
```

Example A.20. A fun way to approximate the square root of two is to start with any positive real number and iterate over the sequence

$$x_{n+1} = \frac{1}{2}x_n + \frac{1}{x_n} \tag{A.4}$$

until we are within any tolerance we like of the square root of 2. Write code that defines the sequence as a function and then iterates in a while loop until we are within 10^{-8} of the square root of 2.

We import the `math` package so that we get the square root function. More about packages in the next section.

```python
from math import sqrt
def f(x):
    return(0.5*x + 1/x)
x = 1.1 # arbitrary starting point
print(f"{'Approximation':<20} | {'Exact':<20} | {'Absolute error':<20}")
print("-" * 68)
while abs(x-sqrt(2)) > 10**(-8):
    x = f(x)
    print(f"{x:<20} | {sqrt(2):<20} | {abs(x - sqrt(2)):<20}")
```

This example also shows how to format the output of a print statement so that it is nicely aligned in columns. The `:<20` means that the string will be left-aligned and padded with spaces to a total width of 20 characters. The `f` before the string allows us to use curly braces `{}` to insert variables into the string. You will see an alternative way for displaying tabular data in Example A.48.

Exercise A.9. The previous example is a special case of the Babylonian Algorithm for calculating square roots. If you want the square root of S then iterate the sequence

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right) \tag{A.5}$$

until you are within an appropriate tolerance.

Modify the code given in the previous example to give a list of approximations of the square roots of the natural numbers 2 through 20, each to within 10^{-8}. This problem will require that you build a function, write a ‘for’ loop (for the integers 2 through 20), and write a ‘while’ loop inside your ‘for’ loop to do the iterations.

A.2.7. Lambda Functions

Using `def` to define a function as in the previous subsection is really nice when you have a function that is complicated or requires some bit of code to evaluate. However, in the case of mathematical functions we have a convenient alternative: `lambda` Functions.

The basic idea of a `lambda` Function is that we just want to state what the variable is and what the rule is for evaluating the function. This is closest to the way that we write mathematical functions. For example, we can define the mathematical function $f(x) = x^2 + 3$ in two different ways.

- Using `def`:

```
def f(x):
    return(x**2+3)
```

- Using `lambda`:

```
f = lambda x: x**2+3
```

You can see that in the Lambda Function we are explicitly stating the name of the variable immediately after the word `lambda`, then we put a colon, and then the function definition. This is somewhat similar to but also annoyingly different from the mathematicians notation $f : x \mapsto x^2 + 3$.

No matter whether we use `def` or `lambda` to define the function `f`, if we want to evaluate the function at a point, say $x = 1.5$, then we can write code just like we would mathematically: $f(1.5)$

```
f(1.5) # evaluate the function at x=1.5
```

We can also define Lambda Functions of several variables. For example, if we want to define the mathematical function $f(x, y) = x^2 + xy + y^3$ we could write the code

```
f = lambda x, y: x**2 + x*y + y**3
```

If we wanted the value $f(2, 4)$ we would now write the code `f(2,4)`.

Exercise A.10. Go back to Exercise A.8 and repeat this exercise using a `lambda` function.

Exercise A.11. Go back to Exercise A.9 and repeat this exercise using a `lambda` function.

A.2.8. Packages

Python was not created as a scientific programming language. The reason Python can be used for scientific computing is that there are powerful extension packages that define additional functions that are needed for scientific calculations.

Let us start with the `math` package.

Example A.21. The code below imports the `math` package into your instance of Python and calculates the cosine of $\pi/4$.

```
import math
x = math.cos(math.pi / 4)
print(x)
```

The answer, unsurprisingly, is the decimal form of $\sqrt{2}/2$.

You might already see a potential disadvantage to Python's packages: there is now more typing involved! Let us fix this. When you import a package you could just import all of the functions so they can be used by their proper names.

Example A.22. Here we import the entire math package so we can use every one of the functions therein without having to use the `math` prefix.

```
from math import * # read this as: from math import everything
x = cos(pi / 4)
print(x)
```

The end result is exactly the same: the decimal form of $\sqrt{2}/2$, but now we had less typing to do.

Now you can freely use the functions that were imported from the math package. There is a disadvantage to this, however. What if we have two packages that import functions with the same name. For example, in the `math` package and in the `numpy` package there is a `cos()` function. In the next block of code we will import both `math` and `numpy`, but instead we will import them with shortened names so we can type things a bit faster.

Example A.23. Here we import `math` and `numpy` under aliases so we can use the shortened aliases and not mix up which functions belong to which packages.

```python
import math as ma
import numpy as np
# use the math version of the cosine function
x = ma.cos(ma.pi / 4)
# use the numpy version of the cosine function
y = np.cos(np.pi / 4)
print(x, y)
```

Both `x` and `y` in the code give the decimal approximation of $\sqrt{2}/2$. This is clearly pretty redundant in this really simple case, but you should be able to see where you might want to use this and where you might run into troubles.

Example A.24 (Contents of a package)**.** Once you have a package imported you can see what is inside of it using the `dir` command. The following block of code prints a list of all of the functions inside the `math` package.

```python
import math
print(dir(math))
```

By the way: you only need to import a package once in a session. The only reason we are repeating the `import` statement in each code block is to make it easier to come back to this material later in a new session, where you will need to import the packages again.

Of course, there will be times when you need help with a function. You can use the `help` function to view the help documentation for any function. For example, you can run the code `help(math.acos)` to get help on the arc cosine function from the `math` package.

Exercise A.12. Import the `math` package, figure out how the `log` function works, and write code to calculate the logarithm of the number 8.3 in base 10, base 2, base 16, and base e (the natural logarithm).

A.3. Numerical Python with NumPy

The base implementation of Python includes the basic programming language, the tools to write loops, check conditions, build and manipulate lists, and all of the other things that we saw in the previous section. In this section we will explore the package `numpy` that contains optimized numerical routines for doing numerical computations in scientific computing.

Example A.25. To start with, let us look at a really simple example. Say you have a list of real numbers and you want to take the sine of every element in the list. If you just try to take the sine of the list you will get an error. Try it yourself.

```python
from math import pi, sin
my_list = [0,pi/6, pi/4, pi/3, pi/2, 2*pi/3, 3*pi/4, 5*pi/6, pi]
sin(my_list)
```

You could get around this error using some of the tools from base Python, but none of them are very elegant from a programming perspective.

```python
from math import pi, sin
my_list = [0,pi/6, pi/4, pi/3, pi/2, 2*pi/3, 3*pi/4, 5*pi/6, pi]
sine_list = [sin(n) for n in my_list]
sine_list
```

```
from math import pi, sin
my_list = [0,pi/6, pi/4, pi/3, pi/2, 2*pi/3, 3*pi/4, 5*pi/6, pi]
sine_list = [ ]
for n in range(0,len(my_list)):
    sine_list.append(sin(my_list[n]))
sine_list
```

Perhaps more simply, say we wanted to square every number in a list. Just appending the code `**2` to the end of the list will fail!

```
my_list = [1,2,3,4]
my_list**2 # This will produce an error
```

If, instead, we define the list as a `numpy` array instead of a Python list then everything will work mathematically exactly the way that we intend.

```
import numpy as np
my_list = np.array([1,2,3,4])
my_list**2 # This will work as expected!
```

Exercise A.13. See if you can take the sine of a full list of numbers that are stored in a `numpy` array.

Hint: you will now see why the numpy package provides its own version of the sine function.

The package `numpy` is used in many (most) mathematical computations in numerical analysis using Python. It provides algorithms for matrix and vector arithmetic. Furthermore, it is optimized to be able to do these computations in the most efficient possible way (both in terms of memory and in terms of speed).

Typically when we import `numpy` we use `import numpy as np`. This is the standard way to name the `numpy` package. This means that we will have lots of function with the prefix "np" in order to call on the `numpy` functions. Let us first see what is inside the package

```
import numpy as np
dir(np)
```

A brief glimpse through the list reveals a huge wealth of mathematical functions that are optimized to work in the best possible way with the Python language. (We are intentionally not showing the output here since it is quite extensive, run it so you can see.)

A.3.1. Numpy Arrays, Array Operations, and Matrix Operations

In the previous section you worked with Python lists. As we pointed out, the shortcoming of Python lists is that they do not behave well when we want to apply mathematical functions to the vector as a whole. The "numpy array", `np.array`, is essentially the same as a Python list with the notable exceptions that

- In a `numpy` array every entry is a floating point number
- In a `numpy` array the memory usage is more efficient (mostly since Python is expecting data of all the same type)
- With a `numpy` array there are ready-made functions that can act directly on the array as a matrix or a vector

Let us just look at a few examples using `numpy`. What we are going to do is to define a matrix A and vectors v and w as

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad v = \begin{pmatrix} 5 \\ 6 \end{pmatrix} \quad \text{and} \quad w = v^T = \begin{pmatrix} 5 & 6 \end{pmatrix}. \tag{A.6}$$

Then we will do the following

- Get the size and shape of these arrays
- Get individual elements, rows, and columns from these arrays
- Treat these arrays as with linear algebra to
  - do element-wise multiplication
  - do matrix a vector products
  - do scalar multiplication
  - take the transpose of matrices
  - take the inverse of matrices

Example A.26 (numpy Matrices)**.** The first thing to note is that a matrix is a list of lists (each row is a list).

```
import numpy as np
A = np.array([[1,2],[3,4]])
print("The matrix A is:\n",A)
v = np.array([[5],[6]]) # this creates a column vector
print("The vector v is:\n",v)
w = np.array([[5,6]]) # this creates a row vector
print("The vector w is:\n",w)
```

Example A.27 (`.shape`)**.** The `.shape` attribute can be used to give the shape of a `numpy` array. Notice that the output is a tuple showing the size `(rows, columns)`.

```
print("The shape of the matrix A is ", A.shape)
print("The shape of the column vector v is ", v.shape)
print("The shape of the row vector w is ", w.shape)
```

Example A.28 (`.size`)**.** The `.size` attribute can be used to give the size of a `numpy` array. The size of a matrix or vector will be the total number of elements in the array. You can think of this as the product of the values in the tuple coming from the shape method.

```
print("The size of the matrix A is ", A.size)
print("The size of the column vector v is ", v.size)
print("The size of the row vector w is ", w.size)
```

Reading individual elements from a `numpy` array is the same, essentially, as reading elements from a Python list. We will use square brackets to get the row and column. Remember that the indexing all starts from 0, not 1!

Example A.29. Let us read the top left and bottom right entries of the matrix A.

```python
import numpy as np
A = np.array([[1,2],[3,4]])
print(A[0,0]) # top left
print(A[1,1]) # bottom right
```

Example A.30. Let us read the first row from that matrix A.

```python
import numpy as np
A = np.array([[1,2],[3,4]])
print(A[0,:])
```

Example A.31. Let us read the second column from the matrix A.

```python
import numpy as np
A = np.array([[1,2],[3,4]])
print(A[:,1])
```

Notice when we read the column it was displayed as a row. Be careful. Reading a row or a column from a matrix will automatically flatten it into a 1-dimensional array.

If we try to multiply either A and v or A and A we will get some funky results. Unlike in some programming languages like MATLAB, the default notion of multiplication is NOT matrix multiplication. Instead, the default is element-wise multiplication. You may be familiar with this from R.

Example A.32. If we write the code `A*A` we do NOT do matrix multiplication. Instead we do element-by-element multiplication. This is a common source of issues when dealing with matrices and Linear Algebra in Python.

```python
import numpy as np
A = np.array([[1,2],[3,4]])
print("Element-wise multiplication:\n", A * A)
print("Matrix multiplication:\n", A @ A)
```

Example A.33. If we write `A * v` Python will do element-wise multiplication across each column since v is a column vector. If we want the matrix `A` to act on `v` we write `A @ v`.

```
import numpy as np
A = np.array([[1,2],[3,4]])
v = np.array([[5],[6]])
print("Element-wise multiplication on each column:\n", A * v)
# A @ v will do proper matrix multiplication
print("Matrix A acting on vector v:\n", A @ v)
```

It is up to you to check that these products are indeed correct from the definitions of matrix multiplication from Linear Algebra.

It remains to show some of the other basic linear algebra operations: inverses, determinants, the trace, and the transpose.

Example A.34 (Transpose)**.** Taking the transpose of a matrix (swapping the rows and columns) is done with the `.T` attribute.

```
A.T # The transpose is relatively simple
```

Example A.35 (Trace)**.** The trace is done with `matrix.trace()`

```
A.trace() # The trace is pretty darn easy too
```

Oddly enough, the trace returns a matrix, not a scalar Therefore you will have to read the first entry (index `[0,0]`) from the answer to just get the trace.

Example A.36 (Determinant)**.** The determinant function is hiding under the `linalg` subpackage inside `numpy`. Therefore we need to call it as such.

```
np.linalg.det(A)
```

You notice an interesting numerical error here. You can do the determinant easily by hand and so know that it should be exactly −2. We'll discuss the source of these kinds of errors in Chapter 1.

Example A.37 (Inverse)**.** In the `linalg` subpackage there is also a function for taking the inverse of a matrix.

```
A_inv = np.linalg.inv(A)
A_inv
```

We can check that we get the identity matrix back:

```
A @ A_inv
```

Exercise A.14. Now that we can do some basic linear algebra with `numpy` it is your turn. Define the matrix B and the vector u as

$$B = \begin{pmatrix} 1 & 4 & 8 \\ 2 & 3 & -1 \\ 0 & 9 & -3 \end{pmatrix} \quad \text{and} \quad u = \begin{pmatrix} 6 \\ 3 \\ -7 \end{pmatrix}. \tag{A.7}$$

Then find

1. Bu
2. B^2 (in the traditional linear algebra sense)
3. The size and shape of B
4. $B^T u$
5. The element-by-element product of B with itself
6. The dot product of u with the first row of B

A.3.2. arange, linspace, zeros, ones, and meshgrid

There are a few built-in ways to build arrays in numpy that save a bit of time in many scientific computing settings.

Example A.38. The `np.arange` (array range) function is great for building sequences.

```
import numpy as np
x = np.arange(0,0.6,0.1)
x
```

`np.arange` builds an array of floating point numbers with the arguments `start`, `stop`, and `step`. Note that the `stop` value itself is not included in the result.

Example A.39. The `np.linspace` function builds an array of floating point numbers starting at one point, ending at the next point, and have exactly the number of points specified with equal spacing in between: `start`, `stop`, `number of points`.

```
import numpy as np
y = np.linspace(0,5,11)
y
```

In a linear space you are always guaranteed to hit the stop point exactly, but you do not have direct control over the step size.

Example A.40. The `np.zeros` function builds an array of zeros. This is handy for pre-allocating memory.

```
import numpy as np
z = np.zeros((3,5)) # create a 3x5 matrix of zeros
z
```

If you already have an array and want to create an array of zeros with the same shape, you can use `np.zeros_like(array)`.

```
import numpy as np
x = np.linspace(0,5,11)
z = np.zeros_like(x)
z
```

Similarly there are the functions `np.ones` and `np.ones_like` to build arrays of ones.

Example A.41. The `np.meshgrid` function builds two arrays that when paired make up the ordered pairs for a 2D (or higher D) mesh grid of points. This is handy for building 2D (or higher dimensional) arrays of data for multi-variable functions. Notice that the output is defined as a tuple.

```
import numpy as np
x, y = np.meshgrid(np.linspace(0,5,6), np.linspace(0,5,6))
print("x = ", x)
print("y = ", y)
```

The thing to notice with the `np.meshgrid()` function is that when you lay the two arrays on top of each other, the matching entries give every ordered pair in the domain.

If the purpose of this is not clear to you yet, don't worry. You will see it used a lot later in the module.

Exercise A.15. Now it is time to practice with some of these `numpy` functions.

a. Create a `numpy` array of the numbers 1 through 10 and square every entry in the list without using a loop.

b. Create a 10×10 identity matrix and change the top right corner to a 5. Hint: `np.identity()`

c. Find the matrix-vector product of the answer to part (b) and the answer to part (a).

d. Change the bottom row of your matrix from part (b) to all 3's, then change the third column to all 7's, and then find the 5^{th} power of this matrix.

A.4. Plotting with Matplotlib

A key part of scientific computing is plotting your results or your data. The tool in Python best-suited to this task is the package `matplotlib`. As with all of the other packages in Python, it is best to learn just the basics first and then to dig deeper later. One advantage to using `matplotlib` in Python is that it is modelled off of MATLAB's plotting tools. People coming from a MATLAB background should feel pretty comfortable here, but there are some differences to be aware of.

A.4.1. Basics with `plt.plot()`

We are going to start right away with an example. In this example, however, we will walk through each of the code chunks one-by-one so that we understand how to set up a proper plot.

Below we will mention some tricks for getting the plots to render that only apply to Jupyter Notebooks. If you are using Google Colab then you may not need some of these little tricks.

Example A.42 (Plotting with matplotlib)**.** In the first example we want to simply plot the sine function on the domain $x \in [0, 2\pi]$, colour it green, put a grid on it, and give a meaningful legend and axis labels. To do so we first need to take care of a couple of housekeeping items.

- Import `numpy` so we can take advantage of some good numerical routines.
- Import `matplotlib`'s `pyplot` module. The standard way to pull it in is with the nickname `plt` (just like with `numpy` when we import it as `np`).

In Jupyter Notebooks the plots will not show up unless you tell the notebook to put them "inline." Usually we will use the following command to get the plots to show up. You do not need to do this in Google Colab. The percent sign is called a *magic* command in Jupyter Notebooks. This is not a Python command, but it is a command for controlling the Jupyter IDE specifically.

```
%matplotlib inline
```

Now we will build a `numpy` array of x values (using the `np.linspace` function) and a `numpy` array of y values from the sine function.

- Next, build the plot with `plt.plot()`. The syntax is: `plt.plot(x, y, 'color', ...)` where you have several options that you can pass (more on that later).

- We send the plot label directly to the plot function. This is optional and we could set the legend up separately if we like.
- Then we will add the grid with `plt.grid()`
- Then we will add the legend to the plot
- Finally we will add the axis labels
- We end the plotting code with `plt.show()` to tell Python to finally show the plot. This line of code tells Python that you are done building that plot.

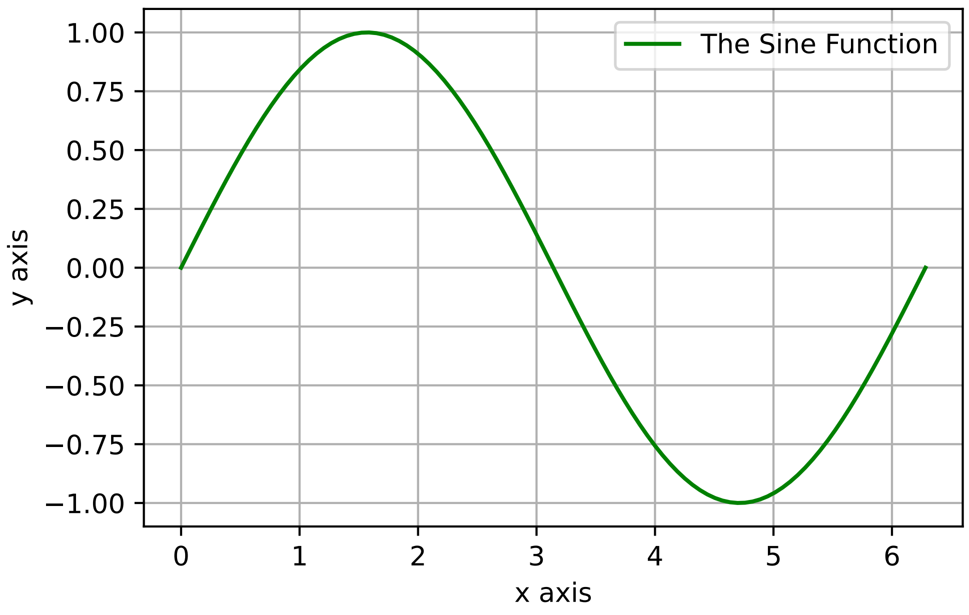


Figure A.1.: The sine function

Example A.43. Now let us do a second example, but this time we want to show four different plots on top of each other. When you start a figure, `matplotlib` is expecting all of those plots to be layered on top of each other. (Note:For MATLAB users, this means that you do not need the `hold on` command since it is automatically "on.")

In this example we will plot

$$y_0 = \sin(2\pi x) \quad y_1 = \cos(2\pi x) \quad y_2 = y_0 + y_1 \quad \text{and} \quad y_3 = y_0 - y_1 \tag{A.8}$$

on the domain $x \in [0,1]$ with 100 equally spaced points. we will give each of the plots a different line style, built a legend, put a grid on the plot, and give axis labels.

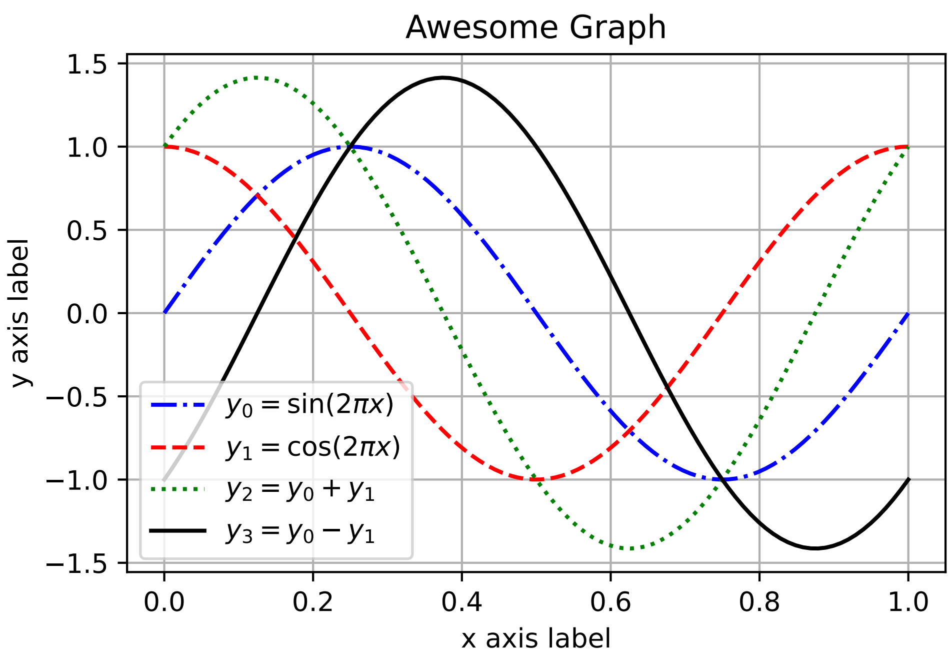


Figure A.2.: Plots of the sine, cosine, and sums and differences.

Notice the `r` in front of the strings defining the legend. This prevents the backslash that is used a lot in LaTeX to be interpreted as an escape character. These strings are referred to as raw strings.

The legend was placed automatically at the lower left of the plot. There are ways to control the placement of the legend if you wish, but for now just let Python and `matplotlib` have control over the placement.

Example A.44. Now let us create the same plot with slightly different code. The `plot` function can take several (x, y) pairs in the same line of code. This can really shrink the amount of coding that you have to do when plotting several functions on top of each other.

Exercise A.16. Plot the functions $f(x) = x^2$, $g(x) = x^3$, and $h(x) = x^4$ on the same axes. Use the domain $x \in [0, 1]$. Put a grid, a legend, a title, and appropriate labels on the axes.

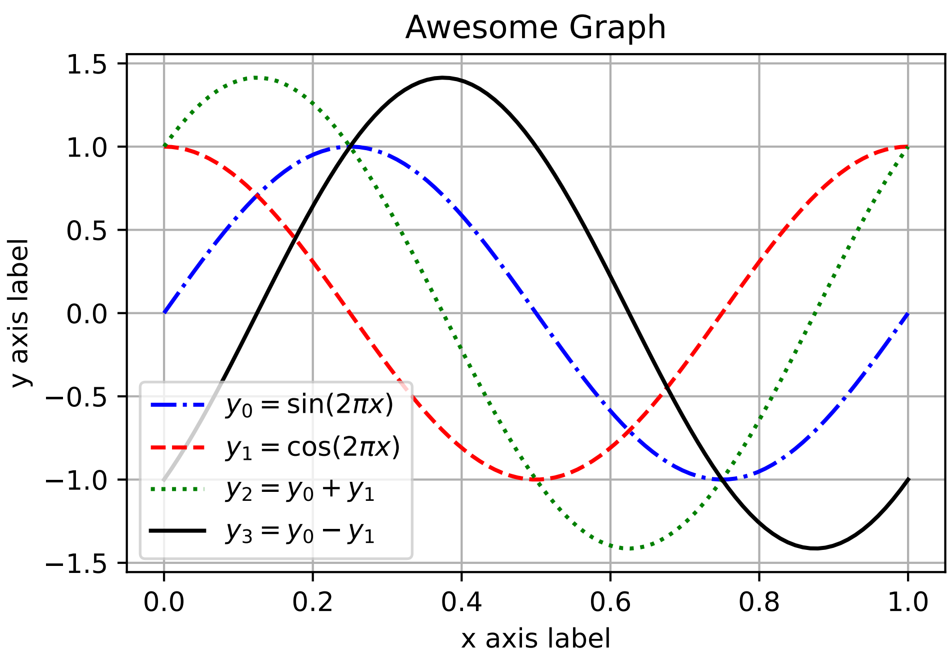


Figure A.3.: A second plot of the sine, cosine, and sums and differences.

A.4.2. Subplots

It is often very handy to place plots side-by-side or as some array of plots. The `subplots` command allows us that control. The main idea is that we are setting up a matrix of blank plots and then populating the axes with the plots that we want.

Example A.45. Let us repeat the previous exercise, but this time we will put each of the plots in its own subplot. There are a few extra coding quirks that come along with building subplots so we will highlight each block of code separately.

- First we set up the plot area with `plt.subplots()`. The first two inputs to the `subplots` command are the number of rows and the number of columns in your plot array. For the first example we will do 2 rows of plots with 2 columns – so there are four plots total.
- Then we build each plot individually telling `matplotlib` which axes to use for each of the things in the plots.
- Notice the small differences in how we set the titles and labels

- In this example we are setting the y-axis to the interval $[-2, 2]$ for consistency across all of the plots.

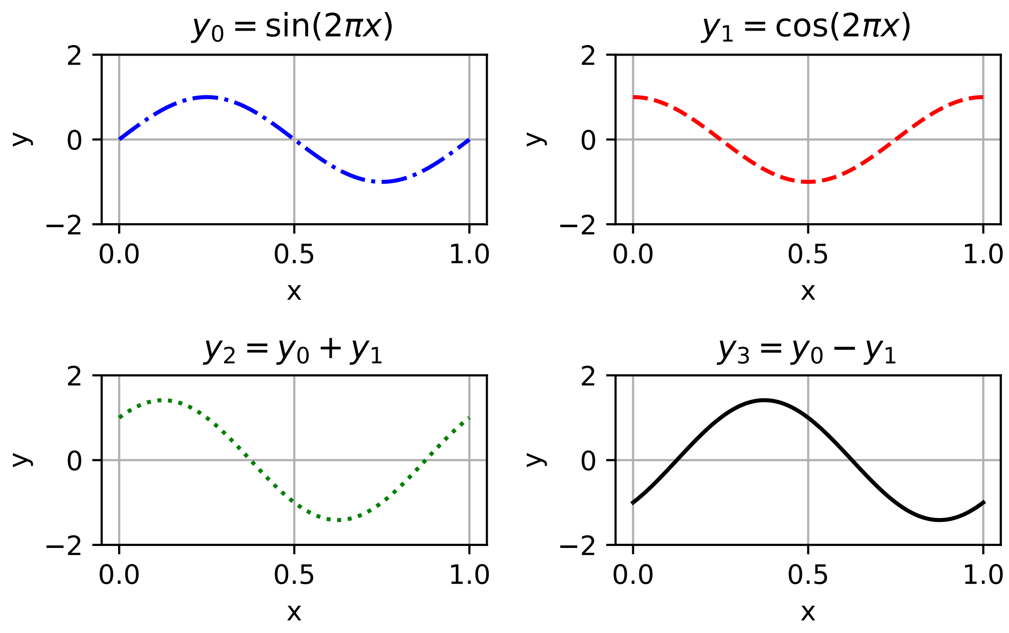


Figure A.4.: An example of subplots

The `fig.tight_layout()` command makes the plot labels a bit more readable in this instance (again, something you can play with).

Exercise A.17. Put the functions $f(x) = x^2$, $g(x) = x^3$ and $h(x) = x^4$ in a subplot environment with 1 row and 3 columns of plots. Use the unit interval as the domain and range for all three plot. Make sure that each plot has a grid, appropriate labels, an appropriate title, and the overall figure has a title.

A.4.3. Logarithmic Scaling with `semilogy`, `semilogx`, and `loglog`

It is occasionally useful to scale an axis logarithmically. This arises most often when we are examining an exponential function, or some other function, that is close to zero for much of the domain. Scaling logarithmically allows us to see how small the function is getting in orders of magnitude instead of as a raw real number. we will use this often in numerical methods.

Example A.46. In this example we will plot the function $y = 10^{-0.01x}$ on a regular (linear) scale and on a logarithmic scale on the y axis. We use the interval $[0, 500]$ on the x axis.

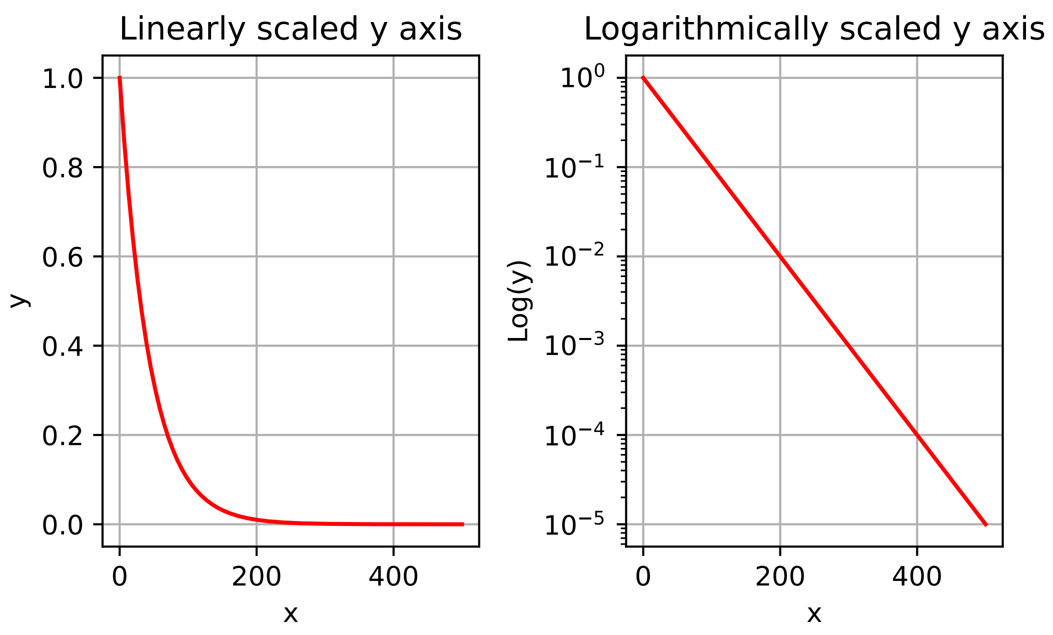


Figure A.5.: An example of using logarithmic scaling.

It should be noted that the same result can be achieved using the `yscale` command along with the `plot` command instead of using the `semilogy` command. So you could replace

```
axis[1].semilogy(x,y, 'r')
```

by

```
axis[1].plot(x,y, 'r')
axis[1].set_yscale("log")
```

to produce identical results.

Exercise A.18. Plot the function $f(x) = x^3$ for $x \in [0, 1]$ on linearly scaled axes, logarithmic axis in the y direction, logarithmically scaled axes in the x direction, and a log-log plot with logarithmic scaling on both axes. Use `subplots` to put your plots side-by-side. Give appropriate labels, titles, etc.

A.5. Dataframes with Pandas

The Pandas package provides Python with the ability to work with tables of data similar to what R provides via its dataframes. As we will not work much with data in this module, we do not need to dive deep into the Pandas package. In some of the optional exercises you will load in data from files using `pd.read_csv()`.

Example A.47.

	Time (sec)	Speed (ft/sec)
0	0	34
1	10	32
2	20	29
3	30	33
4	40	37
5	50	40
6	60	41
7	70	36
8	80	38
9	90	39

Example A.48. Pandas can also be useful to us for collecting computational results into tables for easier display. In this example we will build a table of the first 10 natural numbers and their squares and cubes. We then display the table.

	n	n^2	n^3
0	1	1	1
1	2	4	8
2	3	9	27
3	4	16	64
4	5	25	125
5	6	36	216
6	7	49	343
7	8	64	512
8	9	81	729
9	10	100	1000

This provides an alternative to how we created a tabular display with the `print()` function in Example A.20.

A.6. Problems

These problem exercises here are meant for you to practice and improve your coding skills. Please refrain from relying too much on Gemini or any other AI for solving these exercises. The point is to struggle through the code, get it wrong many times, debug, and then to eventually have working code. So I recommend switching off the AI features in Google Colab for the purpose of these exercises.

You do not need to do all the exercises. Do only as many as you have time for and you feel is useful. It might be a good idea to split the exercises up among your group members and then share the code with each other.

Exercise A.19. (This problem is modified from ("Project Euler" n.d.))
If we list all of the numbers below 10 that are multiples of 3 or 5 we get 3, 5, 6, and 9. The sum of these multiples is 23. Write code to find the sum of all the multiples of 3 or 5 below 1000. Your code needs to run error free and output only the sum. There are of course many ways you could approach this exercise. Compare your approach to that of others in your group.

Exercise A.20. (This problem is modified from ("Project Euler" n.d.))
Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... \tag{A.9}$$

By considering the terms in the Fibonacci sequence whose values do not exceed four million, write code to find the sum of the even-valued terms. Your code needs to run error free and output only the sum.

Exercise A.21. Write computer code that will draw random numbers from the unit interval $[0, 1]$, distributed uniformly (using Python's `np.random.rand()`), until the sum of the numbers that you draw is greater than 1. Keep track of how many numbers you draw. Then write a loop that does this process many many times. On average, how many numbers do you have to draw until your sum is larger than 1?

Hint #1: Use the `np.random.rand()`command to draw a single number from a uniform distribution with bounds $(0, 1)$.

Hint #2: You should do this more than 1,000,000 times to get a good average … and the number that you get should be familiar!

Exercise A.22. (This problem is modified from ("Project Euler" n.d.))
The sum of the squares of the first ten natural numbers is,

$$1^2 + 2^2 + \cdots + 10^2 = 385 \tag{A.10}$$

The square of the sum of the first ten natural numbers is,

$$(1 + 2 + \cdots + 10)^2 = 55^2 = 3025 \tag{A.11}$$

Hence the difference between the square of the sum of the first ten natural numbers and the sum of the squares is $3025 - 385 = 2640$.

Write code to find the difference between the square of the sum of the first one hundred natural numbers and the sum of the squares. Your code needs to run error free and output only the difference.

Exercise A.23. (This problem is modified from ("Project Euler" n.d.))
The prime factors of 13195 are $5, 7, 13$ and 29. Write code to find the largest prime factor of the number 600851475143? Your code needs to run error free and output only the largest prime factor.

Exercise A.24. (This problem is modified from ("Project Euler" n.d.))
The number 2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any remainder. Write code to find the smallest positive number that is evenly divisible by all of the numbers from 1 to 20?

Hint: You will likely want to use modular division for this problem.

Exercise A.25. The following iterative sequence is defined for the set of positive integers:

$$\begin{aligned} n &\to \frac{n}{2} \quad (n \text{ is even}) \\ n &\to 3n+1 \quad (n \text{ is odd}) \end{aligned} \tag{A.12}$$

Using the rule above and starting with 13, we generate the following sequence:

$$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1 \tag{A.13}$$

It can be seen that this sequence (starting at 13 and finishing at 1) contains 10 terms. Although it has not been proved yet (Collatz Problem), it is thought that all starting numbers finish at 1. This has been verified on computers for massively large starting numbers, but this does not constitute a proof that it will work this way for *all* starting numbers.

Write code to determine which starting number, under one million, produces the longest chain. NOTE: Once the chain starts, the terms are allowed to go above one million.

Footnotes

B. Exam solutions

In this appendix you find example solutions to the exam-style questions.

B.1. Numbers

(a) Machine Precision

Machine precision is the gap between the number 1 and the next larger floating point number. With 4 bits for the mantissa, the smallest number greater than 1 that can be represented is 1.0001_2. So machine precision is $\epsilon = 0.0001_2 = 1/16$. The machine precision is the upper bound for the relative rounding error.

(b) Conversion of 13.5

- Sign: Positive (+13.5), so $s = 0$.
- Convert 13.5 to binary:
  - $13_{10} = 1101_2$.
  - $0.5_{10} = 0.1_2$.
  - Result: 1101.1_2.
- Normalize: 1.1011×2^3.
- Exponent ($E = 3$):
  - Stored exponent $e = E + 3 = 3 + 3 = 6$.
  - $6_{10} = 110_2$.
- Mantissa (m): Drop the leading $1 \rightarrow$ `1011`.
- **Final Bit Pattern:** `0 110 1011`.

(c) Addition: 13.5 + 0.25

- **Convert second number (0.25):**
  - $0.25 = 1/4 = 1.0 \times 2^{-2}$.
- **Align Exponents:**
  - $x_1 = 1.1011 \times 2^3$
  - $x_2 = 1.0000 \times 2^{-2}$

- Shift x_2 to match exponent 3: Shift right by $3 - (-2) = 5$ positions.
- $x_2 \to 0.00001 \times 2^3$.

- **Add Significands:**

```
  1.1011      (13.5)
+ 0.00001     (0.25 aligned)
-----------
  1.10111
```

- **Rounding:**
  - The result 1.10111_2 has 5 fractional bits, but we can only store 4.
  - It lies exactly halfway between 1.1011 (13.5) and 1.1100 (14.0).
  - Tie-breaking rule: "Ties to Even".
  - The Least Significant Bit (4th bit) is `1` (odd). To make it even, we round **up** (add 1 to the LSB).
  - $1.1011 + 0.0001 = 1.1100$.

- **Final Result:**
  - Significand: 1.1100
  - Value: $1.1100_2 \times 2^3 = 1110.0_2 = \mathbf{14.0}$.

- **Error:**
  - Exact sum: 13.75.
  - Stored sum: 14.0.
  - Absolute Error: $|13.75 - 14.0| = \mathbf{0.25}$.

(d) Loss of Significance

- **Error Type:** Catastrophic Cancellation (or Loss of Significant Digits).
- **Explanation:** When x is very large (10^8), $x^2+1 \approx x^2$, so $\sqrt{x^2+1} \approx x$. Subtracting two extremely close numbers causes the cancellation of the leading digits, leaving only the random "noise" from the least significant bits.
- **Improved (Stable) Formula:** Multiply by the conjugate:

$$f(x) = \frac{(\sqrt{x^2+1} - x)(\sqrt{x^2+1} + x)}{\sqrt{x^2+1} + x} = \frac{1}{\sqrt{x^2+1} + x}.$$

C. Linear Algebra

You cannot learn too much linear algebra.
– Every mathematician

C.1. Intro to Numerical Linear Algebra

The preceding comment says it all – linear algebra is the most important of all of the mathematical tools that you can learn as a practitioner of the mathematical sciences. The theorems, proofs, conjectures, and big ideas in almost every other mathematical field find their roots in linear algebra. Numerical Linear Algebra is the study of algorithms for solving problems in linear algebra. This subject has a somewhat different flavour than the numerical analysis we are studying in the main text and hence we present it as an optional appendix.

Our goal in this appendix is to explore numerical algorithms for the primary questions of linear algebra:

- solving systems of equations,
- finding eigenvalue-eigenvector pairs for a matrix.

Take careful note that in our current digital age, numerical linear algebra and its fast algorithms are behind the scenes for wide varieties of computing applications. Applications of numerical linear algebra include:

- building neural networks and AI algorithms,
- determining the most important web page in a Google search,
- modelling realistic 3D environments in video games,
- digital image processing,
- and many many more.

What's more, researchers have found provably optimal ways to perform most of the typical tasks of linear algebra so most scientific software works very well and very quickly with linear algebra.

C.2. Notation

Throughout this chapter we will use the following notation conventions.

- A bold mathematical symbol such as x or u will represent a vector.
- If u is a vector then u_j will be the j^{th} entry of the vector.
- Vectors will typically be written vertically with parenthesis as delimiters such as

$$u = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}. \tag{C.1}$$

- Two bold symbols separated by a centred dot such as $u \cdot v$ will represent the dot product of two vectors.
- A capital mathematical symbol such as A or X will represent a matrix
- If A is a matrix then A_{ij} will be the element in the i^{th} row and j^{th} column of the matrix.
- A matrix will typically be written with parenthesis as delimiters such as

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & \pi \end{pmatrix}. \tag{C.2}$$

- The juxtaposition of a capital symbol and a bold symbol such as Ax will represent matrix-vector multiplication.
- A lower case or Greek mathematical symbol such as x, c, or λ will represent a scalar.
- The scalar field of real numbers is given as $\mathbb{R}$ and the scalar field of complex numbers is given as $\mathbb{C}$.
- The symbol $\mathbb{R}^n$ represents the collection of all n-dimensional vectors where the elements are drawn from the real numbers.
- The symbol $\mathbb{C}^n$ represents the collection of all n-dimensional vectors where the elements are drawn from the complex numbers.

It is an important part of learning to read and write linear algebra to give special attention to the symbolic language so you can communicate your work easily and efficiently.

C.3. Vectors and Matrices in Python

We first need to understand how Python's `numpy` library builds and stores vectors and matrices. The following exercises will give you some experience building and working with these data structures and will point out some common pitfalls that mathematicians fall into when using Python for linear algebra.

Example C.1 (numpy Arrays)**.** In Python you can build a list using square brackets such as `[1,2,3]`. This is called a "Python list" and is NOT a vector in the way that we think about it mathematically. It is simply an ordered collection of objects. To build mathematical vectors in Python we need to use `numpy` arrays with `np.array()`. For example, the vector

$$u = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \tag{C.3}$$

would be built with the following code.

```python
import numpy as np
u = np.array([1,2,3])
print(u)
```

Notice that Python defines the vector `u` as a matrix with only one dimension. You can see that in the following code.

```python
import numpy as np
u = np.array([1,2,3])
print("The length of the u vector is \n",len(u))
print("The shape of the u vector is \n",u.shape)
```

Example C.2 (numpy Matrices)**.** In `numpy`, a matrix is validly built as a list of lists. For example, the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{C.4}$$

is defined using `np.array()` where each row is an individual list, and the matrix is a collection of these lists.

```
import numpy as np
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(A)
```

Moreover, we can extract the shape, the number of rows, and the number of columns of A using the `A.shape` command. To be a bit more clear on this one we will use the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \tag{C.5}$$

```
import numpy as np
A = np.array([[1,2,3],[4,5,6]])
print("The shape of the A matrix is \n",A.shape)
print("Number of rows in A is \n",A.shape[0])
print("Number of columns in A is \n",A.shape[1])
```

Example C.3 (Row and Column Vectors in Python)**.** You can more specifically build row or column vectors in Python using the `np.array()` command and then only specifying one row or column. But take careful note that `numpy` treats a 1D array (like `np.array([1,2,3])`) differently from a 2D array (like `np.array([[1,2,3]])`). For example, if you want the vectors

$$u = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad \text{and} \quad v = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix} \tag{C.6}$$

then we would use the following Python code.

```
import numpy as np
u = np.array([[1],[2],[3]])
print("The column vector u is \n",u)
v = np.array([[1,2,3]])
print("The row vector v is \n",v)
```

Alternatively, if you want to define a column vector you can define a row vector (since there are far fewer brackets to keep track of) and then transpose the matrix to turn it into a column. Note that the `.transpose()` method (or `.T`) only swaps dimensions; it won't turn a 1D array into a 2D column vector unless it was already 2D.

```python
import numpy as np
u = np.array([[1,2,3]])
u = u.transpose()
print("The column vector u is \n",u)
```

Example C.4 (Matrix Indexing)**.** Python indexes all arrays, vectors, lists, and matrices starting from index 0. Let us get used to this fact.

Consider the matrix A defined in the previous problem. Mathematically we know that the entry in row 1 column 1 is a 1, the entry in row 1 column 2 is a 2, and so on. However, with Python we need to shift the way that we enumerate the rows and columns of a matrix. Hence we would say that the entry in row 0 column 0 is a 1, the entry in row 0 column 1 is a 2, and so on.

Mathematically we can view all Python matrices as follows. If A is an $n \times n$ matrix then

$$A = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & \cdots & A_{0,n-1} \\ A_{1,0} & A_{1,1} & A_{1,2} & \cdots & A_{1,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n-1,0} & A_{n-1,1} & A_{n-1,2} & \cdots & A_{n-1,n-1} \end{pmatrix} \tag{C.7}$$

Similarly, we can view all vectors as follows. If u is an $n \times 1$ vector then

$$u = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \end{pmatrix} \tag{C.8}$$

The following code should help to illustrate this indexing convention.

```python
import numpy as np
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
print("Entry in row 0 column 0 is",A[0,0])
print("Entry in row 0 column 1 is",A[0,1])
print("Entry in the bottom right corner",A[2,2])
```

Exercise C.1. Build your own matrix in Python and practice choosing individual entries from the matrix.

Example C.5 (Matrix Slicing)**.** The last thing that we need to be familiar with is *slicing* a matrix. The term “slicing” generally refers to pulling out individual rows, columns, entries, or blocks from a list, array, or matrix in Python. Examine the code below to see how to slice parts out of a `numpy` matrix.

```python
import numpy as np
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(A)
print("The first column of A is \n",A[:,0])
print("The second row of A is \n",A[1,:])
print("The top left 2x2 sub matrix of A is \n",A[:-1,:-1])
print("The bottom right 2x2 sub matrix of A is \n",A[1:,1:])
u = np.array([1,2,3,4,5,6])
print("The first 3 entries of the vector u are \n",u[:3])
print("The last entry of the vector u is \n",u[-1])
print("The last two entries of the vector u are \n",u[-2:])
```

Exercise C.2. Define the matrix A and the vector u in Python. Then perform all of the tasks below.

$$A = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \\ -3 & -2 & -1 & 0 \end{pmatrix} \quad \text{and} \quad u = \begin{pmatrix} 10 \\ 20 \\ 30 \end{pmatrix} \tag{C.9}$$

1. Print the matrix A, the vector u, the shape of A, and the shape of u.
2. Print the first column of A.
3. Print the first two rows of A.
4. Print the first two entries of u.
5. Print the last two entries of u.
6. Print the bottom left 2×2 submatrix of A.
7. Print the middle two elements of the middle row of A.

C.4. Matrix and Vector Operations

Now let us start doing some numerical linear algebra. We start our discussion with the basics: the dot product and matrix multiplication. The numerical routines in Python's `numpy` packages are designed to do these tasks in very efficient ways but it is a good coding exercise to build your own dot product and matrix multiplication routines just to further cement the way that Python deals with these data structures and to remind you of the mathematical algorithms. What you will find in numerical linear algebra is that the indexing and the housekeeping in the codes is the hardest part. So why do not we start "easy."

C.4.1. The Dot Product

Exercise C.3. This problem is meant to jog your memory about dot products, how to compute them, and what you might use them for. If your linear algebra is a bit rusty then read ahead a bit and then come back to this problem.

Consider two vectors u and v defined as

$$u = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \text{and} \quad v = \begin{pmatrix} 3 \\ 4 \end{pmatrix}. \tag{C.10}$$

1. Draw a picture showing both u and v.
2. What is $u \cdot v$?
3. What is $\|u\|$?
4. What is $\|v\|$?
5. What is the angle between u and v?
6. Give two reasons why we know that u is not perpendicular to v.
7. What is the scalar projection of u onto v? Draw this scalar projections on your picture from part (1).
8. What is the scalar projection of v onto u? Draw this scalar projections on your picture from part (1).

Now let us get the formal definitions of the dot product on the table.

Definition C.1 (Dot product)**.** The **dot product** of two vectors $u, v \in \mathbb{R}^n$ is

$$u \cdot v = \sum_{j=1}^{n} u_j v_j. \tag{C.11}$$

Without summation notation the dot product of two vectors is ,

$$u \cdot v = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n. \tag{C.12}$$

You may also recall that the dot product of two vectors is given geometrically as

$$u \cdot v = \|u\| \|v\| \cos\theta \tag{C.13}$$

where $\|u\|$ and $\|v\|$ are the magnitudes (or lengths) of u and v respectively, and θ is the angle between the two vectors. In physical applications the dot product is often used to find the angle between two vectors (e.g. between two forces). Hence, the last form of the dot product is often rewritten as

$$\theta = \cos^{-1}\left(\frac{u \cdot v}{\|u\|\|v\|}\right). \tag{C.14}$$

Definition C.2 (Magnitude of a Vector)**.** The **magnitude** of a vector $u \in \mathbb{R}^n$ is defined as [1]

$$\|u\| = \sqrt{u \cdot u}. \tag{C.15}$$

Exercise C.4. Write a Python function that accepts two vectors (defined as `numpy` arrays) and returns the dot product. Write this code without the use any loops.

```python
import numpy as np
def myDotProduct(u,v):
    return # the dot product formula uses a product inside a sum.
```

---

[1]You should also note that $\|u\| = \sqrt{u \cdot u}$ is not the only definition of distance. More generally, if you let $\langle u, v\rangle$ be an inner product for u and v in some vector space $\mathcal{V}$ then $\|u\| = \sqrt{\langle u, u\rangle}$. In most cases in this text we will be using the dot product as our preferred inner product so we will not have to worry much about this particular natural extension of the definition of the length of a vector.

Exercise C.5. Test your `myDotProduct()` function on several dot products to make sure that it works. Example code to find the dot product between

$$u = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad \text{and} \quad v = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \tag{C.16}$$

is given below. Test your code on other vectors. Then implement an error catch into your code to catch the case where the two input vectors are not the same size. You will want to use the `len()` command to find the length of the vectors.

```
u = np.array([1,2,3])
v = np.array([4,5,6])
myDotProduct(u,v)
```

Exercise C.6. Try sending Python lists instead of `numpy` arrays into your `myDotProduct` function. What happens? Why does it happen? What is the cautionary tale here? Modify your `myDotProduct()` function one more time so that it starts by converting the input vectors into `numpy` arrays.

```
u = [1,2,3]
v = [4,5,6]
myDotProduct(u,v)
```

Exercise C.7. The `numpy` library in Python has a built-in command for doing the dot product: `np.dot()`. Test the `np.dot()` command and be sure that it does the same thing as your `myDotProduct()` function.

C.4.2. Matrix Multiplication

Next we will blow the dust off of your matrix multiplication skills.

Exercise C.8. Verify that the product of A and B is indeed what we show below. Work out all of the details by hand.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \qquad B = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} \tag{C.17}$$

$$AB = \begin{pmatrix} 27 & 30 & 33 \\ 61 & 68 & 75 \\ 95 & 106 & 117 \end{pmatrix} \tag{C.18}$$

Now that you have practised the algorithm for matrix multiplication we can formalize the definition and then turn the algorithm into a Python function.

Definition C.3 (Matrix Multiplication)**.** If A and B are matrices with $A \in \mathbb{R}^{n\times p}$ and $B \in \mathbb{R}^{p\times m}$ then the product AB is defined as

$$(AB)_{ij} = \sum_{k=1}^{p} A_{ik}B_{kj}. \tag{C.19}$$

A moment's reflection reveals that each entry in the matrix product is actually a dot product,

$$(\text{Entry in row } i \text{ column } j \text{ of } AB) = (\text{Row } i \text{ of matrix } A) \cdot (\text{Column } j \text{ of matrix } B)\,. \tag{C.20}$$

Exercise C.9. The definition of matrix multiplication above contains the cryptic phrase *a moment's reflection reveals that each entry in the matrix product is actually a dot product.* Let us go back to the matrices A and B defined in Exercise C.8 above and re-evaluate the matrix multiplication algorithm to make sure that you see each entry as the end result of a dot product.

We want to find the product of matrices A and B using dot products.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \qquad B = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} \tag{C.21}$$

1. Why will the product AB clear be a 3×3 matrix?
2. When we do matrix multiplication we take the product of a row from the first matrix times a column from the second matrix ... at least that's how many people think of it when they perform the operation by hand.

   a. The rows of A can be written as the vectors

$$a_0 = \begin{pmatrix} 1 & 2 \end{pmatrix} \tag{C.22}$$

$$a_1 = \begin{pmatrix} \_\_\_\_\_\_ & \_\_\_\_\_\_ \end{pmatrix} \tag{C.23}$$

$$a_2 = \begin{pmatrix} \_\_\_\_\_\_ & \_\_\_\_\_\_ \end{pmatrix} \tag{C.24}$$

   b. The columns of B can be written as the vectors

$$b_0 = \begin{pmatrix} 7 \\ 10 \end{pmatrix} \tag{C.25}$$

$$b_1 = \begin{pmatrix} \_\_\_\_\_\_ \\ \_\_\_\_\_\_ \end{pmatrix} \tag{C.26}$$

$$b_2 = \begin{pmatrix} \_\_\_\_\_\_ \\ \_\_\_\_\_\_ \end{pmatrix} \tag{C.27}$$

3. Now let us write each entry in the product AB as a dot product.

$$AB = \begin{pmatrix} a_0 \cdot b_0 & \_\_\_ \cdot \_\_\_ & \_\_\_ \cdot \_\_\_ \\ \_\_\_ \cdot \_\_\_ & \_\_\_ \cdot \_\_\_ & \_\_\_ \cdot \_\_\_ \\ \_\_\_ \cdot \_\_\_ & \_\_\_ \cdot \_\_\_ & \_\_\_ \cdot \_\_\_ \end{pmatrix} \tag{C.28}$$

4. Verify that you get

$$AB = \begin{pmatrix} 27 & 30 & 33 \\ 61 & 68 & 75 \\ 95 & 106 & 117 \end{pmatrix} \tag{C.29}$$

   when you perform all of the dot products from part (3).

Exercise C.10. The observation that matrix multiplication is just a bunch of dot products is what makes the code for doing matrix multiplication very fast and very streamlined. We want to write a Python function that accepts two `numpy` matrices and returns the product of the two matrices. Inside the code we will leverage the `np.dot()` command to do the appropriate dot products.

Partial code is given below. Fill in all of the details and give ample comments showing what each line does.

```
import numpy as np
def myMatrixMult(A,B):
    # Get the shapes of the matrices A and B.
    # Then write an if statement that catches size mismatches
    # in the matrices.  Next build a zeros matrix that is the
    # correct size for the product of A and B.
    AB = ???
    # AB is a zeros matix that will be filled with the values
    # from the product
    #
    # Next we do a double for-loop that loops through all of
    # the indices of the product
    for i in range(n): # loop over the rows of AB
        for j in range(m): # loop over the columns of AB
            # use the np.dot() command to take the dot product
            AB[i,j] = ???
    return AB
```

Use the following test code to determine if you actually get the correct matrix product out of your code.

~~~
``` python
A = np.array([[1,2],[3,4],[5,6]])
B = np.array([[7,8,9],[10,11,12]])
AB = myMatrixMult(A,B)
print(AB)
~~~

Exercise C.11. Try your `myMatrixMult()` function on several other matrix multiplication problems.

Exercise C.12. Build in an error catch so that your `myMatrixMult()` function catches when the input matrices do not have compatible sizes for multiplication. Write your code so that it returns an appropriate error message in this special case.

Now that you have been through the exercise of building a matrix multiplication function we will admit that using it inside larger coding problems would be a bit cumbersome (and perhaps annoying). It would be nice to just type `@` and have Python just *know* that you mean to do matrix multiplication. This is where `numpy` arrays come in quite handy.

Exercise C.13 (Matrix Multiplication with Python)**.** Python will handle matrix multiplication easily so long as the matrices are defined as `numpy` arrays. For example, with the matrices A and B from above if you can just type `A @ B` (or `np.dot(A, B)`) in Python and you will get the correct result. Pretty nice!! Let us take another moment to notice, though, that regular Python lists do not behave in the same way. Can you guess what happens if you run the following Python code? (Note: `*` does strictly element-wise multiplication for numpy arrays, which we will see in a moment).

```
A = [[1,2],[3,4],[5,6]] # a Python list of lists
B = [[7,8,9],[10,11,12]] # a Python list of lists
A @ B
```

Example C.6 (Element-by-Element Multiplication)**.** Sometimes it is convenient to do naive multiplication of matrices when you code. That is, if you have two matrices that are the same size, "naive multiplication" would just line up the matrices on top of each other and multiply the corresponding entries.[2] In Python you can do this with the `*` operator (or `np.multiply()`). The code below demonstrates this tool with the matrices

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}. \tag{C.30}$$

[2]You might have thought that *naive multiplication* was a much more natural way to do matrix multiplication when you first saw it. Hopefully now you see the power in the definition of matrix multiplication that we actually use. If not, then I give you this moment to ponder that (a) matrix multiplication is just a bunch of dot products, and (b) dot products can be seen as projections. Hence, matrix multiplication is really just a projection of the rows of A onto the columns of B. This has much more rich geometric flavour than *naive multiplication.*

(Note that the product AB does not make sense under the mathematical definition of matrix multiplication, but it does make sense in terms of element-by-element ("naive") multiplication.)

```python
import numpy as np
A = np.array([[1,2],[3,4],[5,6]])
B = np.array([[7,8],[9,10],[11,12]])
print(A * B)
```

The key takeaways for doing matrix multiplication in Python are as follows:

- If you are doing linear algebra in Python then you should define vectors and matrices with `np.array()`.
- If your matrices are defined with `np.array()` then `@` (or `np.dot()`) does regular matrix multiplication and `*` does element-by-element multiplication.

C.5. The LU Factorization

One of the many classic problems of linear algebra is to solve the linear system $Ax = b$ where A is a matrix of coefficients and b is a vector of right-hand sides. You likely recall your go-to technique for solving systems was row reduction (or Gaussian Elimination). Furthermore, you likely rarely actually did row reduction by hand, and instead you relied on a computer to do most of the computations for you. Just what was the computer doing, exactly? Do you think that it was actually following the same algorithm that you did by hand?

C.5.1. A Recap of Row Reduction

Let us blow the dust off your row reduction skills before we look at something better.

Exercise C.14. Solve the following system of equations by hand.

$$\begin{aligned} x_0 + 2x_1 + 3x_2 &= 1 \\ 4x_0 + 5x_1 + 6x_2 &= 0 \\ 7x_0 + 8x_1 &= 2 \end{aligned} \tag{C.31}$$

Note that the system of equations can also be written in the matrix form

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} \tag{C.32}$$

If you need a nudge to get started then jump ahead to the next problem.

Exercise C.15. We want to solve the system of equations

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} \tag{C.33}$$

Row Reduction Process:

Note: Throughout this discussion we use Python-type indexing so the rows and columns are enumerated starting at 0. That is to say, we will talk about row 0, row 1, and row 2 of a matrix instead of rows 1, 2, and 3.

1. Augment the coefficient matrix and the vector on the right-hand side to get

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 0 & 2 \end{array} \right) \tag{C.34}$$

2. The goal of row reduction is to perform elementary row operations until our augmented matrix gets to (or at least gets as close as possible to)

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & \star \\ 0 & 1 & 0 & \star \\ 0 & 0 & 1 & \star \end{array} \right) \tag{C.35}$$

The allowed elementary row operations are:

a. We are allowed to scale any row.

b. We can add two rows.

   c. We can interchange two rows.

3. We are going to start with column 0. We already have the "1" in the top left corner so we can use it to eliminate all of the other values in the first column of the matrix.

   a. For example, if we multiply the 0^{th} row by -4 and add it to the first row we get

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & -3 & -6 & -4 \\ 7 & 8 & 0 & 2 \end{array}\right). \tag{C.36}$$

   b. Multiply row 0 by a scalar and add it to row 2. Your end result should be

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & -3 & -6 & -4 \\ 0 & -6 & -21 & -5 \end{array}\right). \tag{C.37}$$

   What did you multiply by? Why?

4. Now we should deal with column 1.

   a. We want to get a 1 in row 1 column 1. We can do this by scaling row 1. What did you scale by? Why? Your end result should be

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & 1 & 2 & \frac{4}{3} \\ 0 & -6 & -21 & -5 \end{array}\right). \tag{C.38}$$

   b. Now scale row 1 by something and add it to row 0 so that the entry in row 0 column 1 becomes a 0.

   c. Next scale row 1 by something and add it to row 2 so that the entry in row 2 column 1 becomes a 0.

   d. At this point you should have the augmented system

$$\left(\begin{array}{ccc|c} 1 & 0 & -1 & -\frac{5}{3} \\ 0 & 1 & 2 & \frac{4}{3} \\ 0 & 0 & -9 & 3 \end{array}\right). \tag{C.39}$$

5. Finally we need to work with column 2.

   a. Make the value in row 2 column 2 a 1 by scaling row 2. What did you scale by? Why?

   b. Scale row 2 by something and add it to row 1 so that the entry in row 1 column 2 becomes a 0. What did you scale by? Why?

c. Scale row 2 by something and add it to row 0 so that the entry in row 0 column 2 becomes a 0. What did you scale by? Why?

d. By the time you have made it this far you should have the system

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & -\frac{1}{3} \end{array}\right) \tag{C.40}$$

and you should be able to read off the solution to the system.

6. You should verify your answer in two different ways:

   a. If you substitute your values into the original system then all of the equal signs should be true. Verify this.

   b. If you substitute your values into the matrix equation and perform the matrix-vector multiplication on the left-hand side of the equation you should get the right-hand side of the equation. Verify this.

Exercise C.16. Summarize the process for doing Gaussian Elimination to solve a square system of linear equations.

C.5.2. The LU Decomposition

You may have used the `rref()` command either on a calculator in other software to perform row reduction in the past. You will be surprised to learn that there is no `rref()` command in Python's `numpy` library! That's because there are far more efficient and stable ways to solve a linear system on a computer. There is an `rref` command in Python's `sympy` (symbolic Python) library, but given that it works with symbolic algebra it is quite slow.

In solving systems of equations we are interested in equations of the form $Ax = b$. Notice that the b vector is just along for the ride, so to speak, in the row reduction process since none of the values in b actually cause you to make different decisions in the row reduction algorithm. Hence, we only really need to focus on the matrix A. Furthermore, let us change our awfully restrictive view of always seeking a matrix of the form

$$\left(\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & \star \\ 0 & 1 & \cdots & 0 & \star \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \star \end{array}\right) \tag{C.41}$$

and instead say:

> *What if we just row reduce until the system is simple enough to solve by hand?*

That's what the next several exercises are going to lead you to. Our goal here is to develop an algorithm that is fast to implement on a computer and simultaneously performs the same basic operations as row reduction for solving systems of linear equations.

Exercise C.17. Let A be defined as

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}. \tag{C.42}$$

1. The first step in row reducing A would be to multiply row 0 by -4 and add it to row 1. Do this operation by hand so that you know what the result is supposed to be. Check out the following amazing observation. Define the matrix L_1 as follows:

   $$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{C.43}$$

   Now multiply L_1 and A.

   $$L_1 A = \begin{pmatrix} \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \end{pmatrix} \tag{C.44}$$

   What just happened?!

2. Let us do it again. The next step in the row reduction of your result from part (1) would be to multiply row 0 by -7 and add to row 2. Again, do this by hand so you know what the result should be. Then define the matrix L_2 as

   $$L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -7 & 0 & 1 \end{pmatrix} \tag{C.45}$$

   and find the product $L_2 (L_1 A)$.

   $$L_2 (L_1 A) = \begin{pmatrix} \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \end{pmatrix} \tag{C.46}$$

   Pure insanity!!

3. Now let us say that you want to make the entry in row 2 column 1 into a 0 by scaling row 1 by something and then adding to row 2. Determine what the scalar would be and then determine which matrix, call it L_3, would do the trick so that $L_3(L_2L_1A)$ would be the next row reduced step.

$$L_3 = \begin{pmatrix} 1 & \_\_\_ & \_\_\_ \\ \_\_\_ & 1 & \_\_\_ \\ \_\_\_ & \_\_\_ & 1 \end{pmatrix} \tag{C.47}$$

$$L_3\,(L_2L_1A) = \begin{pmatrix} \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \end{pmatrix} \tag{C.48}$$

Exercise C.18. Apply the same idea from the previous problem to do the first three steps of row reduction to the matrix

$$A = \begin{pmatrix} 2 & 6 & 9 \\ -6 & 8 & 1 \\ 2 & 2 & 10 \end{pmatrix} \tag{C.49}$$

Exercise C.19. Now let us make a few observations about the two previous problems.

1. What will multiplying A by a matrix of the form

$$\begin{pmatrix} 1 & 0 & 0 \\ c & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{C.50}$$

   do?

2. What will multiplying A by a matrix of the form

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ c & 0 & 1 \end{pmatrix} \tag{C.51}$$

   do?

3. What will multiplying A by a matrix of the form

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & c & 1 \end{pmatrix} \tag{C.52}$$

   do?

4. More generally: If you wanted to multiply row j of an $n \times n$ matrix by c and add it to row k, that is the same as multiplying by what matrix?

Exercise C.20. After doing all of the matrix products, $L_3L_2L_1A$, the resulting matrix will have zeros in the entire lower triangle. That is, all of the non-zero entries of the resulting matrix will be on the main diagonal or above. We call this matrix U, for upper-triangular. Hence, we have formed a matrix

$$L_3L_2L_1A = U \tag{C.53}$$

and if we want to solve for A we would get

$$A = (\underline{\qquad\quad})^{-1}(\underline{\qquad\quad})^{-1}(\underline{\qquad\quad})^{-1}U \tag{C.54}$$

(Take care that everything is in the right order in your answer.)

Exercise C.21. It would be nice, now, if the inverses of the L matrices were easy to find. Use `np.linalg.inv()` to directly compute the inverse of L_1, L_2, and L_3 for each of the example matrices. Then complete the statement: If L_k is an identity matrix with some non-zero c in row i and column j then L_k^{-1} is what matrix?

Exercise C.22. We started this discussion with A as

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \tag{C.55}$$

and we defined

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -7 & 0 & 1 \end{pmatrix}, \quad \text{and} \quad L_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix}. \tag{C.56}$$

Based on your answer to the previous exercises we know that

$$A = L_1^{-1}L_2^{-1}L_3^{-1}U. \tag{C.57}$$

Explicitly write down the matrices L_1^{-1}, L_2^{-1}, and L_3^{-1}.

Now explicitly find the product $L_1^{-1}L_2^{-1}L_3^{-1}$ and call this product L. Verify that L itself is also a lower-triangular matrix with ones on the main diagonal. Moreover, take note of exactly the form of the matrix. The answer should be super surprising to you!!

Throughout all of the preceding exercises, our final result is that we have factored the matrix A into the product of a lower-triangular matrix and an upper-triangular matrix. Stop and think about that for a minute ... we just factored a matrix!

Let us return now to our discussion of solving the system of equations $Ax = b$. If A can be factored into $A = LU$ then the system of equations can be rewritten as $LUx = b$. As we will see in the next subsection, solving systems of equations with triangular matrices is super fast and relatively simple! Hence, we have partially achieved our modified goal of reducing the row reduction into some simpler case.[3]

It remains to implement the LU decomposition (also called the LU factorization) in Python.

Example C.7 (The LU Factorization)**.** The following Python function takes a square matrix A and outputs the matrices L and U such that $A = LU$. The entire code is given to you. It will be up to you in the next exercise to pick apart every step of the function.

```python
def myLU(A):
    n = A.shape[0] # get the dimension of the matrix A
    L = np.eye(n) # Build the identity part of L
    U = np.copy(A) # start the U matrix as a copy of A
    for j in range(0,n-1):
        for i in range(j+1,n):
            mult = U[i,j] / U[j,j]
            U[i, :] = U[i, :] - mult * U[j,:]
            L[i,j] = mult
    return L,U
```

---

[3]Take careful note here. We have actually just built a special case of the LU decomposition. Remember that in row reduction you are allowed to swap the order of the rows, but in our LU algorithm we do not have any row swaps. The version of LU with row swaps is called LU with partial pivoting. We will not built the full partial pivoting algorithm in this text but feel free to look it up. The wikipedia page is a decent place to start. What you will find is that there are indeed many different versions of the LU decomposition.

Exercise C.23. Go to Example C.7 and go through every iteration of every loop **by hand** starting with the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}. \tag{C.58}$$

Give details of what happens at every step of the algorithm. I will get you started.

- `n=3`, `L` starts as an identity matrix of the correct size, and `U` starts as a copy of `A`.
- Start the outer loop: `j=0`: (`j` is the counter for the column)
  - Start the inner loop: `i=1`: (`i` is the counter for the row)
    - `mult = A[1,0] / A[0,0]` so `mult`$= 4/1$.
    - `A[1, 1:3] = A[1, 1:3] - 4 * A[0,1:3]`. Translated, this states that columns 1 and 2 of matrix A took their original value minus 4 times the corresponding values in row 0.
    - `U[1, 1:3] = A[1, 1:3]`. Now we replace the locations in U with the updated information from our first step of row reduction.
    - `L[1,0]=4`. We now fill the L matrix with the proper value.
    - `U[1,0]=0`. Finally, we zero out the lower triangle piece of the U matrix which we have now taken care of.
  - `i=2`:
    - ... keep going from here ...

Exercise C.24. Apply your new `myLU` code to other square matrices and verify that indeed A is the product of the resulting L and U matrices. You can produce a random matrix with `np.random.randn(n,n)` where `n` is the number of rows and columns of the matrix. For example, `np.random.randn(10,10)` will produce a random 10×10 matrix with entries chosen from the normal distribution with centre 0 and standard deviation 1. Random matrices are just as good as any other when testing your algorithm.

C.5.3. Solving Triangular Systems

We now know that row reduction is just a collection of sneaky matrix multiplications. In the previous exercises we saw that we can often turn our system of equations $Ax = b$ into the system $LUx = b$ where L is lower-triangular (with ones on the main diagonal) and U is upper-triangular. But why was this important?

Well, if $LUx = b$ then we can rewrite our system of equations as two systems:

$$\text{An upper-triangular system: } Ux = y \tag{C.59}$$

and

$$\text{A lower-triangular system: } Ly = b. \tag{C.60}$$

In the following exercises we will devise algorithms for solving triangular systems. After we know how to work with triangular systems we will put all of the pieces together and show how to leverage the LU decomposition and the solution techniques for triangular systems to quickly and efficiently solve linear systems.

Exercise C.25. Outline a fast algorithm (without formal row reduction) for solving the lower-triangular system

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}. \tag{C.61}$$

Exercise C.26. As a convention we will always write our lower-triangular matrices with ones on the main diagonal. Generalize your steps from the previous exercise so that you have an algorithm for solving any lower-triangular system. The most natural algorithm that most people devise here is called **forward substitution**.

Definition C.4 (Forward Substutition Algorithm (`lsolve`))**.** The general statement of the Forward Substitution Algorithm is:

Solve $Ly = b$ for y, where the matrix L is assumed to be lower-triangular with ones on the main diagonal.

The code below gives a full implementation of the **Forward Substitution** algorithm (also called the `lsolve` algorithm).

```python
def lsolve(L, b):
    # L is assumed to be a lower-triangular np.array
    n = b.size # what does this do?
    y = np.zeros(n) # what does this do?
    for i in range(n):
        # start the loop by assigning y to the value on the right
        y[i] = b[i]
        for j in range(i): # now adjust y
            y[i] = y[i] - L[i,j] * y[j]
    return(y)
```

Exercise C.27. Work with your partner(s) to apply the `lsolve()` code to the lower-triangular system

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} \tag{C.62}$$

by hand. It is incredibly important to implement numerical linear algebra routines by hand a few times so that you truly understand how everything is being tracked and calculated.

I will get you started.

- Start: `i=0`:
  - `y[0]=1` since `b[0]=1`.
  - The next `for` loop does not start since `range(0)` has no elements (stop and think about why this is).
- Next step in the loop: `i=1`:
  - `y[1]` is initialized as 0 since `b[1]=0`.
  - Now we enter the inner loop at `j=0`:
    - What does `y[1]` become when `j=0`?
  - Does `j` increment to anything larger?
- Finally we increment `i` to `i=2`:
  - What does `y[2]` get initialized to?

  - Enter the inner loop at `j=0`:
    - What does `y[2]` become when `j=0`?
  - Increment the inner loop to `j=1`:
    - What does `y[2]` become when `j=1`?
- Stop

Exercise C.28. Copy the code from Definition C.4 into a Python function but in your code write a comment on every line stating what it is doing. Write a test script that creates a lower-triangular matrix of the correct form and a right-hand side b and solve for y. Test your code by giving it a large lower-triangular system.

Now that we have a method for solving lower-triangular systems, let us build a similar method for solving upper-triangular systems. The merging of lower and upper-triangular systems will play an important role in solving systems of equations.

Exercise C.29. Outline a fast algorithm (without formal row reduction) for solving the upper-triangular system

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & -9 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -4 \\ 3 \end{pmatrix} \tag{C.63}$$

The most natural algorithm that most people devise here is called **backward substitution**. Notice that in our upper-triangular matrix we do not have a diagonal containing all ones.

Exercise C.30. Generalize your backward substitution algorithm from the previous problem so that it could be applied to any upper-triangular system.

Definition C.5 (Backward Substitution Algorithm)**.** The following code solves the problem $Ux = y$ using backward substitution. The matrix U is assumed to be upper-triangular. You will notice that most of this code is incomplete. It is your job to complete this code, and the next exercise should help.

```
def usolve(U, y):
    # U is assumed to be an upper-triangular np.array
    n = y.size
    x = np.zeros(n)
    for i in range( ??? ):     # what should we be looping over?
        x[i] = y[i] / ???      # what should we be dividing by?
        for j in range( ??? ): # what should we be looping over:
            x[i] = x[i] - U[i,j] * x[j] / ??? # complete this line
            # ... what does the previous line do?
    return(x)
```

Exercise C.31. Now we will work through the backward substitution algorithm to help fill in the blanks in the code. Consider the upper-triangular system

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & -9 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -4 \\ 3 \end{pmatrix} \tag{C.64}$$

Work the code from Definition C.5 to solve the system. Keep track of all of the indices as you work through the code. You may want to work this problem in conjunction with the previous two problems to unpack all of the parts of the *backward substitution* algorithm.

I will get you started.

- In your backward substitution algorithm you should have started with the last row, therefore the outer loop starts at `n-1` and reads backward to 0. (Why are we starting at `n-1` and not `n`?)
- Outer loop: `i=2`:
  - We want to solve the equation $-9x_2 = 3$ so the clear solution is to divide by -9. In code this means that `x[2]=y[2]/U[2,2]`.
  - There is nothing else to do for row 3 of the matrix, so we should not enter the inner loop. How can we keep from entering the inner loop?
- Outer loop: `i=1`:

- Now we are solving the algebraic equation $-3x_1 - 6x_2 = -4$. If we follow the high school algebra we see that $x_1 = \frac{-4-(-6)x_2}{-3}$ but this can be rearranged to

$$x_1 = \frac{-4}{-3} - \frac{-6x_2}{-3}. \tag{C.65}$$

  So we can initialize x_1 with $x_1 = \frac{-4}{-3}$. In code, this means that we initialize with `x[1] = y[1] / U[1,1]`.

- Now we need to enter the inner loop at `j=2`: (why are we entering the loop at `j=2`?)

  * To complete the algebra we need to take our initialized value of `x[1]` and subtract off $\frac{-6x_2}{-3}$. In code this is `x[1] = x[1] - U[1,2] * x[2] / U[1,1]`

- There is nothing else to do so the inner loop should end.

• Outer loop: `i=0`:

- Finally, we are solving the algebraic equation $x_0 + 2x_1 + 3x_2 = 1$ for x_0. The clear and obvious solution is $x_0 = \frac{1-2x_1-3x_2}{1}$ (why am I explicitly showing the division by 1 here?).

- Initialize x_0 at `x[0] = ???`

- Enter the inner loop at `j=2`:

  * Adjust the value of `x[0]` by subtracting off $\frac{3x_2}{1}$. In code we have `x[0] = x[0] - ??? * ??? / ???`

- Increment `j` to `j=1`:

  * Adjust the value of `x[0]` by subtracting off $\frac{2x_1}{1}$. In code we have `x[0] = x[0] - ??? * ??? / ???`

• Stop.

• You should now have a solution to the equation $Ux = y$. Substitute your solution in and verify that your solution is correct.

Exercise C.32. Copy the code from Definition C.5 into a Python function but in your code write a comment on every line stating what it is doing. Write a test script that creates an upper-triangular matrix of the correct form and a right-hand side y and solve for x. Your code needs to work on systems of arbitrarily large size.

C.5.4. Solving Systems with LU Decomposition

We are finally ready for the punch line of this whole LU and triangular systems business!

Exercise C.33. If we want to solve $Ax = b$ then

1. If we can, write the system of equations as $LUx = b$.
2. Solve $Ly = b$ for y using forward substitution.
3. Solve $Ux = y$ for x using backward substitution.

Pick a matrix A and a right-hand side b and solve the system using this process.

Exercise C.34. Try the process again on the 3×3 system of equations

$$\begin{pmatrix} 3 & 6 & 8 \\ 2 & 7 & -1 \\ 5 & 2 & 2 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} -13 \\ 4 \\ 1 \end{pmatrix} \tag{C.66}$$

That is: Find matrices L and U such that $Ax = b$ can be written as $LUx = b$. Then do two triangular solves to determine x.

Let us take stock of what we have done so far.

- Solving lower-triangular systems is super fast and easy!
- Solving upper-triangular systems is super fast and easy (so long as we never divide by zero).
- It is often possible to rewrite the matrix A as the product of a lower-triangular matrix L and an upper-triangular matrix U so $A = LU$.
- Now we can re-frame the equation $Ax = b$ as $LUx = b$.
- Substitute $y = Ux$ so the system becomes $Ly = b$. Solve for y with forward substitution.
- Now solve $Ux = y$ using backward substitution.

We have successfully take row reduction and turned into some fast matrix multiplications and then two very quick triangular solves. Ultimately this will be a faster algorithm for solving a system of linear equations.

Definition C.6 (Solving Linear Systems with the LU Decomposition)**.** Let A be a square matrix in $\mathbb{R}^{n \times n}$ and let $x, b \in \mathbb{R}^n$. To solve the problem $Ax = b$,

1. Factor A into lower and upper-triangular matrices $A = LU$.
   `L, U = myLU(A)`
2. The system can now be written as $LUx = b$. Substitute $Ux = y$ and solve the system $Ly = b$ with forward substitution. `y = lsolve(L,b)`
3. Finally, solve the system $Ux = y$ with backward substitution.
   `x = usolve(U,y)`

Exercise C.35. The *LU* decomposition is not perfect. Discuss where the algorithm will fail.

Exercise C.36. What happens when you try to solve the system of equations

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 7 \\ 9 \\ -3 \end{pmatrix} \tag{C.67}$$

with the *LU* decomposition algorithm? Discuss.

C.6. The QR Factorization

In this section we will try to find an improvement on the *LU* factorization scheme from the previous section. What we will do here is leverage the geometry of the column space of the A matrix instead of leveraging the row reduction process.

Exercise C.37. We want to solve the system of equations

$$\begin{pmatrix} 1/3 & 2/3 & 2/3 \\ 2/3 & 1/3 & -2/3 \\ -2/3 & 2/3 & -1/3 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 6 \\ 12 \\ -9 \end{pmatrix}. \tag{C.68}$$

1. We could do row reduction by hand … yuck … do not do this.
2. We could apply our new-found skills with the LU decomposition to solve the system, so go ahead and do that with your Python code.
3. What do you get if you compute the product A^TA?

   a. Why do you get what you get? In other words, what was special about A that gave such an nice result?

   b. What does this mean about the matrices A and A^T?

4. Now let us leverage what we found in part (3) to solve the system of equations $Ax = b$ much faster. Multiply both sides of the matrix equation by A^T, and now you should be able to just read off the solution. This seems amazing!!
5. What was it about this particular problem that made part (4) so elegant and easy?

The previous exercise tells us something amazing:

Theorem C.1 (Orthonomal Matrices)**.** *If A is an orthonormal matrix where the columns are mutually orthogonal and every column is a unit vector, then $A^T = A^{-1}$ and to solve the system of equation $Ax = b$ we simply need to multiply both sides of the equation by A^T. Hence, the solution to $Ax = b$ is just $x = A^Tb$ in this special case.*

Theorem C.1 begs an obvious question: *Is there a way to turn any matrix A* into an orthogonal matrix so that we can solve $Ax = b$ in this same very efficient and fast way?

The answer: Yes. Kind of.

In essence, if we can factor our coefficient matrix into an orthonormal matrix and some other nicely formatted matrix (like a triangular matrix, perhaps) then the job of solving the linear system of equations comes down to matrix multiplication and a quick triangular solve – both of which are extremely fast!

What we will study in this section is a new matrix factorization called the QR factorization whose goal is to convert the matrix A into a product of two matrices, Q and R, where Q is orthonormal and R is upper-triangular.

Exercise C.38. Let us say that we have a matrix A and we know that it can be factored into $A = QR$ where Q is an orthonormal matrix and R is an upper-triangular matrix. How would we then leverage this factorization to solve the system of equation $Ax = b$ for x?

Before proceeding to the algorithm for the QR factorization let us pause for a moment and review scalar and vector projections from Linear Algebra. In Figure C.1 we see a graphical depiction of the vector u projected onto vector v. Notice that the projection is indeed the perpendicular projection as this is what seems natural geometrically.

The **vector projection** of u onto v is the vector cv. That is, the vector projection of u onto v is a scalar multiple of the vector v. The value of the scalar c is called the **scalar projection** of u onto v.

Figure 4.1: Projection of one vector onto another.

We can arrive at a formula for the scalar projection rather easily if we consider that the vector w in Figure C.1 must be perpendicular to cv. Hence

$$w \cdot (cv) = 0. \tag{C.69}$$

From vector geometry we also know that $w = u - cv$. Therefore

$$(u - cv) \cdot (cv) = 0. \tag{C.70}$$

If we distribute we can see that

$$cu \cdot v - c^2 v \cdot v = 0 \tag{C.71}$$

and therefore either $c = 0$, which is only true if $u \perp v$, or

$$c = \frac{u \cdot v}{v \cdot v} = \frac{u \cdot v}{\|v\|^2}. \tag{C.72}$$

Therefore,

- the **scalar projection of** u onto v is

$$c = \frac{u \cdot v}{\|v\|^2} \tag{C.73}$$

- the **vector projection of** u onto v is

$$cv = \left(\frac{u \cdot v}{\|v\|^2} \right) v \tag{C.74}$$

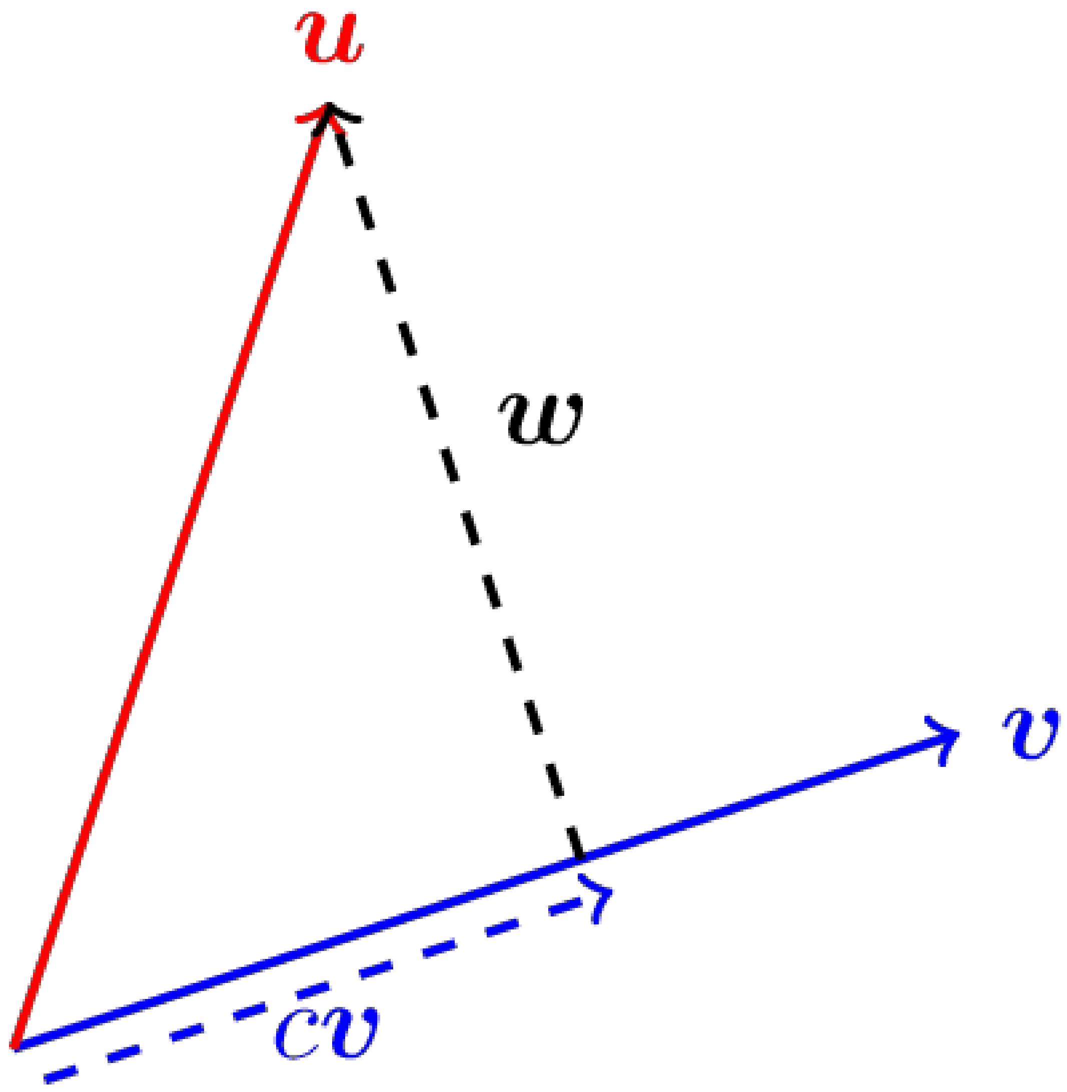


Figure C.1.: Projection of one vector onto another.

Another problem related to scalar and vector projections is to take a basis for the column space of a matrix and transform that basis into an orthogonal (or orthonormal) basis. Indeed, in Figure C.1 if we have the matrix

$$A = \begin{pmatrix} | & | \\ u & v \\ | & | \end{pmatrix} \tag{C.75}$$

it should be clear from the picture that the columns of this matrix are not perpendicular. However, if we take the vector v and the vector w we do arrive at two orthogonal vector that form a basis for the same space. Moreover, if we normalize these vectors (by dividing by their respective lengths) then we can easily transform the original basis for the column space of A into an orthonormal basis. This process is called the Gram-Schmidt process, and you have encountered it in your Linear Algebra module.

Now we return to our goal of finding a way to factor a matrix A into an orthonormal matrix Q and an upper-triangular matrix R. The algorithm that we are about to build depends greatly on the ideas of scalar and vector projections.

Exercise C.39. We want to build a QR factorization of the matrix A in the matrix equation $Ax = b$ so that we can leverage the fact that solving the equation $QRx = b$ is easy. Consider the matrix A defined as

$$A = \begin{pmatrix} 3 & 1 \\ 4 & 1 \end{pmatrix}. \tag{C.76}$$

Notice that the columns of A are NOT orthonormal (they are not unit vectors and they are not perpendicular to each other).

1. Draw a picture of the two column vectors of A in $\mathbb{R}^2$. we will use this picture to build geometric intuition for the rest of the QR factorization process.
2. Define a_0 as the first column of A and a_1 as the second column of A. That is

$$a_0 = \begin{pmatrix} 3 \\ 4 \end{pmatrix} \quad \text{and} \quad a_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}. \tag{C.77}$$

   Turn a_0 into a unit vector and call this unit vector q_0

$$q_0 = \frac{a_0}{\|a_0\|} = \begin{pmatrix} \underline{\qquad} \\ \underline{\qquad} \end{pmatrix}. \tag{C.78}$$

   This vector q_0 will be the first column of the 2×2 matrix Q. Why is this a nice place to start building the Q matrix (think about the desired structure of Q)?

3. In your picture of a_0 and a_1 mark where q_0 is. Then draw the orthogonal projection from a_1 onto q_0. In your picture you should now see a right triangle with a_1 on the hypotenuse, the projection of a_1 onto q_0 on one leg, and the second leg is the vector difference of the hypotenuse and the first leg. Simplify the projection formula for leg 1 and write the formula for leg 2.

$$\text{hypotenuse } = a_1 \tag{C.79}$$

$$\text{leg 1 } = \left(\frac{a_1 \cdot q_0}{q_0 \cdot q_0} \right) q_0 = \underline{\hspace{3cm}} \tag{C.80}$$

$$\text{leg 2 } = \underline{\hspace{3cm}} - \underline{\hspace{3cm}}. \tag{C.81}$$

4. Compute the vector for leg 2 and then normalize it to turn it into a unit vector. Call this vector q_1 and put it in the second column of Q.

5. Verify that the columns of Q are now orthogonal and are both unit vectors.

6. The matrix R is supposed to complete the matrix factorization $A = QR$. We have built Q as an orthonormal matrix. How can we use this fact to solve for the matrix R?

7. You should now have an orthonormal matrix Q and an upper-triangular matrix R. Verify that $A = QR$.

8. An alternate way to build the R matrix is to observe that

$$R = \begin{pmatrix} a_0 \cdot q_0 & a_1 \cdot q_0 \\ 0 & a_1 \cdot q_1 \end{pmatrix}. \tag{C.82}$$

Show that this is indeed true for the matrix A from this problem.

Exercise C.40. Keeping track of all of the arithmetic in the QR factorization process is quite challenging, so let us leverage Python to do some of the work for us. The following block of code walks through the previous exercise without any looping (that way we can see every step transparently). Some of the code is missing so you will need to fill it in.

```
import numpy as np
# Define the matrix $A$
A = np.array([[3,1],[4,1]])
n = A.shape[0]
# Build the vectors a0 and a1
a0 = A[??? , ???] # ... write code to get column 0 from A
a1 = A[??? , ???] # ... write code to get column 1 from A
# Set up storage for Q
Q = np.zeros( (n,n) )
# build the vector q0 by normalizing a0
q0 = a0 / np.linalg.norm(a0)
# Put q0 as the first column of Q
Q[:,0] = q0
# Calculate the lengths of the two legs of the triangle
leg1 = # write code to get the vector for leg 1 of the triangle
leg2 = # write code to get the vector for leg 2 of the triangle
# normalize leg2 and call it q1
q1 = # write code to normalize leg2
q1 = q1 / np.linalg.norm(q1) # Just to be safe with normalization if not implied
Q[:,1] = q1 # What does this line do?
R = # ... build the R matrix out of A and Q

print("The Q matrix is \n",Q,"\n")
print("The R matrix is \n",R,"\n")
print("The A matrix is \n",A,"\n")
print("The product QR is\n",Q @ R)
```

Exercise C.41. You should notice that the code in the previous exercise does not depend on the specific matrix $A$ that we used? Put in a different 2×2 matrix and verify that the process still works. That is, verify that Q is orthonormal, R is upper-triangular, and $A = QR$. Be sure, however, that your matrix $A$ is full rank.

Exercise C.42. Draw two generic vectors in $\mathbb{R}^2$ and demonstrate the process outlined in the previous problem to build the vectors for the Q matrix starting from your generic vectors.

Exercise C.43. Now we will extend the process from the previous exercises to three dimensions. This time we will seek a matrix Q that has three orthonormal vectors starting from the three original columns of a 3×3 matrix A. Perform each of the following steps **by hand** on the matrix

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}. \tag{C.83}$$

In the end you should end up with an orthonormal matrix Q and an upper-triangular matrix R.

- **Step 1**: Pick column a_0 from the matrix A and normalize it. Call this new vector q_0 and make that the first column of the matrix Q.
- **Step 2**: Project column a_1 of A onto q_0. This forms a right triangle with a_1 as the hypotenuse, the projection of a_1 onto q_0 as one of the legs, and the vector difference between these two as the second leg. Notice that the second leg of the newly formed right triangle is perpendicular to q_0 by design. If we normalize this vector then we have the second column of Q, q_1.
- **Step 3:** Now we need a vector that is perpendicular to both q_0 AND q_1. To achieve this we are going to project column a_2 from A onto the plane formed by q_0 and q_1. we will do this in two steps:
  - **Step 3a**: We first project a_2 down onto both q_0 and q_1.
  - **Step 3b**: The vector that is perpendicular to both q_0 and q_1 will be the difference between a_2 the projection of a_2 onto q_0 and the projection of a_2 onto q_1. That is, we form the vector $w = a_2 - (a_2 \cdot q_0)q_0 - (a_2 \cdot q_1)q_1$. Normalizing this vector will give us q_2. (Stop now and prove that q_2 is indeed perpendicular to both q_1 and q_0.)

The result should be the matrix Q which contains orthonormal columns. To build the matrix R we simply recall that $A = QR$ and $Q^{-1} = Q^T$ so $R = Q^T A$.

Exercise C.44. Repeat the previous exercise but write code for each step so that Python can handle all of the computations. Again use the matrix

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}. \tag{C.84}$$

Example C.8. (**QR for** $n = 3$) For the sake of clarity let us now write down the full QR factorization for a 3×3 matrix.

If the columns of A are a_0, a_1, and a_2 then

$$q_0 = \frac{a_0}{\|a_0\|} \tag{C.85}$$

$$q_1 = \frac{a_1 - (a_1 \cdot q_0)\, q_0}{\|a_1 - (a_1 \cdot q_0)\, q_0\|} \tag{C.86}$$

$$q_2 = \frac{a_2 - (a_2 \cdot q_0)\, q_0 - (a_2 \cdot q_1)\, q_1}{\|a_2 - (a_2 \cdot q_0)\, q_0 - (a_2 \cdot q_1)\, q_1\|} \tag{C.87}$$

and

$$R = \begin{pmatrix} a_0 \cdot q_0 & a_1 \cdot q_0 & a_2 \cdot q_0 \\ 0 & a_1 \cdot q_1 & a_2 \cdot q_1 \\ 0 & 0 & a_2 \cdot q_2 \end{pmatrix} \tag{C.88}$$

Exercise C.45 (The QR Factorization)**.** Now we are ready to build general code for the QR factorization. The following Python function definition is partially complete. Fill in the missing pieces of code and then test your code on square matrices of many different sizes. The easiest way to check if you have an error is to find the normed difference between A and QR with `np.linalg.norm(A - Q*R)`.

```python
import numpy as np
def myQR(A):
    n = A.shape[0]
    Q = np.zeros( (n,n) )
    for j in range( ??? ): # The outer loop goes over the columns
        q = A[:,j]
        # The next loop is meant to do all of the projections.
        # When do you start the inner loop and how far do you go?
        # Hint: You do not need to enter this loop the first time
        for i in range( ??? ):
            length_of_leg = np.dot(A[:,j], Q[:,i])
            q = q -  ??? * ??? # This is where we do projections
        Q[:,j] = q / np.linalg.norm(q)
    R = # finally build the R matrix
    return Q, R
```

```
# Test Code
A = np.array( ... )
# or you can build A with use np.random.randn()
# Often time random matrices are good test cases
Q, R = myQR(A)
error = np.linalg.norm(A - Q @ R)
print(error)
```

We now have a robust algorithm for doing QR factorization of square matrices we can finally return to solving systems of equations.

Theorem C.2. *(**Solving Systems with** QR) Remember that we want to solve $Ax = b$ and since $A = QR$ we can rewrite it with $QRx = b$. Since we know that Q is orthonormal by design we can multiply both sides of the equation by Q^T to get $Rx = Q^T b$. Finally, since R is upper-triangular we can use our `usolve` code from the previous section to solve the resulting triangular system.*

Exercise C.46. Solve the system of equations

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix} \tag{C.89}$$

by first computing the QR factorization of A and then solving the resulting upper-triangular system.

Exercise C.47. Write code that builds a random $n \times n$ matrix and a random $n \times 1$ vector. Solve the equation $Ax = b$ using the QR factorization and compare the answer to what we find from `np.linalg.solve()`. Do this many times for various values of n and create a plot with n on the horizontal axis and the normed error between Python's answer and your answer from the QR algorithm on the vertical axis. It would be wise to use a `plt.semilogy()` plot. To find the normed difference you should use `np.linalg.norm()`. What do you notice?

C.7. Over-determined Systems and Curve Fitting

Exercise C.48. Consider the problem of finding the quadratic function $f(x) = ax^2+bx+c$ that *best fits* the points

$$(0, 1.07), (1, 3.9), (2, 14.8), (3, 26.8). \tag{C.90}$$

We do not know the values of a, b, or c but we do have four different (x, y) ordered pairs. Hence, we have four equations:

$$1.07 = a(0)^2 + b(0) + c \tag{C.91}$$

$$3.9 = a(1)^2 + b(1) + c \tag{C.92}$$

$$14.8 = a(2)^2 + b(2) + c \tag{C.93}$$

$$26.8 = a(3)^2 + b(3) + c. \tag{C.94}$$

There are four equations and only three unknowns. This is what is called an **over determined systems** – when there are more equations than unknowns. Let us play with this problem.

1. First turn the system of equations into a matrix equation.

$$\begin{pmatrix} 0 & 0 & 1 \\ \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 1.07 \\ 3.9 \\ 14.8 \\ 26.8 \end{pmatrix}. \tag{C.95}$$

2. None of our techniques for solving systems will likely work here since it is highly unlikely that the vector on the right-hand side of the equation is in the column space of the coefficient matrix. Discuss this.

3. One solution to the unfortunate fact from part (b) is that we can project the vector on the right-hand side into the subspace spanned by the columns of the coefficient matrix. Think of this as casting the shadow of the right-hand vector down onto the space spanned by the columns. If we do this projection we will be able to solve the equation for the values of a, b, and c that will create the projection exactly – and hence be as close as we can get to the actual right-hand side. Draw a picture of what we have said here.

4. Now we need to project the right-hand side, call it b, onto the column space of the the coefficient matrix A. Recall the following facts:

- Projections are dot products
- Matrix multiplication is nothing but a bunch of dot products.
- The projections of b onto the columns of A are the dot products of b with each of the columns of A.
- What matrix can we multiply both sides of the equation $Ax = b$ by in order for the right-hand side to become the projection that we want? (Now do the projection in Python)

5. If you have done part (d) correctly then you should now have a square system (i.e. the matrix on the left-hand side should now be square). Solve this system for a, b, and c.

Theorem C.3 (Solving Overdetermined Systems)**.** *If $Ax = b$ is an overdetermined system (i.e. A has more rows than columns) then we first multiply both sides of the equation by A^T (why do we do this?) and then solve the square system of equations $(A^TA)x = A^Tb$ using a system solving like LU or QR. The answer to this new system is interpreted as the vector x which solves exactly for the projection of b onto the column space of A.*

The equation $(A^TA)x = A^Tb$ is called ***the normal equations*** *and arises often in Statistics and Machine Learning.*

Exercise C.49. Fit a linear function to the following data. Solve for the slope and intercept using the technique outlined in Theorem C.3. Make a plot of the points along with your best fit curve.

x	y
0	4.6
1	11
2	12
3	19.1
4	18.8
5	39.5
6	31.1
7	43.4
8	40.3
9	41.5

x	y
10	41.6

Exercise C.50. Fit a quadratic function to the following data using the technique outlined in Theorem C.3. Make a plot of the points along with your best fit curve.

x	y
0	-6.8
1	11.8
2	50.6
3	94
4	224.3
5	301.7
6	499.2
7	454.7
8	578.5
9	1102
10	1203.2

Code to download the data directly is given below.

```python
import numpy as np
import pandas as pd
URL1 = 'https://raw.githubusercontent.com/NumericalMethodsSullivan'
URL2 = '/NumericalMethodsSullivan.github.io/master/data/'
URL = URL1+URL2
data = np.array( pd.read_csv(URL+'Exercise4_52.csv') )
# Exercise4_52.csv
```

Exercise C.51. The Statistical technique of curve fitting is often called "linear regression." This even holds when we are fitting quadratic functions, cubic functions, etc to the data … we still call that linear regression! Why?

This section of the text on solving over determined systems is just a bit of a teaser for a bit of higher-level statistics, data science, and machine learning. The normal equations and solving systems via projections is the starting point of many modern machine learning algorithms.

C.8. The Eigenvalue-Eigenvector Problem

We finally turn our attention to another major topic in numerical linear algebra.[4]

Definition C.7 (The Eigenvalue Problem)**.** Recall that the eigenvectors, x, and the eigenvalues, λ of a square matrix satisfy the equation $Ax = \lambda x$. Geometrically, the eigen-problem is the task of finding the special vectors x such that multiplication by the matrix A only produces a scalar multiple of x.

Thinking about matrix multiplication, the geometric notion of the eigenvalue problem is rather peculiar since matrix-vector multiplication usually results in a scaling and a rotation of the vector x. Therefore, in some sense the eigenvectors are the only special vectors which avoid geometric rotation under matrix multiplication. For a graphical exploration of this idea see:
https://www.geogebra.org/m/JP2XZpzV.

Recall that to solve the eigen-problem for a square matrix A we complete the following steps:

1. First rearrange the definition of the eigenvalue-eigenvector pair to
$$(Ax - \lambda x) = 0. \tag{C.96}$$

2. Next, factor the x on the right to get
$$(A - \lambda I)x = 0. \tag{C.97}$$

[4]Numerical Linear Algebra is a huge field and there is way more to say … but alas, this is an introductory course in Numerical Analysis so we cannot do everything. Sigh.

3. Now observe that since $x \neq 0$ the matrix $A - \lambda I$ must NOT have an inverse. Therefore,
$$\det(A - \lambda I) = 0. \tag{C.98}$$

4. Solve the equation $\det(A - \lambda I) = 0$ for all of the values of λ.

5. For each λ, find a solution to the equation $(A - \lambda I)x = 0$. Note that there will be infinitely many solutions so you will need to make wise choices for the free variables.

Exercise C.52. Find the eigenvalues and eigenvectors of
$$A = \begin{pmatrix} 1 & 2 \\ 4 & 3 \end{pmatrix}. \tag{C.99}$$

Exercise C.53. In the matrix
$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{C.100}$$
one of the eigenvalues is $\lambda_1 = 0$.

1. What does that tell us about the matrix A?

2. What is the eigenvector v_1 associated with $\lambda_1 = 0$?

3. What is the null space of the matrix A?

OK. Now that you have recalled some of the basics, let us play with a little limit problem. The following exercises are going to work us toward the **power method** for finding certain eigen-structures of a matrix.

Exercise C.54. Consider the matrix

$$A = \begin{pmatrix} 8 & 5 & -6 \\ -12 & -9 & 12 \\ -3 & -3 & 5 \end{pmatrix}. \tag{C.101}$$

This matrix has the following eigen-structure:

$$v_1 = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} \quad \text{with} \quad \lambda_1 = 3 \tag{C.102}$$

$$v_2 = \begin{pmatrix} 2 \\ 0 \\ 2 \end{pmatrix} \quad \text{with} \quad \lambda_2 = 2 \tag{C.103}$$

$$v_3 = \begin{pmatrix} -1 \\ 3 \\ 1 \end{pmatrix} \quad \text{with} \quad \lambda_3 = -1 \tag{C.104}$$

If we have

$$x = -2v_1 + 1v_2 - 3v_3 = \begin{pmatrix} 3 \\ -7 \\ -1 \end{pmatrix} \tag{C.105}$$

then we want to do a bit of an experiment. What happens when we iteratively multiply x by A but at the same time divide by the largest eigenvalue. Let us see:

- What is $A^1x/3^1$?
- What is $A^2x/3^2$?
- What is $A^3x/3^3$?
- What is $A^4x/3^4$?
- ...

It might be nice now to go to some Python code to do the computations (if you have not already). Use your code to conjecture about the following limit.

$$\lim_{k\to\infty} \frac{A^k x}{\lambda_{max}^k} = ???. \tag{C.106}$$

In this limit we are really interested in the direction of the resulting vector, not the magnitude. Therefore, in the code below you will see that we normalize the resulting vector so that it is a unit vector.

Note: be careful, computers do not do infinity, so for powers that are too large you will not get any results.

```python
import numpy as np
A = np.array([[8,5,-6],[-12,-9,12],[-3,-3,5]])
x = np.array([[3],[-7],[-1]])
eigval_max = 3

k = 4
# Note: For numpy arrays, A**k is element-wise power.
# We must use np.linalg.matrix_power for matrix power.
result = np.linalg.matrix_power(A, k) @ x / eigval_max**k
print(result / np.linalg.norm(result) )
```

Exercise C.55. If a matrix A has eigenvectors v_1, v_2, v_3, $\cdots$, v_n with eigenvalues $\lambda_1, \lambda_2, \lambda_3, \dots, \lambda_n$ and x is in the column space of A then what will we get, approximately, if we evaluate $A^k x / \max_j(\lambda_j)^k$ for very large values of k?

Discuss your conjecture with your peers. Then try to verify it with several numerical examples.

Exercise C.56. Explain your result from the previous exercise geometrically.

Exercise C.57. The algorithm that we have been toying with will find the dominant eigenvector of a matrix fairly quickly. Why might you be only interested in the dominant eigenvector of a matrix? Discuss.

Exercise C.58. In this problem we will formally prove the conjecture that you just made. This conjecture will lead us to the **power method** for finding the dominant eigenvector and eigenvalue of a matrix.

1. Assume that A has n linearly independent eigenvectors $v_1, v_2, \dots, v_n$ and choose $x = \sum_{j=1}^n c_j v_j$. You have proved in the past that

$$A^k x = c_1 \lambda_1^k v_1 + c_2 \lambda_2^k v_2 + \cdots c_n \lambda_n^k v_n. \qquad \text{(C.107)}$$

   Stop and sketch out the details of this proof now.

2. If we factor λ_1^k out of the right-hand side we get

$$A^k x = \lambda_1^k \left(c_1 ??? + c_2 \left(\frac{???}{???}\right)^k v_2 + c_3 \left(\frac{???}{???}\right)^k v_3 + \cdots + c_n \left(\frac{???}{???}\right)^k v_n \right) \quad \text{(C.108)}$$

(fill in the question marks)

3. If $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|$ then what happens to each of the $(\lambda_j/\lambda_1)^k$ terms as $k \to \infty$?

4. Using your answer to part (c), what is $\lim_{k\to\infty} A^k x/\lambda_1^k$?

Theorem C.4 (The Power Method)**.** *The following algorithm, called* ***the power method*** *will quickly find the eigenvalue of largest absolute value for a square matrix $A \in \mathbb{R}^{n\times n}$ as well as the associated (normalized) eigenvector. We are assuming that there are n linearly independent eigenvectors of A.*

Step 1: *Given a non-zero vector x, set $v^{(1)} = x/\|x\|$. (Here the superscript indicates the iteration number) Note that the initial vector x is pretty irrelevant to the process so it can just be a random vector of the correct size..*

Step 2: *For $k = 2, 3, \ldots$*

Step 2a: *Compute $\tilde{v}^{(k)} = Av^{(k-1)}$ (this gives a non-normalized version of the next estimate of the dominant eigenvector.)*

Step 2b: *Set $\lambda^{(k)} = \tilde{v}^{(k)} \cdot v^{(k-1)}$. (this gives an approximation of the eigenvalue since if $v^{(k-1)}$ was the actual eigenvector we would have $\lambda = Av^{(k-1)} \cdot v^{(k-1)}$. Stop now and explain this.)*

Step 2c: *Normalize $\tilde{v}^{(k)}$ by computing $v^{(k)} = \tilde{v}^{(k)}/\|\tilde{v}^{(k)}\|$. (This guarantees that you will be sending a unit vector into the next iteration of the loop)*

Exercise C.59. Go through Theorem C.4 carefully and describe what we need to do in each step and why we are doing it. Then complete all of the missing pieces of the following Python function.

```python
import numpy as np
def myPower(A, tol = 1e-8):
    n = A.shape[0]
    x = np.random.randn(n)
    x = # turn x into a unit vector
    # we do not actually need to keep track of the old iterates
    L = 1 # initialize the dominant eigenvalue
    counter = 0 # keep track of how many steps we have taken
    # You can build a stopping rule from the definition
    # Ax = lambda x ...
    while (???) > tol and counter < 10000:
        x = A @ x # update the dominant eigenvector
        x = ??? # normalize
        L = ??? # approximate the eignevalue
        counter += 1 # increment the counter
    return x, L
```

Exercise C.60. Test your `myPower()` function on several matrices where you know the eigenstructure. Then try the `myPower()` function on larger random matrices. You can check that it is working using `np.linalg.eig()` (be sure to normalize the vectors in the same way so you can compare them.)

Exercise C.61. In the Power Method iteration you may end up getting a different sign on your eigenvector as compared to `np.linalg.eig()`. Why might this happen? Generate a few examples so you can see this. You can avoid this issue if you use a `while` loop in your Power Method code and the logical check takes advantage of the fact that we are trying to solve the equation $Ax = \lambda x$. Hint: $Ax = \lambda x$ is equivalent to $Ax - \lambda x = 0$.

Exercise C.62. What happens in the power method iterations when λ_1 is complex. The maximum eigenvalue can certainly be complex if $|\lambda_1|$ (the modulus of the complex number) is larger than all of the other eigenvalues. It may be helpful to build a matrix specifically with complex eigenvalues.[5]

[5]To build a matrix with specific eigenvalues it may be helpful to recall the matrix factorization $A = PDP^{-1}$ where the columns of P are the eigenvectors of A and the diagonal entries of D are the eigenvalues. If you choose P and D then you can build A with your specific eigen-structure. If you are looking for complex eigenvalues then remember that the eigenvectors may well be complex too.

Exercise C.63 (Convergence Rate of the Power Method)**.** The proof that the power method will work hinges on the fact that $|\lambda_1| > |\lambda_2| \ge |\lambda_3| \ge \cdots \ge |\lambda_n|$. In Exercise C.58 we proved that the limit

$$\lim_{k\to\infty} \frac{A^k x}{\lambda_1^k} \tag{C.109}$$

converges to the dominant eigenvector, but how fast is the convergence? What does the speed of the convergence depend on?

Take note that since we are assuming that the eigenvalues are ordered, the ratio λ_2/λ_1 will be larger than λ_j/λ_1 for all $j > 2$. Hence, the speed at which the power method converges depends mostly on the ratio λ_2/λ_1. Let us build a numerical experiment to see how sensitive the power method is to this ratio.

Build a 4×4 matrix A with dominant eigenvalue $\lambda_1 = 1$ and all other eigenvalues less than 1 in absolute value. Then choose several values of λ_2 and build an experiment to determine the number of iterations that it takes for the power method to converge to within a pre-determined tolerance to the dominant eigenvector. In the end you should produce a plot with the ratio λ_2/λ_1 on the horizontal axis and the number of iterations to converge to a fixed tolerance on the vertical axis. Discuss what you see in your plot.

Hint: To build a matrix with specific eigen-structure use the matrix factorization $A = PDP^{-1}$ where the columns of P contain the eigenvectors of A and the diagonal of D contains the eigenvalues. In this case the P matrix can be random but you need to control the D matrix. Moreover, remember that λ_3 and λ_4 should be smaller than λ_2.

C.9. Algorithm Summaries

Exercise C.64. Explain in clear language how to efficiently solve an upper-triangular system of linear equations.

Exercise C.65. Explain in clear language how to efficiently solve a lower-triangular system of linear equations.

Exercise C.66. Explain in clear language how to solve the equation $Ax = b$ using an LU decomposition.

Exercise C.67. Explain in clear language how to solve an overdetermined system of linear equations (more equations than unknowns) numerically.

Exercise C.68. Explain in clear language the algorithm for finding the columns of the Q matrix in the QR factorization. Give all of the mathematical details.

Exercise C.69. Explain in clear language how to find the upper-triangular matrix R in the QR factorization. Give all of the mathematical details.

Exercise C.70. Explain in clear language how to solve the equation $Ax = b$ using a QR decomposition.

Exercise C.71. Explain in clear language how the power method works to find the dominant eigenvalue and eigenvector of a square matrix. Give all of the mathematical details.

C.10. Problems

Exercise C.72. As mentioned much earlier in this chapter, there is an `rref()` command in Python, but it is in the `sympy` library instead of the `numpy` library – it is implemented as a symbolic computation instead of a numerical computation. OK. So what? In this problem we want to compare the time to solve a system of equations $Ax = b$ with each of the following techniques:

- row reduction of an augmented matrix $\begin{pmatrix} A & | & b \end{pmatrix}$ with `sympy`,
- our implementation of the LU decomposition,

- our implementation of the QR decomposition, and
- the `numpy.linalg.solve()` command.

To time code in Python first import the `time` library. Then use `start = time.time()` at the start of your code and `stop = time.time()` and the end of your code. The difference between `stop` and `start` is the elapsed computation time.

Make observations about how the algorithms perform for different sized matrices. You can use random matrices and vectors for A and b. The end result should be a plot showing how the average computation time for each algorithm behaves as a function of the size of the coefficient matrix.

The code below will compute the reduced row echelon form of a matrix (RREF). Implement the code so that you know how it works.

```
import sympy as sp
import numpy as np
# in this problem it will be easiest to start with numpy matrices
A = np.array([[1, 0, 1], [2, 3, 5], [-1, -3, -3]])
b = np.array([[3],[7],[3]])
Augmented = np.c_[A,b] # augment b onto the right hand side of A

Msymbolic = sp.Matrix(Augmented)
MsymbolicRREF = Msymbolic.rref()
print(MsymbolicRREF)
```

To time code you can use code like the following.

```
import time
start = time.time()
# some code that you want to time
stop =  time.time()
total_time = stop - start
print("Total computation time=",total_time)
```

Exercise C.73. Imagine that we have a 1 meter long thin metal rod that has been heated to 100° on the left-hand side and cooled to 0° on the right-hand side. We want to know the temperature every 10 cm from left to right on the rod.

1. First we break the rod into equal 10cm increments as shown. See Figure C.2. How many unknowns are there in this picture?

2. The temperature at each point along the rod is the average of the temperatures at the adjacent points. For example, if we let T_1 be the temperature at point x_1 then
$$T_1 = \frac{T_0 + T_2}{2}. \tag{C.110}$$
Write a system of equations for each of the unknown temperatures.
3. Solve the system for the temperature at each unknown node using either *LU* or *QR* decomposition.

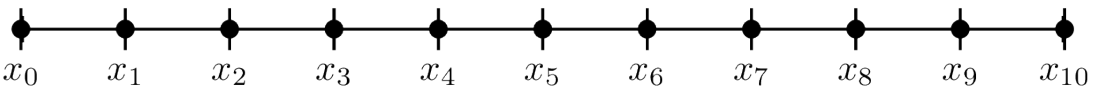


Figure C.2.: A rod to be heated broken into 10 equal-length segments.

Exercise C.74. Write code to solve the following systems of equations via both LU and QR decompositions. If the algorithm fails then be sure to explain exactly why.

1.
$$\begin{aligned} x + 2y + 3z &= 4 \\ 2x + 4y + 3z &= 5 \\ x + y &= 4 \end{aligned} \tag{C.111}$$

2.
$$\begin{aligned} 2y + 3z &= 4 \\ 2x + 3z &= 5 \\ y &= 4 \end{aligned} \tag{C.112}$$

3.
$$\begin{aligned} 2y + 3z &= 4 \\ 2x + 4y + 3z &= 5 \\ x + y &= 4 \end{aligned} \tag{C.113}$$

Exercise C.75. Give a specific example of a non-zero matrix which will NOT have an *LU* decomposition. Give specific reasons why *LU* will fail on your matrix.

Exercise C.76. Give a specific example of a non-zero matrix which will NOT have an *QR* decomposition. Give specific reasons why *QR* will fail on your matrix.

Exercise C.77. Have you ever wondered how scientific software computes a determinant? The formula that you learned for calculating determinants by hand is horribly cumbersome and computationally intractable for large matrices. This problem is meant to give you glimpse of what is *actually* going on under the hood.[6]

If A has an LU decomposition then $A = LU$. Use properties that you know about determinants to come up with a simple way to find the determinant for matrices that have an LU decomposition. Show all of your work in developing your formula.

Once you have your formula for calculating $\det(A)$, write a Python function that accepts a matrix, produces the LU decomposition, and returns the determinant of A. Check your work against Python's `np.linalg.det()` function.

Exercise C.78. For this problem we are going to run a numerical experiment to see how the process of solving the equation $Ax = b$ using the LU factorization performs on random coefficient matrices A and random right-hand sides b. We will compare against Python's algorithm for solving linear systems.

We will do the following:

Create a loop that does the following:

1. Loop over the size of the matrix n.
2. Build a random matrix A of size $n \times n$. You can do this with the code `A = np.random.randn(n,n)`
3. Build a random vector b in $\mathbb{R}^n$. You can do this with the code `b = np.random.randn(n)`
4. Find Python's answer to the problem $Ax = b = 0$ using the command `exact = np.linalg.solve(A,b)`
5. Write code that uses your three LU functions (`myLU`, `lsolve`, `usolve`) to find a solution to the equation $Ax = b$.
6. Find the error between your answer and the exact answer using the code `np.linalg.norm(x - exact)`

[6]Actually, the determinant computation uses LU with partial pivoting which we did not cover here in the text. What we are looking at in this exercise is a smaller sub-case of what happens when you have a matrix A that does not require any row swaps in the row reduction process.

7. Make a plot (`plt.semilogy()`) that shows how the error behaves as the size of the problem changes. You should run this for matrices of larger and larger size but be warned that the loop will run for quite a long time if you go above 300×300 matrices. Just be patient.

Conclusions: What do you notice in your final plot. What does this tell you about the behaviour of our *LU* decomposition code?

Exercise C.79. Repeat Exercise C.78 for the QR decomposition. Your final plot should show both the behaviour of QR and of LU throughout the experiment. What do you notice?

Exercise C.80. Find a least squares solution to the equation $Ax = b$ in two different ways with

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 4 & -2 & 6 \\ 4 & 7 & 8 \\ 3 & 7 & 19 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 5 \\ 2 \\ -2 \\ 8 \end{pmatrix}. \tag{C.114}$$

Exercise C.81. Let A be defined as

$$A = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 1 \end{pmatrix} \tag{C.115}$$

and let b be the vector

$$b = \begin{pmatrix} 2 \\ 3 \end{pmatrix}. \tag{C.116}$$

Notice that A has a tiny, but non-zero, value in the first entry.

1. Solve the linear system $Ax = b$ by hand.
2. Use your `myLU`, `lsolve`, and `usolve` functions to solve this problem using the LU decomposition method.
3. Compare your answers to parts (a) and (b). What went wrong?

Exercise C.82 (Hilbert Matrices). A Hilbert Matrix is a matrix of the form $H_{ij} = 1/(i+j+1)$ where both i and j both start indexed at 0. For example, a 4×4 Hilbert Matrix is

$$H = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{pmatrix}. \tag{C.117}$$

This type of matrix is often used to test numerical linear algebra algorithms since it is known to have some *odd* behaviours … which you will see in a moment.

1. Write code to build a $n \times n$ Hilbert Matrix and call this matrix H. Test your code for various values of n to be sure that it is building the correct matrices.

2. Build a vector of ones called b with code `b = np.ones((n,1))`. We will use b as the right hand side of the system of equations $Hx = b$.

3. Solve the system of equations $Hx = b$ using any technique you like from this chapter.

4. Now let us say that you change the first entry of b by just a little bit, say 10^{-15}. If we were to now solve the equation $Hx_{new} = b_{new}$ what would you expect as compared to solving $Hx = b$.

5. Now let us actually make the change suggested in part (d). Use the code `bnew = np.ones((n,1))` and then `bnew[0] = bnew[0] + 1e-15` to build a new b vector with this small change. Solve $Hx = b$ and $Hx_{new} = b_{new}$ and then compare the maximum absolute difference `np.max(np.abs(x - xnew))`. What do you notice? Make a plot with n on the horizontal axis and the maximum absolute difference on the vertical axis. What does this plot tell you about the solution to the equation $Hx = b$?

6. We know that HH^{-1} should be the identity matrix. As we will see, however, Hilbert matrices are particularly poorly behaved! Write a loop over n that (i) builds a Hilbert matrix of size n, (ii) calculates HH^{-1} (using `np.linalg.inv()` to compute the inverse directly), (iii) calculates the norm of the difference between the identity matrix (`np.identity(n)`) and your calculated identity matrix from part (ii). Finally. Build a plot that shows n on the horizontal axis and the normed difference on the vertical axis. What do you see? What does this mean about the matrix inversion of the Hilbert matrix.

7. There are cautionary tales hiding in this problem. Write a paragraph explaining what you can learn by playing with pathological matrices like the Hilbert Matrix.

Exercise C.83. Now that you have QR and LU code we are going to use both of them! The problem is as follows:
We are going to find the polynomial of degree 4 that best fits the function

$$y = \cos(4t) + 0.1\varepsilon(t) \tag{C.118}$$

at 50 equally spaced points t between 0 and 1. Here we are using $\varepsilon(t)$ as a function that outputs normally distributed random white noise. In Python you will build y as `y = np.cos(4*t) + 0.1*np.random.randn(t.shape[0])`

Build the t vector and the y vector (these are your data). We need to set up the least squares problems $Ax = b$ by setting up the matrix A as we did in the other least squares curve fitting problems and by setting up the b vector using the y data you just built. Solve the problem of finding the coefficients of the best degree 4 polynomial that fits this data. Report the sum of squared error and show a plot of the data along with the best fit curve.

Exercise C.84. Find the largest eigenvalue and the associated eigenvector of the matrix A WITHOUT using `np.linalg.eig()`. (Do not do this by hand either)

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 0 & 1 & 2 \\ 3 & 4 & 5 & 6 \end{pmatrix} \tag{C.119}$$

Exercise C.85. It is possible in a matrix that the eigenvalues λ_1 and λ_2 are equal but with the corresponding eigenvectors not equal. Before you experiment with matrices of this sort, write a conjecture about what will happen to the power method in this case (look back to our proof in Exercise C.58 of how the power method works). Now build several specific matrices where this is the case and see what happens to the power method.

Exercise C.86. Will the power method fail, slow down, or be unaffected if one (or more) of the non-dominant eigenvalues is zero? Give sufficient mathematical evidence or show several numerical experiments to support your answer.

Theorem C.5. *If A is a symmetric matrix with eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_n$ then $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$. Furthermore, the eigenvectors will be orthogonal to each other.*

Exercise C.87 (The Deflation Method)**.** For symmetric matrices we can build an extension to the power method in order to find the second most dominant eigen-pair for a matrix A. Theorem C.5 suggests the following method for finding the second dominant eigen-pair for a symmetric matrix. This method is called the **deflation method**.

- Use the power method to find the dominant eigenvalue and eigenvector.
- Start with a random unit vector of the correct shape.
- Multiplying your vector by A will *pull it toward* the dominant eigenvector. After you multiply, project your vector onto the dominant eigenvector and find the projection error.
- Use the projection error as the new approximation for the eigenvector (Why should we do this? What are we really finding here?)

Note that the deflation method is really exactly the same as the power method with the exception that we orthogonalize at every step. Hence, when you write your code expect to only change a few lines from your power method.

Write a function to find the second largest eigenvalue and eigenvector pair by putting the deflation method into practice. Test your code on a matrix A and compare against Python's `np.linalg.eig()` command. Your code needs to work on symmetric matrices of arbitrary size and you need to write test code that clearly shows the error between your calculated eigenvalue and Python's eigenvalue as well as your calculated eigenvector and 's eigenvector.

To guarantee that you start with a symmetric matrix you can use the following code.

```python
import numpy as np
N = 40
A = np.random.randn(N,N)
# A = np.matrix(A) # No need for matrix class
A = np.transpose(A) @ A # why should this build a symmetric matrix
```

Exercise C.88. (This concept for this problem is modified from (Meerschaert 2013)). The data is taken from NOAA and the National Weather Service with the specific values associated with La Crosse, WI.)

Floods in the Mississippi River Valleys of the upper Midwest have somewhat predictable day-to-day behaviour in that the flood stage today has high predictive power for the flood stage tomorrow. Assume that the flood stages are:

- Stage 0 (Normal): Average daily flow is below 90,000 ft^3/sec (cubic feet per second = cfs). This is the *normal* river level.
- Stage 1 (Action Level): Average daily flow is between 90,000 cfs and 124,000 cfs.
- Stage 2 (Minor Flood): Average daily flow is between 124,000 cfs and 146,000 cfs.
- Stage 3 (Moderate Flood): Average daily flow is between 146,000 cfs and 170,000 cfs.
- Stage 4 (Extreme Flood): Average daily flow is above 170,000 cfs.

The following table shows the probability of one stage transitioning into another stage from one day to the next.

	0 Today	**1 Today**	**2 Today**	**3 Today**	**4 Today**
0 Tomorrow	0.9	0.3	0	0	0
1 Tomorrow	0.05	0.7	0.4	0	0
2 Tomorrow	0.025	0	0.6	0.6	0
3 Tomorrow	0.015	0	0	0.4	0.8
4 Tomorrow	0.01	0	0	0	0.2

Mathematically, if s_k is the state at day k and A is the matrix given in the table above then the difference equation $s_{k+1} = As_k$ shows how a state will transition from day to day. For example, if we are currently in Stage 0 then

$$s_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \tag{C.120}$$

We can interpret this as "*there is a probability of 1 that we are in Stage 0 today and there is a probability of 0 that we are in any other stage today.*"

If we want to advance this model forward in time then we just need to iterate. In our example, the state tomorrow would be $s_1 = As_0$. The state two days from now would be $s_2 = As_1$, and if we use the expression for s_1 we can simplify to $s_2 = A^2 s_0$.

1. Prove that the state at day n is $s_n = A^n s_0$.
2. If n is large then the steady state solution to the difference equation in part (1) is given exactly by the power method iteration that we have studied in this chapter. Hence, as the iterations proceed they will be pulled toward the dominant eigenvector. Use the power method to find the dominant eigenvector of the matrix A.
3. The vectors in this problem are called **probability vectors** in the sense that the vectors sum to 1 and every entry can be interpreted as a probability. Re-scale your answer from part (b) so that we can interpret the entries as probabilities. That is, ensure that the sum of the vector from part (b) is 1.
4. Interpret your answer to part (c) in the context of the problem. Be sure that your interpretation could be well understood by someone that does not know the mathematics that you just did.

Exercise C.89. The LU factorization as we have built it in this chapter is not *smart* about the way that it uses the memory on your computer. In the LU factorization the 1's on the main diagonal do not actually need to be stored since we know that they will always be there. The zeros in the lower triangle of U do not need to be stored either. If you store the upper triangle values in the U matrix on top of the upper triangle of the L matrix then we still store a full matrix for L which contains both L and U simultaneously, but we do not have to store U separately and hence save computer memory. The modifications to the existing code for an LU solve is minimal – every time you call on an entry of the U matrix it is stored in the upper triangle of L instead. Write code to implement this new data storage idea and demonstrate your code on a few examples.

Exercise C.90. In the algorithm that we used to build the QR factorization we built the R matrix as $R = Q^T A$ The trouble with this step is that it fills in a lot of redundant zeros into the R matrix – some of which may not be exactly zero. First explain why this will be the case. Then rewrite your QR factorization code so that the top triangle of R is filled with all of the projections (do this with a double `for` loop). Demonstrate that your code works on a few examples.

C.11. Projects

In this section we propose several ideas for projects related to numerical linear algebra. These projects are meant to be open ended, to encourage creative mathematics, to push your coding skills, and to require you to write and communicate your mathematics.

C.11.1. The Google Page Rank Algorithm

In this project you will discover how the Page Rank algorithm works to give the most relevant information as the top hit on a Google search.

Search engines compile large indexes of the dynamic information on the Internet so they are easily searched. This means that when you do a Google search, you are not actually searching the Internet; instead, you are searching the indexes at Google.

When you type a query into Google the following two steps take place:

1. Query Module: The query module at Google converts your natural language into a language that the search system can understand and consults the various indexes at Google in order to answer the query. This is done to find the list of relevant pages.
2. Ranking Module: The ranking module takes the set of relevant pages and ranks them. The outcome of the ranking is an ordered list of web pages such that the pages near the top of the list are most likely to be what you desire from your search. This ranking is the same as assigning a *popularity score* to each web site and then listing the relevant sites by this score.

This section focuses on the Linear Algebra behind the Ranking Module developed by the founders of Google: Sergey Brin and Larry Page. Their algorithm is called the *Page Rank algorithm*, and you use it every single time you use Google's search engine.

In simple terms: *A webpage is important if it is pointed to by other important pages.*

The Internet can be viewed as a directed graph (look up this term here on Wikipedia) where the nodes are the web pages and the edges are the hyperlinks between the pages. The hyperlinks into a page are called *in links*, and the ones pointing out of a page are called *out links*. In essence, a hyperlink from my page to yours is my endorsement of your page. Thus, a page with more recommendations must be more important than a page with a few links. However, the status of the recommendation is also important.

Let us now translate this into mathematics. To help understand this we first consider the small web of six pages shown in Figure C.3 (a graph of the router level of the internet can be found here). The links between the pages are shown by arrows. An arrow pointing into a node is an *in link* and an arrow pointing out of a node is an *out link*. In Figure C.3, node 3 has three out links (to nodes 1, 2, and 5) and 1 in link (from node 1).

We will first define some notation in the Page Rank algorithm:

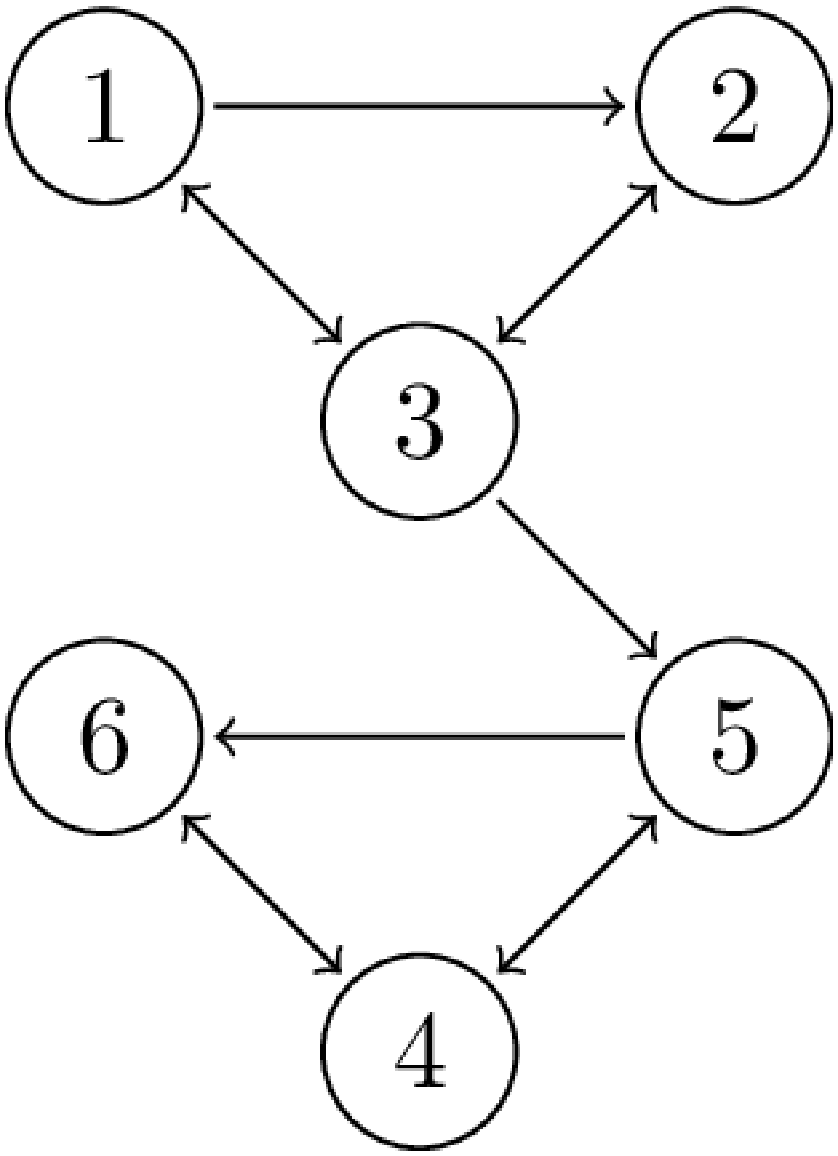


Figure C.3.: Example web graph.

- $|P_i|$ is the number of out links from page P_i
- H is the *hyperlink* matrix defined as

$$H_{ij} = \begin{cases} \frac{1}{|P_j|}, & \text{if there is a link from node } j \text{ to node } i \\ 0, & \text{otherwise} \end{cases} \tag{C.121}$$

  where the "i" and "j" are the row and column indices respectively.
- x is a vector that contains all of the Page Ranks for the individual pages.

The Page Rank algorithm works as follows:

1. Initialize the page ranks to all be equal. This means that our initial assumption is that all pages are of equal rank. In the case of Figure C.3 we would take x_0 to be

$$x_0 = \begin{pmatrix} 1/6 \\ 1/6 \\ 1/6 \\ 1/6 \\ 1/6 \\ 1/6 \end{pmatrix}. \tag{C.122}$$

2. Build the hyperlink matrix.
   As an example we will consider node 3 in Figure C.3. There are three out links from node 3 (to nodes 1, 2, and 5). Hence $H_{13} = 1/3$, $H_{23} = 1/3$, and $H_{53} = 1/3$ and the partially complete hyperlink matrix is

$$H = \begin{pmatrix} - & - & 1/3 & - & - & - \\ - & - & 1/3 & - & - & - \\ - & - & 0 & - & - & - \\ - & - & 0 & - & - & - \\ - & - & 1/3 & - & - & - \\ - & - & 0 & - & - & - \end{pmatrix} \tag{C.123}$$

3. The difference equation $x_{n+1} = Hx_n$ is used to iteratively refine the estimates of the page ranks. You can view the iterations as a person visiting a page and then following a link at random, then following a random link on the next page, and the next, and the next, etc. Hence we see that the iterations evolve exactly as expected for a difference equation.

Iteration	**New Page Rank Estimation**
0	x_0
1	$x_1 = Hx_0$

Iteration	**New Page Rank Estimation**
2	$x_2 = Hx_1 = H^2x_0$
3	$x_3 = Hx_2 = H^3x_0$
4	$x_4 = Hx_3 = H^4x_0$
$\vdots$	$\vdots$
k	$x_k = H^kx_0$

4. When a steady state is reached we sort the resulting vector x_k to give the page rank. The node (web page) with the highest rank will be the top search result, the second highest rank will be the second search result, and so on.

It does not take much to see that this process can be very time consuming. Think about your typical web search with hundreds of thousands of hits; that makes a square matrix H that has a size of hundreds of thousands of entries by hundreds of thousands of entries! The matrix multiplications alone would take many minutes (or possibly many hours) for every search! …but Brin and Page were pretty smart dudes!!

We now state a few theorems and definitions that will help us simplify the iterative Page Rank process.

Theorem C.6. *If A is an $n \times n$ matrix with n linearly independent eigenvectors $v_1, v_2, v_3, \dots, v_n$ and associated eigenvalues $\lambda_1, \lambda_2, \lambda_3, \dots, \lambda_n$ then for any initial vector $x \in \mathbb{R}^n$ we can write A^kx as*

$$A^kx = c_1\lambda_1^kv_1 + c_2\lambda_2^kv_2 + c_3\lambda_3^kv_3 + \cdots c_n\lambda_n^kv_n \tag{C.124}$$

where $c_1, c_2, c_3, \dots, c_n$ are the constants found by expressing x as a linear combination of the eigenvectors.
Note: We can assume that the eigenvalues are ordered such that $|\lambda_1| > |\lambda_2| \ge |\lambda_3| \ge \cdots \ge |\lambda_n|$.

Exercise C.91. Prove the preceding theorem.

Definition C.8. A **probability vector** is a vector with entries on the interval $[0, 1]$ that add up to 1. A **stochastic matrix** is a square matrix whose columns are probability vectors.

Theorem C.7. *If A is a stochastic $n \times n$ matrix then A will have n linearly independent eigenvectors. Furthermore, the largest eigenvalue of a stochastic matrix will be $\lambda_1 = 1$ and the smallest eigenvalue will always be non-negative: $0 \le |\lambda_n| < 1$.*

Some of the following tasks will ask you to prove *a statement or a theorem. This means to clearly write all of the logical and mathematical reasons why the statement is true. Your proof should be absolutely crystal clear to anyone with a similar mathematical background ...if you are in doubt then have a peer from a different group read your proof to you .*

Exercise C.92. Finish writing the hyperlink matrix H from Figure C.3.

Exercise C.93. Write code to implement the iterative process defined previously. Make a plot that shows how the rank evolves over the iterations.

Exercise C.94. What must be true about a collection of n pages such that an $n \times n$ hyperlink matrix H is a stochastic matrix.

Exercise C.95. The statement of the next theorem is incomplete, but the proof is given to you. Fill in the blank in the statement of the theorem and provide a few sentences supporting your answer.

Theorem C.8. *If A is an $n \times n$ stochastic matrix and x_0 is some initial vector for the difference equation $x_{n+1} = Ax_n$, then the steady state vector is*

$$x_{equilib} = \lim_{k \to \infty} A^k x_0 = \underline{\hspace{3cm}}. \tag{C.125}$$

Proof:

First note that A is an $n \times n$ stochastic matrix so from Theorem C.7 we know that there are n linearly independent eigenvectors. We can then substitute the eigenvalues from Theorem C.7 in Theorem C.6. Noting that if $0 < \lambda_j < 1$ we have $\lim_{k \to \infty} \lambda_j^k = 0$ the result follows immediately.

Exercise C.96. Discuss how Theorem C.8 greatly simplifies the PageRank iterative process described previously. In other words: there is no reason to iterate at all. Instead, just find ... what?

Exercise C.97. Now use the previous two problems to find the resulting PageRank vector from the web in Figure C.3? Be sure to rank the pages in order of importance. Compare your answer to the one that you got in problem 2.

Exercise C.98. Consider the web in Figure C.4.

1. Write the H matrix and find the initial state x_0,
2. Find steady state PageRank vector using the two different methods described: one using the iterative difference equation and the other using Theorem C.8 and the dominant eigenvector.
3. Rank the pages in order of importance.

Exercise C.99. One thing that we did not consider in this version of the Google Page Rank algorithm is the random behaviour of humans. One, admittedly slightly naive, modification that we can make to the present algorithm is to assume that the person surfing the web will randomly jump to any other page in the web at any time. For example, if someone is on page 1 in Figure C.4 then they could randomly jump to any page 2 - 8. They also have links to pages 2, 3, and 7. That is a total of 10 possible next steps for the web surfer. There is a 2/10 chance of heading to page 2. One of those is following the link from page 1 to page 2 and the other is a random jump to page 2 without following the link. Similarly, there is a 2/10 chance of heading to page 3, 2/10 chance of heading to page 7, and a 1/10 chance of randomly heading to any other page.

Implement this new algorithm, called the *random surfer algorithm*, on the web in Figure C.4. Compare your ranking to the non-random surfer results from the previous problem.

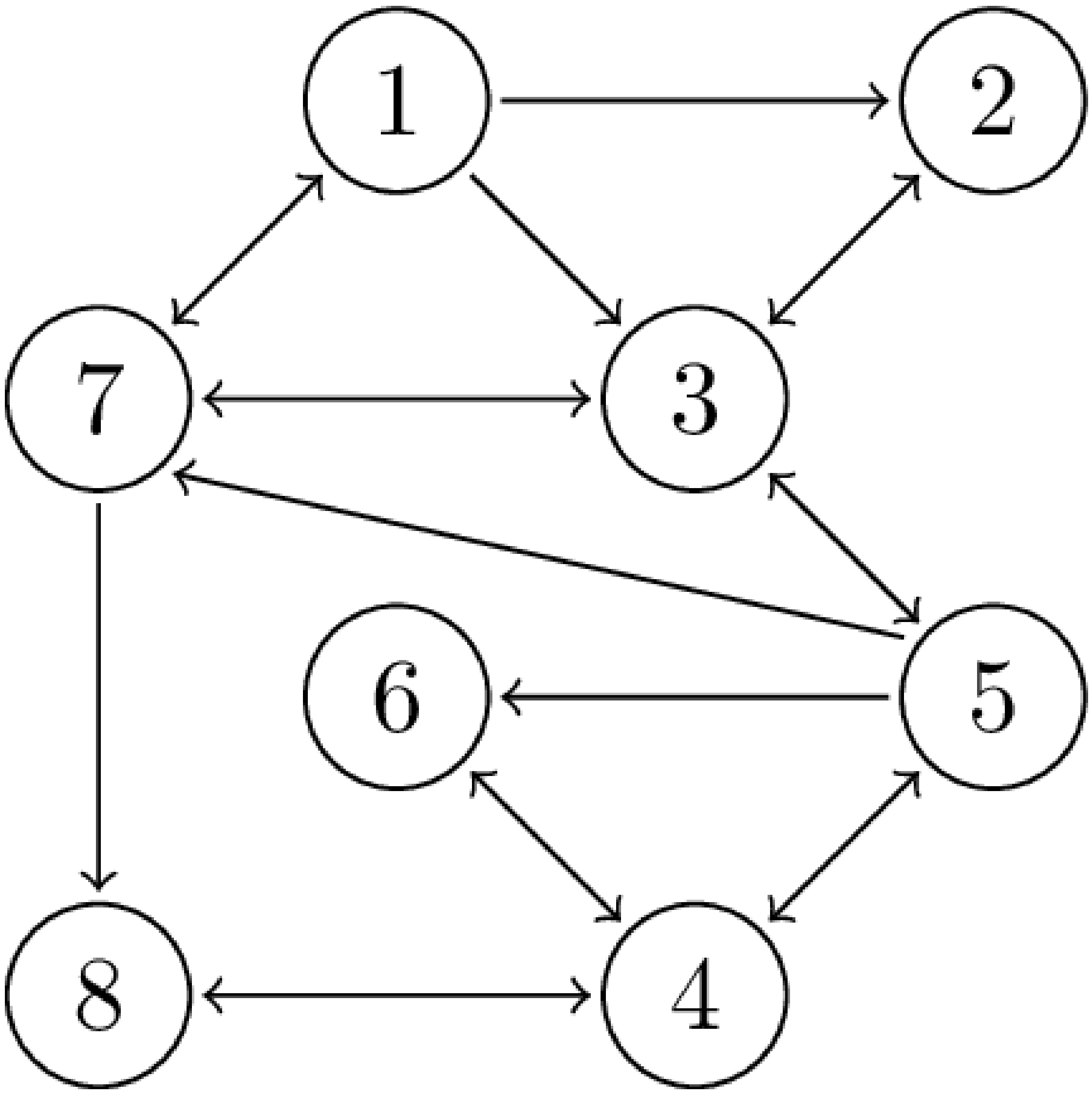


Figure C.4.: A second example web graph.

C.11.2. Alternative Methods For Solving $Ax = b$

Throughout most of the linear algebra chapter we have studied ways to solve systems of equations of the form $Ax = b$ where A is a square $n \times n$ matrix, $x \in \mathbb{R}^n$, and $b \in \mathbb{R}^n$. We have reviewed by-hand row reduction and learned new techniques such as the LU decomposition and the QR decomposition – all of which are great in their own right and all of which have their shortcomings.

Both LU and QR are great solution techniques and they generally work very very well. However (no surprise), we can build algorithms that will *usually* be faster!

In the following new algorithms we want to solve the linear system of equations

$$Ax = b \tag{C.126}$$

but in each we will do so iteratively by applying an algorithm over and over until the algorithm converges to an approximation of the solution vector x. Convergence here means that $\|A\ x - b\|$ is less than some pre-determined tolerance.

Method 1: Start by "factoring"[7] the matrix A into $A = L + U$ where L is a lower-triangular matrix and U is an upper-triangular matrix. Take note that this time we will not force the diagonal entries of L to be 1 like we did in the classical LU factorization . The U in the factorization $A = L + U$ is an upper-triangular matrix where the entries on the main diagonal are exactly 0.

Specifically,

$$\begin{aligned} A = L + U = &\begin{pmatrix} a_{00} & 0 & 0 & \cdots & 0 \\ a_{10} & a_{11} & 0 & \cdots & 0 \\ a_{20} & a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n0} & a_{n1} & a_{n2} & \cdots & a_{n-1,n-1} \end{pmatrix} \\ &+ \begin{pmatrix} 0 & a_{01} & a_{02} & \cdots & a_{0,n-1} \\ 0 & 0 & a_{12} & \cdots & a_{1,n-1} \\ 0 & 0 & 0 & a_{23} & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & & 0 \end{pmatrix}. \end{aligned} \tag{C.127}$$

As an example,

$$\begin{pmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \\ 8 & 9 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 5 & 6 & 0 \\ 8 & 9 & 1 \end{pmatrix} + \begin{pmatrix} 0 & 3 & 4 \\ 0 & 0 & 7 \\ 0 & 0 & 0 \end{pmatrix}. \tag{C.128}$$

[7]Technically speaking we should not call this a "factorization" since we have not split the matrix A into a product of two matrices. Instead we should call it a "partition" since in number theory we call the process of breaking an integer into the sum of two integers is called a "partition." Even so, we will still use the word factorization here for simplicity.

After factoring the system of equations can be rewritten as

$$Ax = b \implies (L+U)x = b \implies Lx + Ux = b. \tag{C.129}$$

Moving the term Ux to the right-hand side gives $Lx = b - Ux$, and if we solve for the unknown x we get $x = L^{-1}(b - Ux)$.

Of course we would never (*ever*!) actually compute the inverse of L, and consequently we have to do something else in place of the matrix inverse. Stop and think here for a moment. We have run into this problem earlier in this chapter and you have some code that you will need to modify for this job (but take very careful note that the L matrix here does not quite have the same structure as the L matrix we used in the past). Moreover, notice that we have the unknown x on both sides of the equation. Initially this may seem like nonsense, but if we treat this as an iterative scheme by first making a **guess** about x and then iteratively find better approximations of solutions via the difference equation

$$x_{k+1} = L^{-1}(b - Ux_k) \tag{C.130}$$

we may, under moderate conditions on A, quickly be able to approximate the solution to $Ax = b$. The subscripts in the iterative scheme represent the iteration number. Hence,

$$x_1 = L^{-1}(b - Ux_0) \tag{C.131}$$

$$x_2 = L^{-1}(b - Ux_1) \tag{C.132}$$

$$\vdots \tag{C.133}$$

What we need to pay attention to is that the method is not guaranteed to converge to the actual solution to the equation $Ax = b$ unless some conditions on A are met, and you will need to experiment with the algorithm to come up with a conjecture about the appropriate conditions.

Method 2: Start by factoring the matrix A into $A = L + D + U$ where L is strictly lower-triangular (0's on the main diagonal and in the entire upper triangle), D is a diagonal matrix, and U is a strictly upper-triangular matrix (0's on the main diagonal and in the entire lower triangle). In this new factorization, the diagonal matrix D simply contains the entries from the main diagonal of A. The L matrix is the lower triangle of A, and the U matrix is the upper triangle of A.

Considering the system of equations $Ax = b$ we get

$$(L + D + U)x = b \tag{C.134}$$

and after simplifying, rearranging, and solving for x we get $x = D^{-1}(b - Lx - Ux)$. A moment's reflection should reveal that the inverse of D is really easy to find (no heavy-duty linear algebra necessary) if some mild conditions on the diagonal entries of A are met. Like before there is an x on both sides of the equation, but if we again make the algorithm iterative we can get successive approximations of the solution with

$$x_{k+1} = D^{-1}(b - Lx_k - Ux_k). \tag{C.135}$$

Your Tasks:

1. Pick a small (larger than 3×3) matrix and an appropriate right-hand side b and work each of the algorithms by hand. You do not need to write this step up in the final product, but this exercise will help you locate where things may go wrong in the algorithms and what conditions we might need on A in order to get convergent sequences of approximate solutions.

2. Build Python functions that accept a square matrix A and complete the factorizations $A = L + U$ and $A = L + D + U$.

3. Build functions to implement the two methods and then demonstrate that the methods work on a handful of carefully chosen test examples. As part of these functions you need to build a way to deal with the matrix inversions as well as build a stopping rule for the iterative schemes. Hint: You should use a `while` loop with a proper logical condition. Think carefully about what we are finding at each iteration and what we can use to check our accuracy at each iteration. It would also be wise to write your code in such a way that it checks to see if the sequence of approximations is diverging.

4. Discuss where each method might fail and then demonstrate the possible failures with several carefully chosen examples. Stick to small examples and work these out by hand to clearly show the failure.

5. Iterative methods such as these will produce a sequence of approximations, but there is no guarantee that either method will actually produce a convergent sequence. Experiment with several examples and propose a condition on the matrix A which will likely result in a convergent sequence. Demonstrate that the methods fail if your condition is violated and that the methods converge if your condition is met. Take care that it is tempting to think that your code is broken if it does not converge. The more likely scenario is that the problem that you have chosen to solve will result in a non-convergent sequence of iterations, and you need to think and experiment carefully when choosing the example problems to solve. One such convergence criterion has something to do with the diagonal entries of A relative to the other entries, but that does not mean that you should not explore other features of the matrices as well (I cannot give you any more hints than that). This task is not asking for a proof; just a conjecture and convincing numerical evidence that the conjecture holds. The actual proofs are beyond the scope of this project and this course.

6. Devise a way to demonstrate how the time to solve a large linear system $Ax = b$ compares between our two new methods, the LU algorithm, and the QR algorithm that we built earlier in the chapter. Conclude this demonstration with appropriate plots and ample discussion.

You need to do this project without the help of your old buddy Google. All code must be originally yours or be modified from code that we built in class. You can ask Google how Python works with matrices and the like, but searching directly for the algorithms (which are actually well-known, well-studied, and named algorithms) is not allowed.

Finally, solving systems of equations with the `|np.linalg.solve()` command can only be done to verify or check your answer(s).

Acton, Forman S. 1990. *Numerical Methods That Work.* 1St Edition edition. Washington, D.C: The Mathematical Association of America.

Burden, Richard L., and J. Douglas Faires. 2010. *Numerical Analysis.* 9th ed. Brooks Cole.

Butcher, J. C. 2016. *Numerical Methods for Ordinary Differential Equations.* Third edition. Wiley. https://yorsearch.york.ac.uk/permalink/f/1kq3a7l/44YORK_ALMA_DS51336126850001381.

Greenbaum, Anne, and Tim P. Chartier. 2012. *Numerical Methods: Design, Analysis, and Computer Implementation of Algorithms.* Princeton University Press.

Kincaid, D. R., and E. W. Cheney. 2009. *Numerical Analysis: Mathematics of Scientific Computing.* Pure and Applied Undergraduate Texts. American Mathematical Society.

Meerschaert, Mark. 2013. *Mathematical Modeling.* 4th edition. Amsterdam ; Boston: Academic Press.

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press. https://numerical.recipes/.

"Project Euler." n.d. Accessed December 14, 2023. https://projecteuler.net/.