

Numerical Analysis 2025

A Learning Guide

Gustav Delius Eric Sullivan

2025-05-19

Table of contents

Introduction	3
What Is Numerical Analysis?	3
Example from Algebra	3
Example from Calculus	4
Example from Differential Equations	6
Reasons to study Numerical Analysis	10
The Inquiry-Based Approach	11
How this module works	12
Assessment	13
Textbooks	14
Your jobs	14
1 Essential Python	16
1.1 Why Python?	16
1.2 Google Colab	17
1.2.1 The use of AI	18
1.3 Python Programming Basics	19
1.3.1 Variables	20
1.3.2 Indexing and Lists	21
1.3.3 List Operations	23
1.3.4 Tuples	25
1.3.5 Control Flow: Loops and If Statements	26
1.3.6 Functions	32
1.3.7 Lambda Functions	35
1.3.8 Packages	36
1.4 Numerical Python with NumPy	38
1.4.1 Numpy Arrays, Array Operations, and Matrix Operations	39
1.4.2 <code>arange</code> , <code>linspace</code> , <code>zeros</code> , <code>ones</code> , and <code>meshgrid</code>	44
1.5 Plotting with Matplotlib	46
1.5.1 Basics with <code>plt.plot()</code>	47
1.5.2 Subplots	52
1.5.3 Logarithmic Scaling with <code>semilogy</code> , <code>semilogx</code> , and <code>loglog</code>	54
1.6 Dataframes with Pandas	57
1.7 Problems	57

2 Numbers	61
2.1 Binary Numbers	62
2.2 Floating Point Numbers	66
2.3 Rounding Errors	69
2.4 Loss of Significant Digits	71
2.5 Problems	73
3 Functions	76
3.1 Polynomial Approximations	77
3.2 Truncation Error	84
3.3 Problems	89
4 Non-linear Equations	92
4.1 Introduction to Numerical Root Finding	92
4.2 The Bisection Method	95
4.2.1 Intuition	95
4.2.2 Implementation	97
4.2.3 Analysis	99
4.3 Fixed Point Iteration	110
4.4 Newton's Method	115
4.4.1 Intuition	115
4.4.2 Implementation	118
4.4.3 Failures	119
4.4.4 Rate of Convergence	125
4.5 The Secant Method	126
4.5.1 Intuition and Implementation	126
4.5.2 Analysis	128
4.6 Order of Convergence	129
4.6.1 Definition	129
4.6.2 Fixed Point Iteration	130
4.6.3 Newton's Method	131
4.6.4 Secant Method	131
4.7 Algorithm Summaries	133
4.8 Problems	134
4.9 Projects	138
4.9.1 Basins of Attraction	138
4.9.2 Artillery	140
5 Derivatives	142
5.1 Finite Differences	142
5.1.1 The First Derivative	142
5.1.2 Truncation error	146
5.1.3 Efficient Coding	150

5.1.4	A Better First Derivative	153
5.1.5	The Second Derivative	157
5.2	Automatic Differentiation	160
5.2.1	Forward mode AD	163
5.2.2	Reverse mode AD	168
5.3	Algorithm Summaries	173
5.4	Problems	174
6	Integrals	176
6.1	Riemann Sums	176
6.2	Trapezoidal Rule	180
6.3	Simpsons Rule	182
6.4	Algorithm Summaries	186
6.5	Problems	186
6.6	Projects	188
6.6.1	Higher Order Integration	188
6.6.2	Dam Integration	188
6.6.3	WHO Data Integration	189
7	Optima	190
7.1	Single Variable Optimization	192
7.1.1	Golden Section Search	193
7.1.2	Gradient Descent	195
7.2	Multivariable Optimization	199
7.2.1	Gradient Descent Algorithm	200
7.3	Optimization with SciPy	202
7.4	Algorithm Summaries	203
7.5	Problems	203
7.6	Projects	205
7.6.1	Edge Detection in Images	205
8	Ordinary Differential Equations	209
8.1	Euler's Method	210
8.2	Higher-order equations and systems of equations	217
8.3	The Midpoint Method	221
8.4	Searching for a better Method	227
8.5	Runge-Kutta Method	232
8.6	The Backwards Euler Method	234
8.7	Numerical Instabilities	237
8.8	Stiff Equations	240
8.9	Algorithm Summaries	241
8.10	Problems	242

8.11 Projects	250
8.11.1 The COVID-19 Pandemic	250
8.11.2 Pain Management	251
8.11.3 The H1N1 Virus	252
8.11.4 The Artillery Problem	253
9 Partial Differential Equations	255
9.1 Intro to PDEs	255
9.2 The Heat Equation	257
9.2.1 In One Spatial Dimensions	257
9.2.2 Different Boundary Conditions	265
9.2.3 In Two Spatial Dimensions	265
9.2.4 Variations on the Heat Equation	270
9.2.5 Implicit Methods	272
9.2.6 Stability	277
9.3 The Wave Equation	280
9.4 The Travelling Wave Equation	284
9.5 The Laplace and Poisson Equations	289
9.6 Algorithm Summaries	298
9.7 Problems	299
9.8 Projects	303
9.8.1 Hunting and Diffusion	303
9.8.2 Heating Adobe Houses	304

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 ($+, -, \times, /$) 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 2 and 2.5.

```

# plot the function cos(x) and the line y=x
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(1, 4, 100)
plt.plot(x, np.log(x), label="log(x)")
plt.plot(x, np.sin(x), label="sin(x)")
plt.legend()
plt.grid(True)
plt.show()

```

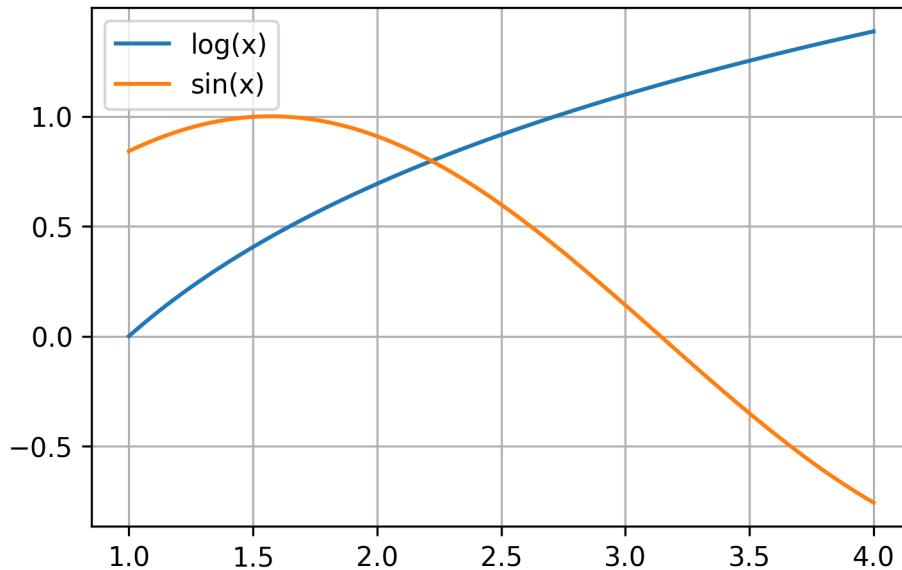


Figure 1: The functions $\log(x)$ and $\sin(x)$ intersect at exactly one point, giving the solution to the equation $\log(x) = \sin(x)$.

Example from Calculus

What if we want to evaluate

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

```

import matplotlib.pyplot as plt
import numpy as np

```

```

def f(x):
    return np.sin(x**2)

a = 0
b = np.pi
n = 1000 # Number of points for numerical integration

x = np.linspace(a, b, n)
y = f(x)

# Calculate the numerical integral using the trapezoidal rule
integral = np.trapz(y, x)

# Shade the positive and negative regions differently
plt.fill_between(x, y, where=y>=0, color='green', alpha=0.5, label="Positive")
plt.fill_between(x, y, where=y<0, color='red', alpha=0.5, label="Negative")

# Plot the curve
plt.plot(x, y, color='black', label=r"\sin(x^2)")

# Set labels and title
plt.xlabel("x")
plt.ylabel("y")
plt.title(r"Integral of \sin(x^2) from 0 to \pi")

# Add legend
plt.legend()

# Show the plot
plt.grid()
plt.show()

```

/tmp/ipykernel_13380/1036798515.py:15: DeprecationWarning:

`trapz` is deprecated. Use `trapezoid` instead, or one of the numerical integration functions

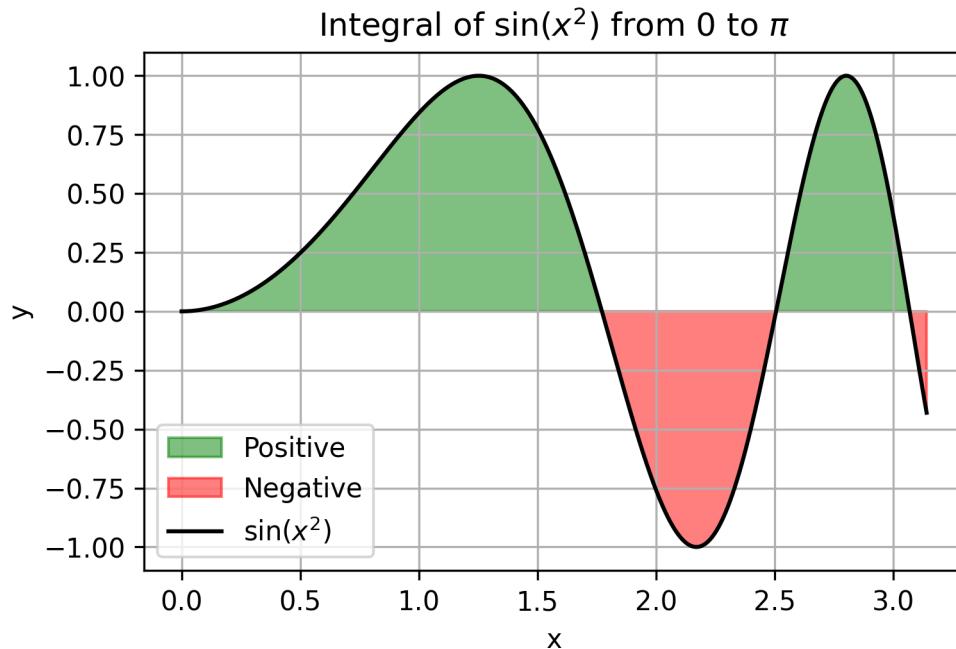


Figure 2: Visual representation of the integral of $\sin(x^2)$ from 0 to π .

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

```
# Use Simpson's rule to approximate the integral of sin(x^2) from 0 to pi
from scipy.integrate import simpson
simpson(y, x = x)

np.float64(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.

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 $y(0) = 0.1$:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp

def f(t, y):
    return np.sin(y**2) + t

# Initial condition
y0 = 0.1

# Time span for the solution
t_span = (0, 2.5)

# Solve the differential equation using SciPy's solver
sol = solve_ivp(f, t_span, [y0], max_step=0.1, dense_output=True)

# Extract the time values and solution
t = sol.t
y = sol.sol(t)[0]

# Plot the numerical solution
plt.plot(t, y)

# Labels and title
plt.xlabel('t')
plt.ylabel('y')

# Show the plot
plt.grid(True)
plt.show()
```

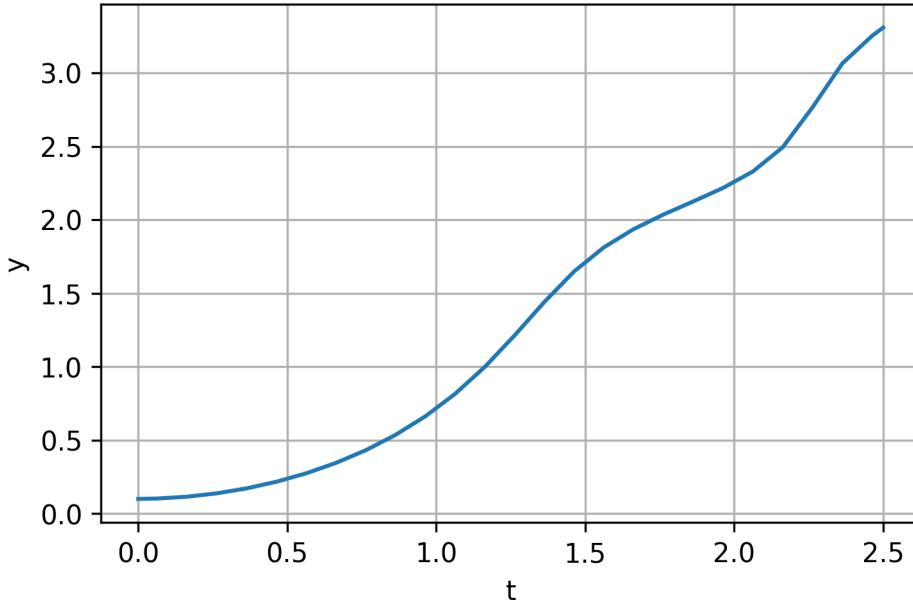


Figure 3: Plot of numerical solution of $dy/dt = \sin(y^2) + t$ with $y(0) = 0.1$.

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:

```

import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# SIR model differential equations
def sir_model(y, t, N, beta, gamma):
    S, I, R = y
    dSdt = -beta * S * I / N
    dIdt = beta * S * I / N - gamma * I
    dRdt = gamma * I
    return dSdt, dIdt, dRdt

# Total population, N
N = 1000
# Initial number of infected and recovered individuals
I0, R0 = 1, 0
# Everyone else is susceptible to infection initially
S0 = N - I0 - R0

```

```

# Contact rate, beta, and mean recovery rate, gamma, (in 1/days)
beta, gamma = 0.25, 1./20
# A grid of time points (in days)
t = np.linspace(0, 160, 160)

# Initial conditions vector
y0 = S0, I0, R0
# Integrate the SIR equations over the time grid, t
ret = odeint(sir_model, y0, t, args=(N, beta, gamma))
S, I, R = ret.T

# Plot the data on three separate curves for S(t), I(t) and R(t)
plt.figure(figsize=(10,6))
plt.plot(t, S, 'b', alpha=0.7, linewidth=2, label='Susceptible')
plt.plot(t, I, 'y', alpha=0.7, linewidth=2, label='Infected')
plt.plot(t, R, 'g', alpha=0.7, linewidth=2, label='Recovered')
plt.xlabel('Time /days')
plt.ylabel('Number (1000s)')
plt.ylim(0, N)
plt.title('SIR Model Simulation')
plt.legend()
plt.show()

```

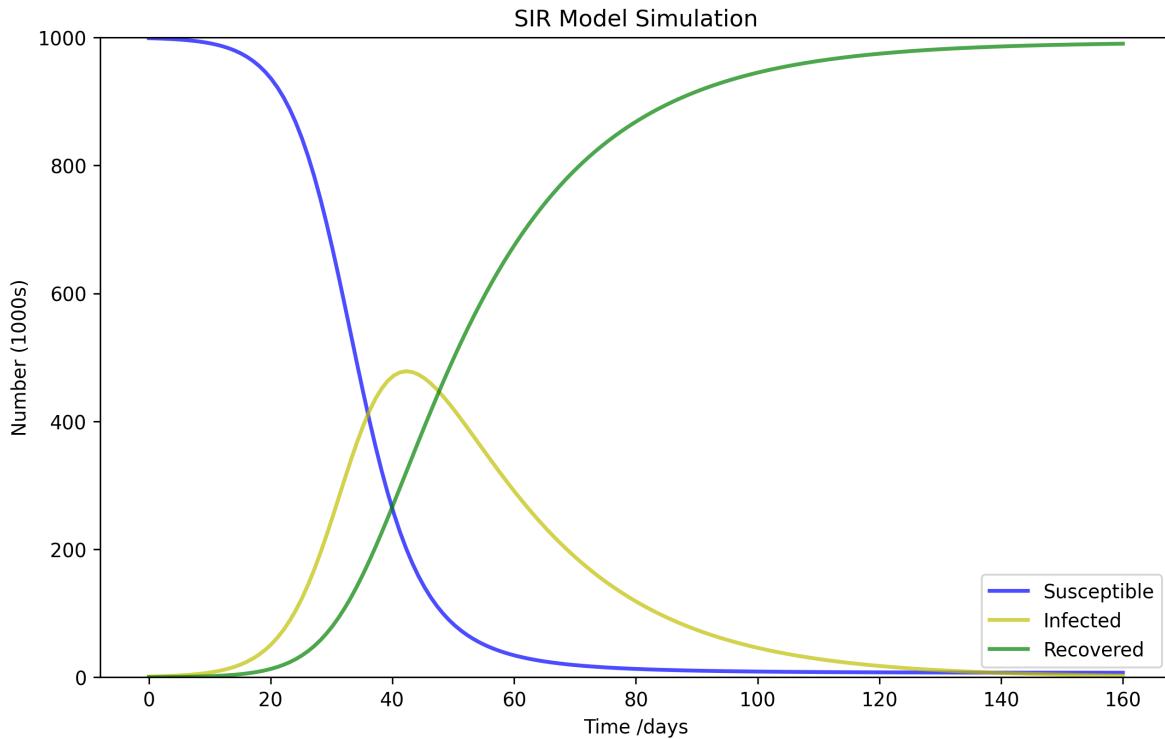


Figure 4: Plot of a numerical solution of the SIR model

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?

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.

The Inquiry-Based Approach

This material is written with an Inquiry-Based Learning (IBL) flavor. In that sense, these notes are not a *traditional textbook* containing all of the expected theorems, proofs, code, examples, and exposition. You are encouraged to work through exercises, problems and projects, present your findings, and work together when appropriate.

In our first session we will start off right away with an exercise designed for groups, discussion, disagreement, and deep critical thinking. This exercise is inspired by Dana Ernst's first day IBL activity titled: [Setting the Stage](#).

Exercise 0.1.

- Get in groups of size 3 or 4.
- Introduce yourself to each other.
- For each of the questions that follow I will ask you to:
 1. **Think** about a possible answer on your own
 2. **Discuss** your answers with the rest of the group
 3. **Share** a summary of each group's discussion

Questions:

Question 1: What are the goals of a university education?

Question 2: How does a person learn something new?

Question 3: What do you reasonably expect to remember from your courses in 20 years?

Question 4: What is the value of making mistakes in the learning process?

Question 5: How do we create a safe environment where risk taking is encouraged and productive failure is valued?

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 your lecturer.

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 .

At the end of each chapter there is a section entitled “**Problems**” that contains additional exercises aimed at consolidating your new understanding and skills. Of these you should aim to do as many as you can but you will not have time to do them all. As the module progresses I will give advice on which of those problems to attack. 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.

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 your lecturer at gustav.delius@york.ac.uk who will be very interested, also after the end of the module.

If you notice any mistakes or unclear things in the learning guide, [please let me know](#). Many thanks go to Ben Mason and Toby Cheshire for the corrections they had sent in last year.

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 while others are coding exercises and some require both writing or sketching and coding. The two notebooks will be linked through the numbering of the exercises.

For the coding notebook I highly recommend using **Google Colab** (or Jupyter Notebook). This will be discussed more in Chapter 1 that introduces Python. Most students find it easiest to have one dedicated Colab notebook (or Jupyter 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 **notebook computer**, bring it along to the class sessions. However this is not a requirement. Your lecturer 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 10 minutes to complete the first two quizzes and 16 minutes each to complete the next 8 quizzes in order to give you a large safety margin and remove stress. 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 and another one at the start of the practical in week 2.

During the assessment quizzes you will be required to work exclusively on a classroom PC rather than your own machine. You will do your work in a Colab notebook in which the AI features have been switched off. You can find more info on the use of Colab notebooks in this module in the [Essential Python](#) chapter of the Numerical Analysis Learning Guide.

While working on the quiz on the classroom PC 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 online notebooks 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.

There will be a practice assessment quiz in week 1 that will not count for anything.

The **60% final exam** will be a 2 hour exam of the usual closed-book form in an exam room during the exam period. 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!
-

© Gustav Delius. Some Rights Reserved.

This learning guide, adapted from the original text by Eric Sullivan, is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. You may copy, distribute, display, remix, rework, and perform this copyrighted work, as long as you give credit to both Gustav Delius for the adaptations and Eric Sullivan for the original work.

Please attribute the original work to Eric Sullivan, formerly Mathematics Faculty at Carroll College, esullivan@carroll.edu, and the adapted work to Gustav Delius, Department of Mathematics, University of York, gustav.delius@york.ac.uk.

The original work by Eric Sullivan is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit <https://creativecommons.org/licenses/by-nc-sa/4.0/>. The adaptations by Gustav Delius are also published under the same Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

For inquiries regarding the use of this learning guide, please contact gustav.delius@york.ac.uk.

1 Essential Python

Simple is better than complex.

—[Guido van Rossum](#)

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

For some of you this material may not be new. For example you may have seen the [Python Programming](#) material that was shared with you in the “2nd year Info” site on Moodle. 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 alright.

If you are new to Python, don’t feel that you need to work through this chapter in one go. Instead, spread the work over the first two weeks of the course and intermingle it with your work on the next two chapters. There is a lot of material in this chapter. Do not feel that you need to learn it all by heart. 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.

1.1 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 chapter just to get going with the language.

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 every scientist 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!

1.2 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. Unless you are already familiar with using Python on your own computer, I highly recommend that you use the Google Colab notebook tool for writing 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](#).

Exercise 1.1. 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 1.2.

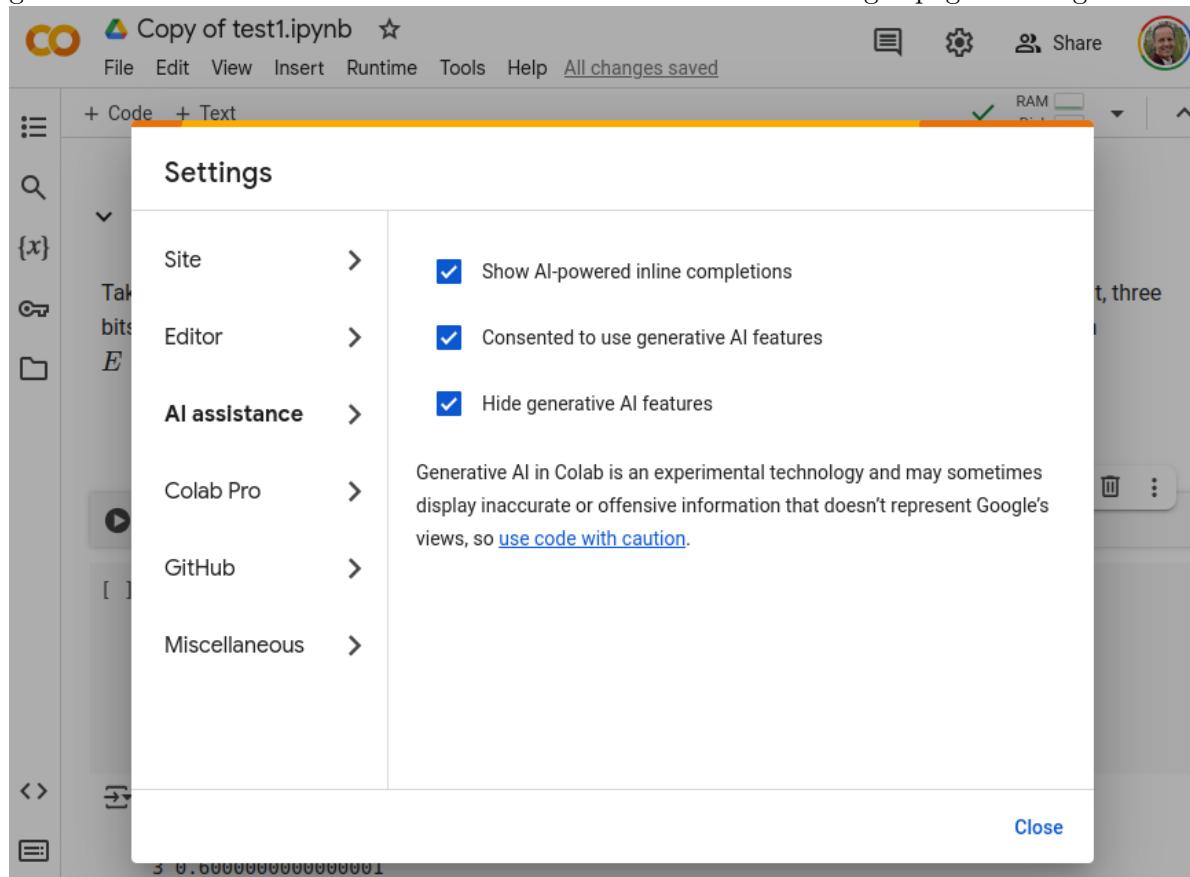
- Click on this [link to a Colab notebook](#). It should open it in Colab.
 - Save a **copy** of it to your Google Drive. You need a copy because you will not have permission to edit the original.
 - Follow the instructions in the notebook.
 - Share that notebook with your lecturer gustav.delius@york.ac.uk, giving him at least “Commenter” privileges.
-

1.2.1 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.



1.3 Python Programming Basics

If you are already very practised in using Python then you can jump straight to Section 1.7 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. 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 in this guide when you hover over that code block.

1.3.1 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.

```
# 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, \wedge , 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 1.3. 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.

1.3.2 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 1.1 (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

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

- Output the list

`MyList`

- Select only the element with index 0.

`MyList[0]`

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

`MyList[:2]`

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

`MyList[-1]`

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

`MyList[1:5]`

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

`MyList[0::2]`

- Select the last three elements of `MyList`

`MyList[-3:]`

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

Example 1.2. 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 is common in many scientific disciplines and is just another thing that we will need to get used to in Python. (For example, j is commonly used as the symbol for the imaginary unit $\sqrt{-1}$ in electrical engineering since i is the symbol commonly used for electric current, and using i for both would be problematic).

```
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 1.4. In this exercise you will put your new list skills into practice.

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

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89. \quad (1.1)$$

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.
-

1.3.3 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, `MyList.append(7)` will append the number 7 to the list `MyList`. We say that `append` is a “method” of the list `MyList`. This is a common programming feature in Python and we will use it often.

Example 1.3. The `.append` method can be used to append an element to the end of a list.

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

Example 1.4. The `.remove` method can be used to remove an element from a list.

```
# Let us remove the 3
MyList.remove(3)
print(MyList)
```

Example 1.5. 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
MyList.insert(0,'A')
# insert the letter `B` at the spot with index 3
MyList.insert(3,'B')
# remember that index 3 means the fourth spot in the list
print(MyList)
```

Exercise 1.5. 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.
-

1.3.4 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 1.6.

```
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 cover in this module. These include dictionaries and sets. We will not cover these because they are not necessary for our work in numerical analysis. We are trying to keep things simple. If you are interested in learning more about these data structures, you can find a lot of information about them in the Python documentation.

1.3.5 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 indentation 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.

1.3.5.1 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 is also nice numerically for building sequences, summing series, or just checking lots of examples. The following are several examples of Python for loops. 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 indentation. There is no need for an **end** command or a curly brace. All of the control statements in Python are white-space-dependent.

Example 1.7. 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.

Example 1.8. 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 1.9. 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 1.10. 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 1.11. As an alternative to the code from the previous example we can pre-allocate the memory in an array 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 1.6. 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 1.7. 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.

1.3.5.2 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 1.12. 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.

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

Example 1.13. 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

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

1.3.5.3 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 1.14. 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 1.15. 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.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 < 0.5:
    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 1.16. 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

```
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 1.9. Write code to give the Collatz Sequence

$$x_{n+1} = \begin{cases} x_n/2, & x_n \text{ is even} \\ 3x_n + 1, & \text{otherwise} \end{cases} \quad (1.2)$$

starting with a positive integer of your choosing. The sequence will converge¹ 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 $x_n/2$.

¹Actually, 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 1.27

1.3.6 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 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 1.17. 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 1.10. 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 $n = 41$?

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

Example 1.18. 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!

```
def Fact(n):
    if n==0:
        return(1)
    else:
        return(n*Fact(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. So let us use it to make a list of the first several factorials. Note the use of a for loop in the following code.

```
FactList = [Fact(n) for n in range(0,10)]
FactList
```

Example 1.19. For this next example let us define the sequence

$$x_{n+1} = \begin{cases} 2x_n, & x_n \in [0, 0.5] \\ 2x_n - 1, & x_n \in (0.5, 1] \end{cases} \quad (1.3)$$

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

```
# Define the function
def MySeq(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(MySeq(x[-1]))
print(x)
```

Example 1.20. A fun way to approximate the square root of two is to start with any positive real number and iterate over the sequence

$$x_{n+1} = \frac{1}{2}x_n + \frac{1}{x_n} \quad (1.4)$$

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.

```
from math import sqrt
def f(x):
    return(0.5*x + 1/x)
x = 1.1 # arbitrary starting point
print("approximation \t\t exact \t\t abs error")
while abs(x-sqrt(2)) > 10**(-8):
    x = f(x)
    print(x, sqrt(2), abs(x - sqrt(2)))
```

Exercise 1.11. 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

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{S}{x_n} \right) \quad (1.5)$$

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 2-20), and write a ‘while’ loop inside your ‘for’ loop to do the iterations.

1.3.7 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.

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 1.12. Go back to Exercise 1.10 and repeat this exercise using a `lambda` function.

Exercise 1.13. Go back to Exercise 1.11 and repeat this exercise using a `lambda` function.

1.3.8 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 1.21. 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 1.22. 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 1.23. 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.

```
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 1.24 (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.

```
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 1.14. 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).

1.4 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 1.25. 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.

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

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.

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

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

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!

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

Exercise 1.15. 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 functions with the prefix “`np`” in order to call on the `numpy` functions. Let us first see what is inside the package with the code `print(dir(np))` after importing `numpy as 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.)

1.4.1 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

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad v = \begin{pmatrix} 5 \\ 6 \end{pmatrix} \quad \text{and} \quad w = v^T = (5 \quad 6). \quad (1.6)$$

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 1.26 (`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 1.27 (`.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 1.28 (`.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 1.29. Let us read the top left and bottom right entries of the matrix A .

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

Example 1.30. Let us read the first row from that matrix A .

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

Example 1.31. Let us read the second column from the matrix A .

```
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 1.32. 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.

```
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 1.33. 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 1.34 (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 1.35 (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 1.36 (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 2.

Example 1.37 (Inverse). In the `linalg` subpackage there is also a function for taking the inverse of a matrix.

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

We can check that we get the identity matrix back:

```
A @ Ainv
```

Exercise 1.16. Now that we can do some basic linear algebra with `numpy` it is your turn. Define the matrix B and the vector u as

$$B = \begin{pmatrix} 1 & 4 & 8 \\ 2 & 3 & -1 \\ 0 & 9 & -3 \end{pmatrix} \quad \text{and} \quad u = \begin{pmatrix} 6 \\ 3 \\ -7 \end{pmatrix}. \quad (1.7)$$

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
-

1.4.2 `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 1.38. 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 1.39. 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 1.40. 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
```

Example 1.41. The `np.ones` function builds an array of ones.

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

Example 1.42. 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 1.17. 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 5th power of this matrix.
-

1.5 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.

1.5.1 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 1.43 (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`).

```
import numpy as np
import matplotlib.pyplot as plt
```

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.

```
# 100 equally spaced points from 0 to 2pi
x = np.linspace(0,2*np.pi, 100)
y = np.sin(x)
```

- 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.

```
plt.plot(x,y, 'green', label='The Sine Function')
plt.grid()
plt.legend()
plt.xlabel("x axis")
plt.ylabel("y axis")
plt.show()
```

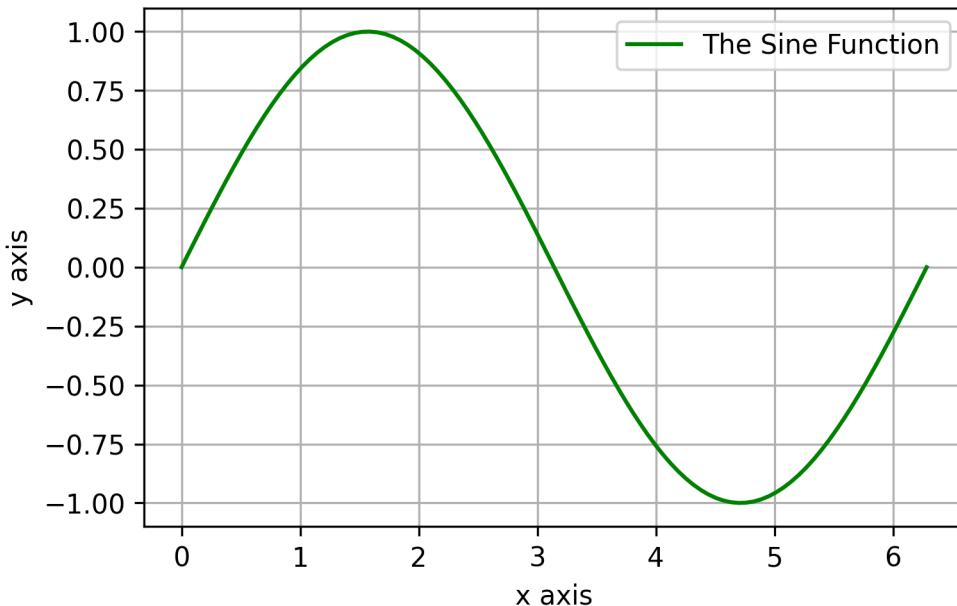


Figure 1.1: The sine function

Example 1.44. 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

$$y_0 = \sin(2\pi x) \quad y_1 = \cos(2\pi x) \quad y_2 = y_0 + y_1 \quad \text{and} \quad y_3 = y_0 - y_1 \quad (1.8)$$

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.

```
import numpy as np
import matplotlib.pyplot as plt
# %matplotlib inline # you may need this in Jupyter Notebooks

# build the x and y values
x = np.linspace(0,1,100)
y0 = np.sin(2*np.pi*x)
y1 = np.cos(2*np.pi*x)
y2 = y0 + y1
y3 = y0 - y1

# plot each of the functions
# (notice that they will be on the same axes)
plt.plot(x, y0, 'b-.', label=r"$y_0 = \sin(2\pi x)$")
plt.plot(x, y1, 'r--', label=r"$y_1 = \cos(2\pi x)$")
plt.plot(x, y2, 'g:', label=r"$y_2 = y_0 + y_1$")
plt.plot(x, y3, 'k-', label=r"$y_3 = y_0 - y_1$")

# put in a grid, legend, title, and axis labels
plt.grid()
plt.legend()
plt.title("Awesome Graph")
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.show()
```

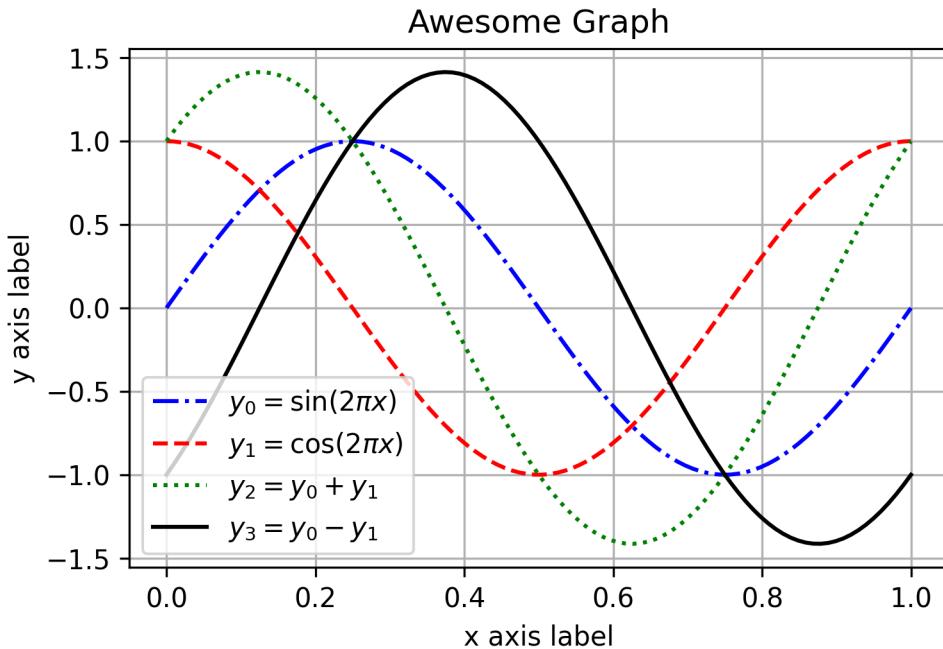


Figure 1.2: Plots of the sine, cosine, and sums and differences.

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 1.45. 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.

```
# The next line of code does all of the plotting of all
# of the functions. Notice the order: x, y, color and
# line style, repeat
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0,1,100)
```

```

y0 = np.sin(2*np.pi*x)
y1 = np.cos(2*np.pi*x)
y2 = y0 + y1
y3 = y0 - y1
plt.plot(x, y0, 'b-.', x, y1, 'r--', x, y2, 'g:', x, y3, 'k-')

plt.grid()
plt.legend([r"$y_0 = \sin(2\pi x)$", r"$y_1 = \cos(2\pi x)$", \
           r"$y_2 = y_0 + y_1$", r"$y_3 = y_0 - y_1$"])
plt.title("Awesome Graph")
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.show()

```

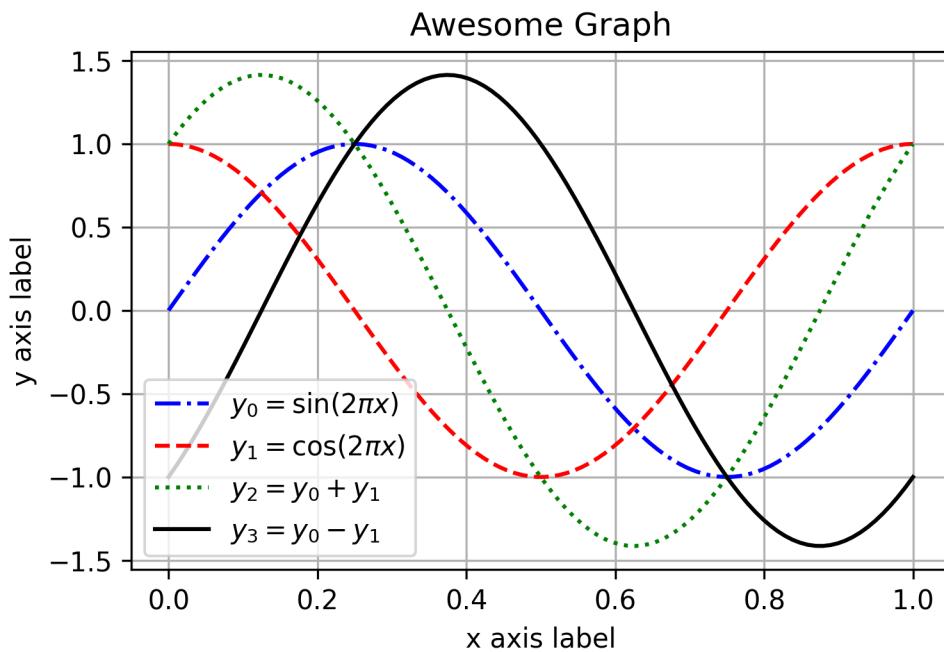


Figure 1.3: A second plot of the sine, cosine, and sums and differences.

Exercise 1.18. 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.

1.5.2 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 1.46. 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.

```
# set up the blank matrix of plots
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0,1,100)
y0 = np.sin(2*np.pi*x)
y1 = np.cos(2*np.pi*x)
y2 = y0 + y1
y3 = y0 - y1

fig, axes = plt.subplots(nrows = 2, ncols = 2)

# Build the first plot
axes[0,0].plot(x, y0, 'b-.')
axes[0,0].grid()
axes[0,0].set_title(r"$y_0 = \sin(2\pi x)$")
axes[0,0].set_ylim(-2,2)
axes[0,0].set_xlabel("x")
axes[0,0].set_ylabel("y")

# Build the second plot
```

```

axes[0,1].plot(x, y1, 'r--')
axes[0,1].grid()
axes[0,1].set_title(r"$y_1 = \cos(2\pi x)$")
axes[0,1].set_ylim(-2,2)
axes[0,1].set_xlabel("x")
axes[0,1].set_ylabel("y")

# Build the first plot
axes[1,0].plot(x, y2, 'g:')
axes[1,0].grid()
axes[1,0].set_title(r"$y_2 = y_0 + y_1$")
axes[1,0].set_ylim(-2,2)
axes[1,0].set_xlabel("x")
axes[1,0].set_ylabel("y")

# Build the first plot
axes[1,1].plot(x, y3, 'k-')
axes[1,1].grid()
axes[1,1].set_title(r"$y_3 = y_0 - y_1$")
axes[1,1].set_ylim(-2,2)
axes[1,1].set_xlabel("x")
axes[1,1].set_ylabel("y")

fig.tight_layout()
plt.show()

```

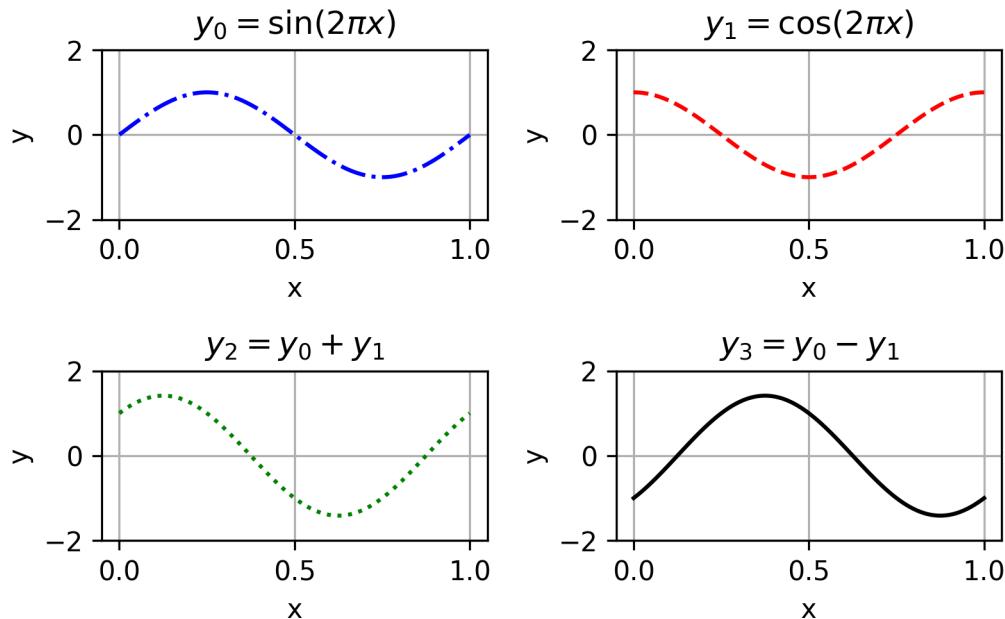


Figure 1.4: An example of subplots

The `fig.tight_layout()` command makes the plot labels a bit more readable in this instance (again, something you can play with).

Exercise 1.19. 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.

1.5.3 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 1.47. 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.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0,500,1000)
y = 10**(-0.01*x)
fig, axis = plt.subplots(1,2)

axis[0].plot(x,y, 'r')
axis[0].grid()
axis[0].set_title("Linearly scaled y axis")
axis[0].set_xlabel("x")
axis[0].set_ylabel("y")

axis[1].semilogy(x,y, 'r')
axis[1].grid()
axis[1].set_title("Logarithmically scaled y axis")
axis[1].set_xlabel("x")
axis[1].set_ylabel("Log(y)")

fig.tight_layout()
plt.show()
```

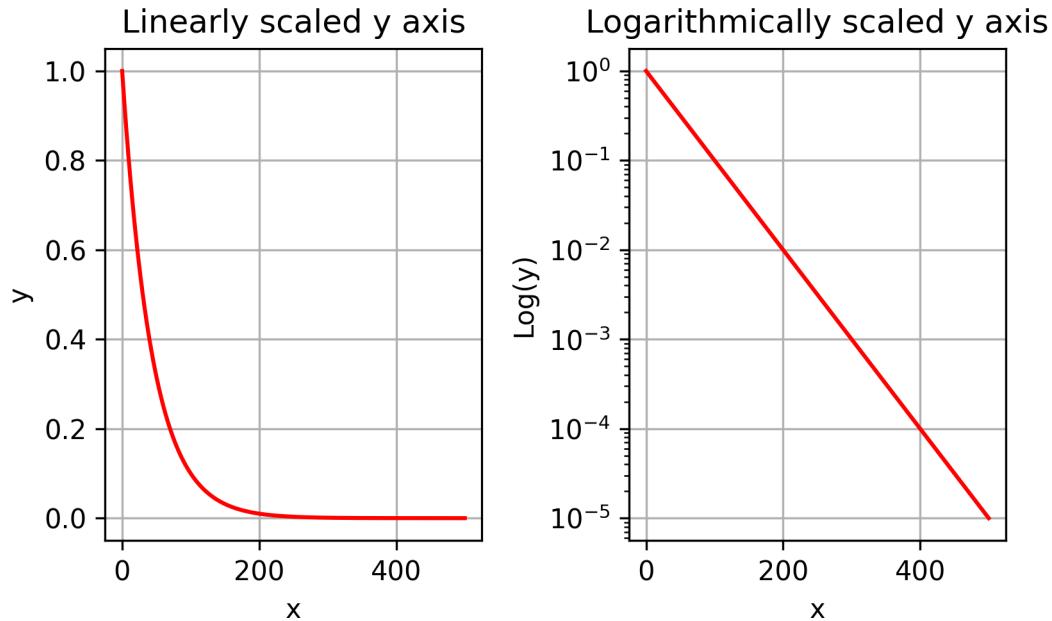


Figure 1.5: An example of using logarithmic scaling.

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 1.20. 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.

1.6 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 with data in this module, we do not need to dive deep into the Pandas package. We will only use it to collect computational results into tables for easier display.

Example 1.48. In this example we will build a simple dataframe with Pandas. We will build a table of the first 10 natural numbers and their squares and cubes. We will then display the table.

```
import numpy as np
import pandas as pd

# Calculate the columns for the table
n = np.arange(1,11)
n2 = n**2
n3 = n**3

# Combine the columns into a data frame with headers
df = pd.DataFrame({'n': n, 'n^2': n2, 'n^3': n3})
df
```

	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

1.7 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.

Exercise 1.21. (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 1.22. (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:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots \quad (1.9)$$

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 1.23. 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 1.24. (This problem is modified from (“Project Euler” n.d.))
The sum of the squares of the first ten natural numbers is,

$$1^2 + 2^2 + \cdots + 10^2 = 385 \quad (1.10)$$

The square of the sum of the first ten natural numbers is,

$$(1 + 2 + \cdots + 10)^2 = 55^2 = 3025 \quad (1.11)$$

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 1.25. (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 1.26. (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 1.27. The following iterative sequence is defined for the set of positive integers:

$$\begin{aligned} n \rightarrow \frac{n}{2} & \quad (n \text{ is even}) \\ n \rightarrow 3n + 1 & \quad (n \text{ is odd}) \end{aligned} \quad (1.12)$$

Using the rule above and starting with 13, we generate the following sequence:

$$13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 \quad (1.13)$$

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

2 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 knows 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 3 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 2.1. By hand (no computers!) compute the first 50 terms of this sequence with the initial condition $x_0 = 1/10$.

$$x_{n+1} = \begin{cases} 2x_n, & x_n \in [0, \frac{1}{2}] \\ 2x_n - 1, & x_n \in (\frac{1}{2}, 1] \end{cases} \quad (2.1)$$

Exercise 2.2. Now use a spreadsheet to do the computations. Do you get the same answers?

Exercise 2.3. Finally, solve this problem with Python. Some starter code is given to you below.

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

(Even if you don't know Python, you should be able to do this exercise after having read up to Section 1.3.1 in the chapter on Essential Python.)

Exercise 2.4. 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 2.5. Now what happens with this problem when you start with $x_0 = 1/8$? Why does this new initial condition work better?

2.1 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

$$735 = 700 + 30 + 5 = 7 \times 10^2 + 3 \times 10^1 + 5 \times 10^0 \quad (2.2)$$

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

$$101,101_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \quad (2.3)$$

(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 2.6. Express the following binary numbers in base-10.

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

Exercise 2.7. Explain the joke: *There are 10 types of people. Those who understand binary and those who do not.*

Exercise 2.8. Discussion: With your group, discuss how you would convert a base-10 number into its binary representation. 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 2.9. 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?
-

Exercise 2.10. Now that you have converted several base-10 numbers to base-2, summarize an efficient technique to do the conversion.

Next we will work with fractions and decimals.

Example 2.1. 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 2.11. 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 2.2. 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 2.12. Convert the following numbers from base 10 to binary.

1. What is $1/2$ in binary?
 2. What is $1/8$ in binary?
 3. What is 4.125 in binary?
 4. What is 0.15625 in binary?
-

Exercise 2.13. Convert the base 10 decimal 0.635 to binary using the following steps.

1. Multiply 0.635 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, until a repeating pattern has revealed itself, or until your precision is close enough.

Explain why each step gives the binary digit that it does.

Exercise 2.14. Convert the base 10 fraction 0.1 into binary. Use this to explain why errors arose in Exercise 2.3.

2.2 Floating Point Numbers

Everything stored in the memory of a computer is a number, but how does a computer actually store 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.

Exercise 2.15. 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).

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

$$x = -1.\underline{\hspace{2em}} \times 2\underline{\hspace{2em}}$$

- b. Based on the fact that every binary number, other than 0, can be written in this way, what three things do you suppose a computer needs to store for any given number?
 - c. Using your answer to part (b), what would a computer need to store for the binary number $x = 10001001.1100110011_2$?
-

Definition 2.1. For any non-zero base-2 number x we can write

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

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

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.

Example 2.3. 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 2.3 below.

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 2.2. 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.
- A **double-precision** number consists of 64 bits with 1 bit for the sign, 11 for the exponent, and 52 for the mantissa.

Definition 2.3. **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 2.16. To make all of these ideas concrete let us play with a small computer system where each number is stored in the following format:

$$s E b_1 b_2 b_3$$

The first entry is a bit for the sign ($0 = +$ and $1 = -$). The second entry, E is for the exponent, and we will assume in this example that the exponent can be 0, 1, or -1 . The three bits on the right represent the significand of the number. Hence, every number in this number system takes the form

$$(-1)^s \times (1 + 0.b_1 b_2 b_3) \times 2^E$$

- 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?
 - What would change if we allowed $E \in \{-2, -1, 0, 1, 2\}$?
-

Exercise 2.17. What are the largest numbers that can be stored in single and double precision?

Exercise 2.18. What is machine precision for the single and double precision standard?

Exercise 2.19. What is the gap between 2^n and the next largest number that can be stored in double precision?

Exercise 2.20. The gap between consecutive floating-point numbers gets larger as the numbers get larger.

- a) Explain why this makes sense from a practical perspective.
 - b) Why might this be problematic when adding a very small number to a very large number?
 - c) How could you rewrite the calculation $(10^8 + 0.1 - 10^8)$ to get a more accurate result?
-

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:

- All numbers in a computer are stored with finite precision.
- Nice 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.

2.3 Rounding Errors

We have seen that 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. In this section you will explore a bit more the rounding errors that arise from this.

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 2.4. 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 2.3, simplify the detailed analysis by working with only a 4 bit mantissa:

Exercise 2.21. Calculate the first 10 terms of the sequence

$$x_{n+1} = \begin{cases} 2x_n, & x_n \in [0, \frac{1}{2}] \\ 2x_n - 1, & x_n \in (\frac{1}{2}, 1] \end{cases} \quad \text{with } x_0 = \frac{1}{10} \quad (2.4)$$

using a number system where the mantissa has only 4 bits.

If you want to delve more deeply into this, take a look at Exercise 2.28.

Exercise 2.22. (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 by-hand 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 $(\frac{1}{49}) \cdot 49$ should just be 1. In Python type `(1/49)*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.

2.4 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 2.3 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 2.5. Consider the expression

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

Mathematically, the two terms of 10^{10} simply cancel out leaving just 0.123456789. However, let us evaluate this in Python:

```
10**10 + 0.123456789 - 10**10
```

```
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 2.23. 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?

- c) Can you think of a real-world scenario where this difference would matter?
-

Exercise 2.24. 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 1.3.8, 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 2.25. 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 larger than 1. 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 2.26. Consider computing the sum of n numbers in two different ways:

- 1) Adding them in order from smallest to largest
- 2) Adding them in order from largest to smallest

Which approach would you expect to give more accurate results? Why? Give an example to illustrate your answer.

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.

2.5 Problems

These problem exercises will let you consolidate what you have learned so far and combine it with the coding skills you picked up in Chapter 1.

Exercise 2.27. (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.

```

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 2.28. (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 $\frac{1}{10}$:

$$0.1_{10} \approx 0.00011001100110011001100_2. \quad (2.5)$$

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 2.29 (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, \wedge , 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).

$$\begin{array}{rcl} 9 & = & \underline{\quad} \underline{\quad} \underline{\quad} \underline{\quad} \\ 5 & = & \underline{\quad} \underline{\quad} \underline{\quad} \underline{\quad} \end{array} \tag{2.6}$$

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.

3 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.

Exercise 3.1. In this problem we 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?

3.1 Polynomial Approximations

Exercise 3.2. Discuss: Could a polynomial function with a high enough degree satisfy everything in the wish list from the previous problem? Explain your reasoning.

Exercise 3.3. 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 $f'(x_0) = g'(x_0)$? (*Hint: Start by taking the derivative of $g(x)$*)
 3. What is the value of C such that $f''(x_0) = g''(x_0)$?
 4. 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 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 sine, cosine, logarithms, etc. 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.*

Exercise 3.4. What is Euler's number e ? You likely remember 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 $f(x) = e^x$.

- The function $g(x) = 1$ 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 $g(x)$ are really close to each other. Hence we could say that $g(x) = 1$ 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 $x = 0$.
- Let us get a better approximation. What if we insist that our approximation $g(x)$ matches $f(x) = e^x$ exactly at $x = 0$ and ALSO has exactly the same first derivative as $f(x)$ at $x = 0$.
 - What is the first derivative of $f(x)$?
 - What is $f'(0)$?
 - Use the point-slope form of a line to write the equation of the function $g(x)$ 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 $g(x) = f(0) + f'(0)(x - 0)$ to get the equation of the line.
- Write Python code to build a plot like Figure 3.1. This plot shows $f(x) = e^x$, our first approximation $g(x) = 1$ and our second approximation $g(x) = 1 + x$. You may want to refer back to Exercise 1.18 in the Python chapter.

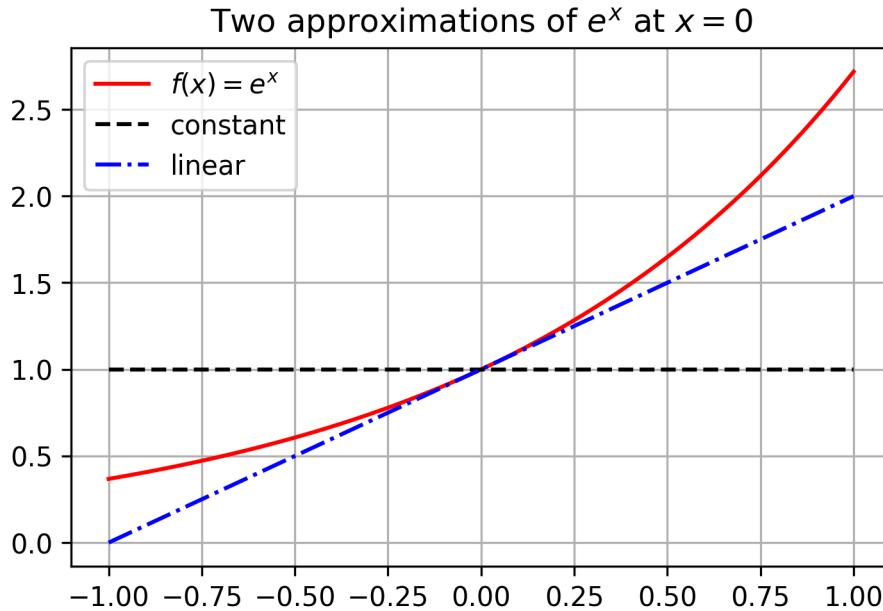


Figure 3.1: The first two polynomial approximations of the exponential function.

Exercise 3.5. 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 $g(x)$ 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

$$y = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2. \quad (3.1)$$

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

$$y = f(0) + f'(0)x + \frac{f''(0)}{2}x^2. \quad (3.2)$$

1. Find the quadratic approximation for $f(x) = e^x$.
 2. How do you know that this function matches $f(x)$ in all of the ways described above at $x = 0$?
 3. 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
- $$y = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 \quad (3.3)$$
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 $x = 0$?
 3. Add your function to the plot.
 4. Pause and think: What's the deal with the $3!$ on the cubic term?
 3. Your turn: Build the next several approximations of $f(x) = e^x$ at $x = 0$. Add these plots to the plot that we have been building all along.
-

Exercise 3.6. Use the functions that you have built to approximate $\frac{1}{e} = e^{-1}$. Check the accuracy of your answer using `np.exp(-1)` in Python.

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

Definition 3.1 (Taylor Series). If $f(x)$ is an infinitely differentiable function at the point x_0 then

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + \dots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + \dots \quad (3.4)$$

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

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k. \quad (3.5)$$

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 3.7. 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

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad (3.6)$$

Exercise 3.8. Let us compute a few Taylor Series that are not centred at $x_0 = 0$. For example, let us approximate the function $f(x) = \sin(x)$ near $x_0 = \frac{\pi}{2}$. Near the point $x_0 = \frac{\pi}{2}$, the Taylor Series approximation will take the form

$$f(x) = f\left(\frac{\pi}{2}\right) + f'\left(\frac{\pi}{2}\right)\left(x - \frac{\pi}{2}\right) + \frac{f''\left(\frac{\pi}{2}\right)}{2!}\left(x - \frac{\pi}{2}\right)^2 + \frac{f'''\left(\frac{\pi}{2}\right)}{3!}\left(x - \frac{\pi}{2}\right)^3 + \dots \quad (3.7)$$

Write the first several terms of the Taylor Series for $f(x) = \sin(x)$ centred at $x_0 = \frac{\pi}{2}$. Then write Python code to build the plot below showing successive approximations for $f(x) = \sin(x)$ centred at $\pi/2$.

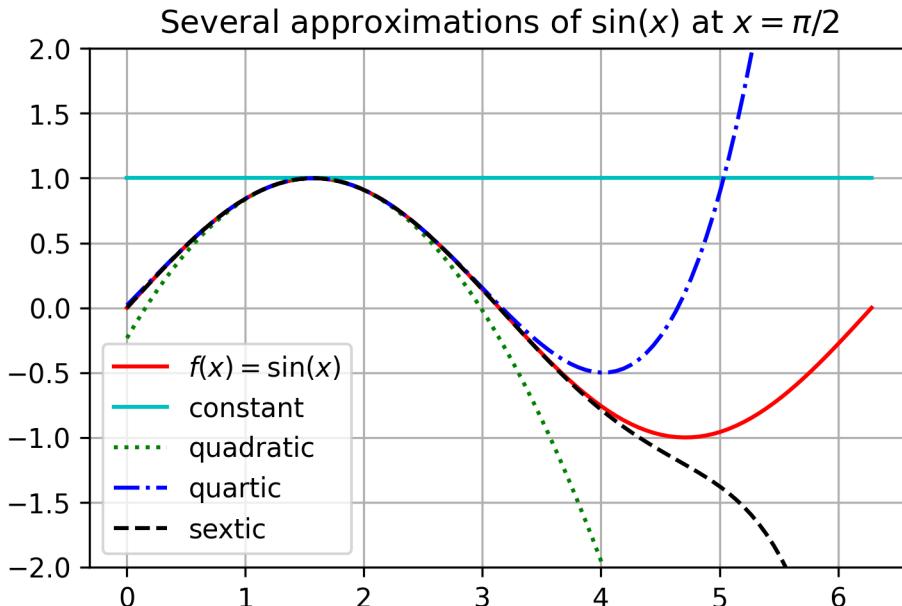


Figure 3.2: Taylor series approximation of the sine function.

Exercise 3.9. Repeat the previous exercise for the function

$$f(x) = \log(x) \text{ centered at } x_0 = 1.$$

Use this to give an approximate value for $\log(1.1)$.

Example 3.1. Let us conclude this brief section by examining an interesting example. Consider the function

$$f(x) = \frac{1}{1-x}. \quad (3.8)$$

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

$$f(x) = 1 + x + x^2 + x^3 + x^4 + x^5 + \dots \quad (3.9)$$

(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 3.3). 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.

```
import numpy as np
import math as ma
import matplotlib.pyplot as plt

# build the x and y values
x = np.linspace(-1,2,101)
y0 = 1/(1-x)
y1 = 1 + 0*x
y2 = 1 + x
y3 = y2 + x**2
y4 = y3 + x**3 + x**4 + x**5 + x**6 + x**7 + x**8

# plot each of the functions
plt.plot(x, y0, 'r-', label=r"$f(x)=1/(1-x)$")
plt.plot(x, y1, 'c-', label="constant")
plt.plot(x, y2, 'g:', label="linear")
plt.plot(x, y3, 'b-.', label="quadratic")
plt.plot(x, y4, 'k--', label="8th order")
```

```

# set limits on the y axis
plt.ylim(-3,5)

# put in a grid, legend, title, and axis labels
plt.grid()
plt.legend()
plt.title(r"Taylor approximations of $f(x)=\frac{1}{1-x}$ around $x=0$")
plt.show()

```

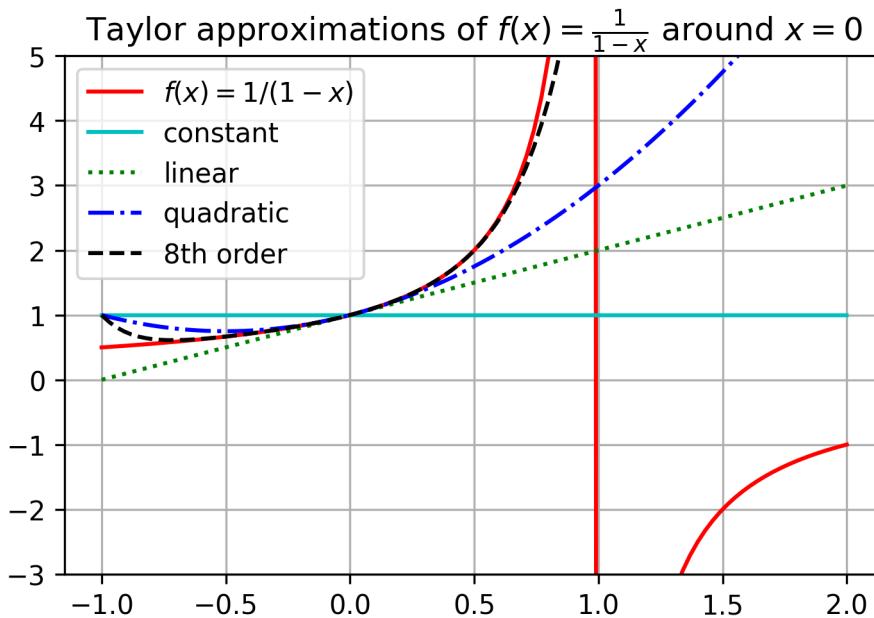


Figure 3.3: Several Taylor Series approximations of the function $f(x) = 1/(1-x)$.

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. While it is beyond the scope of this section to give all of the details for finding the domain of convergence for a Taylor Series, 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.

3.2 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

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots \quad (3.10)$$

0^{th} Order Approximation of $f(x) = e^x$: If we want to use a zeroth-order (constant) approximation $f_0(x)$ of the function $f(x) = e^x$ then we only take the first term in the Taylor Series and the rest is not used for the approximation

$$e^x = \underbrace{1}_{0^{th} \text{ order approximation}} + \underbrace{x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots}_{\text{remainder}}. \quad (3.11)$$

Therefore we would approximate e^x as $e^x \approx 1 = f_0(x)$ for values of x that are close to $x_0 = 0$. Furthermore, for small values of x that are close to $x_0 = 0$ the largest term in the remainder is x (since for small values of x like 0.01, x^2 will be even smaller, x^3 even smaller than that, etc). This means that if we use a 0^{th} order approximation for e^x then we expect our error to be about the same size as x . It is common to then rewrite the truncated Taylor Series as

$$0^{th} \text{ order approximation: } e^x \approx 1 + \mathcal{O}(x) \quad (3.12)$$

where $\mathcal{O}(x)$ (read “Big-O of x ”) tells us that the expected error for approximations close to $x_0 = 0$ is about the same size as x .

1^{st} Order Approximation of $f(x) = e^x$: If we want to use a first-order (linear) approximation $f_1(x)$ of the function $f(x) = e^x$ then we gather the 0^{th} order and 1^{st} order terms together as our approximation and the rest is the remainder

$$e^x = \underbrace{1 + x}_{1^{st} \text{ order approximation}} + \underbrace{\frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots}_{\text{remainder}}. \quad (3.13)$$

Therefore we would approximate e^x as $e^x \approx 1 + x = f_1(x)$ for values of x that are close to $x_0 = 0$. Furthermore, for values of x very close to $x_0 = 0$ the largest term in the remainder is the x^2 term. Using Big-O notation we can write the approximation as

$$1^{st} \text{ order approximation: } e^x \approx 1 + x + \mathcal{O}(x^2). \quad (3.14)$$

Notice that we do not explicitly say what the coefficient is for the x^2 term. Instead we are just saying that *using the linear function $y = 1 + x$ to approximate e^x for values of x near $x_0 = 0$* will result in errors that are of the order of x^2 .

2^{nd} Order Approximation of $f(x) = e^x$: If we want to use a second-order (quadratic) approximation $f_2(x)$ of the function $f(x) = e^x$ then we gather the 0^{th} order, 1^{st} order, and 2^{nd} order terms together as our approximation and the rest is the remainder

$$e^x = \underbrace{1 + x + \frac{x^2}{2!}}_{2^{nd} \text{ order approximation}} + \underbrace{\frac{x^3}{3!} + \frac{x^4}{4!} + \dots}_{\text{remainder}}. \quad (3.15)$$

Therefore we would approximate e^x as $e^x \approx 1 + x + \frac{x^2}{2} = f_2(x)$ for values of x that are close to $x_0 = 0$. Furthermore, for values of x very close to $x_0 = 0$ the largest term in the remainder is the x^3 term. Using Big-O notation we can write the approximation as

$$2^{nd} \text{ order approximation: } e^x \approx 1 + x + \frac{x^2}{2} + \mathcal{O}(x^3). \quad (3.16)$$

Again notice that we do not explicitly say what the coefficient is for the x^3 term. Instead we are just saying that *using the quadratic function* $y = 1 + x + \frac{x^2}{2}$ to approximate e^x for values of x near $x_0 = 0$ will result in errors that are of the order of 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.

For the function $f(x) = e^x$ the idea of approximating the amount of approximation error by truncating the Taylor Series is relatively straight forward: if we want an n^{th} order polynomial approximation $f_n(x)$ of the function of $f(x) = e^x$ near $x_0 = 0$ then

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots + \frac{x^n}{n!} + \mathcal{O}(x^{n+1}), \quad (3.17)$$

meaning that we expect the error to be of the order of x^{n+1} .

Exercise 3.10. 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 $f(0.1) = e^{0.1} \approx \text{np.exp}(0.1) = 1.1051709180756477$ and $f(0.2) = e^{0.2} \approx \text{np.exp}(0.2) = 1.2214027581601699$ 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 want to create the table in your jupyter notebook, you can use Pandas as described in Section 1.6. Alternatively feel free to use a spreadsheet instead of using Python.

Order n	$f_n(0.1)$	$\epsilon_n(0.1) = f(0.1) - f_n(0.1) $	$f_n(0.2)$	$\epsilon_n(0.2) = f(0.2) - f_n(0.2) $
0	1	1.051709e-01	1	2.214028e-01
1	1.1	5.170918e-03	1.2	

Order n	$f_n(0.1)$	$\epsilon_n(0.1) = f(0.1) - f_n(0.1) $	$f_n(0.2)$	$\epsilon_n(0.2) = f(0.2) - f_n(0.2) $
2	1.105			
3				
4				
5				

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

Exercise 3.11. 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	$\epsilon_n(0.1)$	$\epsilon_n(0.2)$	$\epsilon_n(0.2)/\epsilon_n(0.1)$
0	1.051709e-01	2.214028e-01	2.105171
1	5.170918e-03		
2			
3			
4			
5			

Formulate a conjecture about how ϵ_n changes as x changes.

Exercise 3.12. 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

$$f(x) = \sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad (3.18)$$

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	$\epsilon_n(0.1)$	$\epsilon_n(0.2)$	$\epsilon_n(0.2)/\epsilon_n(0.1)$	Your Conjecture
1	1.665834e-04	1.330669e-03		
3	8.331349e-08	2.664128e-06		
5	1.983852e-11			
7				
9				

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: does `np.sin(x)` give you the exact value of $\sin(x)$?

Exercise 3.13. Perform another check of your conjecture by approximating $\log(1.02)$ and $\log(1.1)$ from truncations of the Taylor series around $x = 1$:

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

If you are using Python then use `np.log1p(x)` to calculate $\log(1 + x)$.

Exercise 3.14. Write down your observations about how the truncation error at order n changes as x changes. Explain this in terms of the form of the remainder of the truncated Taylor series.

3.3 Problems

Exercise 3.15. 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 3.16. In this problem we will use Taylor Series to build approximations for the irrational number π .

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

$$f(x) = \frac{1}{1+x}. \quad (3.19)$$

2. 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 $g(x) = \frac{1}{1+x^2}$.
3. Recall from Calculus that

$$\int \frac{1}{1+x^2} dx = \arctan(x). \quad (3.20)$$

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

4. Now recall the following from Calculus:

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

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.

¹There 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.

Exercise 3.17. In this problem we will prove the famous (and the author's favourite) formula

$$e^{i\theta} = \cos(\theta) + i \sin(\theta). \quad (3.21)$$

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:

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \dots \\ e^{i\theta} &= 1 + (i\theta) + \frac{(i\theta)^2}{2!} + \frac{(i\theta)^3}{3!} + \frac{(i\theta)^4}{4!} + \frac{(i\theta)^5}{5!} + \dots \\ &= 1 + i\theta + i^2 \frac{\theta^2}{2!} + i^3 \frac{\theta^3}{3!} + i^4 \frac{\theta^4}{4!} + i^5 \frac{\theta^5}{5!} + \dots \\ &= \dots \text{ keep simplifying } \dots \dots \end{aligned} \quad (3.22)$$

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 (c) to prove that $e^{i\pi} + 1 = 0$.
-

Exercise 3.18. In physics, the *relativistic energy* of an object is defined as

$$E_{rel} = \gamma mc^2 \quad (3.23)$$

where

$$\gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}. \quad (3.24)$$

In these equations, m is the mass of the object, c is the speed of light ($c \approx 3 \times 10^8 \text{ m/s}$), 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

$$E_{rel} = mc^2 + \frac{1}{2}mv^2 + \frac{3}{8}\frac{mv^4}{c^2} + \frac{5}{16}\frac{mv^6}{c^4} + \dots \quad (3.25)$$

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

$$E_{rel} = mc^2 + \frac{1}{2}mv^2 \quad (3.26)$$

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.

²This 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.

4 Non-linear Equations

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

—Zeno of Elea

4.1 Introduction to Numerical Root Finding

In this chapter we want to solve equations using a computer. 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.

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

$$\ell(x) - r(x) = 0. \tag{4.1}$$

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

$$\text{Find } x \text{ such that } f(x) = 0. \tag{4.2}$$

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 4.1. 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.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-4,4, 100)
l = 3 * np.sin(x) + 9
r = x**2 - np.cos(x)

fig, axes = plt.subplots(nrows = 1, ncols = 2)

axes[0].plot(x, l, 'b-', label=r"$3\sin(x)+9$")
axes[0].plot(x, r, 'r-', label=r"$x^2-\cos(x)$")
axes[0].grid()
axes[0].legend()
axes[0].set_title(r"$3\sin(x)+9 = x^2-\cos(x)$")

axes[1].plot(x, l-r, 'g:', label=r"(3\sin(x)+9) - (x^2-\cos(x))")
axes[1].plot(x, np.zeros(100), 'k-')
axes[1].grid()
axes[1].legend()
axes[1].set_title(r"$(3\sin(x)+9) - (x^2-\cos(x))=0$")

fig.tight_layout()
plt.show()
```

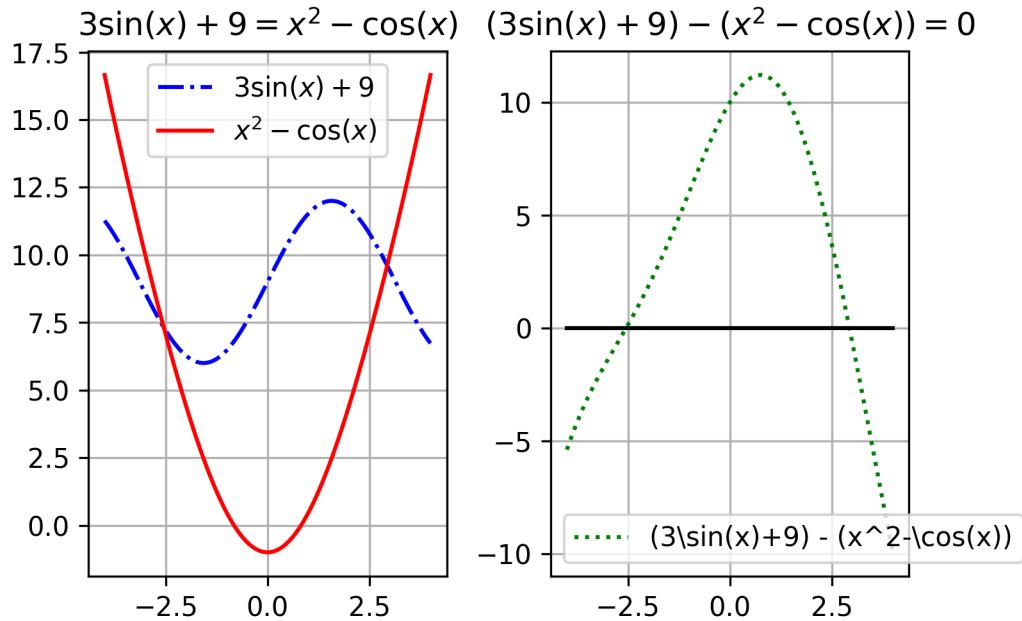


Figure 4.1: Two ways to visualise the same root finding problem

On the left-hand side of Figure 4.1 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 4.1 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.

4.2 The Bisection Method

4.2.1 Intuition

Exercise 4.1. 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 4.2. 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 4.3. 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 4.4. 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 4.1 (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 4.5. Draw a picture of what the intermediate value theorem says graphically.

Exercise 4.6. If $y_* = 0$ the Intermediate Value Theorem gives us important information about solving equations. What does it tell us?

Corollary 4.1. *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 _____.*

Exercise 4.7. Fill in the blank in the previous corollary and then draw several pictures that indicate why this might be true for continuous functions.

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 4.1 to Exercise 4.3 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 4.8. Where is the Intermediate Value Theorem used in the root-guessing strategy?

Exercise 4.9. 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.

4.2.2 Implementation

Exercise 4.10 (The Bisection Method). Goal: We want to solve the equation $f(x) = 0$ for x assuming that the solution x^* is in the interval $[a, b]$.

The Algorithm: Assume that $f(x)$ is continuous on the closed 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, $m = \frac{a+b}{2}$, and evaluate $f(m)$.
 - 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. Repeat steps 2 and 3 and stop when the interval containing the root is small enough.
-

Exercise 4.11. Draw a picture illustrating what the Bisection Method does to approximate a solution to an equation $f(x) = 0$.

Exercise 4.12. 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.

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

    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 4.13. Now use the comments from Exercise 4.12 as structure to complete a Python function for the Bisection Method. Test your Bisection Method code on the following equations.

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. $(5 - x)e^x = 5$ on $x \in [0, 5]$
-
-

4.2.3 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 4.14. Discussion: What must be true in order to use the bisection method?

Exercise 4.15. Discussion: Does the bisection method work if the Intermediate Value Theorem does not apply? (Hint: what does it mean for the IVT to “not apply?”)

Exercise 4.16. 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 4.17. 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	$ b - a $	$\frac{ b-a }{2}$
2	$\frac{ b-a }{2}$	
3	$\frac{ b-a }{2^2}$	
:	:	:
n	$\frac{ b-a }{2^{n-1}}$	

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 4.17 you actually proved the following theorem.

Theorem 4.2 (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*

$$\frac{|b - a|}{2^n} = \varepsilon \quad (4.3)$$

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

Solving for the number n of iterations we get

$$n = \log_2 \left(\frac{|b - a|}{\varepsilon} \right). \quad (4.4)$$

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

Exercise 4.18. Is it possible for a given function and a given interval that the Bisection Method converges to the root in fewer steps than what you just found in the previous problem? Explain.

Exercise 4.19. 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 versions 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 4.2 you know that the bisection method cuts the interval in half at every iteration. You proved in Exercise 4.17 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 4.1. 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:

$$\text{Absolute Error} = |\text{Approximate Solution} - \text{Exact Solution}|. \quad (4.5)$$

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

```

def bisection_with_error_tracking(f, x_exact, a, b, tol):
    """
    Implements the bisection method and tracks absolute error at each iteration.

    Args:
        f (callable): Function for which to find the root
        x_exact (float): The exact root of the function
        a (float): Left endpoint of initial interval
        b (float): Right endpoint of initial interval
        tol (float): Tolerance for stopping criterion

    Returns:
        list: List of absolute errors between approximate and exact solution at each iteration
    """
    errors = []
    while (b - a) / 2.0 > tol:
        midpoint = (a + b) / 2.0
        if f(midpoint) == 0:
            break
        elif f(a) * f(midpoint) < 0:
            b = midpoint
        else:
            a = midpoint
        error = abs(midpoint - x_exact)
        errors.append(error)
    return errors

```

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.

```

import numpy as np

def f(x):
    return x**2 - 2

x_exact = np.sqrt(2)

# Using the interval [1, 2] and a tolerance of 1e-7
tolerance = 1e-7
errors = bisection_with_error_tracking(f, x_exact, 1, 2, tolerance)
errors

```

```
[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 4.2. 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.

```
import matplotlib.pyplot as plt
def plot_errors(errors):
    """
    Plot the absolute errors.

    Args:
        errors (list): List of absolute errors
    """
    # Creating the x values for the plot (iterations)
    iterations = np.arange(len(errors))

    # Plotting the errors
```

```

plt.scatter(iterations, errors, label='Error per Iteration')

plt.xlabel('Iteration')
plt.ylabel('Absolute Error')
plt.title('Absolute Error in Each Iteration')
plt.legend()
plt.show()

plot_errors(errors)

```

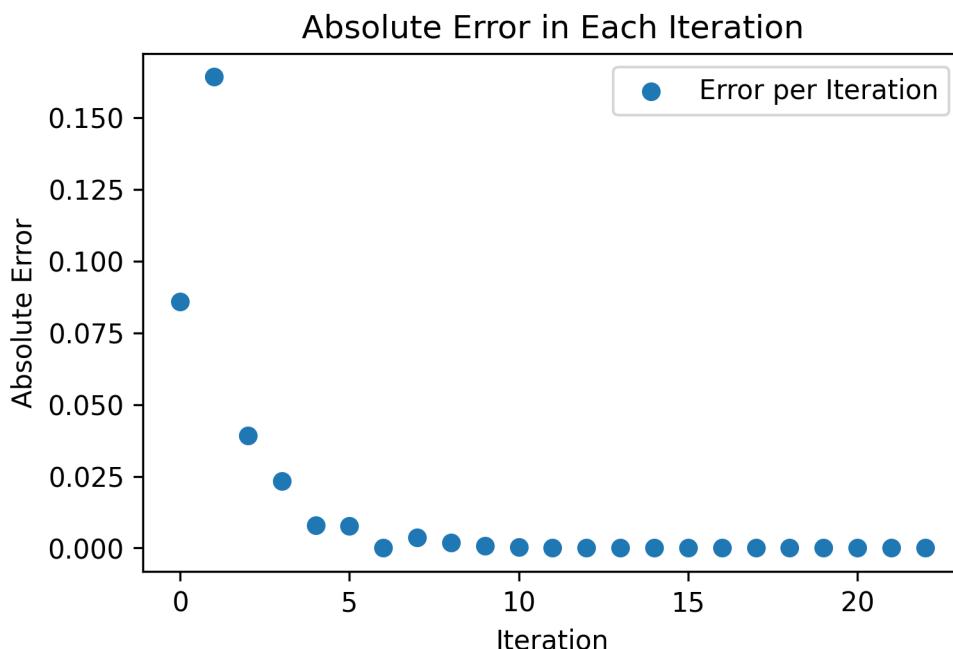


Figure 4.2: The evolution of the absolute error when solving the equation $x^2 - 2 = 0$ with the bisection method.

Without Theorem 4.2 it would be rather hard to tell what the exact behaviour is in the exponential plot above. We know from Theorem 4.2 that the error will divide by 2 at 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 4.3.

```

def plot_log_errors(errors):
    """
    Plot the base-2 logarithm of absolute errors and a best fit line.

```

```
Args:  
    errors (list): List of absolute errors  
"""  
# Convert errors to base 2 logarithm  
log_errors = np.log2(errors)  
# Creating the x values for the plot (iterations)  
iterations = np.arange(len(log_errors))  
  
# Plotting the errors  
plt.scatter(iterations, log_errors, label='Log Error per Iteration')  
  
# Determine slope and intercept of the best-fit straight line  
slope, intercept = np.polyfit(iterations, log_errors, deg=1)  
best_fit_line = slope * iterations + intercept  
# Plot the best-fit line  
plt.plot(iterations, best_fit_line, label='Best Fit Line', color='red')  
  
plt.xlabel('Iteration')  
plt.ylabel('Base 2 Log of Absolute Error')  
plt.title('Absolute Error in Each Iteration')  
plt.legend()  
plt.show()  
  
plot_log_errors(errors)
```

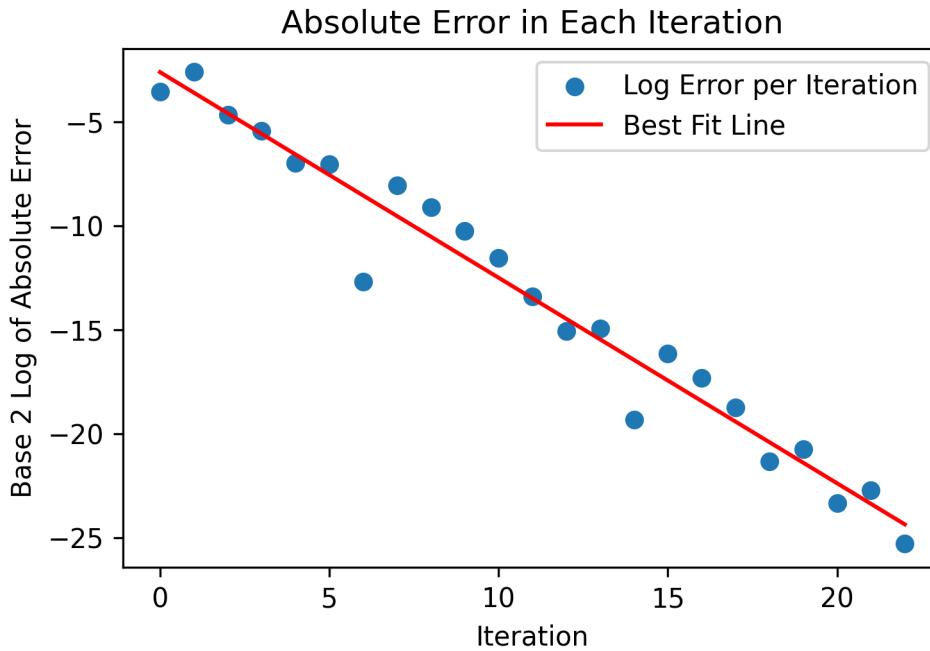


Figure 4.3: Iteration number vs the base-2 logarithm of the absolute error. Notice the slope of -1 indicating that the error is divided by a factor of 2 at each step of the algorithm.

There will be times later in this course where we will not have a nice theorem like Theorem 4.2 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. Visually verify that the slope in the plot below follows this trend (the red dashed line in the plot is shown to help you see the slope).
-

Exercise 4.20. Carefully read and discuss all of the details of the previous example and plots. Then create plots similar to this example to solve a different equation to which you know the exact solution. You should see the same basic behaviour based on the theorem that you proved in Exercise 4.17. If you do not see the same basic behaviour then something has gone wrong.

Example 4.2. Another plot that numerical analysts use quite frequently for determining how an algorithm is behaving as it progresses is shown in Figure 4.4. and is defined by the following axes:

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

```
def plot_error_progression(errors):
    # Calculating the log2 of the absolute error at step k and k+1
    log_errors = np.log2(errors)
    log_errors_k = log_errors[:-1]  # log errors at step k (excluding the last one)
    log_errors_k_plus_1 = log_errors[1:]  # log errors at step k+1 (excluding the first one)

    # Plotting log_errors_k+1 vs log_errors_k
    plt.scatter(log_errors_k, log_errors_k_plus_1, label='Log Error at k+1 vs Log Error at k')

    # Fitting a straight line to the data points
    slope, intercept = np.polyfit(log_errors_k, log_errors_k_plus_1, deg=1)
    best_fit_line = slope * log_errors_k + intercept
    plt.plot(log_errors_k, best_fit_line, color='red', label='Best Fit Line')

    # Setting up the plot
    plt.xlabel('Log2 of Absolute Error at Step k')
    plt.ylabel('Log2 of Absolute Error at Step k+1')
    plt.title('Log2 of Absolute Error at Step k+1 vs Step k')
    plt.legend()
    plt.show()

plot_error_progression(errors)
```

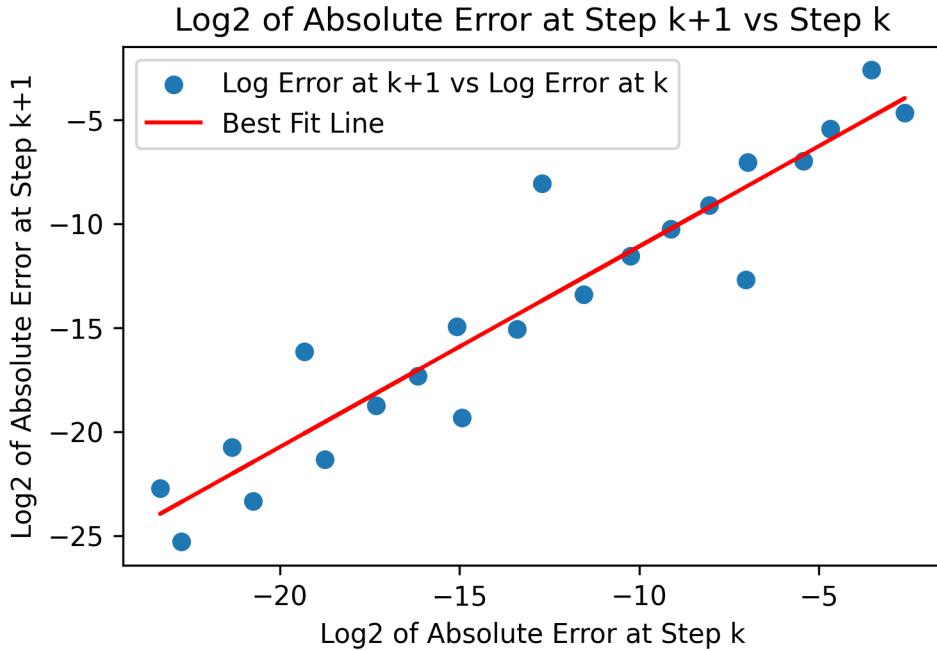


Figure 4.4: The base-2 logarithm of the absolute error at iteration k vs the base-2 logarithm of the absolute error at iteration $k + 1$.

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 k and the y-coordinate is the base-2 logarithm of the absolute error at step $k + 1$. The initial iterations 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 4.4 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

$$\log_2(\text{absolute error at step } k + 1) \approx 1 \cdot \log_2(\text{absolute error at step } k) - 1. \quad (4.6)$$

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

$$(\text{absolute error at step } k + 1) = \frac{1}{2}(\text{absolute error at step } k), \quad (4.7)$$

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 4.4. That's what makes these types of plots so powerful!

Exercise 4.21. Reproduce plots like those in the previous example but for the different equation that you used in Exercise 4.20. Again check that the plots have the expected shape.

4.3 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

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

Just as well, we could set $g(x) := \lambda f(x) + x$ with any $\lambda \in \mathbb{R} \setminus \{0\}$, 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 4.5.

Exercise 4.22. The plot that emerges in Figure 4.5 is known as a cobweb diagram, for obvious reason. Explain to others in your group what is happening in the animation in Figure 4.5 and how that animation is related to the fixed point iteration $x_n = \cos(x_{n-1})$.

Exercise 4.23. The animation in Figure 4.5 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 $\cos(x) - x = 0$.

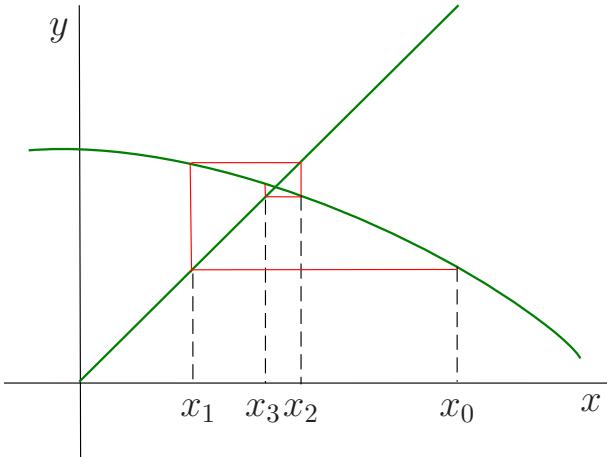


Figure 4.5: Fixed point iteration

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

$$p = \lim_{n \rightarrow \infty} x_n = \lim_{n \rightarrow \infty} g(x_{n-1}) = g\left(\lim_{n \rightarrow \infty} x_{n-1}\right) = g(p). \quad (4.8)$$

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 4.24. Copy the two plots in Figure 4.6 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?

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

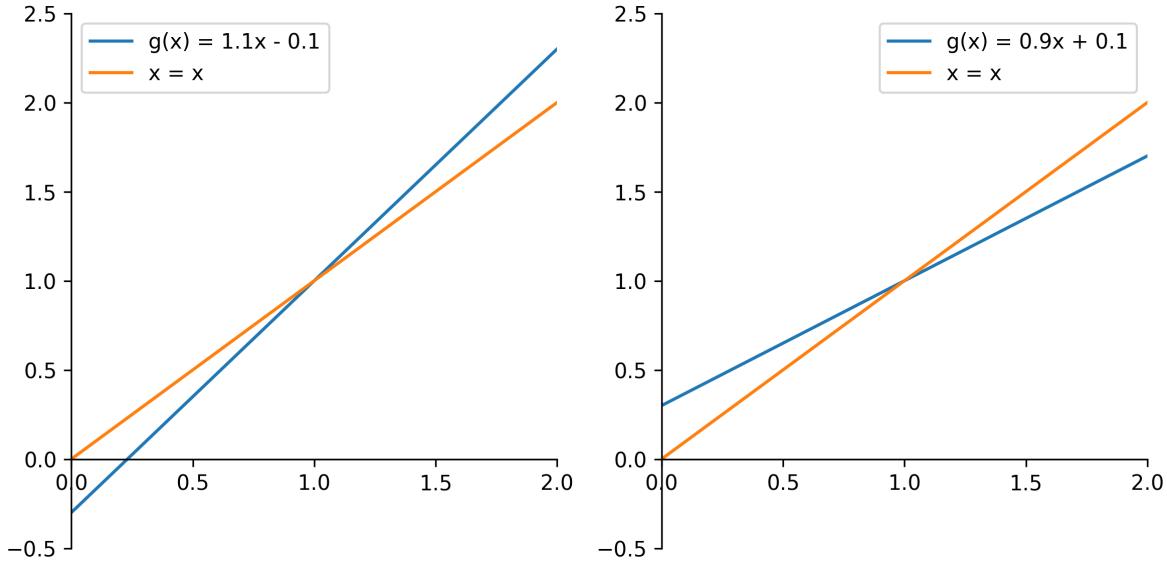


Figure 4.6: Two plots for practicing your cobweb skills.

Exercise 4.25. 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 4.3 (Fixed Point Theorem). *Suppose that $g : [a, b] \rightarrow [a, b]$ is differentiable, and that there exists $0 < k < 1$ such that*

$$|g'(x)| \leq k \quad \text{for all } x \in (a, b). \quad (4.9)$$

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

$$x_n := g(x_{n-1}) \quad \text{for all } n \geq 1 \quad (4.10)$$

converges to p . The following estimate holds:

$$|p - x_n| \leq k^n |p - x_0| \quad \text{for all } n \geq 1. \quad (4.11)$$

Proof. The proof of this theorem is not difficult, but you can skip it and go directly to Exercise 4.26 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. 4.9. 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. 4.9, Eq. 4.10

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

Applying this inequality inductively, we obtain the error estimate Eq. 4.11. Moreover since $k < 1$ we have

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

which implies that (x_n) converges to p . □

Exercise 4.26. This exercise shows why the conditions of the Theorem 4.3 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 4.3 gives only sufficient conditions for convergence; in practice, convergence might occur even if the conditions are violated.

Exercise 4.27. 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. 4.11 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).

4.4 Newton's Method

In the Bisection Method (Section 4.2) we had used only the sign of the function at the guessed points. We will now investigate how we can use also the value and the slope (derivative) of the function to get a much improved method.

4.4.1 Intuition

Root finding is really the process of finding the x -intercept of the function. If the function is complicated (e.g. highly nonlinear or does not lend itself to traditional by-hand techniques) then we can approximate the x -intercept by creating a Taylor Series approximation of the function at a nearby point and then finding the x -intercept of that simpler Taylor Series. The simplest non-trivial Taylor Series is a linear function – a tangent line!

Exercise 4.28. A tangent line approximation to a function $f(x)$ near a point x_0 is

$$y = f(x_0) + f'(x_0)(x - x_0). \quad (4.12)$$

Set y to zero and solve for x to find the x -intercept of the tangent line.

$$x\text{-intercept of tangent line is } x = \underline{\hspace{2cm}} \quad (4.13)$$

The idea of approximating the function by its tangent line gives us an algorithm for approximating the root of a function:

- Given a value of x that is a decent approximation of the root, draw a tangent line to $f(x)$ at that point.
- Find where the tangent line intersects the x axis.
- Use this intersection as the new x value and repeat.

The first step has been shown for you in Figure 4.7. The tangent line to the function $f(x)$ at the point $(x_0, f(x_0))$ is shown in red. The x -intercept of the tangent line is the new x value, x_1 . The process is then repeated with x_1 as the new x_0 and so on.

The graphical method illustrated in Figure 4.7 was introduced by Sir Isaac Newton and is therefore known as Newton's Method. Joseph Raphson then gave the algebraic formulation and popularised the method, which is therefore also known as the Newton-Raphson method.

Exercise 4.29. If we had started not at x_0 in Figure 4.7 but instead at the very end of the x -axis in that figure, what would have happened? Would this initial guess have worked to eventually approximate the root?

Exercise 4.30. Sketch some other function $f(x)$ with a root and choose an initial point x_0 and graphically perform the Newton iteration a few times, similar to Figure 4.7. Does the algorithm appear to converge to the root in your example? Do you think that this will generally take more or fewer steps than the Bisection Method?

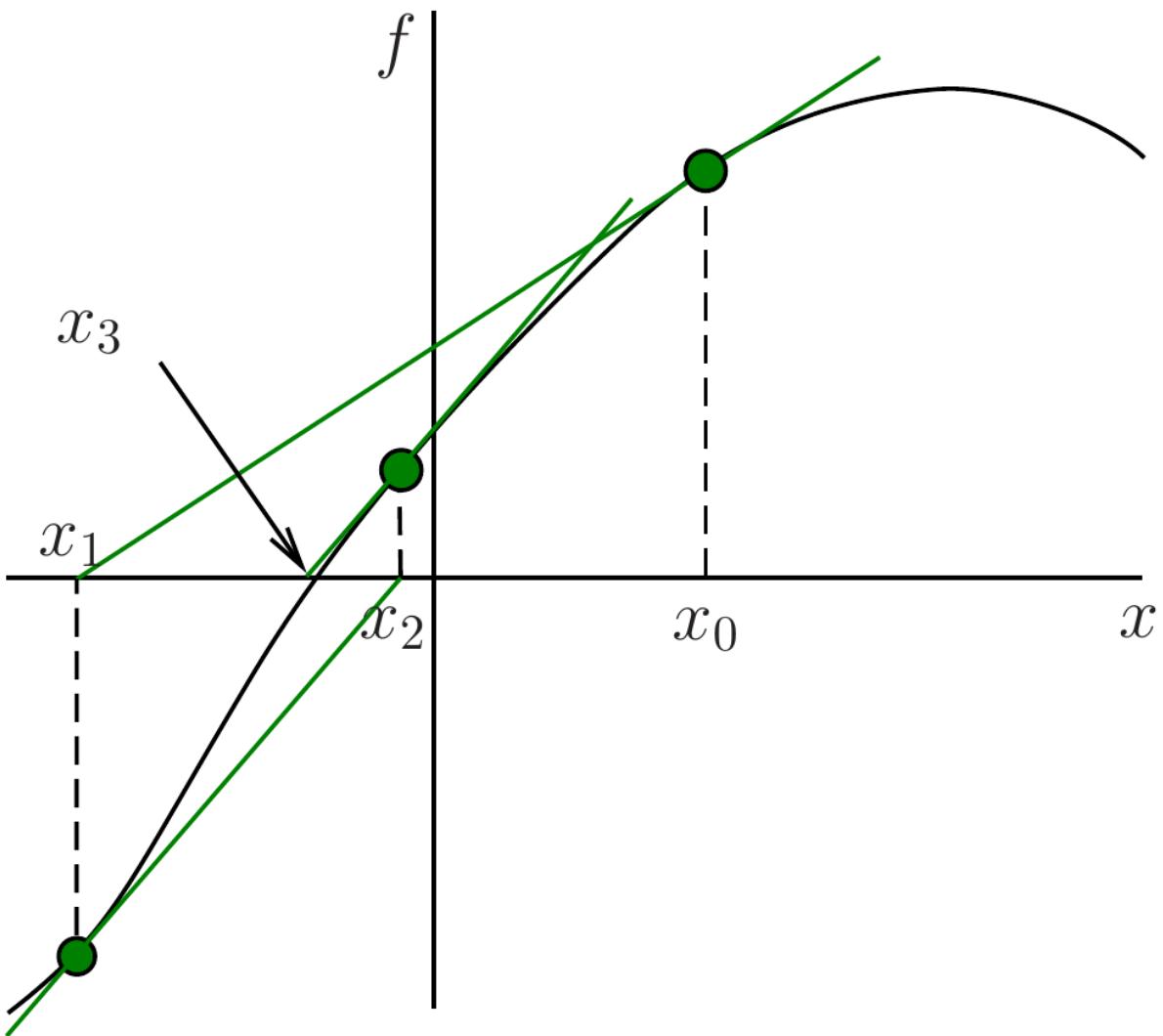


Figure 4.7: Using successive tangent line approximations to find the root of a function

Exercise 4.31. Consider the function $f(x) = \sin(x) + 1$. It has roots at $x = 2n\pi + 3\pi/2$ for $n \in \mathbb{Z}$. However at the roots we have $f'(x) = 0$. Make yourself a sketch to see why. What will happen when you apply Newton's method with a starting value of $x_0 = \pi$? You should be able to answer this just by looking at the sketch.

Exercise 4.32. Using your result from Exercise 4.28, write the formula for the x -intercept of the tangent line to $f(x)$ at the point $(x_n, f(x_n))$. This is the formula for the next guess x_{n-1} in Newton's Method. Newton's method is a fixed point iteration method of the form $x_{n+1} = g(x_n)$ with

$$g(x) = \dots$$

Fill in the blank in the above formula.

Exercise 4.33. Apply Newton's method to find the root of the function $f(x) = x^2 - 2$ with an initial guess of $x_0 = 1$. Calculate the first two iterations of the sequence by hand (you do not need a calculator or computer for this). Use a calculator or computer to calculate the next two iterations and fill in the following table:

n	x_n	$f(x_n)$	$f'(x_n)$
0	$x_0 = 1$	$f(x_0) = -1$	$f'(x_0) = 2$
1	$x_1 = 1 - \frac{-1}{2} = \frac{3}{2}$	$f(x_1) =$	$f'(x_1) =$
2	$x_2 =$	$f(x_2) =$	$f'(x_2) =$
3	$x_3 =$	$f(x_3) =$	$f'(x_3) =$
4	$x_4 =$		

4.4.2 Implementation

Exercise 4.34. The following is an outline a Python function called `newton()` for Newton's method. The function needs to accept a Python function for $f(x)$, a Python function for $f'(x)$, an initial guess, and an optional error tolerance.

```

def newton(f, fprime, x0, tol=1e-12):
    """
    Find root of f(x) using Newton's Method.

    Parameters:
        f (function): Function whose root we want to find
        fprime (function): Derivative of f
        x0 (float): Initial guess for the root
        tol (float, optional): Error tolerance. Defaults to 1e-6

    Returns:
        float: Approximate root of f(x)
        or error message if method fails
    """
    # Set x equal to initial guess x0

    # Loop for maximum of 30 iterations:
    #   1. Calculate next Newton iteration:
    #       x_new = x - f(x)/f'(x)

    #   2. Check if within tolerance:
    #       If |x_new - x| < tol, return x_new as root
    #       If not, update x = x_new

    #   3. If loop completes without finding root,
    #       return message that method did not converge

```

Exercise 4.35. Use your implementation from Exercise 4.34 to approximate the root of the function $f(x) = x^2 - 2$ with an initial guess of $x_0 = 1$. Use a tolerance of 10^{-12} .

4.4.3 Failures

There are several ways in which Newton's Method will behave unexpectedly – or downright fail. Some of these issues can be foreseen by examining the Newton iteration formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \quad (4.14)$$

Some of the failures that we will see are a little more surprising. Also in this section we will look at the convergence rate of Newton's Method and we will show that we can greatly outperform the Bisection method.

Exercise 4.36. There are several reasons why Newton's method could fail. Work with your partners to come up with a list of reasons. Support each of your reasons with a sketch or an example.

Exercise 4.37. One of the failures of Newton's Method is that it requires a division by $f'(x_n)$. If $f'(x_n)$ is zero then the algorithm completely fails. Go back to your Python function and put an `if` statement in the function that catches instances when Newton's Method fails in this way.

Exercise 4.38. An interesting failure can occur with Newton's Method that you might not initially expect. Consider the function $f(x) = x^3 - 2x + 2$. This function has a root near $x = -1.77$. Fill in the table below by hand. You really do not need a computer for this. Then draw the tangent lines into Figure 4.8 for approximating the solution to $f(x) = 0$ with a starting point of $x = 0$.

n	x_n	$f(x_n)$	$f'(x_n)$
0	$x_0 = 0$	$f(x_0) = 2$	$f'(x_0) = -2$
1	$x_1 = 0 - \frac{f(x_0)}{f'(x_0)} = 1$	$f(x_1) = 1$	$f'(x_1) = 1$
2	$x_2 = 1 - \frac{f(x_1)}{f'(x_1)} =$	$f(x_2) =$	$f'(x_2) =$
3	$x_3 =$	$f(x_3) =$	$f'(x_3) =$
4	$x_4 =$	$f(x_4) =$	$f'(x_4) =$
:	:	:	:

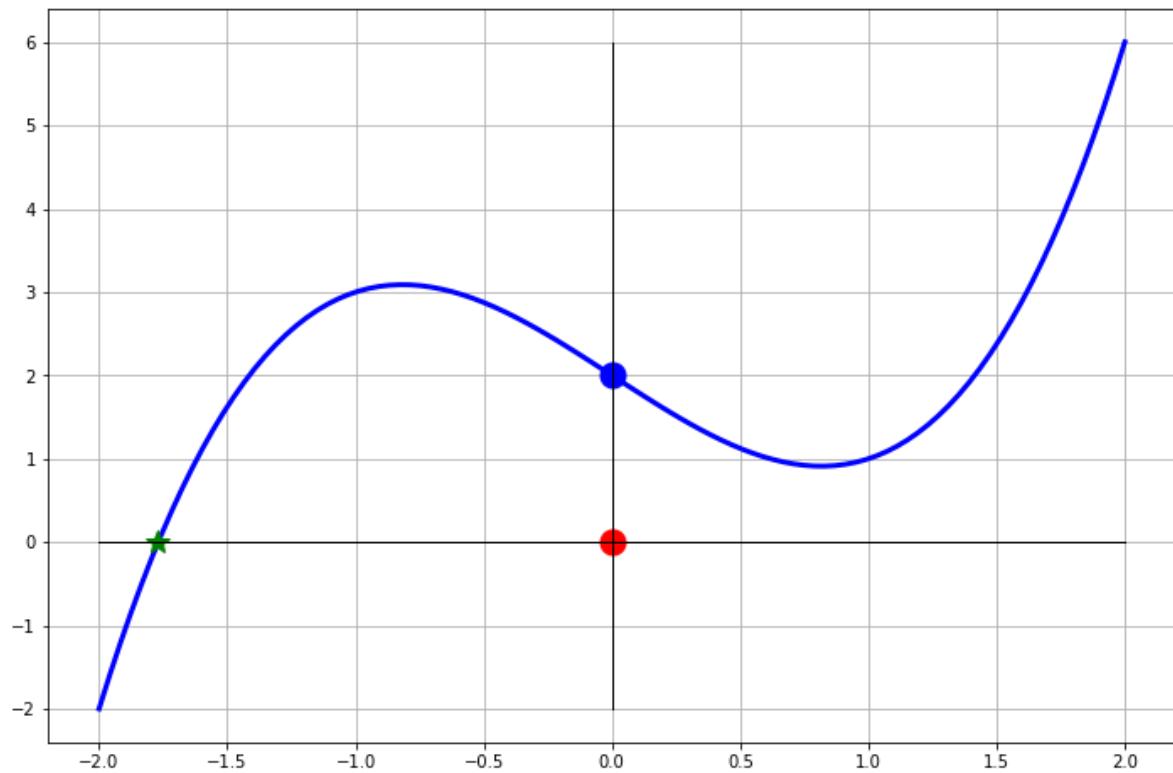


Figure 4.8: An interesting Newton's Method failure when $f(x) = x^3 - 2x + 2$.

Exercise 4.39. Now let us consider the function $f(x) = \sqrt[3]{x}$. This function has a root $x = 0$. Furthermore, it is differentiable everywhere except at $x = 0$ since

$$f'(x) = \frac{1}{3}x^{-2/3} = \frac{1}{3x^{2/3}}. \quad (4.15)$$

The point of this exercise is to show what can happen when the point of non-differentiability is precisely the point that you are looking for.

1. Fill in the table of iterations starting at $x = -1$, draw the tangent lines on the plot, and make a general observation of what is happening with the Newton iterations.

n	x_n	$f(x_n)$	$f(x_n)$
0	$x_0 = -1$	$f(x_0) = -1$	$f'(x_0) =$
1	$x_1 = -1 - \frac{f(-1)}{f'(-1)} =$	$f(x_1) =$	$f'(x_1) =$
2			
3			
4			
\vdots	\vdots	\vdots	\vdots

2. Now let us look at the Newton iteration in a bit more detail. Since $f(x) = x^{1/3}$ and $f'(x) = \frac{1}{3}x^{-2/3}$ the Newton iteration can be simplified as

$$x_{n+1} = x_n - \frac{x_n^{1/3}}{\left(\frac{1}{3}x_n^{-2/3}\right)} = x_n - 3\frac{x_n^{1/3}}{x_n^{-2/3}} = x_n - 3x_n = -2x_n. \quad (4.16)$$

What does this tell us about the Newton iterations?

Hint: You should have found the exact same thing in the numerical experiment in part 1.

3. Was there anything special about the starting point $x_0 = -1$? Will this problem exist for every starting point?

Exercise 4.40. Repeat the previous exercise with the function $f(x) = x^3 - 5x$ with the starting point $x_0 = -1$.

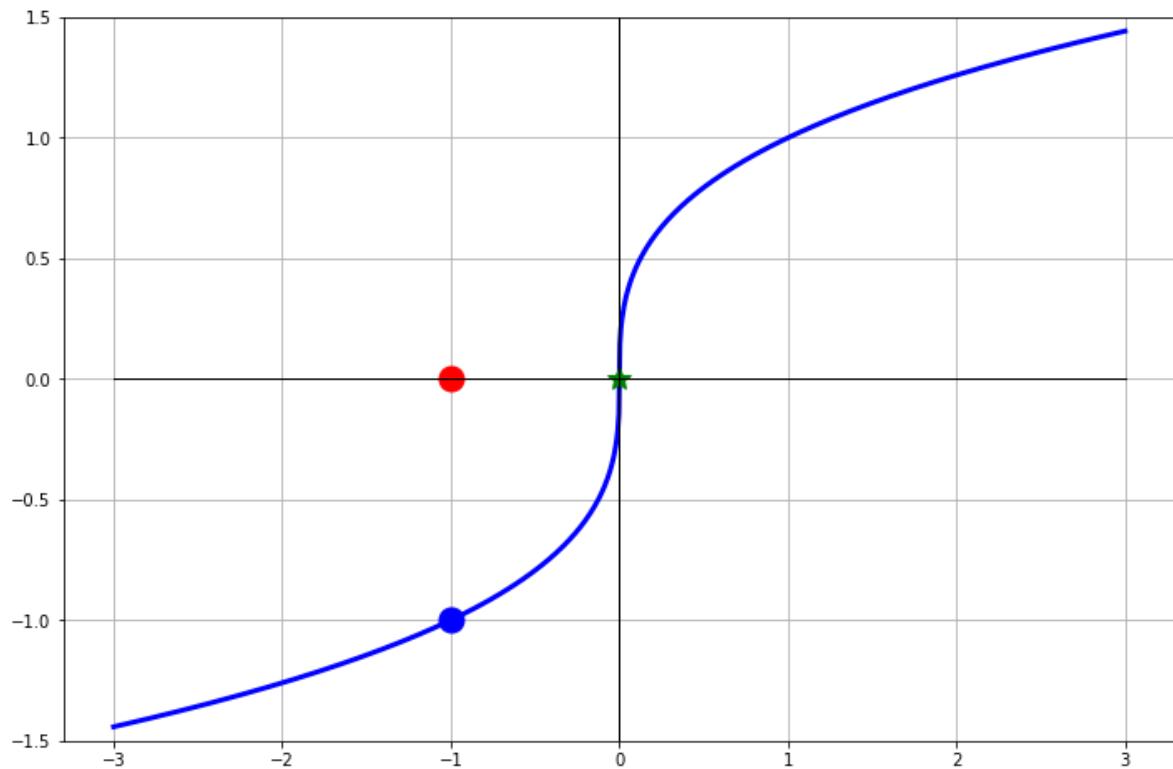


Figure 4.9: Another surprising Newton's Method failure when $f(x) = \sqrt[3]{x}$.

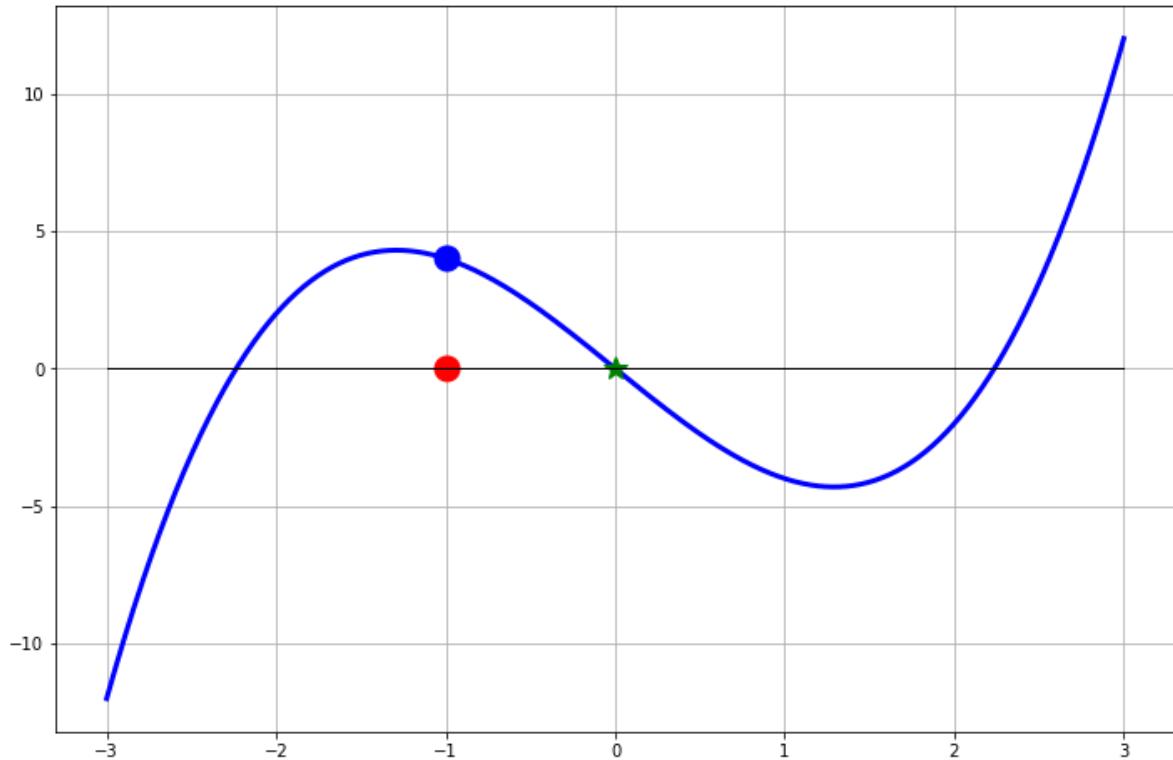


Figure 4.10: Another surprising Newton's Method failure when $f(x) = x^3 - 5x$.

4.4.4 Rate of Convergence

We saw in Example 4.2 how we could empirically determine the rate of convergence of the bisection method by plotting the error in the new iterate on the y -axis and the error in the old iterate on the x axis of a log-log plot. Take a look at that example again and make sure you understand the code and the explanation of the plot. Then, to investigate the rate of convergence of Newton's method you can do the same thing.

Exercise 4.41. Write a Python function `newton_with_error_tracking()` that returns a list of absolute errors between the iterates and the exact solution. You can start with your code from `newton()` from Exercise 4.34 and add the collection of the errors in a list as in the `bisection_with_error_tracking()` function that we wrote in Example 4.1. Your new function should accept a Python function for $f(x)$, a Python function for $f'(x)$, the exact root, an initial guess, and an optional error tolerance.

Then, use this function to list the error progression for Newton's method for finding the root of $f(x) = x^2 - 2$ with the initial guess $x_0 = 1$ and tolerance 10^{-12} . You should find that the error goes down extremely quickly and that for the final iteration Python can no longer detect a difference between the new iterate and its own value for $\sqrt{2}$.

Exercise 4.42.

1. Plot the errors you calculated in Exercise 4.41 using the `plot_error_progression()` function that we wrote in Example 4.2. Note that you will first need to remove the zeros from the error list, otherwise you will get an error. You can remind yourself of how to work with lists in Section 1.3.2.
 2. Give a thorough explanation for how to interpret the plot that you just made.
 3. Extract the slope and intercept of the line that you fitted to the log-log plot of the error progression using the `np.polyfit` function. What does this tell you about how the error at each iteration is related to the error at the previous iteration?
-

Exercise 4.43. Reproduce plots like those in the previous example but for the different equation that you used in Exercise 4.20. Again check that the plots have the expected shape.

4.5 The Secant Method

4.5.1 Intuition and Implementation

Newton's Method has second-order (quadratic) convergence and, as such, will perform faster than the Bisection method. However, Newton's Method requires that you have a function and a derivative of that function. The conundrum here is that sometimes the derivative is cumbersome or impossible to obtain but you still want to have the great quadratic convergence exhibited by Newton's method.

Recall that Newton's method is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \quad (4.17)$$

If we replace $f'(x_n)$ with an approximation of the derivative then we may have a method that is *close* to Newton's method in terms of convergence rate but is less troublesome to compute. Any method that replaces the derivative in Newton's method with an approximation is called a **Quasi-Newton Method**.

The first, and most obvious, way to approximate the derivative is just to use the slope of a secant line instead of the slope of the tangent line in the Newton iteration. If we choose two starting points that are quite close to each other then the slope of the secant line through those points will be approximately the same as the slope of the tangent line.

Exercise 4.44. Use the backward difference

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} \quad (4.18)$$

to approximate the derivative of f at x_n . Discuss why this approximates the derivative. Use this approximation of $f'(x_n)$ in the expression for x_{n+1} of Newton's method. Show that that the result simplifies to

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}. \quad (4.19)$$

Exercise 4.45. Notice that the iteration formula for x_{n+1} that you derived depends on both x_n and x_{n-1} . So to start the iteration you need to choose two points x_0 and x_1 before you can calculate x_2, x_3, \dots . Draw several pictures showing what the secant method does pictorially. Discuss whether it is important to choose these starting points close to the root and close to each other.

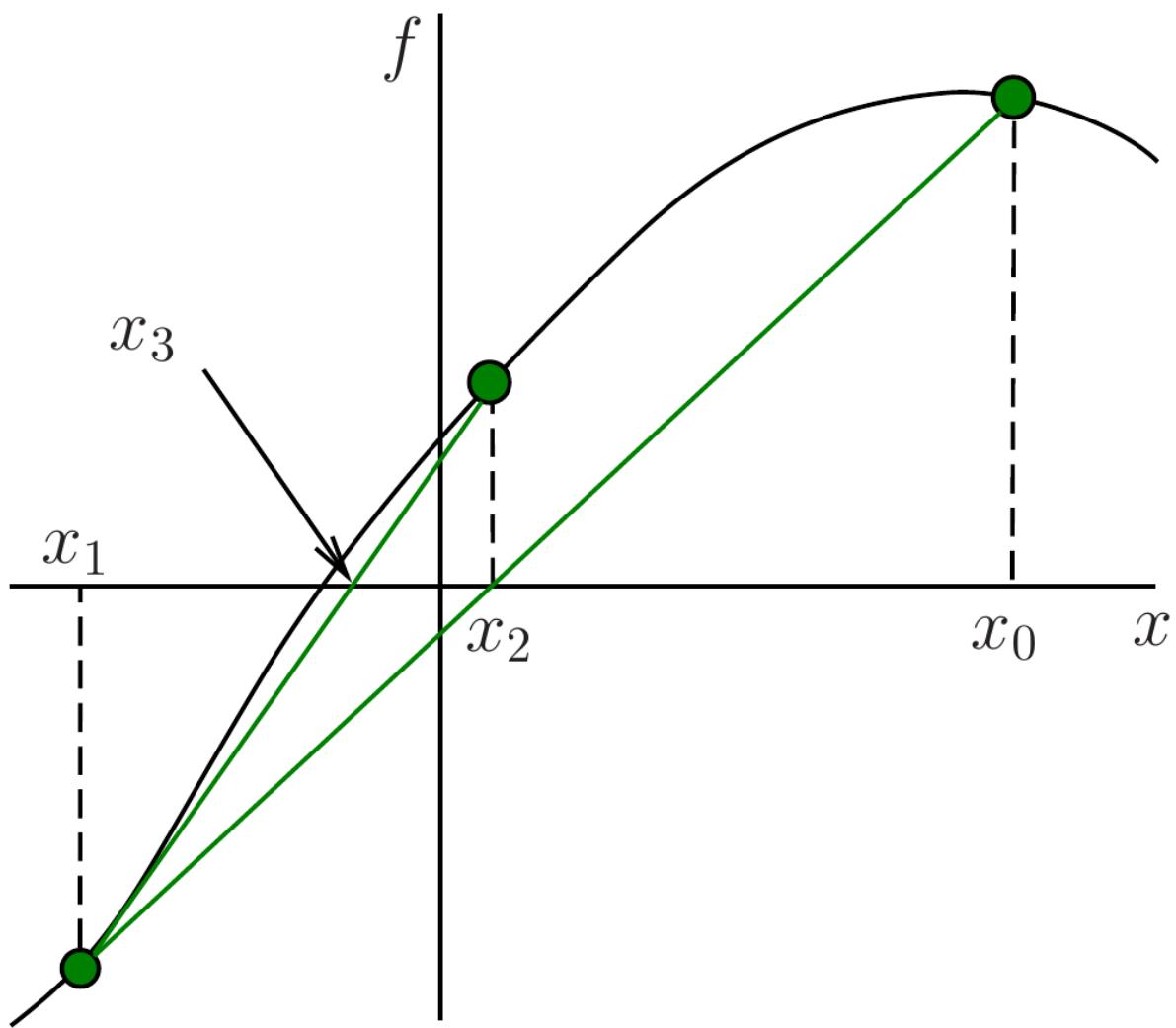


Figure 4.11: Using successive secant line approximations to find the root of a function

Exercise 4.46. Write a Python function for solving equations of the form $f(x) = 0$ with the secant method. Your function should accept a Python function, two starting points, and an optional error tolerance. Also write a test script that clearly shows that your code is working.

4.5.2 Analysis

Up to this point we have done analysis work on the Bisection Method and Newton's Method. We have found that the methods are first order and second order respectively. We end this chapter by doing the same for the Secant Method.

Exercise 4.47. Write a function `secant_with_error_tracking()` that returns a list of absolute errors between the iterates and the exact solution. You can start with your code from `secant()` from Exercise 4.46 and add the collection of the errors in a list as in the `bisection_with_error_tracking()` function that we wrote in Example 4.1. Your new function should accept a Python function, the exact root, two starting points, and an optional error tolerance.

Use this function to list the error progression for the secant method for finding the root of $f(x) = x^2 - 2$ with the starting points $x_0 = 1$ and $x_1 = 3$. Use a tolerance of 10^{-12} . Use the `plot_error_progression()` function that we wrote in Example 4.2 to plot the error progression.

Exercise 4.48. Using the `np.polyfit` function, extract the slope of the line that you had fit to the log-log plot of the error progression in Exercise 4.47. What does this slope tell you about how the error at each iteration is related to the error at the previous iteration?

Exercise 4.49. Make error progression plots for a few different functions and look at the slopes. What do you notice?

4.6 Order of Convergence

You will by now have noticed that Newton's method converges much faster than the bisection method. The secant method also converges faster than the bisection method, though not quite as fast as Newton's method. You will have observed that reflected in the different slopes of the error progression graphs. In this section we summarize your observations theoretically. Thus this section is untypical in that it does not contain further explorations for you to do but instead just consolidates what you have already done. In class this will be presented as a lecture.

4.6.1 Definition

Definition 4.1. Suppose that $x_n \rightarrow p$ as $n \rightarrow \infty$ and $x_n \neq p$ for all n . The sequence $\{x_n\}$ is said to have **order of convergence** $\alpha \geq 1$ if there exists a constant $\lambda > 0$ such that

$$\lim_{n \rightarrow \infty} \frac{E_{n+1}}{E_n^\alpha} = \lambda. \quad (4.20)$$

Here E_n denotes the absolute error in the n th approximation: $E_n = |x_n - p|$.

If $\alpha = 1, 2, 3, \dots$, the convergence is said to be *linear*, *quadratic*, *cubic*, ..., respectively. Note that if the convergence is linear, then the positive constant λ that appears in the above definition must be smaller than 1 ($0 < \lambda < 1$), because otherwise the sequence will not converge.

A sequence with a higher order of convergence converges much more rapidly than a sequence with a lower order of convergence. To see this, let us consider the following example:

Example 4.3. Let $\{x_n\}$ and $\{y_n\}$ be sequences converging to zero and let, for $n \geq 0$,

$$|x_{n+1}| = k|x_n| \quad \text{and} \quad |y_{n+1}| = k|y_n|^2, \quad (4.21)$$

where $0 < k < 1$. According to the definition, $\{x_n\}$ is linearly convergent and $\{y_n\}$ is quadratically convergent.

Also, we have

$$\begin{aligned} |x_n| &= k|x_{n-1}| = k^2|x_{n-2}| = \dots = k^n|x_0|, \\ |y_n| &= k|y_{n-1}|^2 = k|k|y_{n-2}|^2|^2 = k^3|y_{n-2}|^4 = k^7|y_{n-3}|^8 = \dots = k^{2^{n-1}}|y_0|^{2^n}. \end{aligned} \quad (4.22)$$

This illustrates that the quadratic convergence is much faster than the linear convergence.

We have defined the order of convergence of a converging sequence. We will also say than an iterative method has order of convergence α if the sequence of approximations that it produces on a generic problem has order of convergence α .

4.6.2 Fixed Point Iteration

Suppose that $g(x)$ satisfies the conditions of the Fixed Point Theorem on interval $[a, b]$, so that the sequence $\{x_n\}$ generated by the formula $x_{n+1} = g(x_n)$ with $x_0 \in [a, b]$ converges to a fixed point p . Then, using the Mean Value Theorem, we obtain

$$\begin{aligned} E_{n+1} &= |x_{n+1} - p| = |g(x_n) - g(p)| \\ &= |g'(\xi_n)(x_n - p)| = E_n |g'(\xi_n)|, \end{aligned} \quad (4.23)$$

where ξ_n is a number between x_n and p . This implies that if $x_n \rightarrow p$, then $\xi_n \rightarrow p$ as $n \rightarrow \infty$. Therefore,

$$\lim_{n \rightarrow \infty} \frac{E_{n+1}}{E_n} = |g'(p)|. \quad (4.24)$$

In general, $g'(p) \neq 0$, so that the fixed point iteration produces a linearly convergent sequence.

Can the fixed-point iteration produce convergent sequences with convergence of order 2, 3, etc.? It turns out that, under certain conditions, this is possible.

We will prove the following

Theorem 4.4. *Let $m > 1$ be an integer, and let $g \in C^m[a, b]$. Suppose that $p \in [a, b]$ is a fixed point of g , and a point $x_0 \in [a, b]$ exists such that the sequence generated by the formula $x_{n+1} = g(x_n)$ converges to p . If $g'(p) = \dots = g^{(m-1)}(p) = 0$, then $\{x_n\}$ has the order of convergence m .*

Proof. Expanding $g(x_n)$ in Taylor's series at point p , we obtain:

$$\begin{aligned} x_{n+1} &= g(x_n) = g(p) + (x_n - p)g'(p) + \dots \\ &\quad + \frac{(x_n - p)^{m-1}}{(m-1)!}g^{(m-1)}(p) \\ &\quad + \frac{(x_n - p)^m}{m!}g^{(m)}(\xi_n) \\ &= p + \frac{(x_n - p)^m}{(m)!}g^{(m)}(\xi_n), \end{aligned} \quad (4.25)$$

where ξ_n is between x_n and p and, therefore, in $[a, b]$ ($x_n \in [a, b]$ at least for sufficiently large n). Then we have

$$\begin{aligned} E_{n+1} &= |x_{n+1} - p| = |g(x_n) - p| = \left| \frac{(x_n - p)^m}{(m)!}g^{(m)}(\xi_n) \right| \\ &= E_n^m \frac{|g^{(m)}(\xi_n)|}{m!}. \end{aligned} \quad (4.26)$$

Therefore (using the fact that $\xi_n \rightarrow p$),

$$\lim_{n \rightarrow \infty} \frac{E_{n+1}}{E_n^m} = \frac{|g^{(m)}(p)|}{m!}, \quad (4.27)$$

which means that $\{x_n\}$ has convergence of order m . \square

4.6.3 Newton's Method

Newton's method for approximating the root p of the equation $f(x) = 0$ is equivalent to the fixed-point iteration $x_{n+1} = g(x_n)$ with

$$g(x) = x - \frac{f(x)}{f'(x)}. \quad (4.28)$$

Suppose that sequence $\{x_n\}$ converges to p and $f'(p) \neq 0$. We have

$$g'(x) = \frac{f(x)f''(x)}{[f'(x)]^2} \Rightarrow g'(p) = \frac{f(p)f''(p)}{[f'(p)]^2} = 0. \quad (4.29)$$

It follows from the above theorem that the order of convergence of Newton's method is 2 (except in the special case where $g''(p) = 0$).

4.6.4 Secant Method

The situation with the secant method is more complicated (since it cannot be reduced to the fixed point iteration) and requires a separate treatment. The result is that the secant method has order of convergence $\alpha = \frac{1+\sqrt{5}}{2} \approx 1.618$.

Note that α is known as the *golden ratio*. If you are intrigued to see the golden ratio appear in this context, you can find a proof below. If you are happy to just accept the miracle, you can skip the proof and go on to Section 4.7.

Suppose that a sequence $\{x_n\}$, generated by the secant method

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}, \quad (4.30)$$

converges to p . Let

$$e_n = x_n - p, \quad (4.31)$$

so that $E_n = |e_n|$, and we assume that $E_n \ll 1$, which is definitely true for sufficiently large n (since the sequence $\{x_n\}$ is converging to p). Subtracting p from both sides of Eq. 4.30, we obtain

$$e_{n+1} = e_n - \frac{f(p + e_n)(e_n - e_{n-1})}{f(p + e_n) - f(p + e_{n-1})}, \quad (4.32)$$

Expanding $f(p + e_n)$ and $f(p + e_{n-1})$ in Taylor series about p and taking into account that $f(p) = 0$, we find that

$$\begin{aligned} f(p + e_n) &= e_n f'(p) + \frac{e_n^2}{2} f''(p) + \dots \\ &= e_n f'(p)(1 + e_n Q) + \dots, \\ f(p + e_{n-1}) &= e_{n-1} f'(p) + \frac{e_{n-1}^2}{2} f''(p) + \dots \\ &= e_{n-1} f'(p)(1 + e_{n-1} Q) + \dots, \end{aligned} \tag{4.33}$$

where

$$Q = \frac{f''(p)}{2f'(p)}. \tag{4.34}$$

Substitution of Eq. 4.33 into Eq. 4.32 yields

$$\begin{aligned} e_{n+1} &= e_n - \frac{e_n(e_n - e_{n-1})f'(p)(1 + e_n Q) + \dots}{f'(p)[e_n - e_{n-1} + Q(e_n^2 - e_{n-1}^2) + \dots]} \\ &= e_n \left(1 - \frac{1 + e_n Q + \dots}{1 + Q(e_n + e_{n-1}) + \dots} \right). \end{aligned} \tag{4.35}$$

Since, for small x ,

$$\frac{1}{1 + x + \dots} = 1 - x + \dots, \tag{4.36}$$

we obtain

$$\begin{aligned} e_{n+1} &= e_n (1 - (1 + e_n Q + \dots)(1 - Q(e_n + e_{n-1}) + \dots)) \\ &= Q e_n e_{n-1} + \dots. \end{aligned} \tag{4.37}$$

Thus, for sufficiently large n , we have

$$e_{n+1} \approx Q e_n e_{n-1}. \tag{4.38}$$

Hence,

$$E_{n+1} \approx |Q| E_n E_{n-1}. \tag{4.39}$$

Now we assume that (for all sufficiently large n)

$$E_{n+1} \approx \lambda E_n^\alpha, \tag{4.40}$$

where λ and α are positive constants. Substituting Eq. 4.40 into Eq. 4.39, we find

$$\lambda E_n^\alpha \approx |Q| E_n E_{n-1} \quad \text{or} \quad \lambda E_n^{\alpha-1} \approx |Q| E_{n-1}. \tag{4.41}$$

Applying Eq. 4.40 one more time (with n replaced by $n - 1$), we obtain

$$\lambda (\lambda E_{n-1}^\alpha)^{\alpha-1} \approx |Q| E_{n-1} \tag{4.42}$$

or, equivalently,

$$\lambda^\alpha E_{n-1}^{\alpha(\alpha-1)} \approx |Q| E_{n-1}. \quad (4.43)$$

The last equation will be satisfied provided that

$$\lambda^\alpha = |Q|, \quad \alpha(\alpha - 1) = 1, \quad (4.44)$$

which requires that

$$\lambda = |Q|^{1/\alpha}, \quad \alpha = (1 + \sqrt{5})/2 \approx 1.62. \quad (4.45)$$

Thus, we have shown that if $\{x_n\}$ is a convergent sequence generated by the secant method, then

$$\lim_{n \rightarrow \infty} \frac{E_{n+1}}{E_n^\alpha} = |Q|^{1/\alpha}. \quad (4.46)$$

Thus, the secant method has *superlinear* convergence.

Further reading: Section 2.4 of (Burden and Faires 2010).

4.7 Algorithm Summaries

The following four problems are meant to have you re-build each of the algorithms that we developed in this chapter. Write all of the mathematical details completely and clearly. Do not just write “how” the method works, but give all of the mathematical details for “why” it works.

Exercise 4.50. Let $f(x)$ be a continuous function on the interval $[a, b]$ where $f(a) \cdot f(b) < 0$. Clearly give all of the mathematical details for how the Bisection Method approximates the root of the function $f(x)$ in the interval $[a, b]$.

Exercise 4.51. Let $f(x)$ be a differentiable function with a root *near* $x = x_0$. Clearly give all of the mathematical details for how Newton’s Method approximates the root of the function $f(x)$.

Exercise 4.52. Let $f(x)$ be a continuous function with a root *near* $x = x_0$. Clearly give all of the mathematical details for how the Secant Method approximates the root of the function $f(x)$.

4.8 Problems

Exercise 4.53. Can the Bisection Method or Newton's Method be used to find the roots of the function $f(x) = \cos(x) + 1$? Explain why or why not for each technique? What is the rate of convergence? Be careful here, the answer is surprising.

Exercise 4.54. How many iterations of the bisection method are necessary to approximate $\sqrt{3}$ to within $10^{-3}, 10^{-4}, \dots, 10^{-15}$ using the initial interval $[a, b] = [0, 2]$? See Theorem 4.2.

Exercise 4.55. Refer back to Example 4.1 and demonstrate that you get the same results for the order of convergence when solving the problem $x^3 - 3 = 0$. Generate versions of all of the plots from that example and give thorough descriptions of what you learn from each plot.

Exercise 4.56. In this problem you will demonstrate that all of your root finding codes work. At the beginning of this chapter we proposed the equation solving problem

$$3 \sin(x) + 9 = x^2 - \cos(x). \quad (4.47)$$

Write a script that calls upon your Bisection, Newton, and Secant methods one at a time to find the positive solution to this equation. Your script needs to output the solutions in a clear and readable way so you can tell which answer came from which root finding algorithm.

Exercise 4.57. In Figure 4.12 you see six different log-log plots of the new error to the old error for different root finding techniques. What is the order of the approximate convergence rate for each of these methods?

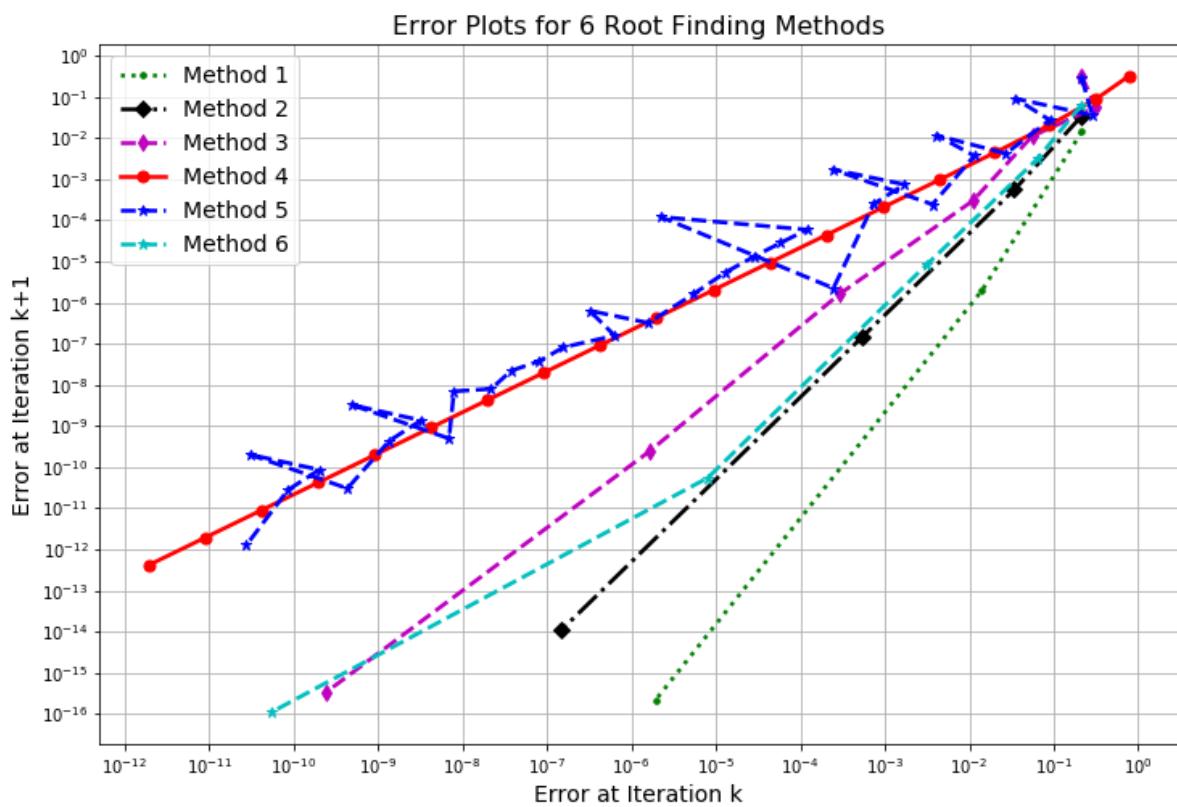


Figure 4.12: Six Error Plots

- d. In your own words, what does it mean for a root finding method to have a “first order convergence rate?” “Second order convergence rate?” etc.
-

Exercise 4.58. There are MANY other root finding techniques beyond the four that we have studied thus far. We can build these methods using Taylor Series as follows:

Near $x = x_0$ the function $f(x)$ is approximated by the Taylor Series

$$f(x) \approx y = f(x_0) + \sum_{n=1}^N \frac{f^{(n)}(x_0)}{n!} (x - x_0)^n \quad (4.48)$$

where N is a positive integer. In a root-finding algorithm we set y to zero to find the root of the approximation function. The root of this function *should* be close to the actual root that we are looking for. Therefore, to find the next iterate we solve the equation

$$0 = f(x_0) + \sum_{n=1}^N \frac{f^{(n)}(x_0)}{n!} (x - x_0)^n \quad (4.49)$$

for x . For example, if $N = 1$ then we need to solve $0 = f(x_0) + f'(x_0)(x - x_0)$ for x . In doing so we get $x = x_0 - f(x_0)/f'(x_0)$. This is exactly Newton’s method. If $N = 2$ then we need to solve

$$0 = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!} (x - x_0)^2 \quad (4.50)$$

for x .

- (a) Solve for x in the case that $N = 2$. Then write a Python function that implements this root-finding method.
 - (b) Demonstrate that your code from part (a) is indeed working by solving several problems where you know the exact solution.
 - (c) Show several plots that estimates the order of the method from part (a). That is, create a log-log plot of the successive errors for several different equation-solving problems.
 - (d) What are the pro’s and con’s to using this new method?
-

Exercise 4.59. (modified from (Burden and Faires 2010)) An object falling vertically through the air is subject to friction due to air resistance as well as gravity. The function describing the position of such an object is

$$s(t) = s_0 - \frac{mg}{k}t + \frac{m^2g}{k^2} \left(1 - e^{-kt/m}\right), \quad (4.51)$$

where m is the mass measured in kg, g is gravity measured in meters per second per second, s_0 is the initial position measured in meters, and k is the coefficient of air resistance.

1. What are the dimensions of the parameter k ?
 2. If $m = 1\text{kg}$, $g = 9.8\text{m/s}^2$, $k = 0.1\text{kg/s}$, and $s_0 = 100\text{m}$ how long will it take for the object to hit the ground? Find your answer to within 0.01 seconds.
 3. The value of k depends on the aerodynamics of the object and might be challenging to measure. We want to perform a sensitivity analysis on your answer to part (b) subject to small measurement errors in k . If the value of k is only known to within 10% then what are your estimates of when the object will hit the ground?
-

Exercise 4.60. In Single Variable Calculus you studied methods for finding local and global extrema of functions. You likely recall that part of the process is to set the first derivative to zero and to solve for the independent variable (remind yourself why you are doing this). The trouble with this process is that it may be very very challenging to solve by hand. This is a perfect place for Newton's method or any other root finding technique!

Find the local extrema for the function $f(x) = x^3(x - 3)(x - 6)^4$ using numerical techniques where appropriate.

Exercise 4.61. (`scipy.optimize.fsolve()`) The `scipy` library in Python has many built-in numerical analysis routines much like the ones that we have built in this chapter. Of particular interest to the task of root finding is the `fsolve` command in the `scipy.optimize` library.

1. Go to the [help documentation](#) for `scipy.optimize.fsolve` and make yourself familiar with how to use the tool.
2. First solve the equation $x \sin(x) - \log(x) = 0$ for x starting at $x_0 = 3$.
 1. Make a plot of the function on the domain $[0, 5]$ so you can eyeball the root before using the tool.
 2. Use the `scipy.optimize.fsolve()` command to approximate the root.
 3. Fully explain each of the outputs from the `scipy.optimize.fsolve()` command. You should use the `fsolve()` command with `full_output=1` so you can see all of the solver diagnostics.
3. Demonstrate how to use `fsolve()` using any non-trivial nonlinear equation solving problem. Demonstrate what some of the options of `fsolve()` do.

4. The `scipy.optimize.fsolve()` command can also solve systems of equations (something we have not built algorithms for in this chapter). Consider the system of equations

$$\begin{aligned}x_0 \cos(x_1) &= 4 \\x_0 x_1 - x_1 &= 5\end{aligned}\tag{4.52}$$

The following Python code allows you to use `scipy.optimize.fsolve()` so solve this system of nonlinear equations in much the same way as we did in part (b) of this problem. However, be aware that we need to think of `x` as a vector of x -values. Go through the code below and be sure that you understand every line of code.

```
import numpy as np
from scipy.optimize import fsolve

def F(x):
    return [x[0]*np.cos(x[1])-4, x[0]*x[1] - x[1] - 5]

fsolve(F, [6,1], full_output=1)
# Note: full_output=1 gives the solver diagnostics
```

5. Solve the system of nonlinear equations below using `.fsolve()`.

$$\begin{aligned}x^2 - xy^2 &= 2 \\xy &= 2\end{aligned}\tag{4.53}$$

4.9 Projects

At the end of every chapter we propose a few projects related to the content in the preceding chapter(s). In this section we propose two ideas for a project related to numerical algebra. The projects in this book 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.

4.9.1 Basins of Attraction

Let $f(x)$ be a differentiable function with several roots. Given a starting x value we should be able to apply Newton's Method to that starting point and we will converge to one of the roots (so long as you are not in one of the special cases discussed earlier in the chapter). It stands to reason that starting points *near* each other should all end up at the same root, and for some functions this is true. However, it is not true in general.

A **basin of attraction** for a root is the set of x values that converges to that root under Newton iterations. In this problem you will produce coloured plots showing the basins of attraction for all of the following functions. Do this as follows:

- Find the actual roots of the function by hand (this should be easy on the functions below).
- Assign each of the roots a different colour.
- Pick a starting point on the x axis and use it to start Newton's Method.
- Colour the starting point according to the root that it converges to.
- Repeat this process for many many starting points so you get a coloured picture of the x axis showing where the starting points converge to.

The set of points that are all the same colour are called the **basin of attraction** for the root associated with that colour. In Figure 4.13 there is an image of a sample basin of attraction image.

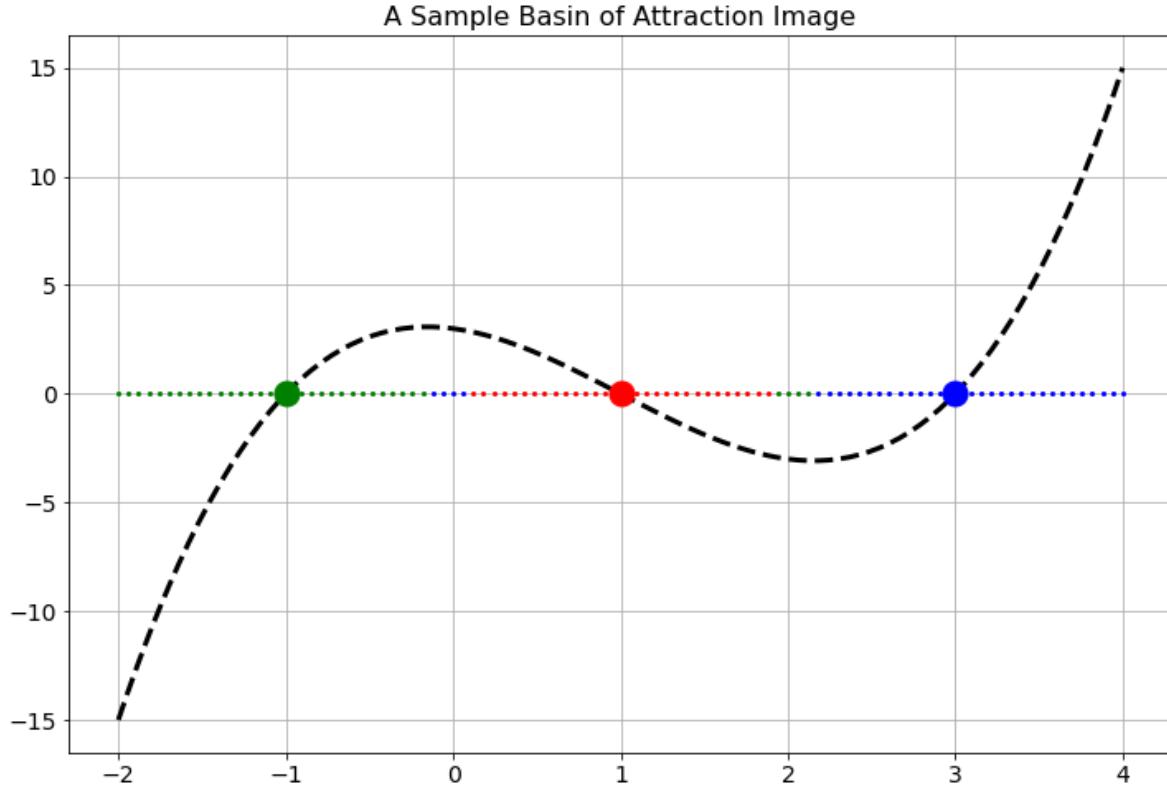


Figure 4.13: A sample basin of attraction image for a cubic function.

1. Create a basin on attraction image for the function $f(x) = (x - 4)(x + 1)$.
2. Create a basin on attraction image for the function $g(x) = (x - 1)(x + 3)$.
3. Create a basin on attraction image for the function $h(x) = (x - 4)(x - 1)(x + 3)$.
4. Find a non-trivial single-variable function of your own that has an interesting picture of the basins of attraction. In your write up explain why you thought that this was an interesting function in terms of the basins of attraction.
5. Now for the fun part! Consider the function $f(z) = z^3 - 1$ where z is a complex variable. That is, $z = x + iy$ where $i = \sqrt{-1}$. From the Fundamental Theorem of Algebra we know that there are three roots to this polynomial in the complex plane. In fact, we know that the roots are $z_0 = 1$, $z_1 = \frac{1}{2}(-1 + \sqrt{3}i)$, and $z_2 = \frac{1}{2}(-1 - \sqrt{3}i)$ (you should stop now and check that these three numbers are indeed roots of the polynomial $f(z)$). Your job is to build a picture of the basins of attraction for the three roots in the complex plane. This picture will naturally be two-dimensional since numbers in the complex plane are two dimensional (each has a real and an imaginary part). When you have your picture give a thorough write up of what you found.
6. Now pick your favourite complex-valued function and build a picture of the basins of attraction. Consider this an art project! See if you can come up with the prettiest basin of attraction picture.

4.9.2 Artillery

An artillery officer wishes to fire his cannon on an enemy brigade. He wants to know the angle to aim the cannon in order to strike the target. If we have control over the initial velocity of the cannon ball, v_0 , and the angle of the cannon above horizontal, θ , then the initial vertical component of the velocity of the ball is $v_y(0) = v_0 \sin(\theta)$ and the initial horizontal component of the velocity of the ball is $v_x(0) = v_0 \cos(\theta)$. In this problem we will assume the following:

- We will neglect air resistance¹ so, for all time, the differential equations $v'_y(t) = -g$ and $v'_x(t) = 0$ must both hold.
- We will assume that the position of the cannon is the origin of a coordinate system so $s_x(0) = 0$ and $s_y(0) = 0$.
- We will assume that the target is at position (x_*, y_*) which you can measure accurately relative to the cannon's position. The landscape is relatively flat but y_* could be a bit higher or a bit lower than the cannon's position.

¹Strictly speaking, neglecting air resistance is a poor assumption since a cannon ball moves fast enough that friction with the air plays a non-negligible role. However, the assumption of no air resistance greatly simplifies the maths and makes this version of the problem more tractable. The second version of the artillery problem in Chapter 8 will look at the effects of air resistance on the cannon ball.

Use the given information to write a nonlinear equation² that relates x_* , y_* , v_0 , g , and θ . We know that $g = 9.8m/s^2$ is constant and we will assume that the initial velocity can be adjusted between $v_0 = 100m/s$ and $v_0 = 150m/s$ in increments of $10m/s$. If we then are given a fixed value of x_* and y_* the only variable left to find in your equation is θ . A numerical root-finding technique can then be applied to your equation to approximate the angle. Create several look up tables for the artillery officer so they can be given v_0 , x_* , and y_* and then use your tables to look up the angle at which to set the cannon. Be sure to indicate when a target is out of range.

Write a brief technical report detailing your methods. Support your work with appropriate mathematics and plots. Include your tables at the end of your report.

²Hint: Symbolically work out the amount of time that it takes until the vertical position of the cannon ball reaches y_* . Then substitute that time into the horizontal position, and set the horizontal position equation to x_* .

5 Derivatives

The calculus was the first achievement of modern mathematics and it is difficult to overestimate its importance.

—Hungarian-American Mathematician John von Neumann

The primary goal of this chapter is to build a solid understanding of the basic techniques for numerical differentiation. This will be crucial in the later chapters of this book when we do optimisation and when we numerically integrate ordinary and partial differential equations.

We use this chapter also to comment on the importance of writing vectorized code using NumPy. And this chapter will again let us investigate truncation errors that we first discussed in Chapter 3. This is because in this chapter too, we will make use of Taylor series that we need to truncate at some order to give us practical methods.

5.1 Finite Differences

5.1.1 The First Derivative

Exercise 5.1. Recall from your first-semester Calculus class that the derivative of a function $f(x)$ is defined as

$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}. \quad (5.1)$$

A Calculus student proposes that it would just be much easier if we dropped the limit and instead just always choose Δx to be some small number, like 0.001 or 10^{-6} . Discuss the following questions:

1. When might the Calculus student's proposal actually work pretty well in place of calculating an actual derivative?
 2. When might the Calculus student's proposal fail in terms of approximating the derivative?
-

In this section we will build several approximation of first and second derivatives. The primary idea for each of these approximations is:

- Partition the interval $[a, b]$ into N sub intervals
- Define the distance between two points in the partition as Δx .
- Approximate the derivative at any point x in the interval $[a, b]$ by using linear combinations of $f(x - \Delta x)$, $f(x)$, $f(x + \Delta x)$, and/or other points in the partition.

Partitioning the interval into discrete points turns the continuous problem of finding a derivative at every real point in $[a, b]$ into a discrete problem where we calculate the approximate derivative at finitely many points in $[a, b]$.

This distance Δx between neighbouring points in the partition is often referred to as the **step size**. It is also common to denote the step size by the letter h . We will use both notations for the step size interchangeably, using mostly h in this section on differentiation and Δx in the next section on integration. Note that in general the points in the partition do not need to be equally spaced, but that is the simplest place to start. Figure 5.1 shows a depiction of the partition as well as making clear that h is the separation between each of the points in the partition.

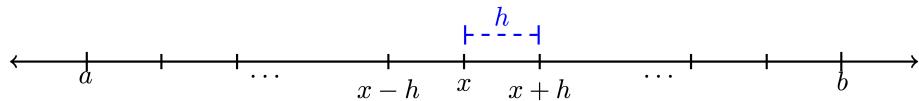


Figure 5.1: A partition of the interval $[a, b]$.

Exercise 5.2. Let us take a close look at partitions before moving on to more details about numerical differentiation.

1. If we partition the interval $[0, 1]$ into 3 equal sub intervals each with length h then:
 - $h = \underline{\hspace{2cm}}$
 - $[0, 1] = [0, \underline{\hspace{1cm}}] \cup [\underline{\hspace{1cm}}, \underline{\hspace{1cm}}] \cup [\underline{\hspace{1cm}}, 1]$
 - There are four total points that define the partition. They are $0, \underline{\hspace{1cm}}, \underline{\hspace{1cm}}, 1$.
2. If we partition the interval $[3, 7]$ into 5 equal sub intervals each with length h then:
 - $h = \underline{\hspace{2cm}}$
 - $[3, 7] = [3, \underline{\hspace{1cm}}] \cup [\underline{\hspace{1cm}}, \underline{\hspace{1cm}}] \cup [\underline{\hspace{1cm}}, \underline{\hspace{1cm}}] \cup [\underline{\hspace{1cm}}, \underline{\hspace{1cm}}] \cup [\underline{\hspace{1cm}}, 7]$

- c. There are 6 total points that define the partition. They are 0, __, __, __, __, 7.
3. More generally, if a closed interval $[a, b]$ contains N equal sub intervals where

$$[a, b] = \underbrace{[a, a+h] \cup [a+h, a+2h] \cup \cdots \cup [b-2h, b-h]}_{N \text{ total sub intervals}} \cup [b-h, b] \quad (5.2)$$

then the length of each sub interval, h , is given by the formula

$$h = \frac{b-a}{N}. \quad (5.3)$$

Exercise 5.3. In Python's `numpy` library there is a nice tool called `np.linspace()` that partitions an interval in exactly the way that we want. The command takes the form `np.linspace(a, b, n)` where the interval is $[a, b]$ and n the number of points used to create the partition. For example, `np.linspace(0, 1, 5)` will produce the list of numbers 0, 0.25, 0.5, 0.75, 1. Notice that there are 5 total points, the first point is a , the last point is b , and there are 4 total sub intervals in the partition. Hence, if we want to partition the interval $[0, 1]$ into 20 equal sub intervals then we would use the command `np.linspace(0, 1, 21)` which would result in a list of numbers starting with 0, 0.05, 0.1, 0.15, etc. What command would you use to partition the interval $[5, 10]$ into 100 equal sub intervals?

Exercise 5.4. Consider the Python command `np.linspace(0, 1, 50)`.

1. What interval does this command partition?
 2. How many points are going to be returned?
 3. How many equal length subintervals will we have in the resulting partition?
 4. What is the length of each of the subintervals in the resulting partition?
-

Now let us get back to the discussion of numerical differentiation. If we recall that the definition of the first derivative of a function is

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}. \quad (5.4)$$

our first approximation for the first derivative is naturally

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x)}{h} =: \Delta f(x). \quad (5.5)$$

In this approximation of the derivative we have simply removed the limit and instead approximated the derivative as the slope. It should be clear that this approximation is only good if the step size h is *small*. In Figure 5.2 we see a graphical depiction of what we are doing to approximate the derivative. The slope of the tangent line ($\Delta y / \Delta x$) is what we are after, and a way to approximate it is to calculate the slope of the secant line formed by looking h units forward from the point x .

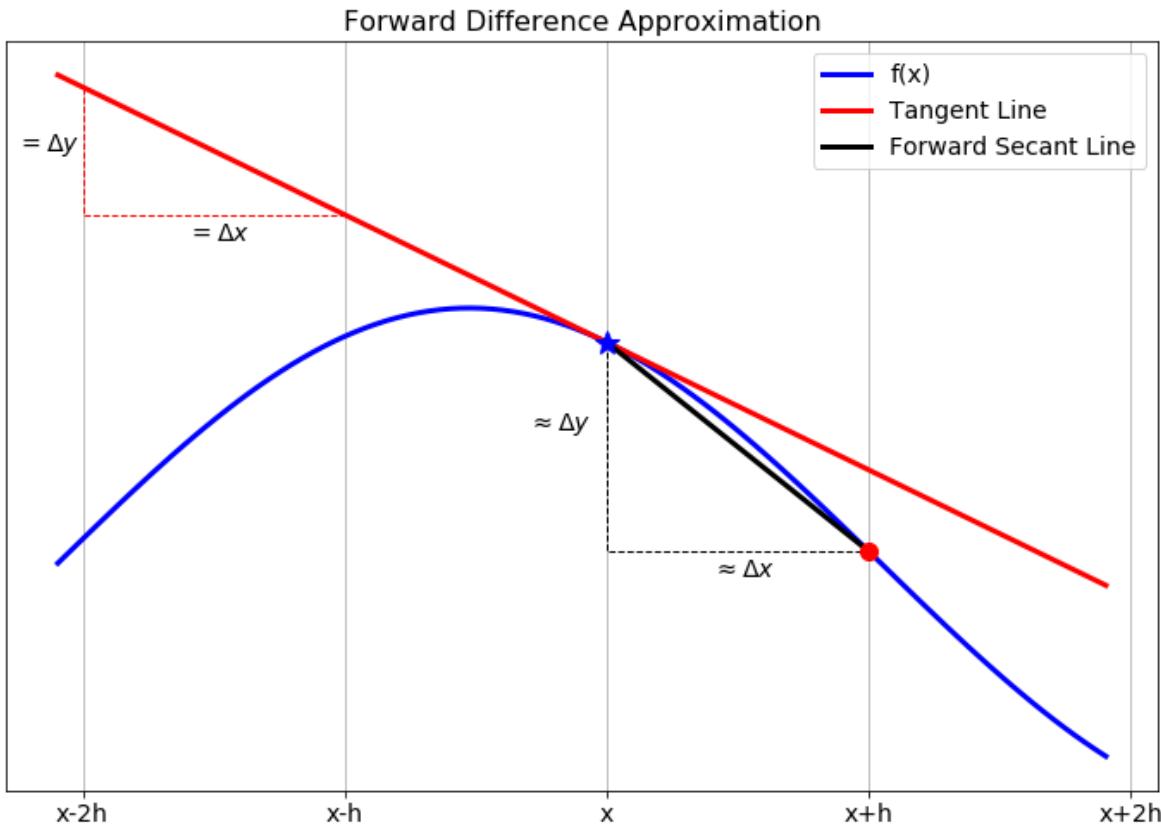


Figure 5.2: The forward difference differentiation scheme for the first derivative.

While this is the simplest and most obvious approximation for the first derivative there is a much more elegant technique, using Taylor series, for arriving at this approximation. Furthermore, the Taylor series technique gives us information about the approximation error and later will suggest an infinite family of other techniques.

5.1.2 Truncation error

Exercise 5.5. From Taylor's Theorem we know that for an infinitely differentiable function $f(x)$,

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0)^1 + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f^{(3)}(x_0)}{3!}(x - x_0)^3 + \dots \quad (5.6)$$

What do we get if we replace every “ x ” in the Taylor Series with “ $x + h$ ” and replace every “ x_0 ” in the Taylor Series with “ x ”? In other words, in Figure 5.1 we want to centre the Taylor series at x and evaluate the resulting series at the point $x + h$.

$$f(x + h) = \underline{\hspace{10cm}} \quad (5.7)$$

Exercise 5.6. Solve the result from the previous exercise for $f'(x)$ to create an approximation for $f'(x)$ using $f(x+h)$, $f(x)$, and some higher order terms. (fill in the blanks and the question marks)

$$f'(x) = \frac{f(x + h) - ???}{??} + \underline{\hspace{10cm}} \quad (5.8)$$

Exercise 5.7. In the formula that you developed in Exercise 5.6, if we were to truncate after the first fraction and drop everything else (called the *remainder*), we know that we would be introducing a truncation error into our derivative computation. If h is taken to be very small then the first term in the remainder is the largest and everything else in the remainder can be ignored (since all subsequent terms should be extremely small ... pause and ponder this fact). Therefore, the amount of error we make in the derivative computation by dropping the remainder depends on the power of h in that first term in the remainder.

What is the power of h in the first term of the remainder from Exercise 5.6?

Definition 5.1 (Order of a Numerical Differentiation Scheme). The **order** of a numerical derivative is the power of the step size in the first term of the remainder of the rearranged Taylor Series. For example, a first order method will have “ h^1 ” in the first term of the remainder. A second order method will have “ h^2 ” in the first term of the remainder. Etc.

For sufficiently small step size h , the error that you make by truncating the series is dominated by the first term in the remainder, which is proportional to the power of h in that term. Hence, the **order** of a numerical differentiation scheme tells you how the error you are making by using the approximation scheme decreases as you decrease the step-size h .

Definition 5.2. (Big O Notation) We say that the error in a differentiation scheme is $\mathcal{O}(h)$ (read: “big O of h ”), if and only if there is a positive constant M such that

$$|\text{Error}| \leq M \cdot h \quad (5.9)$$

when h is sufficiently small. This is equivalent to saying that a differentiation method is “first order.”

More generally, we say that the error in a differentiation scheme is $\mathcal{O}(h^k)$ (read: “big O of h^k ”) if and only if there is a positive constant M such that

$$|\text{Error}| \leq M \cdot h^k. \quad (5.10)$$

when h is sufficiently small. This is equivalent to saying that a differentiation scheme is “ k^{th} order.”

Theorem 5.1. *The approximation you derived in Exercise 5.6 gives a first order approximation of the first derivative:*

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h). \quad (5.11)$$

*This is called the **forward difference approximation** of the first derivative.*

Exercise 5.8. Consider the function $f(x) = \sin(x)(1-x)$. The goal of this exercise is to make sense of the discussion of the “order” of the derivative approximation. You may want to pause first and reread the previous couple of pages.

- a. Find $f'(x)$ by hand.
- b. Use your answer to part (a) to verify that $f'(1) = -\sin(1) \approx -0.8414709848$.
- c. To approximate the first derivative at $x = 1$ numerically with the forward-difference approximation formula from Theorem 5.1 we calculate

$$f'(1) \approx \frac{f(1+h) - f(1)}{h} =: \Delta f(1). \quad (5.12)$$

We want to see how the error in the approximation behaves as h is made smaller and smaller. Fill in the table below with the derivative approximation and the absolute error

associated with each given h . You may want to use a spreadsheet to organize your data (be sure that you are working in radians!).

h	$\Delta f(1)$	$ f'(1) - \Delta f(1) $
$2^{-1} = 0.5$	$\frac{f(1+0.5)-f(1)}{0.5} \approx -0.99749$	0.15602
$2^{-2} = 0.25$	$\frac{f(1+0.25)-f(1)}{0.25} \approx -0.94898$	0.10751
$2^{-3} = 0.125$		
$2^{-4} = 0.0625$		
2^{-5}		
2^{-6}		
2^{-7}		
2^{-8}		
2^{-9}		
2^{-10}		

- d. There was nothing really special in part (c) about powers of 2. Use your spreadsheet to build similar tables for the following sequences of h :

$$\begin{aligned} h &= 3^{-1}, 3^{-2}, 3^{-3}, \dots \\ h &= 5^{-1}, 5^{-2}, 5^{-3}, \dots \\ h &= 10^{-1}, 10^{-2}, 10^{-3}, \dots \\ h &= \pi^{-1}, \pi^{-2}, \pi^{-3}, \dots \end{aligned} \tag{5.13}$$

- e. Observation: If you calculate a numerical derivative with a forward difference and then calculate the absolute error with a fixed value of h , then what do you expect to happen to the absolute error if you divide the value of h by some positive constant M ? It may be helpful at this point to go back to your table and include a column called the *error reduction factor* where you find the ratio of two successive absolute errors. Observe what happens to this error reduction factor as h gets smaller and smaller.
- f. What does your answer to part (e) have to do with the approximation order of the numerical derivative method that you used?

Exercise 5.9. The following incomplete block of Python code will help to streamline the previous exercise so that you do not need to do the computation with a spreadsheet.

- a. Comment every existing line with a thorough description.
- b. Fill in the blanks in the code to perform the spreadsheet computations from the previous exercise.

- c. Run the code for several different sequences of values for h . Do you still observe the same result that you observed in part (e) of the previous exercise?
- d. Run the code for several different choices of the function f and several different choices for the point x . What do you observe?

```

import numpy as np
import matplotlib.pyplot as plt

f = lambda x: np.sin(x) * (1-x) # what does this line do?
exact = -np.sin(1) # what does this line do?

H = 2.0**(-np.arange(1,10)) # what does this line do?
AbsError = [] # start off with a blank list of errors

# Create columns with column headers
print(f"{'h':<12} {'Absolute Error':<22} {'Reduction factor':<20}")
print("-----")

# Fill the rows of the table
for h in H:
    approx = # FINISH THIS LINE OF CODE
    AbsError.append(abs((approx - exact)/exact))
    if h==H[0]:
        reduction_factor = ''
    else:
        reduction_factor = AbsError[-2]/AbsError[-1]
    print(f"{h:<12} {AbsError[-1]:<22} {reduction_factor:<20}")

# Make a plot
plt.loglog(H, AbsError, 'b-*') # Why are we making a loglog plot?
plt.grid()
plt.show()

```

Exercise 5.10. Explain the phrase: *The forward difference approximation $f'(x) \approx \frac{f(x+h)-f(x)}{h}$ is first order.*

5.1.3 Efficient Coding

Now that we have a handle on how the forward-difference approximation scheme for the first derivative works and how the error depends on the step size, let us build some code that will take in a function and output the approximate first derivative on an entire interval instead of just at a single point.

Exercise 5.11. We want to build a Python function that accepts:

- a mathematical function,
- the bounds of an interval,
- and the number of subintervals.

The function will return the forward-difference approximation of the first derivative at every point in the interval except at the right-hand side. For example, we could send the function $f(x) = \sin(x)$, the interval $[0, 2\pi]$, and tell it to split the interval into 100 subintervals. We would then get back an approximate value of the derivative $f'(x)$ at all of the points except at $x = 2\pi$.

1. First of all, why can we not compute the forward-difference approximation of the derivative at the last point?
2. Next, fill in the blanks in the partially complete code below. Every line needs to have a comment explaining exactly what it does.

```
import numpy as np
import matplotlib.pyplot as plt
def ForwardDiff(f,a,b,N):
    x = np.linspace(a,b,N+1) # What does this line of code do?
    # What's up with the N+1 in the previous line?
    h = x[1] - x[0] # What does this line of code do?
    df = [] # What does this line of code do?
    for j in np.arange(len(x)-1): # What does this line of code do?
        # What's up with the -1 in the definition of the loop?
        #
        # Now we want to build the approximation
        # (f(x+h) - f(x)) / h.
        # Obviously "x+h" is just the next item in the list of
        # x values so when we do f(x+h) mathematically we should
        # write f(x[j+1]) in Python (explain this).
```

```

# Fill in the question marks below
df.append((f(???)) - f(??)) / h )
return df

```

3. Now we want to call upon this function to build the first order approximation of the first derivative for some function. We will use the function $f(x) = \sin(x)$ on the interval $[0, 2\pi]$ with 100 sub intervals (since we know what the answer should be). Complete the code below to call upon your `ForwardDiff()` function and to plot $f(x)$, $f'(x)$, and the approximation of $f'(x)$.

```

f = lambda x: np.sin(x)
exact_df = lambda x: np.cos(x)
a = ???
b = ???
N = 100 # What is this?
x = np.linspace(a,b,N+1)
# What does the previous line do?
# What's up with the N+1?

df = ForwardDiff(f,a,b,N) # What does this line do?

# In the next line we plot three curves:
# 1) the function f(x)
# 2) the function f'(x)
# 3) the approximation of f'(x)
# However, we do something funny with the x in the last plot. Why?
plt.plot(x,f(x),'b',x,exact_df(x),'r--',x[0:-1], df, 'k-.')
plt.grid()
plt.legend(['f(x) = sin(x)',
            'exact first deriv',
            'approx first deriv'])
plt.show()

```

4. Implement your completed code and then test it in several ways:
- Test your code on functions where you know the derivative. Be sure that you get the plots that you expect.
 - Test your code with a very large number of sub intervals, N . What do you observe?
 - Test your code with small number of sub intervals, N . What do you observe?
-

Exercise 5.12. Now let us build the first derivative function in a much *smarter* way – using NumPy arrays in Python. Instead of looping over all of the x values, we can take advantage of the fact that NumPy operations can act on all the elements of an array at once and hence we can do all of the function evaluations and all the subtractions and divisions at once without a loop.

1. The line of code `x = np.linspace(a,b,N+1)` builds a numpy vector of $N + 1$ values of x starting at a and ending at b . Then `y = f(x)` builds a vector with the function values at all the elements in `x`. In the following questions remember that Python indexes all lists starting at 0. Also remember that you can call on the last element of a list using an index of `-1`. Finally, remember that if you do `x[p:q]` in Python you will get a list of `x` values starting at index `p` and ending at index `q-1`.
 1. What will we get if we evaluate the code `y[1:]`?
 2. What will we get if we evaluate the code `y[:-1]`?
 3. What will we get if we evaluate the code `y[1:] - y[:-1]`?
 4. What will we get if we evaluate the code `(y[1:] - y[:-1]) / h`?
2. Use the insight from part (1) to simplify your first order first derivative function to look like the code below.

```
def ForwardDiff(f,a,b,N):
    x = np.linspace(a,b,N+1)
    h = x[1] - x[0]
    y = f(x)
    df = # your line of code goes here?
    return df
```

Exercise 5.13. Write code that finds a first order approximation for the first derivative of $f(x) = \sin(x) - x \sin(x)$ on the interval $x \in (0, 15)$. Your script should output two plots (side-by-side).

- a. The left-hand plot should show the function in blue and the approximate first derivative as a red dashed curve. Sample code for this exercise is given below.

```
import matplotlib.pyplot as plt
import numpy as np

f = lambda x: np.sin(x) - x*np.sin(x)
a = 0
b = 15
N = # make this an appropriately sized number of subintervals
```

```

x = np.linspace(a,b,N+1) # what does this line do?
y = f(x) # what does this line do?
df = ForwardDiff(f,a,b,N) # what does this line do?

fig, ax = plt.subplots(1,2) # what does this line do?
ax[0].plot(x,y,'b',x[0:-1],df,'r--') # what does this line do?
ax[0].grid()

```

- b. The right-hand plot should show the absolute error between the exact derivative and the numerical derivative. You should use a logarithmic y axis for this plot.

```

exact = lambda x: # write a function for the exact derivative
# There is a lot going on the next line of code ... explain it.
ax[1].semilogy(x[0:-1],abs(exact(x[0:-1]) - df))
ax[1].grid()

```

- c. Play with the number of sub intervals, N , and demonstrate the fact that we are using a first order method to approximate the first derivative.
-

5.1.4 A Better First Derivative

Next we will build a more accurate numerical first derivative scheme. The derivation technique is the same: play a little algebra game with the Taylor series and see if you can get the first derivative to simplify out. This time we will be hoping to get a second order method.

Exercise 5.14. Consider again the Taylor series for an infinitely differentiable function $f(x)$:

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0)^1 + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f^{(3)}(x_0)}{3!}(x - x_0)^3 + \dots \quad (5.14)$$

1. Replace the “ x ” in the Taylor Series with “ $x + h$ ” and replace the “ x_0 ” in the Taylor Series with “ x ” and simplify.

$$f(x + h) = \underline{\hspace{10cm}} \quad (5.15)$$

2. Now replace the “ x ” in the Taylor Series with “ $x - h$ ” and replace the “ x_0 ” in the Taylor Series with “ x ” and simplify.

$$f(x - h) = \underline{\hspace{10cm}} \quad (5.16)$$

3. Find the difference between $f(x + h)$ and $f(x - h)$ and simplify. Be very careful of your signs.

$$f(x + h) - f(x - h) = \underline{\hspace{10cm}} \quad (5.17)$$

4. Solve for $f'(x)$ in your result from part (3). Fill in the question marks and blanks below once you have finished simplifying.

$$f'(x) = \frac{\underline{\hspace{2cm}} - \underline{\hspace{2cm}}}{2h} + \underline{\hspace{10cm}} \quad (5.18)$$

5. Use your result from part (4) to verify that

$$f'(x) = \underline{\hspace{10cm}} + \mathcal{O}(h^2). \quad (5.19)$$

6. Draw a picture similar to Figure 5.2 showing what this scheme is doing graphically.
-

Exercise 5.15. Let us return to the function $f(x) = \sin(x)(1 - x)$ but this time we will approximate the first derivative at $x = 1$ using the formula

$$f'(1) \approx \frac{f(1 + h) - f(1 - h)}{2h} =: \delta f(1). \quad (5.20)$$

You should already have the first derivative and the exact answer from Exercise 5.8 (if not, then go get them by hand again).

- a. Fill in the table below with the derivative approximation and the absolute error associated with each given h . You may want to use a spreadsheet to organize your data (be sure that you are working in radians!).

h	$\delta f(1)$	$ f'(1) - \delta f(1) $
$2^{-1} = 0.5$	-0.73846	0.10301
$2^{-2} = 0.25$	-0.81531	0.02616
$2^{-3} = 0.125$		
$2^{-4} = 0.0625$		
2^{-5}		
2^{-6}		
2^{-7}		
2^{-8}		
2^{-9}		

h	$\delta f(1)$	$ f'(1) - \delta f(1) $
2^{-10}		

- b. There was nothing really special in part (a) about powers of 2. Use your spreadsheet to build similar tables for the following sequences of h :

$$\begin{aligned} h &= 3^{-1}, 3^{-2}, 3^{-3}, \dots \\ h &= 5^{-1}, 5^{-2}, 5^{-3}, \dots \\ h &= 10^{-1}, 10^{-2}, 10^{-3}, \dots \\ h &= \pi^{-1}, \pi^{-2}, \pi^{-3}, \dots \end{aligned} \tag{5.21}$$

- c. Observation: If you calculate a numerical derivative with a central difference and calculate the resulting absolute error with a fixed value of h , then what do you expect to happen to the absolute error if you divide the value of h by some positive constant M ? It may be helpful to include a column in your table that tracks the error reduction factor as we decrease h .
- d. What does your answer to part (c) have to do with the approximation order of the numerical derivative method that you used?
-

Exercise 5.16. Write a Python function `CentralDiff(f, a, b, N)` that takes a mathematical function f , the start and end values of an interval $[a, b]$ and the number N of subintervals to use. It should return a second order numerical approximation to the first derivative on the interval. This should be a vector with $N - 1$ entries (why?). You should try to write this code without using any loops. (Hint: This should really be a minor modification of your first order first derivative code.) Test the code on functions where you know the first derivative.

Exercise 5.17. The plot shown in Figure 5.3 shows the maximum absolute error between the exact first derivative of a function $f(x)$ and a numerical first derivative approximation scheme. At this point we know two schemes:

$$f'(x) = \frac{f(x + h) - f(x)}{h} + \mathcal{O}(h) \tag{5.22}$$

and

$$f'(x) = \frac{f(x + h) - f(x - h)}{2h} + \mathcal{O}(h^2). \tag{5.23}$$

1. Which curve in the plot matches with which method. How do you know?
2. Recreate the plot with a function of your choosing.

```

import numpy as np
import matplotlib.pyplot as plt

# Choose a function f
f = lambda x: np.sin(x)
# Give its derivative
df = lambda x: np.cos(x)
# Choose interval
a = 0
b = 2*np.pi

m = 16 # Number of different step sizes to plot
# Pre-allocate vectors for errors
fd_error = np.zeros(m)
cd_error = np.zeros(m)
# Pre-allocate vector for step sizes
H = np.zeros(m)

# Loop over the different step sizes
for n in range(m):
    N = 2**(n+2) # Number of subintervals
    x = np.linspace(a, b, N+1)
    y = f(x)
    h = x[1]-x[0] # step size

    # Calculate the derivative and approximations
    exact = df(x)
    forward_diff = (y[1:]-y[:-1])/h
    central_diff = (y[2:]-y[:-2])/(2*h)

    # save the maximum of the errors for this step size
    fd_error[n] = max(abs(forward_diff - df(x[:-1])))
    cd_error[n] = max(abs(central_diff - df(x[1:-1])))
    H[n] = h

# Make a loglog plot of the errors against step size
plt.loglog(H,fd_error,'b-', label='Approximation Method A')
plt.loglog(H,cd_error,'r-', label='Approximation Method B')
plt.xlabel('Steps size h')

```

```

plt.ylabel('Maximum Absolute Error')
plt.title('Comparing Two First Derivative Approximations')
plt.grid()
plt.legend()
plt.show()

```

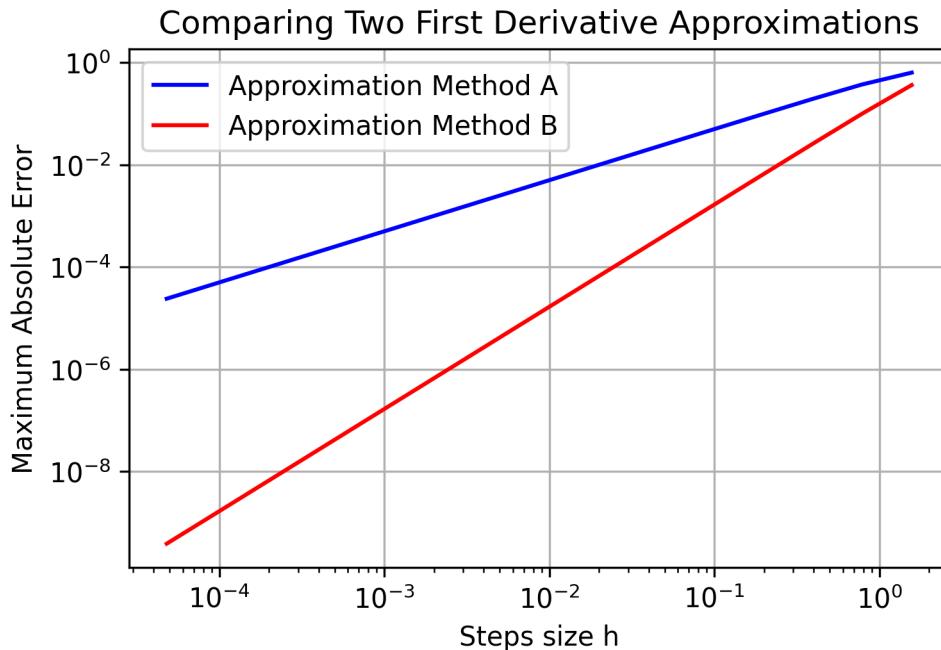


Figure 5.3: Maximum absolute error between the first derivative and two different approximations of the first derivative.

5.1.5 The Second Derivative

Now we will search for an approximation of the second derivative. Again, the game will be the same: experiment with the Taylor series and some algebra with an eye toward getting the second derivative to pop out cleanly. This time we will do the algebra in such a way that the first derivative cancels.

From the previous exercises you already have Taylor expansions of the form $f(x+h)$ and $f(x-$

h). Let us summarize them here since you are going to need them for future computations.

$$\begin{aligned} f(x+h) &= f(x) + \frac{f'(x)}{1!}h + \frac{f''(x)}{2!}h^2 + \frac{f^{(3)}(x)}{3!}h^3 + \dots \\ f(x-h) &= f(x) - \frac{f'(x)}{1!}h + \frac{f''(x)}{2!}h^2 - \frac{f^{(3)}(x)}{3!}h^3 + \dots. \end{aligned} \quad (5.24)$$

Exercise 5.18. The goal of this exercise is to use the Taylor series for $f(x+h)$ and $f(x-h)$ to arrive at an approximation scheme for the second derivative $f''(x)$.

1. Add the Taylor series for $f(x+h)$ and $f(x-h)$ and combine all like terms. You should notice that several terms cancel.

$$f(x+h) + f(x-h) = \underline{\hspace{10cm}}. \quad (5.25)$$

2. Solve your answer in part (1) for $f''(x)$.

$$f''(x) = \frac{\underline{\hspace{2cm}} - 2 \cdot \underline{\hspace{2cm}} + \underline{\hspace{2cm}}}{h^2} + \underline{\hspace{10cm}}. \quad (5.26)$$

3. If we were to drop all of the terms after the fraction on the right-hand side of the previous result we would be introducing some error into the derivative computation. What does this tell us about the order of the error for the second derivative approximation scheme we just built?
-

Exercise 5.19. Again consider the function $f(x) = \sin(x)(1-x)$.

1. Calculate the second derivative of this function analytically and evaluate it at $x = 1$.
 2. If we calculate the second derivative with the central difference scheme that you built in the previous exercise using $h = 0.5$ then we get an absolute error of about 0.044466. Stop now and verify this error calculation.
 3. Based on our previous work with the order of the error in a numerical differentiation scheme, what do you predict the error will be if we calculate $f''(1)$ with $h = 0.25$? With $h = 0.05$? With $h = 0.005$? Defend your answers.
-

Exercise 5.20. Write a Python function `SecondDiff(f, a, b, N)` that takes a mathematical function f , the start and end values of an interval $[a, b]$ and the number N of subintervals to use. It should return a second order numerical approximation to the second derivative on the interval. This should be a vector with $N - 1$ entries (why?). As before, you should write your code without using any loops.

Exercise 5.21. Test your second derivative code on the function $f(x) = \sin(x) - x \sin(x)$ by doing the following.

1. Find the analytic second derivative by hand (you did this already in Exercise 5.19).
 2. Find the numerical second derivative with the code that you just wrote.
 3. Find the absolute difference between your numerical second derivative and the actual second derivative. This is point-by-point subtraction so you should end up with a vector of errors.
 4. Find the maximum of your errors.
 5. Now we want to see how the code works if you change the number of points used. Build a plot showing the value of h on the horizontal axis and the maximum error on the vertical axis. You will need to write a loop that gets the error for many different values of h . Finally, it is probably best to build this plot on a log-log scale.
 6. Discuss what you see? How do you see the fact that the numerical second derivative is second order accurate?
-

The table below summarizes the formulas that we have for derivatives thus far. The exercises at the end of this chapter contain several more derivative approximations. We will return to this idea when we study numerical differential equations in Chapter 8.

Derivative	Formula	Error	Name
1^{st}	$f'(x) \approx \frac{f(x+h)-f(x)}{h}$	$\mathcal{O}(h)$	Forward Difference
1^{st}	$f'(x) \approx \frac{f(x)-f(x-h)}{h}$	$\mathcal{O}(h)$	Backward Difference
1^{st}	$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$	$\mathcal{O}(h^2)$	Central Difference

Derivative	Formula	Error	Name
2 nd	$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$	$\mathcal{O}(h^2)$	Central Difference

Exercise 5.22. Let $f(x)$ be a twice differentiable function. We are interested in the first and second derivative of the function f at the point $x = 1.74$. Use what you have learned in this section to answer the following questions. (For clarity, you can think of “ f ” as a different function in each of the following questions ...it does not really matter exactly what function f is.)

1. Johnny used a numerical first derivative scheme with $h = 0.1$ to approximate $f'(1.74)$ and found an absolute error of 3.28. He then used $h = 0.01$ and found an absolute error of 0.328. What was the order of the error in his first derivative scheme? How can you tell?
2. Betty used a numerical first derivative scheme with $h = 0.2$ to approximate $f'(1.74)$ and found an absolute error of 4.32. She then used $h = 0.1$ and found an absolute error of 1.08. What numerical first derivative scheme did she likely use?
3. Harry wants to compute $f''(1.74)$ to within 1% using a central difference scheme. He tries $h = 0.25$ and gets an absolute percentage error of 3.71%. What h should he try next so that his absolute percentage error is close to 1%?

Exercise 5.23. We said at the start of this section that the spacing of the steps do not have to be constant. Instead of a constant step size h we could have variable step sizes $\Delta x_i := x_{i+1} - x_i$. However, this complicates the expression for the second derivative. In this exercise you can check that you fully understood the derivation of the formula for the second derivative by repeating it for these variable step sizes. So you will need to start with the Taylor expansions of $f(x_{i+1}) = f(x_i + \Delta x_i)$ and $f(x_{i-1}) = f(x_i - \Delta x_{i-1})$ around x_i .

5.2 Automatic Differentiation

In the previous section, we explored numerical differentiation through finite difference methods. These methods approximate derivatives by evaluating the function at different points and calculating differences. While straightforward to implement, they suffer from two main limitations:

1. **Truncation error:** As we've seen, the error scales with some power of the step size h
2. **Round-off error:** As h gets extremely small, floating-point arithmetic leads to precision loss

Automatic differentiation (AD) is a different approach that computes derivatives exactly (to machine precision) without relying on finite differences. AD leverages the chain rule and the fact that all computer programs, no matter how complex, ultimately break down into elementary operations (addition, multiplication, sin, exp, etc.) whose derivatives are known.

Let's introduce the concept of a **computation graph**, which is fundamental to understanding automatic differentiation.

A computation graph represents a mathematical function as a directed graph where:

- **Nodes** represent variables (inputs, outputs, or intermediate values)
 - **Edges** represent dependencies between variables
 - Each node performs a simple operation with known derivatives
-

Example 5.1. Consider the function $f(x, y) = x^2y + \sin(xy)$. We can break this down into elementary operations and visualize it as a computation graph:

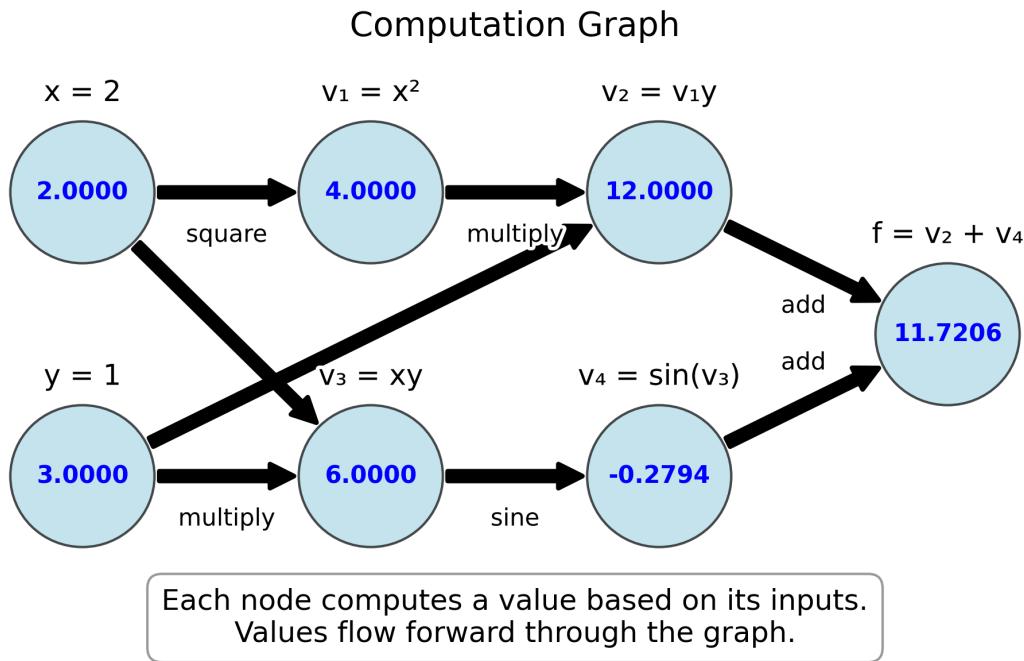


Figure 5.4: Computation graph for $f(x,y) = x^2y + \sin(xy)$ at $(x,y) = (2,1)$

This computation graph for $f(x, y) = x^2y + \sin(xy)$ at $(x, y) = (2, 3)$ shows:

1. Input nodes: $x = 2, y = 3$
2. Intermediate computations:
 - $v_1 = x^2 = 4$
 - $v_3 = xy = 6$
 - $v_2 = v_1y = 12$
 - $v_4 = \sin(v_3) = \sin(6) \approx -0.2794$
3. Output node: $f = v_2 + v_4 \approx 12 + (-0.2794) \approx 11.7206$

For each operation in the computation graph, we know:

1. How to compute the function value (forward evaluation)
2. How to compute the derivative of the operation with respect to its inputs

The computation graph is the foundation for automatic differentiation:

- **Forward mode AD:** Derivatives flow through the graph in the same direction as function evaluation
- **Reverse mode AD:** Function values flow forward through the graph, derivatives flow backward

This will become clearer in sections Section 5.2.1 and Section 5.2.2.

Exercise 5.24.

1. Draw the computation graph for the function $f(x) = x^2 \sin(x)$.
 2. For the function $h(x, y, z) = xy + yz + zx$, identify:
 - The input nodes
 - The intermediate nodes and their operations
 - The output node
 3. Explain why breaking a complex function into a computation graph of elementary operations is useful for derivative computation.
-

5.2.1 Forward mode AD

In forward mode automatic differentiation, we track both the values of variables and their derivatives with respect to the input variables. This allows us to build the derivatives as we compute the function value.

Example 5.2. Consider a simple function $f(x) = x^2 \sin(x)$. We can compute both the value and the derivative at $x = 2$ as follows:

1. Initialize: $x = 2$, $\frac{dx}{dx} = 1$ (The derivative of x with respect to itself is 1)
2. Compute $u = x^2$:
 - Value: $u = 2^2 = 4$
 - Derivative: $\frac{du}{dx} = \frac{d(x^2)}{dx} \cdot \frac{dx}{dx} = 2x \cdot 1 = 2 \cdot 2 = 4$
3. Compute $v = \sin(x)$:
 - Value: $v = \sin(2) \approx 0.9093$
 - Derivative: $\frac{dv}{dx} = \frac{d\sin(x)}{dx} \cdot \frac{dx}{dx} = \cos(x) \cdot 1 = \cos(2) \approx -0.4161$
4. Compute $f = u \cdot v$:
 - Value: $f = 4 \cdot 0.9093 \approx 3.6372$
 - Derivative: $\frac{df}{dx} = \frac{d(u \cdot v)}{dx} = \frac{du}{dx} \cdot v + u \cdot \frac{dv}{dx} = 4 \cdot 0.9093 + 4 \cdot (-0.4161) \approx 1.9728$

This is **forward mode** automatic differentiation.

The figure below illustrates the computational graph for $f(x) = x^2 \sin(x)$ with forward mode AD. The blue values show the function evaluation, while the red values show the derivative calculation:

Note how both the value and the derivative are computed in a single pass through the computation graph, with the derivatives at later nodes being computed with the help of the derivatives at earlier nodes.

Let's formalize the forward mode automatic differentiation process by creating a systematic approach. We'll use the concept of a “**dual number**” that carries both a value and its derivative. We will represent each variable as a pair $\begin{pmatrix} v \\ d \end{pmatrix}$ where v is the variable’s value and d is its derivative with respect to the input we’re differentiating against.

Because we know the differentiation rules for the following basic operations we can define the operations on dual numbers as follows:

Forward Mode Automatic Differentiation

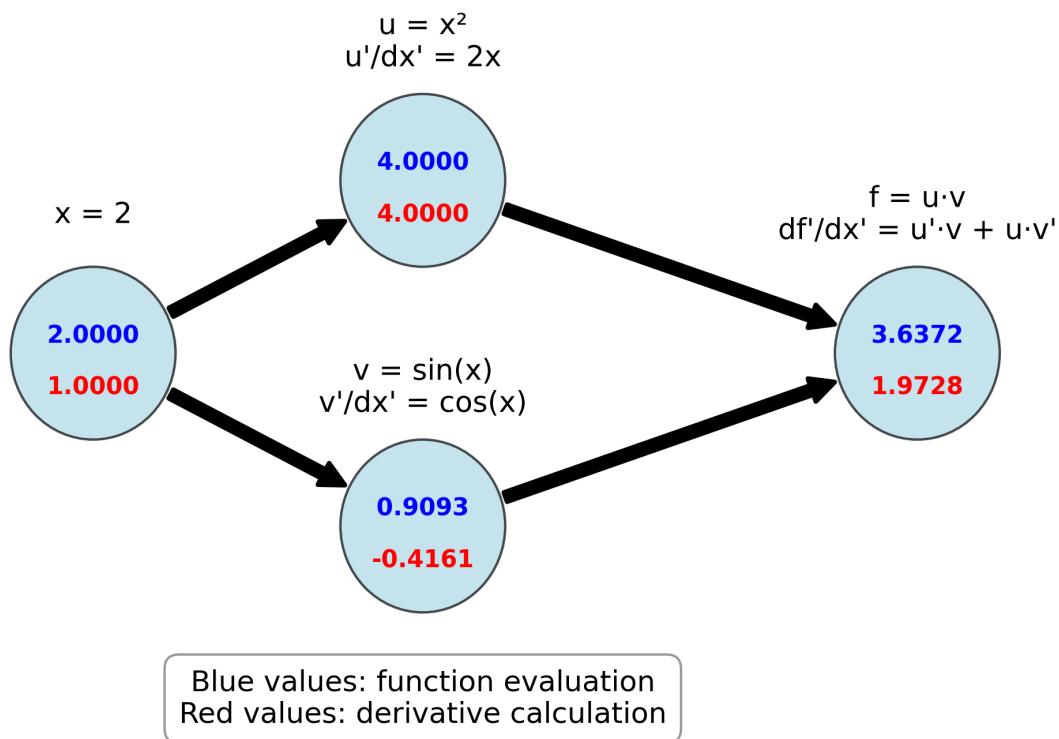


Figure 5.5: Forward mode automatic differentiation for $f(x) = x^2 \cdot \sin(x)$ at $x = 2$

1. Addition: $\begin{pmatrix} a \\ a' \end{pmatrix} + \begin{pmatrix} b \\ b' \end{pmatrix} = \begin{pmatrix} a+b \\ a'+b' \end{pmatrix}$
2. Multiplication: $\begin{pmatrix} a \\ a' \end{pmatrix} \cdot \begin{pmatrix} b \\ b' \end{pmatrix} = \begin{pmatrix} a \cdot b \\ a' \cdot b + a \cdot b' \end{pmatrix}$
3. Division: $\begin{pmatrix} a \\ a' \end{pmatrix} / \begin{pmatrix} b \\ b' \end{pmatrix} = \begin{pmatrix} a/b \\ (a' \cdot b - a \cdot b')/b^2 \end{pmatrix}$
4. Power: $\begin{pmatrix} a \\ a' \end{pmatrix}^n = \begin{pmatrix} a^n \\ n \cdot a^{n-1} \cdot a' \end{pmatrix}$
5. Sine: $\sin \begin{pmatrix} a \\ a' \end{pmatrix} = \begin{pmatrix} \sin(a) \\ \cos(a) \cdot a' \end{pmatrix}$
6. Exponential: $e^{\begin{pmatrix} a \\ a' \end{pmatrix}} = \begin{pmatrix} e^a \\ e^a \cdot a' \end{pmatrix}$

With this notation the calculations from Example 5.2 can be written as:

$$\begin{aligned}\begin{pmatrix} x \\ x' \end{pmatrix} &= \begin{pmatrix} 2 \\ 1 \end{pmatrix} \\ \begin{pmatrix} u \\ u' \end{pmatrix} &= \begin{pmatrix} x \\ x' \end{pmatrix}^2 = \begin{pmatrix} x^2 \\ 2xx' \end{pmatrix} = \begin{pmatrix} 4 \\ 4 \end{pmatrix} \\ \begin{pmatrix} v \\ v' \end{pmatrix} &= \sin \begin{pmatrix} x \\ x' \end{pmatrix} = \begin{pmatrix} \sin(x) \\ \cos(x)x' \end{pmatrix} = \begin{pmatrix} 0.9093 \\ -0.4161 \end{pmatrix} \\ \begin{pmatrix} f \\ f' \end{pmatrix} &= \begin{pmatrix} u \\ u' \end{pmatrix} \cdot \begin{pmatrix} v \\ v' \end{pmatrix} = \begin{pmatrix} u \cdot v \\ u' \cdot v + u \cdot v' \end{pmatrix} = \begin{pmatrix} 3.6372 \\ 1.9728 \end{pmatrix}\end{aligned}$$

Exercise 5.25. Compute the derivative of $f(x) = \frac{x^2+1}{2x-3}$ at $x = 2$ using the dual number approach.

Verify your result by computing the derivative analytically and comparing.

Next let us teach Python how to do this. Here is the implementation for four basic operations:

```

import numpy as np
# Basic operations on dual numbers represented as tuples (value, derivative)
def dual_add(a, b):
    """Add two dual numbers:
    (a, a') + (b, b') = (a + b, a' + b')"""
    return (a[0] + b[0], a[1] + b[1])

def dual_multiply(a, b):
    """Multiply two dual numbers:
    (a, a') * (b, b') = (a*b, a'*b + a*b')"""
    return (a[0] * b[0], a[1] * b[0] + a[0] * b[1])

def dual_power(a, n):
    """Raise dual number to integer power n:
    (a, a')^n = (a^n, n*a^(n-1)*a')"""
    return (a[0]**n, n * a[0]**(n-1) * a[1])

def dual_sin(a):
    """Sine of dual number:
    sin(a, a') = (sin(a), cos(a)*a')"""
    return (np.sin(a[0]), np.cos(a[0]) * a[1])

```

We can now use these operations to compute the derivative of $f(x) = x^2 \sin(x)$ at $x = 2$ as follows:

```

x = (2, 1)
u = dual_power(x, 2)
v = dual_sin(x)
f = dual_multiply(u, v)
print(f)

```

(`np.float64(3.637189707302727)`, `np.float64(1.9726023611141572)`)

Exercise 5.26. Compute the derivative of $f(x) = \frac{x^2+1}{2x-3}$ at $x = 2$ using your Python implementation of the dual number approach. You will need to define the operation for division using the quotient rule.

Verify that you get the same result as in Exercise 6.3.

Exercise 5.27. Compute the derivative of $f(x) = \exp(x)/\cos(x)$ at $x = 2$ using your Python implementation of the dual number approach. You will need to define dual number versions of the exponential and cosine functions.

For multivariate functions, we can compute one directional derivative at a time.

For a function $f(x, y)$, to compute $\frac{\partial f}{\partial x}$, we initialize:

- $x = \begin{pmatrix} x \\ 1 \end{pmatrix}$ (value and derivative of x with respect to x)
- $y = \begin{pmatrix} y \\ 0 \end{pmatrix}$ (value and derivative of y with respect to x)

And to compute $\frac{\partial f}{\partial y}$, we initialize:

- $x = \begin{pmatrix} x \\ 0 \end{pmatrix}$ (value and derivative of x with respect to y)
 - $y = \begin{pmatrix} y \\ 1 \end{pmatrix}$ (value and derivative of y with respect to y)
-

Exercise 5.28.

1. For $f(x, y) = x^2y + \sin(xy)$, compute both $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ at $(x, y) = (2, 1)$ using forward mode AD with the dual number approach.
 2. For $f(x, y, z) = x^2y + y \sin(z) + z \cos(x)$, compute $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$, and $\frac{\partial f}{\partial z}$ at $(x, y, z) = (1, 1, 1)$.
 3. If a function has n inputs and we want all partial derivatives, how many forward mode passes do we need? What implications does this have for functions with many inputs?
-

5.2.2 Reverse mode AD

We have seen that if one has a function of n variables, forward mode AD requires n passes to compute all partial derivatives. Now let's explore **reverse mode** automatic differentiation, which is more efficient for functions with many inputs and few outputs. Reverse mode AD first computes the function value and then propagates derivatives backward from the output to the inputs.

This approach is particularly important in optimisation and machine learning. In neural networks, which can have millions of parameters (inputs) but typically only one output (the loss function), reverse mode AD allows us to compute all partial derivatives in a single backward pass. This is the foundation of the famous backpropagation algorithm used to train neural networks efficiently. Without reverse mode AD, training modern deep learning models would be computationally infeasible, as forward mode would require millions of separate passes to compute all the necessary gradients for parameter updates.

We introduce the concept of **adjoints** or **accumulated gradients**. The adjoint of a variable v is denoted \bar{v} and represents $\frac{\partial f}{\partial v}$ where f is the final output.

Example 5.3. Let's trace through the reverse mode process for $f(x, y) = x^2y + \sin(xy)$ at $(x, y) = (2, 3)$:

1. Define intermediate variables in the computation graph as in Figure 5.4:

- $v_1 = x^2 = 4$
- $v_2 = v_1 \cdot y = 4 \cdot 3 = 12$
- $v_3 = x \cdot y = 2 \cdot 3 = 6$
- $v_4 = \sin(v_3) = \sin(6) \approx -0.2794$
- $f = v_2 + v_4 = 12 + (-0.2794) \approx 11.7206$ (final output)

2. Initialize the adjoint of the output: $\bar{f} = 1$
3. Propagate adjoints backward using the chain rule:

- $\bar{v}_4 = \bar{f} \cdot \frac{\partial f}{\partial v_4} = 1 \cdot 1 = 1$
- $\bar{v}_3 = \bar{v}_4 \cdot \frac{\partial v_4}{\partial v_3} = 1 \cdot \cos(v_3) = \cos(6) \approx 0.9602$
- $\bar{v}_2 = \bar{v}_5 \cdot \frac{\partial f}{\partial v_2} = 1 \cdot 1 = 1$
- $\bar{v}_1 = \bar{v}_2 \cdot \frac{\partial v_2}{\partial v_1} = 1 \cdot y = 3$
- $\bar{x} = \bar{v}_1 \cdot \frac{\partial v_1}{\partial x} + \bar{v}_3 \cdot \frac{\partial v_3}{\partial y} = 3 \cdot 2x + 0.9602 \cdot y = 3 \cdot 2 \cdot 2 + 0.9602 \cdot 3 \approx 14.8806$
- $\bar{y} = \bar{v}_2 \cdot \frac{\partial v_2}{\partial y} + \bar{v}_3 \cdot \frac{\partial v_3}{\partial y} = 1 \cdot v_1 + 0.9602 \cdot x = 1 \cdot 4 + 0.9602 \cdot 2 \approx 5.9204$

4. The final results are $\frac{\partial f}{\partial x} = \bar{x} \approx 14.8806$ and $\frac{\partial f}{\partial y} = \bar{y} \approx 5.9204$

Reverse Mode Automatic Differentiation

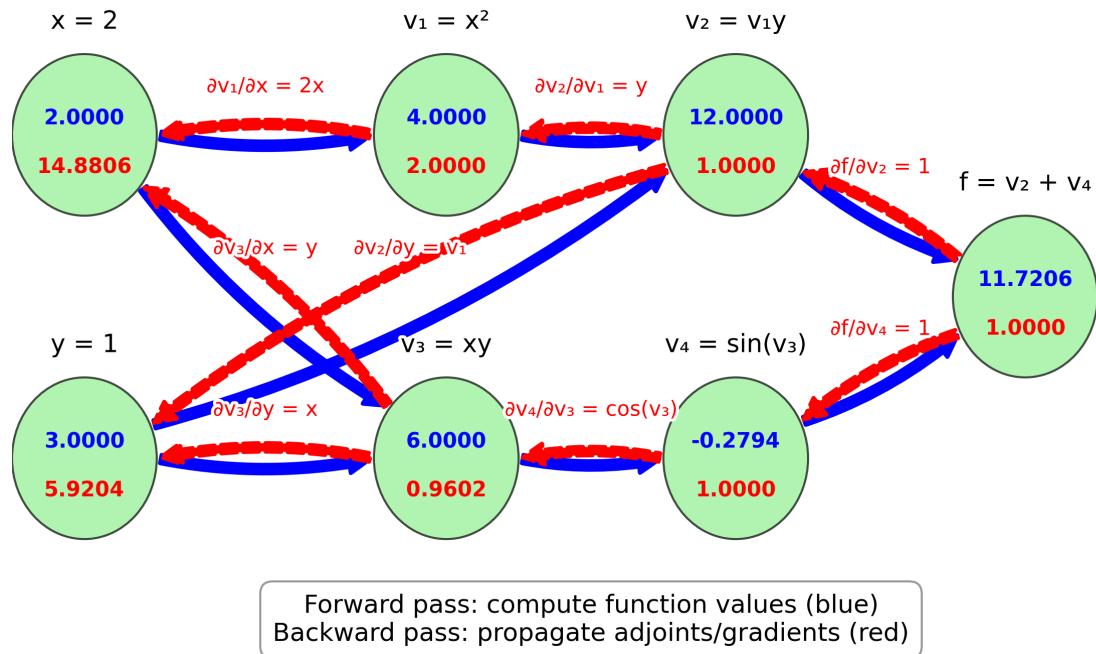


Figure 5.6: Reverse mode automatic differentiation for $f(x,y) = x^2y + \sin(xy)$ at $(x,y) = (2,1)$

Figure 5.6 illustrates the reverse mode AD computation graph for this example.

Note how the backward pass computes the gradients with respect to x and y in a single pass through the computation graph. This pass is going backwards through the graph because the gradients at earlier nodes are computed with the help of the gradients at later nodes. Note how the gradient at a node is the sum of the contributions from the gradients of the nodes that depend on it. Also note how we need the values of the nodes to compute the gradients, so we need to keep track of them in the forward pass. Thus reverse mode is more memory intensive than forward mode.

Exercise 5.29. Perform the forward pass and the backward pass through the computation graph for $f(x, y) = xy^2 \cos xy^2$ at $(x, y) = (\pi, 3)$.

Implementing reverse mode automatic differentiation in Python is not straightforward, because we need to build up the computation graph and store the values of the nodes so that we can use them in the backward pass. So we will not attempt an implementation from scratch but instead use the JAX library that provides automatic differentiation in Python. JAX is designed to be simple to use while providing powerful capabilities.

Example 5.4. Here's a simple example using JAX to compute derivatives:

```
import jax
import jax.numpy as jnp

# Define a function
def f(x):
    return jnp.sin(x) * (1 - x)

# Compute the derivative function
df = jax.grad(f)

# Evaluate at x = 1
print(f"f(1) = {f(1)}")
print(f"f'(1) = {df(1.0)}")

f(1) = 0.0
f'(1) = -0.8414709568023682
```

Two things to note:

- We had to use the jax versions of NumPy functions, like `jnp.sin` instead of `np.sin`.
 - We had to pass a float to the `df` function, rather than an integer. So we had to write `df(1.0)` instead of `df(1)`.
-

Exercise 5.30. Use JAX to compute the derivatives of:

- $f(x) = x^3 - 2x^2 + 4x - 7$ at $x = 1$;
 - $f(x) = (x^2 + 1)/(2x - 3)$ at $x = 2$. Check that you get the same results as in Exercise 6.3.
-

Example 5.5. For multivariate functions, JAX allows us to compute partial derivatives:

```
def g(x, y):
    return x**2 * y + jnp.sin(x * y)

# Compute partial with respect to first argument (x)
dg_dx = jax.grad(g, argnums=0)

# Compute partial with respect to second argument (y)
dg_dy = jax.grad(g, argnums=1)

# Evaluate at (2, 1)
print(f" g/ x at (2,1) = {dg_dx(2.0, 1.0)}")
print(f" g/ y at (2,1) = {dg_dy(2.0, 1.0)}")
```

```
g/ x at (2,1) = 3.583853244781494
g/ y at (2,1) = 3.167706251144409
```

Exercise 5.31. Use JAX to compute the partial derivatives of $f(x, y) = x \exp(y + x) + y \sin(xy)$ at $(x, y) = (1, 2)$.

JAX can also compute higher-order derivatives:

Example 5.6.

```
f = lambda x: x*x*x

# Second derivative
d2f = jax.grad(jax.grad(f))
print(f"f''(1) = {d2f(1.0)}")

# Or more concisely
d2f = jax.hessian(f)
print(f"f''(1) = {d2f(1.0)}")
```

```
f''(1) = 6.0
f''(1) = 6.0
```

Exercise 5.32. Compare the accuracy and efficiency of the differentiation methods we've studied:

1. For the function $f(x) = \sin(x)(1 - x)$, compute the derivative at $x = 1$ using:
 - a. Forward difference with step sizes $h = 0.1, 0.01, 0.001$
 - b. Central difference with step sizes $h = 0.1, 0.01, 0.001$
 - c. JAX automatic differentiation

Create a table giving the value of the derivative and the absolute error for each of these 7 cases.

5.3 Algorithm Summaries

Exercise 5.33. Starting from Taylor series prove that

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad (5.27)$$

is a first-order approximation of the first derivative of $f(x)$. Clearly describe what “first-order approximation” means in this context.

Exercise 5.34. Starting from Taylor series prove that

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \quad (5.28)$$

is a second-order approximation of the first derivative of $f(x)$. Clearly describe what “second-order approximation” means in this context.

Exercise 5.35. Starting from Taylor series prove that

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \quad (5.29)$$

is a second-order approximation of the second derivative of $f(x)$. Clearly describe what “second-order approximation” means in this context.

Exercise 5.36. Explain how to define arithmetic operations on dual numbers and how to use them to compute the derivative of a function with forward mode automatic differentiation.

Exercise 5.37. Given a computation graph of a function, explain how to accumulate the gradients in a backwards pass through the computation graph. What is the advantage of this method over forward mode automatic differentiation?

5.4 Problems

Exercise 5.38. For each of the following numerical differentiation formulas

- (1) prove that the formula is true and
- (2) find the order of the method.

To prove that each of the formulas is true you will need to write the Taylor series for all of the terms in the numerator on the right and then simplify to solve for the necessary derivative. The highest power of the remainder should reveal the order of the method.

$$\begin{aligned}1. \ f'(x) &\approx \frac{\frac{1}{12}f(x-2h) - \frac{2}{3}f(x-h) + \frac{2}{3}f(x+h) - \frac{1}{12}f(x+2h)}{h} \\2. \ f'(x) &\approx \frac{-\frac{3}{2}f(x) + 2f(x+h) - \frac{1}{2}f(x+2h)}{h} \\3. \ f''(x) &\approx \frac{-\frac{1}{12}f(x-2h) + \frac{4}{3}f(x-h) - \frac{5}{2}f(x) + \frac{4}{3}f(x+h) - \frac{1}{12}f(x+2h)}{h^2} \\4. \ f'''(x) &\approx \frac{-\frac{1}{2}f(x-2h) + f(x-h) - f(x+h) + \frac{1}{2}f(x+2h)}{h^3}\end{aligned}$$

Exercise 5.39. Write a function that accepts a list of (x, y) ordered pairs from a spreadsheet and returns a list of (x, y) ordered pairs for a first order approximation of the first derivative of the underlying function. Create a test spreadsheet file and a test script that have graphical output showing that your function is finding the correct derivative.

Exercise 5.40. Write a function that accepts a list of (x, y) ordered pairs from a spreadsheet or a `*.csv` file and returns a list of (x, y) ordered pairs for a second order approximation of the second derivative of the underlying function. Create a test spreadsheet file and a test script that have graphical output showing that your function is finding the correct derivative.

Exercise 5.41. Go to [data.gov](#) or the [World Health Organization Data Repository](#) and find a data set where the variables naturally lead to a meaningful derivative. Use appropriate code to evaluate and plot the derivative. If your data appears to be subject to significant noise then you may want to smooth the data first before doing the derivative. Write a few sentences explaining what the derivative means in the context of the data. Be very cautious of the units on the data sets and the units of your answer.

Exercise 5.42. A bicyclist completes a race course in 90 seconds. The speed of the biker at each 10-second interval is determined using a radar gun and is given in the table in feet per second. How long is the race course?

Time (sec)	0	10	20	30	40	50	60	70	80	90
Speed (ft/sec)	34	32	29	33	37	40	41	36	38	39

You can download the data with the following code.

```
import numpy as np
import pandas as pd
data = np.array(pd.read_csv('https://github.com/gustavdelius/NumericalAnalysis2025/raw/main/'))
```

6 Integrals

Now that we understand how to calculate derivatives, we begin our work on the second principal computation of calculus: evaluating a definite integral. You will be able to transfer much of what you did in the previous chapter, where you investigated the errors in numerical differentiation, to the investigation of the errors in numerical integration.

Exercise 6.1. Remember that a single-variable definite integral can be interpreted as the signed area between the curve and the x axis. Consider the shaded area of the region under the function plotted in Figure 6.1 between $x = 0$ and $x = 2$.

1. What rectangle with area 6 gives an upper bound for the area under the curve? Can you give a better upper bound?
 2. Why must the area under the curve be greater than 3?
 3. Is the area greater than 4? Why/Why not?
 4. Work with your group to give an estimate of the area and provide an estimate for the amount of error that you are making.
-

In this chapter we will study three different techniques for approximating the value of a definite integral.

6.1 Riemann Sums

In this subsection we will build our first method for approximating definite integrals. Recall from Calculus that the definition of the Riemann integral is

$$\int_a^b f(x)dx = \lim_{\Delta x \rightarrow 0} \sum_{j=1}^N f(x_j)\Delta x, \quad (6.1)$$

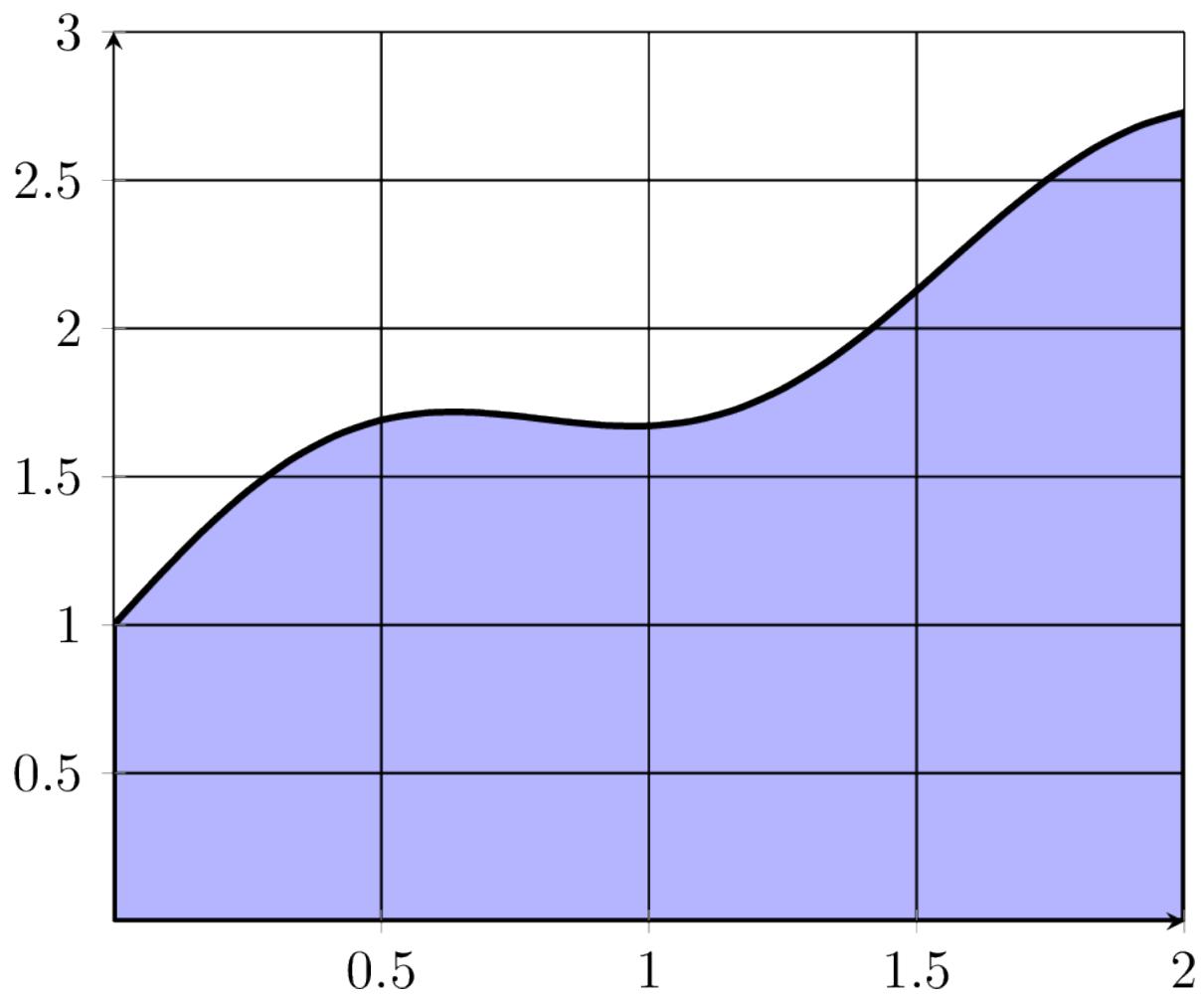


Figure 6.1: A sample integration

where N is the number of sub intervals on the interval $[a, b]$ and Δx is the width of the interval. As with differentiation, we can remove the limit and have a decent approximation of the integral so long as N is large (or equivalently, as long as Δx is small).

$$\int_a^b f(x)dx \approx \sum_{j=1}^N f(x_j)\Delta x. \quad (6.2)$$

You are likely familiar with this approximation of the integral from Calculus. The value of x_j can be chosen anywhere within the sub interval and three common choices are to use the left-aligned, the midpoint-aligned, and the right-aligned.

We see a depiction of this in Figure 6.2.

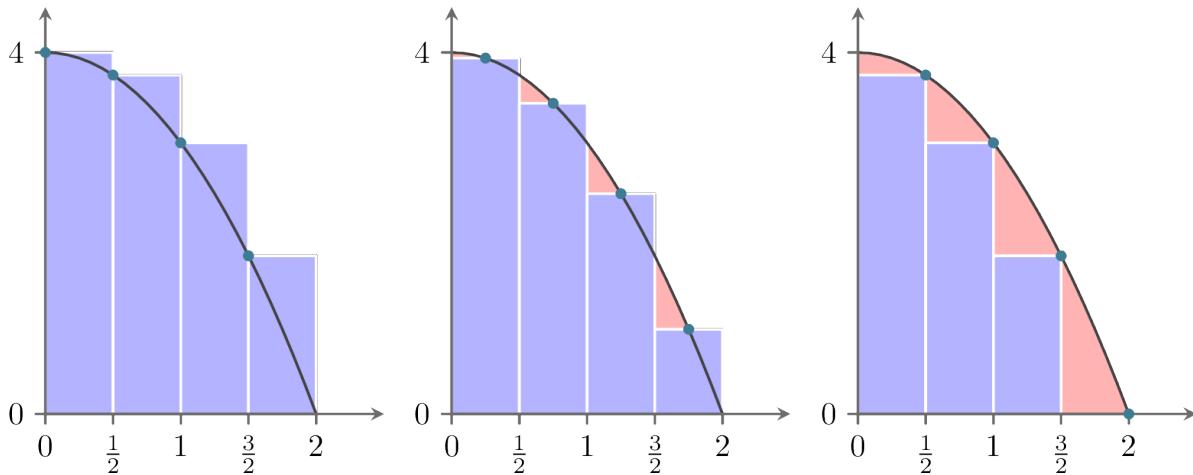


Figure 6.2: Left-aligned Riemann sum, midpoint-aligned Riemann sum, and right-aligned Riemann sum

Clearly, the more rectangles we choose the closer the sum of the areas of the rectangles will get to the integral.

Exercise 6.2. Write a Python function `RiemannSum(f, a, b, N, method='left')` that approximates an integral with a Riemann sum. Your Python function should accept a Python Function `f`, a lower bound `a`, an upper bound `b`, the number of subintervals `N`, and an optional input `method` that allows the user to designate whether they want ‘left’, ‘right’, or ‘midpoint’ rectangles. Test your code on several functions for which you know the integral. You should write your code without any loops.

Exercise 6.3. Consider the function $f(x) = \sin(x)$. We know the antiderivative for this function, $F(x) = -\cos(x) + C$. In this question we are going to get a sense of the order of the error when doing Riemann Sum integration.

1. Find the exact value of

$$I = \int_0^1 f(x)dx. \quad (6.3)$$

2. Now calculate left Riemann Sum approximation (using your `RiemannSum()` function from Exercise 6.2) with various values of Δx . Fill in the table with your results. Note the similarity between this exercise and Exercise 5.8 where you created a similar table for approximating a derivative. If you want to save yourself tedious work, you may want to adapt the code from Exercise 5.9 to this exercise.

Δx	Approx. Integral	Absolute Error
$2^{-1} = 0.5$		
$2^{-2} = 0.25$		
2^{-3}		
2^{-4}		
2^{-5}		
2^{-6}		
2^{-7}		
2^{-8}		

3. There was nothing really special about powers of 2 in part (2) of this exercise. Examine other sequences of Δx with a goal toward answering the question:

If we find an approximation of the integral with a fixed Δx and find an absolute error, then what would happen to the absolute error if we divide Δx by some positive constant M ?

Exercise 6.4. Repeat the previous exercise using the midpoint Riemann sum. Again answer the question what happens to the absolute error if we divide Δx by some positive constant M .

Exercise 6.5. Create a plot with the width of the subintervals on the horizontal axis and the absolute error between your Riemann sum calculations (left, right, and midpoint) and the exact integral for a known definite integral of your choice. Your plot should be on a log-log

scale. Based on your plot, what is the approximate order of the error in the Riemann sum approximation? Again notice the similarity between this exercise and Exercise 5.17 where you created a similar plot for approximating a derivative. And don't be worried if you plot three lines but can only see two of them. The third line is likely very close to one of the other two.

6.2 Trapezoidal Rule

Now let us turn our attention to some slightly better algorithms for calculating the value of a definite integral: The Trapezoidal Rule and Simpson's Rule. There are many others, but in practice these two are relatively easy to implement and have reasonably good error approximations. To motivate the idea of the trapezoidal rule consider Figure 6.3. It is plain to see that trapezoids will make better approximations than rectangles at least in this particular case. Another way to think about using trapezoids, however, is to see the top side of the trapezoid as a secant line connecting two points on the curve. As Δx gets arbitrarily small, the secant lines become better and better approximations for tangent lines and are hence arbitrarily good approximations for the curve. For these reasons it seems like we should investigate how to systematically approximate definite integrals via trapezoids.

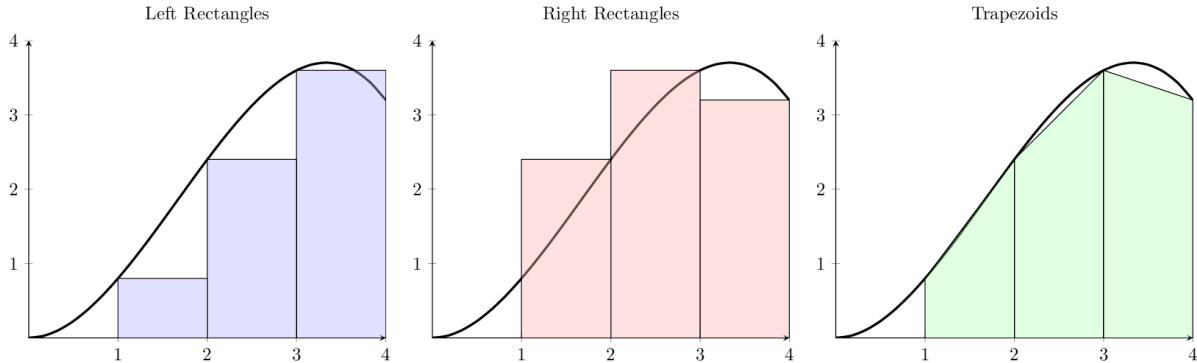


Figure 6.3: Motivation for using trapezoids to approximate a definite integral.

Exercise 6.6. Consider a single trapezoid approximating the area under a curve. From geometry we recall that the area of a trapezoid is

$$A = \frac{1}{2} (b_1 + b_2) h, \quad (6.4)$$

where b_1, b_2 and h are marked in Figure 6.4. The function shown in the picture is $f(x) = \frac{1}{5}x^2(5 - x)$. Find the area of the shaded region as an approximation to

$$\int_1^4 \left(\frac{1}{5}x^2(5 - x) \right) dx. \quad (6.5)$$

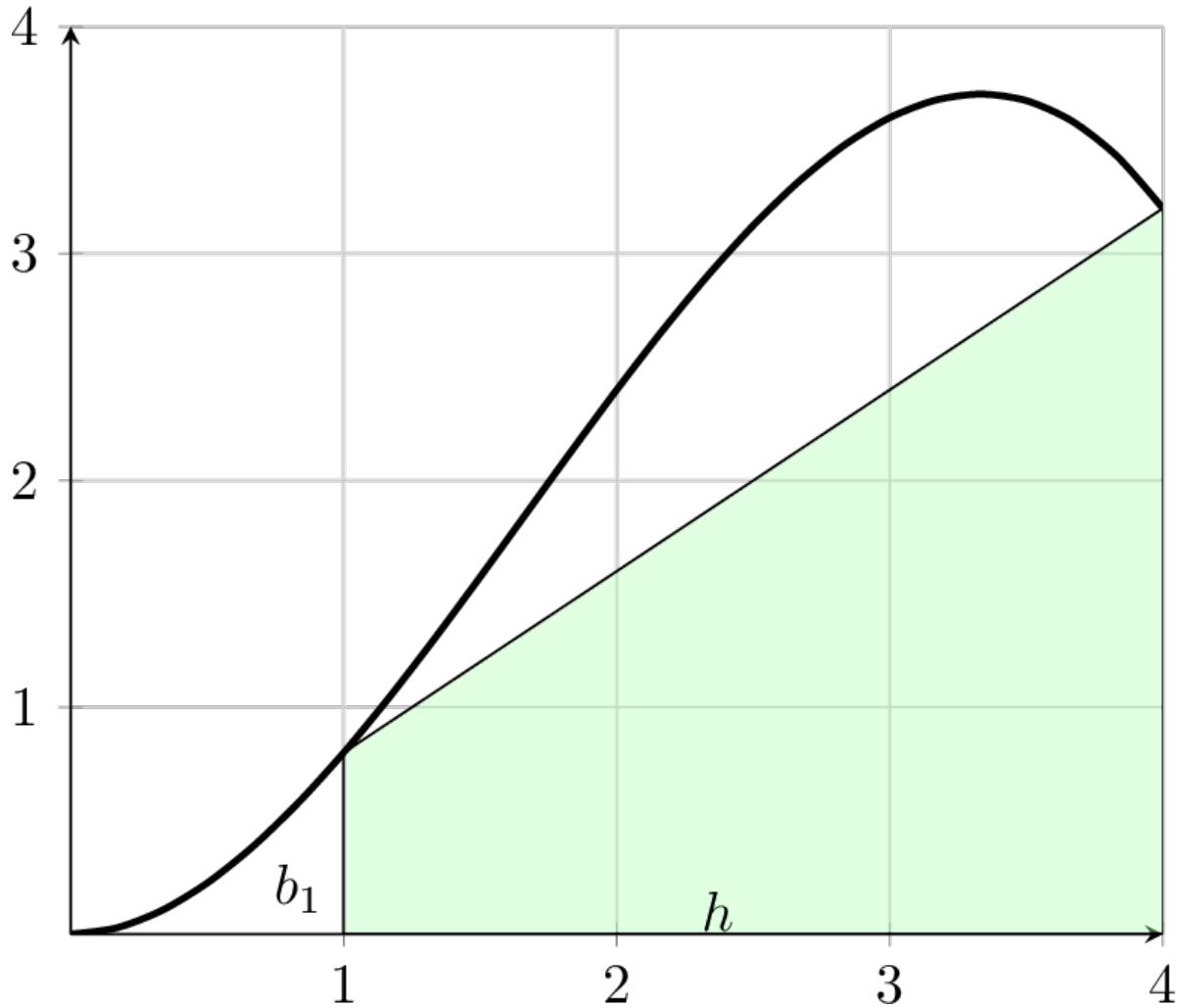


Figure 6.4: A single trapezoid to approximate area under a curve.

Now use the same idea with $h = \Delta x = 1$ to approximate the area under the function using three trapezoids, as illustrated in the last panel of Figure 6.3.

Exercise 6.7 (Trapezoidal Rule). Now generalise the idea from Exercise 6.6 to divide the interval $[a, b]$ into N subintervals with boundaries $\{x_0 = a, x_1, x_2, \dots, x_{N-1}, x_N = b\}$. Fill in the missing bits in the equations below. The area of the trapezoid on the subinterval from x_{j-1} to x_j is

$$A_j = \frac{1}{2} [f(\text{???}) + f(\text{???})] (\text{???} - \text{??}). \quad (6.6)$$

Then the approximation of the integral is

$$\int_a^b f(x)dx \approx \sum_{\text{??}}^{\text{??}} A_j. \quad (6.7)$$

Exercise 6.8. Write a Python function `Trapezoidal(f, a, b, N)` that approximates an integral with the trapezoidal method you derived in the previous exercise. Your Python function should accept a Python Function `f`, a lower bound `a`, an upper bound `b` and the number of subintervals `N`. You should write your code without any loops. Test your code on several functions for which you know the integral.

Exercise 6.9. You have by now developed and repeatedly used ways to investigate how the errors for numerical integration and differentiation schemes depend on the stepsize. It is now up to you to do the same for the trapezoidal rule. Remember that the goal is to answer the question:

If I approximate the integral with a fixed Δx and find an absolute error of P , then what will the absolute error be using a width of $\Delta x/M$?

You can either do this with a table or with a graph. What is the order of the error for the trapezoidal rule?

6.3 Simpsons Rule

The trapezoidal rule does a decent job approximating integrals, but ultimately you are using linear functions to approximate $f(x)$ and the accuracy may suffer if the step size is too large or the function too non-linear. You likely notice that the trapezoidal rule will give an exact answer if you were to integrate a linear or constant function. A potentially better approach would be to get an integral that evaluates quadratic functions exactly. We call this method

Simpson's Rule after [Thomas Simpson \(1710-1761\)](#) who, by the way, was a basket weaver in his day job so he could pay the bills and keep doing math.

Three points are needed to uniquely determine a quadratic function, where two points were enough to uniquely determine a linear function. So for Simpson's method we need to evaluate the function at three points (not two as for the trapezoidal rule). To approximate the integral of a function $f(x)$ on the interval $[a, b]$ we will use the three points $(a, f(a))$, $(m, f(m))$, and $(b, f(b))$ where $m = \frac{a+b}{2}$ is the midpoint of the two boundary points.

We want to find constants A_1 , A_2 , and A_3 in terms of a , b , $f(a)$, $f(b)$, and $f(m)$ such that

$$\int_a^b f(x)dx = A_1f(a) + A_2f(m) + A_3f(b) \quad (6.8)$$

is exact for all constant, linear, and quadratic functions. This would guarantee that we have an exact integration method for all polynomials of order 2 or less but should serve as a decent approximation if the function is not quadratic.

Exercise 6.10. Follow these steps to find A_1 , A_2 , and A_3 .

1. Verify that

$$\int_a^b 1dx = b - a = A_1 + A_2 + A_3. \quad (6.9)$$

2. Verify that

$$\int_a^b xdx = \frac{b^2 - a^2}{2} = A_1a + A_2\left(\frac{a+b}{2}\right) + A_3b. \quad (6.10)$$

3. Verify that

$$\int_a^b x^2dx = \frac{b^3 - a^3}{3} = A_1a^2 + A_2\left(\frac{a+b}{2}\right)^2 + A_3b^2. \quad (6.11)$$

4. Verify that the above linear system of equations has the solution

$$A_1 = \frac{b-a}{6}, \quad A_2 = \frac{4(b-a)}{6}, \quad \text{and} \quad A_3 = \frac{b-a}{6}. \quad (6.12)$$

Exercise 6.11. At this point we can see that an integral can be approximated as

$$\int_a^b f(x)dx \approx \left(\frac{b-a}{6}\right) \left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right) \quad (6.13)$$

and the technique will give an exact answer for any polynomial of order 2 or below.

Verify the previous sentence by integrating $f(x) = 1$, $f(x) = x$ and $f(x) = x^2$ by hand on the interval $[0, 1]$ and using the approximation formula.

To make the punchline of the previous exercises a bit more clear: Using the formula

$$\int_a^b f(x)dx \approx \left(\frac{b-a}{6}\right) (f(a) + 4f(m) + f(b)) \quad (6.14)$$

is the same as fitting a parabola to the three points $(a, f(a))$, $(m, f(m))$, and $(b, f(b))$ and finding the area under the parabola exactly. That is exactly the step up from the trapezoidal rule and Riemann sums that we were after:

- Riemann sums approximate the function with constant functions,
- the trapezoidal rule uses linear functions, and
- now we have a method for approximating with parabolas.

To improve upon this idea we now examine the problem of partitioning the interval $[a, b]$ into small pieces and running this process on each piece. This is called Simpson's Rule for integration.

Exercise 6.12 (Simpson's Rule). We divide the interval $[a, b]$ into N subintervals with boundaries $\{x_0 = a, x_1, x_2, \dots, x_{N-1}, x_N = b\}$. Fill in the missing bits in the equations below. We approximate the integral on the subinterval from x_{j-1} to x_j by

$$A_j = \frac{1}{??} [f(???) + ???f(???) + f(??)] (?? - ???). \quad (6.15)$$

Then the approximation of the integral is

$$\int_a^b f(x)dx \approx \sum_{??}^{??} A_j. \quad (6.16)$$

Exercise 6.13. Write a Python function `Simpsons(f, a, b, N)` that approximates an integral with Simpson's rule that you derived in the previous exercise. Your Python function should accept a Python Function `f`, a lower bound `a`, an upper bound `b` and the number of subintervals `N`. You should write your code without any loops. Test your code on several functions for which you know the integral.

Exercise 6.14. As in Exercise 6.9, use your favourite method to determine how the absolute error in Simpson's rule depends on the step size and hence determine the order of the error for Simpson's rule?

Exercise 6.15. Use the integration problem and exact answer

$$\int_0^{\pi/4} e^{3x} \sin(2x) dx = \frac{3}{13}e^{3\pi/4} + \frac{2}{13} \quad (6.17)$$

and produce a log-log error plot with Δx on the horizontal axis and the absolute error on the vertical axis. Include one graph for each of our integration methods. Fully explain how the error rates show themselves in your plot.

Thus far we have three numerical approximations for definite integrals: Riemann sums (with rectangles), the trapezoidal rule, and Simpson's rule. There are MANY other approximations for integrals and we leave the further research to the curious reader.

Further reading: Sections 4.3 to 4.9 of (Burden and Faires 2010).

6.4 Algorithm Summaries

Exercise 6.16. Explain how to approximate the value of a definite integral with Riemann sums. When will the Riemann sum approximation be exact? Distinguish between left, right and midpoint Riemann sums. State how the error of these approximations depends on the step size, i.e., give the order of the error for each of the three Riemann sums.

Exercise 6.17. Explain how to approximate the value of a definite integral with the trapezoidal rule. When will the trapezoidal rule approximation be exact? What is the order of the Trapezoidal rule?

Exercise 6.18. Explain how to approximate the value of a definite integral with Simpson's rule. Give the full mathematical details for where Simpson's rule comes from. When will the Simpson's rule approximation be exact? What is the order of Simpson's rule?

6.5 Problems

Exercise 6.19. Numerically integrate each of the functions over the interval $[-1, 2]$ with an appropriate technique and verify mathematically that your numerical integral is correct to 10 decimal places. Then provide a plot of the function along with its numerical first derivative.

1. $f(x) = \frac{x}{1+x^4}$
 2. $g(x) = (x-1)^3(x-2)^2$
 3. $h(x) = \sin(x^2)$
-

Exercise 6.20. Write a function that implements the trapezoidal rule on a list of (x, y) order pairs representing the integrand function. The list of ordered pairs should be read from a spreadsheet file. Create a test spreadsheet file and a test script showing that your function is finding the correct integral.

Exercise 6.21. Use numerical integration to answer the question in each of the following scenarios

1. We measure the rate at which water is flowing out of a reservoir (in gallons per second) several times over the course of one hour. Estimate the total amount of water which left the reservoir during that hour.

time (min)	0	7	19	25	38	47	55
flow rate (gal/sec)	316	309	296	298	305	314	322

You can download the data directly from the github repository for this course with the code below.

```
import numpy as np
import pandas as pd
data = np.array(pd.read_csv('https://github.com/gustavdelius/NumericalAnalysis2025/raw/main/'))
```

2. The department of transportation finds that the rate at which cars cross a bridge can be approximated by the function

$$f(t) = \frac{22.8}{3.5 + 7(t - 1.25)^4}, \quad (6.18)$$

where $t = 0$ at 4pm, and is measured in hours, and $f(t)$ is measured in cars per minute. Estimate the total number of cars that cross the bridge between 4 and 6pm. Make sure that your estimate has an error less than 5% and provide sufficient mathematical evidence of your error estimate.

Exercise 6.22. Consider the integrals

$$\int_{-2}^2 e^{-x^2/2} dx \quad \text{and} \quad \int_0^1 \cos(x^2) dx. \quad (6.19)$$

Neither of these integrals have closed-form solutions so a numerical method is necessary. Create a log-log plot that shows the errors for the integrals with different values of h (log of h on the x -axis and log of the absolute error on the y -axis). Write a complete interpretation of the log-log plot. To get the *exact* answer for these plots use Python's `scipy.integrate.quad` command. (What we are really doing here is comparing our algorithms to Python's `scipy.integrate.quad()` algorithm).

6.6 Projects

In this section we propose several ideas for projects related to numerical Calculus. 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.

6.6.1 Higher Order Integration

Riemann sums can be used to approximate integrals and they do so by using piecewise constant functions to approximate the function. The trapezoidal rule uses piece wise linear functions to approximate the function and then the area of a trapezoid to approximate the area. We saw earlier that Simpson’s rule uses piece wise parabolas to approximate the function. The process which we used to build Simpson’s rule can be extended to any higher-order polynomial. Your job in this project is to build integration algorithms that use piece wise cubic functions, quartic functions, etc. For each you need to show all of the mathematics necessary to derive the algorithm, provide several test cases to show that the algorithm works, and produce a numerical experiment that shows the order of accuracy of the algorithm.

6.6.2 Dam Integration

Go to the USGS water data repository:

<https://maps.waterdata.usgs.gov/mapper/index.html>.

Here you will find a map with information about water resources around the country.

- Zoom in to a dam of your choice (make sure that it is a dam).
- Click on the map tag then click “Access Data”
- From the drop down menu at the top select either “Daily Data” or “Current / Historical Data.” If these options do not appear then choose a different dam.
- Change the dates so you have the past year’s worth of information.
- Select “Tab-separated” under “Output format” and press Go. Be sure that the data you got has a flow rate (ft^3/sec).
- At this point you should have access to the entire data set. Copy it into a `csv` file and save it to your computer.

For the data that you just downloaded you have three tasks: (1) plot the data in a reasonable way giving appropriate units, (2) find the total amount of water that has been discharged from the dam during the past calendar year, and (3) report any margin of error in your calculation based on the numerical method that you used in part (2).

6.6.3 WHO Data Integration

Go to [data.gov](#) or the [World Health Organization Data Repository](#) and find a data set where the variables naturally lead to a meaningful definite integral. Use appropriate code to evaluate the definite integral. If your data appears to be subject to significant noise then you might want to smooth the data first before doing the integral. Write a few sentences explaining what the integral means in the context of the data. Be very cautious of the units on the data sets and the units of your answer.

7 Optima

It is not enough to do your best; you must know what to do, and then do your best.

—W. Edwards Deming

In applied mathematics we are often not interested in all solutions of a problem but in the optimal solution. Optimization therefore permeates many areas of mathematics and science. In this section we will look at a few examples of optimization problems and the numerical methods that can be used to solve them.

Exercise 7.1. Here is an atypically easy optimisation problem that you can quickly do by hand:

A piece of cardboard measuring 20cm by 20cm is to be cut so that it can be folded into a box without a lid (see Figure 7.1). We want to find the size of the cut, x , that maximizes the volume of the box.

1. Write a function $V(x)$ for the volume of the box resulting from a cut of size x . What is the domain of your function?
 2. We know that we want to maximize this function so go through the full Calculus exercise to find the maximum:
 - take the derivative $V'(x)$
 - set it to zero to find the critical points
 - determine the critical point that gives the maximum volume
-

An optimization problem is approached by first writing the quantity you want to optimize as a function of the parameters of the model. In the previous exercise that was the function $V(x)$ that gives the volume of the box as a function of the parameter x , which was the length of the cut. That function then needs to be maximized (or minimized, depending on what is optimal).

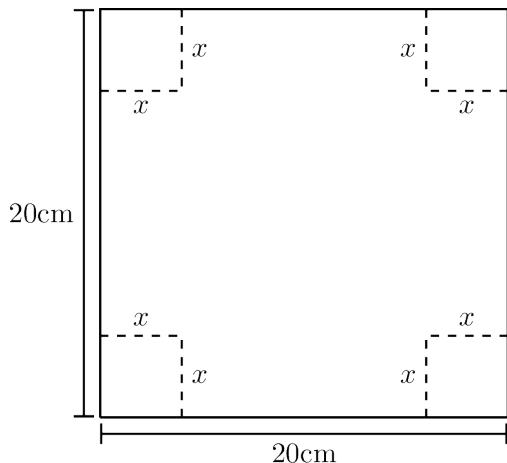


Figure 7.1: Folds to make a cardboard box

Exercise 7.2. In the previous example it was easy to find the value of x that maximized the function analytically. However, in many cases it is not so easy. The equation for the parameters that arises from setting the derivatives to zero is usually not solvable analytically. In these cases we need to use numerical methods to find the extremum. Take for example the function

$$f(x) = e^{-x^2} + \sin(x^2)$$

on the domain $0 \leq x \leq 1.5$. The maximum of this function on this domain can not be determined analytically.

Use Python to make a plot of this function over this domain. You should get something similar to the graph shown in Figure 7.2. What is the x that maximizes the function on this domain? What is the x that minimizes the function on this domain?

The intuition behind numerical optimization schemes is typically to visualize the function as representing a landscape on which you are trying to walk to the highest or lowest point. You however can only sense your immediate neighbourhood and need to use that information to make decisions about where to walk next.

Exercise 7.3. If you were blind folded and standing on a hillside, could you find the top of the hill? (assume no trees and no cliffs ...this is not supposed to be dangerous) How would you do it? Explain your technique clearly.

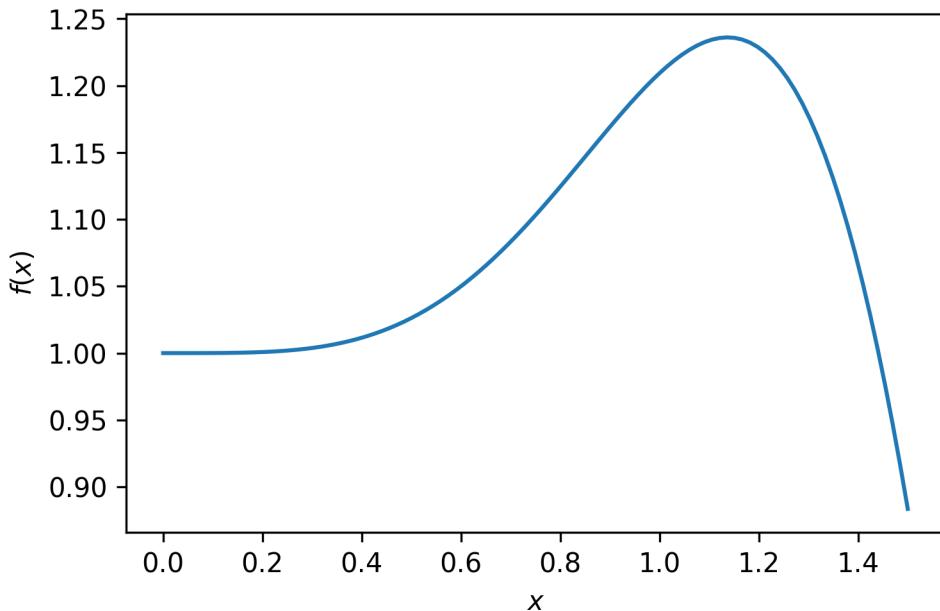


Figure 7.2: Graph of the function $f(x) = e^{-x^2} + \sin(x^2)$.

Exercise 7.4. If you were blind folded and standing on a crater on the moon could you find the lowest point? How would you do it? Remember that you can hop as far as you like ... because gravity ... but sometimes that's not a great thing because you could hop too far.

Clearly there is no difference between finding the maximum of a function and finding the minimum of a function. The maximum of a function f is exactly at the same point as the minimum of the function $-f$. For concreteness we will from now on focus on finding the minimum of a function.

7.1 Single Variable Optimization

The preceding thought exercises have given you intuition about finding extrema in a two-dimensional landscape. But first we will reduce back to one-dimensional optimization problems before generalising to multiple dimensions in the next section.

Exercise 7.5. Did you come up with ideas in the previous two exercises for how you would go about finding a minimum or maximum of a function $f(x)$? If so, try to turn your ideas into step-by-step algorithms which could be coded. Then try out your codes on the function

$$f(x) = -e^{-x^2} - \sin(x^2) \quad (7.1)$$

to see if your algorithms can find the local minimum near $x \approx 1.14$. Try to generate several different algorithms.

One obvious method would be to simply evaluate the function at many points and choose the smallest value. This is called a *brute force* search.

```
import numpy as np
x = np.linspace(0,1.5,1000)
f = -np.exp(-x**2) - np.sin(x**2)
print(x[np.argmin(f)])
```

This method is not very efficient. Just think about how often you would need to evaluate the function for the above approach to give the answer to 12 decimal places. It would be a lot! Your method should be more efficient.

The advantage of this brute force method is that it is guaranteed to find the global minimum in the interval. Any other, more efficient method can get stuck in local minima.

7.1.1 Golden Section Search

Here is an idea for a method that is similar to the bisection method for root finding.

In the bisection method we needed a starting interval so that the function values had opposite signs at the endpoints. You were therefore guaranteed that there would be at least one root in that interval. Then you chose a point in the middle of the interval and by looking at the function value at that new point were able to choose an appropriate smaller interval that was still guaranteed to contain a root. By repeating this you honed in on the root.

Unfortunately by just looking at the function values at two points there is no way of knowing whether there is a minimum between them. However, if you were to look at the function values at three points and found that the value at the middle point was less than the values at the endpoints then you would know that there was a minimum between the endpoints.

Exercise 7.6. Make a sketch of a function and choose three points on the function such that the middle point is lower than the two outer points. Use this to illustrate that there must be at least a local minimum between the two outer points.

The idea now is to choose a new point between the two outer points, compare the function value there to those at the previous three points, and then choose a new triplet of points that is guaranteed to contain a minimum. By repeating this process you would hone in on the minimum.

Exercise 7.7. You want to find a minimum of a continuous function f using the golden section search method. You start with the three points $a = 1, c = 3, b = 5$ where the function takes the values $f(1) = 5, f(3) = 2, f(5) = 3$. For the next step you decided to add the point $d = 2.5$ and find that $f(2.5) = 1$. Which three points should you choose to continue the search?

Exercise 7.8. Complete the following function to implement this idea. You need to think about how to choose the new point and then how to choose the new triplet.

```
def golden_section(f, a, b, c, tol = 1e-12):
    """
    Find an approximation of a local minimum of a function f within the
    interval [a, b] using a bracketing method.

    The function narrows down the interval [a, b] by maintaining a
    triplet (a, c, b) where f(c) < f(a) and f(c) < f(b).
    The process iteratively updates the triplet to home in on the minimum,
    stopping when the interval is smaller than `tol`.

    Parameters:
    f (function): A function to minimize.
    a, b (float): The initial interval bounds where the minimum is to be
                  searched. It is assumed that a < b.
    tol (float): The tolerance for the width of the interval.
    """
```

```

c (float): An initial point within the interval (a, b) where
          f(c) < f(a) and f(c) < f(b).
tol (float): The tolerance for the convergence of the algorithm.
             The function stops when b - a < tol.

Returns:
float: An approximation of a point where f achieves a local minimum.
"""

# Check that the point are ordered a < c < b

# Check that the function value at `c` is lower than at both `a` and `b`

# Loop until you have an interval smaller than the tolerance
while b-a > tol:

    # Choose a new point `d` between `a` and `b`
    # Think about what is the most efficient choice

    # Compare f(d) with f(c) and use the result
    # to choose a new triplet `a`, `b`, `c` in such a way that
    # b-a has decreased but f(c) is still lower than both f(a) and f(b)

    # While debugging, include a print statement to let you know what
    # is happening within your loop

return c

```

Then try out your code on the function

$$f(x) = -e^{-x^2} - \sin(x^2) \quad (7.2)$$

to see if it can find the local minimum near $x \approx 1.14$.

7.1.2 Gradient Descent

Let us next explore the intuitive method of simply taking steps in the downhill direction. That should eventually bring us to a local minimum. The problem is only to know how to choose the step size and the direction. The gradient descent method is a simple and effective way

to do this. By making the step be proportional to the negative gradient of the function we are guaranteed to be moving in the right direction and we are also automatically reducing the step size as we get closer to the minimum where the gradient gets smaller.

Let $f(x)$ be the objective function which you are seeking to minimize.

- Find the derivative of your objective function, $f'(x)$.
 - Pick a starting point, x_0 .
 - Pick a small control parameter, α (in machine learning this parameter is called the “learning rate” for the gradient descent algorithm).
 - Use the iteration $x_{n+1} = x_n - \alpha f'(x_n)$.
 - Iterate (decide on a good stopping rule)
-

Exercise 7.9. What is the Gradient Ascent/Descent algorithm doing geometrically? Draw a picture and be prepared to explain to your peers.

Exercise 7.10. Write code to implement the 1D gradient descent algorithm and use it to solve Exercise 7.1. Compare your answer to the analytic solution.

```
def gradient_descent(df, x0, learning_rate,
                     tol = 1e-12, max_iter=10000):
    """
    Find an approximation of a local minimum of a function f
    using the gradient descent method.

    The function iteratively updates the current guess `x0`
    by moving in the direction of the negative gradient
    of `f` at `x0` multiplied by `alpha`.

    The process stops when the magnitude of the gradient
    is smaller than `tol`.

    Parameters:
    df (function): The derivative of the function you
                   want to minimize.
    x0 (float): The initial guess for the minimum.
    learning_rate (float): The learning rate multiplies the
```

```

        gradient to give the step size.
tol (float): The tolerance for the convergence.
    The function stops when |df(x0)| < tol.
max_iter (int): The maximum number of iterations to perform.

Returns:
float: An approximation of a point where f achieves
      a local minimum.

"""
# Initialize x with the starting value
x = x0
# Loop for a maximum of `max_iter` iterations
for i in range(max_iter):
    # Calculate the step size by multiplying the learning
    # rate with the derivative at the current guess

    # Update `x` by subtracting the step

    # If the step size was smaller than `tol` then
    # return the new `x`

    # If the loop finishes without returning then print a
    # warning that the last step size was larger than `tol`.

# Return the last `x` value

```

Exercise 7.11. Compare and contrast the methods you came up with in Exercise 7.5 with the methods proposed in Exercise 7.8 and Exercise 7.10.

1. What are the advantages to each of the methods proposed?
2. What are the disadvantages to each of the methods proposed?
3. Which method, do you suppose, will be faster in general? Why?
4. Which method, do you suppose, will be slower in general? Why?

Exercise 7.12. Make a plot of the log of the absolute error at iteration $k + 1$ against the log of the absolute error at iteration k , similar to Figure 4.4, for several methods and several choices of function. What do you observe?

Exercise 7.13. Modify your code from Exercise 7.12 so that it prints out the slope and intercept of the best-fit line to the graph. Then use this with the function $f(x) = \cos(x)$ with starting value $x_0 = 3$ and learning rate 0.1 and tolerance 10^{-12} .

Exercise 7.14. Try out your algorithms to find the minimum of the function

$$f(x) = (\sin(4x) + 1)((x - 5)^2 - 25)$$

on the domain $0 \leq x \leq 8$.

```
import numpy as np
import matplotlib.pyplot as plt
f = lambda x: (np.sin(4*x)+1)*((x-5)**2-25)
x = np.linspace(0,8,100)
plt.plot(x,f(x))
```

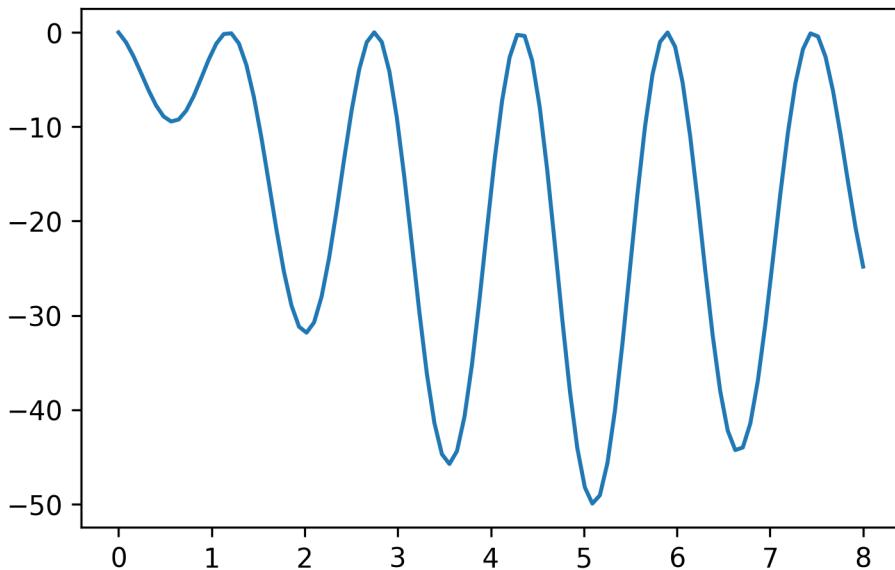


Figure 7.3: Graph of the function $f(x) = (\sin(4x) + 1)((x - 5)^2 - 25)$.

If you choose $x_0 = 3$ as your starting point and a learning rate of 0.001, what approximation do you get for x at the minimum?

Exercise 7.15. Experiment with different values of the learning rate for the previous question, assuming that you can only specify it with up to 3 digits after the decimal point. Which choice of learning rate requires the smallest number of steps for the required tolerance of 10^{-12} ?

7.2 Multivariable Optimization

Now let us look at multi-variable optimization. The idea is the same as in the single-variable case. We want to find the minimum of a function $f(x_1, x_2, \dots, x_n)$. Such higher-dimensional problems are very common and the dimension n can be very large in practical problems. A good example is the loss function of a neural network which is a function of the weights and biases of the network. In a large language model the loss function is a function of many billions of variables and the training of the model is a large optimization problem.

Here is a two-variable example: Find the minimum of the function

$$f(x, y) = \sin(x) \exp\left(-\sqrt{x^2 + y^2}\right)$$

```
import numpy as np
import plotly.graph_objects as go

f = lambda x, y: np.sin(x)*np.exp(-np.sqrt(x**2+y**2))

# Generating values for x and y
x = np.linspace(-2, 2, 100)
y = np.linspace(-1, 3, 100)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)

# Creating the plot
fig = go.Figure(data=[go.Surface(z=Z, x=X, y=Y)])
fig.update_layout(width=800, height=800)
```

Finding the minima of multi-variable functions is a bit more complicated than finding the minima of single-variable functions. The reason is that there are many more directions in which to move. But the basic intuition that we want to move downhill to move towards a minimum of course still works. The gradient descent method is still a good choice for finding

Unable to display output for mime type(s): text/html

(a) Graph of the function $\sin(x) \exp(-\sqrt{x^2 + y^2})$.

Unable to display output for mime type(s): text/html

(b)

(c)

the minimum of a multi-variable function. The only difference is that the gradient is now a vector and the step is in the direction of the negative gradient.

Exercise 7.16. In your group, answer each of the following questions to remind yourselves of multivariable calculus.

1. What is a partial derivative (explain geometrically). For the function $f(x, y) = \sin(x) \exp(-\sqrt{x^2 + y^2})$ what is $\frac{\partial f}{\partial x}$ and what is $\frac{\partial f}{\partial y}$?
 2. What is the gradient of a function? What does it tell us physically or geometrically? If $f(x, y) = \sin(x) \exp(-\sqrt{x^2 + y^2})$ then what is ∇f ?
-

Below we will give the full description of the gradient descent algorithm.

7.2.1 Gradient Descent Algorithm

We want to solve the problem

$$\text{minimize } f(x_1, x_2, \dots, x_n) \text{ subject to } (x_1, x_2, \dots, x_n) \in S. \quad (7.3)$$

1. Choose an arbitrary starting point $x_0 = (x_1, x_2, \dots, x_n) \in S$.
2. We are going to define a difference equation that gives successive guesses for the optimal value:

$$x_{n+1} = x_n - \alpha \nabla f(x_n). \quad (7.4)$$

The difference equation says to follow the negative gradient a certain distance from your present point (why are we doing this). Note that the value of α is up to you so experiment with a few values (you should probably take $\alpha \leq 1$...why?).

3. Repeat the iterative process in step 2 until two successive points are *close enough* to each other.

Exercise 7.17. Write code to implement the gradient descent algorithm for a function $f(x, y)$.

```
def gradient_descent_2d(df, x0, learning_rate, tol=1e-12, max_iter=10000):
    """
    Finds an approximation of a local minimum of a 2D function using
    gradient descent.

    Parameters:
    df (function): A function that returns the gradient of the function to
                    minimize. It should take a NumPy array with two elements
                    as input and return a NumPy array with two elements
                    representing the gradient.
    x0 (NumPy array): The initial guess for the minimum
                      (a NumPy array with two elements).
    learning_rate (float): The learning rate (step size multiplier).
    tol (float): Tolerance for convergence
                  (stops when the magnitude of the step is below this).
    max_iter (int): Maximum number of iterations.

    Returns:
    NumPy array: The approximated minimum point
                 (a NumPy array with two elements).
    """
    # Here comes your code.
```

You can build on your code for the single-variable gradient descent from Exercise 7.10.

Use your function to find the minimum of the function

$$f(x, y) = \sin(x) \exp(-\sqrt{x^2 + y^2}).$$

with a starting point $(x_0, y_0) = (-1, 1)$, a learning rate of 1 and a tolerance of 10^{-6} .

Then find the minimum of the same function with a starting point a starting point $(x_0, y_0) = (0, 0)$, using the same learning rate of 1 and tolerance of 10^{-6} .

Exercise 7.18. How much more complicated would your function have to be to work for a function of n arguments for an arbitrary n ?

It is annoying that one needs to first work out the gradient function by hand before one can use the gradient descent algorithm. This is especially annoying when the function is very complicated. It would be better to use automatic differentiation. If you followed the material on automatic differentiation you will know how to do that.

Of course there are many other methods for finding the minimum of a multi-variable function. An important method that does not need the gradient of the function is the Nelder-Mead method. This method is a direct search method that only needs the function values at the points it is evaluating. The method is very robust and is often used when the gradient of the function is not known or is difficult to calculate. There are also clever variants of the gradient descent method that are more efficient than the basic method. The Adam and RMSprop algorithms are two such methods that are used in machine learning. This subject is a large and active area of research and we will not go into more detail here.

7.3 Optimization with SciPy

You have already seen that there are many tools built into the NumPy and SciPy libraries that will do some of our basic numerical computations. The same is true for numerical optimization problems. Keep in mind throughout the remainder of this section that the whole topic of numerical optimization is still an active area of research and there is much more to the story than what we will see here. However, the Python tools provided by `scipy.optimize` are highly optimized and tend to work quite well.

Exercise 7.19. Let us solve a very simple function minimization problem to get started. Consider the function $f(x) = (x-3)^2 - 5$. A moment's thought reveals that the global minimum of this parabolic function occurs at $(3, -5)$. We can have `scipy.optimize.minimize()` find this value for us numerically. The routine is much like Newton's Method in that we give it a starting point *near* where we think the optimum will be and it will iterate through some algorithm (like a derivative free optimization routine) to approximate the minimum.

```
import numpy as np
from scipy.optimize import minimize
f = lambda x: (x-3)**2 - 5
minimize(f, 2)
```

1. Implement the code above then spend some time playing around with the minimize command to minimize more challenging functions.
 2. Consult the help page and explain what all of the output information is from the `minimize()` command.
-

7.4 Algorithm Summaries

Exercise 7.20. Explain in clear language how the Golden Section Search method works.

Exercise 7.21. Explain in clear language how the Gradient Descent method works.

7.5 Problems

Exercise 7.22. For each of the following functions write code to numerically approximate the local maximum or minimum that is closest to $x = 0$. You may want to start with a plot of the function just to get a feel for where the local extreme value(s) might be.

1. $f(x) = \frac{x}{1+x^4} + \sin(x)$
 2. $g(x) = (x-1)^3 \cdot (x-2)^2 + e^{-0.5 \cdot x}$
-

Exercise 7.23. (This exercise is modified from (Meerschaert 2013))

A pig weighing 200 pounds gains 5 pounds per day and costs 45 cents a day to keep. The market price for pigs is 65 cents per pound, but is falling 1 cent per day. When should the pig be sold to maximize the profit?

Write the expression for the profit $P(t)$ as a function of time t and maximize this analytically (by hand). Then solve the problem with all three methods outlined in Section 7.1.

Exercise 7.24. (This exercise is modified from (Meerschaert 2013))

Reconsider the pig Exercise 7.23 but now suppose that the weight of the pig after t days is

$$w = \frac{800}{1 + 3e^{-t/30}} \text{ pounds.} \quad (7.5)$$

When should the pig be sold and how much profit do you make on the pig when you sell it? Write this situation as a single variable mathematical model. You should notice that the algebra and calculus for solving this problem is no longer really a desirable way to go. Use an appropriate numerical technique to solve this problem.

Exercise 7.25. Go back to your old Calculus textbook or homework and find your favourite optimization problem. State the problem, create the mathematical model, and use any of the numerical optimization techniques in this chapter to get an approximate solution to the problem.

Exercise 7.26. (The Goat Problem) This is a classic problem in recreational mathematics that has a great approximate solution where we can leverage some of our numerical analysis skills. Grab a pencil and a piece of paper so we can draw a picture.

- Draw a coordinate plane
- Draw a circle with radius 1 unit centred at the point $(0, 1)$. This circle will obviously be tangent to the x axis.
- Draw a circle with radius r centred at the point $(0, 0)$. We will take $0 < r < 2$ so there are two intersections of the two circles.
 - Label the left-hand intersection of the two circles as point A . (Point A should be in the second quadrant of your coordinate plane.)
 - Label the right-hand intersection of the circles as point B . (Point B should be in the first quadrant of your coordinate plane.)
- Label the point $(0, 0)$ as the point P .

A rancher has built a circular fence of radius 1 unit centred at the point $(0, 1)$ for his goat to graze. He tethers his goat at point P on the far south end of the circular fence. He wants to make the length of the goat's chain, r , just long enough so that it can graze half of the area of the fenced region. How long should he make the chain?

Hints:

- It would be helpful to write equations for both circles. Then you can use the equations to find the coordinates of the intersection points A and B .
 - You can either solve for the intersection points algebraically or you can use a numerical root finding technique to find the intersection points.
 - In any case, the intersection points will (obviously) depend on the value of r
 - Set up an integral to find the area grazed by the goat.
 - You will likely need to use a numerical integration technique to evaluate the integral.
 - Write code to narrow down on the best value of r where the integral evaluates to half the area of the fenced region.
-

7.6 Projects

In this section we propose several ideas for projects related to numerical optimisation. 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.

7.6.1 Edge Detection in Images

Edge detection is the process of finding the boundaries or edges of objects in an image. There are many approaches to performing edge detection, but one method that is quite robust is to use the gradient vector in the following way:

- First convert the image to gray scale.
- Then think of the gray scale image as a plot of a multivariable function $G(x, y)$ where the ordered pair (x, y) is the pixel location and the output $G(x, y)$ is the value of the gray scale at that point.
- At each pixel calculate the gradient of the function $G(x, y)$ numerically.
- If the magnitude of the gradient is larger than some threshold then the function $G(x, y)$ is steep at that location and it is possible that there is an edge (a transition from one part of the image to a different part) at that point. Hence, if $\|\nabla G(x, y)\| > \delta$ for some threshold δ then we can mark the point (x, y) as an edge point.

Your Tasks:

1. Choose several images on which to do edge detection. You should take your own images, but if you choose not to be sure that you cite the source(s) of your images.
2. Write Python code that performs edge detection as described above on the image. In the end you should produce side-by-side plots of the original picture and the image showing only the edges. To calculate the gradient use a centred difference scheme for the first derivatives

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}. \quad (7.6)$$

In an image we can take $h = 1$ (why?), and since the gradient is two dimensional we get

$$\nabla G(x, y) \approx \left\langle \frac{G(x+1, y) - G(x-1, y)}{2}, \frac{G(x, y+1) - G(x, y-1)}{2} \right\rangle. \quad (7.7)$$

Figure 7.5 depicts what this looks like when we zoom in to a pixel and its immediate neighbours. The pixel labelled $G[i, j]$ is the pixel at which we want to evaluate the gradient, and the surrounding pixels are labelled by their indices relative to $[i, j]$.

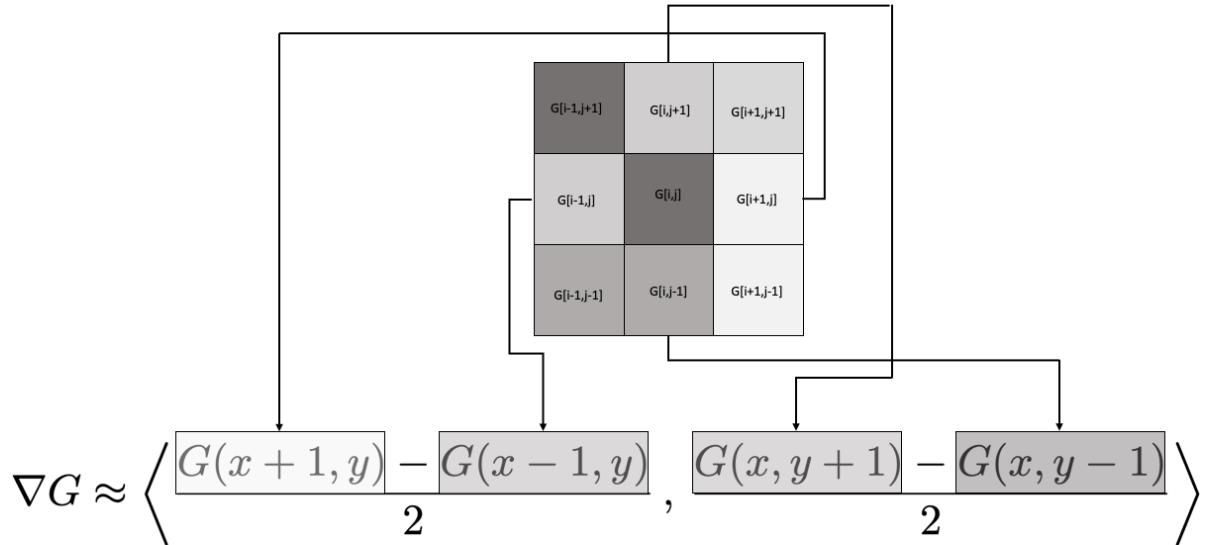


Figure 7.5: The gradient computation on a single pixel using a central difference scheme for the first derivative.

3. There are many ways to approximate numerical first derivatives. The simplest approach is what you did in part (2) – using a centred difference scheme. However, pixels are necessarily tightly packed in an image and the immediate neighbours of a point may not have enough contrast to truly detect edges. If you examine Figure 7.5 you will notice that

we only use 4 of the 8 neighbours of the pixel $[i, j]$. Also notice that we did not reach out any further than a single pixel. Your job now is to build several other approaches to calculating the gradient vector, implement them to perform edge detection, and show the resulting images. For each method you need to give the full mathematical details for how you calculated the gradient as well as give a list of pros and cons for using the new numerical gradient for edge detection based on what you see in your images. As an example, you could use a centred difference scheme that looks two pixels away instead of at the immediate neighbouring pixels

$$f'(x) \approx \frac{???f(x-2) + ???f(x+2)}{???}. \quad (7.8)$$

Of course you would need to determine the coefficients in this approximation scheme. Another idea could use a centred difference scheme that uses pixels that are immediate neighbours AND pixels that are two units away

$$f'(x) \approx \frac{???f(x-2) + ???f(x-1) + ???f(x+1) + ???f(x+2)}{???}. \quad (7.9)$$

In any case, you will need to use Taylor Series to derive coefficients in the formulas for the derivatives as well as the order of the error. There are many ways to approximate the first derivatives so be creative. In your exploration you are not restricted to using just the first derivative. There could be some argument for using the second derivatives and/or the Hessian matrix of the gray scale image function $G(x, y)$ and using some function of the concavity as a means of edge detection. Explore and have fun!

The following code will allow you to read an image into Python as an `np.array()`.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import image
I = np.array(image=plt.imread('ImageName.jpg'))
plt.imshow(I)
plt.axis("off")
plt.show()
```

You should notice that the image, I , is a three dimensional array. The three layers are the red, green, and blue channels of the image. To flatten the image to gray scale you can apply the rule

$$\text{grayscale value} = 0.3\text{Red} + 0.59\text{Green} + 0.11\text{Blue}. \quad (7.10)$$

The output should be a 2 dimensional `numpy` array which you can show with the following Python code.

```

plt.imshow(G, cmap='gray') # "cmap" stands for "color map"
plt.axis("off")
plt.show()

```

Figure 7.6 shows the result of different threshold values applied to the simplest numerical gradient computations. The image was taken by the author.

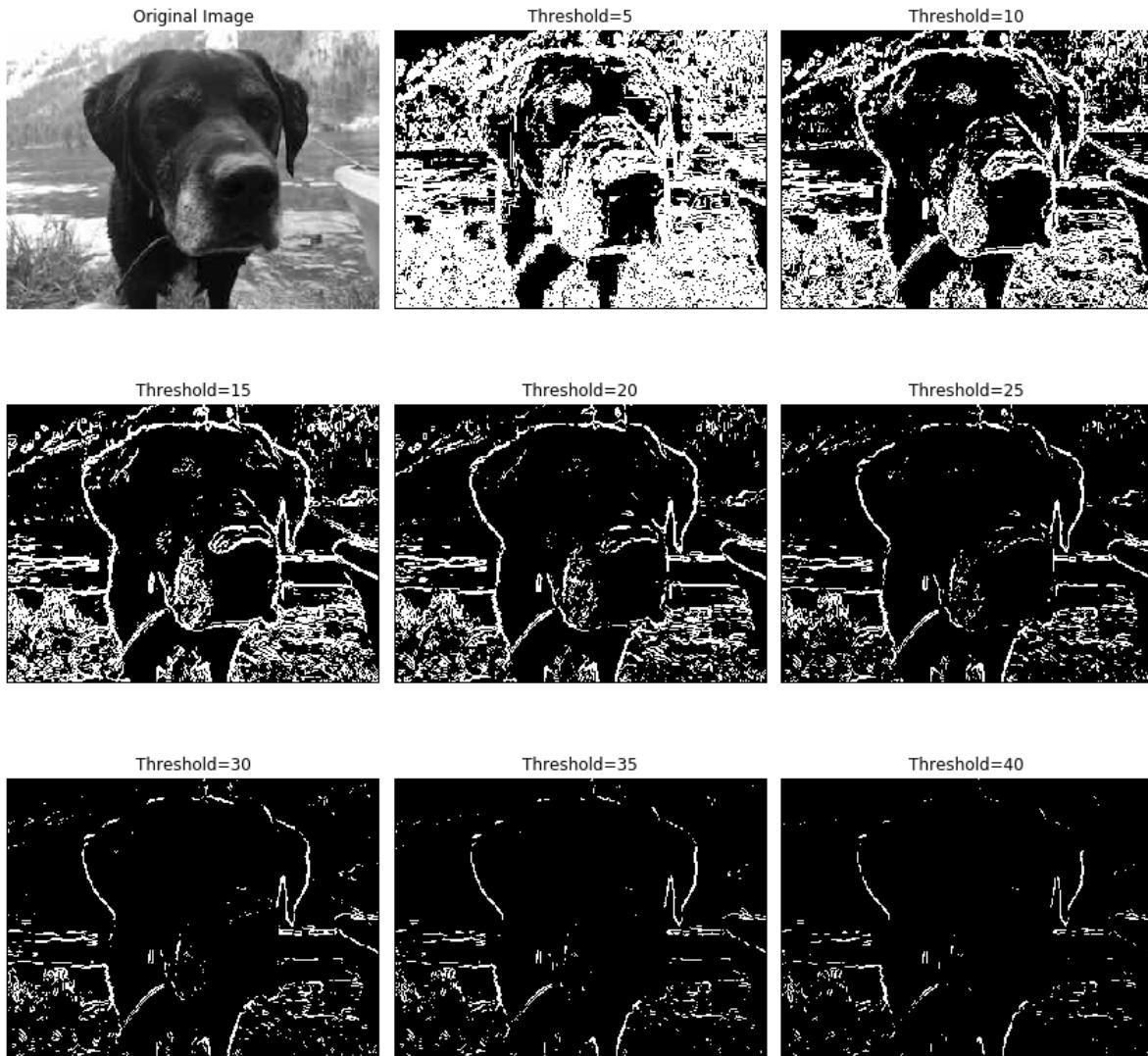


Figure 7.6: Edge detection using different thresholds for the value of the gradient on the grayscale image

8 Ordinary Differential Equations

The mathematical discipline of differential equations furnishes the explanation of all those elementary manifestations of nature which involve time.

—Norwegian Mathematician Sophus Lie

The topic of this chapter is to find *approximate solutions* to *ordinary differential equations*.

Let us briefly recall what an ordinary differential equation (ODE) is. A rather arbitrarily chosen example for an ODE (here, of second order) is

$$y''(x) + 4y'(x) + \sqrt[3]{y(x)} + \cos(x) = 0. \quad (8.1)$$

Equations like this are normally satisfied by many functions $y(x)$: the problem has many solutions. In order to specify a uniquely solvable problem, one needs to fix *initial values*, i.e., the value of y and its first derivative at some point, say, at $x = 0$:

$$y''(x) + 4y'(x) + \sqrt[3]{y(x)} + \cos(x) = 0, \quad y(0) = 1, \quad y'(0) = -2. \quad (8.2)$$

This is a so-called *initial-value problem* (IVP). Another variant is to specify the value of $y(x)$, but not of its derivative, at two different points:

$$y''(x) + 4y'(x) + \sqrt[3]{y(x)} + \cos(x) = 0, \quad y(0) = 2, \quad y(1) = 1. \quad (8.3)$$

This is called a *boundary value problem* (BVP).

Both IVPs and BVPs have a unique solution (under certain mathematical conditions). However, while one can show on abstract grounds that these solutions exist, it is often not practicable to find an explicit expression for them. The best one can hope for is to approximate the solution numerically.

So what *is* a numerical solution to a differential equation?

When solving a differential equation with analytic techniques the goal is to find an expression for the solution in terms of known functions. In a numerical solution the goal is typically to divide the domain for the solution function into a fine partition by introducing a grid of points, just like we did with numerical differentiation and integration, and then to approximate the solution to the differential equation at each point in that partition. Hence, the end result will be a list of approximate solution values at the grid points.

In this chapter we will examine some of the more common ways to create approximations of solutions to initial value problems. Moreover, we will lean heavily on Taylor Series to give us ways to accurately measure the order of the errors that we make in the process.

In this chapter we will often think of the argument of the function described by the ODE as time, but of course the methods are agnostic to the interpretation of the independent variable.

8.1 Euler's Method

Exercise 8.1. Consider the differential equation $x' = -0.5x$ with the initial condition $x(0) = 6$.

- Since we know that $x(0) = 6$ and we know that $x'(0) = -0.5x(0)$ we can approximate the value of x at some future time step. Let us go 1 unit forward in time. That is, approximate $x(1)$ knowing that $x(0) = 6$ and $x'(0) = -3$.

Hint: We know a value, a slope, and the size of the step that we would like to move in the t direction.

$$x(1) \approx \text{_____} \quad (8.4)$$

- Use your answer from part (a) for time $t = 1$ to approximate the x value at time $t = 2$. Then use that value to approximate the value at time $t = 3$. Repeat the process to approximate the value of x at times $t = 2, 3, 4$. Record your answers in the table below. Then find the analytic solution to this differential equation and record the x values at the appropriate times.

t	0	1	2	3	4
Approximation of $x(t)$	6				
Exact value of $x(t)$	6				

- The “approximations of x ” that you found in part (b) are a **numerical approximation** of the solution to the differential equation. You should notice that your numerical solution is pretty far off from the actual solution for most values of t . Why? What could be the sources of this error and how could we fix it? Once you have an idea of how to fix it, put your idea into action and devise some measurement of error to analyse your results.
- In Figure 8.1 you will see a slope field and the exact solution to the differential equation $x' = -0.5x$ with $x(0) = 6$. Mark your approximate solutions at times $t = 1, t = 2, \dots, t = 4$ on the plot and connect them with straight lines.
 - Why are we using straight lines to connect the points?

2. What do you notice about your approximate solutions?
3. Why is it helpful to have the slope field in the background on this plot?

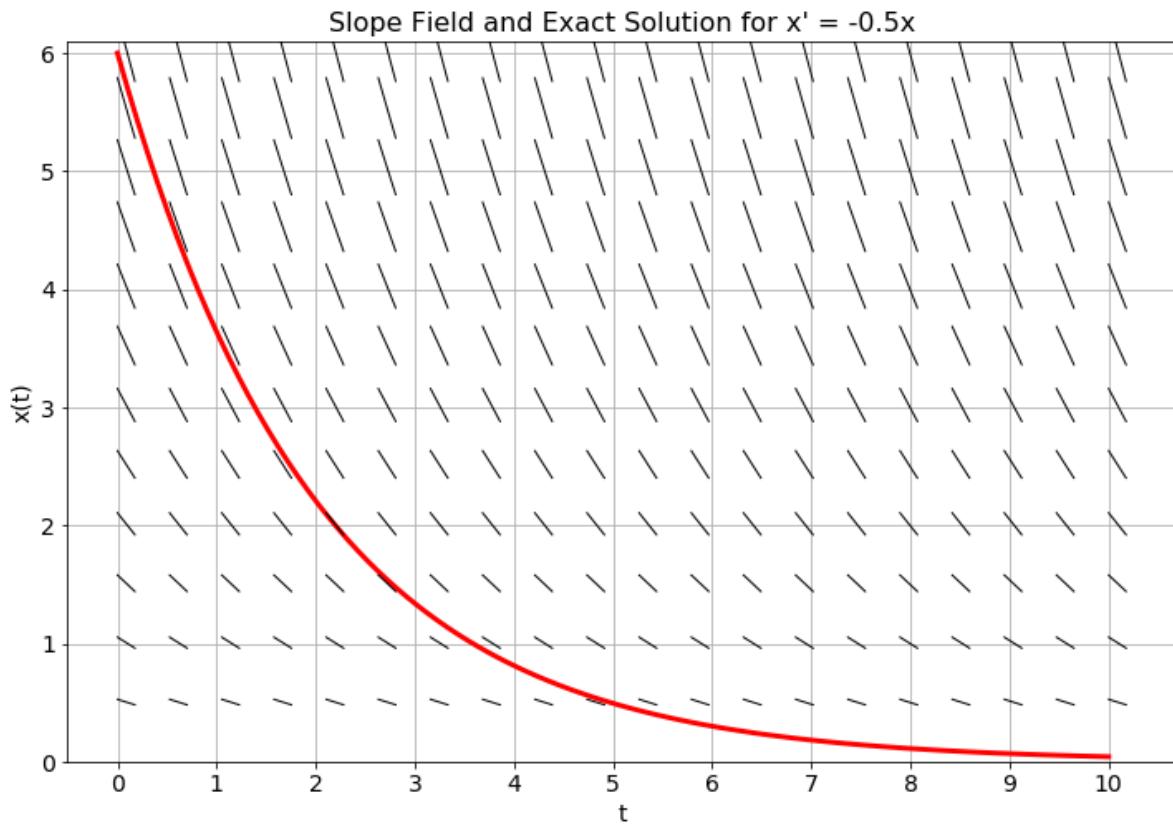


Figure 8.1: Plot your approximate solution on top of the slope field and the exact solution.

Exercise 8.2. In Figure 8.2 you see the analytic solution with initial condition $x(0) = 5$ and a slope field for an unknown differential equation.

- a. Use the slope field and a step size of $\Delta t = 1$ to plot approximate solution values at $t = 1, t = 2, \dots, t = 10$. Connect your points with straight lines. The collection of line segments that you just drew is an approximation to the solution of the unknown differential equation.
- b. Use the slope field and a step size of $\Delta t = 0.5$ to plot approximate solution values at $t = 0.5, t = 1, t = 1.5, \dots, t = 10$. Again, connect your points with straight lines to get an approximation of the solution to the unknown differential equation.

- c. If you could take Δt to be very very small, what difference would you see graphically between the exact solution and your collection of line segments? Why?

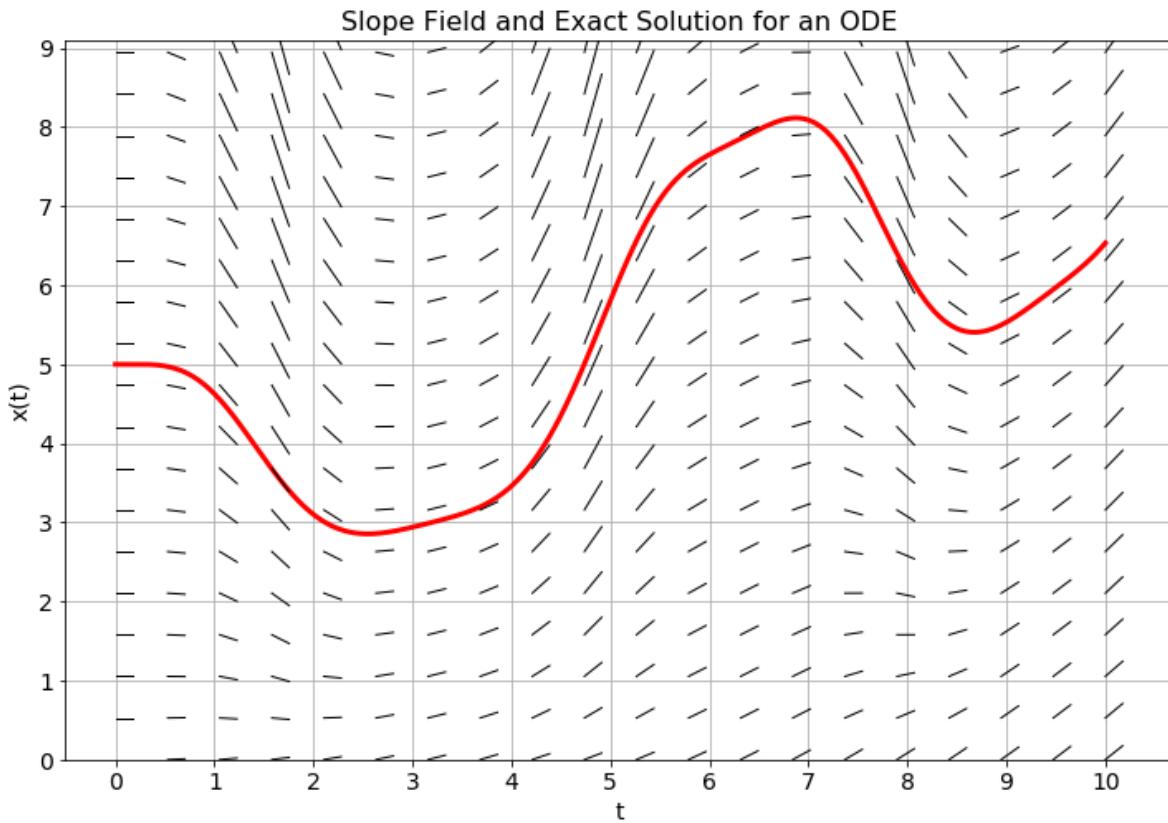


Figure 8.2: Plot your approximate solution on top of the slope field and the exact solution.

The notion of approximating solutions to differential equations is simple in principle:

- make a discrete approximation to the derivative and
- step forward through time as a difference equation.

The challenging part is making the approximation to the derivative(s). There are many methods for approximating derivatives, and that is exactly where we will start.

Definition 8.1 (Euler's Method). Euler's Method is a technique for approximating the solution to the differential equation $x'(t) = f(t, x(t))$. Recall from Exercise 5.6 that the first derivative of a function can be discretized as

$$x'(t) = \frac{x(t+h) - x(t)}{h} + \mathcal{O}(h) \quad (8.5)$$

where $h = \Delta t$ is the step size (or the size of each partition in the domain), so the differential equation $x'(t) = f(t, x(t))$ becomes

$$\frac{x(t+h) - x(t)}{h} \approx f(t, x(t)). \quad (8.6)$$

Rewriting as a difference equation, letting $x_{n+1} = x(t_n + h)$ and $x_n = x(t_n)$, we get

$$x_{n+1} = x_n + hf(t_n, x_n) \quad (8.7)$$

A way to think about Euler's method is that at a given point, the slope is approximated by the value of the right-hand side of the differential equation and then we step forward h units in time following that slope. Figure 8.3 shows a depiction of the idea. Notice in the figure that in regions of high curvature Euler's method will deviate a lot from the exact solution to the differential equation. However, taking the limit as h tends to 0 theoretically gives the exact solution at the trade off of needing infinite computational resources.

Exercise 8.3. Why would Euler's method do poorly in regions where the solution exhibits high curvature?

Exercise 8.4. Consider the differential equation $x'(t) = -2x(t)/3 + 4t$ with initial condition $x(0) = 6$. By hand perform four steps of the Euler method with stepsize $h = 1/2$ to obtain an approximation for $x(1/2), x(1), x(3/2)$ and $x(2)$.

Exercise 8.5. Write code to implement Euler's method for initial value problems. Your function should accept as input a Python function $f(t, x)$, an initial condition, a start time, an end time, and the value of $h = \Delta t$. The output should be vectors for t and x that you can easily plot to show the numerical solution. The code below will get you started.

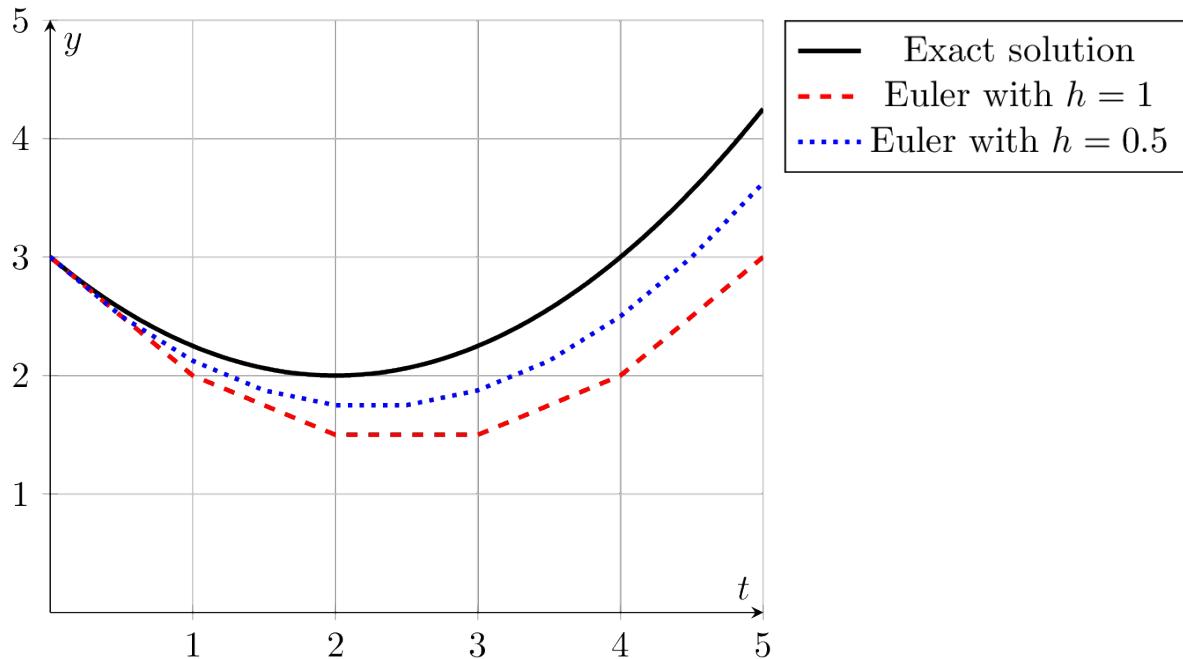


Figure 8.3: Numerical solutions to a differential equation using Euler's method.

```

def euler1d(f, x0, t0, tmax, dt):
    """
    Solves a first-order ordinary differential equation using the Euler method.

    Parameters:
        f      : function, the function defining the differential equation. It should
                  take two arguments, the independent variable t and the dependent
                  variable x, and return the derivative of x with respect to t.
        x0    : float, the initial value of the dependent variable.
        t0    : float, the initial value of the independent variable.
        tmax : float, the maximum value of the independent variable.
        dt    : float, the time step.

    Returns:
        tuple containing two numpy arrays:
            - t : vector of time values.
            - x : vector of solution values at each time value.
    """
    # set up the grid with points from t0 to tmax with stepsize dt
    t = np.arange(??? , ???, ???)

```

```

# set up an array for x that is the same size as t
x = np.zeros(len(t))
# fill in the initial condition
x[0] = ???
for n in range(???): # think about how far we should loop
    # advance the solution forward in time with Euler
    x[n+1] = ???
return t, x

```

Exercise 8.6. Test your code from the previous exercise on a first order differential equation where you know an analytic solution. For example you could use the differential equation

$$x' = -\frac{1}{3}x + \sin(t) \quad \text{where } x(0) = 1. \quad (8.8)$$

This has the analytic solution

$$x(t) = \frac{1}{10} (19e^{-t/3} + 3 \sin(t) - 9 \cos(t)). \quad (8.9)$$

Make a plot of the approximate solution and the exact solution on the same plot for $t \in [0, 10]$. The partial code below should get you started.

```

import numpy as np
import matplotlib.pyplot as plt

# Define the function giving x' in terms of t and x
f = lambda t, x: ???
x0 = ??? # initial condition
t0 = ??? # initial time
tmax = ??? # final time
dt = ??? # time step (your choice, but make it small)
t, x = euler1d(f, x0, t0, tmax, dt)
plt.plot(t, x, 'b-', label='Euler')

# Define a function giving the analytic solution
x_exact = lambda t: ???
# We will plot the exact solution at a higher resolution
t_highres = np.linspace(t0, tmax, 100)
plt.plot(t_highres, x_exact(t_highres), 'r--', label='Exact')
plt.grid()
plt.show()

```

Experiment with different values for dt and see how the numerical solution changes.

Exercise 8.7. The goal of this problem will be to compare the maximum error when you solve the differential equation from the previous exercise on the interval $t \in [0, 10]$ with the Euler method for various values of Δt .

Write a function that produces a plot with the value of Δt on the horizontal axis and the value of the associated absolute error on the vertical axis. You should use a log-log plot. Obviously you will need to run your code many times at many different values of Δt to build your data set. The following incomplete code will get you started.

```
# Create vector with different step sizes
dt = 10**(-np.linspace(0, 4, 50))
# Create vector with the same size as dt for holding the errors
errors = np.zeros_like(dt)
# Loop over the different step sizes to calculate the errors
for i in range(len(dt)):
    # Approximate the solution with Euler's method
    t, x = euler1d(f, x0, t0, tmax, dt[i])
    errors[i] = ??? # Calculate maximum absolute error

# Plot the errors with logarithmic axes
plt.loglog(dt, errors)
plt.xlabel('Step size')
plt.ylabel('Maximum error')
plt.grid()
```

3. What does the plot tell you? In general, if you were to divide your value of Δt by 10, how much approximately does that reduce the error?
-

Exercise 8.8. Shelby solved a first order ODE $x' = f(t, x)$ using Euler's method with a step size of $dt = 0.1$ on a domain $t \in [0, 3]$. To test her code she used a differential equation where she knew the exact analytic solution and she found the maximum absolute error on the interval to be 0.15. Jackson then solves the exact same differential equation, on the same interval, with the same initial condition using Euler's method and a step size of $dt = 0.01$. What would be your best estimate of Jackson's maximum absolute error?

Theorem 8.1. Euler's method is a first order method for approximating the solution to the differential equation $x' = f(t, x)$. Hence, if the step size h of the partition of the domain were to be divided by some positive constant M then the maximum absolute error between the numerical solution and the exact solution would ???

(Complete the last sentence.)

8.2 Higher-order equations and systems of equations

Exercise 8.9 (Harmonic oscillator). If a mass is hanging from a spring then Newton's second law, $\sum F = ma$, gives us the differential equation $mx'' = F_{\text{restoring}} + F_{\text{damping}}$ where x is the displacement of the mass from equilibrium, m is the mass of the object hanging from the spring, $F_{\text{restoring}}$ is the force pulling the mass back to equilibrium, and F_{damping} is the force due to friction or air resistance that slows the mass down.

1. Which of the following is a good candidate for a restoring force in a spring? Defend your answer.
 - a. $F_{\text{restoring}} = kx$: The restoring force is proportional to the displacement away from equilibrium.
 - b. $F_{\text{restoring}} = kx'$: The restoring force is proportional to the velocity of the mass.
 - c. $F_{\text{restoring}} = kx''$: The restoring force is proportional to the acceleration of the mass.
2. Which of the following is a good candidate for a damping force in a spring? Defend your answer.
 - a. $F_{\text{damping}} = bx$: The damping force is proportional to the displacement away from equilibrium.
 - b. $F_{\text{damping}} = bx'$: The damping force is proportional to the velocity of the mass.
 - c. $F_{\text{damping}} = bx''$: The damping force is proportional to the acceleration of the mass.
3. Put your answers to parts (1) and (2) together and simplify to form a second-order differential equation for position:

$$\underline{\quad}x'' = \underline{\quad}x' + \underline{\quad}x \tag{8.10}$$

4. If we want to solve a second order differential equation numerically we need to convert it to first order differential equations (Euler's method is only designed to deal with first order differential equations, not second order). To do so we can introduce a new variable, x_1 , such that $x_1 = x'$. For the sake of notational consistency we define $x_0 = x$. The result is a system of first-order differential equations.

$$\begin{aligned} x'_0 &= x_1 \\ x'_1 &= \underline{\hspace{10em}} \end{aligned} \tag{8.11}$$

5. The code and Euler's method algorithm that we have created thus far in this chapter are only designed to work with a single differential equation instead of a system, so we need to make some modifications. We can discretize the system of differential equations using Euler's method so that

$$x' = F(t, x) \tag{8.12}$$

where F is a function that accepts a vector of inputs, plus time, and returns a vector of outputs. In the context of this particular problem,

$$F(t, x) = \begin{pmatrix} x'_0 \\ x'_1 \end{pmatrix} = \begin{pmatrix} \underline{\hspace{10em}} \\ x_1 \end{pmatrix} \tag{8.13}$$

6. We now need to discretize the derivatives in the system. As with 1D Euler's method, we will use a first-order approximation of the first derivative so that

$$\frac{x_{n+1} - x_n}{h} = F(t_n, x_n) + \mathcal{O}(h). \tag{8.14}$$

Rearranging and solving for x_{n+1} gives

$$x_{n+1} = \underline{\hspace{10em}} + hF(\underline{\hspace{10em}}, \underline{\hspace{10em}}). \tag{8.15}$$

7. The following Python function will implement Euler's method. Complete the code.

```
def euler(F, x0, t0, tmax, dt):
    """
    Solves a system of first-order ordinary differential equations
    using the Euler method.

    Parameters:
        F : function, the function defining the system of differential equations.
            It should take two arguments, the independent variable t and the
            dependent variable x (as a 1D numpy array), and return the
            derivative of x with respect to t (as a 1D numpy array).
        x0 : numpy vector, the initial values of the dependent variables.
    """

    # Your code here
```

```

t0 : float, the initial value of the independent variable.
tmax : float, the maximum value of the independent variable.
dt : float, the time step.

Returns:
tuple containing two numpy arrays:
- t : vector of time values.
- x : array of solutions at each time value. Each column of x
      corresponds to a different dependent variable.

"""
# set up the grid with points from t0 to tmax with stepsize dt
t = ???
# set up array x with one row for each dependent variable and one column
# for each grid point
x = np.zeros((len(t), len(x0)))
# store the initial condition in the first row of x
x[0, :] = x0
# loop over the different time steps
for n in range(???):
    x[n+1, :] = x[???, ???] + dt * F(t[???, ???], x[???, ???])
return t, x

```

8. To use the `euler()` function to solve the system of equations from part (4), complete the following code. Use a mass of $m = 2\text{kg}$, a damping force of $b = 40\text{kg/s}$, and a spring constant of $k = 128\text{N/m}$. Consider an initial position of $x = 0$ (equilibrium) and an initial velocity of $x_1 = 0.6\text{m/s}$.

```

m = 2
b = 40
k = 128
F = lambda t, x: np.array([x[1], ???])
x0 = [???, ???] # initial conditions
t0 = 0
tmax = 5 # pick something reasonable here
dt = 0.01 # your choice. pick something small
t, x = euler(F, x0, t0, tmax, dt)

```

9. Complete the following code to make a plot that shows both position and velocity versus time.

```

plt.plot(t, x[???, ???], 'b-', t, x[???, ???], 'r--')
plt.grid()
plt.title('Time Evolution of Position and Velocity')
plt.legend(['which legend entry here','which legend entry here'])
plt.xlabel('time')
plt.ylabel('position and velocity')
plt.show()

```

10. Complete the following code to make a second plot, called a phase plot, that shows position versus velocity. In a phase plot, time is implicit (not one of the axes).

```

plt.plot(x[???, ???], x[???, ???])
plt.grid()
plt.title('Phase Plot')
plt.xlabel('????')
plt.ylabel('????')
plt.show()

```

Exercise 8.10 (A Lotka-Volterra Predator-Prey Model). Let $x_0(t)$ denote the number of rabbits (prey) and $x_1(t)$ denote the number of foxes (predator) at time t . The relationship between the species can be modelled by the classic 1920's Lotka-Volterra Model:

$$\begin{cases} x'_0 &= \alpha x_0 - \beta x_0 x_1 \\ x'_1 &= \delta x_0 x_1 - \gamma x_1 \end{cases} \quad (8.16)$$

where α, β, γ , and δ are positive constants. For this problems take $\alpha = 1$, $\beta \approx 0.1$, $\gamma = 1$, and $\delta = 0.1$.

1. First rewrite the system of ODEs in the form $x' = F(t, x)$ so you can use your `euler()` code.
2. Modify your code from the previous problem so that it works for this problem. Use `tmax = 20` and `dt = 0.05`. Start with initial conditions $x_0(0) = 10$ rabbits and $x_1(0) = 5$ foxes.
3. Create the time evolution plot. What does this plot tell you in context?
4. Create a phase plot. What does this plot tell you in context?
5. If you decrease the step size by a factor of 10, what do you see in the two plots? Why?

Exercise 8.11 (The SIR Model). A classic model for predicting the spread of a virus or a disease is the SIR Model. In these models, S stands for the proportion of the population which is susceptible to the virus, I is the proportion of the population that is currently infected with the virus, and R is the proportion of the population that has recovered from the virus. The idea behind the model is that

- Susceptible people become infected by having interaction with the infected people. Hence, the rate of change of the susceptible people is proportional to the number of interactions that can occur between the S and the I populations.

$$S' = -\beta SI. \quad (8.17)$$

- The infected population gains people from the interactions with the susceptible people, but at the same time, infected people recover at a predictable rate.

$$I' = \beta SI - \gamma I. \quad (8.18)$$

- The people in the recovered class are then immune to the virus, so the recovered class R only gains people from the recoveries from the I class.

$$R' = \gamma I. \quad (8.19)$$

Find a numerical solution to the system of equations using your `euler()` function. Use the parameters $\beta = 0.0003$ and $\gamma = 0.1$ with initial conditions $S(0) = 999$, $I(0) = 1$, and $R(0) = 0$. Plot the solution against time. Explain all three curves in context.

8.3 The Midpoint Method

Now we get to improve upon Euler's method. There is a long history of wonderful improvements to the classic Euler's method – some that work in special cases, some that resolve areas where the error is going to be high, and some that are great for general purpose numerical solutions to ODEs with relatively high accuracy. In this section we will make a simple modification to Euler's method that has a surprisingly great pay-off in the error rate.

Exercise 8.12. In Euler's method, if we are at the point t_n then we approximate the slope $x'(t_n) = f(t_n, x_n)$ and use the slope to propagate forward one time step. As you have seen, this method can lead to an overshooting of the exact solution in regions of high curvature. It would be nice to be able to look into the future and get a better approximation of the slope so that we did not miss upcoming curvature. If you could build such a method that looks in to the future, finds a slope in the future, and then uses that slope (instead of the slope from Euler's method) to advance forward in time, how far into the future would you look? Why?

Exercise 8.13. Let us return to the simple differential equation $x' = -0.5x$ with $x(0) = 6$ that we saw in Exercise 8.1. Now we will propose a slightly different method for approximating the solution.

- a. At $t = 0$ we know that $x(0) = 6$. If we use the slope at time $t = 0$ to step forward in time then we will get the Euler approximation of the solution. Consider this alternative approach:

- Use the slope at time $t = 0$ and move *half* a step forward.
- Find the slope at the half-way point
- Then use the slope from the half way point to go a full step forward from time $t = 0$.

Perhaps a bit confusing ...let us build this idea together:

- What is the slope at time $t = 0$? $x'(0) = \underline{\hspace{2cm}}$
 - Use this slope to step a half step forward and find the x value: $x(0.5) \approx \underline{\hspace{2cm}}$
 - Now use the differential equation to find the slope at time $t = 0.5$. $x'(0.5) = \underline{\hspace{2cm}}$
 - Now take your answer from the previous step, and go one full step forward from time $t = 0$. What x value do you end up with?
 - Your answers to the previous bullets should be: $x'(0) = -3$, $x(0.5) \approx 4.5$, $x'(0.5) = -2.25$, so if we take a full step forward with slope $m = -2.25$ starting from $t = 0$ we get $x(1) \approx 3.75$.
- b. Repeat the process outlined in part (a) to approximate the solution to the differential equation at times $t = 2, 3, 4$. Also record the exact answer at each of these times by noting that the exact solution is $x(t) = 6e^{-0.5t}$.

t	0	1	2	3	4
Euler approx of $x(t)$	6				
New approx of $x(t)$	6				
Exact value of $x(t)$	6				

- c. Draw a clear picture of what this method is doing in order to approximate the slope at each individual step.
- d. How does your approximation compare to the Euler approximation that you found in Exercise 8.1?
-

Definition 8.2 (The Midpoint Method). The midpoint method is defined by first taking a half step with Euler's method to approximate a solution at time $t_{n+1/2}$. There is no grid point at $t_{n+1/2}$ so we define this as

$$t_{n+1/2} = (t_n + t_{n+1})/2.$$

We then take a full step using the value of f at $t_{n+1/2}$ and the approximate $x_{n+1/2}$.

$$\begin{aligned}x_{n+1/2} &= x_n + \frac{h}{2} f(t_n, x_n) \\x_{n+1} &= x_n + h f(t_{n+1/2}, x_{n+1/2})\end{aligned}$$

Exercise 8.14. As in in Exercise 8.4, consider the differential equation $x'(t) = -2x(t)/3 + 4t$ with initial condition $x(0) = 6$. By hand perform one step of the midpoint method with stepsize $h = 1$ to obtain an approximation for $x(1)$.

Exercise 8.15. Complete the code below to implement the midpoint method in one dimension.

```
def midpoint1d(f, x0, t0, tmax, dt):
    """
    Solves a first-order ordinary differential equation using
    the midpoint method.

    Parameters:
        f      : function, the function defining the differential equation. It should
    
```

```

        take two arguments, the independent variable t and the dependent
        variable x, and return the derivative of x with respect to t.
x0    : float, the initial value of the dependent variable.
t0    : float, the initial value of the independent variable.
tmax : float, the maximum value of the independent variable.
dt    : float, the time step.

>Returns:
tuple containing two numpy arrays:
- t : vector of time values.
- x : vector of solution values at each time value.

"""
t = ??? # build the times
x = ??? # build an array for the x values
x[0] = # save the initial condition
# On the next line: be careful about how far you're looping
for n in range(???):
    slope = ??? # get the slope at the current point
    x_halfstep = ??? # take a half step forward
    x[n+1] = ??? # take a full step forward
return t, x

```

Test your code on several differential equations where you know the solution (just to be sure that it is working).

```

f = lambda t, x: # your ODE right hand side goes here
x0 = # initial condition
t0 = 0
tmax = # ending time (up to you)
dt = # pick something small
t, x = midpoint1d(???, ???, ???, ???, ???)
plt.plot(???, ???, ???)
x_exact = lambda t: # your exact solution goes here
plt.plot(???, ???, ???)
plt.legend(['Midpoint', 'Exact'])
plt.grid()
plt.show()

```

Exercise 8.16. The goal in building the midpoint method was to hopefully capture some of the upcoming curvature in the solution before we overshot it. Consider the differential

equation $x' = -\frac{1}{3}x + \sin(t)$ with initial condition $x(0) = 1$ on the domain $t \in [0, 10]$ as in Exercise 8.6. First get a numerical solution with Euler's method using $\Delta t = 1$. Then get a numerical solution with the midpoint method using the same value for $\Delta t = 1$. Plot the two solutions on top of each other along with the exact solution

$$x(t) = \frac{1}{10} (19e^{-t/3} + 3 \sin(t) - 9 \cos(t)). \quad (8.20)$$

What do you observe?

Exercise 8.17. Repeat Exercise 8.7 with the midpoint method. Compare your results to what you found with Euler's method.

Exercise 8.18. We have studied two methods thus far: Euler's method and the Midpoint method. In Figure 8.5 we see a graphical depiction of how each method works on the differential equation $y' = y$ with $\Delta t = 1$ and $y(0) = 1$. The exact solution at $t = 1$ is $y(1) = e^1 \approx 2.718$ and is shown in red in each figure. The methods can be summarized in the table below.

Discuss what you observe as the pros and cons of each method based on the table and on the Figure.

Euler's Method	Midpoint Method
1. Get the slope at time t_n	1. Get the slope at time t_n
2. Follow the slope for time Δt	2. Follow the slope for time $\Delta t/2$ 3. Get the slope at the point $t_n + \Delta t/2$ 4. Follow the new slope from time t_n for time Δt

When might you want to use Euler's method instead of the midpoint method? When might you want to use the midpoint method instead of Euler's method?

Exercise 8.19 (Midpoint Method in Several Dimensions). Write a function `midpoint()` that can solve a system of first-order ordinary differential equations using the midpoint method. Base your code on the `euler()` code from Exercise 8.9. You should only have to add one line of code and then be careful about the size of the arrays that are in play. Test your code on several problems. Compare and contrast what you see with your Euler solutions and with your Midpoint solutions.

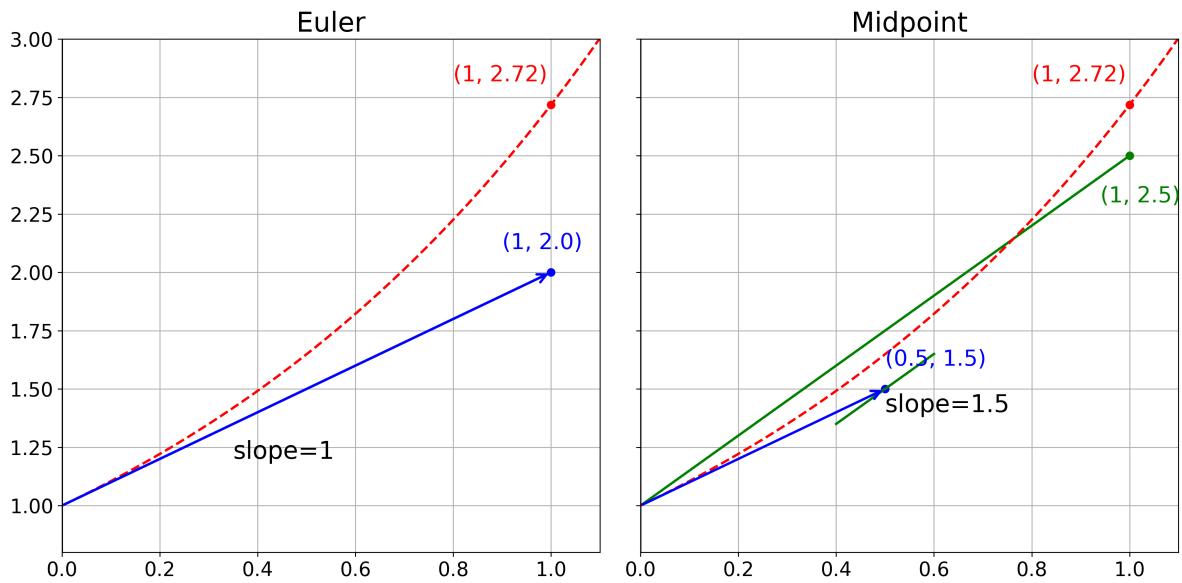


Figure 8.4: Graphical depictions of two numerical methods. Euler (left) and Midpoint (right).
The exact solution is shown in red.

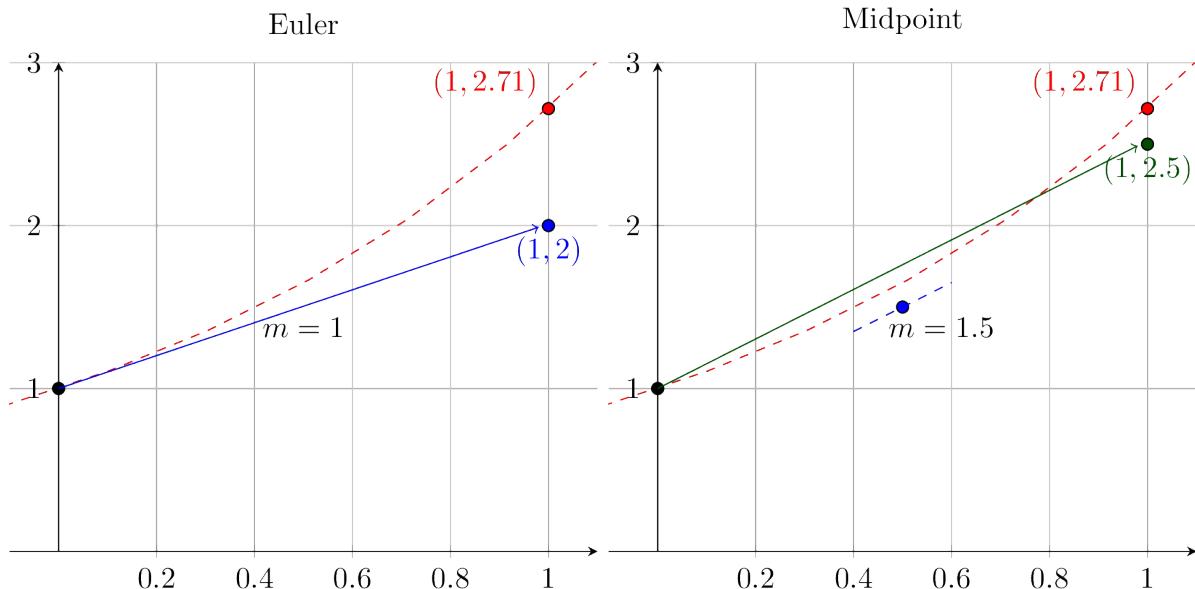


Figure 8.5: Graphical depictions of two numerical methods: Euler (left) and Midpoint (right).
The exact solution is shown in red.

8.4 Searching for a better Method

OK. Ready for some experimentation? We are going to build a few experiments that eventually lead us to a very powerful method for finding numerical solutions to first order differential equations than the midpoint method.

This section is optional. If you feel pressed for time, then you can skip forward to Section 8.5.

Exercise 8.20. Let us talk about the Midpoint Method for a moment. The geometric idea of the midpoint method is outlined in the bullets below. Draw a picture along with the bullets.

- You are sitting at the point (t_n, x_n) .
- The slope of the solution curve to the ODE where you are standing is

$$\text{slope at the point } (t_n, x_n) \text{ is: } m_n = f(t_n, x_n) \quad (8.21)$$

- You take a half a step forward using the slope where you are standing. The new point, denoted $x_{n+1/2}$, is given by

$$\text{location a half step forward is: } x_{n+1/2} = x_n + \frac{\Delta t}{2} m_n. \quad (8.22)$$

- Now you are standing at $(t_n + \frac{\Delta t}{2}, x_{n+1/2})$ so there is a new slope here given by
$$\text{slope after a half of an Euler step is: } m_{n+1/2} = f(t_n + \Delta t/2, x_{n+1/2}). \quad (8.23)$$
- Go back to the point (t_n, x_n) and step a full step forward using slope $m_{n+1/2}$. Hence the new approximation is

$$x_{n+1} = x_n + \Delta t \cdot m_{n+1/2} \quad (8.24)$$

Exercise 8.21. One of the troubles with the midpoint method is that it does not actually use the information at the point (t_n, x_n) . Moreover, it does not leverage a slope at the next time step t_{n+1} . Let us see what happens when we try a solution technique that combined the ideas of Euler and Midpoint as follows:

- The slope at the point (t_n, x_n) can be called m_n and we find it by evaluating $f(t_n, x_n)$.
- The slope at the point $(t_{n+1/2}, x_{n+1/2})$ can be called $m_{n+1/2}$ and we find it by evaluating $f(t_{n+1/2}, x_{n+1/2})$.
- We can now take a full step using slope $m_{n+1/2}$ to get the point x_{n+1} and the slope there is $m_{n+1} = f(t_{n+1}, x_{n+1})$.

- Now we have three estimates of the slope that we can use to actually propagate forward from (t_n, x_n) :
 - We could just use m_n . This is Euler's method.
 - We could just use $m_{n+1/2}$. This is the midpoint method.
 - We could use m_{n+1} . Would this approach be any good?
 - We could use the average of the three slopes.
 - We could use a weighted average of the three slopes where some preference is given to some slopes over the others.

In the code below you will find a function called `ode_test()` that you can use as a starting point to test out the last three ideas. After the function you will see several lines of code that test your method against the differential equation $x'(t) = -\frac{1}{3}x + \sin(t)$ with $x(0) = 1$. The plots that come out are our typical error plots with the step size on the horizontal axis and our maximum absolute error between the numerical solution and the exact solution on the vertical axis. Recall that the exact solution to this differential equation is

$$x(t) = \frac{1}{10} (19e^{-t/3} + 3 \sin(t) - 9 \cos(t)) \quad (8.25)$$

```
import numpy as np
import matplotlib.pyplot as plt

# *****
# You should copy your 1d euler and midpoint functions here.
# We will be comparing to these two existing methods.
# *****

def ode_test(f, x0, t0, tmax, dt):
    t = np.arange(t0, tmax+dt, dt) # set up the times
    x = np.zeros(len(t)) # set up the x
    x[0] = x0 # initial condition
    for n in range(len(t)-1):
        m_n = f(t[n], x[n])
        x_n_plus_half = x[n] + dt/2 * m_n
        m_n_plus_half = f(t[n] + dt/2, x_n_plus_half)
        x_n_plus_1 = x[n] + dt * m_n_plus_half
        m_n_plus_1 = f(t[n] + dt, x_n_plus_1)
        estimate_of_slope = # This is where you get to play
        x[n+1] = x[n] + dt * estimate_of_slope
    return t, x
```

```

f = lambda t, x: -(1/3.0) * x + np.sin(t)
exact = lambda t: (1/10.0)*(19*np.exp(-t/3) + \
                     3*np.sin(t) - \
                     9*np.cos(t))

x0 = 1 # initial condition
t0 = 0 # initial time
tmax = 3 # max time
# set up blank arrays to keep track of the maximum absolute errorrs
err_euler = []
err_midpoint = []
err_ode_test = []
# Next give a list of Delta t values (what list did we give here)
H = 10.0**(-np.arange(1, 7, 1))
for dt in H:
    # Build an euler approximation
    t, xeuler = euler(f, x0, t0, tmax,dt)
    # Measure the max abs error
    err_euler.append(np.max(np.abs(xeuler - exact(t))))
    # Build a midpoint approximation
    t, xmidpoint = midpoint(f, x0, t0, tmax, dt)
    # Measure the max abs error
    err_midpoint.append(np.max(np.abs(xmidpoint - exact(t))))
    # Build your new approximation
    t, xtest = ode_test(f, x0, t0, tmax, dt)
    # Measure the max abs error
    err_ode_test.append(np.max(np.abs(xtest - exact(t)))))

# Finally, we make a loglog plot of the errors.
# Keep an eye on the slopes since they tell you the order of
# the error for the method.
plt.loglog(H, err_euler, 'r*-',
            H, err_midpoint, 'b*-',
            H, err_ode_test, 'k*-')
plt.grid()
plt.legend(['euler', 'midpoint', 'test method'])
plt.show()

```

Exercise 8.22. In the previous exercise you should have found that an average of the three

slopes did just a *little bit* better than the midpoint method but the order of the error (the slope in the log-log plot) stayed about the same. You should have also found that the weighted average

$$\text{estimate of slope} = \frac{m_n + 2m_{n+1/2} + m_{n+1}}{4} \quad (8.26)$$

did just a little bit better than just a plain average. Why might this be? (If you have not tried this weighted average then go back and try it.) Do other weighted averages of this sort work better or worse? Does it appear that we can improve upon the order of the error (the slope in the log-log plot) using any of these methods?

Exercise 8.23. OK. Let us make one more modification. What if we built a fourth slope that resulted from stepping a half step forward using $m_{n+1/2}$? We will call this $m_{n+1/2}^*$ since it is a new estimate of $m_{n+1/2}$.

$$x_{n+1/2}^* = x_n + \frac{\Delta t}{2} m_{n+1/2} \quad (8.27)$$

$$m_{n+1/2}^* = f(t_n + \Delta t/2, x_{n+1/2}^*) \quad (8.28)$$

Then calculate a new estimate m_{n+1}^* of the slope at the end of the step using this new slope:

$$x_{n+1}^* = x_n + \Delta t m_{n+1/2}^* \quad (8.29)$$

$$m_{n+1}^* = f(t_n + \Delta t, x_{n+1}^*) \quad (8.30)$$

1. Draw a picture showing where this slope was calculated.
2. Modify the code from above to include this fourth slope.
3. Experiment with several ideas about how to best combine the four slopes: m_n , $m_{n+1/2}$, $m_{n+1/2}^*$, and m_{n+1}^* .
 - Should we just take an average of the four slopes?
 - Should we give one or more of the slopes preferential treatment and do some sort of weighted average?
 - Should we do something else entirely?

Remember that we are looking to improve the slope in the log-log plot since that indicates an improvement in the order of the error (the accuracy) of the method.

Exercise 8.24. In the previous exercise you no doubt experimented with many different linear combinations of m_n , $m_{n+1/2}$, $m_{n+1/2}^*$, and m_{n+1}^* . Many of the resulting numerical ODE methods likely had the same order of accuracy (again, the order of the method is the slope in the error plot), but some may have been much better or much worse. Work with your team to fill in the following summary table of all of the methods that you devised. If you generated linear combinations that are not listed below then just add them to the list (we have only listed the most common ones here).

	m_n	$m_{n+1/2}$	$m_{n+1/2}^*$	m_{n+1}^*	Order of Error	Name
1	1	0	0	0	$\mathcal{O}(\Delta t)$	Euler's Method
2	0	1	0	0	$\mathcal{O}(\Delta t^2)$	Midpoint Method
3	1/2	1/2	0	0		
4	1/3	1/3	0	1/3		
5	1/4	2/4	0	1/4		
6	0	0	1	0		
7	0	1/2	1/2	0		
8	1/3	1/3	1/3	0		
9	1/4	1/4	1/4	1/4		
10	1/5	2/5	1/5	1/5		
11	1/5	1/5	2/5	1/5		
12	1/6	2/6	2/6	1/6		
13	1/6	3/6	1/6	1/6		
14	1/6	1/6	3/6	1/6		
15	1/7	2/7	3/7	1/7		
16	1/8	3/8	3/8	1/8		
17						
18						

Exercise 8.25. In the previous exercise you should have found at least one of the many methods to be far superior to the others. State which linear combination of slopes seems to have done the trick, draw a picture of what this method does to numerically approximate the next slope for a numerical solution to an ODE, and clearly state what the order of the error means about this method.

You probably discovered that the best method approximates the slope at the point t_n by using the weighted sum

$$\text{estimated slope} = \frac{m_n + 2m_{n+1/2} + 2m_{n+1/2}^* + m_{n+1}^*}{6}. \quad (8.31)$$

If you have not already done so, you should try this method out in the code from Exercise 8.23 and see how it compares to the other methods that you have tried. You should find that the slope in the log-log error plot for this method is 4, which means this is a fourth-order method. That is a huge improvement over the midpoint method. This method is called the Runge-Kutta 4 method and is one of the most commonly used method for solving ordinary differential equations. We present this method in different notation in Definition 8.3. Please make sure you see why that theorem defines the same method that you have just discovered.

8.5 Runge-Kutta Method

Definition 8.3 (The Runge-Kutta 4 Method). The Runge-Kutta 4 (RK4) method for approximating the solution to the differential equation $x' = f(t, x)$ approximates the slope m at the point t_n by using the following calculations:

$$\begin{aligned} k_1 &= f(t_n, x_n) \\ k_2 &= f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1\right) \\ k_3 &= f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2\right) \\ k_4 &= f(t_n + h, x_n + hk_3) \\ m &= \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{8.32}$$

It then uses that slope to advance by one time step:

$$x_{n+1} = x_n + hm = x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{8.33}$$

The order of the error in the RK4 method is $\mathcal{O}(\Delta t^4)$.

Exercise 8.26. Of course there is a price to pay for the improvement provided by the RK4 method over our earlier methods. How many evaluations of the function $f(t, x)$ do we need to make at every time step of the RK4 method? Compare this Euler's method and the midpoint method.

Exercise 8.27. Let us step back for a second and just see what the RK4 method does from a nuts-and-bolts point of view. Consider the differential equation $x' = x$ with initial condition $x(0) = 1$. The solution to this differential equation is clearly $x(t) = e^t$. For the sake of simplicity, take $\Delta t = 1$ and perform 1 step of the RK4 method BY HAND to approximate the value $x(1)$.

Exercise 8.28. Write a Python function that implements the Runge-Kutta 4 method in one dimension.

```
import numpy as np
import matplotlib.pyplot as plt

def rk41d(f, x0, t0, tmax, dt):
    t = np.arange(t0, tmax+dt, dt)
    x = np.zeros(len(t))
    x[0] = x0
    for n in range(len(t)-1):
        # the interesting bits of the code go here
    return t, x
```

Test the problem on several differential equations where you know the solution.

Exercise 8.29 (RK4 Error plot). Make a log-log plot of the error in the RK4 method for a differential equation whose solution you know exactly (you could use $x' = -\frac{1}{3}x + \sin(t)$ with $x(0) = 1$ on the domain $t \in [0, 10]$, similar to what you did for the Euler method and the midpoint method earlier). Check that the error plot shows that the error is $\mathcal{O}(\Delta t^4)$.

Exercise 8.30 (RK4 in Several Dimensions). Modify your Runge-Kutta 4 code to work for any number of dimensions. You may want to start from your `euler()` and `midpoint()` functions that already do this. You will only need to make minor modifications from there. Then test your new generalized RK4 method on all of the same problems which you used to test your `euler()` and `midpoint()` functions.

8.6 The Backwards Euler Method

We have now built up a variety of numerical ODE solvers. All of the solvers that we have built thus far are called **explicit** numerical differential equation solvers since they try to advance the solution explicitly forward in time. Wouldn't it be nice if we could literally just say, *what slope is going to work best in the future time steps ... let us use that?* Seems like an unrealistic hope, but that is exactly what the last method covered in this section does.

Definition 8.4 (Backward Euler Method). We want to solve $x' = f(t, x)$ so:

- Approximate the derivative by looking forward in time(!)

$$\frac{x_{n+1} - x_n}{h} \approx f(t_{n+1}, x_{n+1}) \quad (8.34)$$

- Rearrange to get the difference equation

$$x_{n+1} = x_n + h f(t_{n+1}, x_{n+1}). \quad (8.35)$$

- We will always know the value of t_{n+1} and we will always know the value of x_n , but we do not know the value of x_{n+1} . In fact, that is exactly what we want. The major trouble is that x_{n+1} shows up on both sides of the equation. Can you think of a way to solve for it? ...you have code that does this step!!!
 - This method is called the **Backward Euler** method and is known as an **implicit method** since it does not explicitly give an expression for x_{n+1} but instead it gives an equation that still needs to be solved for x_{n+1} . We will see the main advantage of implicit methods in Section 8.7.
-

Exercise 8.31. Let us take a few steps through the backward Euler method on a problem that we know well: $x' = -0.5x$ with $x(0) = 6$.

Let us take $h = 1$ for simplicity, so the backward Euler iteration scheme for this particular differential equation is

$$x_{n+1} = x_n - \frac{1}{2}x_{n+1}. \quad (8.36)$$

Notice that x_{n+1} shows up on both sides of the equation. A little bit of rearranging gives

$$\frac{3}{2}x_{n+1} = x_n \implies x_{n+1} = \frac{2}{3}x_n. \quad (8.37)$$

1. Complete the following table.

t	0	1	2	3	4	5	6
Euler Approx. of x	6	3	1.5	0.75			
Back. Euler Approx. of x	6	4	2.667	1.778			
Exact value of x	6	3.64	2.207	1.339			

2. Compare now to what we found for the midpoint method on this problem as well.
-

Exercise 8.32. By hand, apply the backwards Euler method with stepsize $h = 1/2$ to the differential equation $x' = xt$ with initial condition $x(0) = 1$ to obtain an approximation for $x(1/2)$. Do your calculations in terms of fractions.

Exercise 8.33. The previous problem could potentially lead you to believe that the backward Euler method will always result in some other nice difference equation after some algebraic rearranging. That is not true! Let us consider a slightly more complicated differential equation and see what happens

$$x' = -\frac{1}{2}x^2 \quad \text{with} \quad x(0) = 0. \quad (8.38)$$

1. Recall that the backward Euler approximation is

$$x_{n+1} = x_n + hf(t_{n+1}, x_{n+1}). \quad (8.39)$$

Let us take $h = 1$ for simplicity (we will make it smaller later). What is the backward Euler formula for this particular differential equation?

2. You should notice that your backward Euler formula is now a quadratic function in x_{n+1} . That is to say, if you are given a value of x_n then you need to solve a quadratic polynomial equation to get x_{n+1} . Let us be more explicit:

We know that $x(0) = 6$ so in our numerical solutions, $x_0 = 6$. In order to get x_1 we consider the equation

$$x_1 = x_0 - \frac{1}{2}x_1^2.$$

Rearranging we see that we need to solve

$$\frac{1}{2}x_1^2 + x_1 - 6 = 0$$

in order to get x_1 . Doing so gives us $x_1 = \sqrt{13} - 1 \approx 2.606$.

3. Go two steps further with the backward Euler method on this problem. Then take the same number of steps with regular (forward) Euler's method.
 4. Work out the analytic solution for this differential equation (using separation of variables perhaps). Then compare the values that you found in parts (2) and (3) of this problem to values of the analytic solution and values that you would find from the regular (forward) Euler approximation. What do you notice?
-

The complications with the backward Euler's method are that you have a nonlinear equation to solve at every time step

$$x_{n+1} = x_n + hf(t_{n+1}, x_{n+1}). \quad (8.40)$$

Notice that this is the same as solving the equation

$$G(x_{n+1}) := x_{n+1} - hf(t_{n+1}, x_{n+1}) - x_n = 0. \quad (8.41)$$

You know the values of $h = \Delta t$, t_{n+1} and x_n , and you know the function f , so, in a practical sense, you should use some sort of Newton's method iteration to solve that equation – at each time step. More simply, we could call upon `scipy.optimize.fsolve()` to quickly implement a built in Python numerical root finding technique for us. For example, to find the root of the function $G(x) = x^2/2 + x - 6$ we could use the following code

```
import numpy as np
from scipy import optimize
G = lambda x: x**2/2 + x - 6
x0 = 6 # initial guess
x = optimize.fsolve(G, x0)[0]
x

np.float64(2.6055512754639896)
```

Exercise 8.34. Complete the function `backwardEuler1d()` below. How do you define the function `G` inside the `for` loop and what starting value do you use for the `fsolve()` command?

```

import numpy as np
from scipy import optimize
def backwardEuler1d(f, x0, t0, tmax, dt):
    t = np.arange(t0, tmax + dt, dt)
    x = np.zeros(len(t))
    x[0] = x0
    for n in range(len(t)-1):
        G = lambda X: ??? # define this function
        # give an appropriate starting value for fsolve
        x[n+1] = optimize.fsolve(G, ???)[0]
    return t, x

```

Test your `backwardEuler1d()` function on several differential equations where you know the solution.

Exercise 8.35. Write a script that outputs a log-log plot with the step size on the horizontal axis and the error in the numerical method on the vertical axis. Plot the errors for Euler, Midpoint, Runge Kutta, and Backward Euler measured against a differential equation with a known analytic solution. Use this plot to conjecture the convergence rates of the four methods. You can use the differential equation $x' = -\frac{1}{3}x + \sin(t)$ with $x(0) = 1$ like we have for many of our past algorithm since we know that the solution is

$$x(t) = \frac{1}{10} (19e^{-t/3} + 3 \sin(t) - 9 \cos(t)). \quad (8.42)$$

From the plot, what is the order of the error on the Backward Euler method?

8.7 Numerical Instabilities

Exercise 8.36. It may not be obvious at the outset, but the Backward Euler method will actually behave better than our regular Euler's method in some sense. Let us take a look. Consider, for example, the really simple linear differential equation $x' = -10x$ with $x(0) = 1$ on the interval $t \in [0, 2]$. The analytic solution is $x(t) = e^{-10t}$. Write Python code that plots the analytic solution, the Euler approximation, and the Backward Euler approximation on top of each other. Use a time step that is larger than you normally would (such as $\Delta t = 0.25$ or $\Delta t = 0.5$ or larger). What do you notice? Why do you think this is happening?

Exercise 8.37. Consider the linear differential equation $x' = -cx$ with $c > 0$ a constant. If you solve this with the forward Euler's method with a step size h , then each step can be written in the form

$$x_{n+1} = \text{???} \cdot x_n.$$

Because the exact solution to this differential equation $x(t) = e^{-ct}$ goes towards 0 as t goes to infinity, Euler's method should also go towards 0 as n goes to infinity. What is the condition on the ??? in the equation above that ensures that in Euler's method x_n will go towards 0 as n goes to infinity? What condition does this impose on the step size h ?

If you now solve this same ODE with the same step size with the backward Euler method, you can solve the equation for x_{n+1} exactly to find that

$$x_{n+1} = \text{???} \cdot x_n.$$

Again you would like this to go towards 0 as n goes to infinity. What condition does this impose on the step size h now?

Because the observations you made in the previous two exercises are so important, I lectured about them. Here are the lecture notes:

The stability of a numerical method is the property that the numerical solution does not diverge from the exact solution as the step size goes to zero. In other words, if we take smaller and smaller steps, the numerical solution should converge to the exact solution. If the numerical solution diverges from the exact solution as the step size goes to zero, then we say that the method is unstable.

To test the stability of a method we can apply it to a simple ODE with a known solution. We can then compare the numerical solution to the exact solution. If the numerical solution is stable, then it will converge to the exact solution as the step size goes to zero. If the numerical solution is unstable, then it will diverge from the exact solution as the step size goes to zero.

The test equation that is always used for this purpose of studying the stability of a numerical method is the linear ODE

$$x' = \lambda x.$$

The exact solution to this equation is $x(t) = x(0)_0 e^{\lambda t}$. So for negative λ the solution decays, and for positive λ the solution grows. The stability of a numerical method can be tested by applying it to this equation and comparing the numerical solution to the exact solution.

Example 8.1 (Stability of Euler's method). Euler's method uses the formula $x_{n+1} = x_n + hf(t_n, x_n)$. Applying this to the test equation $x' = \lambda x$ gives

$$x_{n+1} = x_n + h\lambda x_n = (1 + h\lambda)x_n.$$

So

$$x_{n+1} = (1 + h\lambda)x_n = (1 + h\lambda)^2 x_{n-1} = \dots = (1 + h\lambda)^{n+1} x_0.$$

Because we know the exact solution, we can calculate the absolute error that Euler's method produces:

$$E_n := |x_n - x(t_n)| = |x_0(1 + h\lambda)^n - x_0 e^{\lambda t_n}|.$$

In the case where $\lambda < 0$ we have that the exact solution decays to zero as $n \rightarrow \infty$. So the error will be determined by the behaviour of the first term:

$$\begin{aligned} \lim_{n \rightarrow \infty} E_n &= \lim_{n \rightarrow \infty} |x_0(1 + h\lambda)^n| \\ &= \begin{cases} 0 & \text{if } |1 + h\lambda| < 1 \\ \infty & \text{if } |1 + h\lambda| > 1. \end{cases} \end{aligned}$$

The error grows exponentially unless $|1 + h\lambda| < 1$. This is the stability condition for Euler's method. It requires us to choose a step size

$$h < \frac{2}{|\lambda|}$$

In general, for a method that, when applied to the test equation $x' = \lambda x$, gives

$$x_{n+1} = Q(h\lambda)x_n$$

the stability condition is that

$$|Q(h\lambda)| < 1.$$

Values of $h\lambda =: z$ for which $|Q(z)| < 1$ form the *Region of absolute stability** of the method. This is a region in the complex plane, because while the stepsize h is of course always real, in applications the transient term may be oscillatory, corresponding to a complex λ .

Example 8.2 (Stability of the midpoint method). The midpoint method uses the formula

$$x_{n+1} = x_n + hf\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}f(t_n, x_n)\right).$$

Applying this to the test equation $x' = \lambda x$ gives

$$x_{n+1} = x_n + h\lambda \left(x_n + \frac{h}{2}\lambda x_n\right) = x_n \left(1 + h\lambda + \frac{(h\lambda)^2}{2}\right).$$

The region of absolute stability is the set of $z \in \mathbb{C}$ for which

$$\left|1 + z + \frac{z^2}{2}\right| < 1.$$

Implicit methods tend to have a larger region of absolute stability, often including the entire left half plane, in which case they are called “unconditionally stable”.

Example 8.3 (Stability of the backward Euler method). We demonstrate this in the case of the backward Euler method, which uses the formula

$$x_{n+1} = x_n + h f(t_{n+1}, x_{n+1}).$$

Applying this to the test equation $x' = \lambda x$ gives

$$x_{n+1} = x_n + h \lambda x_{n+1}.$$

So

$$x_{n+1} = \frac{x_n}{1 - h\lambda}.$$

The region of absolute stability is the set of $z \in \mathbb{C}$ for which

$$\left|\frac{1}{1-z}\right| < 1,$$

or, equivalently,

$$|z - 1| > 1.$$

This is almost the entire complex plane, except for the circle of radius 1 around the point $z = 1$. In particular it contains the entire left half plane, so stiff equations do not require any particular choice of step size for the backward Euler method to be stable. We therefore say that the backward Euler method is unconditionally stable.

8.8 Stiff Equations

A differential equation is called “stiff” if the exact solution has a term that decays very rapidly. This is a problem for numerical methods because the step size must be very small to capture the rapid decay. The step size must be small for the entire solution, not just the rapidly decaying part. This can make the solution very slow to compute.

An example of such a rapidly decaying term would be a term of the form $e^{-\gamma t}$ with γ large. As an example consider the ODE

$$x' = -10x + \sin t.$$

This is similar to equations we solved in Exercise 8.6, just with a larger negative coefficient in front of x . The solution to this equation is

$$x(t) = Ce^{-100t} + \frac{100}{10001} \sin t - \frac{1}{10001} \cos t.$$

The term e^{-100t} decays very rapidly. We refer to such a term in the solution as a “transient” because it becomes irrelevant after a short time. In spite of the fact that this term is irrelevant, it forces us to take a very small step size in order to avoid an instability if we are using an explicit method. Thus for a stiff equation it is advisable to use an implicit method instead.

8.9 Algorithm Summaries

Exercise 8.38. Consider the first-order differential equation $x' = f(t, x)$. What is Euler’s method for approximating the solution to this differential equation? What is the order of accuracy of Euler’s method? Explain the meaning of the order of the method in the context of solving a differential equation.

Exercise 8.39. Explain in clear language what Euler’s method does geometrically.

Exercise 8.40. Consider the first-order differential equation $x' = f(t, x)$. What is the Midpoint method for approximating the solution to this differential equation? What is the order of accuracy of the Midpoint method?

Exercise 8.41. Explain in clear language what the Midpoint method does geometrically.

Exercise 8.42. Consider the first-order differential equation $x' = f(t, x)$. What is the Runge Kutta 4 method for approximating the solution to this differential equation? What is the order of accuracy of the Runge Kutta 4 method?

Exercise 8.43. Explain in clear language what the Runge Kutta 4 method does geometrically.

Exercise 8.44. Consider the first-order differential equation $x' = f(t, x)$. What is the Backward Euler method for approximating the solution to this differential equation? What is the order of accuracy of the Backward Euler method?

Exercise 8.45. Explain in clear language what the Backward Euler method does geometrically.

8.10 Problems

Exercise 8.46. Consider the differential equation $x'' + x' + x = 0$ with initial conditions $x(0) = 0$ and $x'(0) = 1$.

1. Solve this differential equation by hand using any appropriate technique. Show your work.
 2. Write code to demonstrate the first order convergence rate of Euler's method, the second order convergence rate of the Midpoint method, and the fourth order convergence rate of the Runge-Kutta 4 method. Take note that this is a second order differential equation so you will need to start by converting it to a system of differential equations. Then take care that you are comparing the correct term from the numerical solution to your analytic solution in part (1).
-

Exercise 8.47. Test the Euler, Midpoint, and Runge Kutta methods on the differential equation

$$x' = \lambda(x - \cos(t)) - \sin(t) \quad \text{with} \quad x(0) = 1.5. \quad (8.43)$$

Find the exact solution by hand using the method of undetermined coefficients and note that your exact solution will involve the parameter λ . Produce log-log plots for the error between your numerical solution and the exact solution for $\lambda = -1, \lambda = -10, \lambda = -10^2, \dots, \lambda = -10^6$. In other words, create 7 plots (one for each λ) showing how each of the 3 methods performs for that value of λ at different values for Δt .

Exercise 8.48. Two versions of Python code for one dimensional Euler's method are given below. Compare and contrast the two implementations. What are the advantages / disadvantages to one over the other? Once you have made your pro/con list, devise an experiment to see which of the methods will actually perform faster when solving a differential equation with a very small Δt . (You may want to look up how to time the execution of code in Python.)

```
def euler(f,x0,t0,tmax,dt):
    t = [t0]
    x = [x0]
    steps = int(np.floor((tmax-t0)/dt))
    for n in range(steps):
        t.append(t[n] + dt)
        x.append(x[n] + dt*f(t[n],x[n]))
    return t, x
```

```
def euler(f,x0,t0,tmax,dt):
    t = np.arange(t0,tmax+dt,dt)
    x = np.zeros(len(t))
    x[0] = x0
    for n in range(len(t)-1):
        x[n+1] = x[n] + dt*f(t[n],x[n])
    return t, x
```

Exercise 8.49. We wish to solve the boundary valued problem $x'' + 4x = \sin(t)$ with initial condition $x(0) = 1$ and boundary condition $x(1) = 2$ on the domain $t \in (0,1)$. Notice that you do not have the initial position and initial velocity as you normally would with a second order differential equation. Devise a method for finding a numerical solution to this problem.

Exercise 8.50. Write code to numerically solve the boundary valued differential equation

$$x'' = \cos(t)x' + \sin(t)x \quad \text{with} \quad x(0) = 0 \quad \text{and} \quad x(1) = 1. \quad (8.44)$$

Exercise 8.51. In this model there are two characters, Romeo and Juliet, whose affection is quantified on the scale from -5 to 5 described below:

- -5 : Hysterical Hatred
- -2.5 : Disgust
- 0 : Indifference
- 2.5 : Sweet Affection
- 5 : Ecstatic Love

The characters struggle with frustrated love due to the lack of reciprocity of their feelings. Mathematically,

- Romeo: “My feelings for Juliet decrease in proportion to her love for me.”
- Juliet: “My love for Romeo grows in proportion to his love for me.”
- Juliet’s emotional swings lead to many sleepless nights, which consequently dampens her emotions.

This give rise to

$$\begin{cases} \frac{dx}{dt} = -\alpha y \\ \frac{dy}{dt} = \beta x - \gamma y^2 \end{cases} \quad (8.45)$$

where $x(t)$ is Romeo’s love for Juliet and $y(t)$ is Juliet’s love for Romeo at time t .

Your tasks:

1. First implement this 2D system with $x(0) = 2$, $y(0) = 0$, $\alpha = 0.2$, $\beta = 0.8$, and $\gamma = 0.1$ for $t \in [0, 60]$. What is the fate of this pair’s love under these assumptions?
 2. Write code that approximates the parameter γ that will result in Juliet having a feeling of indifference at $t = 30$. Your code should not need human supervision: you should be able to tell it that you are looking for *indifference* at $t = 30$ and turn it loose to find an approximation for γ . Assume throughout this problem that $\alpha = 0.2$, $\beta = 0.8$, $x(0) = 2$, and $y(0) = 0$. Write a description for how your code works in your homework document.
-

Exercise 8.52. In this problem we will look at the orbit of a celestial body around the sun. The body could be a satellite, comet, planet, or any other object whose mass is negligible compared to the mass of the sun. We assume that the motion takes place in a two dimensional plane so we can describe the path of the orbit with two coordinates, x and y with the point

$(0, 0)$ being used as the reference point for the sun. According to Newton's law of universal gravitation the system of differential equations that describes the motion is

$$x''(t) = \frac{-x}{(\sqrt{x^2 + y^2})^3} \quad \text{and} \quad y''(t) = \frac{-y}{(\sqrt{x^2 + y^2})^3}. \quad (8.46)$$

- Define the two velocity functions $v_x(t) = x'(t)$ and $v_y(t) = y'(t)$. Using these functions we can now write the system of two second-order differential equations as a system of four first-order equations

$$\begin{aligned} x' &= \underline{\hspace{2cm}} \\ v'_x &= \underline{\hspace{2cm}} \\ y' &= \underline{\hspace{2cm}} \\ v'_y &= \underline{\hspace{2cm}} \end{aligned} \quad (8.47)$$

- Solve the system of equations from part (a) using an appropriate solver. Start with $x(0) = 4$, $y(0) = 0$, the initial x velocity as 0, and the initial y velocity as 0.5. Create several plots showing how the dynamics of the system change for various values of the initial y velocity in the interval $t \in (0, 100)$.
 - Give an animated plot showing $x(t)$ versus $y(t)$.
-

Exercise 8.53. In this problem we consider the pursuit and evasion problem where $E(t)$ is the vector for an evader (e.g. a rabbit or a bank robber) and $P(t)$ is the vector for a pursuer (e.g. a fox chasing the rabbit or the police chasing the bank robber)

$$E(t) = \begin{pmatrix} x_e(t) \\ y_e(t) \end{pmatrix} \quad \text{and} \quad P(t) = \begin{pmatrix} x_p(t) \\ y_p(t) \end{pmatrix}. \quad (8.48)$$

Let us presume the following:

Assumption 1: the evader has a predetermined path (known only to him/her),

Assumption 2: the pursuer heads directly toward the evader at all times, and

Assumption 3: the pursuer's speed is directly proportional to the evader's speed.

From the third assumption we have

$$\|P'(t)\| = k\|E'(t)\| \quad (8.49)$$

and from the second assumption we have

$$\frac{P'(t)}{\|P'(t)\|} = \frac{E(t) - P(t)}{\|E(t) - P(t)\|}. \quad (8.50)$$

Solving for $P'(t)$ the differential equation that we need to solve becomes

$$P'(t) = k\|E'(t)\| \frac{E(t) - P(t)}{\|E(t) - P(t)\|}. \quad (8.51)$$

Your Tasks:

1. Explain assumption #2 mathematically.
2. Explain assumption #3 physically. Why is this assumption necessary mathematically?
3. Write code to find the path of the pursuer if the evader has the parametrised path

$$E(t) = \begin{pmatrix} 0 \\ 5t \end{pmatrix} \quad \text{for } t \geq 0 \quad (8.52)$$

and the pursuer initially starts at the point $P(0) = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$. Write your code so that it stops when the pursuer is within 0.1 units of the evader. Run your code for several values of k . The resulting plot should be animated.

4. Modify your code from part (c) to find the path of the pursuer if the evader has the parametrised path

$$E(t) = \begin{pmatrix} 5 + \cos(2\pi t) + 2 \sin(4\pi t) \\ 4 + 3 \cos(3\pi t) \end{pmatrix} \quad \text{for } t \geq 0 \quad (8.53)$$

and the pursuer initially starts at the point $P(0) = \begin{pmatrix} 0 \\ 50 \end{pmatrix}$. Write your code so that it stops when the pursuer is within 0.1 units of the evader. Run your code for several values of k . The resulting plot should be animated.

5. Create your own smooth path for the evader that is *challenging* for the pursuer to catch. Write your code so that it stops when the pursuer is within 0.1 units of the evader. Run your code for several values of k .
6. (Challenge) If you extend this problem to three spatial dimensions you can have the pursuer and the evader moving on a multivariable surface (i.e. hilly terrain). Implement a path along an appropriate surface but be sure that the velocities of both parties are appropriately related to the gradient of the surface.

Note: It may be easiest to build this code from scratch instead of using one of our pre-written codes.

Exercise 8.54. (This problem is modified from (Meerschaert 2013))

One of the favourite foods of the blue whale is krill. Blue whales are baleen whales and feed almost exclusively on krill. These tiny shrimp-like creatures are devoured in massive amounts to provide the principal food source for the huge whales. In the absence of predators, in uncrowded conditions, the krill population density grows at a rate of 25% per year. The presence of 500 tons/acre of krill increases the blue whale population growth rate by 2% per year, and the presence of 150,000 blue whales decreases krill growth rate by 10% per year. The population of blue whales decreases at a rate of 5% per year in the absence of krill.

These assumptions yield a pair of differential equations (a Lotka-Volterra model) that describe the population of the blue whales (B) and the krill population density (K) over time given by

$$\begin{aligned}\frac{dB}{dt} &= -0.05B + \left(\frac{0.02}{500}\right) BK \\ \frac{dK}{dt} &= 0.25K - \left(\frac{0.10}{150000}\right) BK.\end{aligned}\tag{8.54}$$

1. What are the units of $\frac{dB}{dt}$ and $\frac{dK}{dt}$?
 2. Explain what each of the four terms on the right-hand sides of the differential equations mean in the context of the problem. Include a reason for why each term is positive or negative.
 3. Find a numerical solution to the differential equation model using $B(0) = 75,000$ whales and $K(0) = 150$ tons per acre.
 4. Whaling is a huge concern in the oceans world wide. Implement a *harvesting* term into the whale differential equation, defend your mathematical choices and provide a thorough exploration of any parameters that are introduced.
-

Exercise 8.55. (This problem is modified from (Spindler 2022))

You just received a new long-range helicopter drone for your birthday! After a little practice, you try a long-range test of it by having it carry a small package to your home. A friend volunteers to take it 5 miles east of your home with the goal of flying directly back to your home. So you program and guide the drone to always head directly toward home at a speed of 6 miles per hour. However, a wind is blowing from the south at a steady 4 miles per hour. The drone, though, always attempts to head directly home. We will assume the drone always flies at the same height. What is the drone's flight path? Does it get the package to your home? What happens if the speeds are different? What if the initial distance is different? How much time does the drone's battery have to last to get home? When you make plots of your solution they must be animated.

Exercise 8.56. A trebuchet catapult throws a cow vertically into the air. The differential equation describing its acceleration is

$$\frac{d^2x}{dt^2} = -g - c \frac{dx}{dt} \left| \frac{dx}{dt} \right| \quad (8.55)$$

where $g \approx 9.8 \text{ m/s}^2$ and $c \approx 0.02 \text{ m}^{-1}$ for a typical cow. If the cow is launched at an initial upward velocity of 30 m/s, how high will it go, and when will it crash back into the ground? Hint: Change this second order differential equation into a system of first order differential equations.

Exercise 8.57 (Scipy ODEINT). It should come as no surprise that the `scipy` library has some built-in tools to solve differential equations numerically. One such tool is `scipy.integrate.odeint()`. The code below shows how to use the `.odeint()` tool to solve the differential equation $x' = -\frac{1}{3}x + \sin(t)$ with $x(0) = 1$. Take note that the `.odeint()` function expects a Python function (or `lambda` function), an initial condition, and an array of times.

Make careful note of the following:

- The function `scipy.integrate.odeint()` expects the function f to have the arguments in the order x (or y) then t . In other words, they expect you to define f as $f = f(x, t)$. This is opposite from our convention in this chapter where we have defined f as $f = f(t, x)$.
- The output of `scipy.integrate.odeint()` is an array. This is designed so that `.odeint()` can handle systems of ODEs as well as scalar ODEs. In the code below notice that we plot `x[:, 0]` instead of just `x`. This is overkill in the case of a scalar ODE, but in a system of ODEs this will be important.
- You have to specify the array of time for the `scipy.integrate.odeint()` function. It is typically easiest to use `np.linspace()` to build the array of times.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate
f = lambda x, t: -(1/3.0)*x + np.sin(t)
x0 = 1
t = np.linspace(0, 5, 1000)
x = scipy.integrate.odeint(f, x0, t)
```

```

plt.plot(t,x[:,0], 'b--')
plt.grid()
plt.show()

```

Now let us consider the system of ODEs

$$\begin{aligned}x' &= y \\y' &= -by - c \sin(x).\end{aligned}\tag{8.56}$$

In this ODE $x(t)$ is the angle from equilibrium of a pendulum, and $y(t)$ is the angular velocity of the pendulum. To solve this ODE with `scipy.integrate.odeint()` using the parameters $b = 0.25$ and $c = 5$ and the initial conditions $x(0) = \pi - 0.1$ and $y(0) = 0$ we can use the code below. (The idea to use this ODE was taken from the [documentation page for `scipy.integrate.odeint\(\)`](#).)

```

import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate
F = lambda x, t, b, c: [x[1], -b*x[1] - c*np.sin(x[0])]
x0 = [np.pi - 0.1, 0]
t = np.linspace(0, 10, 1000)
b = 0.25
c = 5
x = scipy.integrate.odeint(F, x0, t, args=(b, c))
plt.plot(t, x[:,0], 'b', t, x[:,1], 'r')
plt.grid()
plt.show()

```

Your Tasks:

1. First implement the two blocks of Python code given above. Be sure to understand what each line of code is doing. Fully comment your code, and then try the code with several different initial conditions.
2. For the pendulum system be sure to describe what your initial conditions mean in the physical setup.
3. Use `scipy.integrate.odeint()` to solve a non-trivial scalar ODE of your choosing. Clearly show your ODE and give plots of your solutions with several different initial conditions.
4. Build a numerical experiment to determine the relationship between your choice of Δt and the absolute maximum error between the solution from `.odeint()` and a known analytic solution to a scalar ODE. Support your work with appropriate plots and discussion.

5. Solve the system of differential equations from Exercise 8.52 using `scipy.integrate.odeint()`. Show appropriate plots of your solution.

8.11 Projects

In this section we propose several ideas for projects related to numerical ordinary differential equations. 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.

8.11.1 The COVID-19 Pandemic

In the paper *Modeling the COVID-19 epidemic and implementation of population-wide interventions in Italy*, by G. Giordana et al., the authors propose a robust extension to the SIR model, which they call the “SIDARTHE” model, to model the spread of the COVID-19 virus in Italy. The acronym stands for

- S = proportion of the population which is Susceptible.
- I = proportion of the population which is presently Infected. Asymptomatic, infected, and undetected.
- D = proportion of the population which has been Diagnosed. Asymptomatic, infected, and detected.
- A = proportion of the population which is Ailing. Symptomatic, infected, and undetected.
- R = proportion of the population which is Recognized. Symptomatic, infected, and detected.
- T = proportion of the population which is Threatened. Acutely symptomatic, infected, and detected.
- H = proportion of the population which is Healed.
- E = proportion of the population which is Extinct.

In the Methods section of the paper (in the paragraph that begins with “*In particular, ...*”) the authors propose initial conditions and values for all of the parameters in the model. Using these values create a numerical solution to the system of differential equations and verify that the basic reproduction number for the model is $R_0 = 2.38$ as the authors say. In the subsequent paragraphs the authors propose ways to modify the parameters to account for social distancing, stay at home orders, and other such measures. Reproduce the authors’ results from these paragraphs and fully explain all of your work. Provide sufficient plots to show the dynamics of the situation.

8.11.2 Pain Management

When a patient undergoing surgery is asked about their pain the doctors often ask patients to rate their pain on a subjective 0 to 10 scale with 0 meaning no pain and 10 meaning excruciating pain. After surgery the unmitigated pain level in a typical patient will be quite high and as such doctors typically treat with narcotics. A mathematical model (inspired by [THIS article](#) and [THIS paper](#)) of a patient's subjective pain level as treated pharmaceutically by three drugs is given as:

$$\begin{aligned}\frac{dP}{dt} &= -(k_0 + k_1 D_1 + k_2 D_2 + k_3 D_3) P + k_0 u \\ \frac{dD_1}{dt} &= -k_{D_1} D_1 + \sum_{j=1}^{N_1} \delta(t - \tau_{1,j}) \\ \frac{dD_2}{dt} &= -k_{D_2} D_2 + \sum_{j=1}^{N_2} \delta(t - \tau_{2,j}) \\ \frac{dD_3}{dt} &= -k_{D_3} D_3 + \sum_{j=1}^{N_3} \delta(t - \tau_{3,j})\end{aligned}\tag{8.57}$$

where

- P is a patient's subjective pain level on a 0 to 10 scale,
- D_i is the amount of the i^{th} drug in the patient's bloodstream,
 - D_1 is a long-acting opioid
 - D_2 is a short-acting opioid
 - D_3 is a non-opioid
- k_0 is the relaxation rate to baseline pain without drugs,
- k_i is the impact of the i^{th} drug on the relaxation rate,
- u is the patient's baseline (unmitigated) pain,
- k_{D_i} is the elimination rate of the i^{th} drug from the bloodstream,
- N_i is the total number of the i^{th} drug doses taken, and
- $\tau_{i,j}$ are the time times the patient takes the i^{th} drug.
- $\delta()$ is the Dirac delta function.

Implement this model with parameters $u = 8.01$, $k_0 = \log(2)/2$, $k_1 = 0.319$, $k_2 = 0.184$, $k_3 = 0.201$, $k_{D_1} = \log(0.5)/(-10)$, $k_{D_2} = \log(0.5)/(-4)$, and $k_{D_3} = \log(0.5)/(-4)$. Take the initial pain level to be $P(0) = 3$ with no drugs on board. Assume that the patient begins dosing the long-acting opioid at hour 2 and takes 1 dose periodically every 24 hours. Assume that the patient begins dosing the short-acting opioid at hour 0 and takes 1 dose periodically every 12 hours. Finally assume that the patient takes 1 dose of the non-opioid drug every 48 hours starts at hour 24. Of particular interest are how the pain level evolves over the first week out of surgery and how the drug concentrations evolve over this time.

Other questions:

- What does this medication schedule do to the patient's pain level?
- What happens to the patient's pain level if he/she forgets the non-opioid drug?
- What happens to the patient's pain level if he/she has a bad reaction to opioids and only takes the non-opioid drug?
- What happens to the dynamics of the system if the patient's pain starts at 9/10?
- In reality, the unmitigated pain u will decrease in time. Propose a differential equation model for the unmitigated pain that will have a stable equilibrium at 3 and has a value of 5 on day 5. Add this fifth differential equation to the pain model and examine what happens to the patient's pain over the first week. In this model, what happens after the first week if the narcotics are ceased?

8.11.3 The H1N1 Virus

The H1N1 virus, also known as the “bird flu,” is a particularly virulent bug but thankfully is also very predictable. Once a person is infected they are infectious for 9 days. Assume that a closed population of $N = 1500$ people (like a small college campus) starts with exactly 1 infected person and hence the remainder of the population is considered susceptible to the virus. Furthermore, once a person is recovered they have an immunity that typically lasts longer than the outbreak. Mathematically we can model an H1N1 outbreak of this kind using 11 compartments: susceptible people (S), 9 groups of infected people (I_j for $j = 1, 2, \dots, 9$), and recovered people (R). Write and numerically solve a system of 11 differential equations modelling the H1N1 outbreak assuming that susceptible people become infected at a rate proportional to the product of the number of susceptible people and the total number of infected people. You may assume that the initial infected person is on the first day of their infection and determine and unknown parameters using the fact that 1 week after the infection starts there are 10 total people infected.

8.11.4 The Artillery Problem

The goal of artillery is to fire a shell (e.g. a cannon ball) so that it lands on a specific target. If we ignore the effects of air resistance the differential equations describing its acceleration are very simple:

$$\frac{dv_x}{dt} = 0 \quad \text{and} \quad \frac{dv_z}{dt} = -g \quad (8.58)$$

where v_x and v_z are the velocities in the x and z directions respectively and g is the acceleration due to gravity ($g = 9.8 \text{ m/s}^2$). We can use these equations to *easily* show that the resulting trajectory is parabolic. Once we know this we can easily calculate the initial speed v_0 and angle θ_0 above the horizontal necessary for the shell to reach the target. We will undoubtedly find that the maximum range will always result from an angle of $\theta_0 = 45^\circ$.

The effects of air resistance are significant when the shell must travel a large distance or when the speed is large. If we modify the equations to include a simple model of air resistance the governing equations become

$$\frac{dv_x}{dt} = -cv_x\sqrt{v_x^2 + v_z^2} \quad \text{and} \quad \frac{dv_z}{dt} = -g - cv_z\sqrt{v_x^2 + v_z^2} \quad (8.59)$$

where the constant c depends on the shape and density of the shell and the density of air. For this project assume that $c = 10^{-3}\text{m}^{-1}$. To calculate the components of the position vector recall that since the derivative of position, $s(t)$, is velocity we have

$$s_x(t) = \int_0^t v_x(\tau)d\tau \quad \text{and} \quad s_z(t) = \int_0^t v_z(\tau)d\tau. \quad (8.60)$$

Now, imagine that you are living 200 years ago, acting as a consultant to an artillery officer who will be going into battle (perhaps against Napoleon – he was known for hiring mathematicians to help his war efforts). Although computers have not yet been invented, given a few hours or a few days to work, a person living in this time could project trajectories using numerical methods (yes, numerical solutions to differential equations were well known back then too). Using this, you can try various initial speeds v_0 and angles θ_0 until you find a pair that reach any target. However, the artillery officer needs a faster and simpler method. He can do maths, but performing hundreds or thousands of numerical calculations on the battlefield is simply not practical. Suppose that our artillery piece will be firing at a target that is a distance Δx away, and that Δx is approximately half a mile away – not exactly half a mile, but in that general neighbourhood.

1. Develop a method for estimating v_0 and θ_0 with reasonable accuracy given the exact range to the target, Δx . Your method needs to be simple enough to use in real time on a historic (Napoleon-era) battle field without the aid of a computer. (Be sure to persuade me that your numerical solution is accurate enough.)
2. Discuss the sensitivity in your solutions to variations in the constant c .

3. Extend this problem to make it more realistic. A few possible extensions are listed below but please do not restrict yourselves just to this list and do not think that you need to do everything on the list.

- You could consider the effects of targets at different altitudes Δz .
- You could consider moving targets.
- You could consider headwinds and/or tailwinds.
- You could consider winds coming from an angle outside the xz -plane.
- You could consider shooting the cannon from a boat with the target on shore (the waves could be interesting!).
- ...You could consider any other physical situation which I have not listed here, but you have to do some amount of extension from the *basics*.

The final product of this project will be:

- a **technical paper** describing your method to a mathematically sophisticated audience, and
- a **field manual** instructing the artillery officer how to use your method.

You can put both products in one paper. Just use a section header to start the field manual.

9 Partial Differential Equations

When you open the toolkit of differential equations you see the hammers and saws of engineering and physics for the past two centuries and for the foreseeable future.

—Benoit Mandelbrot

9.1 Intro to PDEs

Partial differential equations (PDEs) are differential equations involving the partial derivatives of an unknown multivariable function. In most of this chapter we will examine two classical problems from physics: heat transport phenomena and wave phenomena. Do not think, however, that just because we are focusing on these two primary examples that this is the extent of the utility of PDEs. Basically, every scientific field has been impacted by (or has directly impacted) the study of PDEs. Any phenomenon that can be modelled via the change in multiple continuous variables (not restricted to space and time) is likely governed by a PDE model. Some common phenomena that are modelled by PDEs are:

- heat transport
 - The heat equation models heat energy (temperature) diffusing through a metal rod or a solid body
- diffusion of a concentrated substance
 - The diffusion equation is a PDE model for the diffusion of smells, contaminants, or the motion of a solute
- wave propagation
 - The wave equation is a PDE that can be used to model the standing waves on a guitar string, the waves on lake, or sound waves traveling through the air
- travelling waves
 - The traveling wave equation is a PDE that can be used to model pulses of light propagating through a fiber optic cable or regions of high density traffic moving along a highway.

- quantum mechanics
 - The wave functions of quantum mechanics are described by a PDE called the Schrodinger Equation.
- electro-magnetism
 - Maxwell's Equations are a system of PDEs describing the relationships between electricity and magnetism.
- fluid flow
 - The Navier-Stokes equations are a system of PDEs that model fluids in three dimensions – including turbulent flow.
 - Darcy's Law and Richard's equation are PDE models for the motion of fluids moving through saturated and unsaturated soils.
- stress and strain in structures
 - The Linear Elasticity equation is a PDE that models the stresses in a solid body (like a bridge or a building) under load.
- spatial patterns
 - Solutions to the Helmholtz equation are known for exhibiting *Turing patterns* which are patterns like leopard spots or zebra stripes.
- ... and many more ...

In many cases we are interested in solving PDEs in terms of our usual three spatial dimensions along with an extra dimension for time. Often we do not have to work with all three spatial dimensions (like if the domain is much larger in one or two directions versus the others) or in some cases (like in linear elasticity) we do not need to worry about time.

There is a wealth of wonderful theory for finding analytic solutions to many special classes of PDEs. However, most PDEs simply do not lend themselves to analytic solutions that we can write down in terms of the regular mathematical operations of sums, products, powers, roots, trigonometric functions, logarithms, etc. For these PDEs we must turn to numerical methods to approximate the solution.

Recall that numerical solutions to ODEs were approximations of the value of the unknown function at a discrete set of times. Similarly, numerical solutions to PDEs are going to be approximations of the value of the unknown function at a discrete set of points in time AND space.

What we will cover in this chapter will include one primary and powerful technique for approximating solutions to PDEs: **the finite difference method**. There are many other techniques

for approximating solutions to PDEs, and the field of numerical PDEs is still an active area of mathematical and scientific research.

9.2 The Heat Equation

You have probably met the heat equation, also known as the diffusion equation, in a previous module. The heat equation is a partial differential equation that describes how heat diffuses through a material. The heat equation is a parabolic PDE and is given by

$$\frac{\partial u}{\partial t} = D \nabla^2 u$$

where $u(t, x)$ is the temperature of the material at time t and position x and D is the diffusion coefficient. The heat equation is a simple model for heat diffusion but also describes diffusion in general, like the diffusion of a solute in a solvent or of plants in a field or, ... well, you get the idea.

In the remainder of this section we will use a technique called **the finite difference method** to build numerical approximations to solutions of the heat equation in 1D, 2D, and 3D. You of course know that the heat equation is easy to solve analytically, given that it is a linear homogeneous PDE with constant coefficients. However, the finite difference method is a powerful tool for solving similar PDEs that do not have simple analytic solutions. The advantage of using the heat equation as a test case for the finite difference method is that we can easily verify the accuracy of our numerical solutions by comparing them to the known analytic solutions.

9.2.1 In One Spatial Dimensions

For the sake of simplicity we will start by considering the heat equation in 1 spatial dimension:

$$\frac{\partial u}{\partial t} = D \frac{\partial^2 u}{\partial x^2}.$$

We will also use the alternative notation

$$u_t = Du_{xx}, \tag{9.1}$$

where the subscripts denote partial derivatives.

Exercise 9.1. Just as we did in Chapter 8 to approximate solutions to ODEs, we will start by partitioning the spatial domain into finitely many pieces and we will partition time into finitely many pieces. We do this by introducing a grid of points (t_n, x_i) where $t_n = t_0 + n \Delta t$ and $x_i = x_0 + i \Delta x$. Then we want to build a numerical approximation to the function $u(t, x)$ at these grid points.

First we need to introduce some notation for the numerical solution. As you will see in a moment, there is a lot to keep track of in numerical PDEs so careful indexing and well-chosen notation is essential. Let U_i^n be the approximation of the solution to $u(t, x)$ at the point $t = t_n = t_0 + n \Delta t$ and $x = x_i = x_0 + i \Delta x$ (since we have two variables we need two indices). For example, U_4^1 is the value of the approximation at time t_1 and at the spatial point x_4 .

Next we need to approximate both derivatives u_t and u_{xx} in the PDE using methods that we have used before. Now would be a good time to go back to Chapter 5 and refresh your memory for how we build approximations of derivatives.

- (a) Use the forward-difference formula to approximate the time derivative u_t at the point $t = t_n$ and $x = x_i$.

$$u_t(t_n, x_i) \approx \frac{??? - ???}{\Delta t}.$$

- (b) Use the centred-difference formula to approximate the second spatial derivative u_{xx} at the point $t = t_n$ and $x = x_i$.

$$u_{xx}(t_n, x_i) \approx \frac{??? - ??? + ???}{2\Delta x}.$$

- (c) Put your answers from parts (a) and (b) together using the 1D heat equation (Eq. 9.1)

$$\frac{??? - ???}{\Delta t} = D \left(\frac{??? - ??? + ???}{\Delta x^2} \right).$$

Be sure that your indexing is correct: the superscript n is the index for time and the subscript i is the index for space.

- (d) Rearrange your result from part (c) to solve for U_i^{n+1} :

$$U_i^{n+1} = ??? + \frac{D\Delta t}{\Delta x^2} (??? - ??? + ???).$$

The iterative scheme which you just derived is called the **forward difference scheme** for the heat equation. Notice that the term on the left is the only term at the next time step $n+1$. So, for every spatial point x_i we can build U_i^{n+1} by evaluating the right-hand side of the finite difference scheme.

- (e) The numerical errors made by using the forward difference scheme we just built come from two sources: from the approximation of the time derivative and from the approximation of the second spatial derivative. Fill in the question marks in the powers of the following expression:

$$\text{Numerical Error} = \mathcal{O}(\Delta t^{???}) + \mathcal{O}(\Delta x^{???}).$$

- (f) Explain what the result from part (e) means in plain English?

There are many different finite difference schemes due to the fact that there are many different ways to approximate derivatives (See Chapter 5). One convenient way to keep track of which information you are using and what you are calculating in a finite difference scheme is to use a **finite difference stencil image**. Figure 9.1 shows the finite difference stencil for the approximation to the heat equation that you built in the previous exercise. In this figure we are showing that the function values U_{i-1}^n , U_i^n , and U_{i+1}^n at the points x_{i-1} , x_i , and x_{i+1} at time step t_n are used to calculate U_i^{n+1} . We will build similar stencil diagrams for other finite difference schemes throughout this chapter.

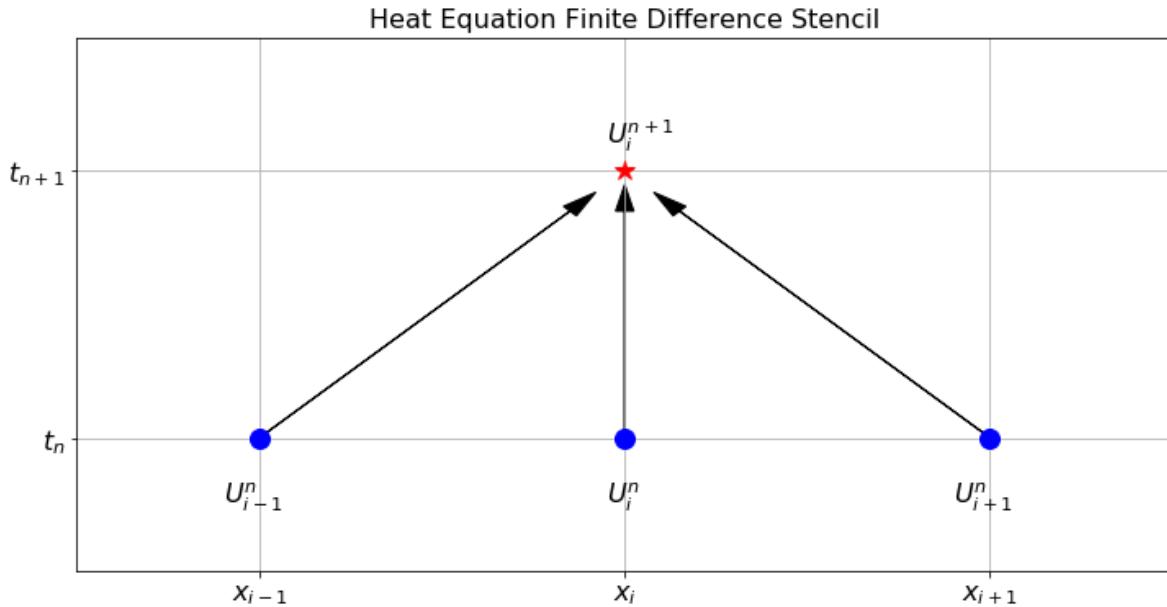


Figure 9.1: The forward difference stencil for the 1D heat equation.

Exercise 9.2. Now we want to implement your answer to part (d) of the previous exercise to approximate the solution to the following problem: Solve

$$u_t = 0.1u_{xx}$$

on the domain $0 < x < 1$ and $0 < t < 1$ with the initial condition with

$$u(0, x) = \sin(2\pi x)$$

and boundary conditions

$$u(t, 0) = 0, \text{ and } u(t, 1) = 0.$$

For this purpose divide the x domain into 20 equal pieces and the t domain into 100 equal pieces.

Some partial code is given below to get you started.

- First we import the proper libraries, set up the time domain, and set up the spatial domain.

```
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interactive

# Write code to give a vector of times starting at t=0 and ending
# at t=1 that divides the interval into 100 equal pieces.

# Calculate the step size `dt`.

# Write code to give a vector of x values starting at x=0 and
# ending exactly at x=1 that divides the interval into 20 equal pieces.

# Calculate the step size `dx`.

# Specify the diffusion coefficient
D = 0.01
# The coefficient "a" appears in the forward difference scheme.
a = D*dt / dx**2

print("dt=", dt, ", dx=", dx, " and D dt/dx^2=", a)
```

- Next we build the array U so we can store all of the approximations at all times and at all spatial points. The array will have the dimensions `len(t)` by `len(x)`. We then need to enforce the boundary conditions so for all times we fill the proper portions of the array with the proper boundary conditions. Lastly, we will build the initial condition for all spatial steps in the first time step.

```
U = np.zeros((len(t),len(x)))
U[:,0] = # left boundary condition
U[:,-1] = # right boundary condition
U[0,:] = # the function for the init. condition (should depend on x)
```

- Now we step through a loop that fills the U array one row at a time. Keep in mind that we want to leave the boundary conditions fixed so we will only fill indices 1 through -2 (stop and explain this). Be careful to get the indexing correct. For example, if we want U_i^n we use $U[n, 1:-1]$, if we want U_{i+1}^n we use $U[n, 2:]$, if we want U_i^{n+1} we use $U[n+1, 1:-1]$, etc.

```
for n in range(len(t)-1):
    U[n+1,1:-1] = U[n,:,:] + a*( U[n,:,:] - 2*U[n,:,:] + U[n,:,:])
```

- It remains to visualise the solutions. You can either make a plot or an animation to illustrate the time evolution of u . For each of these there are various Python packages you could use. Below is a function `plot_solution_1d()` using `plotly` to make a plot and a function `animate_solution_1d()` using `matplotlib` to make an animated 2D plot. You can use either of these or you can use your own plotting code.

```
import plotly.graph_objects as go

def plot_solution_1d(t, x, U):
    """Plots the numerical approximation to a function u(t,x).

    Args:
        t: A vector of time values.
        x: A vector of spatial values.
        U: A 2D array approximating the solution u(x,t) at each grid point.
    """
    fig = go.Figure(data=[go.Surface(z=U, x=x, y=t)])
    fig.update_layout(
        width=800, height=600,
        scene=dict(
            yaxis_title='t',
            xaxis_title='u'
        )
    )
    return fig
```

```
import matplotlib.pyplot as plt
from matplotlib import animation, rc
from IPython.display import HTML

def animate_solution_1d(t, x, U):
    """Animates the numerical approximation to a function u(t,x).
```

```

Args:
    t: A vector of time values.
    x: A vector of spatial values.
    U: A 2D array approximating the solution u(x,t) at each grid point.
"""
fig, ax = plt.subplots()
plt.close()
ax.grid()
ax.set_xlabel("x")
ax.set_ylabel("u")
ax.set_xlim((np.min(x), np.max(x)))
ax.set_ylim((np.min(U), np.max(U)))
frame, = ax.plot([], [], linewidth=2)

# Don't display every time
step = int(len(t)/30)+1
frames = range(0, int(len(t)/step), 1)

def animator(i):
    n = i*step
    ax.set_title(f"t = {t[n]:.2f}")
    frame.set_data(x, U[n,:])
    return (frame, )

ani = animation.FuncAnimation(fig, animator, frames=frames, interval=100)
rc('animation', html='jshtml') # embed in the HTML for Google Colab
return ani

```

Exercise 9.3. Now wrap up your code for solving the one-dimensional heat equation as a function so that you can easily call it with different parameters.

```

def heat1d(u_0, D=0.1, t_0=0, t_max=1, N_t=100, x_left=0, x_right=1, N_x=20):
    """Solves the 1D heat equation using the forward difference method.

```

This function solves the 1D heat equation with given initial and boundary conditions. It also prints a diagnostic message stating the step sizes `dt` and `dx` used and the value of `a = D*dt/dx**2`.

```

Args:

```

```

u_0: A function giving the initial condition u(0,x).
D: The diffusion coefficient. Defaults to 1.
t_0: The initial time. Defaults to 0.
t_max: The maximum time. Defaults to 1.
N_t: The number of time steps. Defaults to 100.
x_left: The left boundary of the spatial domain. Defaults to 0.
x_right: The right boundary of the spatial domain. Defaults to 1.
N_x: The number of spatial steps. Defaults to 20.

>Returns:
A tuple containing the following:
    t: A vector of time values.
    x: A vector of spatial values.
    U: A 2D array approximating the solution u(t,x) at each grid point.
"""
# Your code goes here

```

Use your function to solve the heat equation with diffusion coefficient $D = 0.1$ and the following initial and boundary conditions:

$$u(0, x) = \sin(2\pi x), u(t, 0) = 0, \text{ and } u(t, 1)$$

Use stepsizes $\Delta t = 0.01$ and $\Delta x = 0.01$ to determine an approximate value for $u(0.2, 0.25)$.

Exercise 9.4. Now run the solution method from the previous exercise with the same diffusion coefficient $D = 0.1$, the same step sizes $\Delta t = 0.01$ and $\Delta x = 0.01$, and the same initial and boundary conditions but run it for a longer time $t = 0.5$ and plot the solution on the domain $t \in [0, 0.5]$ and $x \in [0, 1]$. Do you believe what you see? What is happening to the solution?

Exercise 9.5. You will have found that you did not get a sensible solution from your method for the previous problem. The point of this exercise is to show that value of $a = D \frac{\Delta t}{\Delta x^2}$ controls the stability of the forward difference solution to the heat equation, and furthermore that there is a threshold for a above which the forward difference scheme will be unstable. Experiment with values of Δt and Δx and conjecture the values of $a = D \frac{\Delta t}{\Delta x^2}$ that give a stable result. Your conjecture should take the form:

If $a = D \frac{\Delta t}{\Delta x^2} < \underline{\hspace{2cm}}$ then the forward difference solution for the 1D heat equation is stable. Otherwise it is unstable.

Hint: the threshold is a simple fraction. If you think you have found a value for a at which the method is stable, run the simulation for longer (while keeping the same Δt) to check that it is really stable. Close to the threshold the errors grow more slowly (albeit still exponentially).

Exercise 9.6. Consider the one dimensional heat equation with diffusion coefficient $D = 1$:

$$u_t = u_{xx}.$$

We want to solve this equation on the domain $x \in [0, 1]$ and $t \in [0, 0.1]$ subject to the initial condition $u(0, x) = \sin(\pi x)$ and the boundary conditions $u(t, 0) = u(t, 1) = 0$.

- (a) Show that the function $u(t, x) = e^{-\pi^2 t} \sin(\pi x)$ is a solution to this heat equation, satisfies the initial condition, and satisfies the boundary conditions.
- (b) Pick values of Δt and Δx so that you can get a stable forward difference solution to this heat equation. Then make a plot of your numerical solution.
- (c) Compare your plot to the plot of the exact solution that you can get with

```
X, T = np.meshgrid(x, t)
u_exact = np.exp(-np.pi**2*T)*np.sin(np.pi*X)
plot_solution_1d(t, x, u_exact)
```

Exercise 9.7. Now let us change the initial condition to $u(0, x) = \sin(\pi x) + \sin(3\pi x)$. We will keep the same boundary conditions as before: $u(t, 0) = u(t, 1) = 0$.

- (a) Show that the function $u(t, x) = e^{-\pi^2 t} \sin(\pi x) + e^{-9\pi^2 t} \sin(3\pi x)$ is a solution to this heat equation, matches this new initial condition, and matches the boundary conditions.
 - (b) Pick values of Δt and Δx so that you can get a stable forward difference solution to this heat equation. Make a 3d plot of your numerical solution.
 - (c) Compare your plot to the plot of the exact solution.
-

9.2.2 Different Boundary Conditions

In any initial and boundary value problem such as the heat equation, the boundary are often of Dirichlet or Neumann type. In Dirichlet boundary conditions the values of the solution at the boundary are specified. In contrast, Neumann boundary conditions specify the flux at the boundary instead of the value of the solution.

Exercise 9.8 (Time-dependent Dirichlet Boundary Condition). Modify your 1D heat equation code to plot an approximate solution of the diffusion equation $u_t = 0.5u_{xx}$ with $x \in (0, 1)$, $u(0, x) = \sin(2\pi x)$, $u(t, 0) = 0$ and $u(t, 1) = \sin(5\pi t)$.

Exercise 9.9 (Neumann Boundary Condition). Consider the 1D heat equation $u_t = u_{xx}$ with boundary conditions $u_x(t, 0) = 0$ and $u(t, 1) = 0$ with initial condition $u(0, x) = \cos(\pi x/2)$. Notice that the initial condition satisfies both boundary conditions: $\frac{d}{dx}(\cos(\pi \cdot x/2))\Big|_{x=0} = 0$ and $\cos(\pi \cdot 1/2) = 0$. As the heat profile evolves in time the Neumann boundary condition $u_x(t, 0) = 0$ says that the slope of the solution needs to be fixed at 0 at the left-hand boundary.

- (a) Draw several images of what the solution to the PDE should look like as time evolves. Be sure that all boundary conditions are satisfied and that your solution appears to solve the heat equation.
- (b) The Neumann boundary condition $u_x(t, 0) = 0$ can be approximated with the first order approximation

$$u_x(t_n, 0) \approx \frac{U_1^n - U_0^n}{\Delta x} \text{ for all } n.$$

If we set this approximation to 0 (since $u_x(t, 0) = 0$) and solve for U_0^n we get an additional constraint at every time step of the numerical solution to the heat equation:

$$U_0^n = ??? \text{ for all } n.$$

- (c) Modify your 1D heat equation code to implement this Neumann boundary condition, plot the numerical solution and verify visually that the Neumann boundary is satisfied.

9.2.3 In Two Spatial Dimensions

Now we transition to the two dimensional heat equation. Instead of thinking of this as heating a long metal rod we can think of heating a thin plate of metal (like a flat cookie sheet). The

heat equation models the propagation of the heat energy throughout the 2D surface. In two spatial dimensions the heat equation is

$$\frac{\partial u}{\partial t} = D \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right),$$

or, using subscript notation for the partial derivatives,

$$u_t = D(u_{xx} + u_{yy}).$$

Exercise 9.10. Let us build a numerical solution to the 2D heat equation. We need to make a minor modification to our notation since there is now one more spatial dimension to keep track of. Let $U_{i,j}^n$ be the approximation to u at the point (t_n, x_i, y_j) . For example, $U_{2,3}^4$ will be the approximation to the solution at the point (t_4, x_2, y_3) .

- (a) We already know how to approximate the time derivative in the heat equation:

$$u_t(t_n, x_i, y_j) \approx \frac{U_{i,j}^{n+1} - U_{i,j}^n}{\Delta t}.$$

The new challenge now is that we have two spatial partial derivatives: one in x and one in y . Use what you learned in Chapter 5 to write the approximations of u_{xx} and u_{yy} .

$$u_{xx}(t_n, x_i, y_j) \approx \frac{??? - ??? + ???}{\Delta x^2}$$

$$u_{yy}(t_n, x_i, y_j) \approx \frac{??? - ??? + ???}{\Delta y^2}$$

Take careful note that the index i is the only one that changes for the x derivative. Similarly, the index j is the only one that changes for the y derivative.

- (b) Put your answers to part (a) together with the 2D heat equation

$$\frac{U_{i,j}^{n+1} - U_{i,j}^n}{\Delta t} = D \left(\frac{??? - ??? + ???}{\Delta x^2} + \frac{??? - ??? + ???}{\Delta y^2} \right).$$

- (c) Let us make one simplifying assumption. Choose the partition of the domain so that $\Delta x = \Delta y$. Note that we can usually do this in square domains. In more complicated domains we will need to be more careful. Simplify the right-hand side of your answer to part (b) under this assumption.

$$\frac{U_{i,j}^{n+1} - U_{i,j}^n}{\Delta t} = D \left(\frac{??? + ??? - ??? + ??? + ???}{\Delta x^2} \right).$$

- (d) Now solve your result from part (c) for $U_{i,j}^{n+1}$. Your answer is the explicit forward difference scheme for the 2D heat equation.

$$U_{i,j}^{n+1} = U_{i,j}^n + \frac{D \cdot ???}{??} (??? + ??? - ??? + ??? + ???)$$

The finite difference stencil for the 2D heat equation is a bit more complicated since we now have three indices to track. Hence, the stencil is naturally three dimensional. Figure 9.2 shows the stencil for the forward difference scheme that we built in the previous exercise. The left-hand subplot in the figure shows the five points used in time step t_n , and the right-hand subplot shows the one point that is calculated at time step t_{n+1} .

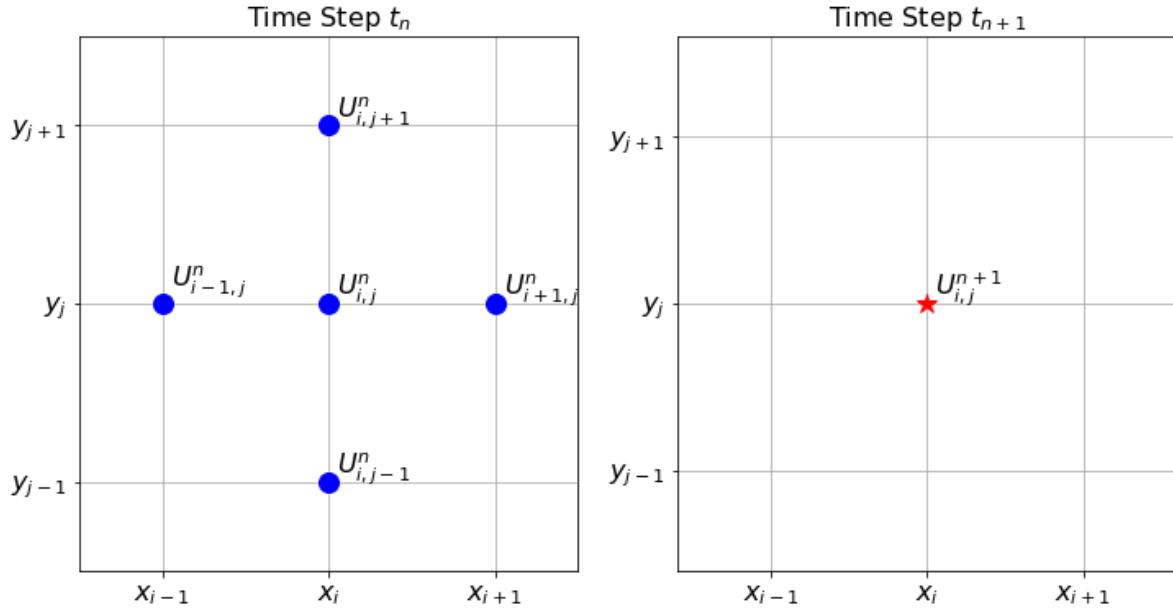


Figure 9.2: The finite difference stencil for the 2D heat equation.

Exercise 9.11. Now we need to implement the finite difference scheme that you developed in the previous problem. As a model problem, consider the 2D heat equation $u_t = D(u_{xx} + u_{yy})$ on the domain $(x, y) \in [0, 1] \times [0, 1]$ with the initial condition $u(0, x, y) = \sin(\pi x) \sin(\pi y)$, Dirichlet boundary conditions $u(t, x, 0) = u(t, x, 1) = u(t, 0, y) = u(t, 1, y) = 0$, and $D = 1$. Fill in the holes in the following code chunks.

- First we import the proper libraries and set up the domains for x , y , and t .

```

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm # this allows for color maps
from ipywidgets import interactive

# Write code to build a linearly spaced array of x values
# starting at 0 and ending at exactly 1
x = # your code here
y = x # this could be generalised later
# The consequence of the previous line is that dy = dx.
dx = # Extract dx from your array of x values.
# Now write code to build a linearly spaced array of time values
# starting at 0 and ending at 0.1.
# You will want to use many more values for time than for space
# (think about the stability conditions from the 1D heat equation).
t = # your code here
dt = # Extract dt from your array of t values

# Next we will use the np.meshgrid() command to turn the arrays of
# x and y values into 2D grids of x and y values.
# If you match the corresponding entries of X and Y then you get
# every ordered pair in the domain.
Y, X = np.meshgrid(y, x)

# Next we set up a 3 dimensional array of zeros to store all of
# the time steps of the solutions.
U = np.zeros((len(t), len(x), len(y)))

```

- Next we have to set up the boundary and initial conditions for the given problem.

```

U[0,:,:] = # initial condition depending on X and Y
U[:,0,:] = # boundary condition for x=0
U[:, -1, :] = # boundary condition for x=1
U[:, :, 0] = # boundary condition for y=0
U[:, :, -1] = # boundary condition for y=1

```

- We know that the value of $D\Delta t/\Delta x^2$ controls the stability of the forward difference method. Therefore, the next step in our code is to calculate this value and print it.

```

D = 1
a = D*dt/dx**2
print(a)

```

- Next for the part of the code that actually calculates all of the time steps. Be sure to keep the indexing straight. Also be sure that we are calculating all of the spatial indices *inside* the domain since the boundary conditions dictate what happens on the boundary.

```
for n in range(len(t)-1):
    U[n+1,1:-1,1:-1] = U[n,1:-1,1:-1] + \
        a*(U[n, ::?, ::?] + \
            U[n, ::?, ::?] - \
            4*U[n, ::?, ::?] + \
            U[n, ::?, ::?] + \
            U[n, ::?, ::?])
```

- Finally, we just need to visualize the solution. We can no longer make a plot of u against t, x and y because that would require four dimensions. So we will animate the solution. You can use the following function:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
from mpl_toolkits.mplot3d import Axes3D
from IPython.display import HTML

def animate_solution_2d(t, x, y, U):
    Y, X = np.meshgrid(y, x)

    # Set up the figure and axis
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    # Initialize the surface plot
    surface = [ax.plot_surface(X, Y, U[0, :, :], cmap='viridis')]

    # Don't display every time
    step = int(len(t)/30)+1
    frames = int(len(t)/step)

    def animate(i):
        n = i*step
        # Update the data of the surface plot for each frame
        ax.clear() # Clear the previous frame
        surface[0] = ax.plot_surface(X, Y, U[n, :, :], cmap='viridis')
        ax.set_zlim(np.min(U), np.max(U))
        ax.set_title(f"Time: {t[n]:.2f}")

    return表面
```

```

# Create animation
ani = animation.FuncAnimation(fig, animate, frames=frames, repeat=True)
plt.close()

# Display the animation
return HTML(ani.to_jshtml())

```

Exercise 9.12. Time to do some experimentation with your new 2D heat equation code! Numerically solve the 2D heat equation with different boundary conditions (both Dirichlet and Neumann). Be prepared to present your solutions.

Exercise 9.13. In order for the forward difference solution to the 2D heat equation on a square domain to be stable we need $D\Delta t/\Delta x^2 < \underline{\hspace{2cm}}$.

Experiment with several parameters to empirically determine the bound.

Exercise 9.14. Now solve the 2D heat equation on a rectangular domain. You will need to make some modifications to your code since it is unlikely that assuming that $\Delta x = \Delta y$ is a good assumption any longer. Again, be prepared to present your solutions.

9.2.4 Variations on the Heat Equation

The heat equation is a parabolic PDE and the forward-difference method that we have developed can be adapted to work for other parabolic PDEs.

9.2.4.1 Reaction-Diffusion Equations

For example, the heat equation can be modified to include a reaction term. The reaction-diffusion equation is a PDE that models the diffusion of a substance in space and time with a reaction term that describes the rate of change of the substance due to some reaction. While it has its origin in chemistry, it shows up in many other fields as well, for example in ecology and epidemiology.

Exercise 9.15 (Fisher-KPP Equation). Modify your 1D heat equation code to calculate an approximate solution of the Fisher-KPP equation

$$u_t = u_{xx} + u(1 - u)$$

with $t \in [0, 10]$, $x \in (0, 50)$, boundary conditions

$$u(t, 0) = 0, \quad u(t, 50) = 1$$

and initial condition

$$u(0, x) = \frac{1 + \tanh\left(\frac{x - 40}{2}\right)}{2}.$$

Use `animate_solution_1d()` to visualize the solution. How does the solution change as time evolves?

9.2.4.2 Advective-Diffusion Equations

The diffusion term usually arises from random spatial motion of particles. However, in some cases the particles are advected by a flow field. In this case we need to add an advection term to the diffusion equation. The advection-diffusion equation is a PDE that models the diffusion of a substance in space and time with an advection term that describes the rate of change of the substance due to some flow field.

Exercise 9.16. Modify your 1D heat equation code to plot an approximate solution of the following simple advection-diffusion equation:

$$u_t = 0.1u_{xx} - u_x$$

Use the forward difference formula for the u_x term and the centred difference formula for the u_{xx} term. Use the initial condition $u(0, x) = \sin(\pi x)$, Dirichlet boundary conditions $u(t, 0) = 0$ and $u(t, 1) = 0$, and $t \in [0, 1]$. Use 20 spatial steps and 100 time steps. Make a plot and an animation of the solution.

9.2.5 Implicit Methods

Let us summarize the stability criteria for the forward difference solutions to the heat equation.

- In the 1D heat equation the forward difference solution is stable if $D\Delta t/\Delta x^2 < \underline{\hspace{2cm}}$.
- In the 2D heat equation the forward difference solution is stable if $D\Delta t/\Delta x^2 < \underline{\hspace{2cm}}$ (assuming a square domain where $\Delta x = \Delta y$)

Exercise 9.17 (Sawtooth Errors). We have already seen that the 1D heat equation is stable if $D\Delta t/\Delta x^2 < 0.5$. The goal of this problem is to show what, exactly, occurs when we choose parameters in the unstable region. We will solve the PDE $u_t = u_{xx}$ on the domain $x \in [0, 1]$ with initial conditions $u(0, x) = \sin(\pi x)$ and homogeneous Dirichlet boundary conditions $u(t, 0) = u(t, 1) = 0$ for all $t \in [0, 0.25]$. The analytic solution is $u(t, x) = e^{-\pi^2 t} \sin(\pi x)$. To build the spatial and temporal grid use 20 spatial steps and 100 time steps. This means that $\Delta x = 0.05$ and $\Delta t = 0.0025$ so the ratio $D\Delta t/\Delta x^2 = 1 > 0.5$ (certainly in the unstable region). Solve the heat equation with your `heat1d()` function using these parameters. Make plots of the approximate solution on top of the exact solution at time steps 0, 10, 20, 30, 31, 32, 33, 34, etc. Describe what you observe.

Exercise 9.18 (Hedgehog Errors). Solve the 2D heat equation on the unit square with homogeneous Dirichlet boundary conditions with the following parameters:

- A diffusion coefficient of $D = 1$;
- A partition of 21 points in both the x and y direction;
- 301 points between 0 and 0.25 for time;
- An initial condition of $u(0, x, y) = \sin(\pi x) \sin(\pi y)$.

What happens near time step number 70?

It is actually possible to beat the stability criteria given in the previous exercises! What follows are two implicit methods that use a forward-looking scheme to help completely avoid unstable solutions. The primary advantage to these schemes is that we will not need to pay as close attention to the ratio of the time step to the square of the spatial step. Instead, we can take time and spatial steps that are appropriate for the application we have in mind.

Exercise 9.19 (Backward Difference Scheme). For the 1D heat equation $u_t = Du_{xx}$ we have been finding the numerical solution using the explicit finite difference scheme

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = D \frac{U_{i+1}^n - 2U_i^n + U_{i-1}^n}{\Delta x^2}$$

where we approximate the spatial derivative with the centred difference and the time derivative with the usual forward difference. If, however, we use the backward difference formula for the time derivative we get the finite difference scheme

$$\frac{U_i^n - U_i^{n-1}}{\Delta t} = D \frac{U_{i+1}^n - 2U_i^n + U_{i-1}^n}{\Delta x^2}.$$

or, shifting to the next timestep,

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = D \frac{U_{i+1}^{n+1} - 2U_i^{n+1} + U_{i-1}^{n+1}}{\Delta x^2}.$$

This may seem completely ridiculous since we do not yet know the information at time step $n + 1$ but some algebraic rearrangement shows that we can treat this as a system of linear equations which can be solved (using something like `np.linalg.solve()`) for the $(n + 1)^{st}$ time step.

We again introduce the coefficient $a = D\Delta t/\Delta x^2$. This will save a little bit of writing in the coming steps.

1. Rearrange the new finite difference scheme so that all of the terms at the $(n + 1)^{st}$ time step are on the left-hand side and all of the term at the n^{th} time step are on the right-hand side.

$$(\underline{\quad})U_{i-1}^{n+1} + (\underline{\quad})U_i^{n+1} + (\underline{\quad})U_{i+1}^{n+1} = U_i^n$$

2. Now we are going to build a very small example with only 6 spatial points so that you can clearly see the structure of the resulting linear system.

- a. If we have 6 total points in the spatial grid (x_0, x_1, \dots, x_5) then we have the following equations (fill in the blanks):

$$\begin{aligned} & (\text{for } x_1:) \quad \underline{\quad}U_0^{n+1} + \underline{\quad}U_1^{n+1} + \underline{\quad}U_2^{n+1} = U_1^n \\ & (\text{for } x_2:) \quad \underline{\quad}U_1^{n+1} + \underline{\quad}U_2^{n+1} + \underline{\quad}U_3^{n+1} = U_2^n \\ & (\text{for } x_3:) \quad \underline{\quad}U_2^{n+1} + \underline{\quad}U_3^{n+1} + \underline{\quad}U_4^{n+1} = U_3^n \\ & (\text{for } x_4:) \quad \underline{\quad}U_3^{n+1} + \underline{\quad}U_4^{n+1} + \underline{\quad}U_5^{n+1} = U_4^n \end{aligned}$$

- b. Notice that we already know U_0^{n+1} and U_5^{n+1} since these are dictated by the boundary conditions (assuming Dirichlet boundary conditions). Hence we can move these known quantities to the right-hand side of the equations and hence rewrite the system of equations as:

$$\begin{aligned}
 \text{(for } x_1:) \quad & \underline{\hspace{2cm}} U_1^{n+1} + \underline{\hspace{2cm}} U_2^{n+1} = U_1^n + \underline{\hspace{2cm}} U_0^{n+1} \\
 \text{(for } x_2:) \quad & \underline{\hspace{2cm}} U_1^{n+1} + \underline{\hspace{2cm}} U_2^{n+1} + \underline{\hspace{2cm}} U_3^{n+1} = U_2^n \\
 \text{(for } x_3:) \quad & \underline{\hspace{2cm}} U_2^{n+1} + \underline{\hspace{2cm}} U_3^{n+1} + \underline{\hspace{2cm}} U_4^{n+1} = U_3^n \\
 \text{(for } x_4:) \quad & \underline{\hspace{2cm}} U_3^{n+1} + \underline{\hspace{2cm}} U_4^{n+1} = U_4^n + \underline{\hspace{2cm}} U_5^{n+1}
 \end{aligned}$$

- c. Now we can write this as a matrix equation:

$$\begin{pmatrix} \underline{\hspace{2cm}} & \underline{\hspace{2cm}} & 0 & 0 \\ \underline{\hspace{2cm}} & \underline{\hspace{2cm}} & \underline{\hspace{2cm}} & 0 \\ 0 & \underline{\hspace{2cm}} & \underline{\hspace{2cm}} & \underline{\hspace{2cm}} \\ 0 & 0 & \underline{\hspace{2cm}} & \underline{\hspace{2cm}} \end{pmatrix} \begin{pmatrix} U_1^{n+1} \\ U_2^{n+1} \\ U_3^{n+1} \\ U_4^{n+1} \end{pmatrix} = \begin{pmatrix} U_1^n \\ U_2^n \\ U_3^n \\ U_4^n \end{pmatrix} + \begin{pmatrix} \underline{\hspace{2cm}} U_0^{n+1} \\ 0 \\ 0 \\ \underline{\hspace{2cm}} U_5^{n+1} \end{pmatrix}$$

3. At this point the structure of the coefficient matrix on the left and the vector sum on the right should be clear (even for more spatial points). It is time for us to start writing some code. we will start with the basic setup of the problem.

```

import numpy as np
import matplotlib.pyplot as plt

D = 1
x = # set up a linearly spaced spatial domain
t = # set up a linearly spaced temporal domain
dx = x[1]-x[0]
dt = t[1]-t[0]
a = D*dt/dx**2
IC = lambda x: # write a function for the initial condition
BCleft = lambda t: 0*t # left boundary condition
BCright = lambda t: 0*t # right boundary condition

U = np.zeros((len(t), len(x))) # set up a blank array for U
U[0,:] = IC(x) # set up the initial condition
U[:,0] = BCleft(t) # set up the left boundary condition
U[:,-1] = BCright(t) # set up the right boundary condition

```

4. Next we write a function that takes in the number of spatial points and returns the coefficient matrix for the linear system. Take note that the first and last rows take a little more care than the rest.

```

def coeffMatrix(M,a): # we are using M=len(x) as the first input
    A = np.zeros((M-2, M-2))
    # why are we using M-2 x M-2 for the size?
    A[0,0] = # top left entry
    A[0,1] = # entry in the first row second column
    A[-1,-1] = # bottom right entry
    A[-1,-2] = # entry in the last row second to last column
    for i in range(1,M-3): # now loop through all of the other rows
        A[i,i] = # entry on the main diagonal
        A[i,i-1] = # entry on the lower diagonal
        A[i,i+1] = # entry on the upper diagonal
    return A

A = coeffMatrix(len(x),a)
print(A)
plt.spy(A)
# spy is a handy plotting tool that shows the structure
# of a matrix (optional)
plt.show()

```

5. Next we write a loop that iteratively solves the system of equations for each new time step.

```

for n in range(len(t)-1):
    b1 = U[n,??]
    # b1 is a vector of U at step n for the inner spatial nodes
    b2 = np.zeros(length(b1)) # set up the second right-hand vector
    b2[0] = ???*BCleft(t[n+1]) # fill in the correct first entry
    b2[-1] = ???*BCright(t[n+1]) # fill in the correct last entry
    b = b1 + b2 # The vector "b" is the right side of the equation
    #
    # finally use a linear algebra solver to fill in the
    # inner spatial nodes at step n+1
    U[n+1,??] = ???

```

6. All of the hard work is now done. It remains to plot the solution. Try this method on several sets of initial and boundary conditions for the 1D heat equation. Be sure to demonstrate that the method is stable no matter the values of Δt and Δx .
 7. What are the primary advantages and disadvantages to the implicit method described in this problem?
-

Exercise 9.20 (The Crank-Nicolson Method). We conclude this section with one more implicit scheme: the **Crank-Nicolson Method**. In this method we take the average of the forward and backward difference schemes:

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = \frac{1}{2} \left[D \left(\frac{U_{i-1}^n - 2U_i^n + U_{i+1}^n}{\Delta x^2} \right) + D \left(\frac{U_{i-1}^{n+1} - 2U_i^{n+1} + U_{i+1}^{n+1}}{\Delta x^2} \right) \right].$$

Letting $r = D\Delta t/(2\Delta x^2)$ we can rearrange to get

$$U_{i-1}^{n+1} + U_i^{n+1} + U_{i+1}^{n+1} = U_{i-1}^n + U_i^n + U_{i+1}^n.$$

This can now be viewed as a system of equations. Let us build this system carefully and then write code to solve the heat equation from the previous problems with the Crank-Nicolson method. For this problem we will assume fixed Dirichlet boundary conditions on both the left- and right-hand sides of the domain.

1. First let us write the equations for several values of i .

$$(x_1) : U_0^{n+1} + U_1^{n+1} + U_2^{n+1} = U_0^n + U_1^n + U_2^n$$

$$(x_2) : U_1^{n+1} + U_2^{n+1} + U_3^{n+1} = U_1^n + U_2^n + U_3^n$$

$$(x_3) : U_2^{n+1} + U_3^{n+1} + U_4^{n+1} = U_2^n + U_3^n + U_4^n$$

$$\vdots \quad \vdots$$

$$(x_{M-2}) : U_{M-3}^{n+1} + U_{M-2}^{n+1} + U_{M-1}^{n+1} = U_{M-3}^n + U_{M-2}^n + U_{M-1}^n$$

where M is the number of spatial points (enumerated $x_0, x_1, x_2, \dots, x_{M-1}$).

2. The first and last equations can be simplified since we are assuming that we have Dirichlet boundary conditions. Therefore for x_1 we can rearrange to move the U_0^{n+1} term to the right-hand side since it is given for all time. Similarly for x_{M-2} we can move the U_{M-1}^{n+1} term to the right-hand side since it is fixed for all time. Rewrite these two equations.
3. Verify that the left-hand side of the equations that we have built in parts (1) and (2) can be written as the following matrix-vector product:

$$\begin{pmatrix} (1+2r) & -r & 0 & 0 & \cdots & 0 \\ -r & (1+2r) & -r & 0 & \cdots & 0 \\ 0 & -r & (1+2r) & -r & \cdots & 0 \\ \vdots & & & & & 0 \\ 0 & \cdots & & 0 & -r & (1+2r) \end{pmatrix} \begin{pmatrix} U_1^{n+1} \\ U_2^{n+1} \\ U_3^{n+1} \\ \vdots \\ U_{M-2}^{n+1} \end{pmatrix}$$

4. Verify that the right-hand side of the equations that we built in parts (1) and (2) can be written as

$$\begin{pmatrix} (1-2r) & r & 0 & 0 & \cdots & 0 \\ r & (1-2r) & r & 0 & \cdots & 0 \\ 0 & r & (1-2r) & r & & 0 \\ \vdots & & & & & r(1-2r) \end{pmatrix} \begin{pmatrix} U_1^n \\ U_2^n \\ U_3^n \\ \vdots \\ U_{M-2}^n \end{pmatrix} + \begin{pmatrix} r(U_0^{n+1} + U_0^n) \\ 0 \\ \vdots \\ 0 \\ r(U_{M-1}^n + U_{M-1}^{n+1}) \end{pmatrix}$$

5. Now for the wonderful part! The entire system of equations from part (a) can be written as

$$A\mathcal{U}^{n+1} = B\mathcal{U}^n + D.$$

What are the matrices A and B and what are the vectors \mathcal{U}^{n+1} , \mathcal{U}^n , and D ?

6. To solve for \mathcal{U}^{n+1} at each time step we simply need to do a linear solve:

$$\mathcal{U}^{n+1} = A^{-1}(B\mathcal{U}^n + D).$$

Of course, we will never do a matrix inverse on a computer. Instead we can lean on tools such as `np.linalg.solve()` to do the linear solve for us.

7. Finally. Write code to solve the 1D Heat Equation implementing the Crank Nicolson method described in this problem. The setup of your code should be largely the same as for the implicit method from Exercise 9.19. You will need to construct the matrices A and B as well as the vector D . Then your time stepping loop will contain the code from part 6 of this problem.
-

Exercise 9.21. To graphically show the Crank Nicolson method we can again use a finite difference stencil to show where the information is coming from and where it is going to. In Figure 9.3 notice that there are three points at the new time step that are used to calculate the value of U_i^{n+1} at the new time step. Sketch a similar image for the original implicit scheme from Exercise 9.19

It turns out that the error terms for the forward and backward difference methods have the form $C\Delta t + O(\Delta t^2)$ and $C\Delta t + O(\Delta t^2)$. Taking the average cancels the $\pm C\Delta t$ terms and leaves an error of order $O(\Delta t^2)$; in combination with the space variable, we have an error of order $O(\Delta t^2) + O(\Delta x^2)$ for the whole method, as compared with $O(\Delta t) + O(\Delta x^2)$ for the forward and backward difference methods. Like the backward difference method, the Crank-Nicolson method is absolutely stable.

9.2.6 Stability

While exploring the explicit finite-difference method for solving the 1d heat equation $u_t = Du_{xx}$ we encountered the stability condition

$$a = \frac{D\Delta t}{(\Delta x)^2} \leq \frac{1}{2}.$$

We now want to understand where this condition comes from.

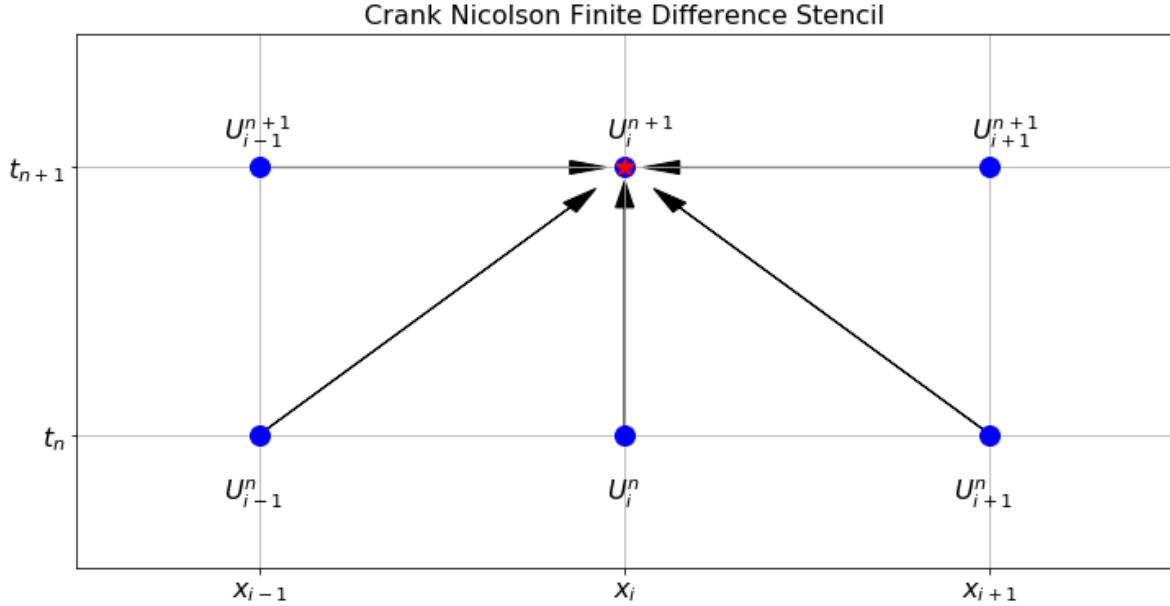


Figure 9.3: The finite difference stencil for the Crank Nicolson method.

We start by setting up the notation for solving the heat equation using the explicit finite-difference method. We discretise the spatial variable x into N intervals with grid points x_0, \dots, x_N with stepsize $\Delta x = (x_N - x_0)/N$. We discretise the time variable t into M intervals with grid points t_0, \dots, t_M with stepsize $\Delta t = (t_M - t_0)/M$. We denote the approximation of $u(t_n, x_i)$ by U_i^n . The initial condition sets $U_i^0 = u(t_0, x_i)$. We work with homogeneous Dirichlet boundary conditions, so $U_0^n = U_N^n = 0$ for all n .

The approximations at the remaining points is then calculated by the formula

$$U_i^{n+1} = U_i^n + a(U_{i+1}^n - 2U_i^n + U_{i-1}^n)$$

for $i = 1, \dots, N - 1$. You derived this in Section 9.2.1 using the finite-difference formulae for the derivatives from Chapter 5. We rewrite this in matrix notation:

$$\begin{pmatrix} U_1^{n+1} \\ \vdots \\ U_{N-1}^{n+1} \end{pmatrix} = \begin{pmatrix} 1 - 2a & a & & & \\ a & 1 - 2a & a & & \\ & \ddots & \ddots & \ddots & \\ & & a & 1 - 2a & \end{pmatrix} \begin{pmatrix} U_1^n \\ \vdots \\ U_{N-1}^n \end{pmatrix}.$$

The matrix is tridiagonal, with $1 - 2a$ on the diagonal and a on the two off-diagonals. We denote the matrix by A and the two vectors by \mathbf{U}_{n+1} and \mathbf{U}_n respectively. So we have the formula

$$\mathbf{U}_{n+1} = A\mathbf{U}_n.$$

The solution at time t_n is then given by

$$\mathbf{U}_n = A^n \mathbf{U}_0.$$

We want to understand how errors evolve over time. If errors grow exponentially over time then we call the method unstable and the method is not useful.

Let us assume that at some step, for convenience let us choose step 0, an error ϵ is introduced:

$$\tilde{\mathbf{U}}_0 = \mathbf{U}_0 + \epsilon.$$

Then after n steps we have

$$\tilde{\mathbf{U}}_n = A^n \tilde{\mathbf{U}}_0 = A^n(\mathbf{U}_0 + \epsilon) = A^n \mathbf{U}_0 + A^n \epsilon = \mathbf{U}_n + A^n \epsilon.$$

So the error at time t_n is

$$e_n = A^n \epsilon.$$

To see if the error grows exponentially over time, we expand the initial error in terms of the eigenvectors of the matrix A :

$$e_n = \sum_i \epsilon_i \mathbf{v}_i,$$

where

$$A\mathbf{v}_i = \lambda_i \mathbf{v}_i$$

and the sum is over all eigenvectors \mathbf{v}_i of A . Then

$$e_n = \sum_i \epsilon_i \lambda_i^n \mathbf{v}_i.$$

This shows that if all eigenvalues have absolute value less than 1, then the method is stable. If at least one eigenvalue has absolute value greater than 1, then the corresponding component of the error will grow exponentially with time and the method is unstable.

The eigenvalues of the matrix A are the roots of the characteristic polynomial

$$\det(A - \lambda I) = 0.$$

There is a nice method to determine the eigenvalues of the matrix A , using Fourier analysis. We will not discuss this in this module and instead just give the result and then look at a simple example. The result (which you do not need to remember) is that the eigenvalues of the matrix A are given by

$$\lambda_k = 1 - 4a \left(\sin \left(\frac{k\pi}{2N} \right) \right)^2$$

for $k = 1, \dots, N-1$. For stability we need $|\lambda_k| < 1$ for all k , i.e.,

$$0 \leq a \left(\sin \left(\frac{k\pi}{2N} \right) \right)^2 \leq \frac{1}{2}$$

for all k . The most stringent condition is that coming from $k = N - 1$, so the stability condition is

$$a \leq \frac{1}{2} \left(\sin \left(\frac{(N-1)\pi}{2N} \right) \right)^{-2}.$$

In the limit $N \rightarrow \infty$ this gives the condition $a \leq 1/2$.

Example 9.1. Consider the heat equation on the spatial domain $x \in [0, 1]$ and divide this into three subintervals, so that our spatial grid consists of $x_0 = 0, x_1 = 1/3, x_2 = 2/3$ and $x_3 = 1$. The matrix A is then

$$A = \begin{pmatrix} 1-2a & a \\ a & 1-2a \end{pmatrix}.$$

The characteristic polynomial is

$$\begin{aligned} \det(A - \lambda I) &= \begin{vmatrix} 1-2a-\lambda & a \\ a & 1-2a-\lambda \end{vmatrix} \\ &= (1-2a-\lambda)^2 - a^2 \\ &= \lambda^2 - 2(1-2a)\lambda + (1-2a)^2 - a^2. \end{aligned}$$

The roots of this polynomial are

$$\lambda_{\pm} = 1 - 2a \pm a.$$

We need both of these to have a magnitude less than 1 for the method to be stable. This gives us an upper bound on the allowed a . The eigenvalue whose magnitude will increase above 1 first as a increases is $\lambda_- = 1 - 3a$, which has magnitude 1 when $a = 2/3$. So the stability condition is $a \leq 2/3$.

9.3 The Wave Equation

Any material below this point is optional and will not be assessed.

The problems that we have dealt with thus far all model natural diffusion processes: heat transport, molecular diffusion, etc. Another interesting physical phenomenon is that of wave propagation. The 1D *wave equation* is

$$u_{tt} = c^2 u_{xx}$$

where c is a parameter modelling the stiffness of the medium the wave is travelling through. With homogeneous Dirichlet boundary conditions we can think of this as the behaviour of a guitar string after it has been plucked. If the boundaries are in motion then the model might be of someone wiggling a taunted string from one end.

Exercise 9.22. Let us write code to numerically solve the 1D wave equation. As before, we use the notation U_i^n to represent the approximate solution $u(t, x)$ at the point $t = t_n$ and $x = x_i$.

1. Give a reasonable discretization of the second derivative in time:

$$u_{tt}(t_n, x_i) \approx \underline{\hspace{2cm}}.$$

2. Give a reasonable discretization of the second derivative in space:

$$u_{xx}(t_n, x_i) \approx \underline{\hspace{2cm}}.$$

3. Put your answers to parts (a) and (b) together with the wave equation to get

$$\frac{??? - ??? + ???}{\Delta t^2} = c^2 \frac{??? - ??? + ???}{\Delta x^2}.$$

4. Solve the equation from part 3 for U_i^{n+1} . The resulting difference equation is the finite difference scheme for the 1D wave equation.
5. You should notice that the finite difference scheme for the wave equation references two different times: U_i^n and U_i^{n-1} . Based on this observation, what information do we need to in order to actually start our numerical solution?
6. Consider the wave equation $u_{tt} = 2u_{xx}$ in $x \in (0, 1)$ with $u(0, x) = 4x(1-x)$, $u_t(0, x) = 0$, and $u(t, 0) = u(t, 1) = 0$. Use the finite difference scheme that you built in this problem to approximate the solution to this PDE.

Figure 9.4 shows the finite difference stencil for the 1D wave equation. Notice that we need two prior time steps in order to advance to the new time step. This means that in order to start the finite difference scheme for the wave equation we need to have information about time t_0 and also time t_1 . We get this information by using the two initial conditions $u(0, x)$ and $u_t(0, x)$.

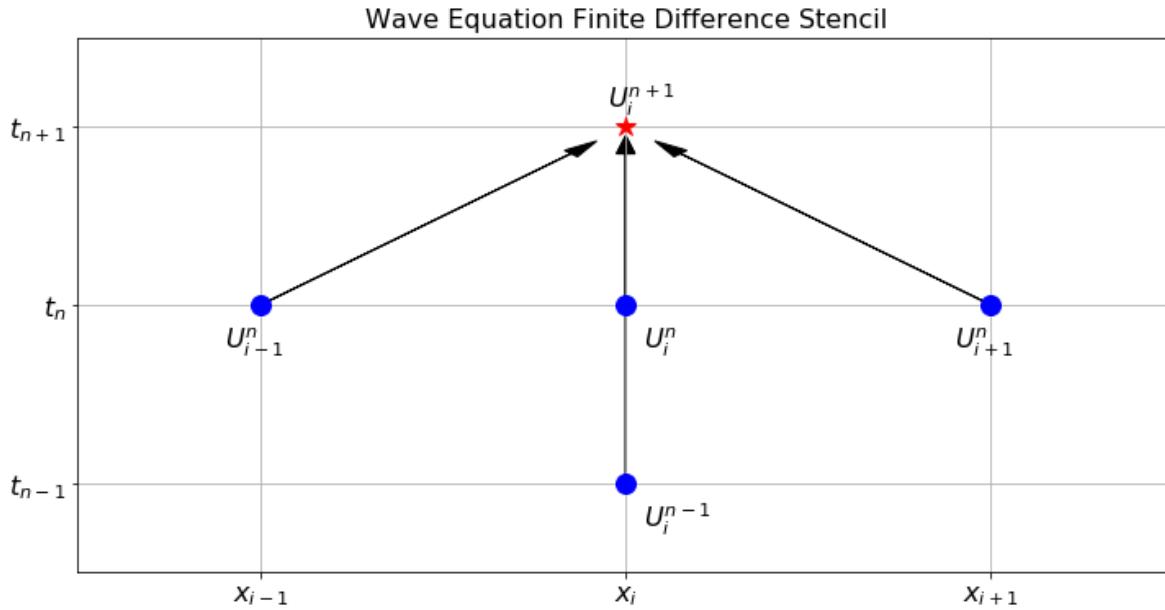


Figure 9.4: The finite difference stencil for the 1D wave equation.

Exercise 9.23. The ratio $c^2\Delta t^2/\Delta x^2$ shows up explicitly in the finite difference scheme for the 1D wave equation. Just like in the heat equation, this parameter controls when the finite difference solution will be stable. Experiment with your finite difference solution and conjecture a value of $a = c^2\Delta t^2/\Delta x^2$ which divides the regions of stability versus instability. Your answer should be in the form:

If $a = c^2\Delta t^2/\Delta x^2 < \underline{\hspace{2cm}}$ then the finite difference scheme for the 1D wave equation will be stable. Otherwise it will be unstable.

Exercise 9.24. Show several plots demonstrating what occurs to the finite difference solution of the wave equation when the parameters are in the unstable region and right on the edge of the unstable region.

Exercise 9.25. What is the expected error in the finite difference scheme for the 1D wave equation? What does this mean in plain English?

Exercise 9.26. Use your finite difference code to solve the 1D wave equation

$$u_{tt} = c^2 u_{xx}$$

with boundary conditions $u(t, 0) = u(t, 1) = 0$, initial condition $u(0, x) = 4x(1 - x)$, and zero initial velocity. Experiment with different values of c^2 . What does the parameter c do to the wave? Give a physical interpretation of c .

Exercise 9.27. Solve the 1D wave equation

$$u_{tt} = u_{xx}$$

with Dirichlet boundary conditions $u(t, 0) = 0.4 \sin(\pi t)$ and $u(t, 1) = 0$ along with initial condition $u(0, x) = 0$ and zero initial velocity. This time the left-hand boundary is being controlled externally and the string starts off at equilibrium. Give a physical situation where this sort of setup might arise. Then modify your solution so that both sides of the string are being wiggled at different frequencies.

Exercise 9.28. Now consider the 2D wave equation

$$u_{tt} = c^2 (u_{xx} + u_{yy}).$$

We want to build a numerical solution to this new PDE. Just like with the 2D heat equation we propose the notation $U_{i,j}^n$ for the approximation of the function $u(t, x, y)$ at the point $t = t_n$, $x = x_i$, and $y = y_j$.

1. Give discretizations of the derivatives u_{tt} , u_{xx} , and u_{yy} .
 2. Substitute your discretizations into the 2D wave equation, make the simplifying assumption that $\Delta x = \Delta y$, and solve for $U_{i,j}^{n+1}$. This is the finite difference scheme for the 2D wave equation.
 3. Write code to implement the finite difference scheme from part 2 on the domain $(x, y) \in (0, 1) \times (0, 1)$ with homogeneous Dirichlet boundary conditions, initial condition $u(0, x, y) = \sin(2\pi(x - 0.5)) \sin(2\pi(y - 0.5))$, and zero initial velocity.
 4. Draw the finite difference stencil for the 2D heat equation.
-

Exercise 9.29. What is the region of stability for the finite difference scheme on the 2D wave equation? Produce several plots showing what happens when we are in the unstable region as well as when we are right on the edge of the stable region.

Exercise 9.30. Solve the 2D wave equation on the unit square with u starting at rest and being driven by a wave coming in from one boundary.

9.4 The Travelling Wave Equation

Now we turn our attention to a new PDE: the transport equation

$$u_t + vu_x = 0.$$

In this equation $u(t, x)$ is the height of a wave at time t and spatial location x . The parameter v is the velocity of the wave. Imagine this as sending a single solitary wave pulsing down a taught rope or as sending a single pulse of light down a fibre optic cable.

Exercise 9.31. Consider the PDE $u_t + vu_x = 0$. There is a very easy way to get an analytic solution to this equation that describes a travelling wave. If we have the initial condition $u(0, x) = f(x) = e^{-(x-4)^2}$ then we claim that $u(t, x) = f(x - vt)$ is an analytic solution to the PDE. More explicitly, we are claiming that

$$u(t, x) = e^{-(x-vt-4)^2}$$

is the analytic solution to the PDE. Let us prove this.

1. Take the t derivative of $u(t, x)$.
 2. Take the x derivative of $u(t, x)$.
 3. The PDE claims that $u_t + vu_x = 0$. Verify that this equal sign is indeed true.
-

Exercise 9.32. Now we would like to visualize the solution to the PDE from the previous exercise. The Python code below gives an interactive visual of the solution. Experiment with different values of v and different initial conditions.

```

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation, rc
from IPython.display import HTML

v = 1
f = lambda x: np.exp(-(x-4)**2)
u = lambda t, x: f(x - v*t)
x = np.linspace(0,10,101)
t = np.linspace(0,10,101)

fig, ax = plt.subplots()
plt.close()
ax.grid()
ax.set_xlabel('x')
ax.set_xlim(( 0, 10))
ax.set_ylim(( -0.1, 1))
frame, = ax.plot([], [], linewidth=2, linestyle='--')

def animator(N):
    ax.set_title(f"Time = {t[N]:.2f}")
    frame.set_data(x,???) # plot the correct time step for u(t,x)
    return (frame,)

PlotFrames = range(0,len(t),1)
anim = animation.FuncAnimation(fig,
                               animator,
                               frames=PlotFrames,
                               interval=100,
                               )

rc('animation', html='jshtml') # embed in the HTML for Google Colab
anim

```

Exercise 9.33. Use the chain rule to prove that for any differentiable function $f(x)$ the function $u(t, x) = f(x - vt)$ is an analytic solution to the transport equation $u_t + vu_x = 0$ with initial condition $u(0, x) = f(x)$.

Thus the travelling wave equation $u_t + vu_x = 0$ has a very nice analytic solution which we can always find. Therefore there is no need to ever find a numerical solution – we can just write down the analytic solution if we are given the initial condition. As it turns out though, the numerical solutions exhibit some very interesting behaviour.

Exercise 9.34. Consider the travelling wave equation $u_t + vu_x = 0$ with initial condition $u(0, x) = f(x)$ for some given function f and boundary condition $u(t, 0) = 0$. To build a numerical solution we will again adopt the notation U_i^n for the approximation to $u(t, x)$ at the point $t = t_n$ and $x = x_i$.

- (a) Write an approximation of u_t using U_i^{n+1} and U_i^n .
 - (b) Write an approximation of u_x using U_{i+1}^n and U_i^n .
 - (c) Substitute your answers from parts (a) and (b) into the travelling wave equation and solve for U_i^{n+1} . This is our first finite difference scheme for the travelling wave equation.
 - (d) Write Python code to get the finite difference approximation of the solution to the PDE. Plot your finite difference solution on top of the analytic solution for $f(x) = e^{-(x-4)^2}$. What do you notice? Can you stabilize this method by changing the values of Δt and Δx like with did with the heat and wave equations?
-

The finite difference scheme that you built in the previous exercise is called the downwind scheme for the travelling wave equation. Figure 9.5 shows the finite difference stencil for this scheme. We call this scheme “downwind” since we expect the wave to travel from left to right and we can think of a fictitious wind blowing the solution from left to right. Notice that we are using information from “downwind” of the point at the new time step.

Exercise 9.35. You should have noticed in the previous exercise that you cannot reasonably stabilize the finite difference scheme. Propose several reasons why this method appears to be unstable no matter what you use for the ratio $v\Delta t/\Delta x$.

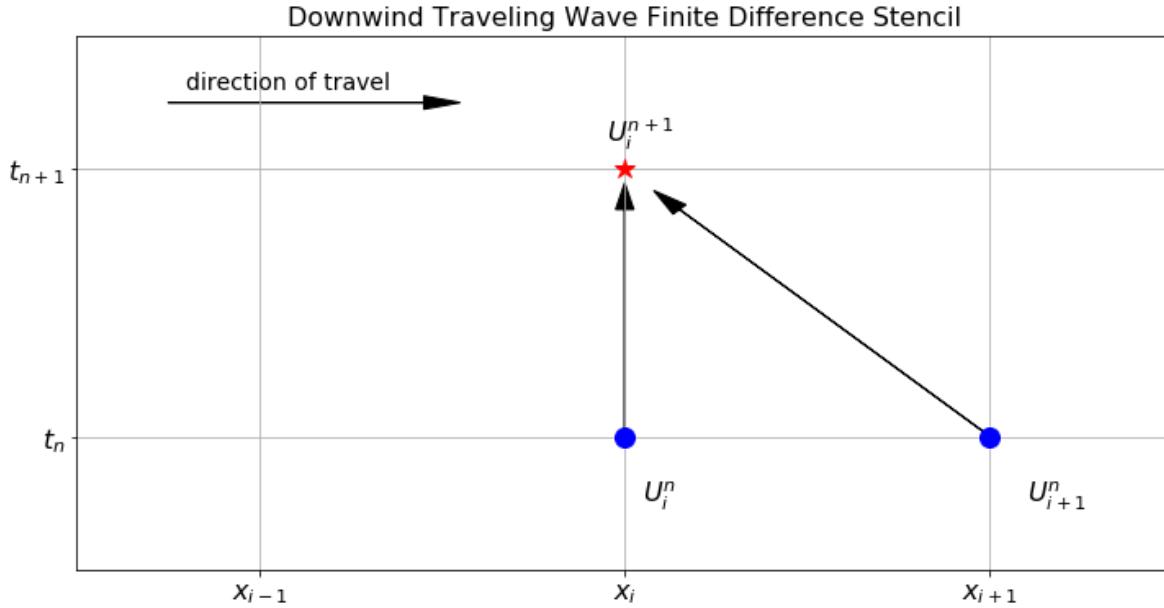


Figure 9.5: The finite difference stencil for the 1D downwind scheme on the traveling wave equation.

Exercise 9.36. One of the troubles with the finite difference scheme that we have built for the travelling wave equation is that we are using the information at our present spatial location and the next spatial location to the right to propagate the solution forward in time. The trouble here is that the wave is moving from left to right, so the interesting information about the next time step's solution is actually coming from the left, not the right. We call this “looking upwind” since you can think of a fictitious *wind* blowing from left to right, and we need to look “upwind” to see what is coming at us. If we write the spatial derivative as

$$u_x \approx \frac{U_i^n - U_{i-1}^n}{\Delta x}$$

we still have a first-order approximation of the derivative but we are now looking left instead of right for our spatial derivative. Make this modification in your finite difference code for the travelling wave equation (call it the “upwind method”). Approximate the solution to the same PDE as we worked with in the previous exercises. What do you notice now?

Figure 9.6 shows the finite difference stencil for the upwind scheme. We call this scheme “up” since we expect the wave to travel from left to right and we can think of a fictitious wind blowing the solution from left to right. Notice that we are using information from “upwind” of the point at the new time step.

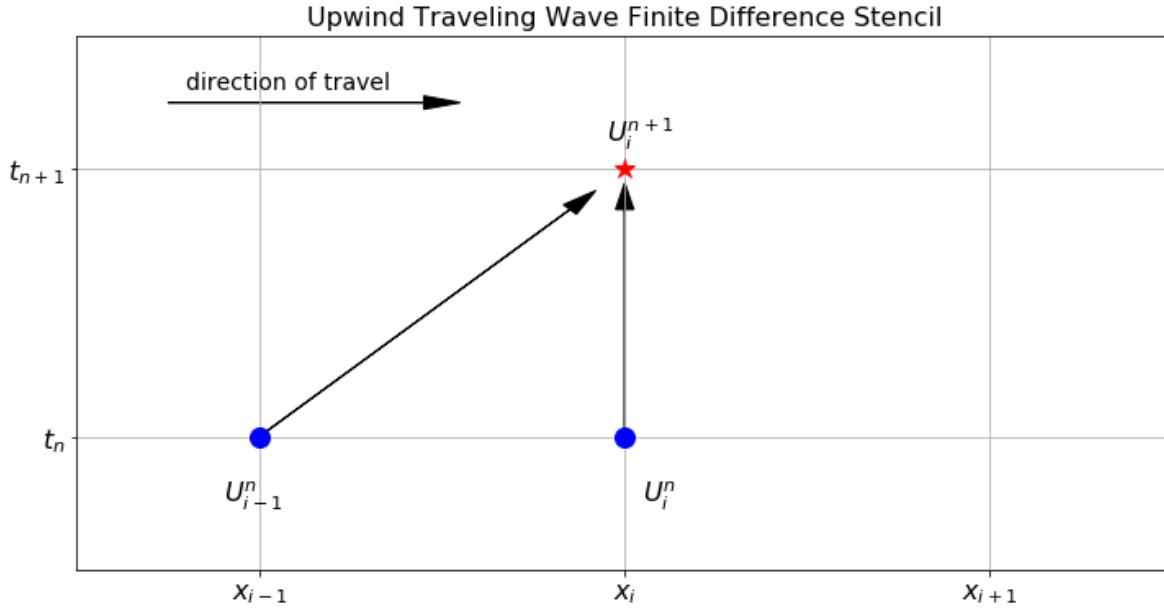


Figure 9.6: The finite difference stencil for the 1D downwind scheme on the traveling wave equation.

Exercise 9.37. Complete the following sentences:

1. In the downwind finite difference scheme for the travelling wave equation, the approximate solution moves at the correct speed, but ...
 2. In the upwind finite difference scheme for the travelling wave equation, the approximate solution moves at the correct speed, but ...
-

Exercise 9.38. Neither the downwind nor the upwind solutions for the travelling wave equation are satisfactory. They completely miss the interesting dynamics of the analytic solution to the PDE. Some ideas for stabilizing the finite difference solution for the travelling wave equation are as follows. Implement each of these ideas and discuss pros and cons of each. Also draw a finite difference stencil for each of these methods.

1. Perhaps one of the issues is that we are using first-order methods to approximate u_t and u_x . What if we used a second-order approximation for these first derivatives

$$u_t \approx \frac{U_i^{n+1} - U_i^{n-1}}{2\Delta t} \quad \text{and} \quad u_x \approx \frac{U_{i+1}^n - U_{i-1}^n}{2\Delta x}?$$

Solve for U_i^{n+1} and implement this method. This is called the **leapfrog method**.

2. For this next method let us stick with the second-order approximation of u_x but we will do something clever for u_t . For the time derivative we originally used

$$u_t \approx \frac{U_i^{n+1} - U_i^n}{\Delta t}$$

what happens if we replace U_i^n with the average value from the two surrounding spatial points

$$U_i^n \approx \frac{1}{2} (U_{i+1}^n + U_{i-1}^n) ?$$

This would make our approximation of the time derivative

$$u_t \approx \frac{U_i^{n+1} - \frac{1}{2} (U_{i+1}^n + U_{i-1}^n)}{\Delta t}.$$

Solve this modified finite difference equation for U_i^{n+1} and implement this method. This is called the **Lax-Friedrichs method**.

3. Finally we will do something very clever (and very counter intuitive). What if we inserted some artificial diffusion into the problem? You know from your work with the heat equation that diffusion spreads a solution out. The downwind scheme seemed to have the issue that it was *bunching up* at the beginning and end of the wave, so artificial diffusion might smooth this out. The **Lax-Wendroff method** does exactly that: take a regular Euler-type step in time

$$u_t \approx \frac{U_i^{n+1} - U_i^n}{\Delta t},$$

use a second-order centred difference scheme in space to approximate u_x

$$u_x \approx \frac{U_{i+1}^n - U_{i-1}^n}{2\Delta x},$$

but add on the term

$$\left(\frac{v^2 \Delta t^2}{2\Delta x^2} \right) (U_{i-1}^n - 2U_i^n + U_{i+1}^n)$$

to the right-hand side of the equation. Notice that this new term is a scalar multiple of the second-order approximation of the second derivative u_{xx} . Solve this equation for U_i^{n+1} and implement the Lax-Wendroff method.

9.5 The Laplace and Poisson Equations

Exercise 9.39. Consider the 1D heat equation $u_t = 1u_{xx}$ with boundary conditions $u(t, 0) = 0$ and $u(t, 1) = 1$ and initial condition $u(0, x) = 0$.

1. Describe the physical setup for this problem.
 2. Recall that the solution to a differential equation reaches a steady state (or equilibrium) when the time rate of change is zero. Based on the physical system, what is the steady state heat profile for this PDE?
 3. Use your 1D heat equation code to show the full time evolution of this PDE. Run the simulation long enough so that you see the steady state heat profile.
-

Exercise 9.40. Now consider the forced 1D heat equation $u_t = u_{xx} + e^{-(x-0.5)^2}$ with the same boundary and initial conditions as the previous exercise. The exponential forcing function introduced in this equation is an external source of heat (like a flame held to the middle of the metal rod).

1. Conjecture what the steady state heat profile will look like for this particular setup. Be able to defend your answer.
 2. Modify your 1D heat equation code to show the full time evolution of this PDE. Run the simulation long enough so that you see the steady state heat profile.
-

Exercise 9.41. Next we will examine 2D steady state heat profiles. Consider the PDE $u_t = u_{xx} + u_{yy}$ with boundary conditions $u(t, 0, y) = u(t, x, 0) = u(t, x, 1) = 0$ and $u(t, 1, y) = 1$ with initial condition $u(0, x, y) = 0$.

1. Describe the physical setup for this problem.
 2. Based on the physical system, describe the steady state heat profile for this PDE. Be sure that your steady state solution still satisfies the boundary conditions.
 3. Use your 2D heat equation code to show the full time evolution of this PDE. Run the simulation long enough so that you see the steady state heat profile.
-

Exercise 9.42. Now consider the forced 2D heat equation $u_t = u_{xx} + u_{yy} + 10e^{-(x-0.5)^2 - (y-0.5)^2}$ with the same boundary and initial conditions as the previous exercise. The exponential forcing function introduced in this equation is an external source of heat (like a flame held to the middle of the metal sheet).

1. Conjecture what the steady state heat profile will look like for this particular setup. Be able to defend your answer.
 2. Modify your 2D heat equation code to show the full time evolution of this PDE. Run the simulation long enough so that you see the steady state heat profile.
-

Up to this point we have studied PDEs that all depend on time. In many applications, however, we are not interested in the transient (time dependent) behaviour of a system. Instead we are often interested in the steady state solution when the forces in question are in static equilibrium. Two very famous time-independent PDEs are the Laplace Equation

$$u_{xx} + u_{yy} + u_{zz} = 0$$

and the Poisson equation

$$u_{xx} + u_{yy} + u_{zz} = f(x, y, z).$$

Notice that both the Laplace and Poisson equations are the equations that we get when we consider the limit $u_t \rightarrow 0$. In the limit when the time rate of change goes to zero we are actually just looking at the eventual steady state heat profile resulting from the initial and boundary conditions of the heat equation. In the previous exercises you already wrote code that will show the steady state profiles in a few setups. The trouble with the approach of letting the time-dependent simulation run for a *long time* is that the finite difference solution for the heat equation is known to have stability issues. Moreover, it may take a lot of computational time for the solution to reach the eventual steady state. In the remainder of this section we look at methods of solving for the steady state directly – without examining any of the transient behaviour. We will first examine a 1D version of the Laplace and Poisson equations.

Exercise 9.43. Consider a 1-dimensional rod that is infinitely thin and has unit length. For the sake of simplicity assume the following:

- the specific heat of the rod is exactly 1 for the entire length of the rod,
- the temperature of the left end is held fixed at $u(0) = 0$,
- the temperature of the right end is held fixed at $u(1) = 1$, and
- the temperature has reached a steady state.

You can assume that the temperatures are *reference temperatures* instead of absolute temperatures, so a temperature of “0” might represent room temperature.

Since there are no external sources of heat we model the steady-state heat profile we must have $u_t = 0$ in the heat equation. Thus the heat equation collapses to $u_{xx} = 0$. This is exactly the one dimensional Laplace equation.

- (a) To get an exact solution of the Laplace equation in this situation we simply need to integrate twice. Do the integration and write the analytic solution (there should be no surprises here).
- (b) To get a numerical solution we first need to partition the domain into finitely many point. For the sake of simplicity let us say that we subdivide the interval into 5 equal sub intervals (so there are 6 points including the endpoints). Furthermore, we know that we can approximate u_{xx} as

$$u_{xx} \approx \frac{U_{i+1} - 2U_i + U_{i-1}}{\Delta x^2}.$$

Thus we have 6 linear equations:

$$\begin{aligned} U_0 &= 1 \quad (\text{left boundary condition}) \\ \frac{U_2 - 2U_1 + U_0}{\Delta x^2} &= 0 \\ \frac{U_3 - 2U_2 + U_1}{\Delta x^2} &= 0 \\ \frac{U_4 - 2U_3 + U_2}{\Delta x^2} &= 0 \\ \frac{U_5 - 2U_4 + U_3}{\Delta x^2} &= 0 \\ U_5 &= 0 \quad (\text{right boundary condition}). \end{aligned}$$

Notice that there are really only four unknowns since the boundary conditions dictate two of the temperature values. Rearrange this system of equations into a matrix equation and solve for the unknowns U_1 , U_2 , U_3 , and U_4 . Your coefficient matrix should be 4×4 .

- (c) Compare your answers from parts (a) and (b).
 - (d) Write code to build the numerical solution with an arbitrary value for Δx (i.e. an arbitrary number of sub intervals). You should build the linear system automatically in your code.
-

Solving the 1D Laplace equation with Dirichlet boundary conditions is rather uninteresting since the answer will always be a linear function connecting the two boundary conditions. The Poisson equation $u_{xx} = f(x)$ is more interesting than the Laplace equation in 1D. The function

$f(x)$ is called a forcing function. You can think of it this way: if u is the amount of force on a linear bridge, then f might be a function that gives the distribution of the forces on the bridge due to the cars sitting on the bridge. In terms of heat we can think of this as an external source of heat energy warming up the one-dimensional rod somewhere in the middle (like a flame being held to one place on the rod).

Exercise 9.44. How would we analytically solve the Poisson equation $u_{xx} = f(x)$ in one spatial dimension? As a sample problem consider $x \in [0, 1]$, the forcing function $f(x) = 5 \sin(2\pi x)$ and boundary conditions $u(0) = 2$ and $u(1) = 0.5$. Of course you need to check your answer by taking two derivatives and making sure that the second derivative exactly matches $f(x)$. Also be sure that your solution matches the boundary conditions exactly.

Exercise 9.45. Now we can solve the Poisson equation from the previous problem numerically. Let us again build this with a partition that contains only 6 points just like we did with the Laplace equation a few exercise ago. We know the approximation for u_{xx} so we have the linear system

$$\begin{aligned} U_0 &= 2 \quad (\text{left boundary condition}) \\ \frac{U_2 - 2U_1 + U_0}{\Delta x^2} &= f(x_1) \\ \frac{U_3 - 2U_2 + U_1}{\Delta x^2} &= f(x_2) \\ \frac{U_4 - 2U_3 + U_2}{\Delta x^2} &= f(x_3) \\ \frac{U_5 - 2U_4 + U_3}{\Delta x^2} &= f(x_4) \\ U_5 &= 0.5 \quad (\text{right boundary condition}). \end{aligned}$$

- (a) Rearrange the system of equations as a matrix equation and then solve the system for U_1, U_2, U_3 , and U_4 . There are really only four equations so your matrix should be 4×4 .
 - (b) Compare your solution from part (a) to the function values that you found in the previous exercise.
 - (c) Now generalize the process of solving the 1D Poisson equation for an arbitrary value of Δx . You will need to build the matrix and the right-hand side in your code. Test your code on new forcing functions and new boundary conditions.
-

Exercise 9.46. The previous exercises only account for Dirichlet boundary conditions (fixed boundary conditions). We would now like to modify our Poisson solution to allow for a Neumann condition: where we know the derivative of u at one of the boundaries. The statement of the problem is as follows:

Solve: $u_{xx} = f(x)$ on $x \in (0, 1)$ with $u_x(0) = \alpha$ and $u(1) = \beta$.

The derivative condition on the boundary can be approximated by using a first-order approximation of the derivative, and as a consequence we have one new equation. Specifically, if we know that $u_x(0) = \alpha$ then we can approximate this condition as

$$\frac{U_1 - U_0}{\Delta x} = \alpha,$$

and we simply need to add this equation to the system that we were solving in the previous exercise. If we go back to our example of a partition with 6 points the system becomes

$$\begin{aligned} \frac{U_1 - U_0}{\Delta x} &= \alpha \quad (\text{left boundary condition}) \\ \frac{U_2 - 2U_1 + U_0}{\Delta x^2} &= f(x_1) \\ \frac{U_3 - 2U_2 + U_1}{\Delta x^2} &= f(x_2) \\ \frac{U_4 - 2U_3 + U_2}{\Delta x^2} &= f(x_3) \\ \frac{U_5 - 2U_4 + U_3}{\Delta x^2} &= f(x_4) \\ U_5 &= \beta \quad (\text{right boundary condition}). \end{aligned}$$

There are 5 equations this time.

- (a) With a 6 point grid solve the Poisson equation $u_{xx} = 5 \sin(2\pi x)$ with $u_x(0) = 0$ and $u(1) = 3$.
 - (b) Modify your code from part (a) to solve the same problem but with a much smaller value of Δx . You will need to build the matrix equation in your code.
-

Exercise 9.47 (The 2D Poisson Equation). We conclude this section, and chapter, by examining the two dimensional Poisson equations. As a sample problem, we want to solve the Poisson equation

$$u_{xx} + u_{yy} = f(x, y)$$

on the domain $(x, y) \in (0, 1) \times (0, 1)$ with homogeneous Dirichlet boundary conditions and forcing function

$$f(x, y) = -20\exp\left(-\frac{(x - 0.5)^2 + (y - 0.5)^2}{0.05}\right)$$

numerically.

We are going to start with a 6×6 grid of points and explicitly write down all of the equations. In Figure 9.7 the red stars represent boundary points where the value of $u(x, y)$ is known and the blue interior points are the ones where $u(x, y)$ is yet unknown. It should be clear that we should have two indices for each point (one for the x position and one for the y position), but it should also be clear that this will cause problems when writing down the resulting system of equations as a matrix equation (stop and think carefully about this). Therefore, in Figure 9.7 we propose an index, k , starting at the top left of the unknown nodes and reading left to right (just like we do with Python arrays).

- (a) Start by discretizing the 2D Poisson equation $u_{xx} + u_{yy} = f(x, y)$. For simplicity we assume that $\Delta x = \Delta y$ so that we can combine like terms from the x derivative and the y derivative. Fill in the missing coefficients and indices below.

$$U_{i+1,j} + U_{i,j-1} - (\underline{\quad})U_{\underline{\quad},\underline{\quad}} + U_{\underline{\quad},\underline{\quad}} + U_{\underline{\quad},\underline{\quad}} = \Delta x^2 f(x_i, y_i)$$

- (b) In Figure 9.7 we see that there are 16 total equations resulting from the discretization of the Poisson equation. Your first task is to write all 16 of these equations. we will get you started:

$$k = 0: \quad U_{k=1} + \textcolor{red}{U_{i=1,j=0}} - 4U_{k=0} + \textcolor{red}{U_{i=0,j=1}} + U_{k=4} = \Delta x^2 f(x_1, y_1)$$

$$k = 1: \quad U_{k=2} + U_{k=0} - 4U_{k=1} + \textcolor{red}{U_{i=0,j=2}} + U_{k=5} = \Delta x^2 f(x_1, y_2)$$

⋮

$$k = 15: \quad \textcolor{red}{U_{i=4,j=5}} + U_{k=14} - 4U_{k=15} + U_{k=11} + \textcolor{red}{U_{i=5,j=4}} = \Delta x^2 f(x_4, y_4)$$

In this particular example we have homogeneous Dirichlet boundary conditions so all of the boundary values are zero. If this was not the case then every boundary value would need to be moved to the right-hand sides of the equations.

- (c) We now have a 16×16 matrix equation to write based on the equations from part (b). Each row and column of the matrix equation is indexed by k . The coefficient matrix A is started for you below. Write the whole thing out and fill in the blanks. Notice that this matrix has a much more complicated structure than the coefficient matrix in the 1D

Poisson and Laplace equations.

$$A = \begin{pmatrix} -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & & \\ 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & \ddots & \\ 0 & & & & & & & & & \\ \vdots & & & & & & & & & \\ & & & & & & & & & -4 \end{pmatrix}$$

- (d) In the coefficient matrix from part (c) notice that the small matrix

$$\begin{pmatrix} -4 & 1 & 0 & 0 \\ 1 & -4 & 1 & 0 \\ 0 & 1 & -4 & 1 \\ 0 & 0 & 1 & -4 \end{pmatrix}$$

shows up in blocks along the main diagonal. If you have a hard copy of the matrix go back and draw a box around these blocks in the coefficient matrix. Also notice that there are diagonal bands of 1^s . Discuss the following:

1. Why are the blocks 4×4 ?
 2. How could you have predicted the location of the diagonal bands of 1^s ?
 3. What would the structure of the matrix look like if we partitioned the domain into a 10×10 grid of points instead of a 6×6 grid (including the boundary points)?
 4. Why is it helpful to notice this structure?
- (e) The right-hand side of the matrix equation resulting the your system of equations from part (b) is

$$b = \Delta x^2 \begin{pmatrix} f(x_1, y_1) \\ f(x_1, y_2) \\ f(x_1, y_3) \\ f(x_1, y_4) \\ f(x_2, y_1) \\ f(x_2, y_2) \\ \vdots \\ f(x_4, y_4) \end{pmatrix}.$$

Notice the structure of this vector. Why is it structured this way? Why is it useful to notice this?

- (f) Write Python code to solve the problem at hand. Recall that $f(x, y) = -20 \exp\left(-\frac{(x-0.5)^2 + (y-0.5)^2}{0.05}\right)$.

Show a contour plot of your solution. This will take a little work changing the indices back from k to i and j . Think carefully about how you want to code this before you put fingers to keyboard. You might want to use the `np.block()` command to build the coefficient matrix efficiently or you can use loops with carefully chosen indices.

- (g) (Challenge) Generalize your code to solve the Poisson equation with a much smaller value of $\Delta x = \Delta y$.
- (h) One more significant observation should be made about the 2D Poisson equation on this square domain. Notice that the corner points of the domain (e.g. $i = 0, j = 0$ or $i = 5, j = 0$) are never included in the system of equations. What does this mean about trying to enforce boundary conditions that only apply at the corners?

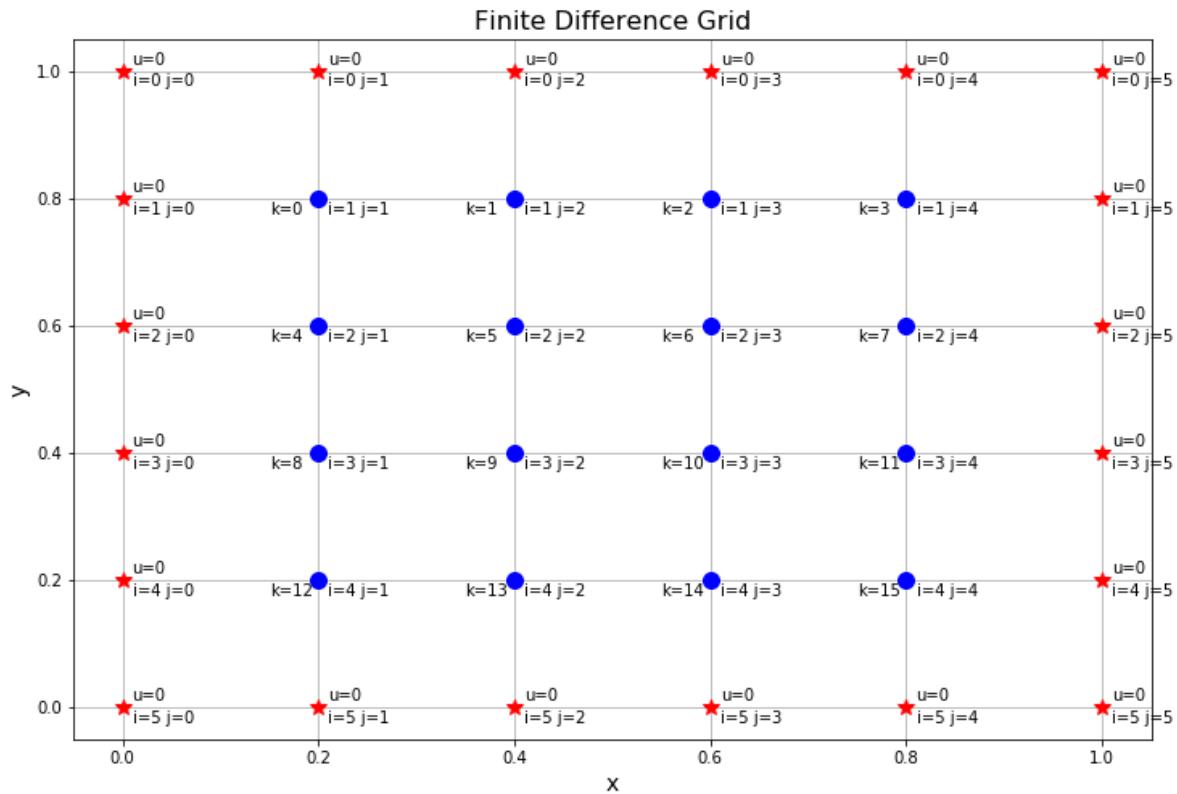


Figure 9.7: A finite difference grid for the Poisson equation with 6 grid points in each direction.

9.6 Algorithm Summaries

Exercise 9.48. Show the full mathematical details for building a first-order in time and second-order in space approximation method for the one-dimensional heat equation. Explain what the order of the error means in this context

Exercise 9.49. Show the full mathematical details for building a second-order in time and second-order in space approximation method for the one-dimensional wave equation. Explain what the order of the error means in this context

Exercise 9.50. Show the full mathematical details for building a first-order in time and second-order in space approximation method for the two-dimensional heat equation. Explain what the order of the error means in this context

Exercise 9.51. Show the full mathematical details for building a second-order in time and second-order in space approximation method for the two-dimensional wave equation. Explain what the order of the error means in this context

Exercise 9.52. Explain in clear language what it means for a finite difference method to be stable versus unstable.

Exercise 9.53. Show the full mathematical details for solving the 1D heat equation using the implicit and Crank-Nicolson methods.

Exercise 9.54. Show the full mathematical details for building a downwind finite difference scheme for the travelling wave equation. Discuss the primary disadvantages of the downwind scheme.

Exercise 9.55. Show the full mathematical details for building an upwind finite difference scheme for the travelling wave equation. Discuss the primary disadvantages of the upwind scheme.

Exercise 9.56. Show the full mathematical details for numerically solving the 1D Laplace and Poisson equations.

9.7 Problems

Exercise 9.57. In this problem we will solve a more realistic 1D heat equation. We will allow the diffusivity to change spatially, so $D = D(x)$ and we want to solve

$$u_t = (D(x)u_x)_x$$

on $x \in (0, 1)$ with Dirichlet boundary conditions $u(t, 0) = u(t, 1) = 0$ and initial condition $u(0, x) = \sin(2\pi x)$. This is “more realistic” since it would be rare to have a perfectly homogeneous medium, and the function D reflects any heterogeneities in the way the diffusion occurs. In this problem we will take $D(x)$ to be the parabola $D(x) = x^3(1 - x)$. We start by doing some calculus to rewrite the differential equation:

$$u_t = D(x)u_{xx}(x) + D'(x)u_x(x).$$

Your jobs are:

1. Describe what this choice of $D(x)$ might mean physically in the heat equation.
2. Write an explicit scheme to solve this problem by using centred differences for the spatial derivatives and an Euler-type discretization for the temporal derivative. Write a clear and thorough explanation for how you are doing the discretization as well as a discussion for the errors that are being made with each discretization.
3. Write a script to find an approximate solution to this problem.
4. Write a clear and thorough discussion about how you will choose Δx and Δt to give stable solutions to this equation.

5. Graphically compare your solution to this problem with a heat equation where D is taken to be the constant average diffusivity found by calculating $D_{ave} = \int_0^1 D(x)dx$. How does the changing diffusivity change the shape of the solution?
-

Exercise 9.58. In a square domain create a function $u(0, x, y)$ that looks like the university logo. The simplest way to do this might be to take a photo of the logo, crop it to a square, and use the `scipy.ndimage.imread` command to read in the image. Use this function as the initial condition for the heat equation on a square domain with homogeneous Dirichlet boundary conditions. Numerically solve the heat equation and show an animation for what happens to the logo as time evolves.

Exercise 9.59. Repeat the previous exercise but this time solve the wave equation with the logo as the initial condition.

Exercise 9.60. The explicit finite difference scheme that we built for the 1D heat equation in this chapter has error on the order of $\mathcal{O}(\Delta t) + \mathcal{O}(\Delta x^2)$. Explain clearly what this means. Then devise a numerical experiment to empirically test this fact. Clearly explain your thought process and show sufficient plots and mathematics to support your work.

Exercise 9.61. Suppose that we have a concrete slab that is 10 meters in length, with the left boundary held at a temperature of 75° and the right boundary held at a temperature of 90° . Assume that the thermal diffusivity of concrete is about $k = 10^{-5}$ m²/s. Assume that the initial temperature of the slab is given by the function $T(x) = 75 + 1.5x - 20 \sin(\pi x/10)$. In this case, the temperature can be analytically solved by the function $T(t, x) = 75 + 1.5x - 20 \sin(\pi x/10)e^{-ct}$ for some value of c .

1. Working by hand (no computers!) test the proposed analytic solution by substituting it into the 1D heat equation and verifying that it is indeed a solution. In doing so you will be able to find the correct value of c .
2. Write numerical code to solve this 1D heat equation. The output of your code should be an animation showing how the error between the numerical solution and the analytic solution evolve in time.

Exercise 9.62. (This problem is modified from (Kimberly Spayd and James Puckett 2022)). The data given below is real experimental data provided courtesy of the authors.)

Harry and Sally set up an experiment to gather data specifically for the heat diffusion through a long thin metal rod. Their experimental setup was as follows.

- The ends of the rod are submerged in water baths at different temperatures and the heat from the hot water bath (on the right hand side) travels through the metal to the cooler end (on the left hand side).
- The temperature of the rod is measured at four locations; those measurements are sent to a Raspberry Pi, which processes the raw data and sends the collated data to be displayed on the computer screen.
- They used a metal rod of length $L = 300\text{mm}$ and square cross-sectional width 3.2mm .
- The temperature sensors were placed at $x_1 = 47\text{mm}$, $x_2 = 94\text{mm}$, $x_3 = 141\text{mm}$, and $x_4 = 188\text{mm}$ as measured from the cool end (the left end).
- Foam tubing, with a thickness of 25 mm, was wrapped around the rod and sensors to provide some insulation.
- The ambient temperature in the room was 22°C and the cool water bath is a large enough reservoir that the left side of the rod is kept at 22°C .

The data table below gives temperature measurements at 60 second intervals for each of the four sensors.

Time (sec)	Sensor 188	Sensor 141	Sensor 94	Sensor 47
0	22.8	22	22	22
60	29.3	24.4	23.2	22.8
120	35.7	27.5	25.9	25.2
180	41.8	30.3	27.9	26.8
240	45.8	33.8	30.6	29.2
300	48.2	36.5	32.6	31.2
360	50.6	37.7	34.2	32
420	53.4	38.5	34.9	32.8
480	53	38.9	35.3	33.6
540	53	40.4	36.5	34.8
600	55.1	41.2	37.3	35.2
660	54.7	42	38.1	35.6
720	54.7	42.4	38.1	36
780	54.7	42.4	38.1	36.4

Time (sec)	Sensor 188	Sensor 141	Sensor 94	Sensor 47
840	54.7	42	38.5	36
900	57.5	41.2	37.7	35.6
960	56.3	40.8	37.3	35.6

- At time $t = 960$ seconds the temperatures of the rod are essentially at a steady state. Use this data to make a prediction of the temperature of the hot water bath located at $x = 300mm$.
- The thermal diffusivity, D , of the metal is unknown. Use your numerical solution in conjunction with the data to approximate the value of D . Be sure to fully defend your process.
- It is unlikely that your numerical solution to the heat equation and the data from part 2 match very well. What are some sources of error in the data or in the heat equation model?

You can load the data directly with the following code.

```
import numpy as np
import pandas as pd
URL = 'https://github.com/gustavdelius/NumericalAnalysis2025/raw/main/data/PDE/'
data = np.array(pd.read_csv(URL+'1dheatdata.csv'))
```

Exercise 9.63. You may recall from your differential equations class that population growth under limited resources is governed by the logistic equation $x' = k_1x(1-x/k_2)$ where $x = x(t)$ is the population, k_1 is the intrinsic growth rate of the population, and k_2 is the carrying capacity of the population. The carrying capacity is the maximum population that can be supported by the environment. The trouble with this model is that the species is presumed to be fixed to a spatial location. Let us make a modification to this model that allows the species to spread out over time while they reproduce. We have seen throughout this chapter that the heat equation $u_t = D(u_{xx} + u_{yy})$ models the diffusion of a substance (like heat or concentration). We therefore propose the model

$$\frac{\partial u}{\partial t} = k_1u \left(1 - \frac{u}{k_2}\right) + D \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

where $u(t, x, y)$ is the population density of the species at time t and spatial point (x, y) , (x, y) is a point in some square spatial domain, k_1 is the growth rate of the population, k_2 is the

carrying capacity of the population, and D is the rate of diffusion. Develop a finite difference scheme to solve this PDE. Experiment with this model showing the interplay between the parameters D , k_1 , and k_2 . Take an initial condition of

$$u(0, x, y) = e^{-((x-0.5)^2+(y-0.5)^2)/0.05}.$$

Exercise 9.64. In Exercise 9.47 you solved the Poisson equation, $u_{xx} + u_{yy} = f(x, y)$, on the unit square with homogeneous Dirichlet boundary conditions and a forcing function $f(x, y) = -20 \exp\left(-\frac{(x-0.5)^2+(y-0.5)^2}{0.05}\right)$. Use a 10×10 grid of points to solve the Poisson equation on the same domain with the same forcing function but with boundary conditions

$$u(0, y) = 0, \quad u(1, y) = 0, \quad u(x, 0) = -\sin(\pi x), \quad u(x, 1) = 0.$$

Show a contour plot of your solution.

9.8 Projects

In this section we propose several ideas for projects related to numerical partial differential equations. 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.

9.8.1 Hunting and Diffusion

Let u be a function modelling a mobile population in an environment where it has an intrinsic growth rate of r and a carrying capacity of K . If we were only worried about the size of the population we could solve the differential equation

$$\frac{du}{dt} = ru \left(1 - \frac{u}{K}\right),$$

but there is more to the story.

Hunters harvest the population at a per-capita rate h so we can append the differential equation with the harvesting term $-hu$ to arrive at the ordinary differential equation

$$\frac{du}{dt} = ru \left(1 - \frac{u}{K}\right) - hu.$$

Since the population is mobile let us make a few assumptions about the environment that they are in and how the individuals move.

- The growing conditions for the population are the same everywhere
- Individuals move around randomly.

Clearly these assumptions imply that our model is a simplification of real populations and real environments, but let us go with it for now. Given the nature of these assumptions we assume that a diffusion term models the spread of the individuals in the population. Hence, the PDE model is

$$\frac{\partial u}{\partial t} = ru \left(1 - \frac{u}{K} \right) - hu + D(u_{xx} + u_{yy}).$$

1. Use any of your ODE codes to solve the ordinary differential equation with harvesting. Give a complete description of the parameter space.
2. Write code to solve the spatial+temporal PDE equation on the 2D domain $(x, y) \in [0, 1] \times [0, 1]$. Choose an appropriate initial condition and choose appropriate boundary conditions.
3. The third assumption is not necessary true for rough terrain. The true form of the spatial component of the differential equation is $\nabla \cdot (D(x, y) \nabla u)$ where $D(x, y)$ is a multivariable function dictating the ease of diffusion in different spatial locations. Propose a (non-negative) function $D(x, y)$ and repeat part 2 with this new diffusion term.

9.8.2 Heating Adobe Houses

Adobe houses, typically built in desert climates, are known for their great thermal efficiency. The heat equation

$$\frac{\partial T}{\partial t} = \frac{k}{c_p \rho} (T_{xx} + T_{yy} + T_{zz}),$$

where c_p is the specific heat of the adobe, ρ is the mass density of the adobe, and k is the thermal conductivity of the adobe, can be used to model the heat transfer through the adobe from the outside of the house to the inside. Clearly, the thicker the adobe walls the better, but there is a trade off to be considered:

- it would be prohibitively expensive to build walls so think that the inside temperature was (nearly) constant, and
- if the walls are too thin then the cost is low but the temperature inside has a large amount of variability.

Your Tasks:

1. Pick a desert location in the southwestern US (New Mexico, Arizona, Nevada, or Southern California) and find some basic temperature data to model the outside temperature during typical summer and winter months.

2. Do some research on the cost of building adobe walls and find approximations for the parameters in the heat equation.
3. Use a numerical model to find the optimal thickness of an adobe wall. Be sure to fully describe your criteria for optimality, the initial and boundary conditions used, and any other simplifying assumptions needed for your model.

- 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.
- Kimberly Spayd, and James Puckett. 2022. “9-020-HeatDiffusion-ModelingScenario.” <https://doi.org/doi:10.25334/WFSN-Q683>.
- 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/>.
- Spindler, Richard. 2022. “6-023-DroneHeadingHome-ModelingScenario.” <https://doi.org/doi:10.25334/F80X-6R33>.