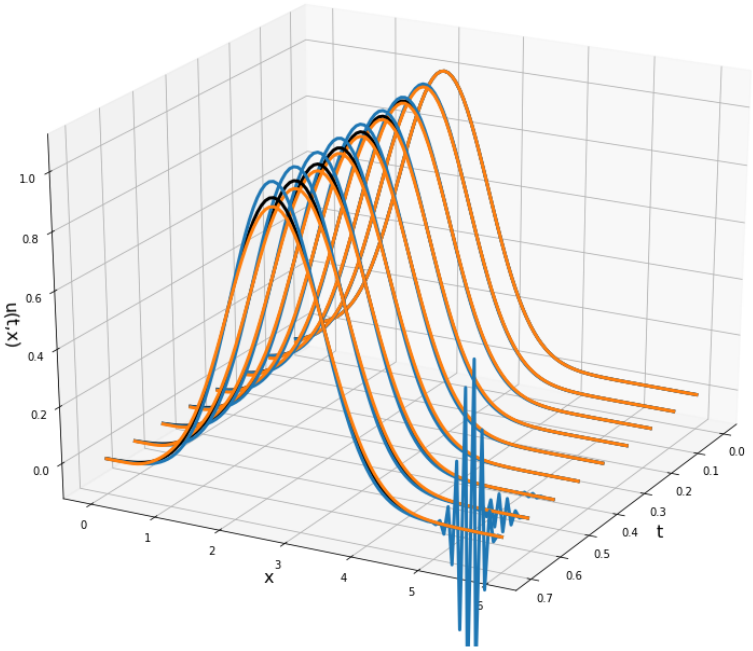
Numerical Methods

An Inquiry-Based Approach With Python



Eric Sullivan

Last Updated: 2020-09-07

Contents

Front Matter

Resources

- HTML Version of this book: https://NumericalMethodsSullivan.github.io
- PDF Version of this book: https://github.com/NumericalMethodsSullivan/NumericalMethodsSullivan.github.io/blob/master/_main.pdf
- Print On Demand Version of this book: *Coming Soon*
- Complete Instructor's Solutions: *Coming Soon*
- Google Colab: https://colab.research.google.com/
- Jupyter Notebooks: https://jupyter.org/

Preface

This book grew out of lecture notes, classroom activities, code, examples, exercises, projects, and challenge problems for my introductory course on numerical methods. The prerequisites for this material include a firm understanding of single variable calculus (though multivariable calculus doesn't hurt), a good understanding of the basics of linear algebra, a good understanding of the basics of differential equations, and some exposure to scientific computing (as seen in other math classes or perhaps from a computer science class). The primary audience is any undergraduate STEM major with an interest in using computing to solve problems.

A note on the book's title: I do not call these materials "numerical analysis" even though that is often what this course is called. In these materials I emphasize "methods" and implementation over rigorous mathematical "analysis." While this may just be semantics I feel that it is important to point out. If you are looking for a book that contains all of the derivations and rigorous proofs of the primary results in elementary numerical analysis, then this not the book for you. I have intentionally written this material with an inquiry-based emphasis which means that this is not a traditional text on numerical analysis – there are plenty of those on the market.

Creative Commons

Acknowledgements

I would first like to thank Dr. Kelly Cline for being brave enough to teach a course that he loves out of a rough draft of my notes. Your time, suggested edits, and thoughts for future directions of the book were, and are, greatly appreciated. Second, I would like to thank Johnanna for simply being awesome and giving your full support along the way. Next I would like to thank my students and colleagues, past, present, and future, for giving feedback and support for this project.

To The Student

The Inquiry-Based Approach

Let's start the book off right away with a problem designed for groups, discussion, disagreement, and deep critical thinking. This problem is inspired by Dana Ernst's first day IBL activity titled: Setting the Stage.

Exercise 0.1. * Get in groups of size 3-4. * Group members should introduce themselves. * For each of the questions that follow I will ask you to:

a. **Think** about a possible answer on your own
b. **Discuss** your answers with the rest of the group
c. **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?

This material is written with an Inquiry-Based Learning (IBL) flavor. In that sense, this document could be used as a stand-alone set of materials for the course but these notes are not a *traditional textbook* containing all of the expected theorems, proofs, code, examples, and exposition. You are encouraged to work through problems and homework, present your findings, and work together when appropriate. You will find that this document contains collections of problems with only minimal interweaving exposition. It is expected that you do every one of the problems as use the sequencing of the problems to guide your learning and understanding.

To learn more about IBL go to http://www.inquirybasedlearning.org/about/. The long and short of it is that you, the student, are the one that is doing the work; proving theorems, writing code, working problems, leading discussions, and pushing the pace. The instructor acts as a guide who only steps in to redirect conversations or to provide necessary insight.

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

1. **Fight!** You will have to fight hard to work through this material. The fight is exactly what we're after since it is ultimately what leads to innovative thinking.
2. **Screw Up!** More accurately, don't be afraid to screw up. You should write code, work problems, and prove theorems then be completely unafraid to scrap what you've done and redo it from scratch.
3. **Collaborate!** You should collaborate with your peers with the following caveats:
    1. When you are done collaborating you should go your separate ways. When you write your solution you should have no written (or digital) record of your collaboration.
    2. Use of the internet to help solve these problems robs you of the most important part of this class; the chance for original thought.
4. **Enjoy!** Part of the fun of IBL 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!

To the Instructor

If you are an instructor wishing to use these materials then I only ask that you adhere to the Creative Commons license. You are welcome to use, distribute, and remix these materials for your own purposes. Thanks for considering my materials for your course! Let me know if you have questions, edits, or suggestions: esullivan@carroll.edu.

I have authored this version of the book using R-Bookdown [1] as the primary authoring tool. This particular tool mixes the LaTeX typesetting language along with the powerful markdown language. It also allows for the Python code to be embedded directly into the book so I can run the code, build the figures, and generate output all in one place.

The Inquiry-Based Approach

I have written these materials with an inquiry-based flavor. This means that this is not a traditional textbook. I don't lecture through hardly any of the material in the book. Instead my classes are structured so that students are given problems to work before class, we build off of those problems in class, and we repeat. The exercises at the end of the chapters are assigned weekly and graded with a revision process in mind – students redo problems if the coding was incorrect, if the mathematics was incorrect, or if they somehow missed the point. The students are tasked with building most of the algorithms, code, intuition, and analysis with my intervention only if I deem it necessary.

Several of the problems throughout the book are meant to be done in groups either at the boards in the classroom or in some way where they can share their work. Much of my class time is spent with students actively building algorithms or group coding. The beauty, as I see it, of IBL is that you can run your course in any way that is comfortable for you. You can lecture through some of the material in a more traditional way, you can let the students completely discover some of the methods, or you can do a mix of both.

You will find that I do not give rigorous (in the mathematical sense) proofs or derivations of many of the algorithms in this book. I tend to lean on numerical experiments to allow students to discover algorithms, error estimates, and other results without the rigor. The makeup of my classes tends to be math majors along with engineering, computer science, physics, and data science students.

The Projects

I have taught this class with anywhere from two to four projects during the semester. Each of the projects is designed to give the students an open-ended task where they can show off their coding skills and, more importantly, build

their mathematical communication skills. Projects can be done in groups or individually depending on the background and group dynamics of your class. Appendix B contains several tips for how to tackle the writing in the projects.

Coding

I expect that my students come with some coding experience from other mathematics or computer science classes. With that, I leave the coding help as an appendix (see Appendix A) and only point the students there for refreshers. If your students need a more thorough ramp up to the coding then you might want to start the course with Appendix A to get the students up to speed. I expect the students to do most of the coding the in the class, but occasionally we will code algorithms together (especially earlier in the semester when the students are still getting their feet underneath them).

I encourage students to learn Python. It is a general purpose language that does extremely well with numerical computing when paired with `numpy` and `matplotlib`. Appendix A has several helpful sections for getting students up to speed with Python.

I encourage you to consider having your students code in Jupyter Notebooks or Google CoLab. The advantage is that students can mix their writing and their code in a seamless way. This allows for an iterative approach to coding and writing and gives the students the tools to explain what they're doing as they code.

Pacing

The following is a typical 15-week semester with these materials.

- Chapter 1 - 1.5 weeks
- Chapter 2 - 1.5 weeks
- Chapter 3 - 2 weeks
- Chapter 4 - 3 weeks
- Chapter 5 - 3 weeks
- Chapter 6 - 3 weeks

If you are starting with Appendix A then you will likely lose time out of the later chapters. Typically I trim Chapters 4 and 6 a bit short – perhaps not covering the power method, traveling wave equations, and the Laplace equation. This buys a bit more time to teach programming at the beginning of the course.

Other Considerations:

Projects: I typically assign a project after Chapter 2 or 3, a second project after Chapter 4, and a third project after Chapter 5. The fourth project, if time allows, typically comes from Chapter 6. I typically dedicate two class days to the first project and then one class day to each subsequent project. For the final project I typically have students present their work so this takes a day or two out of our class time.

Exercises: I typically assign one collection of exercises per week. Students are to work on these outside of class, but in some cases it is worth taking class time to let students work in teams. Of particular note are the coding exercises in Chapter 1. If your students need practice with coding then it might be worthwhile to mix these exercises in through several assignments and perhaps during a few class periods. I have also taken extra class time with the exercises in Chapter 5 to let the students work in pairs on the modeling aspects of some of the problems.

Exams: This is a non-traditional book and as such you might want to consider some non-traditional exam settings. Some ideas that my colleagues and I have used are:

- Use code and functions that you've written to solve several new problems during a class period.
- Give the mathematical details and the derivations of key algorithms.
- Take several problems home (under strict rules about collaboration) and return with working code and a formal write up.
- No exams, but put heavier weight on the projects.

Chapter 1

Preliminary Topics

1.1 What Is Numerical Analysis?

> *"In the 1950s and 1960s, the founding fathers of Numerical Analysis discovered that inexact arithmetic can be a source of danger, causing errors in results that 'ought' to be right. The source of such problems is numerical instability: that is, the amplification of rounding errors from microscopic to macroscopic scale by certain modes of computation."*
> –Oxford Professor Lloyd (Nick) Trefethen (2006)

The field of Numerical Analysis is really the study of how to take mathematical problems and perform them efficiently and accurately on a computer. While the field of numerical analysis is quite powerful and wide-reaching, there are some mathematical problems where numerical analysis doesn't make much sense (e.g. finding an algebraic derivative of a function, proving a theorem, uncovering a pattern in a sequence). However, for many problems a numerical method that gives an approximate answer is both more efficient and more versatile than any analytic technique. Let's look at several examples.

Example from Algebra: Solve the equation $\ln(x) = \sin(x)$ for x in the interval $x \in (0, \pi)$. Stop and think about all of the algebra that you ever learned. You'll 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.

Example from Calculus: What if we want to evaluate the following integral?

$$\int_0^{\pi} \sin(x^2)dx$$

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

Example from Linear Algebra: You have probably never row reduced a matrix larger than 3×3 or perhaps 4×4 by hand. Instead, you often turn to technology to do the row reduction for you. You would be surprised to find that the standard row reduction algorithm (RREF) that you do by hand is not what a computer uses. Instead, there are efficient algorithms to do the basic operations of linear algebra (e.g. Gaussian elimination, matrix factorization, or eigenvalue decomposition)

In this chapter we will discuss some of the basic underlying ideas in Numerical Analysis, and the essence of the above quote from Nick Trefethen will be part of the focus of this chapter. Particularly, we need to know how a computer stores numbers and when that storage can get us into trouble. On a more mathematical side, we offer a brief review of the Taylor Series from Calculus at the end of this chapter. The Taylor Series underpins many of our approximation methods in this class. Finally, at the end of this chapter we provide several coding exercises that will help you to develop your programming skills. It is expected that you know some of the basics of programming before beginning this class. If you need to review the basics then see Appendix A

You'll have more than just the basics by the end.

Let's begin.

1.2 Arithmetic in Base 2

Exercise 1.1. By hand (no computers!) compute the first 50 terms of this sequence with the initial condition $x_0 = 1/10$.

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

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

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

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

Exercise 1.4. 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 1.5. Now what happens with this problem when you start with $x_0 = 1/8$? Why does this new initial condition work better?

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 worth while 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

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

and we can read this number as 7 hundreds, 3 tens, and 5 ones. As you can see, in a "positional number system" such as our base-10 system, the position of the number indicates the power of the base, and the value of the digit itself tells you the multiplier of that power. This is contrary to number systems like Roman Numerals where the symbols themselves give us the number, and meaning of the position is somewhat flexible. The number "48,329" can therefore be interpreted as

$$48,329 = 40,000+8,000+300+20+9 = 4\times10^4+8\times10^3+3\times10^2+2\times10^1+9\times10^0,$$

Four ten thousands, eight thousands, three hundreds, two tens, and nine ones.

Now let's 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

$$101,101_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

(where the subscript "2" indicates the base to the reader). 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 reader should take note that 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're reading.

Exercise 1.6. Express the following binary numbers in base-10.

a. 111_2
b. $10,101_2$
c. $1,111,111,111_2$

Exercise 1.7. Explain the joke: *There are 10 types of people. Those who understand binary and those who don't.*

Exercise 1.8. 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 1.9. Convert to 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 1.10. Now that you have converted several base-10 numbers to base-2, summarize an efficient technique to do the conversion.

Example 1.1. Convert the number 137 from base 10 to base 2.
Solution: One way to do the conversion is to first look for the largest power of 2 less than or equal to your number. In this case, $128 = 2^7$ is the largest power of 2 that is less than 137. Then looking at the remainder, 9, look for the largest

power of 2 that is less than this remainder. Repeat until you have the number.

$$\begin{aligned} 137_{10} &= 128 + 8 + 1 \\ &= 2^7 + 2^3 + 2^0 \\ &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 10001001_2 \end{aligned}$$

Next we'll work with fractions and decimals. For example, let's take the base 10 number 5.341_{10} and expand it out to get

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

The position to the right of the decimal point is the negative power of 10 for the given position. We can do a similar thing with binary decimals.

Exercise 1.11. 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.

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

Exercise 1.12. Repeating digits in binary numbers are rather intriguing. The number $0.\overline{0111} = 0.01110111011101110111\ldots$ surely also has a decimal representation. I'll get you started:

$$\begin{aligned} 0.0_2 &= 0 \times 2^0 + 0 \times 2^{-1} = 0.0_{10} \\ 0.01_2 &= 0.0_{10} + 1 \times 2^{-2} = 0.25_{10} \\ 0.011_2 &= 0.25_{10} + 1 \times 2^{-3} = 0.25_{10} + 0.125_{10} = 0.375_{10} \\ 0.0111_2 &= 0.375_{10} + 1 \times 2^{-4} = 0.4375_{10} \\ 0.01110_2 &= 0.4375_{10} + 0 \times 2^{-5} = 0.4375_{10} \\ 0.011101_2 &= 0.4375_{10} + 1 \times 2^{-6} = 0.453125_{10} \\ \vdots \qquad & \qquad \vdots \qquad \qquad \vdots \end{aligned}$$

We want to know what this series converges to in base 10. Work with your partners to approximate the base-10 number.

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

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

Exercise 1.13. 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 1.14. Convert the base 10 decimal 0.635 to binary using the following steps.

a. Multiply 0.635 by 2. The whole number part of the result is the first binary digit to the right of the decimal point.
b. 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.
c. 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 1.15. Based on your previous problem write an algorithm that will convert base-10 decimals (less than 1) to base decimal expansions.

Exercise 1.16. Convert the base 10 fraction 1/10 into binary. Use your solution to fully describe what went wrong in the Exercise 1.1.

1.3 Floating Point Arithmetic

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. Moreover, since there is finite space in a computer we can only ever store rational numbers (stop and think: why is this). Therefore we need to know what gaps in our number system to expect when using a computer to store and do computations on numbers.

Exercise 1.17. Let's start the discussion with a very concrete example. Consider the number $x = -123.15625$ (in base 10). As we've 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

$$x = -1.\_\_\_\_\_ \times 2\text{——}$$

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 $x = 10001001.1100110011_2$?

For any base-2 number x we can write

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

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

For a number $x = (-1)^s \times (1 + m) \times 2^E$ stored in a computer, the number m is called the **mantissa** or the **significand**, s is known as the sign bit, and E is known as the exponent.

Example 1.3. What are the mantissa, sign bit, and 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 $\frac{1}{10} = 0.000110011001100\cdots = 1.100110011 \times 2^{-4}$ 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 can't happen since we are dealing with computers with finite memory. 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 three standard precisions for storing numbers on a computer: single, double, and extended precision. The double precision standard is what most of our modern computers use.

Definition 1.1. There are three standard precisions for storing numbers in a computer.

- A **single-precision** number consists of 32 bits, with 1 bit for the sign, 8 for the exponent, and 23 for the significand.
- A **double-precision** number consists of 64 bits with 1 bit for the sign, 11 for the exponent, and 52 for the significand.
- An **extended-precision** number consists of 80 bits, with 1 bit for the sign, 15 for the exponent, and 64 for the significand.

Definition 1.2 (Machine Precision)**.** 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 difference is smaller than machine precision. This is of the utmost important when you want to check that something is "zero" since a computer just cannot know the difference between 0 and ϵ.

Exercise 1.18. To make all of these ideas concrete let's 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'll 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

$$(-1)^s \times (1 + 0.b_1b_2b_3) \times 2^E$$

- 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 1.19. What are the largest and smallest numbers that can be stored in single and double precicision?

Exercise 1.20. What is machine epsilon for single and and double precision?

Exercise 1.21. Explain the behavior of the sequence from the first problem in these notes using what you know about how computers store numbers in double precision.

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

In particular, now that you know about how numbers are stored in a computer, how long do you expect it to take until the truncation error creeps into the computation?

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 class and this book to go into all of those details here. Instead, 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.
- Computers cannot tell the difference between two numbers if the difference is less than machine precision.

1.4 Approximating Functions

Numerical analysis is all about doing mathematics on a computer in accurate and predictable ways. Since a computer can only ever store finite bits of information for any number, everything that we do in a computer is naturally an approximation of the real mathematics. In this section we will look at a very powerful way to approximate mathematical functions.

How does a computer *understand* a function like $f(x) = e^x$ or $f(x) = \sin(x)$ or $f(x) = \ln(x)$? 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 darn good at arithmetic, but working with transcendental functions like these, or really any other sufficiently complicated functions for that matter, causes all sorts of problems in a computer. Approximation of the function is something that is always happening under the hood.

Exercise 1.22. In this problem we're going to make a bit of a wish list for all of the things that a computer will do when approximating a function. We're going to complete the following sentence:
If we are going to approximate $f(x)$ near the point $x = x_0$ with a simpler function $g(x)$ then ...

(I'll 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 to the right of $x = x_0$ then $g(x)$ ...
- if $f(x)$ is increasing / decreasing to the left of $x = x_0$ then $g(x)$ ...
- if $f(x)$ is concave up / down to the right of $x = x_0$ then $g(x)$...
- if $f(x)$ is concave up / down to the left of $x = x_0$ then $g(x)$ ...
- if we zoom into plots of the functions $f(x)$ and $g(x)$ near $x = x_0$ then ...
- ... is there anything else that you would add?

Exercise 1.23. Discuss: Could a polynomial function with a high enough degree satify everything in the wish list from the previous problem? Explain your reasoning.

Exercise 1.24. Let's 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

a. What is the value of A such that $f(x_0) = g(x_0)$? *(Hint: substitute $x = x_0$ into the $g(x)$ function)*
b. What is the value of B such that $f'(x_0) = g'(x_0)$? *(Hint: Start by taking the derivative of $g(x)$)*
c. What is the value of C such that $f''(x_0) = g''(x_0)$?
d. What is the value of D such that $f'''(x_0) = g'''(x_0)$?

Exercise 1.25. Let $f(x) = e^x$. Put the answers to the previous question into action and build a cubic polynomial that approximates $f(x) = e^x$ near $x_0 = 0$.

In the previous 4 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've 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'll unpack the approximations of $f(x) = e^x$ a bit more carefully and we'll wrap the whole discussion with a little bit of formal mathematical language. Then we'll examine other functions like sine, cosine, logarithms, etc. One of the points of this whole discussion is to give you a little gimps 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 1.26. 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's work with the function $f(x) = e^x$.

a. 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 $x = 0$.

b. Let's 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 $x = 0$.

   i. What is the first derivative of $f(x)$?
   ii. What is $f'(0)$?
   iii. 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.

c. Write Python code to build a plot like Figure 1.1. This plot shows $f(x) = e^x$, our first approximation $g(x) = 1$ and our second approximation $g(x) = 1 + x$.

Exercise 1.27. Let's extend the idea from the previous problem to much better approximations of the function $f(x) = e^x$.

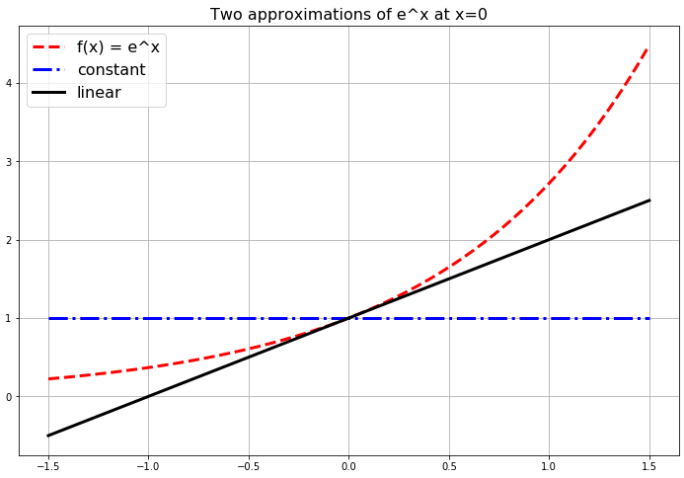


Figure 1.1: The first few Taylor approximations of the exponential function.

a. Let's 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'll 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

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

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

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

i. Find the quadratic approximation for $f(x) = e^x$.
ii. How do you know that this function matches $f(x)$ is all of the ways described above at $x = 0$?
iii. Add your new function to the plot you created in the previous problem.

b. Let's keep going!! Next we'll do a cubic approximation. A cubic approximation takes the form

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

i. Find the cubic approximation for $f(x) = e^x$.
ii. How do we know that this function matches the first, second, and third derivatives of $f(x)$ at $x = 0$?
iii. Add your function function to the plot.
iv. Pause and think: What's the deal with the 3! on the cubic term?

c. Your turn: Build the next several approximations of $f(x) = e^x$ at $x = 0$. Add these plots to the plot that we've been building all along.

Exercise 1.28. We can get a decimal expansion of e pretty easily:

$$e \approx 2.718281828459045$$

In Python just type `np.exp(1)` which will just evaluate $f(x) = e^x$ at $x = 1$ (and hence giving you a value for $e^1 = e$). We built our approximations in the previous problems centered at $x = 0$, and $x = 1$ is *not too terribly* far from $x = 0$ so perhaps we can get a good approximation with the functions that we've already built. Complete the following table to see how we did with our approximations.

Approximation Function	Value at $x = 1$	Absolute Error
Constant	1	$\|2.71828... - 1\| \approx 1.71828...$
Linear	1+1=2	$\|2.71828... - 2\| \approx 0.71828...$
Quadratic		
Cubic		
Quartic		
Quintic		

Exercise 1.29. Use the functions that you've built to approximate $\sqrt{e} = e^{0.5}$. Check the accuracy of your answer using `np.exp(0.5)` in Python.

Exercise 1.30. Use the functions that you've built to approximate $\frac{1}{e} = e^{-1}$. Check the accuracy of your answer using `np.exp(-1)` in Python.

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

Definition 1.3 (Taylor Series)**.** If $f(x)$ is an infinitely differentiable function at the point x_0 then

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

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

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

Don't 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 don't stray too far from the anchor x_0.

Definition 1.4 (Maclaurin Series)**.** A Taylor Series that is centered at $x_0 = 0$ is called a Maclaurin Series after the mathematician Colin Maclaurin. This is just a special case of a Taylor Series, so throughout this book we will refer to both Taylor Series and Maclaurin Series simply as Taylor Series.

Exercise 1.31. Verify from your previous work that the Taylor Series centered at $x_0 = 0$ (i.e. the Maclaurin Series) for $f(x) = e^x$ is indeed

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots.$$

Exercise 1.32. Do all of the calculations to show that the Taylor Series centered at $x_0 = 0$ for the function $f(x) = \sin(x)$ is indeed

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

Exercise 1.33. Do all of the calculations to show that the Taylor Series centered at $x_0 = 0$ for the function $f(x) = \cos(x)$ is indeed

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Exercise 1.34. Let's compute a few Taylor Series that are not centered at $x_0 = 0$ (that is, Taylor Series that are not Maclaurin Series). For example, let's 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

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

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

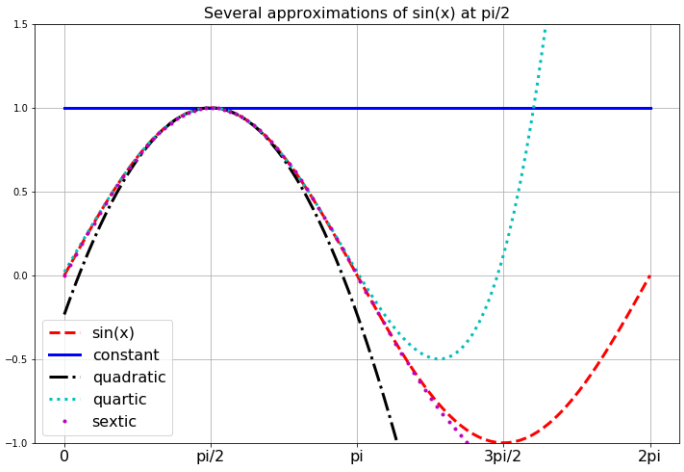


Figure 1.2: Taylor Series approximations of the sine function.

Exercise 1.35. Repeat the previous exercise for the functions

$$f(x) = \cos(x) \text{ centered at } x_0 = \pi \tag{1.1}$$
$$f(x) = \ln(x) \text{ centered at } x_0 = 1 \tag{1.2}$$

Exercise 1.36. Approximate $\cos(3)$ using a Taylor Series.

Exercise 1.37. Approximate $\ln(1.1)$ using a Taylor Series.

Example 1.4. Let's conclude this brief section by examining an interesting example. Consider the function

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

If we build a Taylor Series centered at $x_0 = 0$ (i.e. the Maclaurin Series) it isn't too hard to show that we get

$$f(x) = 1 + x + x^2 + x^3 + x^4 + x^5 + \cdots$$

(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 don't get the correct behavior of the function (see Figure 1.3). More specifically, we cannot get the Taylor Series to change behavior 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 center the expansion. For the function $f(x) = \frac{1}{1-x}$ centered at $x_0 = 0$ we can only get good approximations within the interval $x \in (-1, 1)$ and no further.

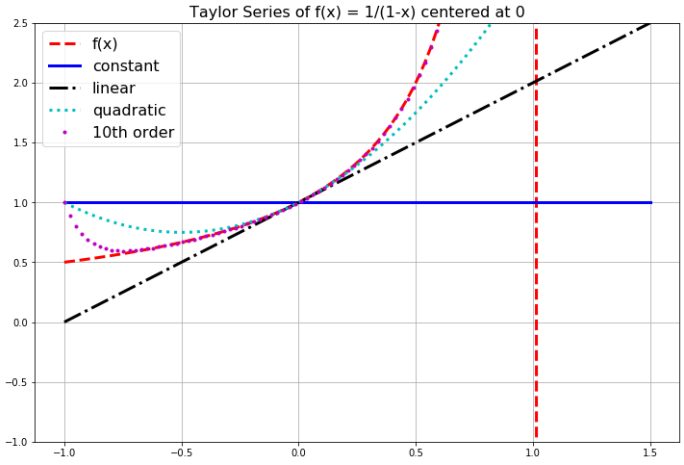


Figure 1.3: Several Taylor Series approximations of the function $f(x) = 1/(1-x)$.

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 center of the series to the nearest asymptote. The domain of convergence is typically symmetric about the center as well. For example:

- If we were to build a Taylor Series approximation for the function $f(x) = \ln(x)$ centered 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 $f(x) = \frac{5}{2x-3}$ centered 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 $f(x) = \frac{1}{1+x^2}$ centered 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's 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" isn't 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 should pause now and build some plots to show yourself that this indeed appears to be true.

A Taylor Series will give good approximations to the function within the domain of convergence, but will give garbage outside of it. For more details about the domain of convergence of a Taylor Series you can refer to the Taylor Series section of the online Active Calculus Textbook [2].

1.5 Approximation Error with Taylor Series

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 we are actually going to be using 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 can't 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's be a bit more concrete about it. The Taylor Series for $f(x) = e^x$ centered at $x_0 = 0$ is

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

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

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

Therefore we would approximate e^x as $e^x \approx 1$ 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

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

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 of $f(x) = e^x$ then we gather the 0^{th} order and 1^{st} order terms together as our approximation and the rest is the remainder

$$e^x = \underbrace{1 + x}_{1^{st} \text{ order approximation}} + \underbrace{\frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots}_{\text{remainder}}.$$

Therefore we would approximate e^x as $e^x \approx 1 + x$ 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

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

Notice that we don't 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 proportional to x^2.*

2^{nd} Order Approximation of $f(x) = e^x$: If we want to use a second-order (quadratic) approximation of $f(x) = e^x$ 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

$$e^x = \underbrace{1 + x + \frac{x^2}{2!}}_{2^{nd} \text{ order approximation}} + \underbrace{\frac{x^3}{3!} + \frac{x^4}{4!} + \cdots}_{\text{remainder}}.$$

Therefore we would approximate e^x as $e^x \approx 1 + x + \frac{x^2}{2}$ 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

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

Again notice that we don't 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 proportional to* x^3.

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 of e^x near $x_0 = 0$ then

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

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

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

Exercise 1.38. Let's make the previous discussion a bit more concrete. We know the Taylor Series for $f(x) = e^x$ quite well at this point so let's use it to approximate the value of $f(0.1) = e^{0.1}$ with different order polynomials. Notice that $x = 0.1$ is pretty close to the center of the Taylor Series $x_0 = 0$ so this sort of approximation is reasonable.

Using `np.exp(0.1)` we have Python's approximation $e^{0.1} \approx$ `np.exp(0.1)` $=$ 1.1051709181.

Fill in the blanks in the table.

Taylor Series	Approximation	Absolute Error	Expected Error
0^{th} Order	1	$\lvert e^{0.1} - 1\rvert =$ 0.1051709181	$\mathcal{O}(x) = 0.1$
1^{st} Order	1.1	$\lvert e^{0.1} - 1.1\rvert =$ 0.0051709181	$\mathcal{O}(x^2) =$ $0.1^2 = 0.01$
2^{nd} Order	1.105		
3^{rd} Order			
4^{th} Order			
5^{th} Order			

Observe in the previous exercise that the actual absolute error is always less than the expected error. Using the first term in the remainder to estimate the approximation error of truncating a Taylor Series is crude but very easy to implement.

Theorem 1.1. *The approximation error when using a truncated Taylor Series is roughly proportional to the size of the next term in the Taylor Series.*

Exercise 1.39. Next we will examine the approximation error for the sine function near $x_0 = 0$. We know that the sine function has the Taylor Series centered at $x_0 = 0$ as

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

a. A linear approximation of $\sin(x)$ near $x_0 = 0$ is $\sin(x) = x + \mathcal{O}(x^3)$.
   i. Use the linear approximation formula to approximate $\sin(0.2)$.
   ii. What does "$\mathcal{O}(x^3)$" mean about the approximation of $\sin(0.2)$? (Take note that we ignore the minus sign on the approximation error since we are really only interested in absolute error (i.e. we don't care if we overshoot or undershoot).)

b. Notice that there are no quadratic terms in the Taylor Series so there is no quadratic approximation for $\sin(x)$ near $x_0 = 0$.

c. A cubic approximation of $\sin(x)$ near $x_0 = 0$ is $\sin(x) = ?? - ?? + \mathcal{O}(??)$.
   i. Fill in the question marks in the cubic approximation formula.
   ii. Use the cubic approximation formula to approximate $\sin(0.2)$.
   iii. What is the approximation error for your approximation?

d. What is the next approximation formula for $\sin(x)$ near $x_0 = 0$? Use it to approximate $\sin(0.2)$, and give the expected approximation error.

e. Now let's check all of our answers against what Python says we should get for $\sin(0.2)$. If you use `np.sin(0.2)` you should get $\sin(0.2) \approx$ `np.sin(0.2)` $= 0.1986693308$. Fill in the blanks in the table below and then discuss the quality of our error approximations.

Taylor Series	Estimated Error	Actual Absolute Error
1^{st} Order	$\mathcal{O}(x^3) = 0.2^3 = 0.008$	0.001330669205
3^{rd} Order		
5^{th} Order		
7^{th} Order		
9^{th} Order		

f. What observations do you make about the estimate of the approximation error and the actual approximation error?

Exercise 1.40. What if we want an approximation of $\ln(1.1)$ and we want that approximation to be within 5 decimal places. The number 1.1 is very close to 1 and we know that $\ln(1) = 0$. Hence, it seems like a good idea to build a Taylor Series approximation for $\ln(x)$ centered at $x_0 = 1$ to solve this problem.

a. Complete the table of derivatives below to get the Taylor coefficients for the Taylor Series of $\ln(x)$ centered at $x_0 = 1$.

Order of Derivative	Derivative	Value at $x_0 = 1$	Taylor Coefficient
0	$\ln(x)$	0	0
1	$\frac{1}{x} = x^{-1}$	1	1
2	$-x^{-2}$	-1	$\frac{-1}{2!} = -\frac{1}{2}$
3	$2x^{-3}$	2	$\frac{2}{3!} = \frac{1}{3}$
4	$-6x^{-4}$	-6	
5			
6			
$\vdots$	$\vdots$	$\vdots$	$\vdots$

b. Based on what you did in part (a), complete the Taylor Series for $\ln(x)$ centered at $x_0 = 1$.

$$\ln(x) = 0 + 1(x-1) - \frac{1}{2}(x-1)^2 + \frac{1}{3}(x-1)^3 - \frac{??}{??}(x-1)^4 + \frac{??}{??}(x-1)^5 + \cdots.$$

c. The n^{th} order Taylor approximation of $\ln(x)$ near $x_0 = 1$ is given below. What is the order of the estimated approximation error?

$$\ln(x) = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \cdots + \frac{(-1)^{n-1}(x-1)^n}{n} + \mathcal{O}(???)$$

d. Finally we want to get an approximation for $\ln(1.1)$ accurate to 5 decimal places (or better). What is the minimum number of terms we expect to need from the Taylor Series? Support your answer mathematically.

e. Use Python's `np.log(1.1)` to get an approximation for $\ln(1.1)$ and then numerically verify your answer to part (d).

Exercise 1.41. In the previous problem you found an approximation for $\ln(1.1)$ to 5 decimal places. In doing so you had to build a Taylor Series at a well-known point nearby 1.1 and then use our approximation of the error to determine the number of terms to keep in the approximation. In this exercise we want an approximation of $\cos\left(\frac{\pi}{2} + 0.05\right)$. To do so you should build a Taylor Series for the cosine function centered at an appropriate point, determine an estimate for the approximation error, and then use that estimate to determine the number of terms to keep in the approximation.

1.6 Exercises

1.6.1 Coding Exercises

The first several exercises here are meant for you to practice and improve your coding skills. If you are stuck on any of the coding then I recommend that you have a look at Appendix A. Please refrain from Googling anything on these problems. The point is to struggle through the code, get it wrong many times, debug, and then to eventually have working code.

Exercise 1.42. (This problem is modified from [3])
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.

Exercise 1.43. (This problem is modified from [3])
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:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

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 1.44. 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 1.45. My favorite prime number is 8675309. Yep. Jenny's phone number is prime! Write a script that verifies this fact.

Hint: You only need to check divisors as large as the square root of 8675309 (why).

Exercise 1.46. (This problem is modified from [3])
Write a function called that accepts an integer and returns a binary variable: * 0 = not prime, * 1 = prime.

Next write a script to find the sum of all of the prime numbers less than 1000.

Hint: Remember that a prime number has exactly two divisors: 1 and itself. You only need to check divisors as large as the square root of n. Your script should probably be smart enough to avoid all of the non-prime even numbers.

Exercise 1.47. (This problem is modified from [3])
The sum of the squares of the first ten natural numbers is,

$$1^2 + 2^2 + \cdots + 10^2 = 385$$

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

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

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 1.48. (This problem is modified from [3])
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 1.49. (This problem is modified from [3])
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 1.50. The following iterative sequence is defined for the set of positive integers:

$$n \to \frac{n}{2} \quad \text{(n is even)}$$
$$n \to 3n + 1 \quad \text{(n is odd)}$$

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

$$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

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.

1.6.2 Applying What You've Learned

Exercise 1.51. (This problem is modified from [4])
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.

a. 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?
b. Mathematically we know that $\left(\frac{1}{49}\right) \cdot 49$ should just be 1. In Python type `(1/49)*49 == 1`. What do you get and why do you get it?
c. Mathematically we know that $e^{\ln(3)}$ 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?
d. Create your own example of where Python gets something incorrect because of floating point arithmetic.

Exercise 1.52. (This problem is modified from [4])
In the 1999 movie *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 is 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:

a. 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;
```

b. 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".
c. Write code to complete your plan from part (b).
d. Using a `while` loop, iterate over your code until the illegal account has accumulated \$1,000,000. How long does it take?

Exercise 1.53. (This problem is modified from [4])
In the 1991 Gulf War, the Patriot missle defense system failed due to roundoff 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 $\frac{1}{10}$:

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

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

Exercise 1.54. Find the Taylor Series for $f(x) = \frac{1}{\ln(x)}$ centered at the point $x_0 = e$. Then use the Taylor Series to approximate the number $\frac{1}{\ln(3)}$ to 4 decimal places.

Exercise 1.55. In this problem we will use Taylor Series to build approximations for the irrational number π.

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

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

b. Now we want to get the Taylor Series for the function $g(x) = \frac{1}{1+x^2}$. It would be quite time consuming to take all of the necessary derivatives to

get this Taylor Series. Instead we will use our answer from part (a) of this problem to shortcut the whole process.

i. Substitute x^2 for every x in the Taylor Series for $f(x) = \frac{1}{1+x}$.

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

c. Recall from Calculus that

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

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]

d. Now recall the following from Calculus:

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

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

Exercise 1.56. In this problem we will prove the famous (and the author's favorite) formula

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

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

a. Write the Taylor series for the functions e^x, $\sin(x)$, and $\cos(x)$.

b. 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

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

the Taylor expansion. I'll get you started:

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

c. 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?

d. Use your result from part (c) to prove that $e^{i\pi} + 1 = 0$.

Exercise 1.57. In physics, the *relativistic energy* of an object is defined as

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

where

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

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

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

a. What do we recover if we consider an object with zero velocity?

b. Why might it be completely reasonable to only use the quadratic approximation

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

for the relativistic energy equation?[2]

c. (some physics knowledge required) What do you notice about the second term in the Taylor Series approximation of the relativistic energy function?

d. Show all of the work to derive the Taylor Series centered at $v = 0$ given above.

---

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

Exercise 1.58 (The Python Caret Operator)**.** Now that you're used to using Python to do some basic computations you are probably comfortable with the fact that the caret, `^`, does NOT do exponentiation like it does in many other programming languages. But what does the caret operator do? That's what we explore here.

a. 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).

$$9 = \_\_\_\_\_$$
$$5 = \_\_\_\_\_$$

b. Now go to Python and evaluate the expression `9^5`. Convert Python's answer to a binary representation (again using four bits).

c. 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.

Chapter 2

Algebra

2.1 Intro to Numerical Root Finding

> The golden rule of numerical analysis: *We compute only when everything else fails.*

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 in high school algebra and Pre-Calculus. 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. As another example, 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're 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:

- 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.

- Most equations do not lend themselves to by-hand solutions. The techniques that we know from high school algebra solve common, and often

quite simplified, problems but when equations arise naturally they are often not *nice*.

Let's 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

$$\ell(x) - r(x) = 0.$$

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

$$\text{If } f(x) = 0, \text{ find } x.$$

This gives us a common language for which to frame all of our numerical algorithms.

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 2.1. You should pause and notice that there is no way that you are going to apply by-hand techniques from algebra to solve this equation ... an approximate answer is pretty much our only hope.

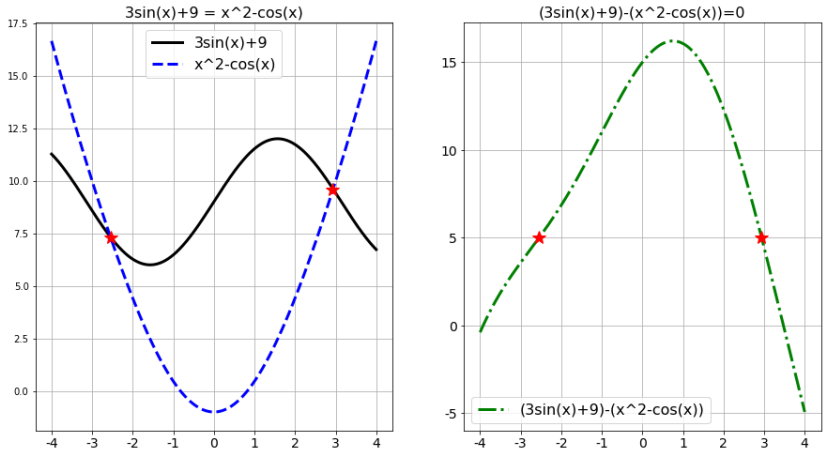


Figure 2.1: A Typical Root Finding Problem

On the left-hand side of Figure 2.1 we see the solutions to the equation $3\sin(x) + 9 = x^2 - \cos(x)$, and on the right-hand side we see the solutions to the equation

$$(3\sin(x) + 9) - \big(x^2 - \cos(x)\big) = 0.$$

From the plots it is apparent that the two equations have the same solutions: $x_1 \approx -2.55$ and $x_2 \approx 2.88$. Figure 2.1 should demonstrate what we mean when

we say that solving equations of the form $\ell(x) = r(x)$ will give the same answer as solving $f(x) = 0$. Pause for a moment and closely examine the plots to verify this for yourself.

We now have one way to view every equation-solving problem. As we'll 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. The following sections give several different techniques for solving equations of the form $f(x) = 0$. We will start with the simplest techniques to implement and then move to the more powerful techniques that require some ideas from Calculus to understand and analyze. Throughout this chapter we will also work to quantify the amount of error that we make while using these techniques.

2.2 The Bisection Method

2.2.1 Intuition and Implementation

Exercise 2.1. 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'm thinking of a number between 1 and 100
- Joe: Is it 35?
- Sally: No, but the number is between 35 and 100
- Joe: Is it 99?
- Sally: No, but the number is between 35 and 99
- ...

Exercise 2.2. Now let's say that Sally has a continuous function that has a root somewhere between $x = 2$ and $x = 10$. Modify your strategy from the number guessing game in the previous problem to narrow down where the root is.

Exercise 2.3. Was it necessary to say that Sally's function was continuous? Could your technique work if the function were not continuous.

Now let's get to the math. We'll start the mathematical discussion with a theorem from Calculus.

Theorem 2.1 (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 2.4. Draw a picture of what the intermediate value theorem says graphically.

Exercise 2.5. If $y_* = 0$ the Intermediate Value Theorem gives us important information about solving equations. What does it tell us?

Corollary 2.1. *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 2.6. 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 don't tell us *how* to find the point that is guaranteed to exist – annoying. The method that you developed in Exercises 2.1 and 2.2 give one possible way to find the root.

In Exercises 2.1 and 2.2 you likely came up with an algorithm such as this:

- Say we know that the root of a continuous function lies between $x = a$ 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 which contains the root to either $[a, m]$ or $[m, b]$.
- Repeat

Now we will turn this optimal 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 2.7. Where is the Intermediate Value Theorem used in the root-guessing strategy?

Exercise 2.8. Why was it important that the function $f(x)$ is continuous when playing this root-guessing game? Provide a few sketches to demonstrate your answer.

Exercise 2.9 (The Bisection Method)**. Goal:** We want to solve the equation $f(x) = 0$ for x assuming that the solution x^* is in the interval $[a, b]$.

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 $f(a)$ vs $f(m)$ and $f(b)$ vs $f(m)$.
   - 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 $f(m)$ is *close enough* to zero.

Exercise 2.10. Draw a picture illustrating what the Bisection Method does to approximate solutions to the equation $f(x) = 0$.

Exercise 2.11. We want to write a Python function for the Bisection Method. Instead of jumping straight into the code we should ALWAYS write pseudo-code first. It is often helpful to write pseudo-code as comments in your file. Use the template below to complete your pseudo-code.

```
def Bisection(f , a , b , tol):
# The input parameters are
# f is a Python function or a lambda function
# a is the lower guess
# b is the upper guess
# tol is an optional tolerance for the accuracy of the root

# if the user doesn't define a tolerance we need code to create a default

# check that there is a root between a and b
# if not we should return an error and break the code
```

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

# start a while loop
#   # in the while loop we need an if statement
#   # if ...
#   # elif ...
#   # elif ...

#   # we should check that the while loop isn't running away

# end the while loop
# define and return the root
```

Exercise 2.12. Now use the pseudo-code as structure to complete a function for the Bisection Method. Also write test code that verifies that your function works properly. Be sure that it can take a Lambda Function as an input along with an initial lower bound, an initial upper bound, and an optional error tolerance. The output should be only 1 single number: the root.

Exercise 2.13. Test your Bisection Method code on the following equations.

a. $x^2 - 2 = 0$ on $x \in [0, 2]$
b. $\sin(x) + x^2 = 2\ln(x) + 5$ on $x \in [0, 5]$ (be careful! make a plot first)
c. $(5 - x)e^x = 5$ on $x \in [0, 5]$

2.2.2 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 2.14. Discussion: What must be true in order to use the bisection method?

Exercise 2.15. 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 2.16. 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'll 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 in order to stop.

Exercise 2.17. Let $f(x)$ be a continuous function on the interval $[a, b]$ and assume that $f(a) \cdot f(b) < 0$. A reoccurring 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.

a. Let's 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?

b. 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	Approximation Error
0	$\lvert b-a\rvert$	$\frac{\lvert b-a\rvert}{2}$
1	$\frac{\lvert b-a\rvert}{2}$	
2	$\frac{\lvert b-a\rvert}{2^2}$	
$\vdots$	$\vdots$	$\vdots$
n	$\frac{\lvert b-a\rvert}{2^n}$	

c. 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 2.17 you actually proved the following theorem.

Theorem 2.2 (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*

$$\frac{|b-a|}{2^{n+1}} = \varepsilon$$

where n is the number of iterations that it takes to achieve that tolerance.
  - *Solving for the number of iterations (n) we get*

$$n = \log_2\left(\frac{|b-a|}{\varepsilon}\right) - 1.$$

  - *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 2.18. 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 2.19. Create a second version of your Python Bisection Method function that uses a `for` loop that takes the optimal number of steps to approximate the root to within some tolerance. This should be in contrast to your first version which likely used a `while` loop to decide when to stop. Is there an advantage to using one of these version of the Bisection Method over the other?

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

From Theorem 2.2 you know that the bisection method cuts the interval in half at every iteration. You proved in Exercise 2.17 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 2.1. Let's 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:

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

a. If we plot the absolute error on the vertical axis and the iteration number on the horizontal axis we get Figure 2.2. As expected, the absolute error follows an exponentially decreasing trend. Notice that it isn't a completely smooth curve since we will have some jumps in the accuracy just due to the face that sometimes the root will be near the midpoint of the interval and sometimes it won't be.

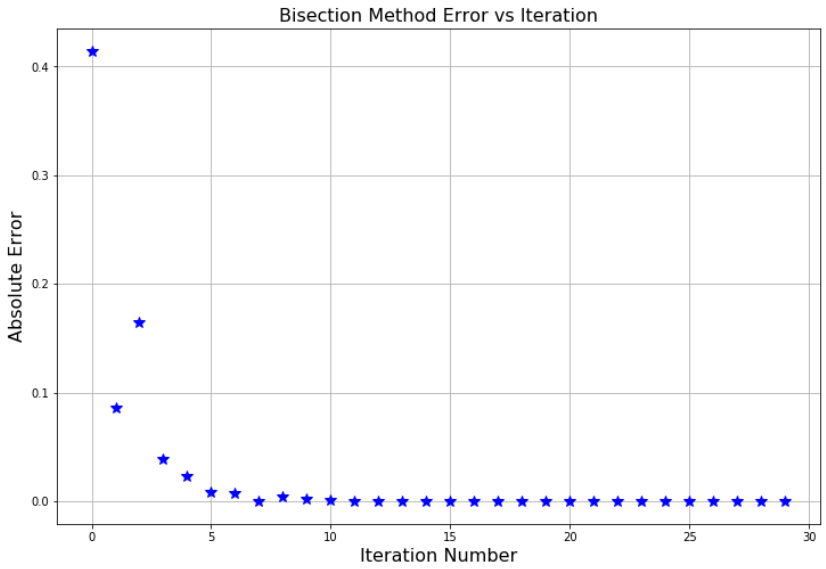


Figure 2.2: The evolution of the absolute error when solving the equation $x^2 - 2 = 0$ with the bisection method.

b. Without Theorem 2.2 it would be rather hard to tell what the exact behavior is in the exponential plot above. We know from Theorem 2.2 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 2.3. There will be times later in this course where we won't have a nice theorem like Theorem 2.2 and instead we will need to deduce the relationship from plots like these.
   i. The trend is linear since logarithms and exponential functions are inverses. Hence, applying a logarithm to an exponential will give a linear function.

ii. The slope of the resulting linear function should be -1 in this case since we are dividing by 1 power 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).

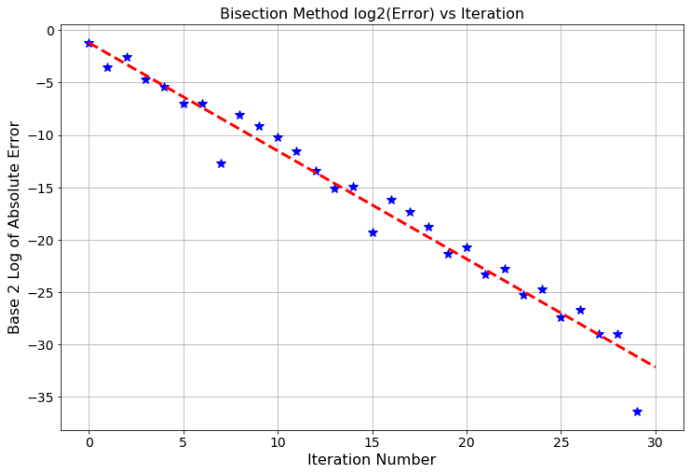


Figure 2.3: Iteration number vs the base-2 logarithm of the absolute error. Notice the slope of -1 indicating that the error is divided by 1 factor of 2 at each step of the algorithm.

c. Another plot that numerical analysts use quite frequently for determining how an algorithm is behaving as it progresses is described by the following bullets:

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

See Figure 2.4 below, but this type of plot takes a bit of explaining the first time you see it. Start on the right-hand side of the plot where the error is the largest (this will be where the algorithm starts). The coordinates of the first point are interpreted as:

$$(\text{absolute error at step } 1\,,\ \text{absolute error at step } 2).$$

The coordinates of the second point are interpreted as:

$$(\text{absolute error at step } 2\,,\ \text{absolute error at step } 3).$$

Etc. 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 the plot below 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

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

Rearranging the algebra a bit we see that this linear relationship turns into

$$\frac{\text{absolute error at step } k+1}{\text{absolute error at step } k} \approx \frac{1}{2}.$$

(You should stop now and do this algebra.) Rearranging a bit more we get

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

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 the plot below!! That's what makes these types of plots so powerful!

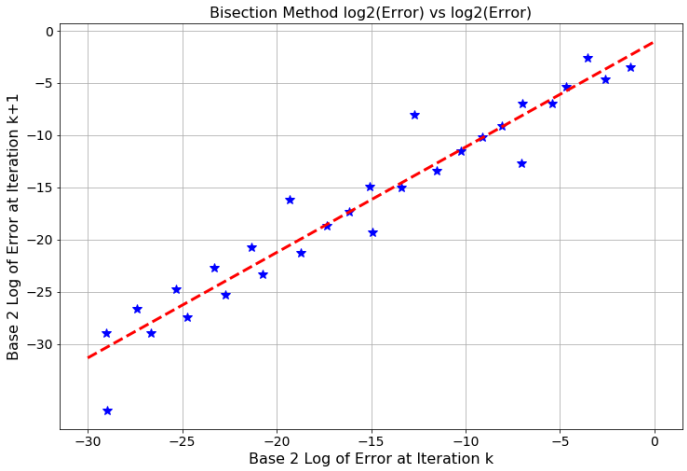


Figure 2.4: The base-2 logarithm of the absolute error at iteration k vs the base-2 logarithm of the absolute error at iteration $k+1$.

d. The final plot that we will make in analyzing the bisection method is the same as the plot that we just made but with the base-10 logarithm instead. See Figure 2.5. In future algorithms we will not know that the error decreases by a factor of 2 so instead we will just try the base-10 logarithm. We will be able to extract the exact same information from this plot. The primary advantage of this last plot is that we can see how the order of magnitude (the power of 10) for the error progresses as the algorithm steps forward. Notice that for every order of magnitude iteration k decreases, iteration $k+1$ decreases by one order of magnitude. That is, the slope of the best fit line in Figure 2.5 is approximately 1. Discuss what this means about how the error in the bisection method behaves as the iterations progress.

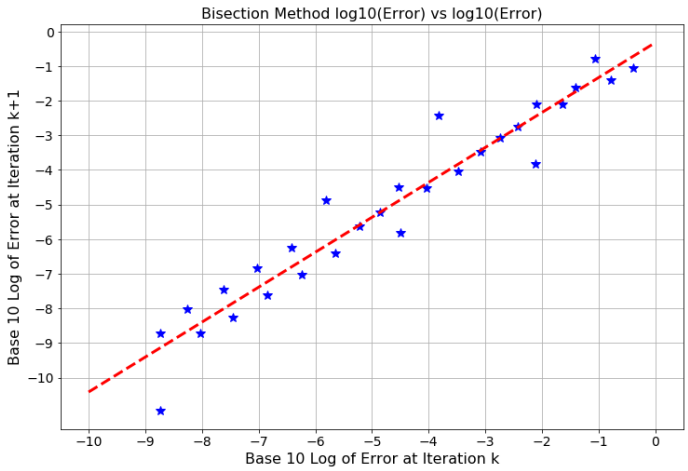


Figure 2.5: The base-10 logarithm of the absolute error at iteration k vs the base-10 logarithm of the absolute error at iteration $k+1$.

Exercise 2.20. Carefully read and discuss all of the details of the previous example and plots. Then create plots similar to this example to solve an equation to which you know the exact solution to. You should see the same basic behavior based on the theorem that you proved in Exercise 2.17. If you don't see the same basic behavior then something has gone wrong.

Hints: You will need to create a modified bisection method function which returns all of the iterations instead of just the final root.

If the logarithms of your absolute errors are in a Python list called `error` then a command like `plt.plot(error[:-1],error[1:],'b*')` will plot the $(k+1)^{st}$ absolute error against the k^{th} absolute error.

If you want the actual slope and intercept of the trend line then you can use `m, b = np.polyfit(error[:-1], error[1:], deg=1)`.

2.3 The Regula Falsi Method

2.3.1 Intuition and Implementation

The bisection method is one of many methods for performing root finding on a continuous function. The next algorithm takes a slightly different approach.

Exercise 2.21. In the Bisection Method, we always used the midpoint of the interval as the next approximation of the root of the function $f(x)$ on the interval $[a, b]$. The three pictures in Figure 2.6 show the same function with three different choices for a and b. Which one will take fewer Bisection-steps to find the root? Which one will take more steps? Explain your reasoning.

(*Note: The root in question is marked with the green star and the initial interval is marked with the red circles.*)

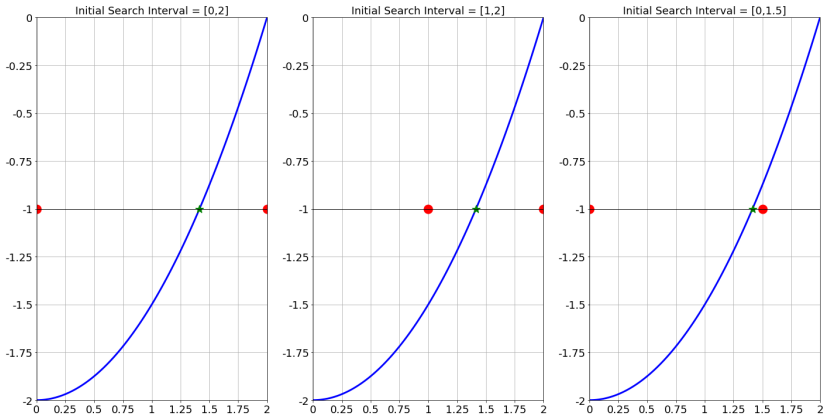


Figure 2.6: In the bisection method you get to choose the starting interval however you like. That choice will make an impact on how fast the algorithm converges to the approximate root.

Exercise 2.22. Now let's modify the Bisection Method approach. Instead of always using the midpoint (which as you saw in the previous problem could take a little while to converge) let's draw a line between the endpoints and use the x-intercept as the updated guess. If we use this method can we improve the speed of convergence on any of the choices of a and b for this function? Which one will now likely take the fewest steps to converge? Figure 2.7 shows three different starting intervals marked in red with the new guess marked as a black X.

The algorithm that you played with graphically in the previous problem is known as the Regula Falsi (false position) algorithm. It is really just a minor tweak on the Bisection method. After all, the algorithm is still designed to use the Intermediate Value Theorem and to iteratively zero in on the root of the function on the given interval. This time, instead of picking the midpoint of the interval that contains the root we draw a line between the function values at either end of the interval and then use the intersection of that line with the x axis as the new approximation of the root. As you can see in Figure 2.7 you might actually

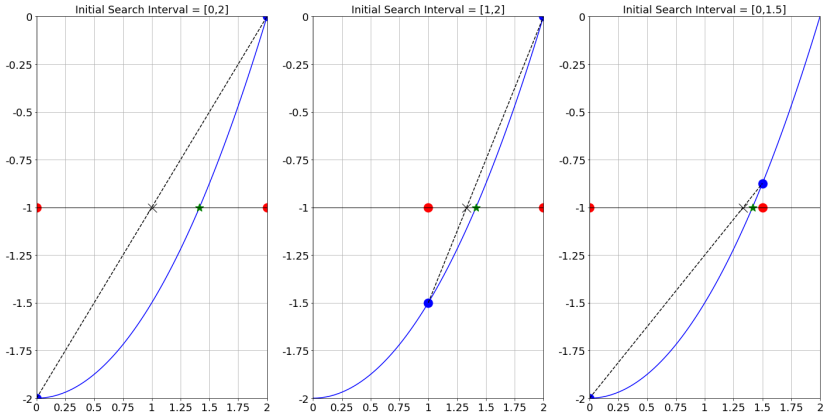


Figure 2.7: In hopes of improving the bisection method we instead propose that we choose the intersection of a line between the endpoints of the interval and the x axis. The intersection (marked with a black X) would be the next approximation instead of the midpoint of the interval.

converge to the approximate root much faster this way (like with the far right plot) or you might gain very little performance (like the far left plot).

Exercise 2.23 (The Regula Falsi Method)**.** Assume that $f(x)$ is continuous on the interval $[a, b]$. To make iterative 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 so that the intermediate value theorem guarantees a root on the interval.
2. We want to write the equation of the line connecting the points $(a, f(a))$ and $(b, f(b))$.
   - What is the slope of this line?

     $$m = \_\_\_\_\_$$

   - Using the point-slope form of a line, $y - y_1 = m(x - x_1)$, what is the equation of the line?

     $$y - \_\_\_\_\_ = \_\_\_\_\_ \cdot (x - \_\_\_\_\_)$$

3. Find the x intercept of the linear function that you wrote in the previous step by setting the y to zero and solving for x. Call this point $x = c$.

   $$c = \_\_\_\_\_$$

   Hint: The x intercept occurs with $y = 0$.

4. Just as we did with the bisection method, compare the signs of $f(a)$ vs $f(c)$ and $f(b)$ vs $f(c)$. Replace one of the endpoints with c. Which one do you replace and why?
5. Repeat steps 2 - 4, and stop when $f(c)$ is *close enough* to zero.

Exercise 2.24. Draw a picture of what the Regula Falsi method does to approximate a root.

Exercise 2.25. Give sketches of functions where the Regula Falsi method will perform faster than the Bisection method and visa versa. Justify your thinking with several pictures and be prepared to defend your answers.

Exercise 2.26. Create a new Python function called `regulafalsi` and write comments giving pseudo-code for the Regula-Falsi method. Remember that starting with pseudo-code is always the best way to start your coding. Write comments that give direction to the code that you're about to write. It is a trap to try and write actual code without any pseudo-code to give you a backbone for the function.

Exercise 2.27. Use your pseudo-code to create a Python function that implements the Regula Falsi method. Write a test script that verifies that your function works properly. Your function should accept a Python function or a Lambda function as input along with an initial lower bound, an initial upper bound, and an optional error tolerance. The output should be only 1 single number: the approximate root.

2.3.2 Analysis

In this subsection we will lean on the fact that we developed a bunch of analysis tools in the Analysis section of the Bisection Method. You may want to go back to that section first and take another look at the plots and tools that we built.

Exercise 2.28. In this problem we are going to solve the equation $x^2 - 2 = 0$ since we know that the exact answer is $x = \sqrt{2}$. You will need to start by modifying your `regulafalsi` function from Exercise 2.26 so that it returns all of the iterations instead of just the root.

a. Start with the interval $[0, 2]$ and solve the equation $x^2 - 2 = 0$ with the Regula-Falsi method.
   i. Find the absolute error between each iteration and the exact answer $x = \sqrt{2}$.

ii. Make a plot of the base-10 logarithm of the absolute error at step k against the base-10 logarithm of the absolute error at step $k+1$. This plot will be very similar to Figure 2.5.

iii. Approximate the slope and intercept of the linear trend in the plot.

$$\log_{10}(\text{abs error at step } k+1) = \_\_\_\_\_ \log_{10}(\text{abs error at step } k+1) + \_\_\_\_\_$$

iv. Based on the work that we did in Example 2.1 estimate the rate of convergence of the Regula-Falsi method.

b. Repeat part (a) with the initial interval $[1, 2]$.

c. Repeat part (a) with the initial interval $[0, 1.5]$.

Exercise 2.29 (Bisection vs Regula-Falsi)**.** Pick a somewhat non-trivial equation where you know the exact answer. Then pick several different starting intervals where you can use both the Bisection Method and the Regula-Falsi Method. Try picking the starting intervals so that some of them converge faster using the Bisection Method and some will converge faster with the Regula-Falsi Method. Show your results with error plots similar to the previous exercise.

Exercise 2.30. Is the Regula-Falsi always better than the bisection method at finding an approximate root for a continuous function that has a known root in a closed interval? Why / why not? Discuss.

2.4 Newton's Method

In the previous two sections we studied techniques for solving equations that required very little sophisticated math. The bisection and regula-falsi methods work very well, but as we'll find in this section we can actually greatly improve the quality of the root-finding algorithms by leveraging some Calculus.

2.4.1 Intuition and Implementation

Exercise 2.31. We will start this section with a reminder from Differential Calculus.

a. If $f(x)$ is a differentiable function at $x = x_0$ then the slope of the tangent line to $f(x)$ at $x = x_0$ is

$$\text{Slope of Tangent Line to } f(x) \text{ at } x = x_0 \text{ is } m = \_\_\_\_\_$$

b. From algebra, the point-slope form of a line is

$$y - y_0 = m(x - x_0)$$

where (x_0, y_0) is a point on the line and m is the slope.

c. If $f(x)$ is a differential function at $x = x_0$ then the equation of the tangent to $f(x)$ at that point is

$$y - \_\_\_\_\_ = \_\_\_\_\_ \cdot (x - \_\_\_\_\_)$$

d. If we rearrange the answer from part (c) we get

$$y = \_\_\_\_\_ + \_\_\_\_\_ \cdot (x - \_\_\_\_\_)$$

The x-intercept of a function is where the function is 0. Root finding is really the process of finding the x-intercept of the function. If the function is complicated (e.g. highly nonlinear or doesn't 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 2.32. A tangent line approximation to a function $f(x)$ near a point $x = x_0$ is

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

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

$$x\text{-intercept of tangent line is } x = \_\_\_\_\_$$

Exercise 2.33. Now let's use the computations you did in the previous exercises to look at an algorithm for approximating the root of a function. In the following sequence of plots we do the following algorithm:

- 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. Take a couple more steps graphically. Does the algorithm appear to converge to the root? Do you think that this will generally take more or fewer steps than the Bisection Method?

Exercise 2.34. If we had started at $x = 0$ in the previous problem what would have happened? Would this initial guess have worked to eventually approximate the root?

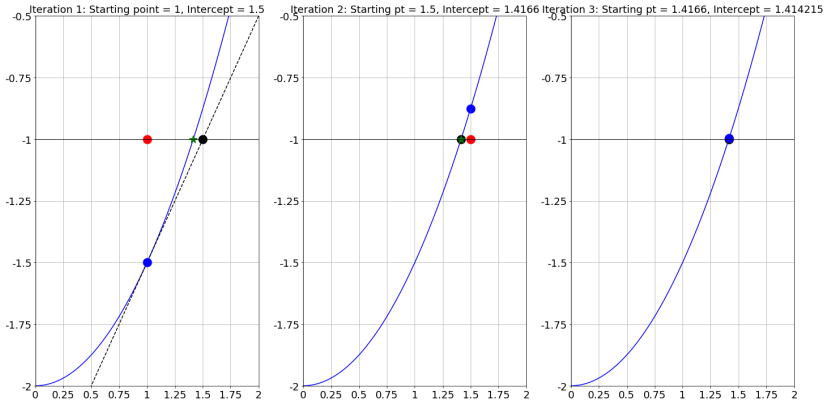


Figure 2.8: Using successive tangent line approximations to find the root of a function

---

Exercise 2.35. Make a complete list of what you must know about the function $f(x)$ for the previous algorithm to work?

The algorithm that we just played with is known as Newton's Method. The method was originally proposed by Isaac Newton, and later modified by Joseph Raphson, for approximating roots of the equation $f(x) = 0$. It should be clear that Newton's method requires the existence of the first derivative so we are asking a bit more of our functions than we were before. In Bisection and Regula Falsi we only asked that the functions be continuous, now we're asking that they be differentiable. Stop and think for a moment ... why is this a more restrictive thing to ask for of the function $f(x)$?

Exercise 2.36 (Newton's Method)**.** The Newton-Raphson method for solving equations can be described as follows:

1. Check that f is differentiable on a given domain and find a way to guarantee that f has a root on that domain (this step happens by hand, not on the computer).
2. Pick a starting point x_0 in the domain
3. We want to write the equation of a tangent line to f at the point $(x_0, f(x_0))$.
   i. What is the slope of the tangent line to the function $f(x)$ at the point $(x_0, f(x_0))$?
   $$m_{tangent} = \_\_\_\_\_$$
   ii. Using the point-slope form of a line, $y - y_1 = m(x - x_1)$, write the

equation of the tangent line to $f(x)$ at the point $(x_0, f(x_0))$.

$$y - \_\_\_\_\_ = \_\_\_\_\_ \cdot (x - \_\_\_\_\_)$$

4. Find the x intercept of the equation of the tangent line by setting $y = 0$ and solving for x. Call this new point x_1.

$$x_1 = \_\_\_\_\_$$

5. Now iterate the process by replacing the labels "x_1" and "x_0" in the previous step with x_{n+1} and x_n respectively.

$$x_{n+1} = \_\_\_\_\_$$

6. Iterate step 5 until $f(x_n)$ is *close* to zero.

Exercise 2.37. Draw a picture of what Newton's method does graphically.

Exercise 2.38. Create a new Python function called `newton()` and write comments giving pseudo-code for Newton's method. Your function needs to accept a Python function for $f(x)$, a Python function for $f'(x)$, an initial guess, and an optional error tolerance. You don't need to set aside any code for calculating the derivative.

Exercise 2.39. Using your pseudocode from the previous problem, write the full `newton()` function. The only output should be the solution to the equation that you are solving. Write a test script to verify that your Newton's method code indeed works.

2.4.2 Analysis

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

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

Some of the failures that we'll 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 and Regula-Falsi methods.

Exercise 2.40. 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 2.41. 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 2.42. 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 and draw the tangent lines on the figure for approximating the solution to $f(x) = 0$ with a starting point of $x = 0$.

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

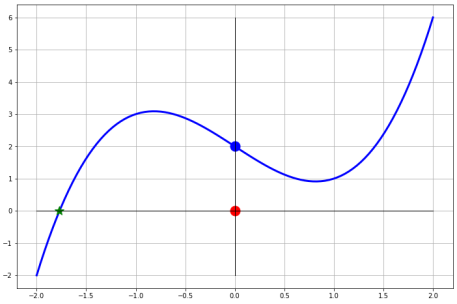


Figure 2.9: An interesting Newton's Method failure.

Exercise 2.43. Now let's 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

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

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

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

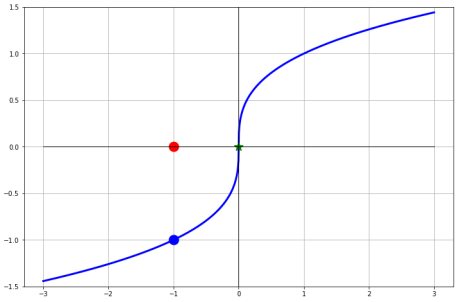


Figure 2.10: Another surprising Newton's Method failure.

b. Now let's 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

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

What does this tell us about the Newton iterations?
Hint: You should have found the exact same thing in the numerical experiment in part (a).

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

Exercise 2.44. Repeat the previous exercise with the function $f(x) = x^3 - 5x$ with the starting point $x_0 = -1$.

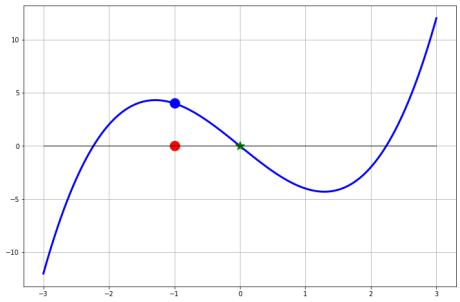


Figure 2.11: Another surprising Newton's Method failure.

Exercise 2.45. Newton's Method is known to have a *quadratic convergence rate*. This means that there is some constant C such that

$$|x_{k+1} - x_*| \leq C|x_k - x_*|^2,$$

where x_* is the root that we're hunting for.

The quadratic convergence implies that if we plot 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 then we will see a constant slope of 2. To see this we can take the log (base 10) of both sides of the previous equation to get

$$\log(|x_{k+1} - x_*|) = \log(C) + 2\log(|x_k - x_*|),$$

and we see that this is a linear function (on a log-log plot) and the slope is 2. We created plots like this back in Example 2.1.

We are going to create an error plot just like what we just described.

a. Pick an equation where you know the solution.

b. Create the error plot with $|x_k - x_*|$ on the horizontal axis and $|x_{k+1} - x_*|$ on the vertical axis
c. Demonstrate that this plot has a slope of 2.
d. Give a thorough explanation for how to interpret the plot that you just made.
e. When solving an equation with Newton's method Joe found that the absolute error at iteration 1 of the process was 0.15. Based on the fact that Newton's method is a second order method this means that the absolute error at step 2 will be less than or equal to some constant times $0.15^2 = 0.0225$. Similarly, the error at step 3 will be less than or equal to some scalar multiple of $0.0025^2 = 0.00050625$. What would Joe's expected error be bounded by for the fourth iteration, fifth iteration, etc?

2.5 The Secant Method

2.5.1 Intuition and Implementation

Newton's Method has second-order (quadratic) convergence and, as such, will perform faster than the Bisection and Regula-Falsi methods. 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

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

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 a 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 2.46 (The Secant Method)**.** Assume that $f(x)$ is continuous and we wish to solve $f(x) = 0$ for x.

1. Determine if there is a root *near* an arbitrary starting point x_0. How might you do that?
2. Pick a second starting point *near* x_0. Call this second starting point x_1. Note well that the points x_0 and x_1 should be close to each other. Why? (The choice here is different than for the Bisection and Regula Falsi methods. We are *not* choosing the left- and right- sides of an interval surrounding the root.)
3. Use the backward difference
$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$
to approximate the derivative of f at x_n. Discuss why this approximates the derivative.
4. Perform the Newton-type iteration
$$x_{n+1} = x_n - \frac{f(x_n)}{\left(\frac{f(x_n)-f(x_{n-1})}{x_n - x_{n-1}}\right)}$$

until $f(x_n)$ is *close enough* to zero. Notice that the new iteration simplifies to

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

Exercise 2.47. Draw several pictures showing what the Secant method does pictorially.

Exercise 2.48. Write pseudo-code to outline how you will implement the Secant Method.

Exercise 2.49. Write Python code 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.

2.5.2 Analysis

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

Exercise 2.50. Choose a non-trivial equation for which you know the solution and write a script to empirically determine the convergence rate of the Secant method.

2.6 Exercises

2.6.1 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. Don't just write "how" the method works, but give all of the mathematical details for "why" it works.

Exercise 2.51. 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 2.52. 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 Regula Falsi Method approximates the root of the function $f(x)$ in the interval $[a, b]$.

Exercise 2.53. 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 2.54. 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 $f(x)$.

2.6.2 Applying What You've Learned

Exercise 2.55. How many iterations of the bisection method are necessary to approximate $\sqrt{3}$ to within 10^{-3}, 10^{-4}, ..., 10^{-15} using the initial interval $[a, b] = [0, 2]$?

Exercise 2.56. Refer back to Example 2.1 and demonstrate that you get the same results by solving the problem $x^3 - 3 = 0$. Generate versions of all of the plots from the Example and give thorough descriptions of what you learn from each plot.

Exercise 2.57. 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

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

Write a script that calls upon your Bisection, Regula Falsi, 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 can from which root finding algorithm.

Exercise 2.58. A root-finding method has a convergence rate of order M if there is a constant C such that

$$|x_{k+1} - x_*| \leq C|x_k - x_*|^M.$$

Here, x_* is the exact root, x_k is the k^{th} iteration of the root finding technique, and x_{k+1} is the $(k+1)^{st}$ iteration of the root finding technique.

a. If we consider the equation

$$|x_{k+1} - x_*| \leq C|x_k - x_*|^M$$

and take the logarithm (base 10) of both sides then we get

$$\log\left(|x_{k+1} - x_*|\right) \leq \_\_\_\_\_ + \_\_\_\_\_$$

b. In part (a) you should have found that the log of new error is a linear function of the log of the old error. What is the slope of this linear function on a log-log plot?

c. In the plots below you will 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?

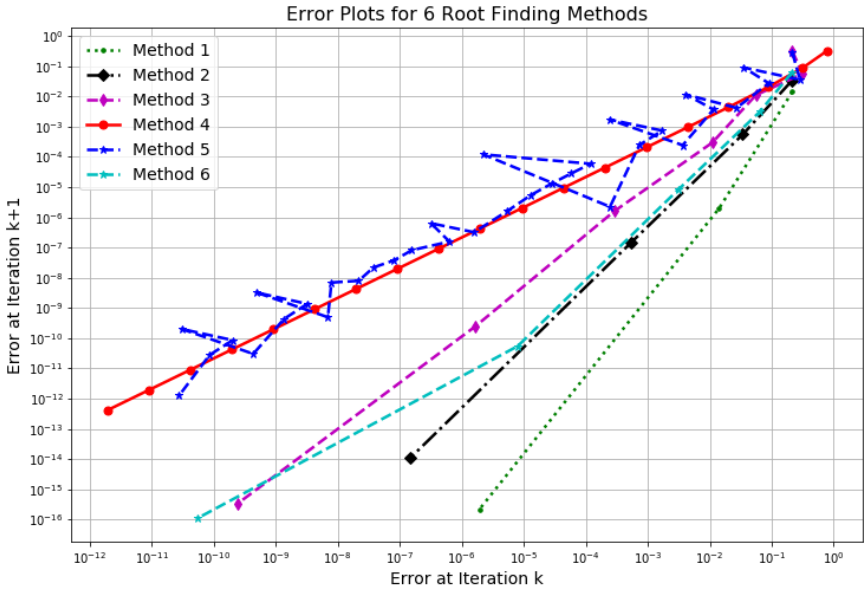


Figure 2.12: Six Error Plots

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 2.59. Shelby started using Newton's method to solve a root-finding problem. To test her code she was using an equation for which she new the solution. Given the starting point the absolute error after one step of Newton's

method was $|x_1 - x_*| = 0.2$. What is the approximate expected error at step 2? What about at step 3? Step 4? Defend your answers by fully describing your thought process.

Exercise 2.60. 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

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

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're looking for. Therefore, to find the next iterate we solve the equation

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

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

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

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 2.61. (modified from [5]) 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 a function is

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

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.

a. What are the units of the parameter k?

b. If $m = 1\text{kg}$, $g = 9.8\text{m/s}^2$, $k = 0.1$, 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.

c. 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 2.62. Can the Bisection Method, Regula Falsi 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?

Exercise 2.63. 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're 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 techinque!
Find the local extrema for the function $f(x) = x^3(x-3)(x-6)^4$ using numerical techniques where appropriate.

Exercise 2.64. A *fixed point* of a function $f(x)$ is a point that solves the equation $f(x) = x$. Fixed points are interesting in iterative processes since fixed points don't change under repeated application of the function f.

For example, consider the function $f(x) = x^2 - 6$. The fixed points of $f(x)$ can be found by solving the equation $x^2 - 6 = x$ which, when simplified algebraically, is $x^2 - x - 6 = 0$. Factoring the left-hand side gives $(x-3)(x+2) = 0$ which implies that $x = 3$ and $x = -2$ are fixed points for this function. That is, $f(3) = 3$ and $f(-2) = -2$. Notice, however, that finding fixed points is identical to a root finding problem.

a. Use a numerical root-finding algorithm to find the fixed points of the function $f(x) = x^2 - 6$ on the interval $[0, \infty)$.

b. Find the fixed points of the function $f(x) = \sqrt{\frac{8}{x+6}}$.

Exercise 2.65 (‘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.

a. Go to the help documentation for `scipy.optimize.fsolve` and make yourself familiar with how to use the tool.
b. First solve the equation $x\sin(x) - \ln(x) = 0$ for x starting at $x_0 = 3$.
   i. Make a plot of the function on the domain $[0, 5]$ so you can eyeball the root before using the tool.
   ii. Use the `scipy.optimize.fsolve()` command to approximate the root.
   iii. 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.
c. Demonstrate how to use `fsolve()` using any non-trivial nonlinear equation solving problem. Demonstrate what some of the options of `fsolve()` do.
d. 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

$$x_0\cos(x_1) = 4$$
$$x_0x_1 - x_1 = 5$$

   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.
e. Solve the system of nonlinear equations below using `scipy.optimize.fsolve()`.

$$x^2 - xy^2 = 2$$
$$xy = 2$$

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

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

# Or alternately we could define the system as a lambda function with
# F = lambda x: [ x[0]*np.cos(x[1])-4 , x[0]*x[1]-x[1]-5 ]

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

2.7 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 one 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. Take the time to read Appendix B before you write your final paper.

2.7.1 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 aren't 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 colored 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 color.
- Pick a starting point on the x axis and use it to start Newton's Method.
- Color the starting point according to the root that it converges to.
- Repeat this process for many many starting points so you get a colored picture of the x axis showing where the starting points converge to.

The set of points that are all the same color are called the **basin of attraction** for the root associated with that color. In Figure 2.13 there is an image of a sample basin of attraction image.

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.

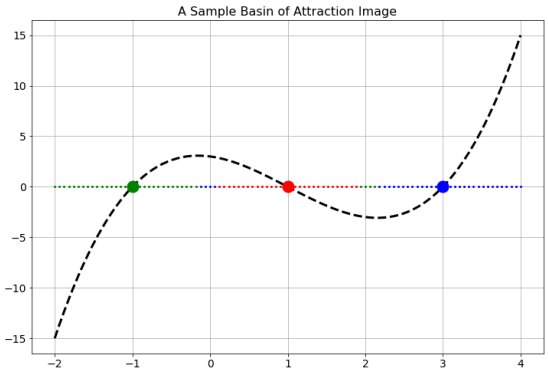


Figure 2.13: A sample basin of attraction image for a cubic function.

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 $z_0 = 1$, $z_1 = \frac{1}{2}\left(-1 + \sqrt{3}i\right)$, and $z_2 = \frac{1}{2}\left(-1 - \sqrt{3}i\right)$ (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 favorite 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.

2.7.2 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 resistance[1] so for all time differential equations $v_y'(t) = -g$ and $v_x'(t) = 0$ must both hold.

[1]Strictly 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 math and makes this version of the problem more tractable. The second version of the artillery problem in Chapter 5 will look at the effects of

- 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.

Use the given information to write a nonlinear equation[2] 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.

air resistance on the cannon ball.

[2]Hint: 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_*.

Chapter 3

Calculus

3.1 Intro to Numerical Calculus

> *The calculus was the first achievement of modern mathematics and it is difficult to overestimate its importance.*
> –Hungarian-American Mathematician John von Neumann

In this chapter we build some of the common techniques for approximating the two primary computations in calculus: taking derivatives and evaluating definite integrals. Beyond differentiation and integration one of the major applications of differential calculus was optimization. The last several sections of this chapter focus on numerical routines for approximating the solutions to optimization problems.

Recall the typical techniques from differential calculus: the power rule, the chain rule, the product rule, the quotient rule, the differentiation rules for exponentials, inverses, and trig functions, implicit differentiation, etc. With these rules, and enough time and patience, we can find a derivative of any algebraically defined function. The truth of the matter is that not all functions are given to us algebraically, and even the ones that are given algebraically are sometimes really cumbersome.

Exercise 3.1. A water quality engineering team wants to find the rate at which the volume of waste water is changing in their containment pond throughout the year. They presently only have data on the specific geometric shape of the containment pond as well as the depth of the waste water each day for the past year. Propose several methods for approximating the first derivative of the volume of the waste water pond.

Exercise 3.2. When a police officer fires a radar gun at a moving car it uses a laser to measure the distance from the officer to the car:

- The speed of light is constant.
- The time between when the laser is fired and when the light reflected off of the car is received can be measured very accurately.
- Using the formula distance = rate $\cdot$ time, the time for the laser pulse to be sent and received can then be converted to a distance.

How does the radar gun then use that information to calculate the speed of the moving car?

Integration, on the other hand, is a more difficult situation. You may recall some of the techniques of integral calculus such as the power rule, u-substitution, and integration by parts. However, these tools are not enough to find an antiderviative for any given function. Furthermore, not every function can be written algebraically.

Exercise 3.3. In statistics the function known as *the normal distribution* (the bell curve) is defined as

$$N(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

One of the primary computations of introductory statistics is to find the area under a portion of this curve since this area gives the probability of some event

$$P(a < x < b) = \int_a^b \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx.$$

The trouble is that there is no known antiderivative of this function. Propose a method for approximating this area.

Exercise 3.4. Give a list of five functions for which an exact algebraic derivative is relatively easy but an exact antiderivative is either very hard or maybe impossible. Be prepared to compare with your peers.

Exercise 3.5. A dam operator has control of the rate at which water is flowing out of a hydroelectric dam. He has records for the approximate flow rate through the dam over the course of a day. Propose a way for the operator to use his data to determine the total amount of water that has passed through the dam during that day.

What you've seen here are just a few examples of why you might need to use numerical calculus instead of the classical routines that you learned earlier in your mathematical career. Another typical need for numerical derivatives and integrals arises when we approximate the solutions to differential equations in the later chapters of this book.

Throughout this chapter we will make heavy use of Taylor's Theorem to build approximations of derivatives and integrals. If you find yourself still a bit shaky on Taylor's Theorem it would probably be wise to go back to Section 1.4 and do a quick review.

At the end of the chapter we'll examine a numerical technique for solving optimization problems without explicitly finding derivatives. Then we'll look at a common use of numerical calculus for fitting curves to data.

3.2 Differentiation

3.2.1 The First Derivative

Exercise 3.6. Recall from your first-semester Calculus class that the derivative of a function $f(x)$ is defined as

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

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:

a. When might the Calculus student's proposal actually work pretty well in place of calculating an actual derivative?

b. When might the Calculus student's proposal fail in terms of approximating the derivative?

In this section we'll 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 h.

- Approximate the derivative at any point x in the interval $[a, b]$ by using linear combinations of $f(x - h)$, $f(x)$, $f(x + h)$, 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 $[a, b]$.

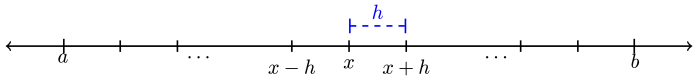


Figure 3.1: A partition of the interval $[a, b]$.

Figure 3.1 shows a depiction of the partition as well as making clear that h is the separation between each of the points in the partition. Note that in general the points in the partition do not need to be equally spaced, but that is the simplest place to start.

Exercise 3.7. Let's take a close look at partitions before moving on to more details about numerical differentiation.

a. If we partition the interval $[0, 1]$ into 3 equal sub intervals each with length h then:
   i. $h =$ _____
   ii. $[0, 1] = [0, \_\_\_\_] \cup [\_\_\_\_, \_\_\_\_] \cup [\_\_\_\_, 1]$
   iii. There are four total points that define the partition. They are $0, ??, ??, 1$.

b. If we partition the interval $[3, 7]$ into 5 equal sub intervals each with length h then:
   i. $h =$ _____
   ii. $[3, 7] = [3, \_\_\_\_] \cup [\_\_\_\_, \_\_\_\_] \cup [\_\_\_\_, \_\_\_\_] \cup [\_\_\_\_, \_\_\_\_] \cup [\_\_\_\_, 7]$
   iii. There are 6 total points that define the partition. They are $0, ??, ??, ??, ??, 7$.

c. More generally, if a closed interval $[a, b]$ contains N equal sub intervals where

$$[a, b] = \underbrace{[a, a+h] \cup [a+h, a+2h] \cup [a+2h, a+3h] \cup \cdots \cup [b-2h, b-h] \cup [b-h, b]}_{N \text{ total sub intervals}}$$

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

$$h = \frac{??-??}{??}.$$

Exercise 3.8. 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 $n-1$ 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)`. What command would you use to partition the interval $[5,10]$ into 100 equal sub intervals?

```
import numpy as np
np.linspace(0,1,5)
```

```
## array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

```
import numpy as np
np.linspace(0,1,20)
```

```
## array([0.        , 0.05263158, 0.10526316, 0.15789474, 0.21052632,
##        0.26315789, 0.31578947, 0.36842105, 0.42105263, 0.47368421,
##        0.52631579, 0.57894737, 0.63157895, 0.68421053, 0.73684211,
##        0.78947368, 0.84210526, 0.89473684, 0.94736842, 1.        ])
```

Exercise 3.9. Consider the Python command `np.linspace(0,1,50)`.

a. What interval does this command partition?
b. How many points are going to be returned?
c. How many equal length subintervals will we have in the resulting partition?
d. What is the length of each of the subintervals in the resulting partition?

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

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

our first approximation for the first derivative is naturally

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x)}{h}.$$

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 h is *small*. In Figure 3.2 we see a graphical depiction of what we're doing to approximate the derivative. The slope of the tangent line ($\Delta y / \Delta x$) is what we're 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.

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 suggests an infinite family of other techniques.

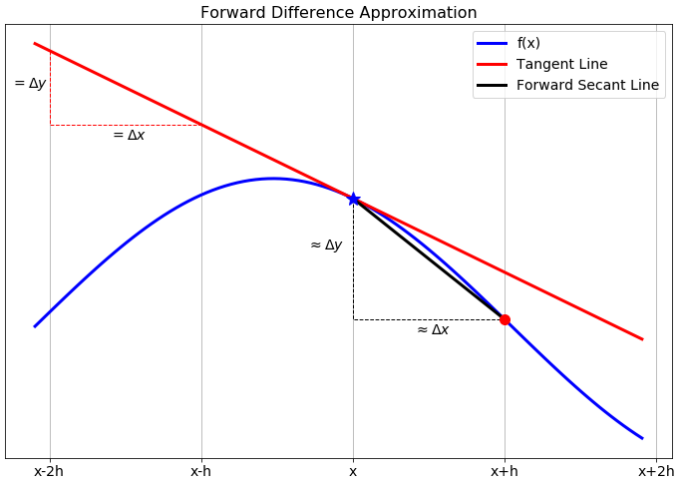


Figure 3.2: The forward difference differentiation scheme for the first derivative.

Exercise 3.10. From Taylor's Theorem we know that for an infinitely differentiable function $f(x)$,

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x-x_0)^1 + \frac{f''(x_0)}{2!}(x-x_0)^2 + \frac{f^{(3)}(x_0)}{3!}(x-x_0)^3 + \frac{f^{(4)}(x_0)}{4!}(x-x_0)^4 + \cdots .$$

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 3.1 we want to center the Taylor series at x and evaluate the resulting series at the point $x+h$.

$$f(x+h) = \_\_\_\_\_$$

Exercise 3.11. Solve the result from the previous problem 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)

$$f'(x) = \frac{f(x+h) - ???}{??} + \_\_\_\_\_$$

Exercise 3.12. In the formula that you developed in Exercise 3.11, if we were to drop everything after the fraction (called the *remainder*) we know that we would be introducing 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 3.11?

Definition 3.1 (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.

The error that you make by dropping the remainder is proportional to the power of h in the first term of the remainder. Hence, the **order** of a numerical differentiation scheme tells you how to quantify the amount of error that you are making by using that approximation scheme.

Definition 3.2 (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

$$|Error| \leq M \cdot h.$$

This is equivalent to saying that a differentiation method is "first order". In other words, if the error in a numerical differentiation scheme is proportional to the length of the subinterval in the partition of the interval (see Figure 3.1) then we call that scheme "first order" and say that the error is $\mathcal{O}(h)$.

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

$$|Error| \leq M \cdot h^k.$$

This is equivalent to saying that a differentiation scheme is "k^{th} order". This means that the error in using the scheme is proportional to h^k.

Theorem 3.1. *In problem 3.11 we derived a first order approximation of the first derivative:*

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

In this formula, $h = \Delta x$ is the step size.

If we approximate the first derivative of a differentiable function $f(x)$ at the point x with the formula

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

then we know that the error in this approximation is proprotional to h since the approximation scheme is $\mathcal{O}(h)$.

3.2.2 Error Analysis

Exercise 3.13. Consider the function $f(x) = \sin(x)(1-x)$. The goal of this problem 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 our first order approximation formula from Theorem 3.1 we calculate

$$f'(1) \approx \frac{f(1+h) - f(1)}{h}.$$

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're working in radians!).

h	Approx. of $f'(1)$	Exact value of $f'(1)$	Abs. % Error
$2^{-1} = 0.5$	$\frac{f(1+0.5)-f(1)}{0.5} \approx -0.99749$	$-\sin(1) \approx -0.841471$	18.54181%
$2^{-2} = 0.25$	$\frac{f(1+0.25)-f(1)}{0.25} \approx -0.94898$	$-\sin(1) \approx -0.841471$	12.77687%
$2^{-3} = 0.125$		$-\sin(1)$	
$2^{-4} = 0.0625$		$-\sin(1)$	
2^{-5}		$-\sin(1)$	
2^{-6}		$-\sin(1)$	
2^{-7}		$-\sin(1)$	
2^{-8}		$-\sin(1)$	
2^{-9}		$-\sin(1)$	
2^{-10}		$-\sin(1)$	

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:

$h = 3^{-1}, 3^{-2}, 3^{-3}, \ldots$ (start with $h = 1/3$ and divide h by 3 for each new approx.)

$h = 5^{-1}, 5^{-2}, 5^{-3}, \ldots$ (start with $h = 1/5$ and divide h by 5 for each new approx.)

$h = 10^{-1}, 10^{-2}, 10^{-3}, \ldots$ (start with $h = 1/10$ and divide h by 10 for each new approx.)

$h = \pi^{-1}, \pi^{-2}, \pi^{-3}, \ldots$ (start with $h = 1/\pi$ and divide h by π for each new approx.)

e. Observation: If you calculate a numerical derivative with a forward difference and then calculate the absolute percent 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 contant M?

f. What does your answer to part (e) have to do with the approximation order of the numerical derivative method that you used?

Exercise 3.14. The following incomplete block of Python code will help to streamline the previous problem so that you don't 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 problem.
c. Run the code for several forms of h
d. Do you still observe the same result that you observed in part (e) of the previous problem?
e. We know that for $h \to 0$ the derivative approximation should mathematically tend toward the exact derivative. Modify the code slightly to see if this is the case. Explain what you see.

```
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?
AbsPctError = [] # start off with a blank list of errors
for h in H:
    approx = # FINISH THIS LINE OF CODE
    AbsPctError.append( np.abs( (approx - exact)/exact ) )
    print("h=",h,"\t Absolute Pct Error=", AbsPctError[-1])
plt.loglog(H,AbsPctError,'b-*') # Why are we build a loglog plot here?
plt.grid()
plt.show()
```

Exercise 3.15. Assume that $f(x)$ is some differentiable function and that we have calculated the value of $f'(c)$ using the forward difference formula

$$f'(c) \approx \frac{f(c+h) - f(c)}{h}.$$

Using what you learned from the previous problem to fill in the following table.

My h	Absolute Percent Error
0.2	2.83%
0.1	
0.05	

My h	Absolute Percent Error
0.02	
0.002	

Exercise 3.16. Explain the phrase: *The first derivative approximation* $f'(x) \approx \frac{f(x+h)-f(x)}{h}$ *is first order.*

3.2.3 Efficient Coding

Now that we have a handle on how the first order approximation scheme for the first derivative works and how the errors will propagate, let's 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 3.17. 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 a first order 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 a value of the derivative $f'(x)$ at all of the points except at $x = 2\pi$.

a. First of all, why can't we compute a derivative at the last point?
b. 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 FirstDeriv(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
```

c. Now we want to call upon this function to build the first order approximation of the first derivative for some function. We'll 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 `FirstDeriv()` 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 this line do?  What's up with the N+1?

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

# In the next line we plot three curves:
# 1) the function f(x) = sin(x)
# 2) the function f'(x) = cos(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()
```

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

Exercise 3.18. Now let's build the first derivative function in a much *smarter* way – using `numpy` lists in Python. Instead of looping over all of the elements we can take advantage of the fact that every thing is stored in lists. Hence we can just do list operations and do all of the subtractions and divisions at once without a loop.

a. From your previous code, comment out the following lines.

```
#     df = []
#     for j in np.arange(len(x)-1):
#         df.append( (f(x[j+1]) - f(x[j])) / h )
```

b. From the line of code `x = np.linspace(a,b,N+1)` we build a list of $N+1$ values of x starting at a and ending at b. 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`.
   i. What will we get if we evaluate the code `x[1:]`?
   ii. What will we get if we evaluate the code `f(x[1:])`?
   iii. What will we get if we evaluate the code `x[0:-1]`?
   iv. What will we get if we evaluate the code `f(x[0:-1])`?
   v. What will we give if we evaluate the code `f(x[1:]) - f(x[0:-1])`?
   vi. What will we give if we evaluate the code `( f(x[1:]) - f(x[0:-1]) ) / h`?

c. Replace the lines that you commented out in part (a) of this exercise with the appropriate single line of code that builds all of the approximations for the first derivative all at once without the need for a loop. What you did in part (b) should help. Your simplified first order first derivative function should look like the code below.

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

Exercise 3.19. 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 problem 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 = FirstDerivFirstOrder(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 all.
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.

3.2.4 A Better First Derivative

Next we'll 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'll be hoping to have a better error approximation.

Exercise 3.20. Consider again the Taylor series for an infinitely differentiable function $f(x)$:

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

a. Replace the "x" in the Taylor Series with "$x+h$" and replace the "x_0" in the Taylor Series with "x" and simplify.

$$f(x+h) = \_\_\_\_\_$$

b. Now replace the "x" in the Taylor Series with "$x-h$" and replace the "x_0" in the Taylor Series with "x" and simplify.

$$f(x-h) = \_\_\_\_\_$$

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

$$f(x+h) - f(x-h) = \_\_\_\_\_$$

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

$$f'(x) = \frac{???-???}{2h} + \underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}.$$

e. Use your result from part (d) to verify that

$$f'(x) = \underline{\qquad\qquad\qquad\qquad\qquad\qquad} + \mathcal{O}(h^2).$$

f. Draw a picture similar to Figure 3.2 showing what this scheme is doing graphically.

Exercise 3.21. Let's 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

$$f'(1) \approx \frac{f(1+h) - f(1-h)}{2h}.$$

You should already have the first derivative and the exact answer from Exercise 3.13 (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're working in radians!).

h	Approx. of $f'(1)$	Exact value of $f'(1)$	Abs. % Error
$2^{-3} = 0.5$		$-\sin(1)$	
$2^{-3} = 0.25$		$-\sin(1)$	
$2^{-3} = 0.125$		$-\sin(1)$	
$2^{-4} = 0.0625$		$-\sin(1)$	
2^{-5}		$-\sin(1)$	
2^{-6}		$-\sin(1)$	
2^{-7}		$-\sin(1)$	
2^{-8}		$-\sin(1)$	
2^{-9}		$-\sin(1)$	

b. 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:

$$h = 3^{-1}, 3^{-2}, 3^{-3}, \ldots$$
$$h = 5^{-1}, 5^{-2}, 5^{-3}, \ldots$$
$$h = 10^{-1}, 10^{-2}, 10^{-3}, \ldots$$
$$h = \pi^{-1}, \pi^{-2}, \pi^{-3}, \ldots$$

c. Observation: If you calculate a numerical derivative with a centered difference and calculate the resulting absolute percent error with a fixed value of h, then what do you expect to happen to the absolute percent error if you divide the value of h by some positive constant M?

d. What does your answer to part (e) have to do with the approximation order of the numerical derivative method that you used?

Exercise 3.22. Assume that $f(x)$ is some differentiable function and that we have calculated the value of $f'(c)$ using the centered difference formula

$$f'(c) \approx \frac{f(c+h) - f(c-h)}{2h}.$$

Using what you learned from the previous problem to fill in the following table.

My h	Absolute Percent Error
0.2	2.83%
0.1	
0.05	
0.02	
0.002	

Exercise 3.23. Write a Python function that takes a mathematical function and an interval and returns a second order numerical approximation to the first derivative on the interval. 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.)

Exercise 3.24. Test the code you wrote in the previous exercise on functions where you know the first derivative.

Exercise 3.25. The plot shown in Figure 3.3 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:

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

and

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

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

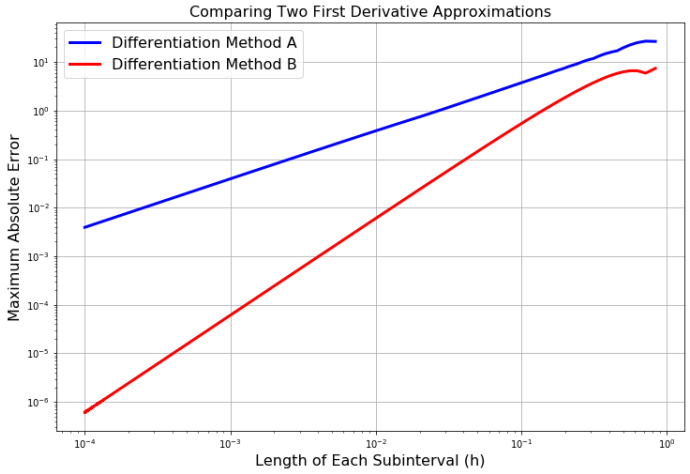


Figure 3.3: Maximum absolute error between the first derivative and two different approximations of the first derivative.

3.2.5 The Second Derivative

Now we'll 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'll do the algebra in such a way that the first derivative cancels.

From the previous problems you already have Taylor expansions of the form $f(x+h)$ and $f(x-h)$. Let's summarize them here since you're going to need them for future computations.

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

Exercise 3.26. The goal of this problem is to use the Taylor series for $f(x+h)$ and $f(x-h)$ to arrive at an approximation scheme for the second derivative $f''(x)$.

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

$f(x+h) + f(x-h) =$ _____.

b. Solve your answer in part (a) for $f''(x)$.

$$f''(x) = \frac{?? - 2\cdot ?? + ??}{h^2} + \text{\_\_\_\_\_}.$$

c. 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 3.27. Again consider the function $f(x) = \sin(x)(1 - x)$.

a. Calculate the derivative of this function and calculate the exact value of $f''(1)$.

b. If we calcuate the second derivative with the central difference scheme that you built in the previous exericse using $h = 0.5$ then we get a 4.115% error. Stop now and verify this percent error calculation.

c. 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 3.28. Write a Python function that takes a mathematical function and a domain and returns a second order numerical approximation to the second derivative on the interval. You should ALWAYS start by writing pseudo-code as comments in your function. As before, you should write your code without using any loops.

Exercise 3.29. Test your second derivative code on the function $f(x) = \sin(x) - x\sin(x)$ by doing the following.

a. Find the analytic second derivative by hand.

b. Find the numerical second derivative with the code that you just wrote.

c. 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.

d. Find the maximum of your errors.

e. 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.

f. 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 5.

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}	$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$	$\mathcal{O}(h^2)$	Centered Difference
2^{nd}	$f''(x) \approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2}$	$\mathcal{O}(h^2)$	Centered Difference

Exercise 3.30. 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 doesn't really matter exactly what function f is.)

a. Johnny used a numerical first derivative scheme with $h = 0.1$ to approximate $f'(1.74)$ and found an abolute percent error of 3.28%. He then used $h = 0.01$ and found an absolute percent error of 0.328%. What was the order of the error in his first derivative scheme? How can you tell?

b. Betty used a numerical first derivative scheme with $h = 0.2$ to approximate $f'(1.74)$ and found an absolute percent error of 4.32%. She then used $h = 0.1$ and found an absolute percent error of 1.08%. What numerical first derivative scheme did she likely use?

c. Shelby did the computation

$$f'(1.74) \approx \frac{f(1.78) - f(1.74)}{0.04}$$

and found an absolute percent error of 2.93%. If she now computes

$$f'(1.74) \approx \frac{f(1.75) - f(1.74)}{0.01}$$

what will the new absolute percent error be?

d. Harry wants to compute $f''(1.74)$ to within 1% using a central difference scheme. He tries $h = 0.25$ and gets an absolute percent error of 3.71%. What h should he try next so that his absolute percent error is less than (but close to) 1%?

3.3 Integration

Now we begin our work on the second principle computation of Calculus: evaluating a definite integral. Remember that a single-variable definite integral can be interpreted as the signed area between the curve and the x axis. In this section we will study three different techniques for approximating the value of a definite integral.

Exercise 3.31. Consider the shaded area of the region under the function plotted in Figure 3.4 between $x = 0$ and $x = 2$.

a.What rectangle with area 6 gives an upper bound for the area under the curve? Can you give a better upper bound? b. Why must the area under the curve be greater than 3? c. Is the area greater than 4? Why/Why not? d. Work with your partner to give an estimate of the area and provide an estimate for the amount of error that you're making.

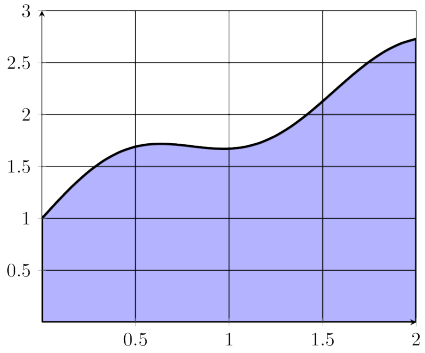


Figure 3.4: A sample integration

3.3.1 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

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

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, if Δx

is small).

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

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 endpoint, the midpoint, and the right endpoint.

We see a depiction of this in Figure 3.5.

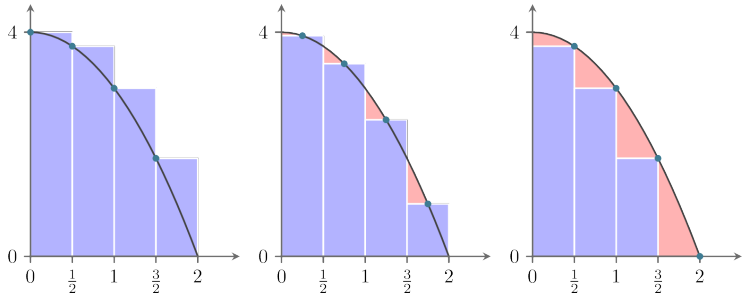


Figure 3.5: Right-aligned Riemann sums, midpoint-aligned Riemann sums, and left-aligned Riemann sums

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

Exercise 3.32. Write code to approximate an integral with Riemann sums. You should ALWAYS start by writing pseudo-code as comments in your function. Your Python function should accept a Python Function, a lower bound, an upper bound, the number of subintervals, and an optional input 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 3.33. Consider the function $f(x) = \sin(x)$. We know the antiderivative for this function, $F(x) = -\cos(x) + C$, but in this question we are going to get a sense of the order of the error when doing Riemann Sum integration.

a. Find the exact value of

$$\int_0^1 f(x)dx.$$

b. Now build a Riemann Sum approximation (using your code) with various values of Δx. For all of your approximation use left-justified rectangles. Fill in the table with your results.

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

c. There was nothing really special about powers of 2 in part (b) of this problem. 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 percent error, then what would happen to the absolute percent error if we divide Δx by some positive constant M?

d. What is the apparent approximation error of the Riemann Sum method using left-justified rectangles.

Exercise 3.34. Repeat the previous problem using right-justified rectangles.

Theorem 3.2. *In approximating the integral $\int_a^b f(x)dx$ with a fixed interval width Δx we find an absolute percent error P.*

- *If we use left rectangles and an interval width of $\frac{\Delta x}{M}$ then the absolute percent error will be approximately _____.*
- *If we use right rectangles and an interval width of $\frac{\Delta x}{M}$ then the absolute percent error will be approximately _____.*

Exercise 3.35. The previous theorem could be stated in an equivalent way.

In approximating the integral $\int_a^b f(x)dx$ with a fixed interval number of subintervals we find an absolute percent error P.

- If we use left rectangles and M times as many subintervals then the absolute percent error will be approximately _____.
- If we use right rectangles and M times as many subintervals then the absolute percent error will be approximately _____.

Exercise 3.36. 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. 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?

3.3.2 Trapezoidal Rule

Now let's 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 Trapezoid rule consider Figure 3.6. 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.

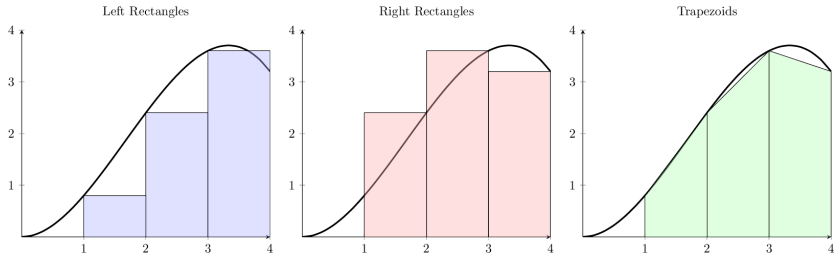


Figure 3.6: Motivation for using trapezoids to approximate a definite integral.

Exercise 3.37. Consider a single trapezoid approximating the area under a curve. From geometry we recall that the area of a trapezoid is

$$A = \frac{1}{2}\left(b_1 + b_2\right) h$$

where b_1, b_2 and h are marked in Figure 3.7. 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

$$\int_1^4 \left(\frac{1}{5}x^2(5-x)\right) dx.$$

Now use the same idea with $h = \Delta x = 1$ from Figure 3.6 to approximate the area under the function $f(x) = \frac{1}{5}x^2(5-x)$ between $x = 1$ and $x = 4$ using three trapezoids.

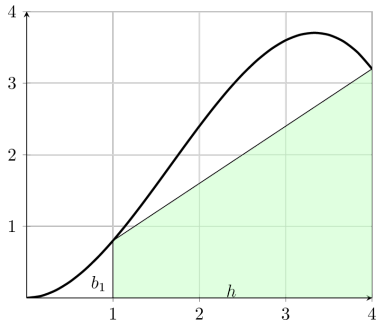


Figure 3.7: A single trapezoid to approximate area under a curve.

Exercise 3.38. Again consider the function $f(x) = \frac{1}{5}x^2(5-x)$ on the interval $[1, 4]$. We want to evaluate the integral

$$\int_1^4 f(x)dx$$

using trapezoids to approximate the area.

a. Work out the exact value of the definite integral by hand.

b. Summarize your answers to the previous problems in the following table then extend the data that you have for smaller and smaller values of Δx.

Δx	Approx. Integral	Exact Integral	Abs. % Error
3			
1			
1/3			
1/9			
⋮	⋮	⋮	⋮

c. From the table that you built in part (b), what do you conjecture is the order of the approximation error for the trapezoid method?

Definition 3.3 (The Trapezoidal Rule)**.** We want to approximate $\int_a^b f(x)dx$. One of the simplest ways is to approximate the area under the function with a trapezoid. Recall from basic geometry that area of a trapezoid is $A = \frac{1}{2}(b_1+b_2)h$. In terms of the integration problem we can do the following:

a. First partition $[a, b]$ into the set $\{x_0 = a, x_1, x_2, \ldots, x_{n-1}, x_n = b\}$.

b. On each part of the partition approximate the area with a trapezoid:

$$A_j = \frac{1}{2}\left[f(x_j) + f(x_{j-1})\right](x_j - x_{j-1})$$

c. Approximate the integral as

$$\int_a^b f(x)dx = \sum_{j=1}^{n} A_j$$

Exercise 3.39. Write code to give the trapezoidal rule approximation for the definite integral $\int_a^b f(x)dx$. Test your code on functions where you know the definite area. Then test your code on functions where you have approximated

---

Exercise 3.40. Use the code that you wrote in the previous problem to test your conjecture about the order of the approximation error for the trapezoid rule. Integrate the function $f(x) = \sin(x)$ from $x = 0$ to $x = 1$ with more and more trapezoids. In each case compare to the exact answer and find the absolute percent error. The goal is to answer the question:
If we calculate the definite integral with a fixed Δx and get an absolute percent error, P, then what absolute percent error will we get if we use a width of $\Delta x/M$ for some positive number M?

3.3.3 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. In order to do this we need to evaluate the function at three points (not two like the trapezoidal rule). Let's integrate a function $f(x)$ on the interval $[a, b]$ by using 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 such that the integral $\int_a^b f(x)dx$ can be written as a linear combination of $f(a)$, $f(m)$, and $f(b)$. Specifically, we want constants A_1, A_2, and A_3 such that

$$\int_a^b f(x)dx = A_1 f(a) + A_2 f(m) + A_3 f(b)$$

is 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 3.41. Draw a picture showing what the previous two paragraphs discussed.

Exercise 3.42. Follow these steps to find A_1, A_2, and A_3.

a. Prove that

$$\int_a^b 1dx = b - a = A_1 + A_2 + A_3.$$

b. Prove that

$$\int_a^b x dx = \frac{b^2 - a^2}{2} = A_1 a + A_2 \left(\frac{a+b}{2}\right) + A_3 b.$$

c. Prove that

$$\int_a^b x^2 dx = \frac{b^3 - a^3}{3} = A_1 a^2 + A_2 \left(\frac{a+b}{2}\right)^2 + A_3 b^2.$$

d. Now solve the linear system of equations to prove that

$$A_1 = \frac{b-a}{6}, \quad A_2 = \frac{4(b-a)}{6}, \quad \text{and} \quad A_3 = \frac{b-a}{6}.$$

Exercise 3.43. At this point we can see that an integral can be approximated as

$$\int_a^b f(x)dx \approx \left(\frac{b-a}{6}\right)\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

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

$$\int_a^b f(x)dx \approx \left(\frac{b-a}{6}\right)\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right).$$

a. Use the method described above to approximate the area under the curve $f(x) = (1/5)x^2(5-x)$ on the interval $[1, 4]$. To be clear, you will be using the points $a = 1, m = 2.5$, and $b = 4$ in the above derivation.

b. Next find the exact area under the curve $g(x) = (-1/2)x^2 + 3.3x - 2$ on the interval $[1, 4]$.

c. What do you notice about the two areas? What does this sample problem tell you about the formula that we derived above?

To make the punchline of the previous exercises a bit more clear, using the formula

$$\int_a^b f(x)dx \approx \left(\frac{a-b}{6}\right)(f(a) + 4f(m) + f(b))$$

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 trapezoid rule and Riemann sums that we were after:

- Riemann sums approximate the function with constant functions,

- the trapezoid 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.

Definition 3.4 (Simpson's Rule)**.** Now we put the process explained above into a form that can be coded to approximate integrals. 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.

a. First partition $[a, b]$ into the set $\{x_0 = a, x_1, x_2, \ldots, x_{n-1}, x_n = b\}$.

b. On each part of the partition approximate the area with a parabola:

$$A_j = \frac{1}{6}\left[f(x_j) + 4f\left(\frac{x_j + x_{j-1}}{2}\right) + f(x_{j-1})\right](x_j - x_{j-1})$$

c. Approximate the integral as

$$\int_a^b f(x)dx = \sum_{j=1}^{n} A_j$$

Exercise 3.44. We have spent a lot of time over the past many pages building approximations of the order of the error for numerical integration and differentiation schemes. It is now up to you.
Build a numerical experiment that allows you to conjecture the order of the approximation error for Simpson's rule. Remember that the goal is to answer the question:
If I approximate the integral with a fixed Δx and find an absolute percent error of P, then what will the absolute percent error be using a width of $\Delta x/M$?

Exercise 3.45. Write a Python function that implements Simpson's Rule. You should ALWAYS start by writing pseudo-code as comments in your file. You shouldn't need a loop in your function.

Exercise 3.46. Test your function on known integrals and approximate the order of the error based on the mesh size.

Thus far we have three numerical approximations for definite integrals: Riemann sums (with rectangles), the trapezoidal rule, and Simpsons's rule. There are MANY other approximations for integrals and we leave the further research to the curious reader.

Theorem 3.3 (Numerical Integration Schemes)**.** *Let $f(x)$ be a continuous function on the interval $[a, b]$. The integral $\int_a^b f(x)dx$ can be approximated with any of the following.*

Riemann Sum: $$\int_a^b f(x)dx \approx \sum_{j=1}^{N} f(x_j)\Delta x$$

Error for Left and Right Riemann Sums: $\mathcal{O}(\Delta x)$

Riemann Sum: $$\int_a^b f(x)dx \approx \sum_{m=1}^{N} f(x_m)\Delta x$$

Error for Midpoint Riemann Sums: $\mathcal{O}(\Delta x^2)$

Trapezoidal Rule: $$\int_a^b f(x)dx \approx \frac{1}{2}\sum_{j=1}^{N} \left(f(x_j) + f(x_{j-1})\right)\Delta x$$

Error for Trapezoidal Rule: $\mathcal{O}(\Delta x^2)$

Simpson's Rule: $$\int_a^b f(x)dx \approx \frac{1}{6}\sum_{j=1}^{N} \left(f(x_j) + 4f\left(\frac{x_j + x_{j-1}}{2}\right) + f(x_{j-1})\right)\Delta x$$

Error for Simpson's Rule: $\mathcal{O}(\Delta x^4)$

where $\Delta x = x_j - x_{j-1}$ and N is the number of subintervals.

Exercise 3.47. Theorem 3.3 simply states the error rates for our three primary integration schemes. For this problem you need to empirically verify these error rates. Use the integration problem and exact answer

$$\int_0^{\pi/4} e^{3x}\sin(2x)dx = \frac{3}{13}e^{3\pi/4} + \frac{2}{13}$$

and write code that produces a log-log error plot with Δx on the horizontal axis and the absolute error on the vertical axis. Fully explain how the error rates show themselves in your plot.

3.4 Optimization

3.4.1 Single Variable Optimization

You likely recall that one of the major applications of Calculus was to solve optimization problems – find the value of x which makes some function as big or as small as possible. The process itself can sometimes be rather challenging

due to either the modeling aspect of the problems and/or the fact that the differentiation might be quite cumbersome. In this section we will revisit those problems from Calculus, but our goal will be to build a numerical method for the Calculus step in hopes to avoid the messy algebra and differentiation.

Exercise 3.48. 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 3.8). We want to find the size of the cut, x, that maximizes the volume of the box.

a. Write a function for the volume of the box resulting from a cut of size x. What is the domain of your function?

b. We know that we want to maximize this function so go through the full Calculus exercise to find the maximum:

- take the derivative
- set it to zero
- find the critical points
- test the critical points and the boundaries of the domain using the extreme value theorem to determine the x that gives the maximum.

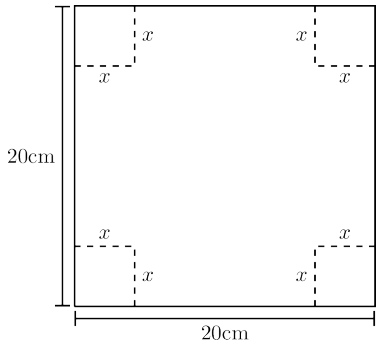


Figure 3.8: Folds to make a cardboard box

The hard part of the single variable optimization process is often solving the equation $f'(x) = 0$. We could use numerical root finding schemes to solve this equation, but we could also potentially do better without actually finding the derivative. In the following we propose a few numerical techniques that can approximate the solution to these types of problems. The basic ideas are simple!

Exercise 3.49. If you were blind folded and standing on a hill could you find the top of the hill? (assume no trees and no cliffs . . . this isn't supposed to be dangerous) How would you do it? Explain your technique clearly.

Exercise 3.50. 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.

The intuition of numerical optimization schemes is typically to visualize the function that you're trying to minimize or maximize and think about either climbing the hill to the top (maximization) or descending the hill to the bottom (minimization).

Exercise 3.51. Let's turn your intuitions into algorithms. If $f(x)$ is the function that you are trying to maximize then turn your ideas from the previous problems into step-by-step algorithms which could be coded. Then try out your codes on the function

$$f(x) = e^{-x^2} + \sin(x^2)$$

to see if your algorithms can find the local maximum near $x \approx 1.14$. Try to generate several different algorithms.

Some of the most common algorithms are listed below. Read through them and see which one(s) you ended up recreating? The intuition for these algorithms is pretty darn simple – travel uphill if you want to maximize – travel downhill if you want to minimize.

Definition 3.5 (Derivative Free Optimization)**.** Let $f(x)$ be the objective function which you are seeking to maximize (or minimize).

- Pick a starting point, x_0, and find the value of your objective function at this point, $f(x_0)$.
- Pick a small step size (say, $\Delta x \approx 0.01$).
- Calculate the objective function one step to the left and one step to the right from your starting point. Which ever point is larger (if you're seeking a maximum) is the point that you keep for your next step.
- Iterate (decide on a good stopping rule)

Exercise 3.52. Write code to implement the 1D derivative free optimization algorithm and use it to solve Exercise 3.48. Compare your answer to the analytic solution.

Definition 3.6 (Gradient Descent/Ascent)**.** Let $f(x)$ be the objective function which you are seeking to maximize (or 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 $x_{n+1} = x_n + \alpha f'(x_n)$ if you're maximizing. Use the iteration $x_{n+1} = x_n - \alpha f'(x_n)$ if you're minimizing.
- Iterate (decide on a good stopping rule)

Exercise 3.53. Write code to implement the 1D gradient descent algorithm and use it to solve Exercise 3.48. Compare your answer to the analytic solution.

Definition 3.7 (Monte-Carlo Search)**.** Let $f(x)$ be the objective function which you are seeking to maximize (or minimize).

- Pick many (perhaps several thousand!) different x values.
- Find the value of the objective function at every one of these points (Hint: use lists, not loops)
- Keep the x value that has the largest (or smallest if you're minimizing) value of the objective function.
- Iterate many times and compare the function value in each iteration to the previous best function value

---

Exercise 3.54. Write code to implement the 1D monte carlo search algorithm and use it to solve Exercise 3.48. Compare your answer to the analytic solution.

Definition 3.8 (Optimization via Numerical Root Finding)**.** Let $f(x)$ be the objective function which you are seeking to maximize (or minimize).

- Find the derivative of your objective function.
- Set the derivative to zero and use a numerical root finding method (such as bisection or Newton) to find the critical point.
- Use the extreme value theorem to determine if the critical point or one of the endpoints is the maximum (or minimum).

Exercise 3.55. Write code to implement the 1D numerical root finding optimization algorithm and use it to solve Exercise 3.48. Compare your answer to the analytic solution.

Exercise 3.56. In this problem we will compare an contrast the four methods proposed in the previous problem.

a. What are the advantages to each of the methods proposed?

b. What are the disadvantages to each of the methods proposed?

c. Which method, do you suppose, will be faster in general? Why?

d. Which method, do you suppose, will be slower in general? Why?

Exercise 3.57. The Gradient Ascent/Descent algorithm is the most geometrically interesting of the four that we have proposed. The others are pretty brute force algorithms. What is the Gradient Ascent/Descent algorithm doing geometrically? Draw a picture and be prepared to explain to your peers.

Exercise 3.58. (This problem is modified from [6])
A pig weighs 200 pounds and gains weight at a rate proportional to its current weight. Today the growth rate if 5 pounds per day. The pig costs 45 cents per day to keep due mostly to the price of food. The market price for pigs if 65 cents per pound but is falling at a rate of 1 cent per day. 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 and solve the problem analytically (by hand). Then solve the problem with all four methods outlined thus far in this section.

Exercise 3.59. (This problem is modified from [6])
Reconsider the pig problem 3.58 but now suppose that the weight of the pig after t days is

$$w = \frac{800}{1 + 3e^{-t/30}} \text{ pounds.}$$

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 3.60. Numerical optimization is often seen as quite challenging since the algorithms that we have introduced here could all get "stuck" at local extrema. To illustrate this see the function shown in Figure 3.9. How will derivative free optimization methods have trouble finding the red point starting at the black point with this function? How will gradient descent/ascent methods have trouble? Why?

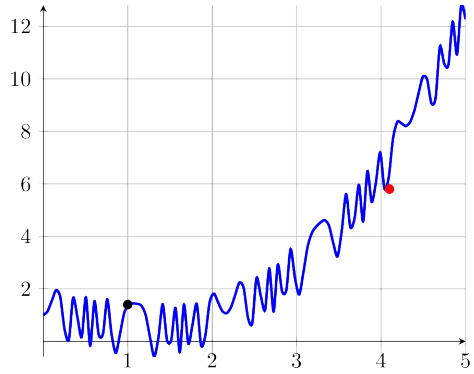


Figure 3.9: A challenging numerical optimization problem. If we start at the black point then how will any of our algorithms find the local minimum at the red point?

3.4.2 Multivariable Optimization

Now let's look at multivariable optimization. The analytic process for finding optimal solutions is essentially the same as for single variable.

- Write a function that models a scenario in multiple variables,
- find the gradient vector (presuming that the function is differentiable),
- set the gradient vector equal to the zero vector and solve for the critical point(s), and
- interpret your answer in the context of the problem.

The trouble with unconstrained multivariable optimization is that finding the critical points is now equivalent to solving a system of nonlinear equations; a task that is likely impossible even with a computer algebra system.

Let's see if you can extend your intuition from single variable to multivariable. This particular subsection is intentionally quite brief. If you want more details on multivariable optimization it would be wise to take a full course in optimization.

Exercise 3.61. The derivative free optimization method discussed in the single variable optimization section just said that you should pick two points and pick the one that takes you furthest uphill.

a. Why is it insufficient to choose just two points if we are dealing with a function of two variables? Hint: think about contour line.

b. For a function of two variables, how many points should you use to compare and determine the direction of "uphill"?

c. Extend your answer from part (b) to n dimensions. How many points should we compare if we are in n dimensions and need to determine which direction is "uphill"?

d. Back in the case of a two-variable function, you should have decided that three points was best. Explain an algorithm for moving one point at a time so that your three points eventually converge to a nearby local maximum. It may be helpful to make a surface plot or a contour plot of a well-known function just as a visual.

The clode below will demonstrate how to make a contour plot.

```
import numpy as np
import matplotlib.pyplot as plt
xdomain = np.linspace(-4,4,100)
ydomain = np.linspace(-4,4,100)
X, Y = np.meshgrid(xdomain,ydomain)
f = lambda x, y: np.sin(x)*np.cos(x)
plt.contour(X,Y,f(X,Y))
plt.grid()
plt.show()
```

Exercise 3.62. Now let's tackle the gradient ascent/descent algorithm. You should recall that the gradient vector points in the direction of maximum change. How can you use this fact to modify the gradient ascent/descent algorithm given previously? Clearly write your algorithm so that a classmate could turn it into code.

Exercise 3.63. How does the Monte Carlo algorithm extend to a two-variable optimization problem? Clearly write your algorithm.

Exercise 3.64. Try out the gradient descent/ascent and Monte Carlo algorithms on the function $f(x, y) = \sin(x)\cos(y) + 0.1x^2$ which has many local extrema and no global maximum. We are not going to code the multidimensional derivative free optimization routine in this section.

The derivative free, gradient ascent/descent, and monte carlo techniques still have good analogues in higher dimensions. We just need to be a bit careful since in higher dimensions there is much more room to move. Below we'll give the full description of the gradient ascent/descent algorithm. We don't give the full description of the derivative free or Monte Carlo algorithms since there are many ways to implement them. The interested reader should see a course in mathematical optimization or machine learning.

Definition 3.9 (The Gradient Descent Algorithm)**.** We want to solve the problem

$$\text{minimize } f(x_1, x_2, \ldots, x_n) \text{ subject to } (x_1, x_2, \ldots, x_n) \in S.$$

a. Choose an arbitrary starting point $\boldsymbol{x}_0 = (x_1, x_2, \ldots, x_n) \in S$.

b. We are going to define a difference equation that gives successive guesses for the optimal value:

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha \nabla f(\boldsymbol{x}_n).$$

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?).

c. Repeat the iterative process in step b until two successive points are *close enough* to each other.

Take Note: If you are looking to maximize your objective function then in the Monte-Carlo search you should examine if z is greater than your current largest value. For gradient descent you should actually do a gradient *ascent* instead and follow the positive gradient instead of the negative gradient.

Exercise 3.65. The functions like $f(x, y) = \sin(x)\cos(y)$ have many local extreme values which makes optimization challenging. Implement your Gradient Descent code on this function to find the local minimum $(0, 0)$. Start somewhere near $(0, 0)$ and show by way of example that your gradient descent code may not converge to this particular local minimum. Why is this important?

3.5 Calculus with `numpy` and `scipy`

In this section we will look at some highly versatile functions built into the `numpy` and `scipy` libraries in Python. These libraries allow us to lean on pre-built numerical routines for calculus and optimization and instead we can focus our energies on setting up the problems and interpreting solutions. The down side here is that we are going to treat some of the optimization routines in Python as black boxes, so part of the goal of this section is to partially unpack these black boxes so that we know what's going on under the hood. If you haven't done Exercise 2.65 yet you may want to do so now in order to get used to some of the syntax used by the Python `scipy` library.

3.5.1 Differentiation

There are two main tools built into the `numpy` and `scipy` libraries that do numerical differentiation. In `numpy` there is the `np.diff()` command. In `scipy` there is the `scipy.misc.derivative()` command.

Exercise 3.66. In the following blocks of Python code we demonstrate what the `np.diff()` command does. Use these examples to give a thorough description for what `np.diff()` does to a Python list.

First example of `np.diff()`:

```
import numpy as np
myList = np.arange(0,10)
print(myList)
```

```
## [0 1 2 3 4 5 6 7 8 9]
```

```
print( np.diff(myList) )
```

```
## [1 1 1 1 1 1 1 1 1]
```

Second example of `np.diff()`:

```
import numpy as np
myList = np.linspace(0,1,6)
print(myList)
```

```
## [0.  0.2 0.4 0.6 0.8 1. ]
```

```
print( np.diff(myList) )
```

```
## [0.2 0.2 0.2 0.2 0.2]
```

Third example of `np.diff()`:

```
import numpy as np
x = np.linspace(0,1,6)
dx = x[1]-x[0]
y = x**2
dy = 2*x
print("function values: \n",y)
```

```
## function values:
##  [0.   0.04 0.16 0.36 0.64 1.  ]
```

```
print("exact values of derivative: \n",dy)
```

```
## exact values of derivative:
##  [0.  0.4 0.8 1.2 1.6 2. ]
```

```
print("values from np.diff(): \n",np.diff(y))
```

```
## values from np.diff():
##  [0.04 0.12 0.2  0.28 0.36]
```

```
print("values from np.diff()/dx: \n",np.diff(y) / dx )
```

```
## values from np.diff()/dx:
##  [0.2 0.6 1.  1.4 1.8]
```

Exercise 3.67. Why does the `np.diff()` command produce a list that is one element shorter than the original list?

Exercise 3.68. If we have a list of x values and a list of y values for a function $y = f(x)$ then how do we use `np.diff()` to approximate the first derivative of $f(x)$? What is the order of the error in the approximation?

Exercise 3.69. What does the following block of Python code do?

```
import numpy as np
x = np.linspace(0,1,6)
dx = x[1]-x[0]
y = x**2
print( np.diff(y,2) / dx**2 )
```

```
## [2. 2. 2. 2.]
```

Exercise 3.70. Use the `np.diff()` command to approximate the first and second derivatives of the function $f(x) = x\sin(x) - \ln(x)$ on the domain $[1, 5]$. Then create a plot that shows $f(x)$ and the approximations of $f'(x)$ and $f''(x)$.

Exercise 3.71. Next we look into the `scipy.misc.derivative()` command from the `scipy` library. This will be another way to calculate the derivative of a function. One advantage will be that you can just send in a Python function (or a lambda function) without actually computing the lists of values. Examine the following Python code and fully describe what it does.

```
import numpy as np
import scipy.misc
f = lambda x: x**2
x = np.linspace(1,5,5)
df = scipy.misc.derivative(f,x,dx = 1e-10)
print(df)
```

```
## [ 2.00000017  4.00000033  6.0000005   8.00000066 10.00000083]
```

One advantage to using `scipy.misc.derivative()` is that you get to dictate the error in the derivative computation, and that error is not tied to the list of values that you provide. In its simplest form you can provide just a single x value just like in the next block of code.

```
import numpy as np
import scipy.misc
f = lambda x: x**2
df = scipy.misc.derivative(f,1,dx = 1e-10) # derivative at x=1
print(df)
```

```
## 2.000000165480742
```

Exercise 3.72. In the following code we find the first and second derivatives of $f(x) = x\sin(x) - \ln(x)$ using `scipy.misc.derivative()`. Notice that we've chosen to take `dx=1e-6` for each of the derivative computations. That may seem like an odd choice, but there is more going on here. Try successively smaller and smaller values for the `dx` parameter. What do you find? Why does it happen?

```
import numpy as np
import scipy.misc
import matplotlib.pyplot as plt
f = lambda x: np.sin(x)*x-np.log(x)
x = np.linspace(1,5,100) # x domain: 100 points between 1 and 5
df = scipy.misc.derivative(f,x,dx=1e-6)
df2 = scipy.misc.derivative(f,x,dx=1e-6,n=2)
plt.plot(x,f(x),'b',x,df,'r--',x,df2,'k--')
plt.legend(["f(x)","f'(x)","f''(x)"])
plt.grid()
plt.show()
```

3.5.2 Integration

In `numpy` there is a nice tool called `np.trapz()` that implements the trapezoidal rule. In the following problem you will find several examples of the `np.trapz()` command. Use these examples to determine how the command works to integrate functions.

Exercise 3.73. First we'll approximate the integral $\int_{-2}^{2} x^2 dx$. The exact answer is

$$\int_{-2}^{2} x^2 dx = \frac{x^3}{3}\Big|_{-2}^{2} = \frac{16}{3} = 5.3333...$$

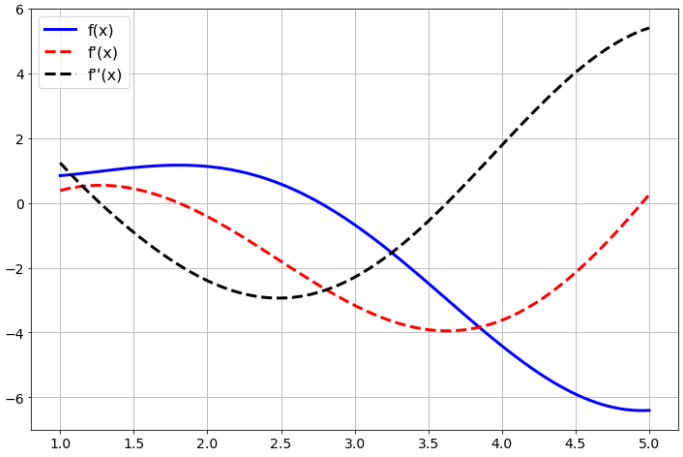


Figure 3.10: Derivatives with scipy

```
import numpy as np
x = np.linspace(-2,2,100)
dx = x[1]-x[0]
y = x**2
print("Approximate integral is ",np.trapz(y)*dx)
```

```
## Approximate integral is  5.334421657653986
```

Next we'll approximate $\int_0^{2\pi} \sin(x)dx$. We know that the exact value is 0.

```
import numpy as np
x = np.linspace(0,2*np.pi,100)
dx = x[1]-x[0]
y = np.sin(x)
print("Approximate integral is ",np.trapz(y)*dx)
```

```
## Approximate integral is  -2.2019371830862577e-17
```

Pick a function and an interval for which you know the exact definite integral. Demonstrate how to use `np.trapz()` on your definite integral.

Exercise 3.74. Notice in the last examples that we multiplied the result of the `np.trapz()` command by `dx`. Why did we do this? What is the `np.trapz()` command doing without the `dx`?

In the `scipy` library there is a more general tool called `scipy.integrate.quad()`. The term "quad" is short for "quadrature". In numerical analysis literature rules

like Simpson's rule are called quadrature rules for integration. The function `scipy.integrate.quad()` accepts a Python function (or a lambda function) and the bounds of the definite integral. It outputs an approximation of the integral along with an approximation of the error in the integral calculation. See the Python code below.

```
import numpy as np
import scipy.integrate
f = lambda x: x**2
I = scipy.integrate.quad(f,-2,2)
print(I)
```

```
## (5.333333333333333, 5.921189464667501e-14)
```

Exercise 3.75. What are the advantages and disadvantages to using the `scipy.integrate.quad()` command as compared to the `np.trapz()` command.

Exercise 3.76. If you have data for the hourly rate at which water is being drained from a dam and you want to find the total amount of water drained over the course of the time in the dataset, then which of the tools that we know would you use? Why?

3.5.3 Optimization

As you've seen in this section there are many tools built into `numpy` and `scipy` 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 that what we'll see here. However, the Python tools that we will use are highly optimized and tend to work quite well.

Exercise 3.77. Let's 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)
```

```
##       fun: -5.0
##  hess_inv: array([[0.49999999]])
##       jac: array([0.])
##   message: 'Optimization terminated successfully.'
##      nfev: 9
##       nit: 2
##      njev: 3
##    status: 0
##   success: True
##         x: array([2.99999998])
```

a. Implement the code above then spend some time playing around with the minimize command to minimize more challenging functions.
b. Explain what all of the output information is from the `.minimize()` command.

Exercise 3.78. There is not a function called `scipy.optimize.maximize()`. Instead, Python expects you to rewrite every maximization problem as a minimization problem. How do you do that?

Exercise 3.79. Solve Exercise 3.48 using `scipy.optimize.minimize()`.

3.6 Least Squares Curve Fitting

In this section we'll change our focus a bit to look at a different question from algebra, and, in turn, reveal a hidden numerical optimization problem where the `scipy.optimize.minimize()` tool will come in quite handy.

Here is the primary question of interest:
If we have a few data points and a reasonable guess for the type of function fitting the points, how would we determine the actual function?

You may recognize this as the basic question of regression from statistics. What we will do here is pose the statistical question of which curve best fits a data set as an optimization problem. Then we will use the tools that we've built so far to solve the optimization problem.

Exercise 3.80. Consider the function $f(x)$ that goes exactly through the points $(0, 1)$, $(1, 4)$, and $(2, 13)$.

a. Find a function that goes through these points exactly. Be able to defend your work.

b. Is your function unique? That is to say, is there another function out there that also goes exactly through these points?

Exercise 3.81. Now let's make a minor tweak to the previous problem. Let's say that we have the data points $(0, 1.07)$, $(1, 3.9)$, $(2, 14.8)$, and $(3, 26.8)$. Notice that these points are *close* to the points we had in the previous problem, but all of the y values have a little noise in them and we have added a fourth point. If we suspect that a function $f(x)$ that *best* fits this data is quadratic then $f(x) = ax^2 + bx + c$ for some constants a, b, and c.

a. Plot the four points along with the function $f(x)$ for arbitrarily chosen values of a, b, and c.

b. Work with your partner(s) to systematically change a, b, and c so that you get a good visual match to the data. The Python code below will help you get started.

```
import numpy as np
import matplotlib.pyplot as plt
xdata = np.array([0, 1, 2, 3])
ydata = np.array([1.07, 3.9, 14.8, 26.8])
a = # conjecture a value of a
b = # conjecture a value of b
c = # conjecture a value of c
x = # build an x domain starting at 0 and going through 4
guess = a*x**2 + b*x + c
# make a plot of the data
# make a plot of your function on top of the data
```

As an alternative to loading the data manually we could download the data from the book's github page. All datasets in the text can be loaded in this way. We will be using the `pandas` library (a Python data science library) to load the `.csv` files.

```
import numpy as np
import pandas as pd
data = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSullivan/Numerica
xdata = data[:,0]
ydata = data[:,1]
```

Exercise 3.82. Now let's be a bit more systematic about things from the previous problem. Let's say that you have a pretty good guess that $b \approx 2$ and $c \approx 0.7$. We need to get a good estimate for a.

a. Pick an arbitrary starting value for a then for each of the four points find

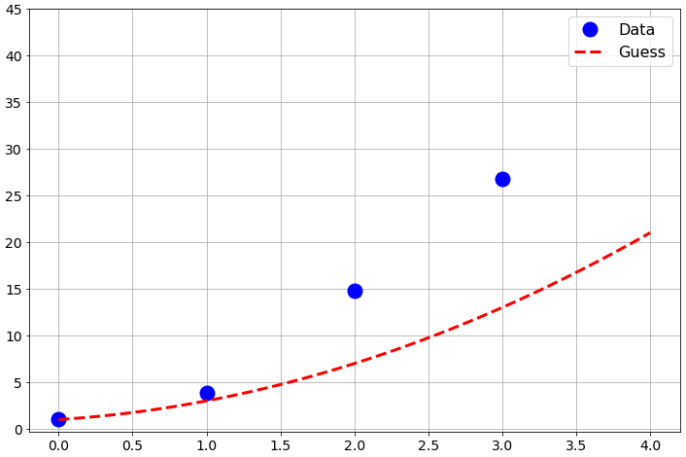


Figure 3.11: Initial attempt at matching data with a quadratic.

the error between the predicted y value and the actual y value. These errors are called the *residuals*.

b. Square all four of your errors and add them up. (Pause, ponder, and discuss: why are we squaring the errors before we sum them?)

c. Now change your value of a to several different values and record the sum of the square errors for each of your values of a. It may be worth while to use a spreadsheet to keep track of your work here.

d. Make a plot with the value of a on the horizontal axis and the value of the sum of the square errors on the vertical axis. Use your plot to defend the optimal choice for a.

Exercise 3.83. We're going to revisit part (c) of the previous problem. Write a loop that tries many values of a in very small increments and calculates the sum of the squared errors. The following partial Python code should help you get started. In the resulting plot you should see a clear local minimum. What does that minimum tell you about solving this problem?

```
import numpy as np
import matplotlib.pyplot as plt
xdata = np.array([0, 1, 2, 3])
ydata = np.array([1.07, 3.9, 14.8, 26.8])
b = 2
c = 0.75
A = # give a numpy array of values for a
SumSqRes = [] # this is blank storage for the sum of the squared residuals
for a in A:
  guess = a*xdata**2 + b*xdata + c
```

```
  residuals = # write code to calculate the residuals
  SumSqRes.append( ??? ) # calculate the sum of the squared residuals
plt.plot(A,SumSqRes,'r*')
plt.grid()
plt.xlabel('Value of a')
plt.ylabel('Sum of squared residuals')
plt.show()
```

Now let's formalize the process that we've described in the previous problems.

Definition 3.10 (Least Squares Regression)**.** Let

$$S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

be a set of n ordered pairs in $\mathbb{R}^2$. If we guess that a function $f(x)$ is a best choice to fit the data and if $f(x)$ depends on parameters $a_1, a_2, \ldots, a_n$ then

a. Pick initial values for the parameters $a_1, a_2, \ldots, a_n$ so that the function $f(x)$ *looks like* it is close to the data (this is strictly a visual step ... take care that it may take some playing around to guess the initial values of the parameters)

b. Calculate the square error between the data point and the prediction from the function $f(x)$

$$\text{error for the point } x_i\text{: } e_i = (y_i - f(x_i))^2 .$$

Note that squaring the error has the advantages of removing the sign, accentuating errors larger than 1, and decreasing errors that are less than 1.

c. As a measure of the total error between the function and the data, sum the squared errors

$$\text{sum of square errors } = \sum_{i=1}^{n} (y_i - f(x_i))^2 .$$

(Take note that if there were a continuum of points instead of a discrete set then we would integrate the square errors instead of taking a sum.)

d. Change the parameters $a_1, a_2, \ldots$ so as to minimize the sum of the square errors.

Exercise 3.84. In 3.10 the last step is a bit vague. That was purposeful since there are many techniques that could be used to minimize the sum of the square errors. However, if we just think about the sum of the squared residuals as a function then we can apply `scipy.optimize.minimize()` to that function in

order to return the values of the parameters that best minimize the sum of the squared residuals. The following blocks of Python code implement the idea in a very streamlined way. Go through the code and comment each line to describe exactly what it does.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
xdata = np.array([0, 1, 2, 3])
ydata = np.array([1.07, 3.9, 14.8, 26.8])

def SSRes(parameters):
  # In the next line of code we want to build our quadratic approximation.
  # y = ax^2 + bx + c
  # We are sending in a list of parameters so
  # a = parameters[0], b = parameters[1], and c = parameters[2]
  yapprox = parameters[0]*xdata**2 + parameters[1]*xdata + parameters[2]
  residuals = np.abs(ydata-yapprox)
  return np.sum(residuals**2)


BestParameters = minimize(SSRes,[2,2,0.75])
print("The best values of a, b, and c are: \n",BestParameters.x)
```

```
## The best values of a, b, and c are:
##  [2.29249989 1.93150033 0.72149989]
```

```
print("The minimization diagnostics are: \n",BestParameters)
```

```
## The minimization diagnostics are:
##        fun: 2.4290450000000545
##  hess_inv: array([[ 0.12500001, -0.37500002,  0.12499999],
##         [-0.37500002,  1.22500001, -0.52499991],
##         [ 0.12499999, -0.52499991,  0.47499989]])
##       jac: array([-2.98023224e-08, -2.98023224e-08, -2.98023224e-08])
##   message: 'Optimization terminated successfully.'
##      nfev: 35
##       nit: 4
##      njev: 7
##    status: 0
##   success: True
##         x: array([2.29249989, 1.93150033, 0.72149989])
```

```
plt.plot(xdata,ydata,'bo',markersize=5)
x = np.linspace(0,4,100)
y = BestParameters.x[0]*x**2 + BestParameters.x[1]*x + BestParameters.x[2]
plt.plot(x,y,'r--')
```

```
plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Best Fit Quadratic')
plt.show()
```

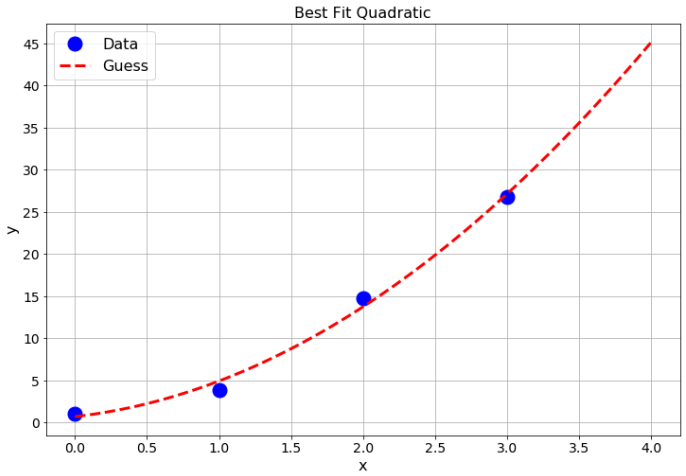


Figure 3.12: Best fit quadratic function.

Exercise 3.85. With a partner choose a function and then choose 10 points on that function. Add a small bit of error into the y-values of your points. Give your 10 points to another group. Upon receiving your new points:

- Plot your points.
- Make a guess about the basic form of the function that might best fit the data. Your general form will likely have several parameters (just like the quadratic had the parameters a, b, and c).

- Modify the code from above to find the best collection of parameters minimize the sum of the squares of the residuals between your function and the data.

- Plot the data along with your best fit function. If you are not satisfied with how it fit then make another guess on the type of function and repeat the process.
- Finally, go back to the group who gave you your points and check your work.

Exercise 3.86. For each dataset associated with this exercise give a functional

form that might be a good model for the data. Be sure to choose the most general form of your guess. For example, if you choose "quadratic" then your functional guess is $f(x) = ax^2 + bx + c$, if you choose "quadratic" then your functional guess should be something like $f(x) = ae^{b(x-c)} + d$, or if you choose "sinusoidal" then your guess should be something like $f(x) = a\sin(bx) + c\cos(dx) + e$. Once you have a guess of the function type create a plot showing your data along with your guess for a reasonable set of parameters. Then write a function that leverages `scipy.optimize.minimize()` to find the best set of parameters so that your function best fits the data. Note that if `scipy.optimize.minimize()` does not converge then try the alternative `scipy` function `scipy.optimize.fmin()`. Also note that you likely need to be very close to the optimal parameters to get the optimizer to work properly.

You can load the data with the following script.

```
import numpy as np
import pandas as pd
datasetA = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSu
datasetB = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSu
datasetC = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSu
```

As a nudge in the right direction, in the left-hand pane of Figure 3.13 the function appears to be exponential. Hence we should choose a function of the form $f(x) = ae^{b(x-c)} + d$. Moreover, we need to pick good approximations of the parameters to start the optimization process. In the left-hand pane of Figure 3.13 the data appears to start near $x = 1970$ so our initial guess for c might be $c \approx 1970$. To get initial guesses for a, b, and d we can observe that the expected best fit curve will approximately go through the points $(1970, 15000)$, $(1990, 40000)$, and $(2000, 75000)$. With this information we get the equations $a+d \approx 15000$, $ae^{20b}+d \approx 40000$ and $ae^{30b}+d \approx 75000$ and work to get reasonable approximations for a, b, and d to feed into the `scipy.optimize.minimize()` command.

3.7 Exercises

3.7.1 Algorithm Summaries

Exercise 3.87. Starting from Taylor series prove that

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

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

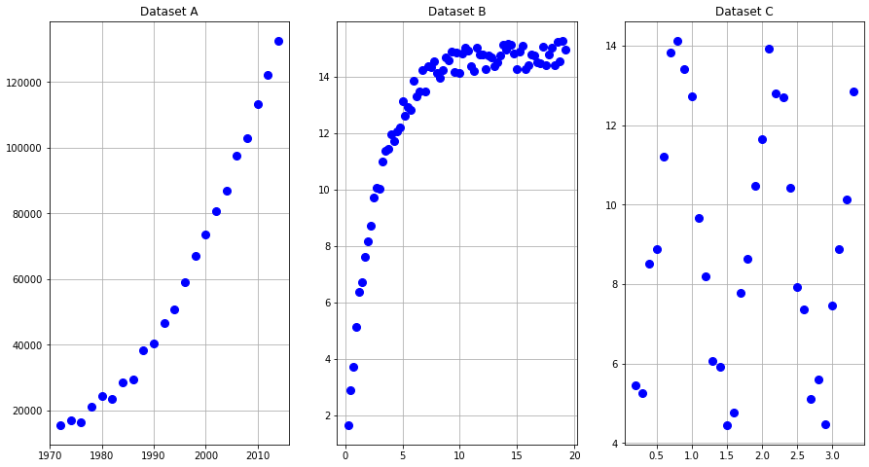


Figure 3.13: Raw data for least squares function matching problems.

Exercise 3.88. Starting from Taylor series prove that

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

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

Exercise 3.89. Starting from Taylor series prove that

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

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

Exercise 3.90. Explain how to approximate the value of a definite integral with Riemann sums. When will the Riemann sum approximation be exact? The Riemann sum approximation is first order. Expain what "first order" means for calculating a definite integral.

Exercise 3.91. Explain how to approximate the value of a definite integral with the Trapezoid rule. When will the Trapezoid rule approximation be exact? The Trapezoidal rule approximation is second order. Expain what "second order" means for calculating a definite integral.

Exercise 3.92. 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? The Simpson's rule approximation is fourth order. Expain what "fourth order" means for calculating a definite integral.

Exercise 3.93. Explain in clear language how the derivative free optimization method works on a single-variable function.

Exercise 3.94. Explain in clear language how the gradient descent/ascent optimization method works on a single-variable function.

Exercise 3.95. Explain in clear language how the Monte Carlo search optimization method works on a single-variable function.

Exercise 3.96. Explain in clear language how you find the optimal set of parameters given a set of data and a proposed general function type.

3.7.2 Applying What You've Learned

Exercise 3.97. 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.

a. $f'(x) \approx \frac{\frac{1}{12}f(x-2h)-\frac{2}{3}f(x-h)+\frac{2}{3}f(x+h)-\frac{1}{12}f(x+2h)}{h}$

b. $f'(x) \approx \frac{-\frac{3}{2}f(x)+2f(x+h)-\frac{1}{2}f(x+2h)}{h}$

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

d. $f'''(x) \approx \frac{-\frac{1}{2}f(x-2h)+f(x-h)-f(x+h)+\frac{1}{2}f(x+2h)}{h^3}$

Exercise 3.98. 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 3.99. 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 3.100. 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 3.101. Use numerical integration to answer the question in each of the following scenarios

a. 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 textbook's github page with the code below.

```
import numpy as np
import pandas as pd
data = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSullivan/Numerica
```

b. The department of transportation finds that the rate at which cars cross a bridge can be approximated by the function

$$f(t) = \frac{22.8}{3.5 + 7(t - 1.25)^4},$$

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 3.102. Consider the integrals

$$\int_{-2}^{2} e^{-x^2/2} dx \quad \text{and} \quad \int_{0}^{1} \cos(x^2) dx.$$

Neither of these integrals have closed-form solutions so a numerical method is necessary. Create a loglog 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 loglog plot. To get the *exact* answer for these plots use Python's `scipy.integrate.quad` command. (What we're really doing here is comparing our algorithms to Python's `scipy.integrate.quad()` algorithm).

Exercise 3.103. Go to data.gov or the World Health Organization Data Repository and find data sets for the following tasks.

a. 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 explaning what the derivative means in the context of the data.

b. Find a data set where the variables naturally lead to a meaningfun 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 explaning what the integral means in the context of the data.

In both of these tasks be very cautious of the units on the data sets and the units of your answer.

Exercise 3.104. 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.

a. $f(x) = \frac{x}{1+x^4}$

b. $g(x) = (x-1)^3(x-2)^2$

c. $h(x) = \sin\left(x^2\right)$

Exercise 3.105. 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://raw.githubusercontent.com/NumericalMethodsSullivan/Numerica
```

Exercise 3.106. 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.

a. $f(x) = \dfrac{x}{1+x^4} + \sin(x)$

b. $g(x) = (x-1)^3 \cdot (x-2)^2 + e^{-0.5 \cdot x}$

Exercise 3.107. Go back to your old Calculus textbook or homework and find your favorite 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 3.108. In the code below you can download several sets of noisy data from measurements of elementary single variable functions.

a. Make a hypothesis about which type of function would best model the data. Be sure to choose the most general (parameterized) form of your function.

b. Use appropriate tools to find the parameters for the function that best fits the data. Report you sum of square residuals for each function.

The functions that you propose must be continuous functions.

```
import numpy as np
import pandas as pd
datasetA = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSullivan/Nume
datasetB = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSullivan/Nume
datasetC = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSullivan/Nume
datasetD = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSullivan/Nume
datasetE = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSullivan/Nume
datasetF = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSullivan/Nume
datasetG = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSullivan/Nume
datasetH = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSullivan/Nume
```

3.8 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. Take the time to read Appendix B before you write your final solution.

3.8.1 Galaxy Integration

To analyze the light from stars and galaxies, scientists use a spectral grating (fancy prism) to split it up into the different frequencies (colors). We can then measure the intensity (brightness) of the light (in units of Watts per square meter) at each frequency (measured in Hertz), to get intensity per frequency (Watts per square meter per Hertz, W/(m^2 Hz)). Light from the dense opaque surface of a star produces a smooth rainbow, which produces a continuous curve when we plot intensity versus frequency. However stars are also surrounded by thin gas which either emits or absorbs light at only a specific set of frequencies, called spectral lines. Every chemical element produces a specific set of lines (or peaks) at fixed frequencies, so by identifying the lines, we can tell what types of atoms and molecules a star is made of. If the gas is cool, then it will absorb light at these wavelengths, and if the gas is hot then it will emit light at these wavelengths. For galaxies, on the other hand, we expect mostly emission spectra: light emitted from the galaxy.

For this project we will be analyzing the galaxy "ngc 1275". The black hole at the center of this galaxy is often referred to as the "Galactic Spaghetti Monster" since the magnetic field "sustains a mammoth network of spaghetti-like gas filaments around it". You can download the data file associated with this project with the following Python code.

```
import numpy as np
import pandas as pd
ngc1275 = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSul
```

In the data you will see the spectral data measuring the light intensity from ncg 1275 at several different wavelengths (measured in Angstroms). You will notice in this data set that there are several emission lines at various wavelengths. Of particular interest are the peaks near 3800 Angstroms, 5100 Angstroms, 6400 Angstroms, and the two peaks around 6700 Angstroms. The data set contains 1,727 data points at different wavelengths. Your first job will be to transform the wavelength data to frequency via the formula

$$\lambda = \frac{c}{f}$$

where λ is the wavelength, c is the speed of light, and f is the frequency (measured

in Hertz). Be sure to double check the units. Given the inverse relationship between frequency and wavelength you should see the emission lines flip to the other side of the plot (right-to-left or left-to-right).

The strength of each emission line (in W/m^2) is defined as the relative intensity of each peak across the associated frequencies. Note that you are not interested in the intensity of the continuous spectrum – just the peaks. That is to say that you are only interested in the area above the background curve and the background noise.

Your primary task is to develop a process for analyzing data sets like this so as to determine the strength of each emission lines. You must demonstrate your process on this particular data set, but your process must be generalizable to any similar data set. Your process must clearly determine the strength of peaks in data sets like this and you must apply your procedure to determine the strength of each of these four lines with an associated margin of error. Keep in mind that you will first want to first develop a method for removing the background noise. Finally, the double peak near 6700 Angstroms needs to be handled with care: the strength of each emission line is only the integral over one peak, not two, so you'll need to determine a way to separate these peaks.

Finally, it would be cool, but is not necessary, to report on which chemicals correspond to the emission lines in the data. Remember that the galaxy is far away and hence there is a non-trivial red-shift to consider. This will take some research but if done properly will likely give a lot more merit to your paper.

3.8.2 Higher Order Integration

Riemann sums can be used to approximate integrals and they do so by using piece wise 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.

3.8.3 Dam Integration

Go to the USGS water data repository:
https://maps.waterdata.usgs.gov/mapper/index.html.

Here you'll 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 don't 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).

3.8.4 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 centered difference scheme for the first derivatives

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

In an image we can take $h = 1$ (why?), and since the gradient is two dimensional we get

$$\nabla G(x,y) \approx \left\langle \frac{G(x+1,y) - G(x-1,y)}{2}, \frac{G(x,y+1) - G(x,y-1)}{2} \right\rangle.$$

Figure 3.14 depicts what this looks like when we zoom in to a pixel and its immediate neighbors. The pixel labeled `G[i,j]` is the pixel at which we want to evaluate the gradient, and the surrounding pixels are labeled by their indices relative to `[i,j]`.

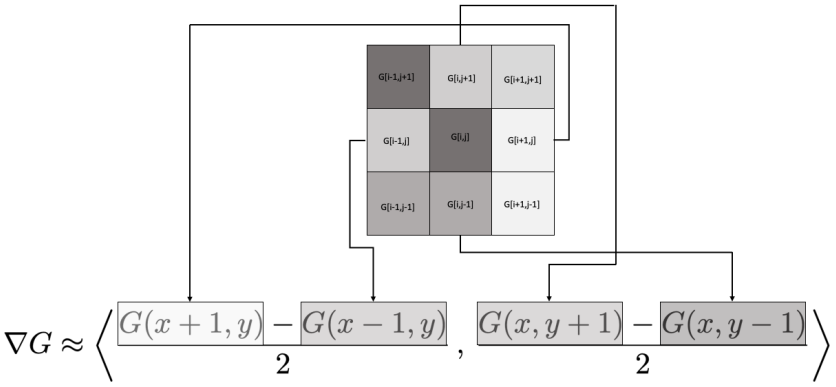


Figure 3.14: The gradient computation on a single pixel using a centered difference scheme for the first derivative.

3. There are many ways to approximate numerical first derivatives. The simplest approach is what you did in part (2) – using a centered difference scheme. However, pixels are necessarily tightly packed in an image and the immediate neighbors of a point may not have enough contrast to truly detect edges. If you examine Figure 3.14 you'll notice that we only use 4 of the 8 neighbors of the pixel `[i,j]`. Also notice that we didn't 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 centered difference scheme that looks two pixels away instead of at the immediate neighboring pixels

$$f'(x) \approx \frac{???f(x-2)+???f(x+2)}{???}.$$

Of course you would need to determine the coefficients in this approximation scheme.
Another idea could use a centered difference scheme that uses pixels that are immediate neighbors AND pixels that are two units away

$$f'(x) \approx \frac{???f(x-2)+???f(x-1)+???f(x+1)+???f(x+2)}{???}.$$

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.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

$$\text{grayscale value} = 0.3\text{Red} + 0.59\text{Green} + 0.11\text{Blue}.$$

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 3.15 shows the result of different threshold values applied to the simplest numerical gradient computations. The image was taken by the author.

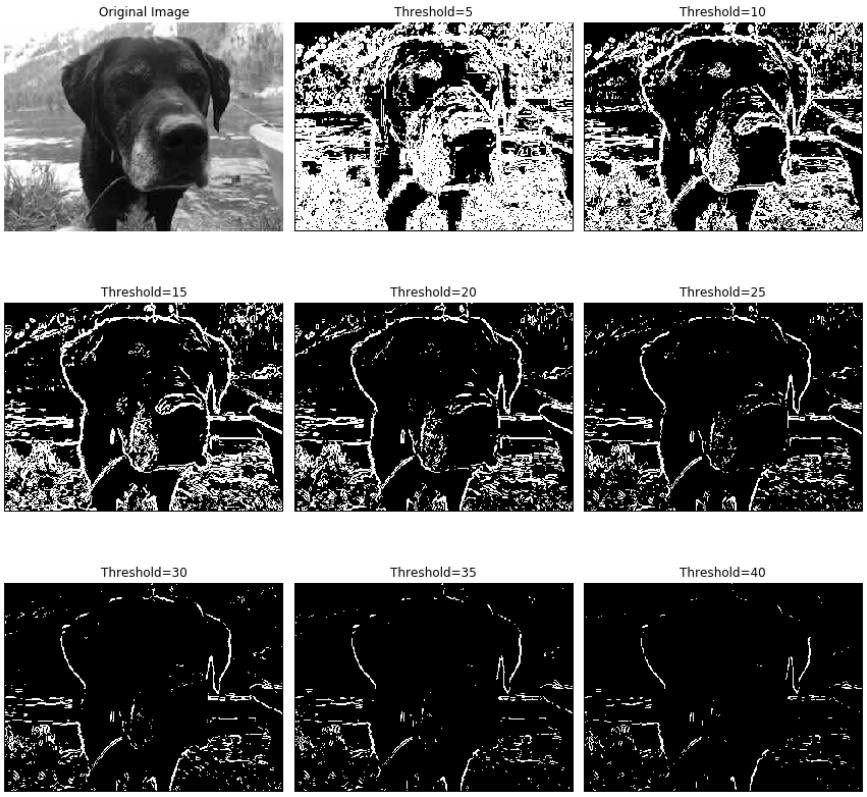


Figure 3.15: Edge detection using different thresholds for the value of the gradient on the grayscale image

Chapter 4

Linear Algebra

4.1 Intro to Numerical Linear Algebra

> *You cannot learn too much linear algebra.*
> – Every mathematician

The preceding comment says it all – linear algebra is the most important of all of the mathematical tools that you can learn as a practitioner of the mathematical sciences. The theorems, proofs, conjectures, and big ideas in almost every other mathematical field find their roots in linear algebra. Our goal in this chapter is to explore numerical algorithms for the primary questions of linear algebra:

- solving systems of equations,
- approximating solutions to over-determined systems of equations, and
- finding eigenvalue-eigenvector pairs for a matrix.

Take careful note, that in our current digital age numerical linear algebra and its fast algorithms are behind the scenes for wide varieties of computing applications. Applications of numerical linear algebra include:

- determining the most important web page in a Google search,
- determine the forces on a car during a crash,
- modeling realistic 3D environments in video games,
- digital image processing,
- building neural networks and AI algorithms,
- and many many more.

What's more, researchers have found provably optimal ways to perform most of the typical tasks of linear algebra so most scientific software works very well and very quickly with linear algebra. For example, we have already seen in Chapter 3 that programming numerical differentiation and numerical integration schemes can be done in Python with the use of vectors instead of loops. We want to use

vectors specifically so that we can use the fast implementations of numerical linear algebra in the background in Python.

Lastly, a comment on notation. Throughout this chapter we will use the following notation conventions.

- A bold mathematical symbol such as $\boldsymbol{x}$ or $\boldsymbol{u}$ will represent a vector.
- If $\boldsymbol{u}$ is a vector then u_j will be the j^{th} entry of the vector.
- Vectors will typically be written vertically with parenthesis as delimiters such as
$$\boldsymbol{u} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}.$$
- Two bold symbols separated by a centered dot such as $\boldsymbol{u} \cdot \boldsymbol{v}$ will represent the dot product of two vectors.
- A capital mathematical symbol such as A or X will represent a matrix
- If A is a matrix then A_{ij} will be the element in the i^{th} row and j^{th} column of the matrix.
- A matrix will typically be written with parenthesis as delimiters such as
$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & \pi \end{pmatrix}.$$
- The juxtaposition of a capital symbol and a bold symbol such as $A\boldsymbol{x}$ will represent matrix-vector multiplication.
- A lower case or Greek mathematical symbol such as x, c, or λ will represent a scalar.
- The scalar field of real numbers is given as $\mathbb{R}$ and the scalar field of complex numbers is given as $\mathbb{C}$.
- The symbol $\mathbb{R}^n$ represents the collection of n-dimensional vectors where the elements are drawn from the real numbers.
- The symbol $\mathbb{C}^n$ represents the collection of n-dimensional vectors where the elements are drawn from the complex numbers.

It is an important part of learning to read and write linear algebra to give special attention to the symbolic language so you can communicate your work easily and efficiently.

4.2 Vectors and Matrices in Python

We first need to understand how Python's `numpy` library builds and stores vectors and matrices. The following exercises will give you some experience building

and working with these data structures and will point out some common pitfalls that mathematicians fall into when using Python for linear algebra.

Example 4.1 ('numpy' Arrays)**.** In Python you can build a list using square brackets such as `[1,2,3]`. This is called a "Python list" and is NOT a vector in the way that we think about it mathematically. It is simply an ordered collection of objects. To build mathematical vectors in Python we need to use `numpy` arrays with `np.array()`. For example, the vector

$$\boldsymbol{u} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

would be built with the following code.

```
import numpy as np
u = np.array([1,2,3])
print(u)
```

```
## [1 2 3]
```

Notice that Python defines the vector `u` as a matrix without a second dimension. You can see that in the following code.

```
import numpy as np
u = np.array([1,2,3])
print("The length of the u vector is \n",len(u))
```

```
## The length of the u vector is
##  3
```

```
print("The shape of the u vector is \n",u.shape)
```

```
## The shape of the u vector is
##  (3,)
```

Example 4.2 ('numpy' Matrices)**.** In `numpy`, a matrix is a list of lists. For example, the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

is defined using `np.matrix()` where each row is an individual list, and the matrix is a collection of these lists.

```
import numpy as np
A = np.matrix([[1,2,3],[4,5,6],[7,8,9]])
print(A)
```

```
## [[1 2 3]
##  [4 5 6]
##  [7 8 9]]
```

Moreover, we can extract the shape, the number of rows, and the number of columns of A using the `A.shape` command. To be a bit more clear on this one we'll use the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

```
import numpy as np
A = np.matrix([[1,2,3],[4,5,6]])
print("The shape of the A matrix is \n",A.shape)
```

```
## The shape of the A matrix is
##  (2, 3)
```

```
print("Number of rows in A is \n",A.shape[0])
```

```
## Number of rows in A is
##  2
```

```
print("Number of columns in A is \n",A.shape[1])
```

```
## Number of columns in A is
##  3
```

Example 4.3 (Row and Column Vectors in Python)**.** You can more specifically build row or column vectors in Python using the `np.matrix()` command and then only specifying one row or column. For example, if you want the vectors

$$\boldsymbol{u} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad \text{and} \quad \boldsymbol{v} = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$$

then we would use the following Python code.

```
import numpy as np
u = np.matrix([[1],[2],[3]])
print("The column vector u is \n",u)
```

```
## The column vector u is
##  [[1]
##  [2]
##  [3]]
```

```
v = np.matrix([[1,2,3]])
print("The row vector v is \n",v)
```

```
## The row vector v is
```

```
##  [[1 2 3]]
```

Alternatively, if you want to define a column vector you can define a row vector (since there are far fewer brackets to keep track of) and then transpose the matrix to turn it into a column.

```
import numpy as np
u = np.matrix([[1,2,3]])
u = u.transpose()
print("The column vector u is \n",u)
```

```
## The column vector u is
##  [[1]
##  [2]
##  [3]]
```

Example 4.4 (Matrix Indexing)**.** Python indexes all arrays, vectors, lists, and matrices starting from index 0. Let's get used to this fact.

Consider the matrix A defined in the previous problem. Mathematically we know that the entry in row 1 column 1 is a 1, the entry in row 1 column 2 is a 2, and so on. However, with Python we need to shift the way that we enumerate the rows and columns of a matrix. Hence we would say that the entry in row 0 column 0 is a 1, the entry in row 0 column 1 is a 2, and so on.

Mathematically we can view all Python matrices as follows. If A is an $n \times n$ matrix then

$$A = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & \cdots & A_{0,n-1} \\ A_{1,0} & A_{1,1} & A_{1,2} & \cdots & A_{1,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n-1,0} & A_{n-1,1} & A_{n-1,2} & \cdots & A_{n-1,n-1} \end{pmatrix}$$

Similarly, we can view all vectors as follows. If $\boldsymbol{u}$ is an $n \times 1$ vector then

$$\boldsymbol{u} = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \end{pmatrix}$$

The following code should help to illustrate this indexing convention.

```
import numpy as np
A = np.matrix([[1,2,3],[4,5,6],[7,8,9]])
print("Entry in row 0 column 0 is",A[0,0])
```

```
## Entry in row 0 column 0 is 1
```

```
print("Entry in row 0 column 1 is",A[0,1])
```

```
## Entry in row 0 column 1 is 2
```

```
print("Entry in the bottom right corner",A[2,2])
```

```
## Entry in the bottom right corner 9
```

Exercise 4.1. Build your own matrix in Python and practice choosing individual entries from the matrix.

Example 4.5 (Matrix Slicing)**.** The last thing that we need to be familiar with is *slicing* a matrix. The term "slicing" gernally refers to pulling out individual rows, columns, entries, or blocks from a list, array, or matrix in Python. Examine the code below to see how to slice parts out of a `numpy` matrix.

```
import numpy as np
A = np.matrix([[1,2,3],[4,5,6],[7,8,9]])
print(A)
```

```
## [[1 2 3]
##  [4 5 6]
##  [7 8 9]]
```

```
print("The first column of A is \n",A[:,0])
```

```
## The first column of A is
##  [[1]
##  [4]
##  [7]]
```

```
print("The second row of A is \n",A[1,:])
```

```
## The second row of A is
##  [[4 5 6]]
```

```
print("The top left 2x2 sub matrix of A is \n",A[:-1,:-1])
```

```
## The top left 2x2 sub matrix of A is
##  [[1 2]
##  [4 5]]
```

```
print("The bottom right 2x2 sub matrix of A is \n",A[1:,1:])
```

```
## The bottom right 2x2 sub matrix of A is
##  [[5 6]
##  [8 9]]
```

```
u = np.array([1,2,3,4,5,6])
print("The first 3 entries of the vector u are \n",u[:3])
```

```
## The first 3 entries of the vector u are
##  [1 2 3]
```

```
print("The last entry of the vector u is \n",u[-1])
```

```
## The last entry of the vector u is
##  6
```

```
print("The last two entries of the vector u are \n",u[-2:])
```

```
## The last two entries of the vector u are
##  [5 6]
```

Exercise 4.2. Define the matrix A and the vector u in Python. Then perform all of the tasks below.

$$A = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \\ -3 & -2 & -1 & 0 \end{pmatrix} \quad \text{and} \quad \boldsymbol{u} = \begin{pmatrix} 10 \\ 20 \\ 30 \end{pmatrix}$$

a. Print the matrix A, the vector $\boldsymbol{u}$, the shape of A, and the shape of $\boldsymbol{u}$.
b. Print the first column of A.
c. Print the first two rows of A.
d. Print the first two entries of $\boldsymbol{u}$.
e. Print the last two entries of $\boldsymbol{u}$.
f. Print the bottom left 2×2 submatrix of A.
g. Print the middle two elements of the middle row of A.

4.3 Matrix and Vector Operations

Now let's start doing some numerical linear algebra. We start our discussion with the basics: the dot product and matrix multiplication. The numerical routines in Python's `numpy` packages are designed to do these tasks in very efficient ways but it is a good coding exercise to build your own dot product and matrix multiplication routines just to further cement the way that Python deals with these data structures and to remind you of the mathematical algorithms. What you will find in numerical linear algebra is that the indexing and the housekeeping in the codes is the hardest part. So why don't we start "easy".

4.3.1 The Dot Product

Exercise 4.3. This problem is meant to jog your memory about dot products, how to compute them, and what you might use them for. If your linear algebra is a bit rusty then read ahead a bit and then come back to this problem.

Consider two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ defined as

$$\boldsymbol{u} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \text{and} \quad \boldsymbol{v} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}.$$

a. Draw a picture showing both $\boldsymbol{u}$ and $\boldsymbol{v}$.
b. What is $\boldsymbol{u} \cdot \boldsymbol{v}$?
c. What is $\|\boldsymbol{u}\|$?
d. What is $\|\boldsymbol{v}\|$?
e. What is the angle between $\boldsymbol{u}$ and $\boldsymbol{v}$?
f. Give two reasons why we know that $\boldsymbol{u}$ is not perpendicular to $\boldsymbol{v}$.
g. What is the scalar projection of $\boldsymbol{u}$ onto $\boldsymbol{v}$? Draw this scalar projections on your picture from part (a).
h. What is the scalar projection of $\boldsymbol{v}$ onto $\boldsymbol{u}$? Draw this scalar projections on your picture from part (a).

Now let's get the formal definitions of the dot product on the table.
Definition 4.1 (The Dot Product)**.** The **dot product** of two vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ is

$$\boldsymbol{u} \cdot \boldsymbol{v} = \sum_{j=1}^{n} u_j v_j.$$

Without summation notation the dot product of two vectors is ,

$$\boldsymbol{u} \cdot \boldsymbol{v} = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n.$$

Alternatively, you may also recall that the dot product of two vectors is given geometrically as

$$\boldsymbol{u} \cdot \boldsymbol{v} = \|\boldsymbol{u}\| \|\boldsymbol{v}\| \cos\theta$$

where $\|\boldsymbol{u}\|$ and $\|\boldsymbol{v}\|$ are the magnitudes (or lengths) of $\boldsymbol{u}$ and $\boldsymbol{v}$ respectively, and θ is the angle between the two vectors. In physical applications the dot product is often used to find the angle between two vectors (e.g. between two forces). Hence, the last form of the dot product is often rewritten as

$$\theta = \cos^{-1}\left(\frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{u}\| \|\boldsymbol{v}\|}\right).$$

Definition 4.2 (Magnitude of a Vector)**.** The **magnitude** of a vector $\boldsymbol{u} \in \mathbb{R}^n$ is defined as

$$\|\boldsymbol{u}\| = \sqrt{\boldsymbol{u} \cdot \boldsymbol{u}}.$$

You should note that in two dimensions this collapses to the Pythagorean Theorem, and in higher dimensions this is just a natural extension of the Pythagorean Theorm.[1]

Exercise 4.4. Verify that $\sqrt{\boldsymbol{u}\cdot\boldsymbol{u}}$ indeed gives the Pythagorean Theorem for $\boldsymbol{u}\in\mathbb{R}^2$.

Exercise 4.5. Our task now is to write a Python function that accepts two vectors (defined as `numpy` arrays) and returns the dot product. Write this code without the use any loops.

```
import numpy as np
def myDotProduct(u,v):
    return # the dot product formula uses a product inside a sum.
```

Exercise 4.6. Test your `myDotProduct()` function on several dot products to make sure that it works. Example code to find the dot product between

$$\boldsymbol{u}=\begin{pmatrix}1\\2\\3\end{pmatrix} \quad \text{and} \quad \boldsymbol{v}=\begin{pmatrix}4\\5\\6\end{pmatrix}$$

is given below. Test your code on other vectors. Then implement an error catch into your code to catch the case where the two input vectors are not the same size. You will want to use the `len()` command to find the length of the vectors.

```
u = np.array([1,2,3])
v = np.array([4,5,6])
myDotProduct(u,v)
```

Exercise 4.7. Try sending Python lists instead of `numpy` arrays into your `myDotProduct` function. What happens? Why does it happen? What is the cautionary tale here? Modify your `myDotProduct()` function one more time so that it starts by converting the input vectors into `numpy` arrays.

```
u = [1,2,3]
v = [4,5,6]
myDotProduct(u,v)
```

---

[1] You should also note that $\|\boldsymbol{u}\|=\sqrt{\boldsymbol{u}\cdot\boldsymbol{u}}$ is not the only definition of distance. More generally, if you let $\langle\boldsymbol{u},\boldsymbol{v}\rangle$ be an inner product for $\boldsymbol{u}$ and $\boldsymbol{v}$ in some vector space $\mathcal{V}$ then $\|\boldsymbol{u}\|=\sqrt{\langle\boldsymbol{u},\boldsymbol{u}\rangle}$. In most cases in this text we will be using the dot product as our prefered inner product so we won't have to worry much about this particular natural extension of the definition of the length of a vector.

Exercise 4.8. The `numpy` library in Python has a built-in command for doing the dot product: `np.dot()`. Test the `np.dot()` command and be sure that it does the same thing as your `myDotProduct()` function.

4.3.2 Matrix Multiplication

Exercise 4.9. Next we will blow the dust off of your matrix multiplication skills. Verify that the product of A and B is indeed what we show below. Work out all of the details by hand.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \qquad B = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

$$AB = \begin{pmatrix} 27 & 30 & 33 \\ 61 & 68 & 75 \\ 95 & 106 & 117 \end{pmatrix}$$

Now that you've practiced the algorithm for matrix multiplication we can formalize the definition and then turn the algorithm into a Python function.

Definition 4.3 (Matrix Multiplicaiton)**.** If A and B are matrices with $A \in \mathbb{R}^{n \times p}$ and $B \in \mathbb{R}^{p \times m}$ then the product AB is defined as

$$(AB)_{ij} = \sum_{k=1}^{p} A_{ik} B_{kj}.$$

A moment's reflection reveals that each entry in the matrix product is actually a dot product,

(Entry in row i column j of AB) = (Row i of matrix A)$\cdot$(Column j of matrix B)$\,.$

Exercise 4.10. The definition of matrix multiplication above contains the cryptic phrase *a moment's reflection reveals that each entry in the matrix product is actually a dot product.* Let's go back to the matrices A and B defined above and re-evaluate the matrix multiplication algorithm to make sure that you see each entry as the end result of a dot product.

We want to find the product of matrices A and B using dot products.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \qquad B = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

a. Why will the product AB clear be a 3×3 matrix?
b. When we do matrix multiplication we take the product of a row from the first matrix times a column from the second matrix ... at least that's how many people think of it when they perform the operation by hand.
   i. The rows of A can be written as the vectors

$$\boldsymbol{a}_0 = \begin{pmatrix} 1 & 2 \end{pmatrix}$$

$$\boldsymbol{a}_1 = (\_\_\_\_\_ \quad \_\_\_\_\_)$$

$$\boldsymbol{a}_2 = (\_\_\_\_\_ \quad \_\_\_\_\_)$$

   ii. The columns of B can be written as the vectors

$$\boldsymbol{b}_0 = \begin{pmatrix} 7 \\ 10 \end{pmatrix}$$

$$\boldsymbol{b}_1 = \begin{pmatrix} \_\_\_\_\_ \\ \_\_\_\_\_ \end{pmatrix}$$

$$\boldsymbol{b}_2 = \begin{pmatrix} \_\_\_\_\_ \\ \_\_\_\_\_ \end{pmatrix}$$

c. Now let's write each entry in the product AB as a dot product.

$$AB = \begin{pmatrix} \boldsymbol{a}_0 \cdot \boldsymbol{b}_0 & \_\_\_ \cdot \_\_\_ & \_\_\_ \cdot \_\_\_ \\ \_\_\_ \cdot \_\_\_ & \_\_\_ \cdot \_\_\_ & \_\_\_ \cdot \_\_\_ \\ \_\_\_ \cdot \_\_\_ & \_\_\_ \cdot \_\_\_ & \_\_\_ \cdot \_\_\_ \end{pmatrix}$$

d. Verify that you get

$$AB = \begin{pmatrix} 27 & 30 & 33 \\ 61 & 68 & 75 \\ 95 & 106 & 117 \end{pmatrix}$$

when you perform all of the dot products from part (c).

Exercise 4.11. The observation that matrix multiplication is just a bunch of dot products is what makes the code for doing matrix multiplication very fast and very streamlined. We want to write a Python function that accepts two `numpy` matrices and returns the product of the two matrices. Inside the code we will leverage the `np.dot()` command to do the appropriate dot products.

Partial code is given below. Fill in all of the details and give ample comments showing what each line does.

```
import numpy as np
def myMatrixMult(A,B):
    # Get the shapes of the matrices A and B.
    # Then write an if statement that catches size mismatches in the matrices.
    # Next build a zeros matrix that is the correct size for the product of A and B.
```

```
    AB = # this is a zeros matix that will be filled with the values from the product
    #
    # Next we do a double for-loop that loops through all of the indices of the produc
    for i in range(n): # loop over the rows of AB
        for j in range(m): # loop over the columns of AB
            # use the np.dot() command to take the appropriate dot product
            AB[i,j] = ???
    return AB
```

Use the following test code to determine if you actually get the correct matrix product out of your code.

```
A = np.matrix([[1,2],[3,4],[5,6]])
B = np.matrix([[7,8,9],[10,11,12]])
AB = myMatrixMult(A,B)
print(AB)
```

Exercise 4.12. Try your `myMatrixMult()` function on several other matrix multiplication problems.

Exercise 4.13. Build in an error catch so that your `myMatrixMul()` function catches when the input matrices do not have compatible sizes for multiplication. Write your code so that it returns an appropriate error message in this special case.

Now that you've been through the exercise of building a matrix multiplication function we will admit that using it inside larger coding problems would be a bit cumbersome (and perhaps annoying). It would be nice to just type `*` and have Python just *know* that you mean to do matrix multiplication. The trouble is that there are many different versions of multiplication and any programming language needs to be told explicitly which type they're dealing with. This is where `numpy` and `np.matrix()` come in quite handy.

Exercise 4.14 (Matrix Multiplication with Python)**.** Python will handle matrix multiplication easily so long as the matrices are defined as `numpy` matrices with `np.matrix()`. For example, with the matrices A and B from above if you can just type `A*B` in Python and you will get the correct result. Pretty nice!! Let's take another moment to notice, though, that regular Python arrays do not behave in the same way. What happens if you run the following Python code?

```
A = [[1,2],[3,4],[5,6]] # a Python list of lists
B = [[7,8,9],[10,11,12]] # a Python list of lists
A*B
```

Example 4.6 (Element-by-Element Multiplication)**.** Sometimes it is convenient to do naive multiplication of matrices when you code. That is, if you have two matrices that are the same size, "naive multiplication" would just line up the matrices on top of each other and multiply the corresponding entries.[2] In Python the tool to do this is `np.multiply()`. The code below demonstrates this tool with the matrices

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}.$$

(Note that the product AB does not make sense under the mathematical definition of matrix multiplication, but it does make sense in terms of element-by-element ("naive") multiplication.)

```
import numpy as np
A = [[1,2],[3,4],[5,6]]
B = [[7,8],[9,10],[11,12]]
np.multiply(A,B)
```

```
## array([[ 7, 16],
##        [27, 40],
##        [55, 72]])
```

The key takeaways for doing matrix multiplication in Python are as follows:

- If you are doing linear algebra in Python then you should define vectors with `np.array()` and matrices with `np.matrix()`.
- If your matrices are defined with `np.matrix()` then `*` does regular matrix multiplication and `np.multiply()` does element-by-element multiplication.

4.4 The LU Factorization

One of the many classic problems of linear algebra is to solve the linear system $A\boldsymbol{x} = \boldsymbol{b}$ where A is a matrix of coefficients and $\boldsymbol{b}$ is a vector of right-hand sides. You likely recall your go-to technique for solving systems was row reduction (or Gaussian Elimination or RREF). Furthermore, you likely recall from your linear

[2] You might have thought that *naive multiplication* was a much more natural way to do matrix multiplication when you first saw it. Hopefully now you see the power in the definition of matrix multiplication that we actually use. If not, then I give you this moment to ponder that (a) matrix multiplication is just a bunch of dot products, and (b) dot products can be seen as projections. Hence, matrix multiplication is really just a projection of the rows of A onto the columns of B. This has much more rich geometric flavor than *naive multiplication*.

algebra class that you rarely actually did row reduction by hand, and instead you relied on a computer to do most of the computations for you. Just what was the computer doing, exactly? Do you think that it was actually following the same algorithm that you did by hand?

4.4.1 A Recap of Row Reduction

Let's blow the dust off your row reduction skills before we look at something better.

Exercise 4.15. Solve the following system of equations by hand.

$$\begin{aligned} x_0 + 2x_1 + 3x_2 &= 1 \\ 4x_0 + 5x_1 + 6x_2 &= 0 \\ 7x_0 + 8x_1 &= 2 \end{aligned}$$

Note that the system of equations can also be written in the matrix form

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}$$

If you need a nudge to get started then jump ahead to the next problem.

Exercise 4.16. We want to solve the system of equations

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}$$

Row Reduction Process:

Note: Throughout this discussion we use Python-type indexing so the rows and columns are enumerated starting at 0. That is to say, we will talk about row 0, row 1, and row 2 of a matrix instead of rows 1, 2, and 3.

a. Augment the coefficient matrix and the vector on the right-hand side to get

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 0 & 2 \end{array}\right)$$

b. The goal of row reduction is to perform elementary row operations until our augmented matrix gets to (or at least gets as close as possible to)

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & \star \\ 0 & 1 & 0 & \star \\ 0 & 0 & 1 & \star \end{array}\right)$$

The allowed elementary row operations are:

i. We are allowed to scale any row.
ii. We can add two rows.
iii. We can interchange two rows.

c. We are going to start with column 0. We already have the "1" in the top left corner so we can use it to eliminate all of the other values in the first column of the matrix.
   i. For example, if we multiply the 0^{th} row by -4 and add it to the first row we get
   $$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & -3 & -6 & -4 \\ 7 & 8 & 0 & 2 \end{array}\right).$$
   ii. Multiply row 0 by a scalar and add it to row 2. Your end result should be
   $$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & -3 & -6 & -4 \\ 0 & -6 & -21 & -5 \end{array}\right).$$
   What did you multiply by? Why?

d. Now we should deal with column 1.
   i. We want to get a 1 in row 1 column 1. We can do this by scaling row 1. What did you scale by? Why? Your end result should be
   $$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & 1 & 2 & \frac{4}{3} \\ 0 & -6 & -21 & -5 \end{array}\right).$$
   ii. Now scale row 1 by something and add it to row 0 so that the entry in row 0 column 1 becomes a 0.
   iii. Next scale row 1 by something and add it to row 2 so that the entry in row 2 column 1 becomes a 0.
   iv. At this point you should have the augmented system
   $$\left(\begin{array}{ccc|c} 1 & 0 & -1 & -\frac{5}{3} \\ 0 & 1 & 2 & \frac{4}{3} \\ 0 & 0 & -9 & 3 \end{array}\right).$$

e. Finally we need to work with column 2.
   i. Make the value in row 2 column 2 a 1 by scaling row 2. What did you scale by? Why?
   ii. Scale row 2 by something and add it to row 1 so that the entry in row 1 column 2 becomes a 0. What did you scale by? Why?
   iii. Scale row 2 by something and add it to row 0 so that the entry in row 0 column 2 becomes a 0. What did you scale by? Why?

iv. By the time you've made it this far you should have the system

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & -\frac{1}{3} \end{array}\right)$$

and you should be able to read off the solution to the system.

f. You should verify your answer in two different ways:
   i. If you substitute your values into the original system then all of the equal signs should be true. Verify this.
   ii. If you substitute your values into the matrix equation and perform the matrix-vector multiplication on the left-hand side of the equation you should get the right-hand side of the equation. Verify this.

Exercise 4.17. Summarize the process for doing Gaussian Elimination to solve a square system of linear equations.

4.4.2 The LU Decomposition

You may have used the `rref()` command either on a calculator in other software to perform row reduction in the past. You will be surprised to learn that there is no `rref()` command in Python's `numpy` library! That's because there are far more efficient and stable ways to solve a linear system on a computer. There is an `rref` command in Python's `sympy` (symbolic Python) library, but given that it works with symbolic algebra it is quite slow.

In solving systems of equations we are interested in equations of the form $A\boldsymbol{x} = \boldsymbol{b}$. Notice that the $\boldsymbol{b}$ vector is just along for the ride, so to speak, in the row reduction process since none of the values in $\boldsymbol{b}$ actually cause you to make different decisions in the row reduction algorithm. Hence, we only really need to focus on the matrix A. Furthermore, let's change our awfully restrictive view of always seeking a matrix of the form

$$\left(\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & \star \\ 0 & 1 & \cdots & 0 & \star \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \star \end{array}\right)$$

and instead say:

> *What if we just row reduce until the system is simple enough to solve by hand?*

That's what the next several exercises are going to lead you to. Our goal here is to develop an algorithm that is fast to implement on a computer and simultaneously

performs the same basic operations as row reduction for solving systems of linear equations.

Exercise 4.18. Let A be defined as

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}.$$

a. The first step in row reducing A would be to multiply row 0 by -4 and add it to row 1. Do this operation by hand so that you know what the result is supposed to be. Check out the following amazing observation. Define the matrix L_1 as follows:

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Now multiply L_1 and A.

$$L_1 A = \begin{pmatrix} \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \end{pmatrix}$$

What just happened?!

b. Let's do it again. The next step in the row reduction of your result from part (b) would be to multiply row 0 by -7 and add to row 2. Again, do this by hand so you know what the result should be. Then define the matrix L_2 as

$$L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -7 & 0 & 1 \end{pmatrix}$$

and find the product $L_2\,(L_1 A)$.

$$L_2\,(L_1 A) = \begin{pmatrix} \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \end{pmatrix}$$

Pure insanity!!

c. Now let's say that you want to make the entry in row 2 column 1 into a 0 by scaling row 1 by something and then adding to row 2. Determine what the scalar would be and then determine which matrix, call it L_3, would do the trick so that $L_3(L_2 L_1 A)$ would be the next row reduced step.

$$L_3 = \begin{pmatrix} 1 & \_\_\_ & \_\_\_ \\ \_\_\_ & 1 & \_\_\_ \\ \_\_\_ & \_\_\_ & 1 \end{pmatrix}$$

$$L_3\left(L_2L_1A\right) = \begin{pmatrix} \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \\ \_\_\_ & \_\_\_ & \_\_\_ \end{pmatrix}$$

Exercise 4.19. Apply the same idea from the previous problem to do the first three steps of row reduction to the matrix

$$A = \begin{pmatrix} 2 & 6 & 9 \\ -6 & 8 & 1 \\ 2 & 2 & 10 \end{pmatrix}$$

Exercise 4.20. Now let's make a few observations about the two previous problems.

a. What will multiplying A by a matrix of the form

$$\begin{pmatrix} 1 & 0 & 0 \\ c & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

do?

b. What will multiplying A by a matrix of the form

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ c & 0 & 1 \end{pmatrix}$$

do?

c. What will multiplying A by a matrix of the form

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & c & 1 \end{pmatrix}$$

do?

d. More generally: If you wanted to multiply row j of an $n \times n$ matrix by c and add it to row k, that is the same as multiplying by what matrix?

Exercise 4.21. After doing all of the matrix products, $L_3L_2L_1A$, the resulting matrix will have zeros in the entire lower triangle. That is, all of the nonzero entries of the resulting matrix will be on the main diagonal or above. We call this matrix U, for upper triangular. Hence, we have formed a matrix

$$L_3L_2L_1A = U$$

and if we want to solve for A we would get

$$A = (\_\_\_\_\_)^{-1}(\_\_\_\_\_)^{-1}(\_\_\_\_\_)^{-1}U$$

(Take care that everything is in the right order in your answer.)

Exercise 4.22. It would be nice, now, if the inverses of the L matrices were easy to find. Use `np.linalg.inv()` to directly compute the inverse of L_1, L_2, and L_3 for each of the example matrices. Then complete the statement: If L_k is an identity matrix with some nonzero c in row i and column j then L_k^{-1} is what matrix?

Exercise 4.23. We started this discussion with A as

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}$$

and we defined

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -7 & 0 & 1 \end{pmatrix}, \quad \text{and} \quad L_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix}.$$

Based on your answer to the previous exercises we know that

$$A = L_1^{-1} L_2^{-1} L_3^{-1} U.$$

Explicitly write down the matrices L_1^{-1}, L_2^{-1}, and L_3^{-1}.

Now explicitly find the product $L_1^{-1} L_2^{-1} L_3^{-1}$ and call this product L. Verify that L itself is also a lower triangular matrix with ones on the main diagonal. Moreover, take note of exactly the form of the matrix. The answer should be super surprising to you!!

Throughout all of the preceding exercises, our final result is that we have factored the matrix A into the product of a lower triangular matrix and an upper triangular matrix. Stop and think about that for a minute ... we just factored a matrix!

Let's return now to our discussion of solving the system of equations $A\boldsymbol{x} = \boldsymbol{b}$. If A can be factored into $A = LU$ then the system of equations can be rewritten as $LU\boldsymbol{x} = \boldsymbol{b}$. As we will see in the next subsection, solving systems of equations with triangular matrices is super fast and relatively simple! Hence, we have partially achieved our modified goal of reducing the row reduction into some simpler case.[3]

It remains to implement the LU decomposition (also called the LU factorization) in Python.

[3]Take careful note here. We have actually just built a special case of the LU decomposition. Remember that in row reduction you are allowed to swap the order of the rows, but in our LU algorithm we don't have any row swaps. The version of LU with row swaps is called LU with partial pivoting. We won't built the full partial pivoting algorithm in this text but feel free to look it up. The wikipedia page is a decent place to start. What you'll find is that there are indeed many different versions of the LU decomposition.

Definition 4.4 (The LU Factorization)**.** The following Python function takes a square matrix A and outputs the matrices L and U such that $A = LU$. The entire code is given to you. It will be up to you in the next exercise to pick apart every step of the function.

```
def myLU(A):
    n = A.shape[0] # get the dimension of the matrix A
    L = np.matrix( np.identity(n) ) # Build the identity part L
    U = A # start a placeholder for the U matrix
    for j in range(0,n-1):
        for i in range(j+1,n):
            mult = A[i,j] / A[j,j]
            A[i, j+1:n] = A[i, j+1:n] - mult * A[j,j+1:n]
            U[i, j+1:n] = A[i, j+1:n]
            L[i,j] = mult
            U[i,j] = 0
    return L,U
```

Exercise 4.24. Go to Definition 4.4 and go through every iteration of every loop **by hand** starting with the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}.$$

Give details of what happens at every step of the algorithm. I'll get you started.

- `n=3`, `L` starts as an identity matrix of the correct size, and `U` starts as a copy of `A`.
- Start the outer loop: `j=0`: (`j` is the counter for the column)
  - Start the inner loop: `i=1`: (`i` is the counter for the row)
    - `mult = A[1,0] / A[0,0]` so `mult`$= 4/1$.
    - `A[1, 1:3] = A[1, 1:3] - 4 * A[0,1:3]`. Translated, this states that columns 1 and 2 of matrix A took their original value minus 4 times the corresponding values in row 0.
    - `U[1, 1:3] = A[1, 1:3]`. Now we replace the locations in U with the updated information from our first step of row reduction.
    - `L[1,0]=4`. We now fill the L matrix with the proper value.
    - `U[1,0]=0`. Finally, we zero out the lower triangle piece of the U matrix which we've now taken care of.
  - `i=2`:
    - ... keep going from here ...

Exercise 4.25. Apply your new `myLU` code to other square matrices and verify that indeed A is the product of the resulting L and U matrices. You can produce

a random matrix with `np.random.randn(n,n)` where `n` is the number of rows and columns of the matrix. For example, `np.random.randn(10,10)` will produce a random 10×10 matrix with entries chosen from the normal distribution with center 0 and standard deviation 1. Random matrices are just as good as any other when testing your algorithm.

4.4.3 Solving Triangular Systems

We now know that row reduction is just a collection of sneaky matrix multiplications. In the previous exercises we saw that we can often turn our system of equations $A\boldsymbol{x} = \boldsymbol{b}$ into the system $LU\boldsymbol{x} = \boldsymbol{b}$ where L us lower triangular (with ones on the main diagonal) and U is upper triangular. But why was this important?

Well, if $LU\boldsymbol{x} = \boldsymbol{b}$ then we can rewrite our system of equations as two systems:

$$\text{An upper triangular system: } U\boldsymbol{x} = \boldsymbol{y}$$

and

$$\text{A lower triangular system: } L\boldsymbol{y} = \boldsymbol{b}.$$

In the following exercises we will devise algorithms for solving triangular systems. After we know how to work with triangular systems we'll put all of the pieces together and show how to leverage the LU decomposition and the solution techniques for triangular systems to quickly and efficiently solve linear systems.

Exercise 4.26. Outline a fast algorithm (without formal row reduction) for solving the lower triangular system

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}.$$

Exercise 4.27. As a convention we will always write our lower triangular matrices with ones on the main diagonal. Generalize your steps from the previous exercise so that you have an algorithm for solving any lower triangular system. The most natural algorithm that most people devise here is called **forward substitution**.

Definition 4.5 (The Forward Substutition Algorithm ('lsolve')). The general statement of the Forward Substitution Algorithm is:

Solve $L\boldsymbol{y} = \boldsymbol{b}$ for $\boldsymbol{y}$, where the matrix L is assumed to be lower triangular with ones on the main diagonal.

The code below gives a full implementation of the **Forward Substitution** algorithm (also called the `lsolve` algorithm).

```
def lsolve(L, b):
    L = np.matrix(L) # make sure L is the correct data type
    n = b.size # what does this do?
    y = np.matrix( np.zeros( (n,1)) ) # what does this do?
    for i in range(n):
        # start the loop by assigning y to the value on the right
        y[i] = b[i]
        for j in range(i): # now adjust y
            y[i] = y[i] - L[i,j] * y[j]
    return(y)
```

Exercise 4.28. Work with your partner(s) to apply the `lsolve()` code to the lower triangular system

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}$$

by hand. It is incredibly important to impelement numerical linear algebra routines by hand a few times so that you truly understand how everything is being tracked and calculated.

I'll get you started.

- Start: `i=0`:
  - `y[0]=1` since `b[0]=1`.
  - The next `for` loop does not start since `range(0)` has no elements (stop and think about why this is).
- Next step in the loop: `i=1`:
  - `y[1]` is initialized as 0 since `b[1]=0`.
  - Now we enter the inner loop at `j=0`:
    - What does `y[1]` become when `j=0`?
  - Does `j` increment to anything larger?
- Finally we increment `i` to `i=2`:
  - What does `y[2]` get initialized to?
  - Enter the inner loop at `j=0`:
    - What does `y[2]` become when `j=0`?
  - Increment the inner loop to `j=1`:
    - What does `y[2]` become when `j=1`?
- Stop

Exercise 4.29. Copy the code from Definition 4.5 into a Python function but in your code write a comment on every line stating what it is doing. Write a test script that creates a lower triangular matrix of the correct form and a

right-hand side $\boldsymbol{b}$ and solve for $\mathbf{y}$. Test your code by giving it a large lower triangular system.

Now that we have a method for solving lower triangular systems, let's build a similar method for solving upper triangular systems. The merging of lower and upper triangular systems will play an important role in solving systems of equations.

Exercise 4.30. Outline a fast algorithm (without formal row reduction) for solving the upper triangular system

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & -9 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -4 \\ 3 \end{pmatrix}$$

The most natural algorithm that most people devise here is called **backward substitution**. Notice that in our upper triangular matrix we do not have a diagonal containing all ones.

Exercise 4.31. Generalize your backward substitution algorithm from the previous problem so that it could be applied to any upper triangular system.

Definition 4.6 (Backward Substitution Algorithm)**.** The following code solves the problem $U\boldsymbol{x} = \boldsymbol{y}$ using backward substitution. The matrix U is assumed to be upper triangular. You'll notice that most of this code is incomplete. It is your job to complete this code, and the next exercise should help.

```
def usolve(U, y):
    U = np.matrix(U)
    n = y.size
    x = np.matrix( np.zeros( (n,1)))
    for i in range( ??? ):      # what should we be looping over?
        x[i] = y[i] / ???       # what should we be dividing by?
        for j in range( ??? ): # what should we be looping over:
            x[i] = x[i] - U[i,j] * x[j] / ??? # complete this line ... what does this line do?
    return(x)
```

Exercise 4.32. Now we will work through the backward substitution algorithm to help fill in the blanks in the code. Consider the upper triangular system

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & -9 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -4 \\ 3 \end{pmatrix}$$

Work the code from Definition 4.6 to solve the system. Keep track of all of the indices as you work through the code. You may want to work this problem in conjunction with the previous two problems to unpack all of the parts of the *backward substitution* algorithm.

I'll get you started.

- In your backward substitution algorithm you should have started with the last row, therefore the outer loop starts at `n-1` and reads backward to 0. (Why are we starting at `n-1` and not `n`?)
- Outer loop: `i=2`:
  - We want to solve the equation $-9x_2 = 3$ so the clear solution is to divide by -9. In code this means that `x[2]=y[2]/U[2,2]`.
  - There is nothing else to do for row 3 of the matrix, so we should not enter the inner loop. How can we keep from entering the inner loop?
- Outer loop: `i=1`:
  - Now we are solving the algebraic equation $-3x_1 - 6x_2 = -4$. If we follow the high school algebra we see that $x_1 = \frac{-4-(-6)x_2}{-3}$ but this can be rearranged to
    $$x_1 = \frac{-4}{-3} - \frac{-6x_2}{-3}.$$
    So we can initialize x_1 with $x_1 = \frac{-4}{-3}$. In code, this means that we initialize with `x[1] = y[1] / U[1,1]`.
  - Now we need to enter the inner loop at `j=2`: (why are we entering the loop at `j=2`?)
    - To complete the algebra we need to take our initialized value of `x[1]` and subtract off $\frac{-6x_2}{-3}$. In code this is `x[1] = x[1] - U[1,2] * x[2] / U[1,1]`
  - There is nothing else to do so the inner loop should end.
- Outer loop: `i=0`:
  - Finally, we are solving the algebraic equation $x_0 + 2x_1 + 3x_2 = 1$ for x_0. The clear and obvious solution is $x_0 = \frac{1-2x_1-3x_2}{1}$ (why am I explicitly showing the division by 1 here?).
  - Initialize x_0 at `x[0] = ???`
  - Enter the inner loop at `j=2`:
    - Adjust the value of `x[0]` by subtracting off $\frac{3x_2}{1}$. In code we have 'x[0] = x[0] - ??? * ??? / ???
  - Increment `j` to `j=1`:
    - Adjust the value of `x[0]` by subtracting off $\frac{2x_1}{1}$. In code we have 'x[0] = x[0] - ??? * ??? / ???
- Stop.
- You should now have a solution to the equation $U\boldsymbol{x} = \boldsymbol{y}$. Substitute your solution in and verify that your solution is correct.

Exercise 4.33. Copy the code from Definition 4.6 into a Python function but in your code write a comment on every line stating what it is doing. Write a test script that creates an upper triangular matrix of the correct form and a right-hand side $\boldsymbol{y}$ and solve for $\boldsymbol{x}$. Your code needs to work on systems of arbitrarily large size.

4.4.4 Solving Systems with LU

We are finally ready for the punch line of this whole LU and triangular systems business!

Exercise 4.34. If we want to solve $A\boldsymbol{x} = \boldsymbol{b}$ then

a. If we can, write the system of equations as $LU\boldsymbol{x} = \boldsymbol{b}$.
b. Solve $L\boldsymbol{y} = \boldsymbol{b}$ for $\boldsymbol{y}$ using forward substitution.
c. Solve $U\boldsymbol{x} = \boldsymbol{y}$ for $\boldsymbol{x}$ using backward substitution.

Pick a matrix A and a right-hand side $\boldsymbol{b}$ and solve the system using this process..

Exercise 4.35. Try the process again on the 3×3 system of equations

$$\begin{pmatrix} 3 & 6 & 8 \\ 2 & 7 & -1 \\ 5 & 2 & 2 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} -13 \\ 4 \\ 1 \end{pmatrix}$$

That is: Find matrices L and U such that $A\boldsymbol{x} = \boldsymbol{b}$ can be written as $LU\boldsymbol{x} = \boldsymbol{b}$. Then do two triangular solves to determine $\boldsymbol{x}$.

Let's take stock of what we have done so far.

- Solving lower triangular systems is super fast and easy!
- Solving upper triangular systems is super fast and easy (so long as we never divide by zero).

- It is often possible to rewrite the matrix A as the product of a lower triangular matrix L and an upper triangular matrix U so $A = LU$.
- Now we can re-frame the equation $A\boldsymbol{x} = \boldsymbol{b}$ as $LU\boldsymbol{x} = \boldsymbol{b}$.

- Substitute $\boldsymbol{y} = U\boldsymbol{x}$ so the system becomes $L\boldsymbol{y} = \boldsymbol{b}$. Solve for $\boldsymbol{y}$ with forward substitution.
- Now solve $U\boldsymbol{x} = \boldsymbol{y}$ using backward substitution.

We have successfully take row reduction and turned into some fast matrix multiplications and then two very quick triangular solves. Ultimately this will be a faster algorithm for solving a system of linear equations.

Definition 4.7 (Solving Linear Systems with the LU Decomposition)**.** Let A be a square matrix in $\mathbb{R}^{n\times n}$ and let $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n$. To solve the problem $A\boldsymbol{x} = \boldsymbol{b}$,

1. Factor A into lower and upper triangular matrices $A = LU$.
   `L, U = myLU(A)`
2. The system can now be written as $LU\boldsymbol{x} = \boldsymbol{b}$. Substitute $U\boldsymbol{x} = \boldsymbol{y}$ and solve the system $L\boldsymbol{y} = \boldsymbol{b}$ with forward substitution. `y = lsolve(L,b)`
3. Finally, solve the system $U\boldsymbol{x} = \boldsymbol{y}$ with backward substitution.
   `x = usolve(U,y)`

Exercise 4.36. Test your `lsolve`, `usolve`, and `myLU` functions on a linear system for which you know the answer. Then test your problem on a system that you don't know the solution to. As a way to compare your solutions you should:

- Find Python's solution using `np.linalg.solve()` and compare your answer to that one using `np.linalg.norm()` to give the error between the two.
- Time your code using the `time` library as follows
  - use the code `starttime = time.time()` before you start the main computation
  - use the code `endtime = time.time()` after the main computation
  - then calculate the total elapsed time with `totaltime = endtime - starttime`
- Compare the timing of your LU solve against `np.linalg.solve()` and against the RREF algorithm in the `sympy` library.

```
A = # Define your matrix
b = # Defind your right-hand side vector

# build a symbolic augmented matrix
import sympy as sp
Ab = sp.Matrix(np.c_[A,b]) # note that np.c_[A,b] does a column concatenation of A wit

t0 = time.time()
Abrref = # row reduce the symbolic augmented matrix
t1 = time.time()
RREFTime = t1-t0


t0=time.time()
```

```
exact = # use np.linalg.solve() to solve the numerical linear system
t1=time.time()
exactTime = t1-t0

t0 = time.time()
L, U = # get L and U from your myLU
y = # use forward substitution to get y
x = # use backward substituation to get x
t1 = time.time()
LUTime = t1-t0

print("Time for symbolic RREF:\t\t\t",RREFTime)
print("Time for np.linalg.solve() solution:\t",exactTime)
print("Time for LU solution:\t\t\t",LUTime)
print("Error between LU and np.linalg.solve():",np.linalg.norm(x-exact))
```

Exercise 4.37. The LU decomposition is not perfect. Discuss where the algorithm will fail.

Exercise 4.38. What happens when you try to solve the system of equations

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 7 \\ 9 \\ -3 \end{pmatrix}$$

with the LU decomposition algorithm? Discuss.

4.5 The QR Factorization

In this section we will try to find an improvement on the LU factorization scheme from the previous section. What we'll do here is leverage the geometry of the column space of the A matrix instead of leveraging the row reduction process.

Exercise 4.39. We want to solve the system of equations

$$\begin{pmatrix} 1/3 & 2/3 & 2/3 \\ 2/3 & 1/3 & -2/3 \\ -2/3 & 2/3 & -1/3 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 6 \\ 12 \\ -9 \end{pmatrix}.$$

a. We could do row reduction by hand ... yuck ... don't do this.
b. We could apply our new-found skills with the LU decomposition to solve the system, so go ahead and do that with your Python code.

c. What do you get if you compute the product A^TA?
   i. Why do you get what you get? In other words, what was special about A that gave such an nice result?
   ii. What does this mean about the matrices A and A^T?
d. Now let's leverage what we found in part (c) to solve the system of equations $A\boldsymbol{x} = \boldsymbol{b}$ much faster. Multiply both sides of the matrix equation by A^T, and now you should be able to just read off the solution. This seems amazing!!
e. What was it about this particular problem that made part (d) so elegant and easy?

Theorem 4.1 (Orthonomal Matrices)**.** *The previous exercise tells us something amazing: If A is an orthonormal matrix where the columns are mutually orthogonal and every column is a unit vector, then $A^T = A^{-1}$ and to solve the system of equation $A\boldsymbol{x} = \boldsymbol{b}$ we simply need to multiply both sides of the equation by A^T. Hence, the solution to $A\boldsymbol{x} = \boldsymbol{b}$ is just $\boldsymbol{x} = A^T\boldsymbol{b}$ in this special case.*

Theorem 4.1 begs an obvious question: *Is there a way to turn any matrix A into an orthogonal matrix so that we can solve $A\boldsymbol{x} = \boldsymbol{b}$ in this same very efficient and fast way?*

The answer: Yes. Kind of.

In essence, if we can factor our coefficient matrix into an orthonormal matrix and some other nicely formatted matrix (like a triangular matrix, perhaps) then the job of solving the linear system of equations comes down to matrix multiplication and a quick triangular solve – both of which are extremely extremely fast!

What we will study in this section is a new matrix factorization called the QR factorization who's goal is to convert the matrix A into a product of two matrices, Q and R, where Q is orthonormal and R is upper triangular.

Exercise 4.40. Let's say that we have a matrix A and we know that it can be factored into $A = QR$ where Q is an orthonormal matrix and R is an upper triangular matrix. How would we then leverage this factorization to solve the system of equation $A\boldsymbol{x} = \boldsymbol{b}$ for $\boldsymbol{x}$?

Before proceeding to the algorithm for the QR factorization let's pause for a moment and review scalar and vector projections from Linear Algebra. In Figure 4.1 we see a graphical depiction of the vector $\boldsymbol{u}$ projected onto vector $\boldsymbol{v}$. Notice that the projection is indeed the perpendicular projection as this is what seems natural geometrically.

The **vector projection** of $\boldsymbol{u}$ onto $\boldsymbol{v}$ is the vector $c\boldsymbol{v}$. That is, the vector projection of $\boldsymbol{u}$ onto $\boldsymbol{v}$ is a scalar multiple of the vector $\boldsymbol{v}$. The value of the

scalar c is called the **scalar projection** of $\boldsymbol{u}$ onto $\boldsymbol{v}$.

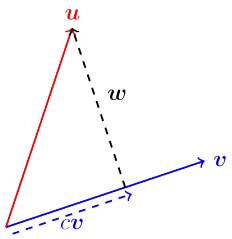


Figure 4.1: Projection of one vector onto another.

We can arrive at a formula for the scalar projection rather easily is we consider that the vector $\boldsymbol{w}$ in Figure 4.1 must be perpendicular to $c\boldsymbol{v}$. Hence

$$\boldsymbol{w} \cdot (c\boldsymbol{v}) = 0.$$

From vector geometry we also know that $\boldsymbol{w} = \boldsymbol{u} - c\boldsymbol{v}$. Therefore

$$(\boldsymbol{u} - c\boldsymbol{v}) \cdot (c\boldsymbol{v}) = 0.$$

If we distribute we can see that

$$c\boldsymbol{u} \cdot \boldsymbol{v} - c^2\boldsymbol{v} \cdot \boldsymbol{v} = 0$$

and therefore either $c = 0$, which is only true if $\boldsymbol{u} \perp \boldsymbol{v}$, or

$$c = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\boldsymbol{v} \cdot \boldsymbol{v}} = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{v}\|^2}.$$

Therefore,

- the **scalar projection of $\boldsymbol{u}$ onto $\boldsymbol{v}$** is
$$c = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{v}\|^2}$$
- the **vector projection of $\boldsymbol{u}$ onto $\boldsymbol{v}$** is
$$c\boldsymbol{v} = \left(\frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{v}\|^2}\right)\boldsymbol{v}$$

Another problem related to scalar and vector projections is to take a basis for the column space of a matrix and transform that basis into an orthogonal (or orthonormal) basis. Indeed, in Figure 4.1 is we have the matrix

$$A = \begin{pmatrix} | & | \\ \boldsymbol{u} & \boldsymbol{v} \\ | & | \end{pmatrix}$$

it should be clear from the picture that the columns of this matrix are not perpendicular. However, if we take the vector $\boldsymbol{v}$ and the vector $\boldsymbol{w}$ we do arrive at two orthogonal vector that form a basis for the same space. Moreover, if we normalize these vectors (by dividing by their respective lengths) then we can easily transform the original basis for the column space of A into an orthonormal basis. This process is called the Gramm-Schmidt process, and you may have encountered it in your Linear Algebra class.

Now we return to our goal of finding a way to factor a matrix A into an orthonormal matrix Q and an upper triangular matrix R. The algorithm that we are about to build depends greatly on the ideas of scalar and vector projections.

Exercise 4.41. We want to build a QR factorization of the matrix A in the matrix equation $A\boldsymbol{x} = \boldsymbol{b}$ so that we can leverage the fact that solving the equation $QR\boldsymbol{x} = \boldsymbol{b}$ is easy. Consider the matrix A defined as

$$A = \begin{pmatrix} 3 & 1 \\ 4 & 1 \end{pmatrix}.$$

Notice that the columns of A are NOT othonormal (they are not unit vectors and they are not perpendicular to each other).

a. Draw a picture of the two column vectors of A in $\mathbb{R}^2$. We'll use this picture to build geometric intuition for the rest of the QR factorization process.

b. Define $\boldsymbol{a}_0$ as the first column of A and $\boldsymbol{a}_1$ as the second column of A. That is

$$\boldsymbol{a}_0 = \begin{pmatrix} 3 \\ 4 \end{pmatrix} \quad \text{and} \quad \boldsymbol{a}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Turn $\boldsymbol{a}_0$ into a unit vector and call this unit vector $\boldsymbol{q}_0$

$$\boldsymbol{q}_0 = \frac{\boldsymbol{a}_0}{\|\boldsymbol{a}_0\|} = \begin{pmatrix} \_\_\_\_\_ \\ \_\_\_\_\_ \end{pmatrix}.$$

This vector $\boldsymbol{q}_0$ will be the first column of the 2×2 matrix Q. Why is this a nice place to start building the Q matrix (think about the desired structure of Q)?

c. In your picture of $\boldsymbol{a}_0$ and $\boldsymbol{a}_1$ mark where $\boldsymbol{q}_0$ is. Then draw the orthogonal projection from $\boldsymbol{a}_1$ onto $\boldsymbol{q}_0$. In your picture you should now see a right triangle with $\boldsymbol{a}_1$ on the hypotenuse, the projection of $\boldsymbol{a}_1$ onto $\boldsymbol{q}_0$ on one leg, and the second leg is the vector difference of the hypotenuse and the first leg. Simplify the projection formula for leg 1 and write the formula for leg 2.

$$\begin{aligned} \text{hypotenuse} &= \boldsymbol{a}_1 \\ \text{leg 1} &= \left(\frac{\boldsymbol{a}_1 \cdot \boldsymbol{q}_0}{\boldsymbol{q}_0 \cdot \boldsymbol{q}_0} \right) \boldsymbol{q}_0 = \_\_\_\_\_ \\ \text{leg 2} &= \_\_\_\_\_ - \_\_\_\_\_. \end{aligned}$$

d. Compute the vector for leg 2 and then normalize it to turn it into a unit vector. Call this vector $\boldsymbol{q}_1$ and put it in the second column of Q.

e. Verify that the columns of Q are now orthogonal and are both unit vectors.
f. The matrix R is supposed to complete the matrix factorization $A = QR$. We have built Q as an orthonormal matrix. How can we use this fact to solve for the matrix R?
g. You should now have an orthonormal matrix Q and an upper triangular matrix R. Verify that $A = QR$.
h. An alternate way to build the R matrix is to observe that

$$R = \begin{pmatrix} \boldsymbol{a}_0 \cdot \boldsymbol{q}_0 & \boldsymbol{a}_1 \cdot \boldsymbol{q}_0 \\ 0 & \boldsymbol{a}_1 \cdot \boldsymbol{q}_1 \end{pmatrix}.$$

Show that this is indeed true for the matrix A from this problem.

Exercise 4.42. Keeping track of all of the arithmetic in the QR factorization process is quite challenging, so let's leverage Python to do some of the work for us. The following block of code walks through the previous exercise without any looping (that way we can see every step transparently). Some of the code is missing so you'll need to fill it in.

```
import numpy as np
# Define the matrix $A$
A = np.matrix([[3,1],[4,1]])
n = A.shape[0]
# Build the vectors a0 and a1
a0 = A[??? , ???] # ... write code to get column 0 from A
a1 = A[??? , ???] # ... write code to get column 1 from A
# Set up storage for Q
Q = np.matrix( np.zeros( (n,n) ) )
# build the vector q0 by normalizing a0
q0 = a0 / np.linalg.norm(a0)
# Put q0 as the first column of Q
Q[:,0] = q0
# Calculate the lengths of the two legs of the triangle
leg1 = # write code to get the vector for leg 1 of the triangle
leg2 = # write code to get the vector for leg 2 of the triangle
# normalize leg2 and call it q1
q1 = # write code to normalize leg2
Q[:,1] = q1 # What does this line do?
R = # ... build the R matrix out of A and Q

print("The Q matrix is \n",Q,"\n")
print("The R matrix is \n",R,"\n")
print("The A matrix is \n",A,"\n")
print("The product QR is\n",Q*R)
```

Exercise 4.43. You should notice that the code in the previous exercise does not depend on the specific matrix A that we used? Put in a different 2×2 matrix and verify that the process still works. That is, verify that Q is orthonormal, R is upper triangular, and $A = QR$. Be sure, however, that your matrix A is full rank.

Exercise 4.44. Draw two generic vectors in $\mathbb{R}^2$ and demonstrate the process outlined in the previous problem to build the vectors for the Q matrix starting from your generic vectors.

Exercise 4.45. Now we'll extend the process from the previous exercises to three dimensions. This time we will seek a matrix Q that has three othonormal vectors starting from the three original columns of a 3×3 matrix A. Perform each of the following steps **by hand** on the matrix

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}.$$

The steps are shown graphically in Figure 4.2. In the end you should end up with an orthonormal matrix Q and an upper triangular matrix R.

- **Step 1**: Pick column $\boldsymbol{a}_0$ from the matrix A and normalize it. Call this new vector $\boldsymbol{q}_0$ and make that the first column of the matrix Q.
- **Step 2**: Project column $\boldsymbol{a}_1$ of A onto $\boldsymbol{q}_0$. This forms a right triangle with $\boldsymbol{a}_1$ as the hypotenuse, the projection of $\boldsymbol{a}_1$ onto $\boldsymbol{q}_0$ as one of the legs, and the vector difference between these two as the second leg. Notice that the second leg of the newly formed right triangle is perpendicular to $\boldsymbol{q}_0$ by design. If we normalize this vector then we have the second column of Q, $\boldsymbol{q}_1$.
- **Step 3:** Now we need a vector that is perpendicular to both $\boldsymbol{q}_0$ AND $\boldsymbol{q}_1$. To achieve this we are going to project column $\boldsymbol{a}_2$ from A onto the plane formed by $\boldsymbol{q}_0$ and $\boldsymbol{q}_1$. We'll do this in two steps:
  - **Step 3a**: We first project $\boldsymbol{a}_2$ down onto both $\boldsymbol{q}_0$ and $\boldsymbol{q}_1$.
  - **Step 3b**: The vector that is perpendicular to both $\boldsymbol{q}_0$ and $\boldsymbol{q}_1$ will be the difference between $\boldsymbol{a}_2$ the projection of $\boldsymbol{a}_2$ onto $\boldsymbol{q}_0$ and the projection of $\boldsymbol{a}_2$ onto $\boldsymbol{q}_1$. That is, we form the vector $\boldsymbol{w} = \boldsymbol{a}_2 - (\boldsymbol{a}_2 \cdot \boldsymbol{q}_0)\boldsymbol{q}_0 - (\boldsymbol{a}_2 \cdot \boldsymbol{q}_1)\boldsymbol{q}_1$. Normalizing this vector will give us $\boldsymbol{q}_2$. (Stop now and prove that $\boldsymbol{q}_2$ is indeed perpendicular to both $\boldsymbol{q}_1$ and $\boldsymbol{q}_0$.)

The result should be the matrix Q which contains orthonormal columns. To build the matrix R we simply recall that $A = QR$ and $Q^{-1} = Q^T$ so $R = Q^T A$.

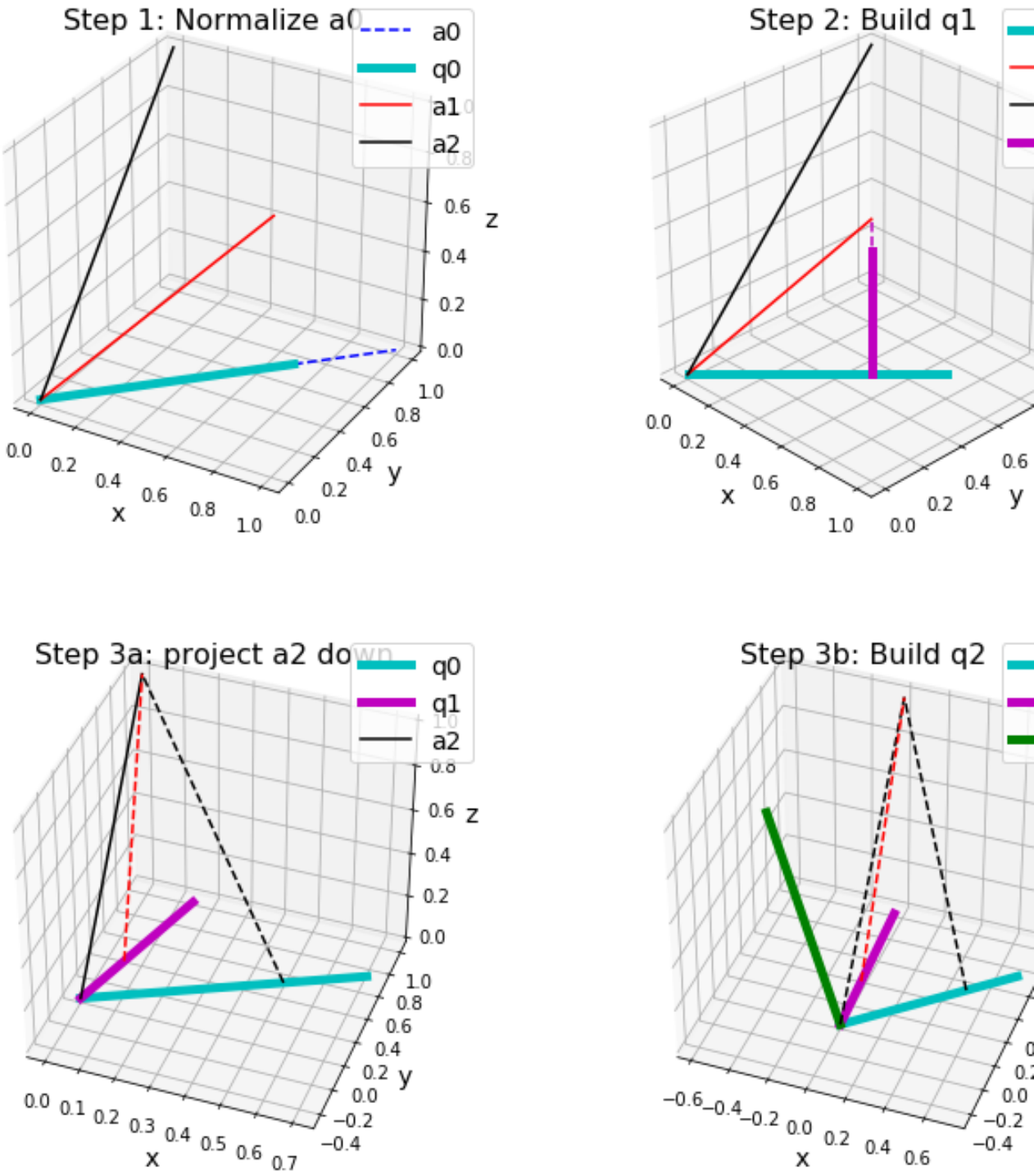


Figure 4.2: The steps to building an orthonormal basis starting with 3 vectors.

Exercise 4.46. Repeat the previous exercise but write code for each step so that Python can handle all of the computations. Again use the matrix

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}.$$

Example 4.7 (QR for $n = 3$)**.** For the sake of clarity let's now write down the full QR factorization for a 3×3 matrix.

If the columns of A are $\boldsymbol{a}_0$, $\boldsymbol{a}_1$, and $\boldsymbol{a}_2$ then

$$\boldsymbol{q}_0 = \frac{\boldsymbol{a}_0}{\|\boldsymbol{a}_0\|}$$

$$\boldsymbol{q}_1 = \frac{\boldsymbol{a}_1 - (\boldsymbol{a}_1 \cdot \boldsymbol{q}_0)\,\boldsymbol{q}_0}{\|\boldsymbol{a}_1 - (\boldsymbol{a}_1 \cdot \boldsymbol{q}_0)\,\boldsymbol{q}_0\|}$$

$$\boldsymbol{q}_2 = \frac{\boldsymbol{a}_2 - (\boldsymbol{a}_2 \cdot \boldsymbol{q}_0)\,\boldsymbol{q}_0 - (\boldsymbol{a}_2 \cdot \boldsymbol{q}_1)\,\boldsymbol{q}_1}{\|\boldsymbol{a}_2 - (\boldsymbol{a}_2 \cdot \boldsymbol{q}_0)\,\boldsymbol{q}_0 - (\boldsymbol{a}_2 \cdot \boldsymbol{q}_1)\,\boldsymbol{q}_1\|}$$

and

$$R = \begin{pmatrix} \boldsymbol{a}_0 \cdot \boldsymbol{q}_0 & \boldsymbol{a}_1 \cdot \boldsymbol{q}_0 & \boldsymbol{a}_2 \cdot \boldsymbol{q}_0 \\ 0 & \boldsymbol{a}_1 \cdot \boldsymbol{q}_1 & \boldsymbol{a}_2 \cdot \boldsymbol{q}_1 \\ 0 & 0 & \boldsymbol{a}_2 \cdot \boldsymbol{q}_2 \end{pmatrix}$$

Exercise 4.47 (The QR Factorization)**.** Now we're ready to build general code for the QR factorization. The following Python function definition is partially complete. Fill in the missing pieces of code and then test your code on square matrices of many different sizes. The easiest way to check if you have an error is to find the normed difference between A and QR with `np.linalg.norm(A - Q*R)`.

```
import numpy as np
def myQR(A):
    n = A.shape[0]
    Q = np.matrix( np.zeros( (n,n) ) )
    for j in range( ??? ): # The outer loop goes over the columns
        q = A[:,j]
        # The next loop is meant to do all of the projections.
        # When do you start the inner loop and how far do you go?
        # Hint: You don't need to enter this loop the first time through
        for i in range( ??? ):
            length_of_leg = np.sum(A[:,j].T * Q[:,i])
            q = q -  ??? * ??? # This is where the projects come in.
        Q[:,j] = q / np.linalg.norm(q)
    R = # finally build the R matrix
    return Q, R
```

```
# Test Code
A = np.matrix( ... ) # or you can use np.random.randn() for random matrices
Q, R = myQR(A)
error = np.linalg.norm(A - Q*R)
print(error)
```

We now have a robust algorithm for doing QR factorization of square matrices we can finally return to solving systems of equations.

Theorem 4.2 (Solving Systems with QR)**.** *Remember that we want to solve $A\boldsymbol{x} = \boldsymbol{b}$ and since $A = QR$ we can rewrite it with $QR\boldsymbol{x} = \boldsymbol{b}$. Since we know that Q is orthonormal by design we can multiply both sides of the equation by Q^T to get $R\boldsymbol{x} = Q^T\boldsymbol{b}$. Finally, since R is upper triangular we can use our `usolve` code from the previous section to solve the resulting triangular system.*

Exercise 4.48. Solve the system of equations

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}$$

by first computing the QR factorization of A and then solving the resulting upper triangular system.

Exercise 4.49. Write code that builds a random $n \times n$ matrix and a random $n \times 1$ vector. Solve the equation $A\boldsymbol{x} = \boldsymbol{b}$ using the QR factorization and compare the answer to what we find from `np.linalg.solve()`. Do this many times for various values of n and create a plot with n on the horizontal axis and the normed error between Python's answer and your answer from the QR algorithm on the vertical axis. It would be wise to use a `plt.semilogy()` plot. To find the normed difference you should use `np.linalg.norm()`. What do you notice?

4.6 Over Determined Systems and Curve Fitting

Exercise 4.50. In Exercise 3.81 we considered finding the quadratic function $f(x) = ax^2 + bx + c$ that *best fits* the points

$$(0, 1.07), (1, 3.9), (2, 14.8), (3, 26.8).$$

Back in Exercise 3.81 and the subsequent problems we approached this problem using an optimization tool in Python. You might be surprised to learn that there is a way to do this same optimization with linear algebra!!

We don't know the values of a, b, or c but we do have four different (x, y) ordered pairs. Hence, we have four equations:

$$1.07 = a(0)^2 + b(0) + c$$

$$3.9 = a(1)^2 + b(1) + c$$

$$14.8 = a(2)^2 + b(2) + c$$

$$26.8 = a(3)^2 + b(3) + c.$$

There are four equations and only three unknowns. This is what is called an **over determined systems** – when there are more equations than unknowns. Let's play with this problem.

a. First turn the system of equations into a matrix equation.

$$\begin{pmatrix} 0 & 0 & 1 \\ \_\_ & \_\_ & \_\_ \\ \_\_ & \_\_ & \_\_ \\ \_\_ & \_\_ & \_\_ \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 1.07 \\ 3.9 \\ 14.8 \\ 26.8 \end{pmatrix}.$$

b. None of our techniques for solving systems will likely work here since it is highly unlikely that the vector on the right-hand side of the equation is in the column space of the coefficient matrix. Discuss this.

c. One solution to the unfortunate fact from part (b) is that we can project the vector on the right-hand side into the subspace spanned by the columns of the coefficient matrix. Think of this as casting the shadow of the right-hand vector down onto the space spanned by the columns. If we do this projection we will be able to solve the equation for the values of a, b, and c that will create the projection exactly – and hence be as close as we can get to the actual right-hand side. Draw a picture of what we've said here.

d. Now we need to project the right-hand side, call it $\boldsymbol{b}$, onto the column space of the the coefficient matrix A. Recall the following facts:
   - Projections are dot products
   - Matrix multiplication is nothing but a bunch of dot products.
   - The projections of $\boldsymbol{b}$ onto the columns of A are the dot products of $\boldsymbol{b}$ with each of the columns of A.
   - What matrix can we multiply both sides of the equation $A\boldsymbol{x} = \boldsymbol{b}$ by in order for the right-hand side to become the projection that we want? (Now do the projection in Python)

e. If you have done part (d) correctly then you should now have a square system (i.e. the matrix on the left-hand side should now be square). Solve this system for a, b, and c. Compare your answers to what you found way back in Exercise 3.81.

Theorem 4.3 (Solving Overdetermined Systems)**.** *If $A\boldsymbol{x} = \boldsymbol{b}$ is an overdetermined system (i.e. A has more rows than columns) then we first multiply both sides of the equation by A^T (why do we do this?) and then solve square the $(A^TA)\boldsymbol{x} = A^T\boldsymbol{b}$ using any system solving technique. The answer to this new system is interpreted as the vector $\boldsymbol{x}$ which solves exactly for the projection of $\boldsymbol{b}$ onto the column space of A.*

The equation $(A^TA)\boldsymbol{x} = A^T\boldsymbol{b}$ is called ***the normal equations*** *and arises often in Statistics and Machine Learning.*

Exercise 4.51. Fit a linear function to the following data. Solve for the slope and intercept using the technique outlined in Theorem 4.3. Make a plot of the points along with your best fit curve.

x	y
0	4.6
1	11
2	12
3	19.1
4	18.8
5	39.5
6	31.1
7	43.4
8	40.3
9	41.5
10	41.6

Code to download the data directly is given below.

```
import numpy as np
import pandas as pd
data = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSullivan/Numerica
```

Exercise 4.52. Fit a quadratic function to the following data using the technique outlined in Theorem 4.3. Make a plot of the points along with your best fit curve.

x	y
0	-6.8
1	11.8
2	50.6
3	94
4	224.3
5	301.7

x	y
6	499.2
7	454.7
8	578.5
9	1102
10	1203.2

Code to download the data directly is given below.

```
import numpy as np
import pandas as pd
data = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSulliva
```

Exercise 4.53. The Statistical technique of curve fitting is often called "linear regression". This even holds when we are fitting quadratic functions, cubic functions, etc to the data ... we still call that linear regression! Why?

This section of the text on solving over determined systems is just a bit of a teaser for a bit of higher-level statistics, data science, and machine learning. The normal equations and solving systems via projections is the starting point of many modern machine learning algorithms. For more information on this sort of problem look into taking some statistics, data science, and/or machine learning courses. You'll love it!

4.7 The Eigenvalue-Eigenvector Problem

We finally turn our attention to the last major topic in numerical linear algebra in this course.[4]

Definition 4.8 (The Eigenvalue Problem)**.** Recall that the eigenvectors, $\boldsymbol{x}$, and the eigenvalues, λ of a square matrix satisfy the equation $A\boldsymbol{x} = \lambda\boldsymbol{x}$. Geometrically, the eign-problem is the task of finding the special vectors $\boldsymbol{x}$ such that multiplication by the matrix A only produces a scalar multiple of $\boldsymbol{x}$.

Thinking about matrix multiplication, the geometric notion of the eigenvalue problem is rather peculiar since matrix-vector multiplication usually results in a scaling and a rotation of the vector $\boldsymbol{x}$. Therefore, in some sense the

[4] Numerical Linear Algebra is a huge field and there is way more to say ... but alas, this is an introductory course in numerical methods so we can't do everything. Sigh.

eigenvectors are the only special vectors which avoid geometric rotation under matrix multiplication. For a graphical exploration of this idea see: https://www.geogebra.org/m/JP2XZpzV.

Theorem 4.4. *Recall that to solve the eigen-problem for a square matrix A we complete the following steps:*

a. *First rearrange the definition of the eigenvalue-eigenvector pair to*

$$(A\boldsymbol{x} - \lambda\boldsymbol{x}) = \mathbf{0}.$$

b. *Next, factor the $\boldsymbol{x}$ on the right to get*

$$(A - \lambda I)\boldsymbol{x} = \mathbf{0}.$$

c. *Now observe that since $\boldsymbol{x} \neq 0$ the matrix $A - \lambda I$ must NOT have an inverse. Therefore,*

$$\det(A - \lambda I) = 0.$$

d. *Solve the equation $\det(A - \lambda I) = 0$ for all of the values of λ.*
e. *For each λ, find a solution to the equation $(A - \lambda I)\boldsymbol{x} = \mathbf{0}$. Note that there will be infinitely many solutions so you will need to make wise choices for the free variables.*

Exercise 4.54. Find the eigenvalues and eigenvectors of

$$A = \begin{pmatrix} 1 & 2 \\ 4 & 3 \end{pmatrix}.$$

Exercise 4.55. In the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

one of the eigenvalues is $\lambda_1 = 0$.

a. What does that tell us about the matrix A?
b. What is the eigenvector $\boldsymbol{v}_1$ associated with $\lambda_1 = 0$?
c. What is the null space of the matrix A?

OK. Now that you recall some of the basics let's play with a little limit problem. The following exercises are going to work us toward the **power method** for finding certain eigen-structure of a matrix.

Exercise 4.56. Consider the matrix

$$A = \begin{pmatrix} 8 & 5 & -6 \\ -12 & -9 & 12 \\ -3 & -3 & 5 \end{pmatrix}.$$

This matrix has the following eigen-structure:

$$\boldsymbol{v}_1 = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} \quad \text{with} \quad \lambda_1 = 3$$

$$\boldsymbol{v}_2 = \begin{pmatrix} 2 \\ 0 \\ 2 \end{pmatrix} \quad \text{with} \quad \lambda_2 = 2$$

$$\boldsymbol{v}_3 = \begin{pmatrix} -1 \\ 3 \\ 1 \end{pmatrix} \quad \text{with} \quad \lambda_3 = -1$$

If we have

$$\boldsymbol{x} = -2\boldsymbol{v}_1 + 1\boldsymbol{v}_2 - 3\boldsymbol{v}_3 = \begin{pmatrix} 3 \\ -7 \\ -1 \end{pmatrix}$$

then we want to do a bit of an experiment. What happens when we iteratively multiply $\boldsymbol{x}$ by A but at the same time divide by the largest eigenvalue. Let's see:

- What is $A^1\boldsymbol{x}/3^1$?
- What is $A^2\boldsymbol{x}/3^2$?
- What is $A^3\boldsymbol{x}/3^3$?
- What is $A^4\boldsymbol{x}/3^4$?
- ...

It might be nice now to go to some Python code to do the computations (if you haven't already). Use your code to conjecture about the following limit.

$$\lim_{k\to\infty} \frac{A^k\boldsymbol{x}}{\lambda_{max}^k} = ???.$$

In this limit we are really interested in the direction of the resulting vector, not the magnitude. Therefore, in the code below you will see that we normalize the resulting vector so that it is a unit vector.

Note: be careful, computers don't do infinity, so for powers that are too large you won't get any results.

```
import numpy as np
A = np.matrix([[8,5,-6],[-12,-9,12],[-3,-3,5]])
x = np.matrix([[3],[-7],[-1]])
```

```
eigval_max = 3

k = 4
result = A**k * x / eigval_max**k
print(result / np.linalg.norm(result) )
```

Exercise 4.57. If a matrix A has eigenvectors $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, $\boldsymbol{v}_3$, $\cdots$, $\boldsymbol{v}_n$ with eigenvalues $\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_n$ and $\boldsymbol{x}$ is in the column space of A then what will we get, approximately, if we evaluate $A^k\boldsymbol{x}/\max_j(\lambda_j)^k$ for very large values of k?

Discuss your conjecture with your peers. Then try to verify it with several numerical examples.

Exercise 4.58. Explain your result from the previous exercise geometrically.

Exercise 4.59. The algorithm that we've been toying with will find the dominant eigenvector of a matrix fairly quickly. Why might you be only interested in the dominant eigenvector of a matrix? Discuss.

Exercise 4.60. In this problem we will formally prove the conjecture that you just made. This conjecture will lead us to the **power method** for finding the dominant eigenvector and eigenvalue of a matrix.

a. Assume that A has n linearly independent eigenvectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ and choose $\boldsymbol{x} = \sum_{j=1}^{n} c_j\boldsymbol{v}_j$. You have proved in the past that

$$A^k\boldsymbol{x} = c_1\lambda_1^k\boldsymbol{v}_1 + c_2\lambda_2^k\boldsymbol{v}_2 + \cdots c_n\lambda_n^k\boldsymbol{v}_n.$$

Stop and sketch out the details of this proof now.

b. If we factor λ_1^k out of the right-hand side we get

$$A^k\boldsymbol{x} = \lambda_1^k\left(c_1??? + c_2\left(\frac{???}{???}\right)^k\boldsymbol{v}_2 + c_3\left(\frac{???}{???}\right)^k\boldsymbol{v}_3 + \cdots + c_n\left(\frac{???}{???}\right)^k\boldsymbol{v}_n\right)$$

(fill in the question marks)

c. If $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|$ then what happens to each of the $(\lambda_j/\lambda_1)^k$ terms as $k \to \infty$?

d. Using your answer to part (c), what is $\lim_{k\to\infty} A^k\boldsymbol{x}/\lambda_1^k$?

Theorem 4.5 (The Power Method)**.** *The following algorithm, called* ***the power method*** *will quickly find the eigenvalue of largest absolute value for a square matrix $A \in \mathbb{R}^{n\times n}$ as well as the associated (normalized) eigenvector. We are assuming that there are n linearly independent eigenvectors of A.*

Step #1: *Given a nonzero vector $\boldsymbol{x}$, set $\boldsymbol{v}^{(1)} = \boldsymbol{x}/\|\boldsymbol{x}\|$. (Here the superscript indiates the iteration number) Note that the initial vector $\boldsymbol{x}$ is pretty irrelevant to the process so it can just be a random vector of the correct size..*

Step #2: *For $k = 2, 3, \ldots$*

Step #2a: *Compute $\tilde{\boldsymbol{v}}^{(k)} = A\boldsymbol{v}^{(k-1)}$ (this gives a non-normalized version of the next estimate of the dominant eigenvector.)*

Step #2b: *Set $\lambda^{(k)} = \tilde{\boldsymbol{v}}^{(k)} \cdot \boldsymbol{v}^{(k-1)}$. (this gives an approximation of the eigenvalue since if $\boldsymbol{v}^{(k-1)}$ was the actual eigenvector we would have $\lambda = A\boldsymbol{v}^{(k-1)} \cdot \boldsymbol{v}^{(k-1)}$. Stop now and explain this.)*

Step #2c: *Normalize $\tilde{\boldsymbol{v}}^{(k)}$ by computing $\boldsymbol{v}^{(k)} = \tilde{\boldsymbol{v}}^{(k)}/\|\tilde{\boldsymbol{v}}^{(k)}\|$. (This guarantees that you will be sending a unit vector into the next iteration of the loop)*

Exercise 4.61. Go through Theorem 4.5 carefully and describe what we need to do in each step and why we're doing it. Then complete all of the missing pieces of the following Python function.

```
import numpy as np
def myPower(A, tol = 1e-8):
    n = A.shape[0]
    x = np.matrix( np.random.randn(n,1) )
    v = # turn x into a unit vector

    # The variable "difference" will initialize the normed
    # difference between two successive approximations of
    # the dominant eigenvector.  We start the value to be
    # large so that our code enters the while loop.
    difference = 1000
    counter = 0 # keep track of how many steps we've taken
    while difference > tol and counter < 1000:
        vnew = # Calcuate the new approximation of v
        L = ( np.transpose(vnew) * v).item() # approx of the eigenvalue
        vnew = # normalized vnew to make it a unit vector
        # use np.linalg.norm() to determine how close v and vnew are
        difference = ???
        counter += 1 # increment the counter
        v = vnew # set v to vnew so we can start again.
    return v, L
```

Exercise 4.62. Test your `myPower()` function on several matrices where you know the eigenstructure. Then try the `myPower()` function on larger random matrices. You can check that it is working using `np.linalg.eig()` (be sure to

normalize the vectors in the same way so you can compare them.)

Exercise 4.63. The check that we built into our code to stop the `while` loop in our `myPower()` function will fail if $\lambda_1 < 0$. Why? What happens to the approximation of the dominant eigenvector throughout the iterations when $\lambda_1 < 1$. Modify your code to account for this issue so that it will still stop appropriately.

Exercise 4.64. What happens in the power method iterations when λ_1 is complex. The maximum eigenvalue can certainly be complex if $|\lambda_1|$ (the modulus of the complex number) is larger than all of the other eigenvalues. It may be helpful to build a matrix specifically with complex eigenvalues.[5]

Exercise 4.65 (Convergence Rate of the Power Method)**.** The proof that the power method will work hinges on the fact that $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|$. In Exercise 4.60 we proved that the limit

$$\lim_{k\to\infty} \frac{A^k \boldsymbol{x}}{\lambda_1^k}$$

converges to the dominant eigenvector, but how fast is the convergence? What does the speed of the convergence depend on?

Take note that since we're assuming that the eigenvalues are ordered, the ratio λ_2/λ_1 will be larger than λ_j/λ_1 for all $j > 2$. Hence, the speed at which the power method converges depends mostly on the ratio λ_2/λ_1. Let's build a numerical experiment to see how sensitive the power method is to this ratio.

Build a 4×4 matrix A with dominant eigenvalue $\lambda_1 = 1$ and all other eigenvalues less than 1 in absolute value. Then choose several values of λ_2 and build an experiment to determine the number of iterations that it takes for the power method to converge to within a pre-determined tolerance to the dominant eigenvector. In the end you should produce a plot with the ratio λ_2/λ_1 on the horizontal axis and the number of iterations to converge to a fixed tolerance on the vertical axis. Discuss what you see in your plot.

Hint: To build a matrix with specific eigen-structure use the matrix factorization $A = PDP^{-1}$ where the columns of P contain the eigenvectors of A and the diagonal of D contains the eigenvalues. In this case the P matrix can be random but you need to control the D matrix. Moreover, remember that λ_3 and λ_4 should be smaller than λ_2.

---

[5] To build a matrix with specific eigenvalues it may be helpful to recall the matrix factorization $A = PDP^{-1}$ where the columns of P are the eigenvectors of A and the diagonal entries of D are the eigenvalues. If you choose P and D then you can build A with your specific eigen-structure. If you are looking for complex eigenvalues then remember that the eigenvectors may well be complex too.

4.8 Exercises

4.8.1 Algorithm Summaries

Exercise 4.66. Explain in clear language how to efficiently solve an upper triangular system of linear equations.

Exercise 4.67. Explain in clear language how to efficiently solve a lower triangular system of linear equations.

Exercise 4.68. Explain in clear language how to solve the equation $A\boldsymbol{x} = \boldsymbol{b}$ using an LU decomposition.

Exercise 4.69. Explain in clear language how to solve an overdetermined system of linear equations (more equations than unknowns) numerically.

Exercise 4.70. Explain in clear language the algorithm for finding the columns of the Q matrix in the QR factorization. Give all of the mathematical details.

Exercise 4.71. Explain in clear language how to find the upper triangular matrix R in the QR factorization. Give all of the mathematical details.

Exercise 4.72. Explain in clear language how to solve the equation $A\boldsymbol{x} = \boldsymbol{b}$ using a QR decomposition.

Exercise 4.73. Explain in clear language how the power method works to find the dominant eigenvalue and eigenvector of a square matrix. Give all of the mathematical details.

4.8.2 Applying What You've Learned

Exercise 4.74. As mentioned much earlier in this chapter, there is an `rref()` command in Python, but it is in the `sympy` library instead of the `numpy` library – it is implemented as a symbolic computation instead of a numerical computation. OK. So what? In this problem we want to compare the time to solve a system of equations $A\boldsymbol{x} = \boldsymbol{b}$ with each of the following techniques:

- row reduction of an augmented matrix $\left(A \quad | \quad b\right)$ with `sympy`,
- our implementation of the LU decomposition,
- our implementation of the QR decomposition, and

- the `numpy.linalg.solve()` command.

To time code in Python first import the `time` library. Then use `start = time.time()` at the start of your code and `stop = time.time()` and the end of your code. The difference between `stop` and `start` is the elapsed computation time.

Make observations about how the algorithms perform for different sized matrices. You can use random matrices and vectors for A and $\boldsymbol{b}$.

The code below will compute the reduced row echelon form of a matrix (RREF). Implement the code so that you know how it works.

```
import sympy as sp
import numpy as np
# in this problem it will be easiest to start with numpy matrices
A = np.matrix([[1, 0, 1], [2, 3, 5], [-1, -3, -3]])
b = np.matrix([[3],[7],[3]])
Augmented = np.c_[A,b] # augment b onto the right hand side of A

Msymbolic = sp.Matrix(Augmented)
MsymbolicRREF = Msymbolic.rref()
print(MsymbolicRREF)
```

To time code you can use code like the following.

```
import time
start = time.time()
# some code that you want to time
stop =  time.time()
total_time = stop - start
print("Total computation time=",total_time)
```

Exercise 4.75. Imagine that we have a 1 meter long thin metal rod that has been heated to 100° on the left-hand side and cooled to 0° on the right-hand side. We want to know the temperature every 10 cm from left to right on the rod.

a. First we break the rod into equal 10cm increments as shown. See Figure 4.3. How many unknowns are there in this picture?

b. The temperature at each point along the rod is the average of the temperatures at the adjacent points. For example, if we let T_1 be the temperature at point x_1 then

$$T_1 = \frac{T_0 + T_2}{2}.$$

Write a system of equations for each of the unknown temperatures.

c. Solve the system for the temperature at each unknown node using either LU or QR decomposition.

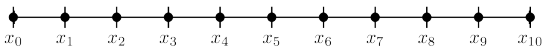


Figure 4.3: A rod to be heated broken into 10 equal-length segments.

Exercise 4.76. Write code to solve the following systems of equations via both LU and QR decompositions. If the algorithm fails then be sure to explain exactly why.

a.
$$\begin{aligned} x + 2y + 3z &= 4 \\ 2x + 4y + 3z &= 5 \\ x + y &= 4 \end{aligned}$$

b.
$$\begin{aligned} 2y + 3z &= 4 \\ 2x + 3z &= 5 \\ y &= 4 \end{aligned}$$

c.
$$\begin{aligned} 2y + 3z &= 4 \\ 2x + 4y + 3z &= 5 \\ x + y &= 4 \end{aligned}$$

Exercise 4.77. Give a specific example of a nonzero matrix which will NOT have an LU decomposition. Give specific reasons why LU will fail on your matrix.

Exercise 4.78. Give a specific example of a nonzero matrix which will NOT have an QR decomposition. Give specific reasons why QR will fail on your matrix.

Exercise 4.79. Have you ever wondered how scientific software computes a determinant? The formula that you learned for calculating determinants by hand is horribly cumbersome and computationally intractible for large matrices. This problem is meant to give you glimpse of what is *actually* going on under the hood.[6]

If A has an LU decomposition then $A = LU$. Use properties that you know about determinants to come up with a simple way to find the determinant for

[6]Actually, the determinant computation uses LU with partial pivoting which we did not cover here in the text. What we are looking at in this exercise is a smaller subcase of what happens when you have a matrix A that does not require any row swaps in the row reduction process.

matrices that have an LU decomposition. Show all of your work in developing your formula.

Once you have your formula for calculating $\det(A)$, write a Python function that accepts a matrix, produces the LU decomposition, and returns the determinant of A. Check your work against Python's `np.linalg.det()` function.

Exercise 4.80. For this problem we are going to run a numerical experiment to see how the process of solving the equation $A\boldsymbol{x} = \boldsymbol{b}$ using the LU factorization performs on random coefficient matrices A and random right-hand sides $\boldsymbol{b}$. We will compare against Python's algorithm for solving linear systems.

We will do the following:

Create a loop that does the following:

a. Loop over the size of the matrix n.
b. Build a random matrix A of size $n \times n$. You can do this with the code `A = np.matrix( np.random.randn(n,n) )`
c. Build a random vector $\boldsymbol{b}$ in $\mathbb{R}^n$. You can do this with the code `b = np.matrix( np.random.randn(n,1) )`
d. Find Python's answer to the problem $A\boldsymbol{x} = \boldsymbol{b}$ =0 using the command `exact = np.linalg.solve(A,b)`
e. Write code that uses your three LU functions (`myLU`, `lsolve`, `usolve`) to find a solution to the equation $A\boldsymbol{x} = \boldsymbol{b}$.
f. Find the error between your answer and the exact answer using the code `np.linalg.norm(x - exact)`
g. Make a plot (`plt.semilogy()`) that shows how the error behaves as the size of the problem changes. You should run this for matrices of larger and larger size but be warned that the loop will run for quite a long time if you go above 300×300 matrices. Just be patient.

Conclusions: What do you notice in your final plot. What does this tell you about the behavior of our LU decomposition code?

Exercise 4.81. Repeat Exercise 4.80 for the QR decomposition. Your final plot should show both the behavior of QR and of LU throughout the experiement. What do you notice?

Exercise 4.82. Find a least squares solution to the equation $A\boldsymbol{x} = \boldsymbol{b}$ in two different ways with

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 4 & -2 & 6 \\ 4 & 7 & 8 \\ 3 & 7 & 19 \end{pmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{pmatrix} 5 \\ 2 \\ -2 \\ 8 \end{pmatrix}.$$

Exercise 4.83. Let A be defined as

$$A = \begin{pmatrix} 10^{-15} & 1 \\ 1 & 1 \end{pmatrix}$$

and let $\boldsymbol{b}$ be the vector

$$\boldsymbol{b} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}.$$

Notice that A has a tiny, but nonzero, value in the first entry.

a. Solve the linear system $A\boldsymbol{x} = \boldsymbol{b}$ by hand.

b. Use your `myLU`, `lsolve`, and `usolve` functions to solve this problem using the LU decomposition method.

c. Compare your answers to parts (a) and (b). What went wrong?

Exercise 4.84 (Hilbert Matrices)**.** A Hilbert Matrix is a matrix of the form $H_{ij} = 1/(i + j + 1)$ where both i and j both start indexed at 0. For example, a 4×4 Hilbert Matrix is

$$H = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{pmatrix}.$$

This type of matrix is often used to test numerical linear algebra algorithms since it is known to have some *odd* behaviors ... which you'll see in a moment.

a. Write code to build a $n \times n$ Hilbert Matrix and call this matrix H. Test your code for various values of n to be sure that it is building the correct matrices.
b. Build a vector of ones called $\boldsymbol{b}$ with code `b = np.ones( (n,1) )`. We will use $\boldsymbol{b}$ as the right hand side of the system of equations $H\boldsymbol{x} = \boldsymbol{b}$.
c. Solve the system of equations $H\boldsymbol{x} = \boldsymbol{b}$ using any technique you like from this chapter.
d. Now let's say that you change the first entry of $\boldsymbol{b}$ by just a little bit, say 10^(-15). If we were to now solve the equation $H\boldsymbol{x}_{new} = \boldsymbol{b}_{new}$ what would you expect as compared to solving $H\boldsymbol{x} = \boldsymbol{b}$.
e. Now let's actually make the change suggested in part (d). Use the code `bnew = np.ones( (n,1) )` and then `bnew[0] = bnew[0] + 1e-15` to build a new $\boldsymbol{b}$ vector with this small change. Solve $H\boldsymbol{x} = \boldsymbol{b}$ and $H\boldsymbol{x}_{new} = \boldsymbol{b}_{new}$ and then compare the maximum absolute difference `np.max( np.abs( x - xnew ) )`. What do you notice? Make a plot with n on the horizontal axis and the maximum absolute difference on the vertical axis. What does this plot tell you about the solution to the equation $H\boldsymbol{x} = \boldsymbol{b}$?
f. We know that HH^{-1} should be the identity matrix. As we'll see, however, Hilbert matrices are particularly poorly behaved! Write a loop over n

that (i) builds a Hilbert matrix of size n, (ii) calculates $H * H^{-1}$ (using `np.linalg.inv()` to compute the inverse directly), (iii) calculates the norm of the difference between the identity matrix (`np.identity(n)`) and your calculated identity matrix from part (ii). Finally. Build a plot that shows n on the horizontal axis and the normed difference on the vertical axis. What do you see? What does this mean about the matrix inversion of the Hilbert matrix.

g. There are cautionary tales hiding in this problem. Write a paragraph explaining what you can learn by playing with pathological matrices like the Hilbert Matrix.

Exercise 4.85. Now that you have QR and LU code we're going to use both of them! The problem is as follows:
We are going to find the polynomial of degree 4 that best fits the function $

$$y = \cos(4t) + 0.1\varepsilon(t)$$

at 50 equally spaced points t between 0 and 1. Here we are using $\varepsilon(t)$ as a function that outputs normally distributed random white noise. In Python you will build y as `y = np.cos(4*t) + 0.1*np.random.randn(t.shape[0])`

Build the t vector and the y vector (these are your data). We need to set up the least squares problems $A\boldsymbol{x} = \boldsymbol{b}$ by setting up the matrix A as we did in the other least squares curve fitting problems and by setting up the $\boldsymbol{b}$ vector using the y data you just built. Solve the problem of finding the coefficients of the best degree 4 polynomial that fits this data. Report the sum of squared error and show a plot of the data along with the best fit curve.

Exercise 4.86. Find the largest eigenvalue of the matrix A WITHOUT using any built-in Python command for finding eigenstructure.

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 0 & 1 & 2 \\ 3 & 4 & 5 & 6 \end{pmatrix}$$

Exercise 4.87. It is possible in a matrix that the eigenvalues λ_1 and λ_2 are equal but with the corresponding eigenvectors not equal. Before you experiment with matrices of this sort, write a conjecture about what will happen to the power method in this case (look back to our proof in Exercise 4.60 of how the power method works). Now build several specific matrices where this is the case and see what happens to the power method.

Exercise 4.88. Will the power method fail, slow down, or be uneffected if one (or more) of the non-dominant eigenvalues is zero? Give sufficient mathematical evidence or show several numerical experiments to support your answer.

Exercise 4.89. Find a cubic function that best fits the following data. you can download the data directly with the code below.

x Data	y Data
0	1.0220
0.0500	1.0174
0.1000	1.0428
0.1500	1.0690
0.2000	1.0505
0.2500	1.0631
0.3000	1.0458
0.3500	1.0513
0.4000	1.0199
0.4500	1.0180
0.5000	1.0156
0.5500	0.9817
0.6000	0.9652
0.6500	0.9429
0.7000	0.9393
0.7500	0.9266
0.8000	0.8959
0.8500	0.9014
0.9000	0.8990
0.9500	0.9038
1.0000	0.8989

```
import numpy as np
import pandas as pd
data = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSulliva
```

Theorem 4.6. *If A is a symmetric matrix with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ then $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$. Furthermore, the eigenvectors will be orthogonal to each other.*

Exercise 4.90 (The Deflation Method)**.** For symmetric matrices we can build an extension to the power method in order to find the second most dominant eigen-pair for a matrix A. Theorem 4.6 suggests the following method for finding the second dominant eigen-pair for a symmetric matrix. This method is called the **deflation method**.

- Use the power method to find the dominant eigenvalue and eigenvector.
- Start with a random unit vector of the correct shape.
- Multiplying your vector by A will *pull it toward* the dominant eigenvector. After you multiply, project your vector onto the dominant eigenvector and

find the projection error.
- Use the projection error as the new approximation for the eigenvector (Why should we do this? What are we really finding here?)

Note that the deflation method is really exactly the same as the power method with the exception that we orthogonalize at every step. Hence, when you write your code expect to only change a few lines from your power method.

Write a function to find the second largest eigenvalue and eigenvector pair by putting the deflation method into practice. Test your code on a matrix A and compare against Python's `np.linalg.eig()` command. Your code needs to work on symmetric matrices of arbitrary size and you need to write test code that clearly shows the error between your calculated eigenvalue and Python's eigenvalue as well as your calculated eigenvector and 's eigenvector.

To guarantee that you start with a symmetric matrix you can use the following code.

```
import numpy as np
N = 40
A = np.random.randn(N,N)
A = np.matrix(A)
A = np.transpose(A) * A # this should build a symmetric matrix (why?)
```

Exercise 4.91. (This concept for this problem is modified from [6]. The data is taken from NOAA and the National Weather Service with the specific values associated with La Crosse, WI.)

Floods in the Mississippi River Valleys of the upper midwest have somewhat predictable day-to-day behavior in that the flood stage today has high predictive power for the flood stage tomorrow. Assume that the flood stages are:

- Stage 0 (Normal): Average daily flow is below 90,000 ft^3/sec (cubic feet per second = cfs). This is the *normal* river level.
- Stage 1 (Action Level): Average daily flow is between 90,000 cfs and 124,000 cfs.
- Stage 2 (Minor Flood): Average daily flow is between 124,000 cfs and 146,000 cfs.
- Stage 3 (Moderate Flood): Average daily flow is between 146,000 cfs and 170,000 cfs.
- Stage 4 (Extreme Flood): Average daily flow is above 170,000 cfs.

The following table shows the probability of one stage transitioning into another stage from one day to the next.

	0 Today	1 Today	2 Today	3 Today	4 Today
0 Tomorrow	0.9	0.3	0	0	0
1 Tomorrow	0.05	0.7	0.4	0	0

	0 Today	1 Today	2 Today	3 Today	4 Today
2 Tomorrow	0.025	0	0.6	0.6	0
3 Tomorrow	0.015	0	0	0.4	0.8
4 Tomorrow	0.01	0	0	0	0.2

Mathematically, if $\boldsymbol{s}_k$ is the state at day k and A is the matrix given in the table above then the difference equation $\boldsymbol{s}_{k+1} = A\boldsymbol{s}_k$ shows how a state will transition from day to day. For example, if we are currently in Stage 0 then

$$\boldsymbol{s}_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

We can interpret this as "*there is a probability of 1 that we are in Stage 0 today and there is a probability of 0 that we are in any other stage today.*"

If we want to advance this model forward in time then we just need to iterate. In our example, the state tomorrow would be $\boldsymbol{s}_1 = A\boldsymbol{s}_0$. The state two days from now would be $\boldsymbol{s}_2 = A\boldsymbol{s}_1$, and if we use the expression for $\boldsymbol{s}_1$ we can simplify to $\boldsymbol{s}_2 = A^2\boldsymbol{s}_0$.

a. Prove that the state at day n is $\boldsymbol{s}_n = A^n\boldsymbol{s}_0$.
b. If n is large then the steady state solution to the difference equation in part (a) is given exactly by the power method iteration that we have studied in this chapter. Hence, as the iterations proceed they will be pulled toward the dominant eigenvector. Use the power method to find the dominant eigenvector of the matrix A.

c. The vectors in this problem are called **probability vectors** in the sense that the vectors sum to 1 and every entry can be interpreted as a probability. Re-scale your answer from part (b) so that we can interpret the entries as probabilities. That is, ensure that the sum of the vector from part (b) is 1.

d. Interpret your answer to part (c) in the context of the problem. Be sure that your interpretation could be well understood by someone that does not know the mathematics that you just did.

4.9 Projects

In this section we propose several ideas for projects related to numerical linear algebra. These projects are meant to be open ended, to encourage creative mathematics, to push your coding skills, and to require you to write and communicate your mathematics. Take the time to read Appendix B before you write your final solution.

4.9.1 The Google Page Rank Algorithm

In this project you will discover how the Page Rank algorithm works to give the most relevant information as the top hit on a Google search.

Search engines compile large indexes of the dynamic information on the Internet so they are easily searched. This means that when you do a Google search, you are not actually searching the Internet; instead, you are searching the indexes at Google.

When you type a query into Google the following two steps take place:

1. Query Module: The query module at Google converts your natural language into a language that the search system can understand and consults the various indexes at Google in order to answer the query. This is done to find the list of relevant pages.

2. Ranking Module: The ranking module takes the set of relevant pages and ranks them. The outcome of the ranking is an ordered list of web pages such that the pages near the top of the list are most likely to be what you desire from your search. This ranking is the same as assigning a *popularity score* to each web site and then listing the relevant sites by this score.

This section focuses on the Linear Algebra behind the Ranking Module developed by the founders of Google: Sergey Brin and Larry Page. Their algorithm is called the *Page Rank algorithm*, and you use it every single time you use Google's search engine.

In simple terms: *A webpage is important if it is pointed to by other important pages.*

The Internet can be viewed as a directed graph (look up this term here on Wikipedia) where the nodes are the web pages and the edges are the hyperlinks between the pages. The hyperlinks into a page are called *in links*, and the ones pointing out of a page are called *out links*. In essence, a hyperlink from my page to yours is my endorsement of your page. Thus, a page with more recommendations must be more important than a page with a few links. However, the status of the recommendation is also important.

Let us now translate this into mathematics. To help understand this we first consider the small web of six pages shown in Figure 4.4 (a graph of the router level of the internet can be found here). The links between the pages are shown by arrows. An arrow pointing into a node is an *in link* and an arrow pointing out of a node is an *out link*. In Figure 4.4, node 3 has three out links (to nodes 1, 2, and 5) and 1 in link (from node 1).

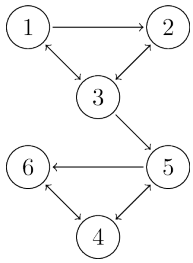


Figure 4.4: Example web graph.

We will first define some notation in the Page Rank algorithm:

- $|P_i|$ is the number of out links from page P_i
- H is the *hyperlink* matrix defined as

$$H_{ij} = \begin{cases} \frac{1}{|P_j|}, & \text{if there is a link from node } j \text{ to node } i \\ 0, & \text{otherwise} \end{cases}$$

  where the "i" and "j" are the row and column indices respectively.

- $\boldsymbol{x}$ is a vector that contains all of the Page Ranks for the individual pages.

The Page Rank algorithm works as follows:

1. Initialize the page ranks to all be equal. This means that our initial assumption is that all pages are of equal rank. In the case of Figure 4.4 we would take $\boldsymbol{x}_0$ to be

$$\boldsymbol{x}_0 = \begin{pmatrix} 1/6 \\ 1/6 \\ 1/6 \\ 1/6 \\ 1/6 \\ 1/6 \end{pmatrix}.$$

2. Build the hyperlink matrix.
   As an example we'll consider node 3 in Figure 4.4. There are three out links from node 3 (to nodes 1, 2, and 5). Hence $H_{13} = 1/3$, $H_{23} = 1/3$,

and $H_{53} = 1/3$ and the partially complete hyperlink matrix is

$$H = \begin{pmatrix} - & - & 1/3 & - & - & - \\ - & - & 1/3 & - & - & - \\ - & - & 0 & - & - & - \\ - & - & 0 & - & - & - \\ - & - & 1/3 & - & - & - \\ - & - & 0 & - & - & - \end{pmatrix}$$

3. The difference equation $\boldsymbol{x}_{n+1} = H\boldsymbol{x}_n$ is used to iteratively refine the estimates of the page ranks. You can view the iterations as a person visiting a page and then following a link at random, then following a random link on the next page, and the next, and the next, etc. Hence we see that the iterations evolve exactly as expected for a difference equation.

Iteration	New Page Rank Estimation
0	$\boldsymbol{x}_0$
1	$\boldsymbol{x}_1 = H\boldsymbol{x}_0$
2	$\boldsymbol{x}_2 = H\boldsymbol{x}_1 = H^2\boldsymbol{x}_0$
3	$\boldsymbol{x}_3 = H\boldsymbol{x}_2 = H^3\boldsymbol{x}_0$
4	$\boldsymbol{x}_4 = H\boldsymbol{x}_3 = H^4\boldsymbol{x}_0$
$\vdots$	$\vdots$
k	$\boldsymbol{x}_k = H^k\boldsymbol{x}_0$

4. When a steady state is reached we sort the resulting vector $\boldsymbol{x}_k$ to give the page rank. The node (web page) with the highest rank will be the top search result, the second highest rank will be the second search result, and so on.

It doesn't take much to see that this process can be very time consuming. Think about your typical web search with hundreds of thousands of hits; that makes a square matrix H that has a size of hundreds of thousands of entries by hundreds of thousands of entries! The matrix multiplications alone would take many minutes (or possibly many hours) for every search! ... but Brin and Page were pretty smart dudes!!

We now state a few theorems and definitions that will help us simplify the iterative Page Rank process.

Theorem 4.7. *If A is an $n \times n$ matrix with n linearly independent eigenvectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3, \ldots, \boldsymbol{v}_n$ and associated eigenvalues $\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_n$ then for any initial vector $\boldsymbol{x} \in \mathbb{R}^n$ we can write $A^k\boldsymbol{x}$ as*

$$A^k\boldsymbol{x} = c_1\lambda_1^k\boldsymbol{v}_1 + c_2\lambda_2^k\boldsymbol{v}_2 + c_3\lambda_3^k\boldsymbol{v}_3 + \cdots c_n\lambda_n^k\boldsymbol{v}_n$$

where $c_1, c_2, c_3, \ldots, c_n$ are the constants found by expressing $\boldsymbol{x}$ as a linear combination of the eigenvectors.
Note: We can assume that the eigenvalues are ordered such that $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|$.

Exercise 4.92. Prove the preceding theorem.

A **probability vector** is a vector with entries on the interval $[0, 1]$ that add up to 1.

A **stochastic matrix** is a square matrix whose columns are probability vectors.

Theorem 4.8. *If A is a stochastic $n \times n$ matrix then A will have n linearly independent eigenvectors. Furthermore, the largest eigenvalue of a stochastic matrix will be $\lambda_1 = 1$ and the smallest eigenvalue will always be nonnegative: $0 \leq |\lambda_n| < 1$.*

Some of the following tasks will ask you to *prove* a statement or a theorem. This means to clearly write all of the logical and mathematical reasons why the statement is true. Your proof should be absolutely crystal clear to anyone with a similar mathematical background . . . if you are in doubt then have a peer from a different group read your proof to you .

Exercise 4.93. Finish writing the hyperlink matrix H from Figure 4.4.

Exercise 4.94. Write code to implement the iterative process defined previously. Make a plot that shows how the rank evolves over the iterations.

Exercise 4.95. What must be true about a collection of n pages such that an $n \times n$ hyperlink matrix H is a stochastic matrix.

Exercise 4.96. The statement of the next theorem is incomplete, but the proof is given to you. Fill in the blank in the statement of the theorem and provide a few sentences supporting your answer.

Theorem 4.9. *If A is an $n \times n$ stochastic matrix and $\boldsymbol{x}_0$ is some initial vector for the difference equation $\boldsymbol{x}_{n+1} = A\boldsymbol{x}_n$, then the steady state vector is*

$$\boldsymbol{x}_{equilib} = \lim_{k \to \infty} A^k \boldsymbol{x}_0 = \_\_\_\_\_.$$

Proof:

First note that A is an $n \times n$ stochastic matrix so from Theorem 4.8 we know that there are n linearly independent eigenvectors. We can then substitute the eigenvalues from Theorem 4.8 in Theorem 4.7. Noting that if $0 < \lambda_j < 1$ we have $\lim_{k\to\infty} \lambda_j^k = 0$ the result follows immediately.

Exercise 4.97. Discuss how Theorem 4.9 greatly simplifies the PageRank iterative process described previously. In other words: there is no reason to iterate at all. Instead, just find ... what?

Exercise 4.98. Now use the previous two problems to find the resulting PageRank vector from the web in Figure 4.4? Be sure to rank the pages in order of importance. Compare your answer to the one that you got in problem 2.

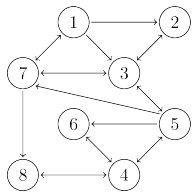


Figure 4.5: A second example web graph.

Exercise 4.99. Consider the web in Figure 4.5.

1. Write the H matrix and find the initial state $\boldsymbol{x}_0$,
2. Find steady state PageRank vector using the two different methods described: one using the iterative difference equation and the other using Theorem 4.9 and the dominant eigenvector.
3. Rank the pages in order of importance.

Exercise 4.100. One thing that we didn't consider in this version of the Google Page Rank algorithm is the random behavior of humans. One, admittedly slightly naive, modification that we can make to the present algorithm is to assume that the person surfing the web will randomly jump to any other page in the web at any time. For example, if someone is on page 1 in Figure 4.5 then they could randomly jump to any page 2 - 8. They also have links to pages 2, 3, and 7. That is a total of 10 possible next steps for the web surfer. There is a 2/10 chance of heading to page 2. One of those is following the link from page 1 to page 2 and the other is a random jump to page 2 without following the link. Similarly, there is a 2/10 chance of heading to page 3, 2/10 chance of heading to page 7, and a 1/10 chance of randomly heading to any other page.

Implement this new algorithm, called the *random surfer algorithm*, on the web

in Figure 4.5. Compare your ranking to the non-random surfer results from the previous problem.

Chapter 5

Ordinary Differential Equations

5.1 Intro to Numerical ODEs

> *"The mathematical discipline of differential equations furnishes the explanation of all those elementary manifestations of nature which involve time."*
> –Norwegian Mathematician Sophus Lie

Ordinary Differential Equations (ODEs) arise in all manner of contexts, but the most prevalent and frequently cited are from physics and engineering. The applications of Newton's Second Law, for example, give differential equations relating position, velocity, and acceleration. Newton's second law is the equation $F = ma$ where F is a force acting on a body, m is the mass of the body, and a is the acceleration. From Calculus recall that $a = v' = s''$, so Newton's second law can be rephrased as $F = mv'$, a differential equation with velocity as the unknown, or $F = ms''$, a second order differential equation with position as the unknown. Some of the cases where we use Newton's second law have nice analytic solutions, such as the motion of an object on a frictionless surface under constant force. However, many of these cases are highly idealized and not reflective of true reality. It doesn't take much to cause the differential equation(s) resulting from Newton's second law to be terribly hard, or maybe impossible, to solve. Just add some friction, perhaps allow multiple bodies to interact, or consider the forces that act differently on multiple length scales. A famous example is the three body problem where gravitational forces drive the motion of three celestial bodies. The resulting differential equations have no analytic solution and the only way to show how the positions of the bodies evolve in time is with a numerical approximation ... And that's just 3 celestial bodies! Imagine how

complicated the differential equations get trying to model our solar system (or our galaxy)!

Other examples of ODEs that are impossible to solve analytically are

- The motion of a pendulum where the angle from equilibrium is allowed to be large or the pendulum is allowed to swing over the top (e.g. the nonlinear pendulum).
- Systems of differential equations that model nonlinear predator-prey interactions (e.g. the Lotka-Voltera equations).
- Some types of damped oscillations in electric circuits (e.g. the Van der Pol oscillator).
- ... and many others.

The impossibility of solving a differential equation stems partly from the impossibility of integrating most functions. If we were to just randomly choose functions to integrate we would find that the vast majority do not have antiderivatives. The story in ODEs is the same: pick any combination of a function, its derivatives, and other forcing functions and you will find that there is no way to arrive at an analytic solution involving the regular operations and functions of mathematics: linear combinations, powers, roots, trigonometric functions, logarithms, etc. There are theorems from differential equations that will guarantee the existence and uniqueness of solutions to many differential equations, but just knowing that the solution exists isn't enough to actually go and find it. Numerical techniques give us an avenue to at least approximate these solutions.

So what *is* a numerical solution to a differential equation?
When solving a differential equation with analytic techniques the goal is to come up with a function. In a numerical solution the goal is typically to divide the domain (typically the domain is time) for the solution function into a fine partition, 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 associated with each time. In the strictest sense a list of approximate solutions on a partition *is* actually a function (a relation between input and output), but this isn't a function in terms of sines, powers, roots, logarithms, etc. The best way to deliver a numerical solution is just to make a plot. Your intuition of what the plot should look like based on the context of the problem is one of the best tools for you to check your work.

Exercise 5.1. Sketch a plot of the function that would model each of the following scenarios.

a. A population of an endangered species is slowly dying off. The rate at which the population decreases is proportional to the amount of population that is presently there. What does the population as a function of time look like for this species?

b. Consider a mass hanging from a spring that is suspended vertically from the ceiling. If the mass is given an initial upward *bump* and then left alone, what will the position of the mass relative to its equilibrium state be as a function of time?

c. A pollutant has entered a tributary for a certain reservoir, and a small concentration leaks into the water over a long period of time. The reservoir is dam controlled so the rate of release is well known and relatively constant. What does the function modeling the amount of pollutant look like as time goes on?

d. A drug is eliminated from the body via natural metabolism. Assume that there is some initial amount of drug in the body. What does the function modeling the amount of drug in the system look like over time?

Now let's formalize the conversation about differential equations, analytic solutions, and numerical solutions.

Definition 5.1 (Differential Equation)**.** A **differential equation** is an equation that relates the derivative (or derivatives) of an unknown function to itself.

Definition 5.2 (Solution to a Differential Equations)**.** A **solution to a differential equation** (also called an **analytic solution**) is a function which, when substituted into the differential equation, creates a true statement.

Example 5.1. The function $x(t) = 3e^{-0.25t}$ is a solution to the differential equation $x' = -0.25x$ with initial condition $x(0) = 3$. We can verify this by substituting $x(t)$ into the differential equation:

$$\begin{aligned} \frac{d}{dt}\left(x(t)\right) &\stackrel{?}{=} -0.25x(t) \\ \implies \frac{d}{dt}\left(3e^{-0.25t}\right) &\stackrel{?}{=} -0.25\left(3e^{-0.25t}\right) \\ \implies -0.25\left(3e^{-0.25t}\right) &\stackrel{\checkmark}{=} -0.25\left(3e^{-0.25t}\right) \end{aligned}$$

Furthermore, $x(0) = 3e^{-0.25\cdot 0} = 3e^0 = 3\checkmark$. Hence, the function $x(t) = 3e^{-0.25t}$ is indeed a solution to the differential equation $x' = -0.25x$ with $x(0) = 3$.

Definition 5.3 (Numerical Solution to a Differential Equation)**.** A **numerical solution to a differential equation** is a list of ordered pairs that gives a point-wise approximation to the actual solution.

In this chapter we will examine some of the more common ways to create approximations of solutions to differential equations. 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.

5.2 Recalling the Basics of ODEs

You should be familiar with the basics of differential equations from previous classes, but just in case you're a bit rusty, this section gives a very brief review of some of the basics.

Solving differential equations analytically is a subject unto itself, but it is worth our time here to revisit some of the basic techniques for solving differential equations. It should be noted that if an analytic solution exits then there is no reason to do any of the numerical techniques that we will discuss in this chapter – if you have an exact analytic solution then why on earth would you then approximate the solution!? The fact of the matter is, however, that the techniques for finding analytic solutions to differential equations are rather limited relative to the wild zoo of possible ODEs, and when the differential equations get complicated we will only have numerical approximations to lean back on. However, when we build approximation methods we will test them on differential equations for which we have the answer. So let's get started with some review.

Exercise 5.2. Identify which of the following problems are *differential* equations and which are *algebraic* equations. (Do not try to solve any of these equations)

a. $x^2 + 5x = 7x^3 - 2$
b. $x'' + 5x = 7x''' - 2$
c. $x' + 5 = -3x$
d. $x''x'x = 8$
e. $x^2 \cdot x = 8$

Exercise 5.3. Consider the differential equation $x' = 3x$ with an initial condition $x(0) = 4$. Which of the following functions is a solution to this differential equation, and what is the value of the constant in the function?

a. $x(t) = C\sin(3t)$
b. $x(t) = Ce^{3t}$
c. $x(t) = Ct^3$
d. $x(t) = t^3 + C$
e. $x(t) = e^{3t} + C$
f. $x(t) = \sin(3t) + C$

Exercise 5.4. Consider the differential equation $x' = 3x + t$ with an initial condition $x(0) = 4$. Which of the following functions is a solution to this

differential equation, and what are the values of the constants?

a. $x(t) = C_0 \sin(\sqrt{3}t) + C_1 t + C_2$
b. $x(t) = C_0 e^{3t} + C_1 t + C_2$
c. $x(t) = C_0 t^3 + C_1 t + C_2$
d. $x(t) = C_3 t^3 + C_2 t^2 + C_1 t + C_0$
e. $x(t) = e^{3t} + C_1 t + C_2$
f. $x(t) = \sin(3t) + C_1 t + C_2$

Exercise 5.5. Prove that the function $x(t) = -\frac{1}{2}\cos(2t) + \frac{7}{2}$ solves the differential equation $x' = \sin(2t)$ with the initial condition $x(0) = 3$.

Next we can recall one of the easiest techniques of solving ODEs by hand: separation of variables. We review separation here since we will often choose very easy (i.e. separable) differential equations to check our numerical work.

Theorem 5.1 (Separation of Variables)**.** *To solve a differential equation of the form*

$$\frac{dx}{dt} = f(x)g(t)$$

we can separate the variables and rewrite the problem as

$$\int \frac{dx}{f(x)} = \int g(t)dt.$$

Integrating both sides and solving for $x(t)$ gives the solution.

Proof. If $\frac{dx}{dt} = f(x)g(t)$ then we can first divide both sides by $f(x)$ (assuming that it is nonzero) and integrate both sides of the equation with respect to t to get

$$\int \frac{1}{f(x)} \frac{dx}{dt} dt = \int g(t)dt.$$

The expression $\frac{dx}{dt}dt$ in the left-hand integral is the definition of the differential dx so the integral equation can be rewritten as

$$\int \frac{1}{f(x)} dx = \int g(t)dt.$$

Note that it may be quite challenging to actually integrate the functions resulting from separation of variables. □

Exercise 5.6. Use separation of variables to solve the differential equation

$$\frac{dx}{dt} = x\sin(t)$$

with the initial condition $x(0) = 1$.

Exercise 5.7. Solve the differential equation $x' = -2x + 12$ with $x(0) = 2$ using separation of variables.

Exercise 5.8. Consider the differential equation

$$\frac{dx}{dt} = -\frac{1}{4}x + 4$$

with the initial condition $x(0) = 7$.

a. Solve the differential equation using separation of variables.
b. Substitute your solution into the differential equation and verify that you are indeed correct in your work in part (a).

There are MANY other techniques for solving differential equations, but a full discussion of all of those techniques is beyond the scope of this book. For the remainder of this chapter we will focus on finding *approximate* solutions to differential equations. It will be handy, however, to be able to check our work on problems where an analytic solution is available. Techniques you should remind yourself of are:

- The method of undetermined coefficients for first- and second-order linear differential equations.
- The method of integrating factors.
- The eigenvalue-eigenvector method for solving linear systems of differential equations.

5.3 Euler's Method

Exercise 5.9. Consider the differential equation $x' = -0.5x$ with the initial condition $x(0) = 6$.

a. Since we know that $x(0) = 6$ and we know that $x'(0) = -0.5 \cdot x(0)$ we can approximate the value of x at some future time step. Let's 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.

$$x(1) \approx \_\_\_\_\_$$

b. 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 $t = 2, 3, 4, 5, \ldots, 10$. 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	5	6	7	8	9	10
Approximation of $x(t)$	6										
Exact value of $x(t)$	6										

c. The "approximations of x" that you found in part (b) are a **numerical approximation** of the solution to the differentialequation. 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 analyze your results.

d. In Figure 5.1 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 $t = 1$, $t = 2$, ..., $t = 10$ on the plot and connect them with straight lines.

i. Why are we using straight lines to connect the points?
ii. What do you notice about your approximate solutions?

iii. Why is it helpful to have the slope field in the background on this plot?

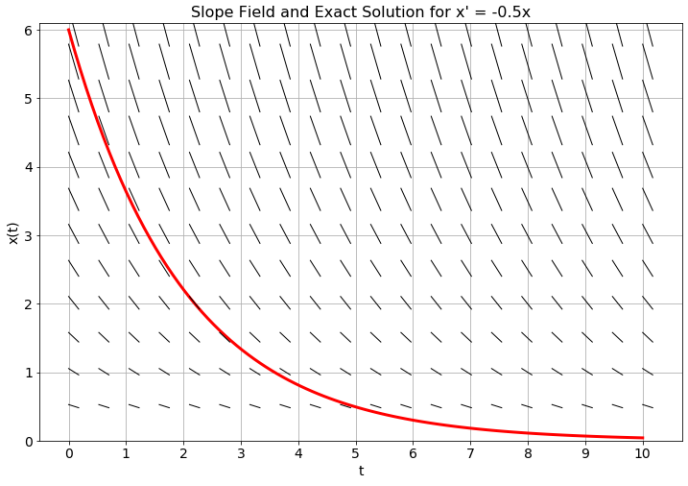


Figure 5.1: Plot your approximate solution on top of the slope field and the exact solution.

Exercise 5.10. In Figure 5.2 you see the analytic solution at $x(0) = 5$ 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 $t = 1$, $t = 2$, ..., $t = 10$. Connect your points with straight lines. The collection of line segments that you just drew is an aproximation 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 avlues at $t = 0.5$, $t = 1$, $t = 1.5$, ..., $t = 10$. 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?

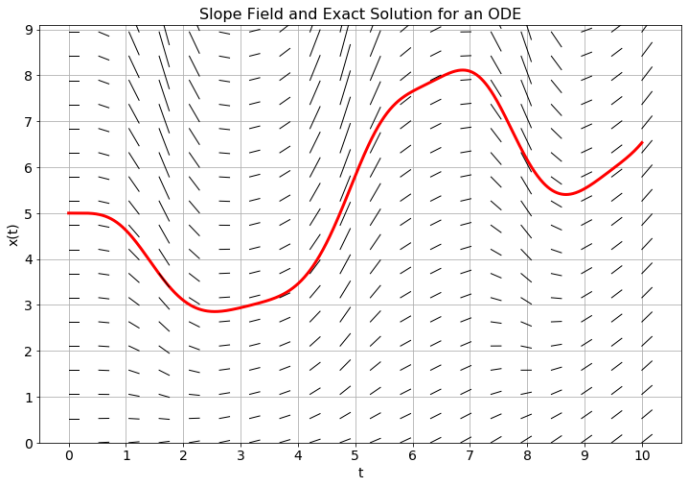


Figure 5.2: Plot your approximate solution on top of the slope field and the exact solution.

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'll start.

Definition 5.4 (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 Problem 3.11 that the first derivative of a function can be discretized as

$$x'(t) = \frac{x(t+h) - x(t)}{h} + \mathcal{O}(h)$$

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

$$\frac{x(t+h) - x(t)}{h} \approx f(t, x(t)).$$

Rewriting as a difference equation, letting $x_{n+1} = x(t_n + h)$ and $x_n = x(t_n)$, we get

$$x_{n+1} = x_n + hf(t_n, x_n)$$

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 5.3 shows a depiction of the idea. Notice in the figure that in regions of high curvature Euler's method will overshoot 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.

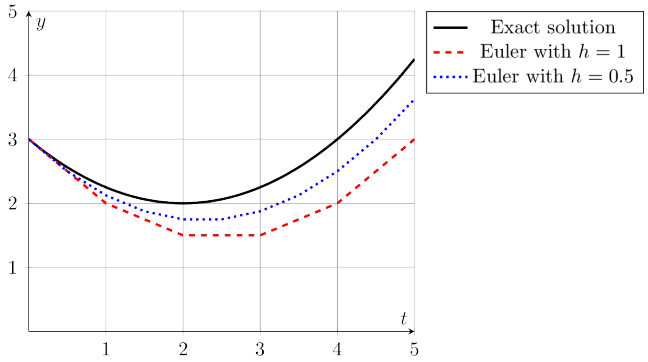


Figure 5.3: Numerical solutions to a differential equation using Euler's method.

Exercise 5.11. Why would Euler's method overshoot the exact solution in regions where the solution exhibits high curvature?

Exercise 5.12. 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.

```
def euler1d(f,x0,t0,tmax,dt):
    t = # set up the domain based on t0, tmax, and dt
    x = np.zeros_like(t) # set up an array for x that is the same size a t
    x[0] = # fill in the initial condition
    for n in range( ??? ): # think about how far we should loop
        x[n+1] = # advance the solution forward in time with Euler
    return t, x
```

Exercise 5.13. Test your code from the previous exercise on a first order differential equation where you know the answer. Then test your code on the differential equation

$$x' = -\frac{1}{3}x + \sin(t) \quad \text{where} \quad x(0) = 1.$$

The partial code below should get you started.

```
import numpy as np
import matplotlib.pyplot as plt
# put the f(t,x) function on the next line
# (be sure to specify t even if it doesn't show up in your test ODEs)
f = lambda t, x: # your function goes here
x0 = # initial condition
t0 = # initial time
tmax = # final time (your choice)
dt = # Delta t (your choice, but make it small)
t, x = euler1d(f,x0,t0,tmax,dt)
plt.plot(t,x,'b-')
plt.grid()
plt.show()
```

Exercise 5.14. The differential equation $x' = -\frac{1}{3}x + \sin(t)$ with $x(0) = 1$ has an analytic solution

$$x(t) = \frac{1}{10}\left(19e^{-t/3} + 3\sin(t) - 9\cos(t)\right).$$

The goal of this problem will be to compare the maximum error on the interval $t \in [0, 5]$ for various values of Δt in your Euler solver.

a. Write code that gives the maximum point-wise error between your numerical solution and the analytic solution given a value of Δt.

b. Using your code from part (a), build a plot with the value of Δt on the horizontal axis and the value of the associated 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.

c. In general, if you were to cut your value of Δt in half, what would that do to the value of the error? What about dividing Δt by 10? 100? 1000?

Exercise 5.15. 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 new 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 is Jackson's expected maximum absolute error?

Theorem 5.2. *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.)

Exercise 5.16. If we want to numerically solve the first order differential equation $x' = f(t, x)$ on the interval $t \in [0, 1]$ with Euler's method so that we realize a maximum absolute error of 10^{-8} between the numerical solution and the exact solution, then how many points do we need to subdivide the interval $[0, 1]$ into?

Exercise 5.17. If a mass is hanging from a spring then Newton's second law, $\sum F = ma$, gives us the differential equation $mx'' = F_{restoring} + F_{damping}$ where x is the displacement of the mass from equilibrium, m is the mass of the object hanging from the spring, $F_{restoring}$ 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.

a. Which of the following is a good candidate for a restoring force in a spring? Defend your answer.
   i. $F_{restoring} = kx$: The restoring force is proportional to the displacement away from equilibrium.
   ii. $F_{restoring} = kx'$: The restoring force is proportional to the velocity of the mass.
   iii. $F_{restoring} = kx''$: The restoring force is proportional to the acceleration of the mass.

b. Which of the following is a good candidate for a damping force in a spring? Defend your answer.

i. $F_{damping} = bx$: The damping force is proportional to the displacement away from equilibrium.
ii. $F_{damping} = bx'$: The damping force is proportional to the velocity of the mass.
iii. $F_{damping} = bx''$: The damping force is proportional to the acceleration of the mass.

c. Put your answers to parts (a) and (b) together and simplify to form a second-order differential equation for position:

$$\_\_\_x'' + \_\_\_x' + \_\_\_x = 0$$

d. 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.

$$x_0' = x_1$$
$$x_1' = \_\_\_\_\_$$

e. The code and Euler's method algorithm that we've 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

$$\boldsymbol{x}' = F(t, \boldsymbol{x})$$

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,

$$F(t, \boldsymbol{x}) = \begin{pmatrix} x_0' \\ x_1' \end{pmatrix} = \begin{pmatrix} x_1 \\ \_\_\_\_\_ \end{pmatrix}$$

f. 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

$$\frac{\boldsymbol{x}_{n+1} - \boldsymbol{x}_n}{h} = F(t_n, \boldsymbol{x}_n) + \mathcal{O}(h).$$

Rearranging and solving for $\boldsymbol{x}_{n+1}$ gives

$$\boldsymbol{x}_{n+1} = \_\_\_\_\_ + hF(\_\_\_, \_\_\_).$$

g. We now have a choice about how we're going to code this new 2D version of Euler's method. We could just include one more input function and one more input initial condition into the `euler()` function so that the Python function call is `euler(f0,f1,x0,x1,t0,tmax,dt)` where `f0` and `f1` are the two right-hand sides of the system, and `x0` and `x1` are the two initial

conditions. Alternatively, we could rethink our `euler()` function so that it accepts an array of functions and an array of initial conditions so that the Python function call is `euler(F,X,t0,tmax,dt)` where `F` is a Python array of functions and `X` is a Python array of initial conditions. Discuss the pros and cons of each approach.

h. The following Python function and associated script will implement the vector version of Euler's method. Complete the code and then use it to solve the system of equations from part (d). Use a mass of $m = 2$kg, a damping force of $b = 40$kg/s, and a spring constant of $k = 128$N/m. Consider an initial position of $x = 0$m (equilibrium) and an initial velocity of $x_1 = 0.6$m/s. Show two plots: a plot that shows both position and velocity vs time and a second plot, called a phase plot, that shows position vs velocity.

```
def euler(F,x0,t0,tmax,dt):
    t = # same code as before to set up a vector for time
    # Next we set up x so that it is an array where the columns are the
    # different dimensions of the problem.  For example, in this
    # problem there will be 2 columns and len(t) rows
    x = np.zeros( (len(t), len(x0)) )
    x[0,:] = x0 # store the initial condition in the first row
    for n in range(len(t)-1):
        x[n+1,:] = x[ ??? , ??? ] + dt*F(t[ ??? ], x[ ??? , ??? ])
    return t, x
```

To use the `euler()` function defined above we can use the following code. Fill in the code for this system of differential equations with this problem.

```
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)
# Next we plot the solutions against 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()
# Then we plot one solution against the other for a phase plot
# In a phase plot time is implicit (not one of the axes)
plt.plot(x[ ??? , ???], x[ ??? , ???], 'k--')
plt.grid()
```

```
plt.title('Phase Plot')
plt.xlabel('???')
plt.ylabel('???')
plt.show()
```

Exercise 5.18. Consider a collection of two connected mass-spring oscillators where there is a mass hanging from a fixture and a second mass is connected directly to the first (hanging vertically). For simplicity in this problem we will neglect damping. Let k_0 and k_1 be the spring constants for the two springs, respectively. Also let m_0 and m_1 be the respective masses. For simplicity in this problem we will take $m_0 = m_1 = m$.

a. Draw a picture of the physical setup described. Let x_0 be the position of mass 0 relative to its equilibrium. Let x_1 be the position of mass 1 relative to its equilibrium. Label the coordinate systems for the two springs on your picture.
b. Give a thorough explanation for why the following second order differential equation models the position of the first mass

$$mx_0'' = -k_0x_0 - k_1(x_0 - x_1).$$

c. Using similar logic from part (b), write a second order differential equation for the position of the second mass

$$mx_1'' = \_\_\_\_\_$$

d. We now have a system of two second order differential equations. We can convert this to four first order differential equations by introducing two new variables: $x_2 = x_0'$ and $x_3 = x_1'$. Write the full system of first order differential equations.
e. Use your vector-based `euler()` function to numerically solve the system of equations in several different physical scenarios. There are four variables so you will need to think carefully about which plots are the most explanatory. Also, if may be easiest to take $k_0 = 1$ and then take k_1 as the stiffness of spring 1 relative to spring 0 (e.g. if $k_1 = 1$ then the springs are the same stiffness, if $k_1 = 0.5$ then spring 1 is half as stiff, etc). To start with choose $m = m_0 = m_1 = 1$kg.

Exercise 5.19. Extend the previous exercise to that there are three masses hanging in a chain.

Exercise 5.20. If the speed of the mass in the mass-spring oscillator is *fast enough* then the damping force will no longer just be proportional to the velocity. Instead, at higher speeds the drag force is proportional to the square of the velocity. You can think of this as a bungee jumper jumping off of a bridge. Modify the

single mass-spring oscillator equation to allow for nonlinear quadratic damping. Solve the system numerically under several different physical conditions (stiff spring, non-stiff spring, high damping, low damping, different initial conditions, etc).

Exercise 5.21 (A Lotka-Volterra Model)**.** Test your code from the previous problems on the following system of differential equations by showing a time evolution plot (time on x_0 and populations on x_1) as well as a phase plot (x_0 on the x and x_1 on the y with time understood implicitly):
The 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 modeled by the classic 1920's Lotka-Volterra Model:

$$\begin{cases} x_0' &= \alpha x_0 - \beta x_0 x_1 \\ x_1' &= \delta x_0 x_1 - \gamma x_1 \end{cases}$$

where α, β, γ, and δ are positive constants. For this problems take $\alpha \approx 1.1$, $\beta \approx 0.4$, $\gamma \approx 0.1$, and $\delta \approx 0.4$.

a. First rewrite the system of ODEs in the form $\boldsymbol{x}' = F(t, \boldsymbol{x})$ so you can use your `euler()` code.

b. Modify your code from the previous problem so that it works for this problem. Use `tmax = 200` and an appropriately small time step. Start with initial conditions $x_0(0) = 20$ rabbits and $x_1(0) = 1$ fox.
c. Create the time evolution plot. What does this plot tell you in context?
d. Create a phase plot. What does this plot tell you in context?

e. If you cut your time step in half, what do you see in the two plots? Why? What is Euler's method doing here?

Exercise 5.22 (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 hanving 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.

$$S' = -\alpha SI$$

- The infected population gains people from the interactions with the susceptible people, but at the same time, infected people recover at a predictable rate.

$$I' = \alpha SI - \beta I$$

- 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.

$$R' = \beta I$$

a. Explain the minus sign in the S' equation in the context of the spread of a virus.
b. Explain the product SI in the S' equation in the context of the spread of a virus.
c. Find a numerical solution to the system of equations using your `euler()` function. Use the parameters $\alpha = 0.4$ and $\beta = 0.04$ with initial conditions $S(0) = 0.99$, $I(0) = 0.01$, and $R(0) = 0$. Explain all three curves in context.

5.4 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'll make a simple modification to Euler's method that has a surprisingly great payoff in the error rate.

Exercise 5.23. 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 didn't 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 5.24. Let's return to the simple differential equation $x' = -0.5x$ with $x(0) = 6$ that we saw in Exercise 5.9. Now we'll 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's build this idea together:

- What is the slope at time $t = 0$? $x'(0) =$ _____
- Use this slope to step a half step forward and find the x value: $x(0.5) \approx$ _____
- Now use the differential equation to find the slope at time $t = 0.5$. $x'(0.5) =$ _____
- 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 $t = 2, 3, \ldots, 10$. Also record the exact answer at each of these times.

t	0	1	2	3	4	5	6	7	8	9	10
Approximation 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 5.9?

Definition 5.5 (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 not 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 $x_{n+1/2}$.

$$x_{n+1/2} = x_n + \frac{h}{2} f(t_n, x_n)$$
$$x_{n+1} = x_n + h f(t_{n+1/2}, x_{n+1/2})$$

Note: Indexing by 1/2 in a computer is nonsense. Instead, we implement the

midpoint method with:

$$\begin{aligned} m_n &= f(t_n, x_n) \\ x_{temp} &= x_n + \frac{h}{2} m_n \\ x_{n+1} &= x_n + hf\left(t_n + \frac{\Delta t}{2}, x_{temp}\right) \end{aligned}$$

Exercise 5.25. Complete the code below to implement the midpoint method in one dimension.

```
def midpoint1d(f,x0,t0,tmax,dt):
    t = # build the times
    x = # build an array for the x values
    x[0] = # build the initial condition
    for n in range( ??? ): # be careful about how far you're looping
        # The interesting part of the code goes here.
    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( ??? , ??? , ??? )
plt.grid()
plt.show()
```

Exercise 5.26. 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 $t \in [0, 4]$. First get a numerical solution with Euler's method using $\Delta t = 0.1$. Then get a numerical solution with the midpoint method using the same value for Δt. Plot the two solutions on top of each other along with the exact solution

$$x(t) = \frac{1}{10}\left(19e^{-t/3} + 3\sin(t) - 9\cos(t)\right).$$

What do you observe? What do you observe if you make Δt a bit larger (like 0.2 or 0.3)? What do you observe if you make Δt very very small (like 0.001 or 0.0001)?

There are several key takeaways from this problem. Discuss.

Exercise 5.27. Repeat Exercise 5.14 with the midpoint method. Compare your results to what you found with Euler's method.

Exercise 5.28. We have studied two methods thus far: Euler's method and the Midpoint method. In Figure 5.4 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 2. Follow the slope for time Δt	1. Get the slope at time t_n 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

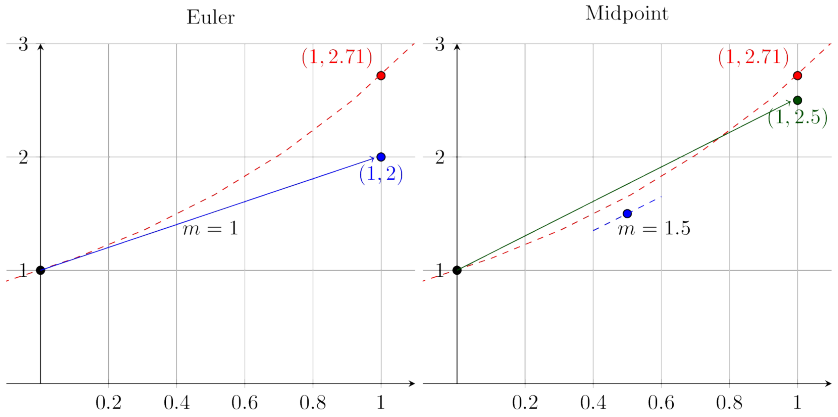


Figure 5.4: Graphical depictions of two numerical methods: Euler (left) and Midpoint (right). The exact solution is shown in red.

Exercise 5.29. 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 5.30 (Midpoint Method in Several Dimensions)**.** Modify your `euler()`

code from Exercise 5.17 so that you can use the midpoint method in as many dimensions as you like. 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.

5.5 The Runge-Kutta 4 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.

Exercise 5.31. Let's 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're sitting at the point (t_n, x_n).
- The slope of the solution curve to the ODE where you're standing is

$$\text{slope at the point } (t_n, x_n) \text{ is: } m_n = f(t_n, x_n)$$

- You take a half a step forward using the slope where you're standing. The new point, denoted $x_{n+1/2}$, is given by

$$\text{location a half step forward is: } x_{n+1/2} = x_n + \frac{\Delta t}{2} m_n.$$

- Now you're standing at $(t_n + \frac{\Delta t}{2}, x_{n+1/2})$ so there is a new slope here given by

$$\text{slope after a half of an Euler step is: } m_{n+1/2} = f(t_n + \Delta t/2, x_{n+1/2}).$$

- 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

$$x_{n+1} = x_n + \Delta t \cdot m_{n+1/2}$$

Exercise 5.32. One of the troubles with the midpoint method is that it doesn't actually use the information at the point (t_n, x_n). Moreover, it doesn't leverage a slope at the next time step t_{n+1}. Let's 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 our 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

$$x(t) = \frac{1}{10}\left(19e^{-t/3} + 3\sin(t) - 9\cos(t)\right)$$

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

# *********
# You should copy your 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 = x[n] + dt * m_n_plus_half
        m_n_plus_1 = f(t[n]+dt, x_n )
        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 5.33. 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 loglog plot) stayed about the same. You should have also found that the weighted average

$$\text{estimate of slope} = \frac{m_n + 2m_{n+1/2} + m_{n+1}}{4}$$

did just a little bit better than just a plain average. Why might this be? (If you haven't 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 loglog plot) using any of these methods?

Exercise 5.34. OK. Let's 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'll call this $m^*_{n+1/2}$ since it is a new estimate of $m_{n+1/2}$.

$$x^*_{n+1/2} = x_n + \frac{\Delta t}{2} m_{n+1/2}$$

$$m^*_{n+1/2} = f(t_n + \Delta t/2, x^*_{n+1/2})$$

Then calculate m_{n+1} using this new slope instead of what we did in the previous problem.

a. Draw a picture showing where this slope was calculated.
b. Modify the code from above to include this fourth slope.

c. 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 loglog plot since that indicates an improvement in the order of the error (the accuracy) of the method.

Exercise 5.35. 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. 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've only listed the most common ones here).

Experiment	m_n	$m_{n+1/2}$	$m^*_{n+1/2}$	m_n	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		

Experiment	m_n	$m_{n+1/2}$	$m^*_{n+1/2}$	m_n	Order of Error	Name
11	1/6	2/6	2/6	1/6		
12	1/6	3/6	1/6	1/6		
13	1/6	1/6	3/6	1/6		
14	1/7	2/7	3/7	1/7		
15	1/8	3/8	3/8	1/8		
16						
17						

Exercise 5.36. 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.

Theorem 5.3 (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 at the point t_n by using the following weighted sum:*

$$\textit{estimated slope} = \frac{m_n + 2m_{n+1/2} + 2m^*_{n+1/2} + m_n}{6}.$$

The order of the error in the RK4 method is $\mathcal{O}(\Delta t^4)$.

Exercise 5.37. In Theorem 5.3 we state the Runge-Kutta 4 method in terms of the estimates of the slope built up previously in this section. The notation that is commonly used in most numerical analysis sources is slightly different. Typically, the RK4 method is presented as follows:

$$\begin{aligned} k_1 &= f(t_n, x_n) \\ k_2 &= f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1) \\ k_3 &= f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2) \\ k_4 &= f(t_n + h, x_n + hk_3) \\ x_{n+1} &= x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

a. Show that indeed we have derived the same exact algorithm.

b. What is the advantage to posing the RK4 method in this way?

c. 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. Why is this important?

Exercise 5.38. Jackson wants to solve the differential equation $x' = f(t, x)$ on the domain $t \in [0, 1]$ so that the maximum absolute error is less than 10^{-8}.

a. What value of Δt would Jackson need if he were using Euler's method? How many function evaluations would Jackson's Euler algorithm end up doing in order to achieve his desired level of accuracy.
b. What value of Δt would Jackson need if he were using the midpoint method? How many function evaluations would Jackson's midpoint algorithm end up doing in order to achieve his desired level of accuracy.
c. What value of Δt would Jackson need if he were using the RK4 method? How many function evaluations would Jackson's RK4 algorithm end up doing in order to achieve his desired level of accuracy.
d. Discuss the implications of what you found in parts (a) - (c) of this problem.

Exercise 5.39. It would nice, but it would be completely impractical, to have a numerical method compute the approximate solution so that the maximum absolute error is less than machine precision 10^{-16}. That is an impracticality since we can't actually detect errors that small on a computer using double precision arithmetic. However, what if we wanted accuracy of 10^{-15} instead? Repeat the previous exercise with 10^{-15} as the goal for the maximum absolute error.

Exercise 5.40. Let's 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 same of simplicity, take $\Delta t = 1$ and perform 1 step of the RK4 method BY HAND to approximate the value $x(1)$.

Exercise 5.41. Write a Python function that implements the Runge-Kutta 4 method in one dimension. Test the problem on several differential equations where you know the solution.

```
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_like(t)
    x[0] = x0
    for n in range(len(t)-1):
        # the interesting bits of the code go here
    return t, x

f = lambda x, t: -(1/3.0)*x + np.sin(t)
```

```
x0 = # initial condition
t0 = 0
tmax = # your choice
dt = # pick something reasonable
t, x = rk41d(f,x0,t0,tmax,dt)
plt.plot(t,x,'b.-')
plt.grid()
plt.show()
```

Exercise 5.42 (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'll 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.

5.6 Animating ODE Solutions

Differential equations that depend on time are often best visualized when they are animated. This can also be said about any parameterized function, but in this present case we will focus on visualizing differential equations. There are several animation tools with python and we'll demonstrate only two primary technique here:

- `ipywidgets.interactive` is a tool that will produce an image with sliders that can be used to manually control an animation. The big advantage to `ipywidgets.iteractive` is that you can animate over several parameters, and hence use this tool as a playground for learning how parameters interact with each other.
- `matplotlib.animation` is a tool built directly into `matplotlib` that gives a playable animation (like a small movie). In this sort of animation we can only animate over one parameter or variable (like time), but this is most like what we would expect when animating a function that changes over time.

The reader should take careful note that the tools described here are meant to be used in Google Colab. These tools may not work as expected in other instances of Python and you may have to do some playing around (and Googling) to get it to work properly on your Python installation. Moreover, the animations are not built directly into the book since this book is delivered in several formats (HTML, PDF, and print). Instead you will find links to Google Colab documents that have the contain the code and animations.

5.6.1 ipywidgets.interactive

Consider the differential equation $x' = f(t, x)$ with $x(0) = x_0$. We would like to build an animation of the numerical solution to this differential equation over the parameters t but also over x_0 and Δt. The following blocks of python code walk through this animation.

Example 5.2 (ipywidgets.interactive)**.** Let's say that we want to control the numerical solution to the differential equation $x' = -\frac{1}{3}x + \sin(t)$ by manually altering the values of $x(0) = x_0$, t_{max}, and Δt. In this case we will solve the differential equation using Euler's method but note that our code could be easily modified to use other solvers.

First we import all of the appropriate libraries. Of particular interest is the `ipywidgets.interactive` library. This allows for images to be interactive with the use of sliders. Moving the sliders will provide a nice way to animate a plot manually.

```
from ipywidgets import interactive
import matplotlib.pyplot as plt
import numpy as np
```

In the next block of code we define our `euler()` solver. This particular step is only included because we are using Euler's method to solve this specific problem. In general, include an functions or code that are going to be used to produce the data that you will be plotting. We will also introduce the function `f` and the parameter `t0` since we will not be animating over these parameters.

```
def euler(f,x0,t0,tmax,dt):
  N = int(np.floor((tmax-t0)/dt)+1)
  t = np.linspace(t0,tmax,N+1)
  x = np.zeros_like(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

f = lambda t, x: -(1/3.0)*x + np.sin(t)
t0 = 0
```

Next we build a function that accepts only the parameters that we want to animate over and produces only a plot. This function will be called later by the `ipywidgets.interactive` function every time we change one of the parameters so be sure that this is a clean and fast function to evaluate (keep the code simple).

```
def eulerAnimator(x0,tmax,dt):
    t, x = euler(f,x0,t0,tmax,dt) # call on the euler function to build the solution
    plt.plot(t, x, 'b-') # plot the solution
    plt.xlim(0,tmax)
    plt.ylim( np.min(x)-1, np.max(x)+1)
    plt.grid()
    plt.show()
```

Now that we have everything set up we need to call on the `ipywidgets.interactive` command to turn the graphic into a visualization which can be controlled by sliders. In the code below we are allowing the initial condition to range between $x_0 = -2$ and $x_0 = 5$ in steps of 0.5, the time to range from $t_{max} = 1$ to $t_{max} = 30$ in steps of 0.1, and the time step to range from $\Delta t = 0.01$ to $\Delta t = 0.75$ in steps of 0.005.

```
interactive_plot = interactive(eulerAnimator,
                               x0=(-2, 5, 0.5),
                               tmax=(1, 30, 0.1),
                               dt=(0.01, 0.75, 0.005))
interactive_plot
```

A static snapshot of the animation applet is shown in Figure 5.5. When you build this animation you will have control over all three parameters. Like we mentioned before, this sort of animation can be a great playground for building insight into the interplay between parameters.

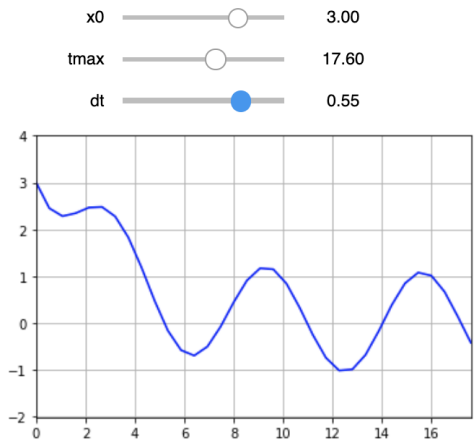


Figure 5.5: Snapshot of the ODE animation applet with ipywidgets.

Exercise 5.43. Modify the previous exercise to use a different numerical solver (e.g. the midpoint method) instead of Euler's method.

Exercise 5.44. Modify the animation routine above to simultaneously show the Euler, Midpoint, and RK4 solutions to a differential equation on top of each other. Animate over different values of Δt for fixed values of x_0 and t_{max}.

5.6.2 `matplotlib.animation`

The next animation package that we discuss is the `matplotlib.animation` package. This particular package is very similar to `ipywidgets.interactive`, but results only in a playable movie that is embedded within the Google Colab environment.

Exercise 5.45. Again we will consider the differential equation $x' = -\frac{1}{3}x + \sin(t)$ but this time we will only be interested in an animation over time.

We start the code by importing all of the necessary libraries. Take note that we import the `matplotlib.animation` and `matplotlib.rc` libraries in order to build the animation. We then import the `IPython.display.HTML` library to take care of embedding the player into the Google Colab environment.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation, rc
from IPython.display import HTML
```

Next we write all of the code necessary to build an Euler solution for the differential equation. Take note, of course, that much of this code is specific only to this problem and what we really need here is code that produces data for the animation.

```
def euler(f,x0,t0,tmax,dt): # this is the Euler function
  N = int(np.floor((tmax-t0)/dt)+1)
  t = np.linspace(t0,tmax,N+1)
  x = np.zeros_like(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

# Now we define the parameters for the Euler function
dt = 1e-2
x0 = 3 # initial condition
t0 = 0
tmax = 10
```

```
f = lambda t, x: -(1/3.0)*x + np.sin(t)

# Next we get the full Euler solution associate with these parameters.
# Be careful that you put this outside your animation loop so that you
# don't build this over and over.
t, x = euler(f,x0,t0,tmax,dt)
```

Next we have to set up the figure that we are going to animate. This involves:

- setting up the axes,
- building any features onto the axes that we want (e.g. a grid, axis labels, axis limits, etc)
- and then we build a variable that we call `frame`.
  - The variable `frame` contains a blank plot with no data.
  - Notice that we define the line and marker styles here.
  - Also notice the comma in the definition of the `frame` variable. This is here since there are several Python objects inside `ax.plot()` and we only want to unpack the first one into `frame`.

```
fig, ax = plt.subplots()
plt.close()
# Below we set up many of the global parameters for the plot.
# Much of what we do here depends on what we are trying to animate.
ax.grid()
ax.set_xlabel('Time')
ax.set_ylabel('Approximate Solution')
ax.set_xlim(( t0, tmax))
ax.set_ylim((np.min(x)-0.5, np.max(x)+0.5))
frame, = ax.plot([], [], linewidth=2, linestyle='--') # also set line and marker param
```

Now we build a function that accepts only the animation frame number, `N`, and adds appropriate elements to the plot defined by `frame`.

```
def animator(N): # N is the animation frame number
  T = t[:N] # get t data up to the frame number
  X = x[:N] # get x data up to the frame number
  ax.set_title('Time='+t[N]) # display the current simulation time in the title
  frame(T,X) # put the data for the current frame into the varable "frame"
  return (frame,)
```

In the next block of code we define which frames we want to use in the animation and then we call upon the `matplotlib.animation` function to build the animation.

```
# The Euler solution takes many very small time steps.  To speed up the
# animation we view every 10th iteration.
PlotFrames = range(0,len(t),10)
```

```
anim = animation.FuncAnimation(fig, # call on the figure
                               animator, # call the function that builds the animation frame
                               frames=PlotFrames, # tell which frames to pass to animator
                               interval=100 # delay between frame
                               )
```

Finally, we embed the animation into the Google Colab environment. Take note that if you are using a different Python IDE then you may need to experiment with how to show the resulting animation.

```
rc('animation', html='jshtml') # embed in the HTML for Google Colab
anim # show the animation
```

A static snapshot of the resulting animation can be seen in Figure 5.6. The controls for the animation should be familiar from other media players.

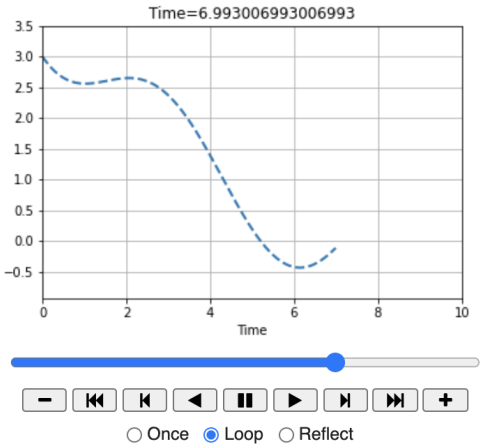


Figure 5.6: Snapshot of the ODE animation applet with matplotlib animation.

Exercise 5.46. Modify the code from the previous exercise to show faster and slower animations.

Exercise 5.47. Modify the `matplotlib.animation` code from Exercise 5.45 to use a different differential equation solver.

Exercise 5.48. Modify the `matplotlib.animation` code from Exericse 5.45 to show the Euler, Midpoint, and RK4 solutions to a differential equation on top of each other for a fixed value of Δt.

5.7 The Backwards Euler Method

We have now built up a fairly large 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's use that*? Seems like an unrealistic hope, but that is exactly what the last method covered in this section does.

Definition 5.6 (Backward Euler Method)**.** We want to solve $x' = f(t, x)$ so:

- Approximate the derivative by looking forward in time(!)

$$\frac{x_{n+1} - x_n}{h} \approx f(t_{n+1}, x_{n+1})$$

- Rearrange to get the difference equation

$$x_{n+1} = x_n + hf(t_{n+1}, x_{n+1}).$$

- We will always know the value of t_{n+1} and we will always know the value of x_n, but we don't 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 you do not explicitly calculate x_{n+1} but instead there is some intermediate calculation that needs to happen to solve for x_{n+1}. The (usual) advantage to an implicit method such as Backward Euler is that you can take far fewer steps with reasonably little loss of accuracy. We'll see that in the coming problems.

Exercise 5.49. Let's take a few steps through the backward Euler method on a problem that we know well: $x' = -0.5x$ with $x(0) = 6$.
Let's take $h = 1$ for simplicity, so the backward Euler iteration scheme for this particular differential equation is

$$x_{n+1} = x_n - \frac{1}{2}x_{n+1}.$$

Notice that x_{n+1} shows up on both sides of the equation. A little bit of rearranging gives

$$\frac{3}{2}x_{n+1} = x_n \quad \Longrightarrow \quad x_{n+1} = \frac{2}{3}x_n.$$

a. Complete the following table.

t	0	1	2	3	4	5	6	7	8	9	10
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							

b. Compare now to what we found for the midpoint method on this problem as well.

Exercise 5.50. 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 isn't true! Let's consider a slightly more complicated differential equation and see what happens

$$x' = -\frac{1}{2}x^2 \quad \text{with} \quad x(0) = 0.$$

a. Recall that the backward Euler approximation is

$$x_{n+1} = x_n + hf(t_{n+1}, x_{n+1}).$$

Let's take $h = 1$ for simplicity (we'll make it smaller later). What is the backward Euler formula for this particular differential equation?

b. 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's be more explicit:
We know that $x(0) = 6$ so in our numerical solutions, $x_1 = 6$. In order to get x_2 we consider the equation $x_2 = x_1 - \frac{1}{2}x_2^2$. Rearranging we see that we need to solve $\frac{1}{2}x_2^2 + x_2 - 6 = 0$ in order to get x_2. Doing so gives us $x_2 = \sqrt{13} - 1 \approx 2.606$.

c. 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.

d. Work our the analytic solution for this differential equation (using separation of variables perhaps). Then compare the values that you found in parts (b) and (c) of this problem to values of the analytic solution and values that you would find form 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

$$x_{n+1} = x_n + hf(t_{n+1}, x_{n+1}).$$

Notice that this is the same as solving the equation

$$x_{n+1} - hf(t_{n+1}, x_{n+1}) - x_n = 0.$$

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.

Exercise 5.51. Consider the function `backwardEuler1d()` below. How do you define the function `G` inside the `for` loop and what seed do you use to start 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_like(t)
    x[0] = x0
    for n in range(len(t)-1):
        G = lambda X: ??? # define this function
        x[n+1] = optimize.fsolve(G, ??? )[0] # give the correct seed
    return t, x
```

Exercise 5.52. Test the Backward Euler method from the previous problem on several differential equations where you know the solution.

Exercise 5.53. 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

$$x(t) = \frac{1}{10}\left(19e^{-t/3} + 3\sin(t) - 9\cos(t)\right)$$

Exercise 5.54. What is the order of the error on the Backward Euler method? Given this answer, what are the pros and cons of the Backward Euler method over the regular Euler method? What about compared to the Midpoint or Runge Kutta methods?

Exercise 5.55. 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's take a look. Consider, for example, the really simply differential equation $x' = -x$ with $x(0) = 1$ on the interval $t \in [0, 2]$. The analytic solution is $x(t) = e^{-t}$. 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). Try the same experiment on another differential equation where we know the exact solution and the solution has some regions of high curvature. What do you notice? What does Backward Euler do that is an improvement on regular Euler?

5.8 Fitting ODE Models to Data

To end this chapter we will examine a very common scientific situation: We have data from an experiment and a (challenging to solve) differential equation modeling the data that has some parameter that controls the behavior. We want to find the value of the parameter that gives us the best fit between our numerical solution and the data. For example, say we have temperature data for a cooling liquid and we have a differential equation for temperature that depends on a parameter related to the thermal properties of the container. We would like to use the data and the differential equation to determine the parameter for the container. As another example, say we have the number of patients that become ill with a virus each day and we actually want to know the long-term impacts on the population. An SIR differential equation model might describe the dynamics of the situation well, and the data can be used to determine the transmission rate parameters in the model.

Data fitting has been examined a few times in this book (see the Least Squares section in Chapter 3 and the Over determined Systems section in Chapter 4). The present situation is really not that much different than regular least squares curve fitting.

- Propose a model function: In this case our model function will be a numerical solution to a differential equation given some value for an unknown parameter.
- Calculate the sum of the squared residuals: In this case, we need to match the times between the numerical solution and the data. There will likely be far more points in the numerical solution than there will be in the data so we will have to carefully select the points that closely match between the two. Then calculating the sum of the squared error is simple.
- Use an optimization routine to find the value of the best parameter: In this case this is no different than regular least squares. We are trying to

find the value of the parameter that minimizes the sum of the squared residuals.

Exercise 5.56 (Newton's Law of Cooling)**.** From Calculus you may recall Newton's Law of Cooling:

$$\frac{dT}{dt} = -k(T - T_{ambient})$$

where T is the temperature of some object (like a cup of coffee), $T_{ambient}$ is the temperature of the ambient environment, and k is the proportionality constant the governs the rate of cooling. This is a classic differential equation with a well known solution.[1] In the present situation we don't want the analytic solution, but instead we will work with a numerical solution since we are thinking ahead to where the differential equation may be very hard to solve in future problems. We also don't want to just look at the data and guess an algebraic form for the function that best fits the data. That would be a trap! (why?) Instead, we rely on our knowledge of the physics of the situation to give us the differential equation.

The following data table gives the temperature (degrees F) at several times while a cup of tea cools on a table [7]. The ambient temperature of the room is $65°F$.

Time (sec)	Temperature
0	160
60	155
180	145
210	142
600	120

Plot the data as a scatter plot.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
data = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSulliva
# or you can load the data directly with
# data = np.array([[0,160],[60,155],[180,145],[210,142],[600,120]])
plt.plot(data[??? , ???] , data[??? , ???], 'b*')
plt.grid()
plt.show()
```

Now we will build several Python functions as well as several additional lines of code that are created specifically for this problem. Note that every parameter

[1] If you don't know the solution to Newton's Law of Cooling then take a moment and do the separation of variables to solve for $T(t)$.

estimate problem of this type will take similar form, but there may be subtle differences depending on the data that you need to account for in each problem. You will need to taylor make parts of each parameter estimation script for each new problem.

- First we set the stage by defining Δt, a collection of times that contains the data, the function $f(t, x; k)$ which depends on the parameter k, and any other necessary parameters of our specific problem.

```
import numpy as np
Tambient = ???
# Next choose an appropriate value of dt.  Choosing dt so that values
# of time in the data fall within the times for the numerical solution
# is typically a good practice (but is not always possible).
dt = ???
t0 = 0 # time where the data starts
tmax = ??? # just beyond where the data ends
t = np.arange(t0,tmax+dt,dt) # set up the times
f = lambda t, x, k: -k*(x - Tambient) # our specific differentia equation
x0 = ??? # initial condition pulled from the data
```

- Now we build a Python function that will accept a value of the parameter k as the only input and will return a high quality numerical solution to the proposed differential equation.

```
def numericalSolution(k):
    x = np.zeros_like(t)
    x[0] = x0
    for n in range(len(t)-1):
        # put the code necessary to build a good numerical solver here
        # be sure to account for the parameter k in each of your function calls.
    return t, x
```

- Spend a little time now playing with different parameters and plotting numerical solutions along with the data to determine the proper ballpark value of the parameter.
- Now we need to write a short Python script that will find all of the indices where the value of time in the data closely match values of time in the numerical solution. There are many ways to do this, but the most readable is a pair of nested `for` loops. Outline what the following code does. Why are we using `dt/2` in the code below? You should work to find more efficient ways to code this for bigger problems since the nested `for` loops is potentially quite time consuming.

```
indices = []
for j in range(len(t)):
    for k in range(len(data)):
        if np.abs(t[j] - data[k,0]) < dt/2:
```

```
            indices.append(j)
```

- Now we build a Python function `dataMatcher(k)` which accepts the parameter k and outputs the sum of the squared residuals between the numerical solution associated with k and our data. Carefully dissect the following code.

```
def dataMatcher(k):
    t, x = numericalSolution(k)
    err = []
    counter = 0
    for n in indices:
        err.append( (data[counter,1] - x[int(n)])**2 )
        counter += 1
    print("For k=",k[0],",  SSRes=",np.sum(err)) # optional
    return np.sum(err)
```

- Test your `dataMatcher()` function to be sure that it is working properly on a value of k which visually matches the data well.

- Finally, we call upon the `scipy.optimize.minimize()` function to iteratively try different values of the parameter k and to find the one that minimizes the sum of the squared residuals. Be sure to start k at a value that gives a reasonably good visual match between the numerical solution and the data. Once the optimization routine is done you should plot your best solution on top of the data to verify that it indeed found a good solution. You'll notice that there are several options that you can send to the `scipy.optimize.minimize()` command. Play with these options to see what they do and how they impact the quality of your solution.

```
import scipy.optimize as sp
# Choose an initial value of k and put it into the following code
# in place of the "???".  Note that we are sending a few parameters
# to the optimization tool.  Be sure to understand these options
# and take care that these options problem dependent and you will
# need to choose these again for the next new problem.
K = sp.minimize(dataMatcher,???, options = {'maxiter': 5}, tol=1e-2)
print(K)
t, x = numericalSolution(K.x[0])
plt.plot(t,x,'r--',data[:,0],data[:,1],'b*')
plt.grid()
plt.show()
```

- Note: If your optimization does not terminate successfully then you'll need to go back to the point where you guess a few values for the parameter so that your initial guess for `scipy.optimize.minimize()` is *close* to what it should be. It is always helpful to think about the physical context of

the problem to help guide your understanding of which value(s) to choose for your parameter.

To recap:

- We have data and a proposed differential equation with an unknown parameter.
- We matched numerical solutions to the differential equation to the data for various values of the parameter.
- We used an optimization routine to find the value of the parameter that minimized the sum of the squared residuals between the data and the numerical solution.

At this point you can now use the *best* numerical solution to answer questions about the scientific setup (e.g. extrapolation).

Exercise 5.57. In the paper *Steeping Tea: A differential equations approach to a great cup of fruit tea* [7], the authors give color data from photographs of tea that is steeping a clear mason jar. The temperature data from the previous exercise in this section were taken from this paper.

a. Read the introduction, methods, and experimental setup in the paper.
b. Think carefully about the physics of the problem to propose a differential equation (NOT an algebraic function) which would best models the grayscale data found on page 3 of the paper. Your model will likely involve at least one unknown parameter.
c. Use the least squares data fitting routine outlined in the previous exercise to find the value(s) of your parameter(s) which will create a high quality match between the numerical solution to your ODE and the data.
d. Plot your solution curve along with the data.

Exercise 5.58 (Village Epidemic)**.** (This exercise is modified from [8])

In the mid seventeenth century in a small village in England a form of the Plague spread from July 3 through October 20 in one year. We note three classes of individuals: Susceptible, Infective, and Removed. The latter group consists of those who have died from the disease or who developed an immunity from the disease, having already had the disease. We keep track of the following:

- $S(t)$ = the number of Susceptibles on day t of the epidemic. $S(0) = 235$.
- $I(t)$ = the number of Infectives on day t of the epidemic. $I(0) = 14$
- $R(t)$ = the number of Removeds on day t of the epidemic. $R(0) = 0$.

A standard SIR model takes the form

$$\begin{aligned} S' &= -\alpha SI \\ I' &= \alpha SI - \beta I \\ R' &= \beta I. \end{aligned}$$

Data was gathered on the outbreak and is shown in the table below.

Time (days)	Susceptibles	Infectives
0	235	14
16	201	22
31	153.5	29
47	121	21
62	108	8
78	97	8
109	83	0

Use the least squares fitting technique discussed in this section to find the parameters α and β that minimize the sum of the squared residuals between a numerical solution of the SIR model and the data. You can load the data directly with the code below.

Note: The total population is fixed.

```
import numpy as np
import pandas as pd
data = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSulliva
```

Exercise 5.59 (Bedridden Boys Problem)**.** (This problem is modified from [9])

A boarding school is a relatively closed community in which all students live on campus, teachers tend to live on or near campus, and students do not regularly interact with people not in the boarding school community. The table below gives data for an influenza outbreak at a boarding school in England during which there were no fatalities. There were 763 boys at the English boarding school from which the data was obtained.

Time (days)	Number of Bedridden Boys
0	1
1	3
2	25
3	72
4	222
5	282
6	256
7	233
8	189
9	123
10	70
11	25
12	11

Time (days)	Number of Bedridden Boys
13	4

Propose a differential equation model that includes the number of bedridden (sick) boys. Your model will likely have one or more unknown parameters. Use the technique from this section to find the parameters. Complete the problem by showing a plot of the number of bedridden boys along with the data. You can load the data directly with the code below.

```
import numpy as np
import pandas as pd
data = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSullivan/Numerica
```

5.9 Exercises

5.9.1 Algorithm Summaries

Exercise 5.60. 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 5.61. Explain in clear language what Euler's method does geometrically.

Exercise 5.62. 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? Explain the meaning of the order of the method in the context of solving a differential equation.

Exercise 5.63. Explain in clear language what the Midpoint method does geometrically.

Exercise 5.64. 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? Explain the meaning of the order of the method in the context of solving a differential equation.

Exercise 5.65. Explain in clear language what the Runge Kutta 4 method does geometrically.

Exercise 5.66. 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? Explain the meaning of the order of the method in the context of solving a differential equation.

Exercise 5.67. Explain in clear language what the Backward Euler method does geometrically.

5.9.2 Applying What You've Learned

Exercise 5.68. Test the Euler, Midpoint, and Runge Kutta methods on the differential equation

$$x' = \lambda\,(x - \cos(t)) - \sin(t) \quad \text{with} \quad x(0) = 1.5.$$

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 $\lambda = -1$, $\lambda = -10$, $\lambda = -10^2$, $\ldots$, $\lambda = -10^6$. 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 5.69. 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 = (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_like(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 5.70. We wish to solve the boundary valued problem $x'' + 4x = \sin(t)$ with initial condition $x(0) = 1$ and boundary condition $x(1) = 2$. 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.
Hint: First write the problem as a system of first order differential equations. Then think about how your bisection method code might help you.

Exercise 5.71. Write code to solve the boundary valued differential equation

$$y'' = \cos(t)y' + \sin(t)y \quad \text{with} \quad y(0) = 0 \quad \text{and} \quad y(1) = 1.$$

Exercise 5.72. 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

$$\begin{cases} \frac{dx}{dt} &= -\alpha y \\ \frac{dy}{dt} &= \beta x - \gamma y^2 \end{cases}$$

where $x(t)$ is Romeo's love for Juliet and $y(t)$ is Juliet's love for Romeo at time t.

Your tasks:

a. 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?

b. 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're 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.
Hint: One way to do this problem is very similar to a bisection method for root finding.

Exercise 5.73. In this problem we'll 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

$$x''(t) = \frac{-x}{\left(\sqrt{x^2 + y^2}\right)^3} \quad \text{and} \quad y''(t) = \frac{-y}{\left(\sqrt{x^2 + y^2}\right)^3}.$$

a. 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

$$x' = \_\_\_\_\_$$
$$v_x' = \_\_\_\_\_$$
$$y' = \_\_\_\_\_$$
$$v_y' = \_\_\_\_\_$$

b. 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 $(0, 0.5]$.

c. Give an animated plot showing $x(t)$ versus $y(t)$.

Exercise 5.74. 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)

$$E(t) = \begin{pmatrix} x_e(t) \\ y_e(t) \end{pmatrix} \quad \text{and} \quad P(t) = \begin{pmatrix} x_p(t) \\ y_p(t) \end{pmatrix}.$$

Let's 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

$$\|P'(t)\| = k\|E'(t)\|$$

and from the second assumption we have

$$\frac{P'(t)}{\|P'(t)\|} = \frac{E(t) - P(t)}{\|E(t) - P(t)\|}.$$

Solving for $P'(t)$ the differential equation that we need to solve becomes

$$P'(t) = k\|E'(t)\|\frac{E(t) - P(t)}{\|E(t) - P(t)\|}.$$

Your Tasks:

a. Explain assumption #2 mathematically.

b. Explain assumption #3 physically. Why is this assumption necessary mathematically?

c. Write code to find the path of the pursuer if the evader has the parameterized path

$$E(t) = \begin{pmatrix} 0 \\ 5t \end{pmatrix} \quad \text{for} \quad t \geq 0$$

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.

d. Modify your code from part (c) to find the path of the pursuer if the evader has the parameterized path

$$E(t) = \begin{pmatrix} 5 + \cos(2\pi t) + 2\sin(4\pi t) \\ 4 + 3\cos(3\pi t) \end{pmatrix} \quad \text{for} \quad t \geq 0$$

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.

e. 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.

f. (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 5.75. (This problem is modified from [6])

One of the favorite 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

$$\frac{dB}{dt} = -0.05B + \left(\frac{0.02}{500}\right) BK$$
$$\frac{dK}{dt} = 0.25K - \left(\frac{0.10}{150000}\right) BK.$$

a. What are the units of $\frac{dB}{dt}$ and $\frac{dK}{dt}$?

b. 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.

c. Find a numerical solution to the differential equation model using $B(0) = 75,000$ whales and $K(0) = 150$ tons per acre.

d. 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 5.76. (This problem is modified from [10])

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 5.77. A trebuchet catapult throws a cow vertically into the air. The differential equation describing its acceleration is

$$\frac{d^2x}{dt^2} = -g - c\frac{dx}{dt}\left|\frac{dx}{dt}\right|$$

where $g \approx 9.8$ m/s^2 and $c \approx 0.02$ 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 5.78 (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's consider the system of ODEs

$$x' = y$$
$$y' = -by - c\sin(x).$$

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. (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:

a. 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 to code with several different initial conditions. For the pendulum system be sure to describe what your initial conditions mean in the physical setup.

b. Use `scipy.integrate.odeint()` to solve a nontrivial scalar ODE of your choosing. Clearly show your ODE and give plots of your solutions with several different initial conditions.

c. 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.

d. Solve the system of differential equations from Exercise 5.73 using `scipy.integrate.odeint()`. Show appropriate plots of your solution.

5.10 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. Take the time to read Appendix B before you write your final solution.

5.10.1 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.

5.10.2 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:

$$\begin{aligned}
\frac{dP}{dt} &= -\left(k_0 + k_1 D_1 + k_2 D_2 + k_3 D_3\right) P + k_0 u \\
\frac{dD_1}{dt} &= -k_{D_1} D_1 + \sum_{j=1}^{N_1} \delta(t - \tau_{1,j}) \\
\frac{dD_2}{dt} &= -k_{D_2} D_2 + \sum_{j=1}^{N_2} \delta(t - \tau_{2,j}) \\
\frac{dD_3}{dt} &= -k_{D_3} D_3 + \sum_{j=1}^{N_3} \delta(t - \tau_{3,j})
\end{aligned}$$

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 $u = 8.01$, $k_0 = \ln(2)/2$, $k_1 = 0.319$, $k_2 = 0.184$, $k_3 = 0.201$, $k_{D_1} = \ln(0.5)/(-10)$, $k_{D_2} = \ln(0.5)/(-4)$, and $k_{D_3} =$

$\ln(0.5)/(-4)$. 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?

5.10.3 The H1N1 Virus

The H1N1 virus, also known as the "bird flu", is a particularly virulent bug but thankfully is also very predicable. 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 $j = 1, 2, \cdots, 9$), and recovered people (R). Write and numerically solve a system of 11 differential equations modeling 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.

5.10.4 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:

$$\frac{dv_x}{dt} = 0 \quad \text{and} \quad \frac{dv_z}{dt} = -g$$

where v_x and v_z are the velocities in the x and z directions respectively and g is the acceleration due to gravity ($g = 9.8$ m/s^2). We can use these equations to *easily* show that the resulting trajectory is parabolic. Once we know this we can easily[2] 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 $\theta_0 = 45°$.

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

$$\frac{dv_x}{dt} = -cv_x\sqrt{v_x^2 + v_z^2} \quad \text{and} \quad \frac{dv_z}{dt} = -g - cv_z\sqrt{v_x^2 + v_z^2}$$

where the constant c depends on the shape and density of the shell and the density of air. For this project assume that $c = 10^{-3}m^{-1}$. To calculate the components of the position vector recall that since the derivative of position, $s(t)$, is velocity we have

$$s_x(t) = \int_0^t v_x(\tau)d\tau \quad \text{and} \quad s_z(t) = \int_0^t v_z(\tau)d\tau.$$

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 math, 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 neighborhood.

[2] I'm showing you the algebra here, but it really isn't necessary to show this level of routine algebra in your papers. Only show the algebra and other calculations that are necessary for the reader to understand the work that you're doing.

a. 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.)

b. Discuss the sensitivity in your solutions to variations in the constant c.

c. 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 haven't 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.

Chapter 6

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

Partial differential equations (PDEs) are differential equations involving the partial derivatives of an unknown multivariable function. The study of PDEs is highly motivated by physics. In most of this chapter we will examine two classical problems from physics: heat transport phenomenon and wave phenomenon. Don't think, however, that just because we're focusing only 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 modeled via the change in multiple dimensions (and time) is likely governed by a PDE model. Some common phenomena that are modeled 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
- traveling 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 ultimately solving PDEs in terms of our usual three spatial dimensions along with an extra dimension for time. However, in many cases we don't 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 don't need to worry about time.

So what *is* a Partial Differential Equation?
Definition 6.1 (Partial Differential Equation)**.** A partial differential equation (PDE) is an equation that relates a function and its partial derivatives. Typically we use the function name u for the unknown function, and in most cases that we consider in this book we are thinking of u as a function of time t as well as one, two, or three spatial dimensions x, y, and z.

Specific examples of some common PDEs are:

- In one spatial dimension the "heat equation" takes the form

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}.$$

  This PDE states that the time derivative of the function u is proportional to the second derivative with respect to the spatial dimension x. This PDE can be used to model the time evolution of temperature in a heated one-dimensional rod.

- In three dimensions the heat equation takes the form

$$\frac{\partial u}{\partial t} = D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right).$$

  This PDE states that the time derivative of u is proportional to the sum of the three spatial second derivatives. This PDE can be used to model the time evolution of temperature in a heated three-dimensional object.
- As a third example, consider the Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0.$$

  This PDE states that the sum of the three second order derivatives is always zero. The Laplace equation gives the shape of an object that has minimum surface area while fixed to some boundary (like a soap bubble attached to a wire frame).

- Finally, consider the three dimensional wave equation

$$\frac{\partial^2 u}{\partial t^2} = k\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right).$$

  This PDE states that the acceleration at a point is proportional to the sum of the spatial second derivatives. The wave equation can be used to model the propagation of a sound wave through the air.

There is a wealth of wonderful theory for finding analytic solutions to many special classes of PDEs. However, most PDEs simply do not easily 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. Just like with ODEs, the trouble comes in that you are ultimately trying to integrate to solve the PDE, and we know that finding an antiderivative is usually an impossible task!

Recall that numerical solutions to ODEs were approximations of the value of the unknown function at every time. Similarly, numerical solutions to PDEs are going to be approximations of the value of the unknown function at every time AND at every point in the spatial domain.

What we'll 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.

Lastly, since PDEs require a strong background in the notions of multivariable calculus let's at least start with an exercise that should jog your memory about such things as the partial derivative, the gradient, and the divergence operators.

Exercise 6.1. With your partner answer each of the following questions. The main ideas in this problem *should* be review from multivariable calculus. If you and your partner are stuck then ask another group.

a. What is a partial derivative (explain geometrically)
b. What is the gradient of a function? What does it tell us physically or geometrically? If $u(x, y) = x^2 + \sin(xy)$ then what is ∇u?
c. What is the divergence of a vector-valued function? What does it tell us physically or geometrically? If $F(x, y) = \langle \sin(xy), x^2 + y^2 \rangle$ then what is $\nabla \cdot F$?
d. If u is a function of x, y, and z then what is $\nabla \cdot \nabla u$?

6.1 Solutions to PDEs

Example 6.1. If we were to claim that $x(t) = 7e^{3t}$ is a solution to the ordinary differential equation $\frac{dx}{dt} = 3x$ with $x(0) = 7$ then you could easily check that the claim was true by doing two things:

1. Check that the proposed solution matches the initial condition. In this example we see that $x(0) = 7e^{3 \cdot 0} = 7$ ✓.
2. Check that the function satisfies the differential equation. In other words, substitute the function $x(t)$ into the differential equation $x' = 3x$ and verify that the equal sign is actually true. In this case, $x' = 3 \cdot 7e^{3t} = 3x$ ✓.

Checking a solution to a differential equation amounts to substituting the function into the differential equation and the associated conditions and verifying that everything is true. Let's do the same for some partial differential equations.

Exercise 6.2. Consider the PDE $u_t = Du_{xx}$ where $u(t, x)$ is the temperature of a long thin metal rod at time t (in seconds) and spatial location x (in meters).

Note: the symbol u_t is quick shorthand for the partial derivative $\frac{\partial u}{\partial t}$ and u_{xx} is a quick shorthand for the second partial derivative $\frac{\partial^2 u}{\partial x^2}$.

a. What are the units of the constant D?
b. For each of the following functions, test whether it is an analytical solution to this PDE by taking the first derivative with respect to time, the second derivative with respect to position, and substituting them into this equation to see if we get an identity (a true statement). If $D = 3$, which of these functions is a solution? Be able to defend your answer.
   i. $u(t, x) = 4x^3 + 6t^2$
   ii. $u(t, x) = 7x + 5$
   iii. $u(t, x) = 8x^2 t$

iv. $u(t,x) = e^{3t+x}$
v. $u(t,x) = 6e^{3t+x} + 5x - 2$
vi. $u(t,x) = e^{-3t} + \sin(x)$
vii. $u(t,x) = e^{3t}\sin(x)$
viii. $u(t,x) = e^{-3t}\sin(x)$
ix. $u(t,x) = 5e^{-3t}\sin(x) + 6x + 7$
x. $u(t,x) = -4e^{-3t}\sin(x) + 3t + 2$
xi. $u(t,x) = e^{-2t}\sin(3x)$
xii. $u(t,x) = e^{-12t}\cos(3x)$
xiii. $u(t,x) = e^{-12t}\cos(3x) + 4x^2 + 8$
xiv. $u(t,x) = e^{-75t}\cos(5x)$
xv. $u(t,x) = 9e^{-75t}\cos(5x) + 2x + 7$

Exercise 6.3. Consider the PDE $u_t = Du_{xx}$, and suppose that $D = 4$. For each of the following functions, find the value of the parameter a that will make the function solve the PDE, by taking derivatives and substituting them into the equation.

a. $u(t,x) = 6e^{-8t}\sin(ax)$
b. $u(t,x) = -5e^{38t}\cos(ax)$
c. $u(t,x) = 3e^{at}\sin(5x)$
d. $u(t,x) = 7e^{at}\cos(2x)$
e. $u(t,x) = ae^{-36t}\cos(3x)$
f. $u(t,x) = ae^{-4t}\cos(6x)$

Exercise 6.4. Consider again the PDE $u_t = Du_{xx}$. Is the function $u(t,x) = x^2 - t^3$ a valid solution for this differential equation? If so, what is the value of the constant k?

a. Calculate u_t and u_{xx}

$$u_t = \_\_\_\_\_ \quad \text{and} \quad u_{xx} = \_\_\_\_\_$$

b. If the differential equation $u_t = ku_{xx}$ is to be satisfied then what equation must be true?

$$\_\_\_\_\_ = \_\_\_\_\_$$

c. Is there a single value of D that makes the PDE true with this proposed solution? If so, then we must have a solution to the PDE, if not then we must not have a solution to the PDE. Is $u(t,x)$ a solution to the equation $u_t = Du_{xx}$?

Exercise 6.5. The PDE $u_t = Du_{xx}$ can be seen as asking two questions: (1) the time derivative of the function $u(t,x)$ is related to the function u itself, and (2) the second spatial derivative of the function $u(t,x)$ is related to the function u itself.

a. What sort of function has the property that when you take the derivative you get a scaled version of the function back.
b. What sort of function has the property that when you take two derivatives you get a scaled version of the function back.
c. Based on your answers to parts (a) and (b), propose a function that might be a solution to the PDE.

Exercise 6.6. Is the function $u(t,x) = e^{-0.2t}\sin(\pi x)$ a solution to the PDE $u_t = Dku_{xx}$? If this function is a solution to the PDE then what is the associated value of D?

Exercise 6.7. Is a scalar multiple of the function in the previous exercise also a solution to the PDE $u_t = Du_{xx}$ with the exact same value for k? Will this always be true? That is, if we have one solution $u(t,x)$ to the PDE $u_t = Du_{xx}$ then will $cu(t,x)$ be another solution for any real number c?

Exercise 6.8. When we studied ODEs we always had a starting point for a solution – the initial condition. In the case of a PDE we also need to have an initial condition, but the initial condition is associated with every point in the spatial domain. Hence, the initial condition is actually a function of x. In the previous exercise you found the $u(t,x) = e^{-0.2t}\sin(\pi x)$ is a solution to the PDE $u_t = Du_{xx}$. What is the initial condition that this solution satisfies? In other words, what is the function $u(t,x)$ at time $t = 0$?

Exercise 6.9. Since we have both temporal and spatial variables in PDEs it stands to reason that we need conditions on both variables in order to get a unique solution to the PDE. For the PDE $u_t = Du_{xx}$ we already saw that $u(t,x) = e^{-0.2t}\sin(\pi x)$ is a solution to the PDE and the initial condition for that solution is $u(0,x) = \sin(\pi x)$. If we are solving the PDE on the domain $x \in [0,1]$ then what are the conditions that holds for all time at the points $x = 0$ and $x = 1$? These conditions are called **boundary conditions**.

Exercise 6.10 (Visualizing Solutions to PDEs)**.** Solutions to PDEs are multivariable functions. In the previous few problems we have examined the heat equation $u_t = Du_{xx}$. The function u is a function of time, t, and one spatial variable, x. We have several choices when we make a plot of this type of function. Implement and complete the blocks of code below to get three different visualizations of the solution $u(t,x) = e^{-0.2t}\sin(\pi x)$ on the domain $t \in [0,1]$ and $x \in [0,1]$.

a. The first idea is to show several discrete snapshots of time and to arrange the plots in an array so we can read from left to right to see the evolution in time.

```
import numpy as np
import matplotlib.pyplot as plt
u = lambda t, x: np.exp(-0.2*t) * np.sin(np.pi*x)
x = # code that gives 100 equally spaced points from 0 to 1
t = # code that gives 16 equally spaced points from 0 to 10
fig, ax = plt.subplots(nrows=4,ncols=4)
counter = 0 # this counter will count through the times
for n in range(4):
    for m in range(4):
        ax[n,m].plot(??? , ???, 'b') # plot x vs u(t[counter],x)
        ax[n,m].grid()
        ax[n,m].set_ylim(0,1) # same axis for every plot
        ax[n,m].set_xlabel('x')
        ax[n,m].set_ylabel('u')
        ax[n,m].set_title("time="+np.str(t[counter]))
        counter += 1 # increment the counter
fig.tight_layout()
plt.show()
```

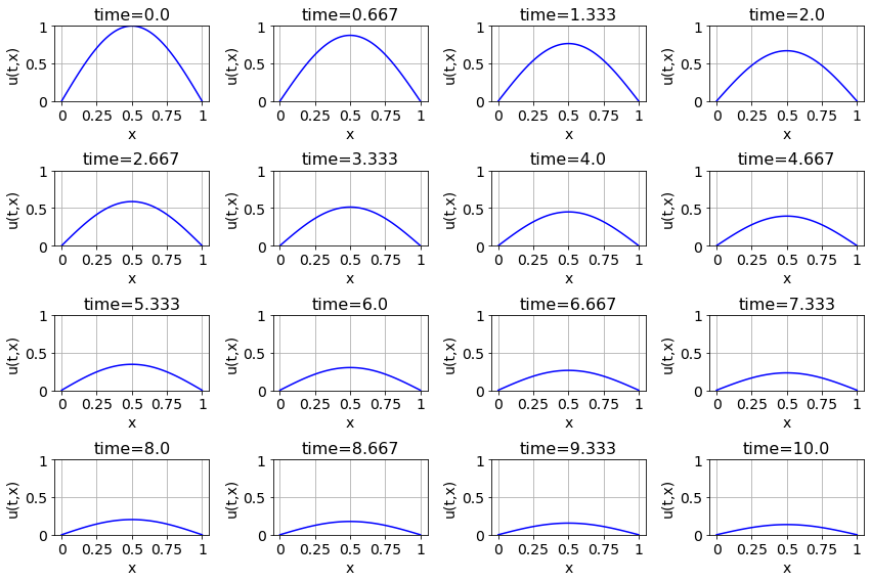


Figure 6.1: Time evolution of a solution to the PDE

b. A second idea for plotting the solution to a PDE is to give an interactive plot where we can use a slider to advance (or reverse) time.

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

```
from ipywidgets import interactive

u = lambda t, x: np.exp(-0.2*t) * (1*np.sin(1*np.pi*x))
x = np.linspace(0,1,100)

def plotter(T):
  plt.plot(x , u(T,x), 'b')
  plt.grid()
  plt.ylim(0,1)
  plt.show()

interactive_plot = interactive(plotter, T=(0,20,0.1))
interactive_plot
```

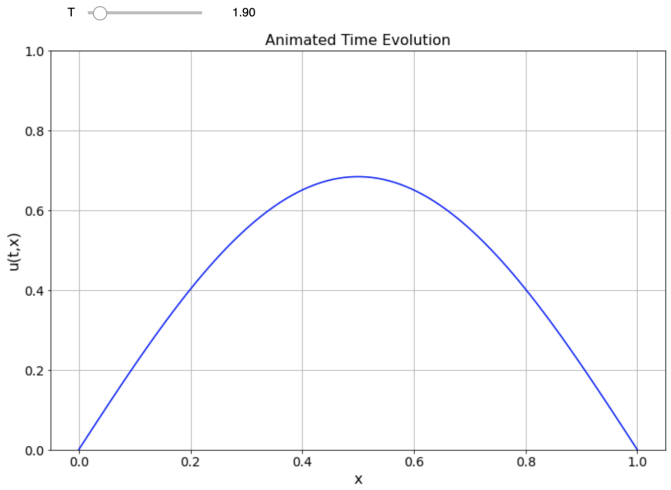


Figure 6.2: Snapshot of animated time evolution of a solution to the PDE

c. A third idea for plotting the solution to a PDE is to create a three dimensional plot with time on one axis, x on the second axis, and u on the vertical axis. To read this plot we start our eyes at $t = 0$ and then scan down the t axis. In this way we can see the whole time evolution of the PDE in one plot without animation. Of course, if the PDE had two or more spatial dimensions plus time then this sort of plot would not be feasible.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(10,8))
ax = fig.gca(projection='3d') # gca stands for "Get Current Axis"
```

```
u = lambda t, x: np.exp(-0.2*t)*np.sin(np.pi*x)
x = np.linspace(0,1,25)
t = np.linspace(0,10,25)
T, X = np.meshgrid(t,x)
ax.plot_wireframe(T,X,u(T,X))
ax.set_xlabel('time')
ax.set_ylabel('x')
ax.set_zlabel('u(t,x)')
plt.show()
```

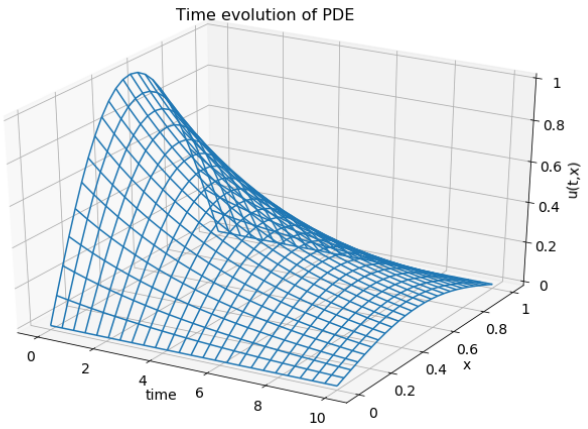


Figure 6.3: 3D plot showing the time evolution of the solution to the PDE.

Exercise 6.11. In the previous problem you built several plots of the function $u(t,x) = e^{-0.2t}\sin(\pi x)$ as a solution to the heat equation $u_t = Du_{xx}$.

a. Based on the plots, why do you think the equation $u_t = Du_{xx}$ called the "heat equation"? That is, why do the solutions look like dissipating heat?

b. What is the limit

$$\lim_{t\to\infty} u(t,x)?$$

Explain why your answer makes sense if we are solving an equation, called the "heat equation", that models the diffusion of heat through an object. Hint: think of this *object* as a long thing metal rod and take note that the boundary conditions are both 0.

Exercise 6.12. Propose another solution to the PDE $u_t = Du_{xx}$ that exactly matches the boundary conditions $u(t,0) = 0$ and $u(t,1) = 0$ for all time t AND

exactly the same value for D as with the function $u(t,x) = e^{-0.2t}\sin(\pi x)$. What is the new initial condition associated with your new solution?

Hint: You may want to start with a function of the form $u(t,x) = e^{at}\sin(bx)$ and then determine values of a and b that will satisfy all of the required conditions.

Exercise 6.13. We now have two solutions to the PDE $u_t = Du_{xx}$ that satisfy both the PDE and the boundary conditions $u(t,0) = 0$ and $u(t,1) = 0$.

a. Prove that the sum of the two solutions also satisfies the PDE and the same boundary conditions? If so, then the sum appears to be *another* valid solution to the PDE.
b. What is the initial condition associated with the new solution you found in part (a)?
c. Use the code that you built above to show the time evolution of your new solution.

Let's take stock of what we've investigated thus far.

Theorem 6.1. *If $u_0(t,x)$ and $u_1(t,x)$ are both solutions to the PDE $u_t = Du_{xx}$ matching the same boundary conditions $u(t,0) = u(t,1) = 0$ then for real scalars c_0 and c_1 the function $c_0u_0(t,x) + c_1u_1(t,x)$ is another solution to the PDE matching the same boundary conditions but perhaps having a different initial condition.*

Exercise 6.14. Prove the previous theorem. Then extend the theorem to show that if there are many functions that satisfy $u_t = Du_{xx}$ and the boundary conditions $u(t,0) = u(t,1) = 0$ then the sum of all of the functions is also a solution and also satisfies the boundary conditions.

Exercise 6.15. Propose several solutions to the PDE $u_t = Du_{xx}$ with the boundary conditions $u(t,0) = 0$ and $u_x(t,1) = 0$. That is to say that the function $u(t,x)$ is 0 at $x = 0$ and the derivative of u with respect to x at $x = 1$ is 0 (there is a horizontal tangent line to the function u at $x = 1$ for all times t). Then use your plotting code to verify that your solution satisfies the boundary conditions and visually shows the diffusion of heat as time evolves.

At this point we have a good notion of what the solutions to the PDE $u_t = Du_{xx}$ look and behave like. Now let's ramp this up to two spatial dimensions.

Exercise 6.16. Leverage what you learned in the previous exercises to propose a function $u(t,x,y)$ that solves the equation

$$u_t = D\left(u_{xx} + u_{yy}\right)$$

on the domain $x \in [0,1]$ and $y \in [0,1]$ with $D = 1$, the boundary conditions $u(t,0,y) = 0$, $u(t,1,y) = 0$, $u(t,x,0) = u(t,x,1) = 0$, and the initial condition $u(0,x,y) = \sin(2\pi x)\sin(2\pi y)$ (showing in Figure 6.4. Then use the code below to show the time evolution of your solution.

```
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interactive

u = lambda t, x, y: # your function goes here
x, y = np.meshgrid( np.linspace(0,1,25), np.linspace(0,1,25) )

def plotter(T):
    fig = plt.figure(figsize=(15,12))
    ax = fig.gca(projection='3d')
    z = u(T,x,y)
    ax.plot_surface(x,y,z)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('u(t,x,y)')
    ax.set_zlim(0,1)
    plt.show()

interactive_plot = interactive(plotter, T = (0,10,0.1) )
interactive_plot
```

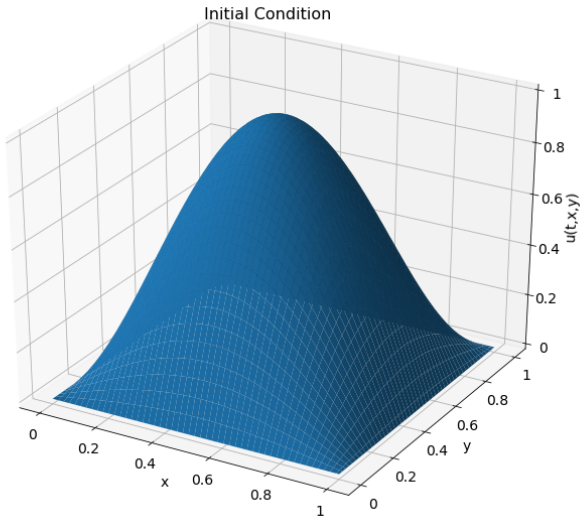


Figure 6.4: Initial condition

Exercise 6.17. Prove that the function $u(t, x, y) = e^{-0.2t} \sin(\pi x) \sin(\pi y)$ is a solution to the two dimensional heat equation $u_t = D(u_{xx} + u_{yy})$. Determine the value of D for this particular solution. What are the boundary conditions and the initial condition?

Let's move on to a different PDE: the wave equation.

Exercise 6.18. Consider the wave equation

$$\frac{\partial^2 u}{\partial t^2} = c \frac{\partial^2 u}{\partial x^2}.$$

a. What are the units of c?
b. Reading from left-to-right, the partial differential equation says that the second derivative of some function of t is related to that same function. If you had to guess the *type* of function, what would you guess and why?
c. Reading from right-to-left, the partial differential equation says that the second derivative of some function of x is related to that same function. If you had to guess the *type* of function, what would you guess and why?
d. Based on your guesses from parts (a) and (b), what type of function would think is a reasonable solution for the differential equation? Why?
e. If $u(0, x) = \sin(\pi x)$ is the initial condition for the PDE and the boundary conditions are $u(t, 0) = u(t, 1) = 0$ then propose a solution that matches these conditions and make plots showing how the solution behaves over time.

Theorem 6.2. *If the function $u(t, x)$ solves the 1D wave equation $u_{tt} = cu_{xx}$ then $u(t, x)$ likely has the functional form*

$$u(t, x) = \_\_\_\_\_.$$

Exercise 6.19. Prove your hypotheses from the previous theorem.

Exercise 6.20. Make several plots of your solution showing the time evolution of the function. Examples of the plots are shown in Figures 6.5 and 6.6. Your plots may look different given the oscillation period and the initial condition.

Exercise 6.21. If $u_0(t, x)$ and $u_1(t, x)$ are both solutions the wave equation $u_{tt} = cu_{xx}$ matching boundary conditions $u(t, 0) = u(t, 1) = 0$, then is a linear combination of u_0 and u_1 also a solution that matches the particular boundary conditions?

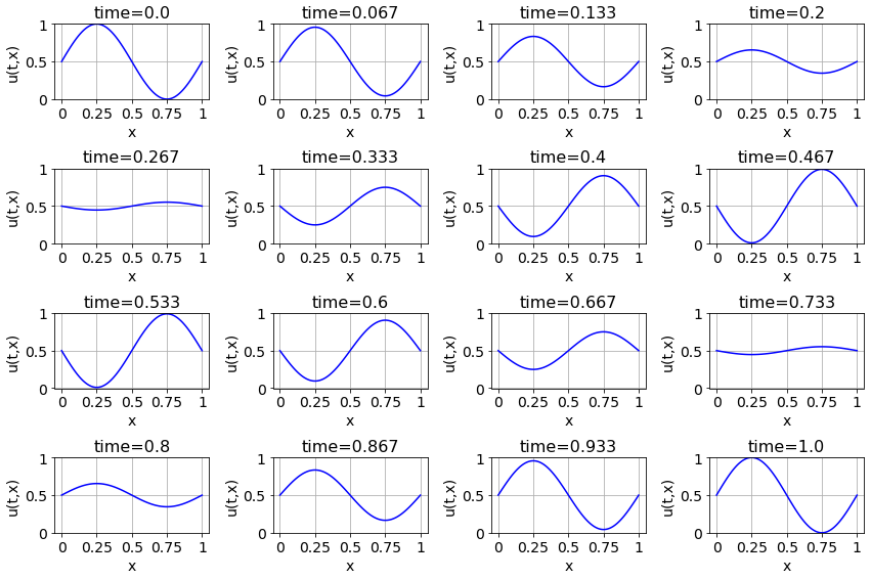


Figure 6.5: Time evolution of a solution to the wave equation.

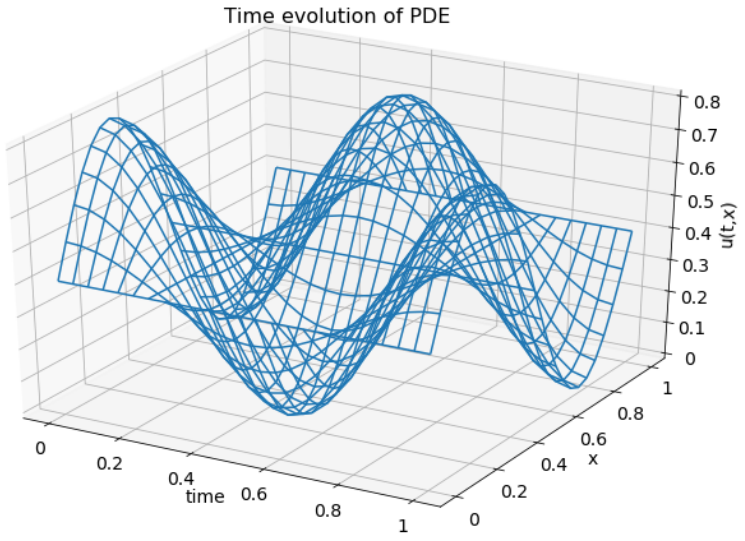


Figure 6.6: 3D plot of the time evolution of a solution to the wave equation.

Exercise 6.22. Consider the wave equation $u_{tt} = c(u_{xx} + u_{yy})$ where $u(x, y, t)$ is the height (in centimeters) of a wave at time t in seconds and spatial location (x, y) (each in centimeters).

a. What are the units of the constant c?
b. For each of the following functions, test whether it is an analytical solution to this PDE by substituting the derivatives into the equation. If $c = 2$, which of these functions is a solution?
   i. $u(t, x, y) = 3x + 2y + 5t - 6$
   ii. $u(t, x, y) = 3x^2 + 2y^2 + 5t^2 - 6$
   iii. $u(t, x, y) = \sin(2x) + \cos(3y) + \sin(4t)$
   iv. $u(t, x, y) = \sin(2x)\cos(3y)\sin(4t)$
   v. $u(t, x, y) = \sin(3x)\cos(4y)\sin(10t)$
   vi. $u(t, x, y) = -6\sin(3x)\cos(4y)\sin(10t) + 2x - 3y + 9 - 12$
   vii. $u(t, x, y) = \cos(7x)\cos(3y)\cos(12t)$
   viii. $u(t, x, y) = \cos(5x)\sin(12y)\cos(26t)$
c. Make plots of the time evolution of the solutions from part (b). What phenomena do you observe in the plots?

Theorem 6.3. *If the function $u(t, x, y)$ solves the 2D wave equation $u_{tt} = c\,(u_{xx} + u_{yy})$ then $u(t, x, y)$ likely has the functional form*

$$u(t, x, y) = \_\_\_\_\_.$$

Exercise 6.23. Prove your hypotheses from the previous theorem.

Exercise 6.24. Propose a solution to the wave equation $u_{tt} = cu_{xx}$ where $u(t, 0) = 0$ and $u_x(t, 1) = 0$.

At this point we have only examined two PDEs, the heat and wave equations, and have proposed possible forms of the analytic solutions. These two particular PDEs have nice analytic solutions in terms of exponential and trigonometric functions so it isn't terribly challenging to *guess* at the proper functional forms of the solutions. However, if we were to change the initial conditions, boundary conditions, or the differential equation by just a bit it may be more challenging to propose analytic solutions. It is not the purpose of this chapter to give a complete treatment of the analytic solutions to PDEs (not even the heat or wave equations). The purpose of what we just did was to build some intuition about the types of behaviors that we should expect from these prototypical PDEs. This way we will be able to determine if our numerical solutions in future sections are reasonable or not.

6.2 Boundary Conditions

When we were solving ODEs we typically needed initial conditions to tell us where the solutions starts at time $t = 0$. Since PDEs require both spatial and temporal information we need to tell the differential equation how to behave both at time zero AND on the boundaries of the domain.

Definition 6.2. Let's say that we want to solve a PDE with variable t and x on the domain $x \in [0, 1]$.

- The **initial condition** is a function $\eta(x)$ where $u(0, x) = \eta(x)$. In other words, we are dictating the value of u at every point x at time $t = 0$.
- The boundary conditions are restrictions for how the solution behaves at $x = 0$ and $x = 1$ (for this problem).
  - If the value of the solution u at the boundary is either a fixed value or a fixed function of time then we call the boundary condition a **Dirichlet boundary condition.** For example, $u(t, 0) = 1$ and $u(t, 1) = 5$ are Dirichlet boundary conditions for this domain. Similarly, the conditions $u(t, 0) = 0$ and $u(t, 1) = \sin(100\pi t)$ are also Dirichlet boundary conditions. Dirichlet boundary conditions give the exact value of u the the boundary points.
  - If the value of the solution u depends on the flux of u at the boundary then we call the boundary condition a **Neumann boundary condition.** For example, $\frac{\partial u}{\partial x}(t, 0) = 0$ and $\frac{\partial u}{\partial x}(t, 1) = 0$ are Neumann boundary conditions. They state that the flux of temperature is fixed at the boundaries.

Let's play with a couple problems that should help to build your intuition about boundary conditions in PDEs. Again, we will do this graphically instead of numerically.

Exercise 6.25. Consider solving the heat equation $u_t = Du_{xx}$ in 1 spatial dimension.

a. If a long thin metal rod is initially heated in the middle and the temperature at the ends of the rod is held fixed at 0 then the heat diffusion is described by the heat equation. What type of boundary conditions do we have in this setup? How can you tell? Draw a picture showing the expected evolution of the heat equation with these boundary conditions.

b. What if we take the initial condition for the 1D heat equation to be $u(0, x) = \cos(2\pi x)$ and enforce the conditions $\left.\frac{\partial u}{\partial x}\right|_{x=0} = 0$ and $u(t, 1) = 1$. What types of boundary conditions are these? Draw a collection of pictures showing the expected evolution of the heat equation with these boundary conditions.

Exercise 6.26. Consider solving the wave equation $u_{tt} = cu_{xx}$ in 1 spatial dimension.

a. If a guitar string is pulled up in the center and held fixed at the frets then the resulting vibrations of the string are described by the wave equation. What type of boundary conditions do we have in this setup? How can you tell? Draw a picture showing the expected evolution of the heat equation with these boundary conditions.
b. What if we take the initial condition for the 1D wave equation to be $u(0, x) = \cos(2\pi x)$ and enforce the conditions $\frac{\partial u}{\partial x}\Big|_{x=0} = 0$ and $u(t, 1) = 1$. What types of boundary conditions are these? Draw a collection of pictures showing the expected evolution of the wave equation with these boundary conditions.

An important lesson when solving partial differential equations is that if you get the boundary conditions wrong then the solution to your problem is meaningless. The next two problems should help you to understand some of the basic scenarios that we might wish to solve with the heat and wave equation.

Exercise 6.27. For each of the following situations propose meaningful boundary conditions for the 1D or 2D heat equation.

a. A thin metal rod 1 meter long is heated to 100°C on the left end and is cooled to 0°C on the right end. We model the heat transport with the 1D heat equation $u_t = Du_{xx}$. What are the appropriate boundary and initial conditions?
b. A thin metal rod 1 meter long is insulated on the left end so that the heat flux through that end is 0. The rod is held at a constant temperature of 50°C on the right end. We model the heat transport with the 1D heat equation $u_t = Du_{xx}$. What are the appropriate boundary conditions?
c. In a soil-science lab a column of packed soil is insulated on the sides and cooled to 20°C at the bottom. The top of the column is exposed to a heat lamp that cycles periodically between 15°C and 25°C and is supposed to mimic the heating and cooling that occurs during a day due to the sun. We model the heat transport within the column with the 1D heat equation $u_t = Du_{xx}$. What are the appropriate boundary conditions?
d. A thin rectangular slab of concrete is being designed for a sidewalk. Imagine the slab as viewed from above. We expect the right-hand side to be heated to 50°C due to radiant heating from the road and the left-hand side to be cooled to approximately 20°C due to proximity to a grassy hillside. The top and bottom of the slab are insulated with a felt mat so that the flux of heat through both ends is zero. We model the heat transport with the 2D heat equation $u_t = D(u_{xx} + u_{yy})$. What are the appropriate boundary conditions?

Exercise 6.28. For each of the following situations propose meaningful boundary conditions for the 1D and 2D wave equation.

a. A guitar string is held tight at both ends and plucked in the middle. We model the vibration of the guitar string with the 1D wave equation $u_{tt} = cu_{xx}$. What are the appropriate boundary conditions?
b. A rope is stretched between two people. The person on the left holds the rope tight and doesn't move. The person on the right wiggles the rope in a periodic fashion completing one full oscillation per second. We model the waves in the rope with the 1D wave equation $u_{tt} = cu_{xx}$. What are the appropriate boundary conditions?
c. A rubber membrane is stretched taught on a rectangular frame. The frame is held completely rigid while the membrane is stretched from equilibrium and then released. We model the vibrations in the membrane with the 2D wave equation $u_{tt} = c(u_{xx} + u_{yy})$. What are the appropriate boundary conditions?

6.3 The Heat Equation

In this section we'll use a technique called **the finite difference method** to build numerical approximations to the heat equation

$$u_t = D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) + f(x).$$

This equation governs the process of heat diffusion in a three dimensional object make of one material (e.g. a block of aluminum).

In one spatial dimension the heat equation can be written as $u_t = Du_{xx} + f(x)$ and in two spatial dimensions it can be written as $u_t = D\left(u_{xx} + u_{yy}\right) + f(x, y)$. The function f is called a forcing term and in the case of thermal diffusion it is an external source of heat in the system. We'll let $f(x) = 0$ for the majority of this section for simplicity, but you can modify any of the code that you write in this section to include a forcing term.

For the sake of simplicity we will start by considering the time dependent heat equation in 1 spatial dimension with no external forcing function

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}.$$

The constant D is the called the diffusivity (the rate of diffusion) so in terms of physical problems, if D is small then the diffusion occurs slowly and if D is large then the diffusion occurs quickly. Just as we did in Chapter 3 to approximate derivatives and integrals numerically, and also in 5 to approximate solutions to

ODEs, we will start by partitioning the domain into finitely many pieces and we will partition time into finitely many pieces.

Exercise 6.29. In 1 spatial dimension, the heat equation is simply

$$u_t = Du_{xx}.$$

We want to build a numerical approximation to the function $u(t, x)$ for a given collection of initial and boundary conditions.

Frist we need to introduce some notation for the numerical solution. As you'll see, there is a lot to keep track of in numerical PDEs so careful index and well-chosen notation is essential. Let U_i^n be the approximation of the solution to $u(t, x)$ at the point $t = t_n$ and $x = x_i$ (since we have two variables we need to 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 3 and refresh your memory for how we build approximations of derivatives.

a. Give an approximation of u_t similar to Euler's method

$$u_t \approx \frac{???-???}{???}.$$

b. Give an approximation of u_{xx} using the approximation for the second derivative from Chapter 3

$$u_{xx} \approx \frac{???-???+???}{???}.$$

c. Put your answers from parts (a) and (b) together using the 1D heat equation

$$\frac{???-???}{\Delta t} = D\left(\frac{???-???+???}{\Delta x^2}\right).$$

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}:

$$U_i^{n+1} = ??? + \frac{D\Delta t}{\Delta x^2}\left(???-???+???\right).$$

The iterative scheme which you just derived is called a **finite 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. What is the expected order of the error for the approximation of the time derivative in the finite difference scheme from part (d)?
f. What is the expected order of the error for the approximation of the spatial second derivative in the finite difference scheme from part (d)?
g. The nuemrical errors made by using the finite 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. The total error is the sum of the two errors. Fill in the question marks in the powers of the following expression:

$$\text{Numerical Error} = \mathcal{O}(\Delta t^{???}) + \mathcal{O}(\Delta x^{???}).$$

h. Explain what the result from part (g) 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 3). 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 6.7 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} are being used at time step t_n to calculate U_i^{n+1}. We will build similar stencil diagrams for other finite difference schemes throughout this chapter.

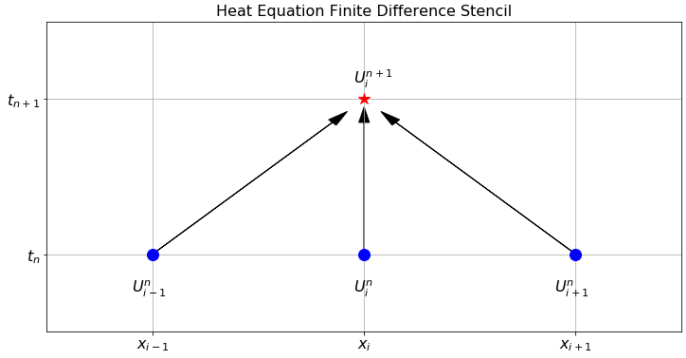


Figure 6.7: The finite difference stencil for the 1D heat equation.

Exercise 6.30. Now we want to implement your answer to part (d) of the previous exercise to approximate the solution to the following problem:

Solve: $u_t = 0.5u_{xx}$ with $x \in (0,1)$, $u(0,x) = \sin(2\pi x)$, $u(t,0) = 0$, and $u(t,1) = 0$.

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 an array of times starting at t=0 and ending
# at t=1.  Be sure that you use many points in the partition of
# the time domain.  Be sure to either specify or calculate the
# value of Delta t.

# Write code to give an array of x values starting at x=0 and
# ending exactly at x=1. This is best done with the np.linspace()
# command since you can guarantee that you end exactly at x=1.
# Be sure to either specify or calculate the value of Delta x as
# part of your code.

# The next two lines build two parameters that are of interest
# for the finite difference scheme.
D = 0.5 # The diffusion coefficient for the heat equation given.
# The coefficient "a" appears in the finite 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)` versus `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 initial 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 plot the solutions. One way to do this is with the `ipywidgets.interactive` tool. We first need to create a function which

returns a plot at a particular time step. Then we call the function inside the `interactive` function.

```
def plotter(Frame):
    plt.plot(x,U[Frame,:],'b')
    plt.grid()
    plt.ylim(-1,1)
    plt.show()
interactive_plot = interactive(plotter, Frame=(0,len(t)-1,1))
interactive_plot
```

Note: If you don't want to do an interactive plot then you can produce several snapshots of the solutions with the following code.

```
for Frame in range(0,len(t),20): # built every 20th frame, for example
    plotter(Frame)
```

Exercise 6.31. You may have found that you didn't get a sensible solution out 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 finite difference solution to the heat equation, and furthermore that there is a cutoff for a below which the finite difference scheme will be stable. 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 $a = D\frac{\Delta t}{\Delta x^2} <$ _____ *then the finite difference solution for the 1D heat equation is stable. Otherwise it is unstable.*

Exercise 6.32. The 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 $t \in (0, 0.5)$ subject to the initial condition $u(0, x) = \sin(\pi x)$ and the boundary conditions $u(t, 0) = u(t, 1) = 0$.

a. Prove 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 finite difference solution to this heat equation. Plot your results on top of the analytic solution from part (a).

c. Now let's change the initial condition to $u(0, x) = \sin(\pi x) + 0.1\sin(100\pi x)$. Prove that the function $u(t, x) = e^{-\pi^2 t}\sin(\pi x) + 0.1e^{-10^4\pi^2 t}\sin(100\pi x)$ is a solution to this heat equation, matches this new initial condition, and matches the boundary conditions.

d. Pick values of Δt and Δx so that you can get a stable finite difference solution to this heat equation. Plot your results on top of the analytic solution from part (c).

Exercise 6.33. In any initial and boundary value problem such as the heat equation, the boundary values can either be classified as Dirichlet or Neumann type. In Dirichlet boundary conditions the values of the solution at the boundary are dictated specifically. So far we have only solved the heat equation with Dirichlet boundary conditions. In contrast, Neumann boundary conditions dictate the flux at the boundary instead of the value of the solution. 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

$$u_x(t_n, 0) \approx \frac{U_1^n - U_0^n}{\Delta x}.$$

If we set this approximation to 0 (since $u_x(t,0) = 0$) and solve for U_0^n we get an additional contraint at every time step of the numerical solution to the heat equation.
c. Modify your 1D heat equation code to implement this Neumann boundary condition. Give plots that demonstrate that the Neumann boundary is indeed satisfied.

Exercise 6.34. Modify your 1D heat equation code to solve the following problems. For each be sure to classify the type of boundary conditions given. Notice that we are now using initial and boundary conditions where it would be quite challenging to built the analytic solution so we weill only show numerical solutions. Be sure that you choose Δt and Δx so that your solution is stable.

a. Solve $u_t = 0.5u_{xx}$ with $x \in (0,1)$, $u(0,x) = x^2$, $u(t,0) = 0$ and $u(t,1) = 1$.
b. Solve $u_t = 0.5u_{xx}$ with $x \in (0,1)$, $u(0,x) = 1 - \cos(\pi x/2)$, $u_x(t,0) = 0$ and $u(t,1) = 1$.
c. Solve $u_t = 0.5u_{xx}$ with $x \in (0,1)$, $u(0,x) = \sin(2\pi x)$, $u(t,0) = 0$ and $u(t,1) = \sin(5\pi t)$.
d. Solve $u_t = 0.5u_{xx} + x^2$ with $x \in (0,1)$, $u(0,x) = \sin(2\pi x)$, $u(t,0) = 0$ and $u(t,1) = 0$.

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,

$$u_t = D\left(u_{xx} + u_{yy}\right).$$

Exercise 6.35. Let's 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:

$$u_t(t_{n+1}, x_i, y_j) \approx \frac{U_{i,j}^{n+1} - U_{i,j}^n}{\Delta t}.$$

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 3 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's 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.

$$\frac{U_{i,j}^{n+1} - U_{i,j}^n}{\Delta t} = D\left(\frac{???+???-???+???+???}{???}\right).$$

d. Now solve your result from part (c) for $U_{i,j}^{n+1}$. Your answer is the explicit finite differene scheme for the 2D heat equation.

$$U_{i,j}^{n+1} = U_{???,???}^{???} + \frac{D \cdot ???}{???}(???+???-???+???+???)$$

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 6.8 shows the stencil for the finite 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}.

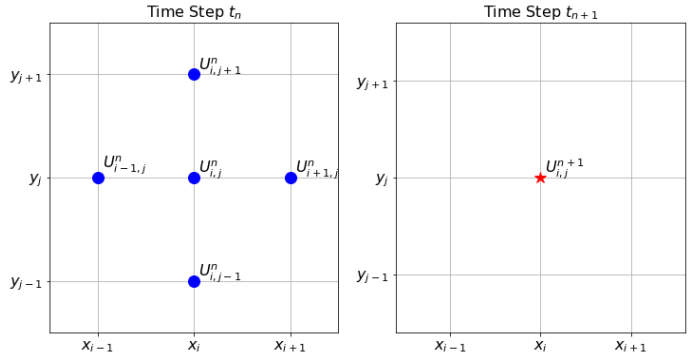


Figure 6.8: The finite difference stencil for the 2D heat equation.

Exercise 6.36. 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 $u(0, x, y) = \sin(\pi x)\sin(\pi y)$, homogeneous Dirichlet boundary conditions, and $D = 1$.[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 of a surface plot
from ipywidgets import interactive

# Write code to build a linearly spaced array of x values starting at 0
# and ending at exactly 1
```

[1] Take note that homogeneous boundary conditions are "0", so saying that a PDE has homogeneous Dirichlet boundary conditions on this domain means that $u(t, x, 0) = u(t, x, 1) = u(t, 0, y) = u(t, 1, y) = 0$.

```
x = # your code here
y = x # This is a step that allows for us to have y = x and hence 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 0.25.
# 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.
X, Y = np.meshgrid(x,y)

# 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 finite element methods. 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. Again we use the `ipywidgets.interactive` tool to build an interactive plot with time as

the slider.

```
def plotter(Frame):
  fig = plt.figure(figsize=(12,10))
  ax = fig.gca(projection='3d')
  ax.plot_surface(X,Y,U[Frame,:,:], cmap=cm.coolwarm)
  ax.set_zlim(0,1)
  plt.show()

interactive_plot = interactive(plotter, Frame=(0,len(t)))
interactive_plot
```

Fill in all of the holes in the code and verify that your solution appears to solve a heat dissipation problem.

Theorem 6.4. *In order for the finite difference solution to the 2D heat equation on a square domain to be stable then we need* $D\Delta t/\Delta x^2 <$ _____.

Experiment with several parameters to imperically determine the bound.

Exercise 6.37. 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 6.38. 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.

6.4 Analysis of the Heat Equation

Exercise 6.39 (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'll solve the PDE $u_t = u_{xx}$ on the domain $x \in [0,1]$ with initial conditions $u(0,x) = \sin(\pi x)$ then the analytic solution is $u(t,x) = e^{-\pi^2 t}\sin(\pi x)$. To build the spatial and temporal domains we will use `x = np.linspace(0,1,21)` and `t = np.linspace(0,0.25,101)`. 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 finite differences 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 as the time step exceeds 30.

Exercise 6.40. Solve the 2D heat equation on the unit square with the following parameters:

- 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?

Exercise 6.41. What you saw in the previous two exercises is an example of a **sawtooth error** that occurs when a numerical solution technique for a PDE is unstable. Propose a conjecture for why this type of error occurs.

Theorem 6.5. *Let's summarize the stability criteria for the finite difference solutions to the heat equation.*

- *In the 1D heat equation the finite difference solution is stable if* $D\Delta t/\Delta x^2 <$ _____.
- *In the 2D heat equation the finite difference solution is stable if* $D\Delta t/\Delta x^2 <$ _____ *(assuming a square domain where* $\Delta x = \Delta y$*)*
- *Propose a stability criterion for the 3D heat equation.*

Exercise 6.42. Rewrite your finite difference code so that it produces an error message when the parameters will result in an unstable finite difference solution. Do the same for your 2D heat equation code.

Next we'll examine the order of the error for the 1D heat equation.

Exercise 6.43. In the finite difference schemes for the 1D heat equation we have built methods that are first order in time and second order in space. That is to say

$$\text{Numerical Error} = \mathcal{O}(\Delta t) + \mathcal{O}(\Delta x^2).$$

We want to know what this means.

a. Build a Python function that accepts an array of times, an array of x values, the boundary conditions, and the initial condition and returns the array U for the finite difference approximation of the 1D heat equation.

```
def buildU(t,x,leftBC,rightBC,IC):
  U = np.zeros( (len(t),len(x)) )
  U[:,0] = ???
  U[:,-1] = ???
  U[0,:] = ???
  for n in range(len(t)-1):
```

```
    # build the finite difference solution here
  return U
```

b. Build a Python function that accepts an array U, an array of times, an array of x values, a specific time step N, and a `lambda` function for the analytic solution to the heat equation and returns the relative normed error between the approximation at time step N and the approximate solution at time step N. Explain exactly what this function is meant to measure.

```
def errorT(U,t,x,TimeStep,exact):
  approxSoln = U[N,:]
  exactSoln = exact(t[N],x)
  return( np.linalg.norm(approxSoln - exactSoln) / np.linalg.norm(exactSoln))
```

c. If we start with the initial condition $u(0,x) = \sin(\pi x)$ then the exact solution the heat equation $u_t = u_{xx}$ is $u(t,x) = e^{-\pi^2 t}\sin(\pi t)$. Write a loop that builds finite difference solutions to this heat equation with different values of Δt ranging from 10^{-1} to 10^{-7} and a fixed value of Δx. On a log-log scale, plot the value of Δt against the value of the relative normed error at a specific time step of your choosing (don't choose time step 0). What do you see in the log-log plot? Given a specific time, what will happen to the error in your finite difference solution when you divide your value of Δt by 10?

d. Now we want to repeat the process for the spatial step. Fix a very small value of Δt and build a function that measures the relative normed error between the full time evolution of the solution and the analytic solution at a fixed spatial point (e.g. the center point in the domain). Make a log-log plot with the value of Δx on the horizontal axis and the value of the relative normed error on the vertical axis.

e. What does the plot you built in part (d) tell you? If you were to divide your value of Δx by 10 then what would we expect to divide the relative normed error by?

The next problem addresses the issue of stability in solving the heat equation with the finite difference method. There are MANY different techniques for dealing with stability issues in numerical PDEs, and, as we'll see, the methods can get quite complicated. However, if we ignore the slightly more complicated solver, having a solver which is stable no matter the value of Δt and Δx could save us substantial computational time.

Exercise 6.44. The instabilities of the heat equation with and Euler-type time discretization and a central differencing scheme is maddening. Thankfully, we can avoid this issue almost entirely by considering an implicit scheme called the **Crank-Nicolson method**. In this method we approximate the temporal

derivative with an Euler-type approximation, but we approximate the spatial derivative as the average of the central difference at the old time step and the central difference at the new time step. That is:

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = \frac{1}{2}\left[D\left(\frac{U_{i+1}^n - 2U_i^n + U_{i+1}^n}{\Delta x^2}\right) + D\left(\frac{U_{i+1}^{n+1} - 2U_i^{n+1} + U_{i+1}^{n+1}}{\Delta x^2}\right)\right].$$

Letting $r = D\Delta t/(2\Delta x^2)$ we can rearrange to get

$$-rU_{i-1}^{n+1} + (1+2r)U_i^{n+1} - rU_{i+1}^{n+1} = rU_{i-1}^n + (1-2r)U_i^n + rU_{i+1}^n.$$

This can now be viewed as a system of equations. Let's 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.

a. First let's write the equations for several values of i.

$$\begin{aligned}
(i=2): &\quad -rU_1^{n+1} + (1+2r)U_2^{n+1} - rU_3^{n+1} = rU_1^n + (1-2r)U_2^n + rU_3^n \\
(i=3): &\quad -rU_2^{n+1} + (1+2r)U_3^{n+1} - rU_4^{n+1} = rU_2^n + (1-2r)U_3^n + rU_4^n \\
(i=4): &\quad -rU_3^{n+1} + (1+2r)U_4^{n+1} - rU_5^{n+1} = rU_3^n + (1-2r)U_4^n + rU_5^n \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \quad \vdots \\
(i=N-1): &\quad -rU_{N-2}^{n+1} + (1+2r)U_{N-1}^{n+1} - rU_N^{n+1} = rU_{N-2}^n + (1-2r)U_{N-1}^n + rU_N^n
\end{aligned}$$

where N is the number of spatial points.

b. The first and last equations can be simplified since we have the Dirichlet boundary conditions. Therefore for $j = 2$ we can rearrange to move U_1 to the right-hand side since it is fixed for all time. Similarly for $j = N-1$ we can move U_N to the right-hand side since it is fixed for all time. Rewrite these two eqution.

c. Verify that the left-hand side of the equations that we have built in parts (a) and (b) can be written as the following matrix-vector product:

$$\begin{pmatrix}
(1+2r) & -r & 0 & 0 & 0 & \cdots & 0 \\
-r & (1+2r) & -r & 0 & 0 & \cdots & 0 \\
0 & -r & (1+2r) & -r & 0 & \cdots & 0 \\
\vdots & & & \ddots & & & 0 \\
0 & \cdots & & & 0 & -r & (1+2r)
\end{pmatrix}
\begin{pmatrix}
U_2^{n+1} \\ U_3^{n+1} \\ U_4^{n+1} \\ \vdots \\ U_{N-1}^{n+1}
\end{pmatrix}$$

d. Verify that the right-hand side of the equations that we built in parts (a)

and (b) can be written as

$$\begin{pmatrix} (1-2r) & r & 0 & 0 & 0 & \cdots & 0 \\ r & (1-2r) & r & 0 & 0 & \cdots & 0 \\ 0 & r & (1-2r) & r & 0 & \cdots & 0 \\ \vdots & & & \ddots & & & 0 \\ 0 & \cdots & & & 0 & r & (1-2r) \end{pmatrix} \begin{pmatrix} U_2^n \\ U_3^n \\ U_4^n \\ \vdots \\ U_{N-1}^n \end{pmatrix} + \begin{pmatrix} 2rU_1 \\ 0 \\ \vdots \\ 0 \\ 2rU_N \end{pmatrix}$$

e. Now for the wonderful part! The entire system of equations from part (a) can be written as

$$AU^{n+1} = BU^n + D.$$

What are the matrices A and B and what are the vectors U^{n+1}, U^n, and D?

f. To solve for U^{n+1} at each time step we simply need to do a linear solve:

$$U^{n+1} = A^{-1}\left(BU^n + D\right).$$

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.

g. 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 regular heat equation. 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 (f) of this problem.

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 6.9 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.

6.5 The Wave Equation

The problems that we've dealt with thus far all model natural diffusion processes: heat transport, molecular diffusion, etc. Another interesting physical phenomenon is that of wave propagation. In 1 spatial dimension the *wave equation* is

$$u_{tt} = cu_{xx}$$

where c is a parameter modeling the stiffness of the medium the wave is traveling through. With homogeneous Dirichlet boundary conditions we can think of this

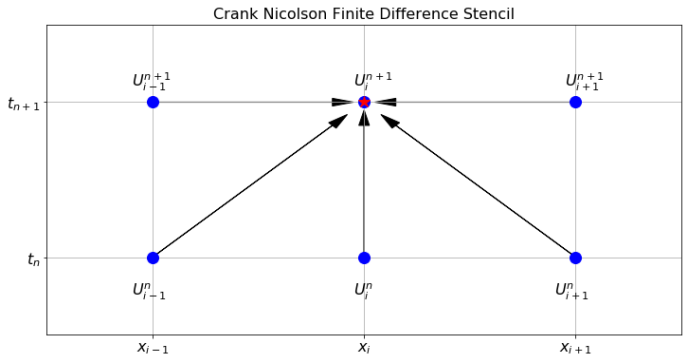


Figure 6.9: The finite difference stencil for the Crank Nicolson method.

as the behavior of a guitar string after it has been plucked. If the boundaries are in motion then the model might be of someone wiggling a taught string from one end.

Exercise 6.45. Let's 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 $x = x_i$.

a. Give a reasonable discretization of the second derivative in time:

$$u_{tt}(t_{n+1}, x_i) = \_\_\_\_\_.$$

b. Give a reasonable discretization of the second derivative in space:

$$u_{xx}(t_n, x_i) = \_\_\_\_\_.$$

c. Put your answers to parts (a) and (b) together with the wave equation to get

$$\frac{???-???+???}{\Delta t^2} = c\frac{???-???+???}{\Delta x^2}.$$

d. Solve the equation from part (c) for U_i^{n+1}. The resulting difference equation is the finite difference scheme for the 1D wave equation.

e. 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?

f. 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 6.10 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 $u_t(0,x)$.

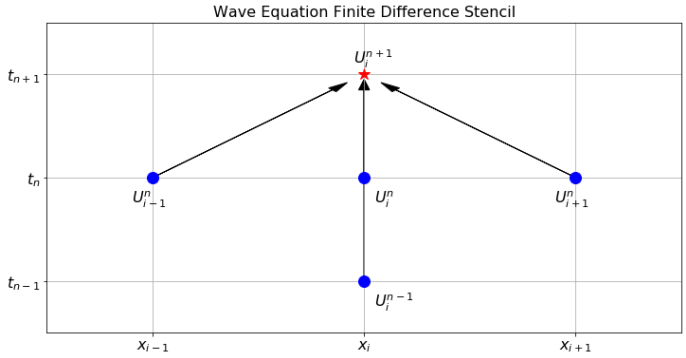


Figure 6.10: The finite difference stencil for the 1D wave equation.

Exercise 6.46. The ratio $c\Delta t^2/\Delta x^2$ shows up explicitly in the finite difference scheme for the 1D wave equation. Just like in the heat equation, this parameter controlls when the finite difference solution will be stable. Experiment with your finite difference solution and conjecture a value of $a = c\Delta t^2/\Delta x^2$ which divides the regions of stability versus instability. Your answer should be in the form:

If $a = c\Delta t^2/\Delta x^2 <$ _____ then the finite difference scheme for the 1D wave equation will be stable. Otherwise it will be unstable.

Exercise 6.47. 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 6.48. What is the expected error in the finite difference scheme for the 1D wave equation? What does this mean in plain English?

Exercise 6.49. Use your finite difference code to solve the 1D wave equation

$$u_{tt} = cu_{xx}$$

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. What does the parameter c to the wave? Give a physical interpretation of c.

Exercise 6.50. 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 6.51. Now consider the 2D wave equation

$$u_{tt} = c\left(u_{xx} + u_{yy}\right).$$

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 $y = y_j$.

a. Give discretizations of the derivatives u_{tt}, u_{xx}, and u_{yy}.

b. 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.
c. Write code to implement the finite difference scheme from part (b) 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.
d. Draw the finite difference stencil for the 2D heat equation.

Exercise 6.52. 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 6.53. 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.

6.6 Traveling Waves

Now we turn our attention to a new PDE: the traveling wave 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 fiber optic cable.

Exercise 6.54. Consider the PDE $u_t + vu_x = 0$. There is a very easy way to get an analytic solution to this traveling wave equation. 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 explicitely, we are claiming that

$$u(t,x) = e^{-(x-vt-4)^2}$$

is the analytic solution to the PDE. Let's prove this.

a. Take the t derivative of $u(t,x)$.
b. Take the x derivative of $u(t,x)$.
c. The PDE claims that $u_t + vu_x = 0$. Verify that this equal sign is indeed true.

Exercise 6.55. Now 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 ipywidgets import interactive
v = 1
f = lambda x: np.exp(-(x-4)**2)
u = lambda t, x: f(x - v*t)
x = np.linspace(0,10,100)
t = np.linspace(0,10,100)

def plotter(N):
  plt.plot(x,u(t[N],x),'b')
  plt.grid()
  plt.ylim(0,1)
  plt.show()
interactive_plot = interactive(plotter, N=(0,len(t)-1,1))
interactive_plot
```

Theorem 6.6. *If $u_t + vu_x = 0$ with initial condition $u(0,x) = f(x)$ then the function $u(t,x) = f(x-vt)$ is an analytic solution to the PDE.*

Exercise 6.56. Use the chain rule to prove the previous theorem.

The traveling 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 behavior.

Exercise 6.57. Consider the traveling 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 traveling wave equation and solve for U_i^{n+1}. This is our first finite difference scheme for the traveling 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 traveling wave equation. Figure 6.11 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.

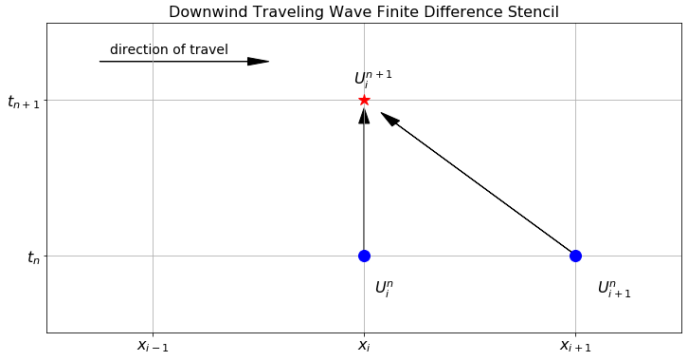


Figure 6.11: The finite difference stencil for the 1D downwind scheme on the traveling wave equation.

Exercise 6.58. You should have noticed in the previous exercise that you cannot reasonbly stabilize the finite difference scheme. Propose several reasons why this method appears to be unstable no matter what you use for the ratio $v\Delta t/\Delta x$.

Exercise 6.59. One of the troubles with the finite difference scheme that we have built for the traveling wave equation is that we are using the information at our present spatial location and the next spatial location to the right to propogate 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 traveling 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 6.12 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.

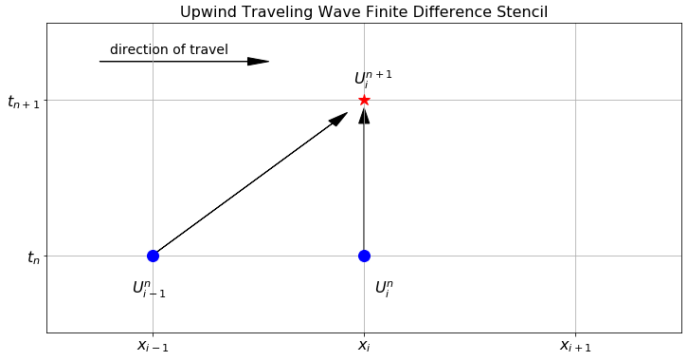


Figure 6.12: The finite difference stencil for the 1D downwind scheme on the traveling wave equation.

Exercise 6.60. Complete the following sentences:

a. In the downwind finite difference scheme for the traveling wave equation, the approximate solution moves at the correct speed, but ...

b. In the upwind finite difference scheme for the traveling save equation, the approximate solution moves at the correct speed, but ...

Exercise 6.61. Neither the downwind nor the upwind solutions for the traveling 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 traveling 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.

a. 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.**

b. For this next method let's stick with the second-order approximation of u_x but we'll 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

$$U_i^n \approx \frac{1}{2}\left(U_{i+1}^n + U_{i-1}^n\right)?$$

This would make our approximation of the time derivative

$$u_t \approx \frac{U_i^{n+1} - \frac{1}{2}\left(U_{i+1}^n + U_{i-1}^n\right)}{\Delta t}.$$

Solve this modified finite difference equation for U_i^{n+1} and implement this method. This is called the **Lax-Friedrichs** method.

c. Finally we'll 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 centered 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

$$\left(\frac{v^2 \Delta t^2}{2\Delta x^2}\right)\left(U_{j-1}^n - 2U_j^n + U_{j+1}^n\right)$$

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.

6.7 The Laplace and Poisson Equations

Exercise 6.62. Consider the 1D heat equation $u_t = 1u_{xx}$ with boundary conditions $u(t, 0) = 0$ and $u(t, 1) = 1$ and initial condition $u(0, x) = 0$.

a. Describe the physical setup for this problem.

b. 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?

c. 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 6.63. 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).

a. Conjecture what the steady state heat profile will look like for this particular setup. Be able to defend your answer.

b. 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 6.64. Next we'll 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 $u(0, x, y) = 0$.

a. Describe the physical setup for this problem.
b. 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.
c. 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 6.65. Now consider the forced 2D heat equation $u_t = u_{xx} + u_{yy} + e^{-(x-0.5)^2-(y-0.5)^2}$ 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).

a. Conjecture what the steady state heat profile will look like for this particular setup. Be able to defend your answer.
b. 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) behavior 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 $u_t \to 0$. 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 behavior. We will first examine a 1D version of the Laplace and Poisson equations.

Exercise 6.66. 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's 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:

$$\begin{aligned}
U_0 &= 1 \quad \text{(left boundary condition)} \\
\frac{U_2 - 2U_1 + U_0}{\Delta x} &= 0 \\
\frac{U_3 - 2U_2 + U_1}{\Delta x} &= 0 \\
\frac{U_4 - 2U_3 + U_2}{\Delta x} &= 0 \\
\frac{U_5 - 2U_4 + U_3}{\Delta x} &= 0 \\
U_5 &= 0 \quad \text{(right boundary condition).}
\end{aligned}$$

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.

Exercise 6.67. Solve the 1D Laplace equation with Dirichlet boundary conditions is rather uninteresting since the answer will alway be a linear function connecting the two boundary conditions. Prove this.

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 6.68. 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 6.69. Now we can solve the Poisson equation from the previous problem numerically. Let's 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

$$\begin{aligned}
U_0 &= 2 \quad \text{(left boundary condition)} \\
\frac{U_2 - 2U_1 + U_0}{\Delta x} &= f(x_1) \\
\frac{U_3 - 2U_2 + U_1}{\Delta x} &= f(x_2) \\
\frac{U_4 - 2U_3 + U_2}{\Delta x} &= f(x_3) \\
\frac{U_5 - 2U_4 + U_3}{\Delta x} &= f(x_4) \\
U_5 &= 0.5 \quad \text{(right boundary condition).}
\end{aligned}$$

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 6.70. 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:

$$\text{Solve: } u_{xx} = f(x) \quad \text{on} \quad x \in (0,1) \quad \text{with} \quad u_x(0) = \alpha \quad \text{and} \quad u(1) = \beta.$$

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

$$\begin{aligned}
\frac{U_1 - U_0}{\Delta x} &= \alpha \quad \text{(left boundary condition)} \\
\frac{U_2 - 2U_1 + U_0}{\Delta x} &= f(x_1) \\
\frac{U_3 - 2U_2 + U_1}{\Delta x} &= f(x_2) \\
\frac{U_4 - 2U_3 + U_2}{\Delta x} &= f(x_3) \\
\frac{U_5 - 2U_4 + U_3}{\Delta x} &= f(x_4) \\
U_5 &= \beta \quad \text{(right boundary condition).}
\end{aligned}$$

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 6.71. We conclude this section by increasing from one dimension to two. Solve the Poisson equation $u_{xx} + u_{yy} = f(x,y)$ on the domain $(x,y) \in (0,1) \times (0,1)$ with homogenous Dirichlet boundary conditions and focing function $f(x,y) = 20\exp\left(-\frac{(x-0.5)^2+(y-0.5)^2}{0.05}\right)$ numerically. Start with a 6×6 grid of points and explicitly write down all of the equations. There should only be 16 total equations since in a 6×6 grid there are 20 points on the boundary and

16 points in the interior of the domain. Then generalize your code to solve the Poisson equation with a much smaller value of $\Delta x = \Delta y$.

Hint: You will want to number the nodes in your 2D domain and then carefully construct the coefficient matrix based on your numbering scheme. You should observe a nice pattern in your coefficient matrix that will allow you to more easily generalize your code.

6.8 Exercises

6.8.1 Algorithm Summaries

Exercise 6.72. Explain in clear language what it means to check an analytic solution to a differential equation.

Exercise 6.73. Explain in clear language what Dirichlet boundary conditions are.

Exercise 6.74. Explain in clear language what Neumann boundary conditions are.

Exercise 6.75. 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 6.76. 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 6.77. 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 6.78. 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 6.79. Explain in clear language what it means for a finite difference method to be stable vs unstable.

Exercise 6.80. Show the full mathematical details for building a downwind finite difference scheme for the traveling wave equation. Discuss the primary disadvantages of the downwind scheme.

Exercise 6.81. Show the full mathematical details for building an upwind finite difference scheme for the traveling wave equation. Discuss the primary disadvantages of the upwind scheme.

6.8.2 Applying What You've Learned

Exercise 6.82. 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 homogenous 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 $D(x) = x(1-x)$. 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:

a. Describe what this choice of $D(x)$ might mean physically in the heat equation.

b. Write an explicit scheme to solve this problem by using centered 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.

c. Write a script to find an approximate solution to this problem.

d. Write a clear and thorough discussion about how your will choose Δx and Δt to give stable solutions to this equation.

e. 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 6.83. In a square domain create a function $u(0, x, y)$ that looks like your college 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 6.84. Repeat the previous exercise but this time solve the wave equation with the logo as the initial condition.

Exercise 6.85. Consider the time-independent partial differential equation $-\varepsilon u_{xx} + u_x = 1$ on the domain $x \in (0, 1)$ with boundary conditions $u(0) = u(1) = 0$ and parameter ε with $0.001 < \varepsilon < 1$. Write code to solve this boundary valued problem and provide plots of your numerical solution for various values of ε.

Exercise 6.86. 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 $k = 10^{-5}$ m^2/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 $T(x, t) = 75 + 1.5x - 20\sin(\pi x/10)e^{-ct}$ for some value of c.

a. Working by hand (no computers!) test this function 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.

b. 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 6.87. (This problem is modified from [11]. 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 $L = 300mm$ and square cross-sectional width $3.2mm$.

- The temperature sensors were placed at $x_1 = 47mm$, $x_2 = 94mm$, $x_3 = 141mm$, and $x_4 = 188mm$ 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
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

a. At time 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 $x = 300mm$.

b. 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.
c. It is unlikely that your numerical solution to the heat equation and the data from part (b) 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
data = np.array( pd.read_csv('https://raw.githubusercontent.com/NumericalMethodsSullivan/Numerica
```

Exercise 6.88. You may recall from your differential equations class that population growth under limited resources is goverened by the logistic equation $x' = k_1 x(1 - x/k_2)$ 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's 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

$$\frac{\partial u}{\partial t} = k_1 u \left(1 - \frac{u}{k_2}\right) + D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

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

$$u(0, x, y) = e^{-((x-0.5)^2+(y-0.5)^2)/0.05}.$$

6.9 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. Take the time to read Appendix B before you write your final solution.

6.9.1 Hunting and Diffusion

Let u be a function modeling a mobile population that in an environment where it has a growth rate of $r\%$ per year with 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 $h\%$ of the population per year 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's make a few assumptions about the environment that they're in and how the individuals move.

- Food is abundant in the entire environment.
- Individuals in the population like to spread out so that they don't interfere with each others' hunt for food.
- It is equally easy for the individuals to travel in any direction in the environment.

Clearly some of these assumptions are unreasonable for real populations and real environments, but let's 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

$$\frac{\partial u}{\partial t} = ru\left(1 - \frac{u}{K}\right) - hu + D\left(u_{xx} + u_{yy|}\right).$$

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 isn't 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 (b) with this new diffusion term.

6.9.2 Heating Adobe Houses

Adobe houses, typically built in desert climates, are known for their great thermal efficiency. The heat equation

$$\frac{\partial T}{\partial t} = \frac{k}{c_p \rho}\left(T_{xx} + T_{yy} + T_{zz}\right),$$

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.

Appendix A

Introduction to Python

In this optional Chapter we will walk through some of the basics of using Python3 - the powerful general-purpose programming language that we'll use throughout this class. I'm assuming throughout this Chapter that you're familiar with other programming languages such as R, Java, C, or MATLAB. Hence, I'm assuming that you know the basics about what a programming langue "is" and "does". There are a lot of similarities between several of these languages, and in fact they borrow heavily from each other in syntax, ideas, and implementation.

While you work through this chapter it is expected that you do every one of the examples and exercises on your own.

A.1 Why Python?

We are going to be using Python since

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

It is important to keep in mind that Python is a general purpose language that we will be using for Scientific Computing. The purpose of Scientific Computing is *not* to build apps, build software, manage databases, or develop user interfaces. Instead, Scientific Computing is the use of a computer programming language (like Python) along with mathematics to solve scientific and mathematical problems. For this reason it is definitely not our purpose to write an all-encompassing guide for how to use Python. We'll only cover what is necessary

for our computing needs. You'll learn more as the course progresses so use this chapter as a reference just to get going with the language.

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/, and
- `sympy` (symbolic Python) https://www.sympy.org/en/index.html.

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

A.2 Getting Started

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 course. For this class we 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 meant to be 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.

If you insist on installing Python on your own machine then we highly recommend that you start with the Anaconda downloader https://www.anaconda.com/distribution/ since it includes the most up-to-date version of Python as well as some of the common tools for writing Python code.

In the rest of this chapter we will assume that you have a working version of Python along with a working version of Jupyter Notebooks to work in. The exercises in this appendix are meant to get you going with Python – you should do *every* one of them. Remember that this is not a full replacement for a "how to program in Python" resource. We have only included the essential aspects of the Python language in this chapter as they relate to the mathematical goals of the book. There is definitely more to say about Python and we don't intend to cover it all here.

A.3 Hello, World!

As is tradition for a new programming language, we should create code that prints the words "Hello, world!" to the screen. The code below does just that.

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

```
## Hello, world!
```

In a Jupyter Notebook you will write your code in a code block, and when you're ready to run it you can press Shift+Enter (or Control+Enter) and you'll see your output. Shift+Enter will evaluate your code and advance to the next block of code. Control+Enter will evaluate without advancing the cursor to the next block.

Exercise A.1. Have Python print `Hello, world!` to the screen.

Exercise A.2. Write code to print your name to the screen.

Exercise A.3. You should now spend a bit of time poking around in Jupyter Notebooks. Figure out how to

- save a file
- load a new iPython Notebook (Jupyter Notebook) file from your computer or your Google Drive
- write text in a Jupyter Notebook
- use the keyboard to switch between writing text and writing code.

A.4 Python Programming Basics

A.4.1 Variables

Variables in Python can contain letters (lower case or capital), numbers 0-9, and some special characters such as the underscore. Variable names should start with a letter. Of course there are a bunch of reserved words (just like in any other language). You should look up what the reserved words are in Python so you don't accidentally use them.

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 don't 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 decimal number, not an integer. However, in Python 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 is",x," and has type", type(x),". \n")
print("The variable y is",y," and has type", type(y),". \n")
```

```
## The variable x is 7  and has type <class 'int'> .
```

```
## The variable y is 7.0  and has type <class 'float'> .
```

```
# 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)
```

```
## Multiplying x by 1.0 gives 7.0
```

```
print("The type of this value is", type(1.0*x),". \n")
```

```
## The type of this value is <class 'float'> .
```

Note that the allowed mathematical operations are:

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

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

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

```
## 49.0
```

Exercise A.4. What happens if you type `7^2` into Python? What does it give you? Can you figure out what it is doing?

Exercise A.5. 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. Some sample output is shown below.

```
## a= 3 , b= 4 , c= 5
## a, b, and c form a Pythagorean triple.
## a= 2 , b= 3 , c= 7
## a, b, and c do not form a Pythagorean triple.
```

A.4.2 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. 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.

Some things to keep in mind with Python lists:

- Python starts indexing at 0
- Python stops reading at the second to last index
- The following blocks of code show this feature in action for several different lists.

Example A.1 (Lists and Indexing)**.** Let's 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 and then print only the element with index 0.

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

```
## 0
```

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

```
MyList = [0,1,2,3,4,5,6,7,8]
print(MyList[:2])
```

```
## [0, 1]
```

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

```
MyList = [0,1,2,3,4,5,6,7,8]
print(MyList[-1])
```

```
## 8
```

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

```
MyList = [0,1,2,3,4,5,6,7,8]
print(MyList[1:5])
```

```
## [1, 2, 3, 4]
```

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

```
MyList = [0,1,2,3,4,5,6,7,8]
print(MyList[0::2])
```

```
## [0, 2, 4, 6, 8]
```

- Print the last three elements of `MyList`

```
MyList = [0,1,2,3,4,5,6,7,8]
print(MyList[-3:])
```

```
## [6, 7, 8]
```

Example A.2 (Range and Lists)**.** Let's look at another example of indexing in lists. In this one we'll use the `range` command to build the initial list of numbers. Read the code carefully so you know what each line does.

```
# range is a handy command for creating a sequence of integers
MySecondList = range(4,20)
print(MySecondList) # this is a "range object" in Python.
# When using range() we won't actually store all of the values in memory.
```

```
## range(4, 20)
```

```
print(list(MySecondList)) # notice that we didn't create the last element!
```

```
## [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
print(MySecondList[0]) # print the first element ... the one with index 0
```

```
## 4
```

```
print(MySecondList[-5]) # print the fifth element from the end
```

```
## 15
```

```
print(MySecondList[-1:0:-1]) # this creates a new range object.
# Take careful note of how the above range object is defined.
# Print the last element to the one indexed by 1 counting backwards
```

```
## range(19, 4, -1)
```

```
print(list(MySecondList[-1:0:-1]))
```

```
## [19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5]
```

```
print(MySecondList[-1::-1]) # this creates another new range object
```

```
## range(19, 3, -1)
```

```
print(list(MySecondList[-1::-1])) # print the whole list backwards
```

```
## [19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4]
```

```
print(MySecondList[::2]) # create another new range object
```

```
## range(4, 20, 2)
```

```
print(list(MySecondList[::2])) # print every other element
```

```
## [4, 6, 8, 10, 12, 14, 16, 18]
```

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

Example A.3. 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 j in Python. This is common in many scientific disciplines and is just another thing that we'll need to get used to in Python.

```
MixedList = [1.0, 7, 'Bob', 1-1j]
print(MixedList)
```

```
## [1.0, 7, 'Bob', (1-1j)]
```

```
print(type(MixedList[0]))
```

```
## <class 'float'>
```

```
print(type(MixedList[1]))
```

```
## <class 'int'>
```

```
print(type(MixedList[2]))
```

```
## <class 'str'>
```

```
print(type(MixedList[3])) # Notice that we use 1j for the imaginary number "i".
```

```
## <class 'complex'>
```

Exercise A.6. In this exercise you will put your new list skills into practice.

a. Create the list of the first several Fibonacci numbers: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89$.
b. Print the first four elements of the list.
c. Print every third element of the list starting from the first.
d. Print the last element of the list.
e. Print the list in reverse order.
f. Print the list starting at the last element and counting backward by every other element.

A.4.3 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.command`
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`. This is a common programming feature in Python and we'll use it often.

Example A.4 (.append)**.** The `.append` command can be used to append an element to the end of a list.

```
MyList = [0,1,2,3]
print(MyList)
```

```
## [0, 1, 2, 3]
```

```
MyList.append('a') # append the string 'a' to the end of the list
print(MyList)
```

```
## [0, 1, 2, 3, 'a']
```

```
MyList.append('a') # do it again ... just for fun
print(MyList)
```

```
## [0, 1, 2, 3, 'a', 'a']
```

```
MyList.append(15) # append the number 15 to the end of the list
print(MyList)
```

```
## [0, 1, 2, 3, 'a', 'a', 15]
```

Example A.5 (.remove)**.** The `.remove` command can be used to remove an element from a list.

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

```
## [0, 1, 2, 3, 'a', 15]
```

```
MyList.remove(3) # now let's remove the 3
print(MyList)
```

```
## [0, 1, 2, 'a', 15]
```

Example A.6 (.insert)**.** The `.insert` command can be used to insert an element at a location in a list.

```
MyList = [0,1,2,3]
MyList.append('a') # append the string 'a' to the end of the list
MyList.append('a') # do it again ... just for fun
MyList.append(15) # append the number 15 to the end of the list
MyList.remove('a') # remove the first instance `a` from the list
MyList.remove(3) # now let's remove the 3
print(MyList)
```

```
## [0, 1, 2, 'a', 15]
```

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

```
## ['A', 0, 1, 'B', 2, 'a', 15]
```

Exercise A.7. In this exercise you will go a bit further with your list operation skills.

a. Create the list of the first several Lucas Numbers: $1, 3, 4, 7, 11, 18, 29, 47$.
b. Add the next three Lucas Numbers to the end of the list.
c. Remove the number 3 from the list.
d. Insert the 3 back into the list in the correct spot.
e. Print the list in reverse order.

f. Do a few other list operations to this list and report your findings.

A.4.4 Tuples

In Python, a "tuple" is like an ordered pair (or order triple, or order quadruple, ...) in mathematics. We will occasionally see tuples in our work in numerical analysis so for now let's 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.

```
point = 10, 20 # notice that I don't need the parenthesis
print(point, type(point))
```

```
## (10, 20) <class 'tuple'>
```

We can also define a tuple with parenthesis if we like. Python doesn't care.

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

```
## (10, 20) <class 'tuple'>
```

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

```
point = (10, 20)
x, y = point
print("x = ", x)
```

```
## x =  10
```

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

```
## y =  20
```

A.4.5 Control Flow: Loops and If Statements

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

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

A.4.5.1 `for` Loops

A `for` loop is designed to do a task a certain number of times and then stop. This is a great tool for automating repetitive tasks, but it also nice numerically for building sequences, summing series, or just checking lots of examples. The following are 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 indention. There is no need for an `end` command or a curly brace. All of the control statements in Python are white-space-dependent.

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

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

```
## 1
## 4
## 9
## 16
## 25
## 36
```

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

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

```
## Alice
## Billy
## Charlie
## Dom
## Enrique
## Francisco
```

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

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

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

```
## [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
print("The sum of the first 9 perfect squares is",sum(CoolList))
```

```
## The sum of the first 9 perfect squares is 285
```

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

Example A.10. In the following code we write a for loop that outputs a list of the first 7 iterations of the sequence $x_{n+1} = -0.5x_n + 1$ starting with $x_0 = 3$. Notice that we're 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
```

```
## [3.0, -0.5]
## [3.0, -0.5, 1.25]
## [3.0, -0.5, 1.25, 0.375]
## [3.0, -0.5, 1.25, 0.375, 0.8125]
## [3.0, -0.5, 1.25, 0.375, 0.8125, 0.59375]
## [3.0, -0.5, 1.25, 0.375, 0.8125, 0.59375, 0.703125]
## [3.0, -0.5, 1.25, 0.375, 0.8125, 0.59375, 0.703125, 0.6484375]
```

Example A.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'll 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
```

```
## [3.0, -0.5, 0, 0, 0, 0, 0]
## [3.0, -0.5, 1.25, 0, 0, 0, 0]
## [3.0, -0.5, 1.25, 0.375, 0, 0, 0]
## [3.0, -0.5, 1.25, 0.375, 0.8125, 0, 0]
## [3.0, -0.5, 1.25, 0.375, 0.8125, 0.59375, 0]
## [3.0, -0.5, 1.25, 0.375, 0.8125, 0.59375, 0.703125]
```

Exercise A.8. We want to sum the first 100 perfect cubes. Let's do this in two ways.

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

b. 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 command.

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

Exercise A.9. Write a `for` loop that builds the first 20 terms of the sequence $x_{n+1} = 1 - x^2$ with $x_0 = 0.1$. Pre-allocate enough memory in your list and then fill it with the terms of the sequence. Only print the list after all of the computations have been completed.

A.4.5.2 While Loops

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

Example A.12. Print the numbers 0 through 4 and then the word "done". We'll 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
```

```
## 0
## 1
## 2
## 3
## 4
```

```
print("done")
```

```
## done
```

Example A.13. Now let's 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 don't 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])
Fib
```

```
## [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597]
```

Exercise A.10. Write a `while` loop that sums the terms in the Fibonacci sequence until the sum is larger than 1000

A.4.5.3 If Statements

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

Example A.14. Let's 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?")
```

```
## Hello, Alice.  Isn't it a lovely day to learn Python?
```

```
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?")
```

```
## You're not Alice.  Where is Alice?
```

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

Note: We had to import the `numpy` package to get the random number generator in Python. Don't worry about that for now. We'll 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")
```

```
## 0.0026618925802063265  is less than a half
```

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

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

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

```
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," is less than 1/3")
elif x < 0.67:
    print(x, "is less than 2/3 but greater than or equal to 1/3")
else:
    print(x, "is greater than or equal to 2/3")
```

```
## 0.6175500105294413 is less than 2/3 but greater than or equal to 1/3
```

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

Exercise A.11. Write code to give the Collatz Sequence

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

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

A.4.6 Functions

Mathematicians and programmers talk about functions in very similar ways, but they aren't 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 we can have multiple outputs for a programmatic function. We are not going to be talking about symbolic computation on functions in this section. Symbolic computations will have to wait for the 'sympy' tutorial.

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

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

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

```
## 35.937
```

Take careful note of several things in the previous example:

- To define the function we can not just type it like we would see it one paper. This is not how Python recognizes functions. We just have to get used to it.
- 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. See the section on `sympy` in this chapter if you want to do symbolic computation.

---

[1] Actually, it is still an open mathematical question that *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.

Exercise A.12. 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 $n = 39$. It is curious to note that each of these outputs is a prime number (check this on your own). Will the function produce a prime for $n = 40$? For $n = 41$?

Example A.18. One cool thing that you can do with Python 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.

Now let's 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) ) # 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's 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
```

```
## [1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

Example A.19. For this next example let's define the sequence

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

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's only build the first 5 terms
    x.append(MySeq(x[-1]))
print(x)
```

```
## [0.125, 0.25, 0.5, 1.0, 1.0, 1.0]
```

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

$$x_{n+1} = \frac{1}{2}x_n + \frac{1}{x_n}$$

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.

Hint: Import the `math` package so that you get the square root. 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)))
```

```
## approximation         exact       abs error
## 1.459090909090909 1.4142135623730951 0.04487734671781385
## 1.414903709997168 1.4142135623730951 0.0006901476240728233
## 1.4142137306897584 1.4142135623730951 1.6831666327377093e-07
## 1.4142135623731051 1.4142135623730951 9.992007221626409e-15
```

Exercise A.13. 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

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

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.

A.4.7 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 the most like the way that we write mathematical functions. For example, let's define the mathematical function $f(x) = x^2 + 3$ in two different ways.

- As a Python function with `def`:

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

- As a `lambda` function:

```
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.

Now 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 = lambda x: x**2+3
f(1.5) # evaluate the function at x=1.5
```

```
## 5.25
```

where the result is exactly the floating point number we were interested in.

The distinct mathematical advantage for using `lambda` functions is that the code for setting up a Lambda Function is about as close as we're going to get to a mathematically defined function as we would write it on paper, but the code for evaluating a `lambda` Function is *exactly* what we would write on paper. Additionally, there is less coding overhead than for defining a function with the command.

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 could now write the code `f(2,4)`.

Example A.21. You may recall Euler's Method from your differential equations training. Euler's Method will give a list of approximate values of the solution to a first order differential equation at given times.

Consider the differential equation $y' = -0.25y + 2$ with the initial condition $y(0) = 1.1$. We can define the right-hand side of the differential equation as a

`lambda` Function in our code so that we can call on it over and over again in our Euler's Method solution. We'll take 10 Euler steps starting at the proper initial condition. Pay particular attention to how we use the `lambda` function.

```
import numpy as np
RightSide = lambda y: -0.25*y + 2 # define the right-hand side
dt = 0.125 # define the delta t in Euler's method
t = [0] # initial time
y = [1.1] # initial condition
for n in range(0,10):
    t.append(t[n] + dt) # increment the time
    y.append(y[n] + dt*RightSide(y[n])) # find approx soln at next pt
print(t) # print the times
```

```
## [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.0, 1.125, 1.25]
```

```
print(np.round(y,4)) # round the approx y values to 4 decimal places
```

```
## [1.1    1.3156 1.5245 1.7269 1.9229 2.1128 2.2968 2.475  2.6477 2.8149
##  2.977 ]
```

Exercise A.14. Go back to Exercise A.12 and repeat this exercise using a `lambda` Function instead of a Python function.

Exercise A.15. Go back tp problem A.13 and repeat this exercise using a `lambda` function instead of a Python function.

A.4.8 Packages

Unlike mathematical programming languages like MATLAB, Maple, or Mathematica, where every package is already installed and ready to use, Python allows you to only load the packages that you might need for a given task. There are several advantages to this along with a few disadvantages.

Advantages:

1. You can have the same function doing different things in different scenarios. For example, there could be a symbolic differentiation command and a numerical differentiation command coming from different packages that are used in different ways.

2. Housekeeping. It is highly advantageous to have a good understanding of where your functions come from. MATLAB, for example, uses the same name for multiple purposes with no indication of how it might

behave depending on the inputs. With Python you can avoid that by only importing the appropriate packages for your current use.

3. Your code will be ultimately more readable (more on this later).

Disadvantages:

1. It is often challenging to keep track of which function does which task when they have exactly the same name. For example, you could be working with the `sin()` function numerically from the `numpy` package or symbolically from the `sympy` package, and these functions will behave differently in Python - even though they are exactly the same mathematically.
2. You need to remember which functions live in which packages so that you can load the right ones. It is helpful to keep a list of commonly used packages and functions at least while you're getting started.

Let's start with the `math` package.

Example A.22. 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)
```

```
## 0.7071067811865476
```

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's fix this. When you import a package you could just import all of the functions so they can be used by their proper names.

Example A.23. 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)
```

```
## 0.7071067811865476
```

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'll import both `math` and `numpy`, but instead we will import them with shortened names so we can type things a bit faster.

Example A.24. 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
x = ma.cos( ma.pi / 4) # use the math version of the cosine function
y = np.cos( np.pi / 4) # use the numpy version of the cosine function
print(x, y)
```

```
## 0.7071067811865476 0.7071067811865476
```

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

Example A.25 (Contents of a Library)**.** 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))
```

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

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

A.5 Numerical Python with `numpy`

The base implementation of Python includes the basic programming language, the tools to write loops, check conditions, build and manipulate lists, and all of the other things that we saw in the previous section. In this section we will

explore the package numpy that contains optimized numerical routines for doing numerical computations in scientific computing.

Example A.26. To start with let's look at a really simple example. Say you have a list of real numbers and you want to take the sine 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]
print(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]) )
print(SineList)
```

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

Typically when we import numpy we use `import numpy as np`. This is the standard way to name the numpy package. This means that we will have lots of function with the prefix "np" in order to call on the numpy commands. Let's 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.)

A.5.1 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 don't 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 exception 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's just look at a few examples. What we're going to do is to define a matrix A and vectors v and w as

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

Then we'll do the following

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

Example A.27 (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)
```

```
## The matrix A is:
##  [[1 2]
##  [3 4]]
```

```
v = np.array([[5],[6]]) # this creates a column vector
print("The vector v is:\n",v)
```

```
## The vector v is:
##  [[5]
##  [6]]
```

```
w = np.array([5,6]) # this creates a row vector
print("The vector w is:\n",w)
```

```
## The vector w is:
##  [5 6]
```

Example A.28 (variable.shape)**.** The `variable.shape` command can be used to give the shape of a `numpy` array. Notice that the output is a tuple showing the size `(rows, columns)`. Also notice that the row vector doesn't give (1,2) as expected. Instead it just gives (2,).

```
import numpy as np
A = np.array([[1,2],[3,4]])
print(A.shape) # Shape of the matrix A
```

```
## (2, 2)
```

```
v = np.array([[5],[6]])
print(v.shape) # Shape of the column vector v
```

```
## (2, 1)
```

```
w = np.array([5,6])
print(w.shape) # Shape of the row vector w
```

```
## (2,)
```

Example A.29 (variable.size)**.** The `variable.size` command 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 command.

```
import numpy as np
A = np.array([[1,2],[3,4]])
v = np.array([[5],[6]])
w = np.array([5,6])
print(A.size) # Size (number of elements) of A
```

```
## 4
```

```
print(v.size) # Size (number of elements) of v
```

```
## 2
```

```
print(w.size) # Size (number of elements) of w
```

```
## 2
```

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

Example A.30. Let's 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
```

```
## 1
```

```
print(A[1,1]) # bottom right
```

```
## 4
```

Example A.31. Let's read the first row from that matrix A.

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

```
## [1 2]
```

Example A.32. Let's read the second column from the matrix A.

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

```
## [2 4]
```

Notice when we read the column it was displayed as a column. Be careful. Reading a column from a matrix will automatically flatten it into an array, not a matrix.

If we try to multiply either A and v or A and A we will get some funky results. Unlike programming languages like MATLAB, the default notion of multiplication is NOT matrix multiplication. Instead, the default is element-wise multiplication.

Example A.33. 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]])
```

```
# Notice that A*A is NOT the same as A*A with matrix multiplication
print(A * A)
```

```
## [[ 1  4]
##  [ 9 16]]
```

Example A.34. If we write `A * v` Python will do element-wise multiplication across each column since v is a column vector.

```
import numpy as np
A = np.array([[1,2],[3,4]])
v = np.array([[5],[6]])
print(A * v) # This will do element wise multiplication on each column
```

```
## [[ 5 10]
##  [18 24]]
```

If, however, we recast these arrays as matrices we can get them to behave as we would expect from Linear Algebra. It is up to you to check that these products are indeed correct from the definitions of matrix multiplication from Linear Algebra.

Example A.35. Recasting the numpy arrays as matrices allows you to use multiplication as we would expect from linear algebra.

```
import numpy as np
A = np.matrix([[1,2],[3,4]])
v = np.matrix([[5],[6]])
w = np.matrix([5,6])
print("The product A*A is:\n",A*A)
```

```
## The product A*A is:
##  [[ 7 10]
##  [15 22]]
```

```
print("The product A*v is:\n",A*v)
```

```
## The product A*v is:
##  [[17]
##  [39]]
```

```
print("The product w*A is:\n",w*A)
```

```
## The product w*A is:
##  [[23 34]]
```

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

Example A.36 (Matrix Transpose)**.** Taking the transpose of a matrix (swapping the rows and columns) is done with the `matrix.T` command. This is just like other array commands we have seen in Python (like `.append`, `.remove`, `.shape`, etc.).

```
import numpy as np
A = np.matrix([[1,2],[3,4]])
print(A.T) # The transpose is relatively simple
```

```
## [[1 3]
##  [2 4]]
```

Example A.37 (Matrix Inverse)**.** The inverse of a square matrix is done with `A.I`.

```
import numpy as np
A = np.matrix([[1,2],[3,4]])
Ainv = A.I # Taking the inverse is also pretty simple
print(Ainv)
```

```
## [[-2.   1. ]
##  [ 1.5 -0.5]]
```

```
print(A * Ainv) # check that we get the identity matrix back
```

```
## [[1.00000000e+00 1.11022302e-16]
##  [0.00000000e+00 1.00000000e+00]]
```

Example A.38 (Matrix Determinant)**.** The determinant command is hiding under the `linalg` subpackage inside `numpy`. Therefore we need to call it as such.

```
import numpy as np
A = np.matrix([[1,2],[3,4]])
# The determinant is inside the numpy.linalg package
print(np.linalg.det(A))
```

```
## -2.0000000000000004
```

Example A.39 (Trace of a Matrix)**.** The trace is done with `matrix.trace()`

```
import numpy as np
A = np.matrix([[1,2],[3,4]])
print(A.trace()) # The trace is pretty darn easy too
```

```
## [[5]]
```

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

Exercise A.17. Now that we can do some basic linear algebra with `numpy` it is your turn. Define the matrix B and the vector u as

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

Then find

a. Bu
b. B^2 (in the traditional linear algebra sense)
c. The size and shape of B
d. $B^T u$
e. The element-by-element product of B with itself
f. The dot product of u with the first row of B

A.5.2 `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.

- `arange` (array range) builds an array of floating point numbers with the arguments `start`, `stop`, and `step`. Note that you may not actually get to the `stop` point if the distance `stop-start` is not evenly divisible by the 'step'.
- `linspace` (linearly spaced points) 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`. In a linear space you are always guaranteed to hit the stop point exactly, but you don't have direct control over the stop size.
- The `zeros` and `ones` commands create matrices of zeros or ones.
- `meshgrid` builds two arrays that when paired make up the ordered pairs for a 2D (or higher D) mesh grid of points. This is the same as the `meshgrid` command in MATLAB.

Note: Just like with all Python lists, the "stop" number is the one immediately after where you intended to stop.

Example A.40 (np.arange)**.** The `np.arange` command is great for building sequences.

```
import numpy as np
x = np.arange(0,5,0.1)
print(x)
```

```
## [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7
##  1.8 1.9 2.  2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.  3.1 3.2 3.3 3.4 3.5
##  3.6 3.7 3.8 3.9 4.  4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9]
```

Example A.41 (np.linspace)**.** The `np.linspace` command builds a list with equal (linear) spacing between the starting and ending values.

```
import numpy as np
y = np.linspace(0,5,10)
print(y)
```

```
## [0.         0.55555556 1.11111111 1.66666667 2.22222222 2.77777778
##  3.33333333 3.88888889 4.44444444 5.        ]
```

Example A.42 (np.zeros)**.** The `np.zeros` command 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
print(z)
```

```
## [[0. 0. 0. 0. 0.]
##  [0. 0. 0. 0. 0.]
##  [0. 0. 0. 0. 0.]]
```

Example A.43 (np.ones)**.** The `np.ones` command builds an array of ones.

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

```
## [[1. 1. 1. 1. 1.]
##  [1. 1. 1. 1. 1.]
##  [1. 1. 1. 1. 1.]]
```

Example A.44 (mesh grid)**.** The `np.meshgrid` command creates a mesh grid. 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)
```

```
## x =  [[0. 1. 2. 3. 4. 5.]
##  [0. 1. 2. 3. 4. 5.]
##  [0. 1. 2. 3. 4. 5.]
##  [0. 1. 2. 3. 4. 5.]
##  [0. 1. 2. 3. 4. 5.]
##  [0. 1. 2. 3. 4. 5.]]
```

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

```
## y =  [[0. 0. 0. 0. 0. 0.]
##  [1. 1. 1. 1. 1. 1.]
##  [2. 2. 2. 2. 2. 2.]
##  [3. 3. 3. 3. 3. 3.]
##  [4. 4. 4. 4. 4. 4.]
##  [5. 5. 5. 5. 5. 5.]]
```

The thing to notice with the `np.meshgrid()` command is that when you lay the two matrices on top of each other, the matching entries give every ordered pair in the domain.

Exercise A.18. Now time to practice with some of these `numpy` commands.

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 (a) (as a column) and the answer to part (b).
d. Change the bottom row of your matrix from part (b) to all 3's, then change the third column to all 7's, and then find the 5^{th} power of this matrix.

A.6 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 modeled 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.

Note: The reader should note that we will NOT be plotting symbolically defined functions in this section. The `plot` command that we will be using is reserved for numerically defined plots (i.e. plots of data points), not functions that are

symbolically defined. If you have a symbolically defined function and need a plot, then pick a domain, build some x data, use the function to build the corresponding y data, and use the plotting tools discussed here. If you need a plot of a symbolic function and for some reason these steps are too much to ask, then look to the section of this Appendix on `sympy`.

A.6.1 Basics with `plt.plot()`

We are going to start right away with an example. In this example, however, we'll walk through each of the code chunks one-by-one so that we understand how to set up a proper plot. Something to keep in mind. The author strongly encourages students and readers to use Jupyter Notebooks for their Python coding. As such, there are some tricks for getting the plots to render that only apply to Jupyter Notebooks. If you are using Google Colab then you may not need some of these little tricks.

Example A.45 (Plotting with matplotlib)**.** In the first example we want to simply plot the sine function on the domain $x \in [0, 2\pi]$, color 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 is 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'll build a `numpy` array of x values (using the `np.linspace` command) and a `numpy` array of y values for the sine function.

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

- Finally, 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).
- Notice that we send the plot legend in directly to the plot command. This is optional and could set the legend up separately if we like.

- Then we'll add the grid with `plt.grid()`
- Then we'll add the legend to the plot
- Finally we'll 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're 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()
```

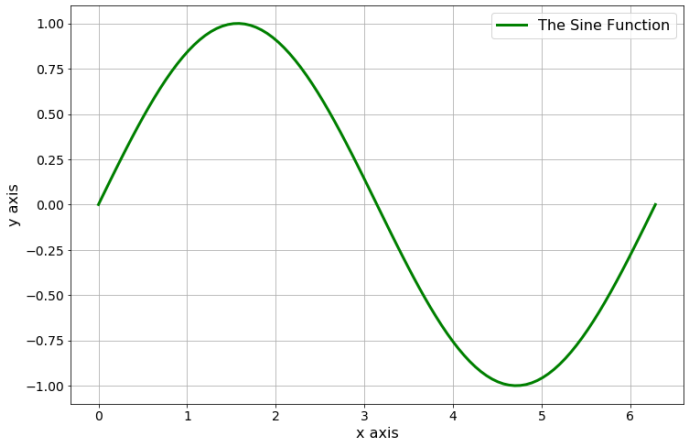


Figure A.1: The sine function

Example A.46. Now let's 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

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

on the domain $x \in [0, 1]$ with 100 equally spaced points. We'll 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 Title")
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.show()
```

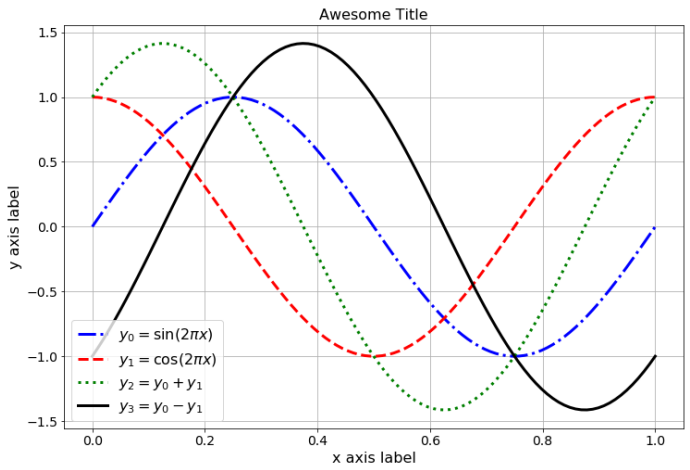


Figure A.2: Plots of the sine, cosine, and sums and differences.

Notice that the legend was placed automatically. There are ways to control the placement of the legend if you wish, but for now just let Python and `matplotlib` have control over the placement.

Example A.47. Now let's create the same plot with slightly different code.

The `plot` command 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 Title")
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.show()
```

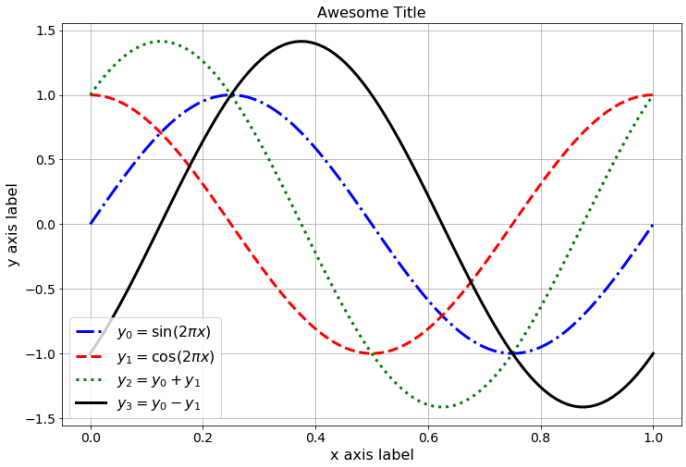


Figure A.3: A second plot of the sine, cosine, and sums and differences.

Exercise A.19. Plot the functions $f(x) = x^2$, $g(x) = x^3$, and $h(x) = x^4$ on the same axes. Use the domain $x \in [0, 1]$ and the range $y \in [0, 1]$. Put a grid, a legend, a title, and appropriate labels on the axes.

A.6.2 Subplots

It is often very handy to place plots side-by-side or as some array of plots. The `subplots` command allows us that control. The main idea is that we are setting up a matrix of blank plots and then populating the axes with the plots that we want.

Example A.48. Let's 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'll 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. The last input for the `subplots` command is the size of the figure (this is really just so that it shows up well in Jupyter Notebooks – spend some time playing with the figure size to get it to look right).
- 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, figsize = (10,5))

# 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()
```

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

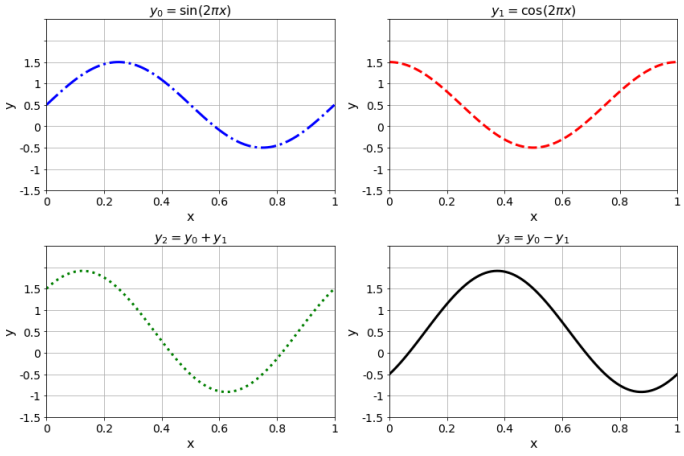


Figure A.4: An example of subplots

Exercise A.20. 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, but sure that each plot has a grid, appropriate labels, an appropriate title, and the overall figure has a title.

A.6.3 Logarithmic Scaling with `semilogy`, `semilogx`, and `loglog`

It is occasionally useful to scale an axis logarithmically. This arises most often when we're 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'll use this often in numerical methods.

Example A.49. In this example we'll plot the function $y = 10^{-0.01x}$ on a regular (linear) scale and on a logarithmic scale on the y axis. Use the interval $[0, 500]$.

```
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, figsize = (10,5))

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, 'k--')
axis[1].grid()
axis[1].set_title("Logarithmically scaled y axis")
axis[1].set_xlabel("x")
axis[1].set_ylabel("Log(y)")
plt.show()
```

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. Pay careful attention to the subtle changes in the following code.

```
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, figsize = (10,5))

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].plot(x,y, 'k--') # <----- Notice the change here
axis[1].set_yscale("log") # <----- And we added this line
axis[1].grid()
axis[1].set_title("Logarithmically scaled y axis")
axis[1].set_xlabel("x")
axis[1].set_ylabel("Log(y)")
```

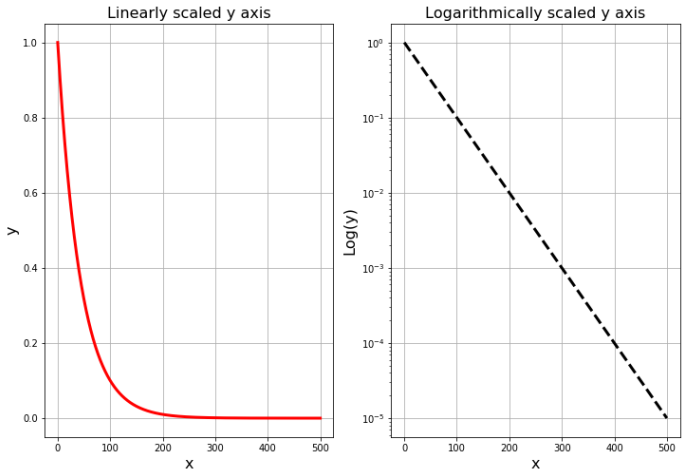


Figure A.5: An example of using logarithmic scaling.

Exercise A.21. Plot the function $f(x) = x^3$ for $x \in [0, 1]$ on linearly scaled axes, logarithmic axis in the y direction, logarithmically scaled axes in the x direction, and a log-log plot with logarithmic scaling on both axes. Use `subplots` to put your plots side-by-side. Give appropriate labels, titles, etc.

A.7 Symbolic Python with `sympy`

In this section we will learn the tools necessary to do symbolic mathematics in Python. The relevant package is `sympy` (symbolic Python) and it works much like Mathematica, Maple, or MATLAB's symbolic toolbox. That being said, Mathematica and Maple are designed to do symbolic computation in the fastest and best possible ways, so in some sense, `sympy` is a little step-sibling to these much bigger pieces of software. Remember: Python is free, and this is a book on numerical analysis – we will not be doing much symbolic work in this particular class, but these tools do occasionally come in handy.

Let's import `sympy` in the usual way. We will use the nickname `sp` (just like we used `np` for `numpy`). This is not a standard nickname in the Python literature, but it will suffice for our purposes.

Exercise A.22. Load `sympy` and use the `dir()` command to see what functions are inside the `sympy` library.

Example A.50. If you include the command `sp.init_printing()` after you load `sympy` you will get some nice LaTeX style output in your Jupyter Notebooks.

A.7.1 Symbolic Variables with `symbols`

When you are working with symbolic variables you have to tell Python that that's what you're doing. In other words, we actually have to type-cast the variables when we name them. Otherwise Python won't know what to do with them – we need to explicitly tell it that we are working with symbols!

Example A.51. Let's define the variable x as a symbolic variable. Then we'll define a few symbolic expressions that use x as a variable.

```
import sympy as sp
x = sp.Symbol('x') # note the capitalization
```

Now we'll define the function $f(x) = (x+2)^3$ and spend the next few examples playing with it.

```
f = (x+2)**3 # A symbolic function
print(f)
```

```
## (x + 2)**3
```

Notice that the output of these lines of code is not necessarily very nicely formatted as a symbolic expression. What we would really want to see is $(x+2)^3$.

If you include the code `sp.init_printing()` after you import the `sympy` library then you should get nice LaTeX style formatting in your answers.

Example A.52. Be careful that you are using symbolically defined function along with your symbols. For example, see the code below:

```
# this line gives an error since it doesn't know which "sine" to use.
g = sin(x)
```

```
import sympy as sp
x = sp.Symbol('x')
g = sp.sin(x) # this one works
print(g)
```

```
## sin(x)
```

A.7.2 Symbolic Algebra

One of the primary purposes of doing symbolic programming is to do symbolic algebra (the other is typically symbolic calculus). In this section we'll look at a few of the common algebraic exercises that can be handled with `sympy`.

Example A.53 (symbolic expand)**.** Expand the function $f(x) = (x+2)^3$. In other words, multiply this out fully so that it is a sum or difference of monomials instead of the cube of a binomial.

```
import sympy as sp
x = sp.Symbol('x')
f = (x+2)**3
sp.expand(f) # do the multiplication to expand the polynomial
```

```
## x**3 + 6*x**2 + 12*x + 8
```

Example A.54 (symbolic factoring)**.** We will factor the polynomial $h(x) = x^2 + 4x + 3$.

```
import sympy as sp
x = sp.Symbol('x')
h = x**2 + 4*x + 3
sp.factor(h) # factor this polynomial
```

```
## (x + 1)*(x + 3)
```

Example A.55 (Trigonometric Simplification)**.** The `sympy` package knows how to work with trigonometric identities. In this example we show how `sympy` expands $\sin(a+b)$.

```
import sympy as sp
a, b = sp.symbols('a b')
j = sp.sin(a+b)
sp.expand(j, trig=True) # Trig identities are built in!
```

```
## sin(a)*cos(b) + sin(b)*cos(a)
```

Example A.56 (Symbolic Simplification)**.** In this example we will simplify the function $g(x) = x^3 + 5x^3 + 12x^2 + 1$.

```
import sympy as sp
x = sp.Symbol('x')
g = x**3 + 5*x**3 + 12*x**2 + 1
sp.simplify(g) # Simplify some algebraic expression
```

```
## 6*x**3 + 12*x**2 + 1
```

Example A.57. In this example we'll simplify an expression that involves trigonometry.

```
import sympy as sp
x = sp.Symbol('x')
sp.simplify( sp.sin(x) / sp.cos(x)) # simplify a trig expression.
```

```
## tan(x)
```

Example A.58 (Symbolic Equation Solving)**.** The primary goal of many algebra problems is to solve an equation. We will dedicate more time to algebraic equation solving later in this section, but this example gives a simple example of how it works in `sympy`.

We want to solve the equation $x^2 + 4x + 3 = 0$ for x.

```
import sympy as sp
x = sp.Symbol('x')
h = x**2 + 4*x + 3
sp.solve(h,x)
```

```
## [-3, -1]
```

As expected, the roots of the function $h(x)$ are $x = -3$ and $x = 1$ since $h(x)$ factors into $h(x) = (x+3)(x-1)$.

A.7.3 Symbolic Function Evaluation

In `sympy` we cannot simply just evaluate functions as we would on paper. Let's say we have the function $f(x) = (x+2)^3$ and we want to find $f(5)$. We would say that we "substitute 5 into f for x", and that is exactly what we have to tell Python. Unfortunately we cannot just write `f(5)` since that would mean that `f` is a Python function and we are sending the number 5 into that function. This is an unfortunate double-use of the word "function", but stop and think about it for a second: When we write `f = (x+2)**3` we are just telling Python that `f` is a symbolic expression in terms of the symbol `x`, but we did not use `def` to define it as a function as we did for all other function.

Example A.59. The following code is what the mathematicians in us would like to do:

```
import sympy as sp
x = sp.Symbol('x')
f = (x+2)**3
f(5) # This gives an error!
```

... but this is how it should be done:

```
import sympy as sp
x = sp.Symbol('x')
f = (x+2)**3
f.subs(x,5) # This actually substitutes 5 for x in f
```

```
## 343
```

A.7.4 Symbolic Calculus

The `sympy` package has routines to take symbolic derivatives, antiderivatives, limits, and Taylor series just like other computer algebra systems.

A.7.4.1 Derivatives

The `diff` command in `sympy` does differentiation: `sp.diff(function, variable, [order])`.

Take careful note that `diff` is defined both in `sympy` and in `numpy`. That means that there are symbolic and numerical routines for taking derivatives in Python ...and we need to tell our instance of Python which one we're working with every time we use it.

Example A.60 (Symbolic Differentiation)**.** In this example we'll differentiate the function $f(x) = (x+2)^3$.

```
import sympy as sp
x = sp.Symbol('x') # Define the symbol x
f = (x+2)**3 # Define a symbolic function f(x) = (x+2)^3
df = sp.diff(f,x) # Take the derivative of f and call it "df"
print("f(x) = ", f)
```

```
## f(x) =  (x + 2)**3
```

```
print("f'(x) = ",df)
```

```
## f'(x) =  3*(x + 2)**2
```

```
print("f'(x) = ", sp.expand(df))
```

```
## f'(x) =  3*x**2 + 12*x + 12
```

Example A.61. Now let's get the first, second, third, and fourth derivatives of the function `f`.

```
import sympy as sp
x = sp.Symbol('x') # Define the symbol x
f = (x+2)**3 # Define a symbolic function f(x) = (x+2)^3
df = sp.diff(f,x,1) # first derivative
ddf = sp.diff(f,x,2) # second deriative
dddf = sp.diff(f,x,3) # third deriative
ddddf = sp.diff(f,x,4) # fourth deriative
print("f'(x) = ",df)
```

```
## f'(x) =  3*(x + 2)**2
```

```
print("f''(x) = ",sp.simplify(ddf))
```

```
## f''(x) =  6*x + 12
```

```
print("f'''(x) = ",sp.simplify(dddf))
```

```
## f'''(x) =  6
```

```
print("f''''(x) = ",sp.simplify(ddddf))
```

```
## f''''(x) =  0
```

Example A.62. Now let's do some partial derivatives. The `diff` command is still the right tool. You just have to tell it which variable you're working with.

```
import sympy as sp
x, y = sp.symbols('x y') # Define the symbols
```

```
f = sp.sin(x*y) + sp.cos(x**2) # Define the function
fx = sp.diff(f,x)
fy = sp.diff(f,y)
print("f(x,y) = ", f)
```

```
## f(x,y) =  sin(x*y) + cos(x**2)
```

```
print("f_x(x,y) = ", fx)
```

```
## f_x(x,y) =  -2*x*sin(x**2) + y*cos(x*y)
```

```
print("f_y(x,y) = ", fy)
```

```
## f_y(x,y) =  x*cos(x*y)
```

Example A.63. It is worth noting that when you have a symbolically defined function you can ask `sympy` to give you the LaTeX code for the symbolic function so you can use it when you write about it.

```
import sympy as sp
x, y = sp.symbols('x y') # Define the symbols
f = sp.sin(x*y) + sp.cos(x**2) # Define the function
sp.latex(f)
```

```
## '\\sin{\\left(x y \\right)} + \\cos{\\left(x^{2} \\right)}'
```

A.7.4.2 Integrals

For integration, the `sp.integrate` tool is the command for the job: `sp.integrate(function, variable)` will find an antiderivative and `sp.integrate(function, (variable, lower, upper))` will evaluate a definite integral.

The `integrate` command in `sympy` accepts a symbolically defined function along with the variable of integration and optional bounds. If the bounds aren't given then the command finds the antiderivative. Otherwise it finds the definite integral.

Example A.64. Find the antiderivative of the function $f(x) = (x+2)^3$.

```
import sympy as sp
x = sp.Symbol('x')
f = (x+2)**3
F = sp.integrate(f,x)
print(F)
```

```
## x**4/4 + 2*x**3 + 6*x**2 + 8*x
```

The output of these lines of code is the expression $\frac{x^4}{4} + 2x^3 + 6x^2 + 8x$ which is indeed the antiderivative.

Example A.65. Consider the multivariable antiderivative

$$\int \sin(xy) + \cos(x) dx.$$

The `sympy` package deals with the second variable just as it should.

```
import sympy as sp
x, y = sp.symbols('x y')
g = sp.sin(x*y) + sp.cos(x)
G = sp.integrate(g,x)
print(G)
```

```
## Piecewise((-cos(x*y)/y, Ne(y, 0)), (0, True)) + sin(x)
```

It is apparent that `sympy` was sensitive to the fact that there was some trouble at $y = 0$ and took care of it with a piece wise function.

Example A.66. Consider the integral

$$\int_0^{\pi} \sin(x) dx.$$

Notice that the variable and the bounds are sent to the `integrate` command as a tuple. Furthermore, notice that we had to send the symbolic version of π instead of any other version (e.g. `numpy`).

```
import sympy as sp
x = sp.Symbol('x')
sp.integrate( sp.sin(x), (x,0,sp.pi))
```

```
## 2
```

Example A.67. This is a fun one. Let's do the definite integral

$$\int_{-\infty}^{\infty} e^{-x^2} dx.$$

We have to use the "infinity" symbol from `sympy`. It is two lower-case O's next to each other: `oo`. It kind of looks like and infinity I suppose.

```
import sympy as sp
x = sp.Symbol('x')
sp.integrate( sp.exp(-x**2) , (x, -sp.oo, sp.oo))
```

```
## sqrt(pi)
```

A.7.4.3 Limits

The `limit` command in `sympy` takes symbolic limits: `sp.limit(function, variable, value, [direction])`

The direction (left or right) is optional and if you leave it off then the limit is considered from both directions.

Example A.68. Let's take the limit

$$\lim_{x\to 0} \frac{\sin(x)}{x}.$$

```
import sympy as sp
x = sp.Symbol('x')
sp.limit( sp.sin(x)/x, x, 0)
```

```
## 1
```

Example A.69. Let's do the difference quotient

$$\lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$$

for the function $f(x) = (x+2)^3$. Taking the limit should give the derivative so we'll check that the `diff` command gives us the same thing using `==` . . . warning!

```
import sympy as sp
x = sp.Symbol('x')
f = (x+2)**3
print(sp.diff(f,x))
```

```
## 3*(x + 2)**2
```

```
h = sp.Symbol('h')
df = sp.limit( (f.subs(x,x+h) - f) / h , h , 0 )
print(df)
```

```
## 3*x**2 + 12*x + 12
```

```
print(df == sp.diff(f,x)) # notice that these are not "symbolically" equal
```

```
## False
```

```
print(df == sp.expand(sp.diff(f,x))) # but these are
```

```
## True
```

Notice that when we check to see if two symbolic functions are equal they must be in the same exact symbolic form. Otherwise `sympy` won't recognize them as actually being equal even though they are mathematically equivalent.

Exercise A.23. Define the function $f(x) = 3x^2 + x\sin(x^2)$ symbolically and then do the following:

a. Evaluate the function at $x = 2$ and get symbolic and numerical answers.
b. Take the first and second derivative
c. Take the antiderivative
d. Find the definite integral from 0 to 1
e. Find the limit as x goes to 3

A.7.4.4 Taylor Series

The `sympy` package has a tool for expanding Taylor Series of symbolic functions: `sp.series( function, variable, [center], [num terms])`.

The center defaults to 0 and the number of terms defaults to 5.

Example A.70. Find the Taylor series for $f(x) = e^x$ centered at $x = 0$ and centered at $x = 1$.

```
import sympy as sp
x = sp.Symbol('x')
sp.series( sp.exp(x),x)
```

```
## 1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + O(x**6)
```

```
import sympy as sp
x = sp.Symbol('x')
sp.series( sp.exp(x), x, 1, 4) # expand at x=1 (4 terms)
```

```
## E + E*(x - 1) + E*(x - 1)**2/2 + E*(x - 1)**3/6 + O((x - 1)**4, (x, 1))
```

Finally, if we want more terms then we can send the number of desired terms to the `series` command.

```
import sympy as sp
x = sp.Symbol('x')
sp.series( sp.exp(x), x, 0, 3) # expand at x=0 and give 3 terms
```

```
## 1 + x + x**2/2 + O(x**3)
```

A.7.5 Solving Equations Symbolically

One of the big reasons to use a symbolic toolboxes such as `sympy` is to solve algebraic equations exactly. This isn't always going to be possible, but when it is we get some nice results. The `solve` command in `sympy` is the tool for the job: `sp.solve( equation, variable )`

The equation doesn't actually need to be the whole equation. For any equation-solving problem we can always re-frame it so that we are solving $f(x) = 0$ by subtracting the right-hand side of the equation to the left-hand side. Hence we can leave the equal sign and the zero off and `sympy` understands what we're doing.

Example A.71. Let's solve the equation $x^2 - 2 = 0$ for x. We know that the roots are $\pm\sqrt{2}$ so this should be pretty trivial for a symbolic solver.

```
import sympy as sp
x = sp.Symbol('x')
sp.solve( x**2 - 2, x)
```

```
## [-sqrt(2), sqrt(2)]
```

Example A.72. Now let's solve the equation $x^4 - x^2 - 1 = 0$ for x. You might recognize this as a quadratic equation in disguise so you can definitely do it by hand ... if you want to. (You could also recognize that this equation is related to the golden ratio!)

```
import sympy as sp
x = sp.Symbol('x')
sp.solve( x**4 - x**2 - 1, x)
```

Run the code yourself to see the output. In nicer LaTeX style formatting, the answer is

$$\left[-i\sqrt{-\frac{1}{2}+\frac{\sqrt{5}}{2}},\ i\sqrt{-\frac{1}{2}+\frac{\sqrt{5}}{2}},\ -\sqrt{\frac{1}{2}+\frac{\sqrt{5}}{2}},\ \sqrt{\frac{1}{2}+\frac{\sqrt{5}}{2}}\right]$$

Notice that `sympy` has no problem dealing with the complex roots.

In the previous example the answers may be a bit hard to read due to their symbolic form. This is particularly true for far more complicated equation solving problems. The next example shows how you can loop through the solutions and then print them in decimal form so they are a bit more readable.

Example A.73. We will again solve the equation $x^4 - x^2 - 1 = 0$ for x, but this time we will output the answers as floating point decimals. We are using the `N` command to convert from symbolic to numerical.

```
import sympy as sp
x = sp.Symbol('x')
soln = sp.solve( x**4 - x**2 - 1, x)
for j in range(0, len(soln)):
    print(sp.N(soln[j]))
```

```
## -0.786151377757423*I
## 0.786151377757423*I
## -1.27201964951407
## 1.27201964951407
```

The `N` command gives a numerical approximation for a symbolic expression (this is taken straight from Mathematica!).

Exercise A.24. Give the exact and floating point solutions to the equation $x^4 - x^2 - x + 5 = 0$.

When you want to solve a symbolic equation numerically you can use the `nsolve` command. This will do something like Newton's method in the background. You need to give it a starting point where it can look for you the solution to your equation: `sp.nsolve( equation, variable, intial guess )`

Example A.74. Let's solve the equation $x^3 - x^2 - 2$ for x both symbolically and numerically. The numerical solution with `nsolve` will search for the solution near $x = 1$.

```
import sympy as sp
x = sp.Symbol('x')
ExactSoln = sp.solve(x**3 - x**2 - 2, x) # symbolic solution
print(ExactSoln)
```

Run the code yourself to see the exact solution. In nicer LaTeX style formatting

the answer is:

$$\left[\frac{1}{3}+\left(-\frac{1}{2}-\frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{\sqrt{87}}{9}+\frac{28}{27}}+\frac{1}{9\left(-\frac{1}{2}-\frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{\sqrt{87}}{9}+\frac{28}{27}}},\right.$$
$$\frac{1}{3}+\frac{1}{9\left(-\frac{1}{2}+\frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{\sqrt{87}}{9}+\frac{28}{27}}}+\left(-\frac{1}{2}+\frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{\sqrt{87}}{9}+\frac{28}{27}},$$
$$\left.\frac{1}{9\sqrt[3]{\frac{\sqrt{87}}{9}+\frac{28}{27}}}+\frac{1}{3}+\sqrt[3]{\frac{\sqrt{87}}{9}+\frac{28}{27}}\right]$$

which is rather challenging to read. We can give all of the floating point approximations with the following code.

```
import sympy as sp
x = sp.Symbol('x')
ExactSoln = sp.solve(x**3 - x**2 - 2, x) # symbolic solution
print("First Solution: ",sp.N(ExactSoln[0]))
```

```
## First Solution:  -0.347810384779931 - 1.02885225413669*I
```

```
print("Second Solution: ",sp.N(ExactSoln[1]))
```

```
## Second Solution:  -0.347810384779931 + 1.02885225413669*I
```

```
print("Third Solution: ",sp.N(ExactSoln[2]))
```

```
## Third Solution:  1.69562076955986
```

If we were only looking for the floating point real solution near $x = 1$ then we could just use `nsolve`.

```
import sympy as sp
x = sp.Symbol('x')
NumericalSoln = sp.nsolve(x**3 - x**2 - 2, x, 1) # solution near x=1
print(NumericalSoln)
```

```
## 1.69562076955986
```

Exercise A.25. Solve the equation

$$x^3 \ln(x) = 7$$

and give your answer both symbolically and numerically.

A.7.6 Symbolic Plotting

In this final section we will show how to make plots of symbolically defined functions. Be careful here. There are times when you want to plot a symbolically defined function and there are times when you want to plot data: `sp.plot( function, (variable, left, right) )`

It is easy to get confused since they both use the `plot` function in their own packages (`sympy` and `matplotlib` respectively).

Note: For MATLAB users, the `sympy.plot` command is similar to MATLAB's `ezplot` command or `fplot` command.

In numerical analysis we do not often need to make plots of symbolically defined functions. There is more that could be said about `sympy`'s plotting routine, but since it won't be used often in this text it doesn't seem necessary to give those details here. When you need to make a plot just make a careful consideration as to whether you need a symbolic plot (with `sympy`) or a plot of data points (with `matplotlib`).

Example A.75. Let's get a quick plot of the function $f(x) = (x+2)^3$ on the domain $x \in [-5, 2]$.

```
import sympy as sp
x = sp.Symbol('x')
f = (x+2)**3
sp.plot(f,(x,-5,2))
```

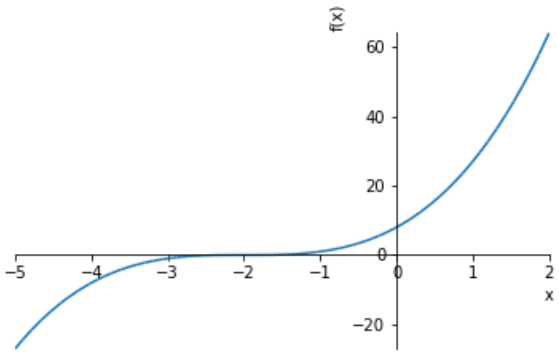


Figure A.6: A plot of a symbolically defined function.

Example A.76. Multiple plots can be done at the same time with the `sympy.plot` command.

Plot $f(x) = (x+2)^3$ and $g(x) = 20\cos(x)$ on the same axes on the domain $x \in [-5, 2]$.

```
import sympy as sp
x = sp.Symbol('x')
f = (x+2)**3
g = 20*sp.cos(x)
sp.plot(f, g, (x,-5,2))
```

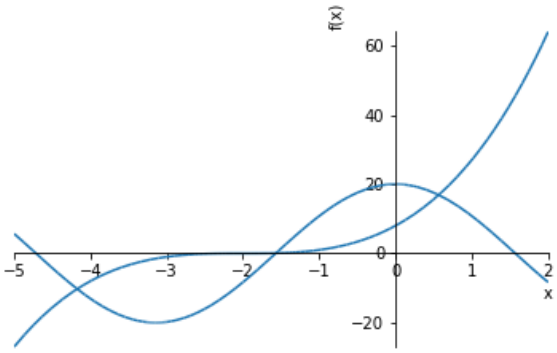


Figure A.7: A second plot of symbolically defined functions.

Exercise A.26. Make a symbolic plot of the function $f(x) = x^3 \ln(x) - 7$ on the domain $[0, 3]$.

Appendix B

Mathematical Writing

This appendix is designed to give you helpful hints for the writing requires of the homework and the projects. You will find that mathematical writing is different than writing for literature, for general consumption, or perhaps other scientific disciplines. Pay careful attention to the conventions mentioned in this chapter when you write the results for your projects especially.

A few words of advice:

- Do all of the math first without worrying too much about the writing.
- When you have your mathematical results you can start writing.
- Write the introduction last since at that point you know what you've written.
- You will spend more time creating well-crafted figures than any other part of a mathematical writing project. Expect the figures to take at least as long as the math, the writing, and the editing.

B.1 The Paper

Write your work in a formal paper that is typed and written at a college level using appropriate mathematical typesetting. This paper must be organized into sections, starting with a Summary or Abstract, followed by an Introduction, and ending with Conclusions and References. Each of these sections should begin with these headings in a large bold font (using the LaTeX `\section` and `\subsection` commands where appropriate). Within sections I would suggest using subheadings to further organize things and aid in clarity.

B.2 Figures and Tables

Figures and tables are a very important part of these projects. Never break tables or figures across pages. Each figure or table must fit completely onto one sheet of paper. If your table has too much information to fit onto one sheet, divide it into two separate tables. In addition to the figure, this sheet must contain the figure number, the figure title, and a brief caption; for example "Figure 2: A plot of heating oil price versus time from Model F1. We see that the effects of seasonal variation in price are dominated by random fluctuations." In the text, refer to the figure/table by its number. For example in the text you might say "As we see in Figure 2, in model F1 the effects of seasonal variation in price are dominated by random fluctuations." Every figure and table must be mentioned (by number) somewhere in the text of your paper. If you do not refer to it anywhere in the text, then you do not need it, and subsequently it will be ignored.

Think of figures and tables as containing the evidence that you are using to support the point you are trying to make with your paper. Always remember that the purpose of a figure or a table is to show a pattern, and when someone looks at the figure this pattern should be obvious. Figures should not be cluttered and confusing: They should make things very clear. Always label the horizontal and vertical axes of plots.

B.3 Writing Style

The real goal of mathematical writing is to take a complex and intricate subject and to explain it so simply and so plainly that the results are obvious for everyone. I want your paper to demonstrate that you not only did the right calculations, but that you understand what you did and why your methods worked.

Write this paper using the word 'we' instead of 'I.' For example: "First we calculate the sample mean." This 'we' refers to you and the reader as you guide the reader through the work that you've done. Also please avoid the word "prove" or "proof." Numerical methods usually deal with approximations, not absolutes, and in mathematics we reserve the word "prove" for things that are absolutely 100% certain. Often the word "test" can be used instead of "prove."

B.4 Tips For Writing Clear Math

At this point you know just enough mathematics and LaTeX to be dangerous. It is time to clean up your act and teach you some of the formalities about writing mathematics. The following sections stem from a document that I give all of my upper level mathematics students.

B.4.1 Audience

The following suggestions will help you to submit properly written homework solutions, papers, projects, labs, and proofs. The goal of any writing is to clearly communicate ideas to another person. Remember that the other person may even be your future self. When you write for another person, you will need to include ideas that may be in your mind but omitted when you are writing a rough draft on scratch paper. If you keep your intended audience in mind, you will produce higher quality work. For a course in mathematics, the intended audience is usually your instructor, your classmates, or a student grader. This implies that your task is to show that you thoroughly understand your solution. Consequently, you should routinely include more details.

One rule of thumb must prevail throughout all mathematical writing:

> When you read a mathematical solution out loud it needs to make sense as grammatically correct English writing. This includes reading all of the symbols with the proper language.
>
> **Don't forget that mathematics is a language that is meant to be spoken and read just like works of literature!**

B.4.2 How To Make Mathematics Readable – 10 Things To Do

1. When read aloud, the text and formulas must form complete English sentences. If you get lost, say aloud what you mean and write it down.
2. Every mathematical statement must be complete and meaningful. Avoid fragments.
3. If a statement is something you want to prove or something you assume temporarily, e.g., to discuss possible cases or to get a contradiction, say so clearly. Otherwise, anything you put down must be a true statement that follows from your up front assumptions.
4. Write what your plan is. It will also help you focus on what to do.
5. There must be sufficient detail to verify your argument. If you do not have the details, you have no way of knowing if what you wrote is correct or not. Keep the level of detail uniform.
6. If you are not sure, even slightly, about something, work out the details on the side with utmost honesty, going as deep as necessary. Decide later how much detail to include.
7. Do not write irrelevant things just to fill paper and show you know something.

8. Your argument should flow well. Make the reading easy. Logical and intuitive notation matters.

9. Keep in mind what the problem is and make sure you are not doing something else. Many problems are solved and proofs done simply by understanding what is what.

10. The state of mind when you are inventing a solution is completely different from the mode of work when you are writing the solution down and verifying it. Learn how to go back and forth between the two. The act of typing your solutions forces you to iterate over this process but remember that the process isn't done until you've proofread what you typed.

B.4.3 Some Writing Tips

Use sentences: The feature that best distinguishes between a properly written mathematical exposition and a piece of scratch paper is the use (or lack) of sentences. Properly written mathematics can be read in the same manner as properly written sentences in any other discipline. Sentences force a linear presentation of ideas. They provide the connections between the various mathematical expressions you use. This linearity will also keep you from handing in a page with randomly scattered computations with no connections. The sentences may contain both words and mathematical expressions. Keep in mind that the way your present your solution may be different than the way that you arrived at the solution. It is imperative that you work problems on scratch paper first before formally writing the solution.

The following extract illustrates these ideas.

> Let n be odd. Then Definition 3.10 indicates that there does not exist an integer, k, such that $n = 2k$. That is, n is not divisible by 2. The Quotient– Remainder theorem asserts that n can be uniquely expressed in the form $n = 2q + r$, where r is an integer with $0 \leq r < 2$. Thus, $r \in \{0, 1\}$. Since n is not divisible by 2, the only admissible choice is $r = 1$. Thus, $n = 2q + 1$, with q an integer.

Read out loud: The sentences you write should read well out loud. This will help you to avoid some common mistakes. Avoid sentences like:

> Suppose the graph has n number of vertices.
> The piggy bank contains n amount of coins.

If you substitute an actual number for n (such as 4 or 6) and read these out loud they will sound wrong (because they are wrong). The variable n is already a numeric variable so it should be read just like an actual number. The correct versions are:

Suppose the graph has n vertices.
(Read this as: "Suppose the graph has en vertices".)
The piggy bank contains n coins.

You should also avoid sentences like:

From the previous computation $x = 5$ is true.

A better way to say this is:

From the previous computation we see that $x = 5$.

When you read the equal sign as part of the sentence you realize that there is no reason to write "is true".

= is NOT a conjunction: The mathematical symbol = is an assertion that the expression on its left and the expression on its right are equal. Do not use it as a connection between steps in a series of calculations. Use words for this purpose. Here is an example that misuses the = symbol when solving the equation $3x = 6$:

$$\text{Incorrect!} \qquad 3x = \underbrace{6 = \frac{3x}{3}}_{\text{false!}} = \frac{6}{3} = x = 2$$

One proper way to write his is:
$3x = 6$. Dividing both sides by 3 leads to $\frac{3x}{3} = \frac{6}{3}$, which simplifies to $x = 2$.

" $\Longrightarrow$ " means "implies": The double arrow " $\Longrightarrow$ " means that the statement on the left logically implies the statement on the right. This symbol is often misused in place of the "=" sign.

Do not merge steps: Suppose you need to calculate the final price for a \$20 item with 7% sales tax. One strategy is to first calculate the tax, then add the \$20. Here is an incorrect way to write this.

$$\text{Incorrect!} \qquad 20 \cdot 0.07 \underbrace{=}_{false!} 1.4 + 20 \underbrace{=}_{false!} \$21.4.$$

The main problem (besides the magically-appearing dollar sign at the end) is that $20 \cdot 0.07 \neq 1.4 + 20$. The writer has taken the result of the multiplication (1.4) and merged directly into the addition step, creating a lie (since $1.4 \neq 21.4$). The calculations could be written as:

$$\$20 \cdot 0.07 = \$1.40 \text{ so the total price is } \$1.40 + \$20 = \$21.40$$

Avoid ambiguity: When in doubt, repeat a noun rather using unspecific words like "it" or "the". For example, in the sentences

> Let G be a simple graph with $n \geq 2$ vertices that is not complete and let G be its complement. Then it must contain at least one edge.

there is some ambiguity about whether "it" refers to G or to the complement of G. The second sentence is better written as "Then G must contain at least one edge".

Use Proper Notation: There are many notational conventions in mathematics. You need to follow the accepted conventions when using notation. For example, A summation or integral symbol always needs something to act on. The expressions

$$\sum_{i=1}^{n} \qquad \int_a^b$$

by themselves are meaningless. The expressions

$$\sum_{i=1}^{n} a_n \qquad \int_a^b f(x)dx$$

have well-understood meanings.

An another example,

$$\underbrace{\lim_{h\to 0} = \frac{2x+h}{2}}_{\text{incorrect!}} = \frac{2x}{2} = x$$

is incorrect. It should be written

$$\lim_{h\to 0} \frac{2x+h}{2} = \frac{2x}{2} = x$$

Parenthesis are important: Parenthesis show the grouping of terms, and the omission of parenthesis can lead to much unneeded confusion. For example,

$$x^2 + 5 \cdot x - 3 \quad \text{is very different than} \quad \left(x^2+5\right) \cdot (x-3)\,.$$

This is very important in differentiation and summation notation:

$$\frac{d}{dx}\sin(x) + x^2 \quad \text{is not the same as} \quad \frac{d}{dx}\left(\sin(x) + x^2\right)$$

$$\sum_{k=1}^{n} 2k + 3 \quad \text{is not the same as} \quad \sum_{k=1}^{n} (2k+3)$$

Label and reference equations: When you need to refer to an equation later it is common practice to label the equation with a number and then to refer to this equation by that number. This avoids ambiguity and gives the

reader a better chance at understanding what you're writing. Furthermore, avoid using words like "below" and "above" since the reader doesn't really know where to look. One implication to this style of referencing is that you should never reference an equation before you define it.

Incorrect:

In the equation below we consider the domain $x \in (-1, 1)$

$$f(x) = \sum_{j=1}^{\infty} \frac{x^n}{n!}.$$

Correct:

Consider the summation

$$f(x) = \sum_{j=1}^{\infty} \frac{x^n}{n!}.$$

In this equation we are assuming the domain $x \in (-1, 1)$.

"Timesing": The act of multiplication should not be called "timesing" as in "I can times 3 and 5 to get 15". The correct version of this sentence is "I can multiply 3 and 5 to get 15". The phrase "3 times 5 is 15", on the other hand, is correct and is likely the root of the confusion. The mathematical operation being performed is not called "timesing". It seems as if this is an unfortunate carry over from childhood when a child hears "3 times 5", sees "3×5", and then incorrectly associates the symbol "$\times$" with the word multiply in the statement "I can multiply 3 and 5 to get 15".

B.4.4 Mathematical Vocabulary

Function: The word function can be used to refer just to the name of a function, such as "The function $s(t)$ gives the position of the particle as a function of time." Or function can refer to both the function name and the rule that describes the function. For example, we could elaborate and say, "The function $s(t) = t2 - 3t$ gives the position of the particle as a function of time." Notice that both times the word function is used twice, where the second usage is describing the mathematical nature of the relationship between time and position. (Remember that if position can be described as a function of time, then the position can be uniquely determined from the time.)

Equation: To begin with, an equation must have an equal sign (=), but just having an equal sign isn't enough to deserve the name equation. Generally, an equation is something that will be used to solve for a particular variable, and/or it expresses a relationship between variables. So you might say,

"We solved the equation $x + y = 5$ for x to find that $x = 5 - y$," or you might say "The relationship between the variables can be expressed with the following equation: $xy = 2z$."

Formula: A formula might in fact be an equation or even a function, but generally the word formula is used when you are going to substitute numbers for some or all of the variables. For example, we might say, "The formula for the area of a circle is $A = \pi r^2$. Since $r = 2$ in this case, we find $A = \pi 2^2 = 4\pi$." The bottom line: If you're going to use algebra to solve for a variable, call it an equation. If you're going to use it exactly as it is and just put in numbers for the variables, then call it a formula.

Definition: A definition might be any of the above, but it is specifically being used to define a new term. For example, the definition of the derivative of a function f at a point a is

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}.$$

Now this does give us a formula to use to compute the derivative, but we prefer to call this particular formula a definition to highlight the fact that this is what we have chosen the word derivative to mean.

Expression: The word expression is used when there isn't an equal sign. You probably won't need this word very often, but it is used like this: "The factorization of the expression $x^2 - x - 6$ is $(x - 3)(x + 2)$."

Solve/Evaluate: Equations are solved, whereas functions are evaluated. So you would say, "We solved the equation for x," but you would say "We evaluated the function at $x = 5$ and found the function value to be 26."

Add Subtract vs Plus Minus: The word subtract is used when discussing what needs to be done: "Subtract two from five to get three." Add is used similarly: "Add two and five to get seven." Minus is used when reading a mathematical equation or expression. For example, the equation $x - y = 5$ would be read as "x minus y is equal to five". Plus is used similarly. So the equation $x + y = 5$ would be read as "x plus y is equal to five". Some things we don't say are "We plus 2 and 5 to get 7" or "We minus x from both sides of the equation."

Number/Amount: The word number is used when referring to discrete items, such as "there were a large number of cougars", or "there are a large number of books on my shelf". The word amount is used when referring to something that might come in a pile, such as "that is a huge amount of sand!" or, "I only use a small amount of salt when I cook".

Many/Much: These words are used in much the same way as number and amount, with many in place of number and much in place of amount. For example, we might say, "There aren't as many cougars here as before", or "I don't use as much salt as you do."

Fewer/Less: These are the diminutive analogues of many and much. So, "There are fewer cougars here than before", or "You use less salt than I do."

B.5 Sensitivity Analysis

(This section is paraphrased partly from Dianne O'Leary's book *Computing in Science & Engineering* and partly from Mark Meerschaert's text *Mathematical Modeling.*)

In contrast to to classroom exercises, the real world rarely presents us with a problem in which the data is known with absolute certainty. Some parameters (such as π) we can define with certainty, and others (such as Planck's constant $\hbar$) we know to high precision, but most data is measured and therefore contains measurement error.

Thus, what we really solve isn't the problem we want, but some *nearby* problem, and in addition to reporting the computed solution we really need to report a bound on either

- the difference between the true solution and the computed solution, or
- the difference between the problem we solved and the problem we wanted to solve.

This need occurs throughout computational science. For example,

- If we compute the resonant frequencies of a model of a building, we want to know how these frequencies change if the load within the building changes.
- If we compute the stresses on a bridge, we want to know how sensitive these values are to changes that might occur as the bridge ages.
- If we develop a model for our data and fit the parameters using least squares, we want to know how much the parameters would change if the data were wiggled within the uncertainty limits.

One of the best ways to measure the sensitivity of a parameter k on an output x is to measure the ratio between the relative change in x to the relative change in k. That is, one measure of sensitivity is

$$S = \left| \left(\frac{\Delta x}{x} \right) \Big/ \left(\frac{\Delta k}{k} \right) \right|.$$

Simplifying a bit gives

$$S = \left| \left(\frac{\Delta x}{\Delta k} \right) \cdot \left(\frac{k}{x(k)} \right) \right|$$

where we are now explicitly stating that the output x is a function of k: $x = x(k)$. Taking $\Delta k = \delta$ as well as taking $\Delta x = x(x \pm \delta) - x(k)$ we can rewrite one more

time to give

$$S = \left| \left(\frac{x(k \pm \delta) - x(k)}{\delta} \right) \cdot \left(\frac{k}{x(k)} \right) \right|.$$

Notice that we could take the change of x by increasing k by δ or by decreasing k by δ.

The value of δ is related to known or estimated information about how the parameter varies. It is likely that k is a value from some statistical distribution (like a normal distribution) with an approximately known or estimated standard deviation. The value of δ should be related to the standard deviation and some basic statistics can be used to choose the δ for your sensitivity analysis. Recall that if you sample a parameter from the normal distribution then

- roughly 68% of the sampled parameters will be within 1 standard deviation of the mean of the normal distribution, and
- roughly 95% of the sampled parameters will be within 1.96 standard deviations of the mean of the normal distribution.

This means that if k comes from a normal distribution then a very typical choice for δ is 1.96 times the value of the estimated standard deviation (up or down from k). If, on the other hand, the values of the parameter are uniformly distributed then δ can be chosen as the maximum estimated deviation from the mean of the distribution (up or down from k).

Generally:

- If the value of S is approximately 1 then the relative changes are approximately the same and the output is not very sensitive to changes in the parameter.
- If the value of S is less than 1 then the relative changes in the output are less than the changes in the parameter and the output is not sensitive to changes in the parameter.
- Finally, if the value of S is larger than 1 then the relative changes in the output are greater than the changes in the parameter and the output is considered sensitive to changes in the parameter.

B.5.1 Example of Sensitivity Analysis

Let's do a more specific example. If we are analyzing the differential equation $P' = kP$ and we estimate that the growth rate is normally distributed with sample mean $k \approx 0.009$ and a standard deviation of $\sigma \approx 0.001$, then we can estimate the sensitivity of the time needed for the population to double as a function of the growth rate. In this case, the *doubling time* is the output and the *growth rate* is the parameter of interest.

The analytic solution to the differential equation is $P(t) = P_0 e^{kt}$ and the population doubling can be found by solving $2P_0 = P_0 e^{kT_d}$ where T_d is the time to double the population. Using some basic algebra we see that the doubling time as a function of the growth rate is $T_d(k) = \frac{\ln(2)}{k}$. Therefore, to measure the sensitivity of the doubling time to changes in the parameter k we can take $\delta = 1.96 \times 0.001 = 0.00196$. To measure sensitivity in doubling time to an increase in the growth rate we see that S is given as follows:

$$\begin{aligned}
S &= \left| \left(\frac{T_d(k+\delta) - T_d(k)}{\delta} \right) \cdot \left(\frac{k}{T_d(k)} \right) \right| \\
&= \left| \left(\frac{\frac{\ln(2)}{k+\delta} - \frac{\ln(2)}{k}}{\delta} \right) \cdot \left(\frac{k}{\frac{\ln(2)}{k}} \right) \right| \\
&= \left| \left(\frac{k\ln(2) - (k+\delta)\ln(2)}{\delta k(k+\delta)} \right) \cdot \left(\frac{k^2}{\ln(2)} \right) \right| \\
&= \left| \frac{k}{k+\delta} \right| \\
&= \frac{0.009}{0.009 + 1.96 \times 0.001} = \frac{0.009}{0.01096} \approx 0.8212
\end{aligned}$$

To measure sensitivity in doubling time to a decrease in the growth rate we see that S is given as[1]

$$\begin{aligned}
S &= \left| \left(\frac{T_d(k-\delta) - T_d(k)}{\delta} \right) \cdot \left(\frac{k}{T_d(k)} \right) \right| \\
&= \left| \left(\frac{\frac{\ln(2)}{k-\delta} - \frac{\ln(2)}{k}}{\delta} \right) \cdot \left(\frac{k}{\frac{\ln(2)}{k}} \right) \right| \\
&= \left| \left(\frac{k\ln(2) - (k-\delta)\ln(2)}{\delta k(k-\delta)} \right) \cdot \left(\frac{k^2}{\ln(2)} \right) \right| \\
&= \left| \frac{k}{k-\delta} \right| \\
&\approx \frac{0.009}{0.009 - 1.96 \times 0.001} = \frac{0.009}{0.00704} \approx 1.2784.
\end{aligned}$$

In the present example, the doubling time is not considered to be very sensitive to increases in the growth rate but it is sensitive to decreases in the growth rate. Given that in this simple example we are dealing with an exponential decay function this should also be intuitively *obvious*.

[1] I'm showing you the algebra here, but it really isn't necessary to show this level of routine algebra in your papers. Only show the algebra and other calculations that are necessary for the reader to understand the work that you're doing.

Appendix C

Optional Material

This Appendix contains a few sections that are considered optional for the course as we teach it. Instructors may be interested in expanding upon what is here for their classes.

C.1 Interpolation

The least squares problem that we studied in Chapter 4 seeks to find a best fitting function that is *closest* (in the Euclidean distance sense) to a set of data. What if, instead, we want to match the data points with a function. This is the realm of interpolation. Take note that there are many many forms of interpolation that are tailored to specific problems. In this brief section we cover only a few of the simplest forms of interpolation involving only polynomial functions. The problem that we'll focus on can be phrased as:
*Given a set of $n+1$ data points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, find a polynomial of degree at most n that **exactly** matches these points.*

C.1.1 Vandermonde Interpolation

Exercise C.1. Consider the data set

$$S = \{(0,1)\,,\,(1,2)\,,\,(2,5)\,,\,(3,10)\}.$$

If we want to fit a polynomial to this data then we can use a cubic function (which has 4 parameters) to match the data perfectly. Why is a cubic polynomial the best choice?

Exercise C.2. Using the data from the previous problem, if we choose $p(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$ then the resulting system of equations is

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 5 \\ 10 \end{pmatrix}.$$

a. Notice that the system of equations is square (same number of equations and unknowns). Why is this important?
b. Solve the system for $\beta_1, \beta_2, \beta_3$ and β_4 using any method discussed in Chapter 4.
c. Write the final polynomial $p(x)$ and verify that it matches the data points exactly.

d. Make a plot showing the data and your interoplated polynomial.

Definition C.1 (Vandemonde Interpolation)**.** Let $S = \{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$ be a set of ordered pairs where the x values are all unique. The goal of interpolation is to find a function $f(x)$ that matches the data exactly. Vandermonde interpolation uses a polynomial of degree $n-1$ since with such a polynomial we have n unknowns and we can solve the least squares problem exactly. Doing so, we arrive at the system of equations

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}.$$

The matrix on the left-hand side of is called the **Vandermonde Matrix**.

Exercise C.3. Write a python function that accepts an array of ordered pairs (where each x value is unique) and builds a Vandermonde interpolation polynomial. Test your function on the simple example given above and then on several larger problems. It may be simplest to initially test on data generated from functions that we know.

Exercise C.4. Build a Vandermonde interpolation polynomial to interpolate the function $f(x) = \cos(2\pi x)$ with n points that are linearly spaced on the interval $x \in [0, 2]$. Repeat this experiment with $n = 5$, $n = 10$, $n = 15$, …, $n = 100$. Make a plot for each value of n. What do you observe?

Exercise C.5. Vandermonde interpolation is relatively easy to conceptualize and code, but there is an inherent problem. Use your Vandemonde interploation

code to create a plot where the horizontal axis is the order of the interpolating polynomial and the vertical axis is the ratio of the maximum eigenvalue to the minimum eigenvalue of the Vandemonde matrix $|\lambda_{max}|/|\lambda_{min}|$. What does this plot tell you about Vandermonde interpolation for high-order polynomials? You can use the same model function as from the previous exercise.

C.1.2 Lagrange Interpolation

Lagrange interpolation is a rather clever interpolation scheme where we build up the interpolating polynomial from simpler polynomials. For interpolation we want to build a polynomial $p(x)$ such that $p(x_j) = y_j$ for every datapoint in the set $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$. If we can find a polynomial $\phi_j(x)$ such that

$$\phi_j(x) = \begin{cases} 0, & \text{if } x = x_i \text{ and } i \neq j \\ 1, & \text{if } x = x_j \end{cases}$$

then for Lagrange interpolation we build $p(x)$ as a linear combination of the ϕ_j functions. Let's look at an example.

Exercise C.6. Consider the data set $S = \{(0,1)\,,\,(1,2)\,,\,(2,5)\,,\,(3,10)\}$.

a. Based on the descriptions of the $p(x)$ and $\phi_j(x)$ functions, why would $p(x)$ be defined as

$$p(x) = 1\phi_0(x) + 2\phi_1(x) + 5\phi_2(x) + 10\phi_3(x)?$$

b. Verify that $\phi_0(x)$ can be defined as

$$\phi_0(x) = \frac{(x-1)(x-2)(x-3)}{(0-1)(0-2)(0-3)}.$$

That is to say, verify that $\phi_0(0) = 1$ and $\phi_0(1) = \phi_0(2) = \phi_0(3) = 0$.

c. Verify that $\phi_1(x)$ can be defined as

$$\phi_1(x) = \frac{(x-0)(x-2)(x-3)}{(1-0)(1-2)(1-3)}.$$

That is to say, verify that $\phi_1(1) = 1$ and $\phi_1(0) = \phi_1(2) = \phi_1(3) = 0$.

d. Define $\phi_2(x)$ and $\phi_3(x)$ in a similar way.

e. Build the linear combination from part (a) and create a plot showing that this polynomial indeed interpolates the points in the set S.

Exercise C.7. Is the Lagrange interpolation polynomial built form the previous problem the same as the Vandermonde interpolation polynomial for the same data?

Definition C.2 (Lagrange Interpolation)**.** To build an **Lagrange polynomial** $p(x)$ for the set of points $\{(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ we first build the polynomials $\phi_j(x)$ for each $j = 0, 1, 2, \ldots, n$ and then construct the polynomial $p(x)$ as

$$p(x) = \sum_{j=0}^{n} y_j \phi_j(x).$$

The $\phi_j(x)$ functions are defined as

$$\phi_j(x) = \prod_{i \neq j} \frac{x - x_i}{x_j - x_i}.$$

Example C.1. Build a Lagrange interpolation polynomial for the set of points

$$S = \{(1, 5), (2, 9), (3, 11)\}.$$

Solution: We first build the three ϕ_j functions.

$$\phi_0(x) = \frac{(x-2)(x-3)}{(1-2)(1-3)}$$
$$\phi_1(x) = \frac{(x-1)(x-3)}{(2-1)(2-3)}$$
$$\phi_2(x) = \frac{(x-1)(x-2)}{(3-1)(3-2)}.$$

Take careful note that the ϕ functions are built in a very particular way. Indeed, $\phi_0(1) = 1$, $\phi_0(2) = 0$, and $\phi_0(3) = 0$. Also, $\phi_1(1) = 0$, $\phi_1(2) = 1$), and $\phi_1(3) = 0$. Finally, note that $\phi_2(1) = 0$, $\phi_2(1) = 0$ and $\phi_2(3) = 1$. Thus, the polynomial $p(x)$ can be built as

$$p(x) = 5\phi_0(x) + 9\phi_1(x) + 11\phi(2(x) = 5\frac{(x-2)(x-3)}{(1-2)(1-3)} + \frac{(x-1)(x-3)}{(2-1)(2-3)} + \frac{(x-1)(x-2)}{(3-1)(3-2)}.$$

The remainder of the simplification is left to the reader.

Exercise C.8. Write a python function that accepts a list of list of ordered pairs (where each x value is unique) and builds a Lagrange interpolation polynomial. Test your function on several examples.

C.1.3 Chebyshev Points

Exercise C.9. Using either Vandermonde or Lagrange interpolation build a polynomial that interpolates the function

$$f(x) = \frac{1}{1+x^2}$$

for $x \in [-5, 5]$ with polynomials of order $n = 2, 3, \ldots$ and linearly spaced interpolation points. What do you notice about the quality of the interpolating polynomial near the endpoints?

Exercise C.10. As you should have noticed the quality of the interpolation gets rather terrible near the endpoints when you use linearly spaced points for the interpolation. A fix to this was first proposed by the Russian mathematician Pafnuty Chebyshev (1821-1894). The idea is as follows:

- Draw a semicircle above the closed interval on which you are interpolating.
- Pick n equally spaced points along the semicircle (i.e. same arc length between each point).
- Project the points on the semicircle down to the interval. Use these projected points for the interpolation.

a. Draw a picture of what we just described.
b. What do you notice about the x-values of these projected points? Why might it be desirable to use a collection of points like this for interpolation?

Definition C.3 (Chebyshev Interpolation Points)**.** It should be clear that since we are projecting down to the x-axis from a circle then all we need are the cosine values from the circle. Hence we can form the Chebyshev interpolation points for the interval $x \in [-1, 1]$ from the formula

$$x_j = \cos\left(\frac{\pi j}{n}\right), \quad \text{for} \quad j = 0, 1, \ldots, n.$$

To transform the Chebyshev points from the interval $[-1, 1]$ to the interval $[a, b]$ we can apply a linear transformation which maps -1 to a and 1 to b:

$$x_j \leftarrow \left(\frac{b-a}{2}\right)(x_j + 1) + a$$

where the "x_j" on the left is on the interval $[a, b]$ and the "x_j" on the right is on the interval $[-1, 1]$.

Exercise C.11. Consider the function $f(x) = \frac{1}{1+x^2}$ just as we did for the first problem in this subsection. Write code that overlays an interpolation with linearly spaced points and interpolation with Chebyshev points. Give plots for polynomial of order $n = 2, 3, 4, \ldots$. Be sure to show the original function on your plots as well. What do you notice?

Exercise C.12. Demonstrate that the Chebyshev interpolation nodes will improve the stability of the Vandermonde matrix over using linearly spaced nodes.

C.2 Multi-Dimensional Newton's Method

Now that we know some linear algebra let's return to the Newton's Method root finding technique from earlier in the book. This time we will consider root finding problems where we are not just solving the equation $f(x) = 0$ as we did Chapter 2. Instead consider the function F that takes a vector of variables in and outputs a vector. An example of such a function is

$$F(x, y) = \begin{pmatrix} x \sin(y) \\ \cos(x) + \sin(y^2) \end{pmatrix}.$$

It should be clear that making a picture of this type of function is a frivolous endeavor! In the case of the previous example, there are two inputs and two outputs so the "picture" would have to be four dimensional. Even so, we can still ask the question:
For what values of x and y does the function F give the zero vector?

That is, what if we have F defined as

$$F(x, y) = \begin{pmatrix} f(x, y) \\ g(x, y) \end{pmatrix}$$

and want to solve the system of equations

$$\begin{aligned} f(x, y) &= 0 \\ g(x, y) &= 0. \end{aligned}$$

In the present problem this amounts to solving the nonlinear system of equations

$$\begin{aligned} x \sin(y) &= 0 \\ \cos(x) + \sin(y^2) &= 0. \end{aligned}$$

In this case it should be clear that we are implicitly defining $f(x, y) = x \sin(y)$ and $g(x, y) = \cos(x) + \sin(y^2)$. A moment's reflection (or perhaps some deep

meditation) should reveal that $(\pm\pi/2, 0)$ are two solutions to the system, and given the trig functions it stands to reason that $(\pi/2 + \pi k, \pi j)$ will be a solution for all integer values of k and j.

Exercise C.13. To build a numerical solver for a nonlinear system of equations, let's just recall Newton's Method in one dimension and then mimic that for systems of higher dimensions. We'll stick to two dimensions in this problem for relative simplicity.

a. In Newton's Method we first found the derivative of our function. In a nonlinear system such as this one, talking about "the" derivative is a bit nonsense since there are many first derivatives. Instead we will define the Jacobian matrix $J(x, y)$ as a matrix of the first partial derivatives of the functions f and g.

$$J(x, y) = \begin{pmatrix} f_x & f_y \\ g_x & g_y \end{pmatrix}.$$

In the present example (fill in the rest of the blanks),

$$J(x, y) = \begin{pmatrix} \sin(y) & \_\_\_\_\_ \\ \_\_\_\_\_ & \_\_\_\_\_ \end{pmatrix}.$$

b. Now let's do some Calculus and algebra. Your job in this part of this problem is to follow all of the algebraic steps.

i. In one-dimensional Newton's Method we then write the equation of a tangent line at a point $(x_0, f(x_0))$ as

$$f(x) - f(x_0) \approx f'(x_0)(x - x_0)$$

to give a local approximation to the function. We'll do the exact same thing here, but in place of "x" we need to have a vector and in place of the derivative we need to have the Jacobian

$$F(x, y) - F(x_0, y_0) \approx J(x_0, y_0) \left(\begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \right).$$

ii. In one-dimensional Newton's Method we then set $f(x)$ to zero since we were ultimately trying to solve the equation $f(x) = 0$. Hence we got the equation

$$0 - f(x_0) \approx f'(x_0)(x - x_0)$$

and then rearranged to solve for x. This gave us

$$x \approx x_0 - \frac{f(x_0)}{f'(x_0)}.$$

In the multi-dimensional case we have the same goal. If we set $F(x, y)$ to the zero vector and solve for the vector $\begin{pmatrix} x \\ y \end{pmatrix}$ then we get

$$\begin{pmatrix} x \\ y \end{pmatrix} \approx \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} - [J(x_0, y_0)]^{-1} F(x_0, y_0).$$

Take very careful note here that we didn't divide by the Jacobian. Why not?

iii. The final step in one-dimensional Newton's Method was to turn the approximation of x into an iterative process by replacing x with x_{n+1} and replacing x_0 with x_n resulting in the iterative form of Newton's Method

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

We can do the exact same thing in the two-dimensional version of Newton's Method to arrive at

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} x_n \\ y_n \end{pmatrix} - J^{-1}(x_n, y_n) F(x_n, y_n).$$

Writing this in full matrix-vector form we get

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} x_n \\ y_n \end{pmatrix} - \begin{pmatrix} f_x & f_y \\ g_x & g_y \end{pmatrix}^{-1} \begin{pmatrix} f(x_n, y_n) \\ g(x_n, y_n) \end{pmatrix}.$$

c. Write down the Newton iteration formula for the system

$$\begin{aligned} x \sin(y) &= 0 \\ \cos(x) + \sin(y^2) &= 0. \end{aligned}$$

Do not actually compute the matrix inverse of the Jacobian.

d. The inverse of the Jacobian needs to be dealt with carefully. We typically don't calculate inverses directly in numerical analysis, but since we have some other tools to do the work we can think of it as follows:

- We need the vector $\boldsymbol{b} = J^{-1}(x_n, y_n) F(x_n, y_n)$.
- The vector $\boldsymbol{b}$ is the same as the solution to the equation $J(x_n, y_n)\boldsymbol{b} = F(x_n, y_n)$ at each iteration of Newton's Method.
- Therefore we can so a relatively fast linear solve (using any technique from Chapter 4) to find $\boldsymbol{b}$.
- The Newton iteration becomes

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} x_n \\ y_n \end{pmatrix} - \boldsymbol{b}.$$

Exercise C.14. Write code to solve the present nonlinear system of equations. Implement some sort of linear solver within your code and be able to defend your technique. Try to pick a starting point so that you find the solution $(\pi/2, \pi)$ on your first attempt at solving this problem. Then play with the starting point to verify that you can get the other solutions.

Exercise C.15. Test your code from the previous problem on the system of nonlinear equations

$$\begin{aligned} 1 + x^2 - y^2 + e^x \cos(y) &= 0 \\ 2xy + e^x \sin(y) &= 0. \end{aligned}$$

Note here that $f(x, y) = 1 + x^2 - y^2 + e^x \cos(y)$ and $g(x, y) = 2xy + e^x \sin(y)$.

Let's generalize the process a bit so we can numerically approximate solutions to systems of nonlinear algebraic equations in any number of dimensions. The Newton's method that we derived in Chapter 2 is only applicable to functions $f : \mathbb{R} \to \mathbb{R}$ (functions mapping a real number to a real number). In the previous problem we build a method for solving the equation $F(x, y) = (0, 0)$ where $F : \mathbb{R}^2 \to \mathbb{R}^2$. What about vector-valued functions in n dimensions? In particular, we would like to have an analogous method for finding roots of a function F where $F : \mathbb{R}^k \to \mathbb{R}^k$ for any dimension k.

Let $\boldsymbol{x}$ be a vector in $\mathbb{R}^k$, let

$$F(\boldsymbol{x}) = \begin{pmatrix} f_1(\boldsymbol{x}) \\ f_2(\boldsymbol{x}) \\ \vdots \\ f_k(\boldsymbol{x}) \end{pmatrix}$$

be a vector valued function, and let J be the Jacobian matrix

$$J(\boldsymbol{x}) = \begin{pmatrix} \partial f_1/\partial x_1(\boldsymbol{x}) & \partial f_1/\partial x_2(\boldsymbol{x}) & \cdots & \partial f_1/\partial x_k(\boldsymbol{x}) \\ \partial f_2/\partial x_1(\boldsymbol{x}) & \partial f_2/\partial x_2(\boldsymbol{x}) & \cdots & \partial f_2/\partial x_k(\boldsymbol{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \partial f_k/\partial x_1(\boldsymbol{x}) & \partial f_k/\partial x_2(\boldsymbol{x}) & \cdots & \partial f_k/\partial x_k(\boldsymbol{x}) \end{pmatrix}$$

By analogy, the multi-dimensional Newton's method is

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - J^{-1}(\boldsymbol{x}_n) F(\boldsymbol{x}_n)$$

where $J^{-1}(\boldsymbol{x}_n)$ is the inverse of the Jacobian matrix evaluated at the point $\boldsymbol{x}_n$. Take note that you should not be calculating the inverse directly, but instead you should be using a linear solve to get the vector $\boldsymbol{b}$ where $J(\boldsymbol{x}_n)\boldsymbol{b} = F(\boldsymbol{x}_n)$.

Exercise C.16. Write code that accepts any number of functions and an initial vector guess and returns an approximation to the root for the problem $F(\boldsymbol{x}) = \mathbf{0}$.

Exercise C.17. Use Newton's method to find an approximate solution to the system of equations

$$\begin{aligned} x^2 + y^2 + z^2 &= 100 \\ xyz &= 1 \\ x - y - \sin(z) &= 0 \end{aligned}$$

Exercise C.18. When will the multi-dimensional version of Newton's Method fail? Compare and contrast this with what you found about the one-dimensional version of Newton's Method in Chapter 2. Extend your discussion to talk about the eigenvalues of the Jacobian matrix for a nonlinear system.

Exercise C.19. One place that solving nonlinear systems arises naturally is when we need to find equilibrium points for systems of differential equations. Remember that to find the equilibrium points for a first order differential equation we set the derivative term to zero and solve the resulting equation.

Find the equilibrium point(s) for the system of differential equations

$$\begin{aligned} x' &= \alpha x - \beta xy \\ y' &= \delta y + \gamma xy \end{aligned}$$

where $\alpha = 1, \beta = 0.05, \gamma = 0.01$ and $\delta = 1$.

Exercise C.20. Find the equilibrium point(s) for the system of differential equations

$$\begin{aligned} x' &= -0.1xy - x \\ y' &= -x + 0.9y \\ z' &= \cos(y) - xz \end{aligned}$$

if they exist.

Exercise C.21. (This problem is modified from [6])
A manufacturer of lawn furniture makes two types of lawn chairs, one with a wood frame and one with a tubular aluminum frame. The wood-frame model costs \$18 per unit to manufacture, and the aluminum-frame model costs \$10 per unit. The company operates in a market where the number of units that can be sold depends on the price. It is estimated that in order to sell x units per day of the wood-frame model and y units per day of the aluminum-frame model, the selling price cannot exceed

$$10 + \frac{31}{\sqrt{x}} + \frac{1.3}{y^{0.2}} \text{ dollars per unit}$$

for wood-frame chairs, and

$$5 + \frac{15}{y^{0.4}} + \frac{0.8}{x^{0.08}} \text{ dollars per unit}$$

for the aluminum chairs. We want to find the optimal production levels. Write this situation as a multi-variable mathematical model, use a computer algebra system (or by-hand computation) to find the gradient vector, and then use the multi-variable Newton's method to find the critical points. Classify the critical points as either local maximums or local minimums.

---

[1] Y. Xie, "Bookdown: Authoring books and technical documents with r markdown." https://bookdown.org/yihui/bookdown/, 2019.

[2] M. Boelkins, "Active calculus." https://activecalculus.org/single/frontmatter.html, 2018.

[3] "ProjectEuler.net." https://projecteuler.net/.

[4] A. Greenbaum and T. Chartier, *Numerical methods: Design, analysis, and computer implementation of algorithms*. Princeton University Press, 2012.

[5] R. Burden, D. Faires, and A. Burden, *Numerical analysis, 10ed*. Cengage Learning, 2016.

[6] M. Meerschaert, *Mathematical modeling, 4ed*. Academic Press, 2013.

[7] E. Sullivan, J. Bauer, and E. Wiens, "1-094-s-steepingtea." Sep. 2017, [Online]. Available: https://www.simiode.org/resources/4190.

[8] B. Winkel, "6-004-s-villageepidemic." Jul. 2016, [Online]. Available: https://simiode.org/resources/2372.

[9] S. Miller, "6-001-s-epidemic." Jun. 2015, [Online]. Available: https://simiode.org/resources/572.

[10] R. Spindler, "6-023-s-droneheadinghome." Apr. 2017, [Online]. Available: https://www.simiode.org/resources/3476.

[11] K. Spayd and J. Puckett, "9-020-t-heatdiffusion." Oct. 2019, [Online]. Available: https://simiode.org/resources/6452.