Numerical Methods:
An Inquiry-Based Approach
with Python

Dr. Eric Sullivan
Department of Mathematics
Carroll College, Helena, MT
esullivan@carroll.edu

Content Last Updated: November 29, 2019

Contents

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 and I cite several wonderful traditional texts in the bibliography at the end of this book.

0.1 To The Student

0.1.1 The Inquiry-Based Approach

Problem 0.1 (Setting The Stage). 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.

- Get in groups of size 3-4.

- Group members should introduce themselves.

- For each of the questions that follow I will ask you to:

    1. **Think** about a possible answer on your own

    2. **Discuss** your answers with the rest of the group

    3. **Share** a summary of each group's discussion

Questions:

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

Question #2: How does a person learn something new?

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

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

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

▲

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 and then only use other more traditional texts (or Google) as a backup when you are completely stuck. Let me say that again: this is not the only set of material for the course. Your brain, your peers, and the books linked in the next section are your best resources when you are stuck.

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. Learning this material is most definitely a non-linear path.

3. **Collaborate!** You should collaborate with your peers with the following caveats:

   (a) 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.

   (b) The internet is not a collaborator. 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!

0.1.2   Online Texts and Other Resources

If you are looking for online textbooks for numerical methods or numerical analysis I can point you to a few of my favorites. Some of the following online resources may be a good place to help you when you're stuck but they will definitely say things a bit differently. Use these resources wisely.

- Holistic Numerical Methods http://nm.mathforcollege.com/
  The Holistic Numerical Methods book is probably the most complete free reference that I've found on the web. This should be your source to look up deeper explanations of problems, algorithms, and code.

- Scientific Computing with MATLAB http://gribblelab.org/scicomp/scicomp.pdf

- Tea Time Numerical Analysis http://lqbrin.github.io/tea-time-numerical/

0.2   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.

0.2.1   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. While the math majors need to eventually see the rigorous derivations the other majors, I think, are better served working from an experimental approach rather than a theorem-proof approach.

0.2.2 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. Appendix C gives students tips for writing in LaTeX if indeed you want your students using that tool.

0.2.3 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 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. 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.

0.2.4 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

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.

0.3   Special Thanks

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. Next I would like to thank the institution of Carroll College for seeing this project as a worthy academic pursuit even though the end result is not a book or publication in the traditional sense. Finally, I would like to thank all of my colleagues and students, both past and present, in the math department at Carroll. The suggestions, questions, struggles, and triumphs of these folks are what have shaped this work into something that I'm proud of and that I hope will be a useful resource for future students and instuctors.

Chapter 1

Introductory Topics

1.1 An Introduction to 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."

–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. There are some problems where numerical analysis doesn't make much sense (e.g. finding an algebraic derivative of a function) but for many problems a numerical method that gives an approximate answer is both more efficient and more versatile than any analytic technique. For example, if we needed to solve the differential equation $\frac{dy}{dt} = \sin(y^2) + t$ the nonlinear nature of the problem makes it hard to work with analytically but 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. Similarly, efficient algorithms to do the basic operations of linear algebra (e.g. Gaussian elimination, matrix factorization, or eigenvalue decomposition) are highly sought after and useful when the matrices used for a model are very large.

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 Python 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.

<div align="center">Let's begin.</div>

1.2   Base 2 and Binary Arithmetic

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

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

▲

Problem 1.2. Now use Excel and Python to do the computations. Do you get the same answers? ▲

Problem 1.3. 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? ▲

Problem 1.4. 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

$$48,329 = 40,000 + 8,000 + 300 + 20 + 9 = 4 \times 10^4 + 8 \times 10^3 + 3 \times 10^2 + 2 \times 10^1 + 9 \times 10^0,$$

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 based 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.

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

(a) 111_2

(b) $10,101_2$

(c) $1,111,111,111_2$

▲

Problem 1.6. Explain the joke:

There are 10 types of people. Those who understand binary and those who don't.

▲

Problem 1.7. 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} who's base-2 expression is $11,101,101_2$ ▲

Problem 1.8. 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?

▲

Problem 1.9. Now that you have converted several base-10 numbers to base-2, summarize an efficient technique to do the conversion.   ▲

Example 1.10. 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.

Problem 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.

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

▲

Problem 1.12. Repeating digits in binary numbers are rather intriguing. The number $0.\overline{0111} = 0.011101110111011101 11\ldots$ surely also has a decimal representation. I'll get you started:

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

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

Example 1.13. Convert 11.01011_2 to base 10.

Solution:

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

Problem 1.14. Convert the following numbers from base 10 to binary.

(a) What is 1/2 in binary?

(b) What is 1/8 in binary?

(c) What is 4.125 in binary?

(d) What is 0.15625 in binary?

▲

Problem 1.15. Convert the base 10 decimal 0.635 to binary using the following steps. Explain why each step gives the binary digit that it does.

(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*.

▲

Problem 1.16. Based on your previous problem write an algorithm that will convert base-10 decimals (less than 1) to base decimal expansions. ▲

Problem 1.17. Convert the base 10 fraction 1/10 into binary. Use your solution to fully describe what went wrong in problems 1.1 - 1.3? ▲

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.

Problem 1.18. Consider the number $x = -129.15625$ (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

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

(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 $0 \le m < 1$.

Definition 1.19. 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 1.20. 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 1.21 (Computer Precision Standards)**.** 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 1.22. 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.

Problem 1.23. To make all of these ideas concrete let's play with a small computer system where each number is stored in the following format:

$$\boxed{s} \boxed{E} \boxed{b_1 b_2 b_3}$$

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

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

- What is the smallest positive number that can be represented in this form?

- What is the largest positive number that can be represented in this form?

- What is the machine precision in this number system?

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

▲

Problem 1.24. What are the largest and smallest numbers that can be stored in single and double precicision?   ▲

Problem 1.25. What is machine epsilon for single and and double precision?   ▲

Problem 1.26. A typical computer number:

| 0 | E=4 | 10011001100000000000000 |

What is this number? Is it stored in single or double precision?   ▲

Problem 1.27. 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.

1.4 Polynomial Approximation and The Taylor Series

Now we turn our attention in this introductory chapter to something more mathematical. The Taylor Series is a mathematical tool that is widely used in approximation theory, and since a computer can only make approximations we need ways to approximate functions so that a computer can understand them. The Taylor Series will be just the right tool for our purposes.

Keep the following basic idea in your mind as you work through this section:

> *The Taylor Series is a tool for expressing complicated functions as infinite polynomials. Polynomials are great for computers since they only involve addition, subtraction, multiplication, and powers – all things that computers are good at.*

Consider the function $f(x) = e^x$. Euler's number

$$e = 2.718281828459045\cdots$$

is irrational and impossible for a computer to represent directly. How, do you suppose, does a computer actually *understand* a function like e^x (or any other transcendental function for that matter)?

Answer: Polynomials!

Polynomials are some of the simplest types of functions since they involve very basic mathematical operations: really just addition and multiplication (since subtraction and division are just *special* addition and multiplication).

Let's get a feel for how we approximate functions like $f(x) = e^x$ with a simple exercise. In the following exercise we will build a polynomial function with certain properties to *match* the exponential function.

Problem 1.28. Let $f(x) = e^x$.

(a) Find a linear function of the form $g(x) = a_0 + a_1 x$ such that $g(0) = f(0)$ and $g'(0) = f'(0)$. Plot both $f(x)$ and $g(x)$ on the same axes.

(b) Find a quadratic function of the form $g(x) = a_0 + a_1 x + a_2 x^2$ such that $g(0) = f(0)$, $g'(0) = f'(0)$, and $g''(0) = f''(0)$. Plot both $f(x)$ and $g(x)$ on the same axes.

(c) Find a polynomial of order n that matches the function $f(x) = e^x$ such that $g^{(k)}(0) = f^{(k)}(0)$ for all $k \le n$. Plot both $f(x)$ and $g(x)$ on the same axes for several values of n. What do you observe about the function $f(x)$ and the function $g(x)$?

▲

Problem 1.29. Repeat Problem 1.28 with the function $f(x) = \sin(x)$. ▲

Problem 1.30. Repeat Problem 1.28 with the function $f(x) = \cos(x)$. ▲

Problem 1.31. Now let's graphically explore the results that you got from Problems 1.28, 1.29, and 1.30. You should have found the following results (check your work from the previous problems):

- The exponential function:

$$f(x) = e^x \approx 1 + x \qquad \text{(First order approximation)}$$

$$f(x) = e^x \approx 1 + x + \frac{x^2}{2} \qquad \text{(Second order approximation)}$$

$$f(x) = e^x \approx 1 + x + \frac{x^2}{2} + \frac{x^3}{6} \qquad \text{(Third order approximation)}$$

$$f(x) = e^x \approx 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots + \frac{x^n}{n!} \qquad (n^{th} \text{ order approximation})$$

- The sine function

$$f(x) = \sin(x) \approx x \qquad \text{(First order approximation)}$$

$$f(x) = \sin(x) \approx x - \frac{x^3}{6} \qquad \text{(Third order approximation)}$$

$$f(x) = \sin(x) \approx x - \frac{x^3}{6} + \cdots + \frac{(-1)^n x^{2n+1}}{(2n+1)!} \qquad (n^{th} \text{ order approximation})$$

- The cosine function

$$f(x) = \cos(x) \approx 1 - \frac{x^2}{2} \qquad \text{(Second order approximation)}$$

$$f(x) = \cos(x) \approx 1 - \frac{x^2}{2} + \frac{x^4}{24} \qquad \text{(Fourth order approximation)}$$

$$f(x) = \cos(x) \approx 1 - \frac{x^2}{2} + \cdots + \frac{(-1)^n x^{2n}}{(2n)!} \qquad (n^{th} \text{ order approximation})$$

(a) Using any convenient graphing tool (e.g. Desmos, a graphing calculator, Python , etc) make a plot of the exponential function along with the first, second, and third order approximations. What do you notice and what do you wonder?

(b) Repeat part (a) with the sine function.

(c) Repeat part (a) with the cosine function.

(d) Graphically test the following conjecture:

> **Conjecture:** *If I add more and more terms to the exponential, sine, or cosine polynomial approximation then the resulting polynomial will match the actual function better and better over a larger domain.*

Based on the graphs that you made do you think that this conjecture is true or false?

▲

Problem 1.32. In Problem 1.31 you should have confirmed that the Taylor series for the exponential, sine, and cosine functions appear to get better and better on larger and larger domains as the degree of the polynomial increases. Now we'll test the conjecture in general:

> **Conjecture:** *The Taylor polynomial always be a better approximation over a larger domain when we add more terms?*

Below you will find several functions along with their polynomial approximations. Test the above conjecture graphically on these approximations.

(a) $f(x) = \dfrac{1}{1-x} \approx 1 + x + x^2 + x^3 + x^4 + \cdots x^n$.

(b) $f(x) = \ln(1+x) \approx x - \dfrac{x^2}{2} + \dfrac{x^3}{3} - \dfrac{x^4}{4} + \cdots + \dfrac{x^n}{n}$.

(c) $f(x) = \arctan(x) = x - \dfrac{x^3}{3} + \dfrac{x^5}{5} - \dfrac{x^7}{7} + \cdots + \dfrac{(-1)^n x^{2n+1}}{2n+1}$.

▲

Now we'll formally state the actual definition of a Taylor Series.

Definition 1.33 (Taylor Series)**.** If f is infinitely smooth (has infinitely many derivatives) then f can be expressed as an infinite sum of power functions

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

in some neighborhood of $x = a$.

It is sometimes easier to look at the definition of the Taylor series without the summation notation.

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

Example 1.34. Verify that the Taylor series for $f(x) = e^x$ is exactly what we found in Problems 1.28 and 1.31.
Solution: First note that $f(x) = e^x$ has the beautiful property that $f^{(n)}(x) = e^x$. That is, every derivative of the exponential function is just the exponential function again. If

we take $a = 0$ in the definition of the Taylor Series then we see that

$$e^x = f(0) + \frac{f'(0)}{1!}(x-0)^1 + \frac{f''(0)}{2!}(x-0)^2 + \frac{f'''(0)}{3!}(x-0)^3 + \frac{f^{(4)}(0)}{4!}(x-0)^4 + \cdots$$

$$= e^0 + e^0 x + \frac{e^0}{2}x^2 + \frac{e^0}{3!}x^3 + \frac{e^0}{4!}x^4 + \cdots$$

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

Problem 1.35. We have encountered 6 different Taylor series thus far:

$$e^x, \quad \sin(x), \quad \cos(x), \quad \frac{1}{1-x}, \quad \ln(1+x), \quad \text{and} \quad \arctan(x).$$

Using the definition of the Taylor series, verify each of the formulas that we have used. You can see a verification for e^x in Example 1.34.   ▲

One should keep in mind that the Taylor Series for a function may not make sense on the whole real line even if the function's domain is the entire real line. You should have taken note of this in problem 1.32 (if you didn't then go back to problem 1.32). The domain on which the Taylor series makes sense is called the **domain of convergence**. We can also talk about a Taylor series having a **radius of convergence** where the radius is the distance from the center (the "a" in Definition 1.33) to the first point where the Taylor Series does not converge.

To determine the radius of convergence for a Taylor Series we should recall the Ratio Test from Calculus

Theorem 1.36 (The Ratio Test from Calculus)**.** Let $\sum_{n=0}^{\infty} a_n$ be an infinite series. Suppose that

$$\lim_{n \to \infty} \frac{|a_{n+1}|}{|a_n|} = r.$$

(a) If $0 < r < 1$ then the infinite series converges.

(b) If $r > 1$ then the infinite series diverges.

(c) If $r = 1$ then the ratio test is inconclusive.

For the purposes of Taylor series we need a more robust statement of the Ratio Test. Notice that in Theorem 1.36 we are only considering an infinite series of numbers, not an infinite series of functions. The following corollary gives a more thorough statement of the Ratio Test as it applies to Taylor Series.

Corollary 1.37 (The Ratio Test for Taylor Series)**.** Let $f(x)$ be given as a Taylor Series

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

The series converges for all x such that

$$\lim_{n\to\infty}\left|\frac{f^{(n+1)}(a)(x-a)^{n+1}}{(n+1)!}\right|\Bigg/\left|\frac{f^{(n)}(a)(x-a)^k}{n!}\right| < 1.$$

Simplifying the fractions gives

$$\lim_{n\to\infty}\frac{|f^{(n+1)}(a)(x-a)|}{|f^{(n)}(a)(n+1)|} < 1.$$

The values of x that satisfy this limit are called the *domain of convergence* for the Taylor Series.

Example 1.38. Find the domain of convergence of the Taylor Series for the exponential function $f(x) = e^x$.
Solution:
For simplicity we will take the Taylor series to be centered at $a = 0$. We know that

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

so we need to determine the following limit.

$$\lim_{k\to\infty}\left|\frac{x^{k+1}}{(k+1)!}\frac{k!}{x^k}\right|.$$

Simplifying the fractions inside the absolute values gives

$$\lim_{k\to\infty}\left|\frac{x}{(k+1)}\right|$$

which for every $x \in \mathbb{R}$ we see that

$$\lim_{k\to\infty}\left|\frac{x}{(k+1)}\right| = 0 < 1.$$

Therefore, for every value of $x \in \mathbb{R}$ we know that the Taylor Series for e^x converges to the actual function. Hence the "=" sign that we used in the beginning of this solution is actually valid for every x. Be careful since this is not true for all functions.

Example 1.39. Find the Taylor series centered at $a = 0$ for the function $f(x) = \frac{1}{1-x}$ and determine the domain of convergence.
Solution:

Recall from the definition of the Taylor series that

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

Hence we need to find a pattern in the sequence of derivatives $f^{(n)}(0)$ first.

$$
\begin{aligned}
f(x) &= (1-x)^{-1} & \implies & \quad f(0) = 1 \\
f'(x) &= (1-x)^{-2} & \implies & \quad f'(0) = 1 \\
f''(x) &= 2(1-x)^{-3} & \implies & \quad f''(0) = 2 \\
f'''(x) &= 6(1-x)^{-4} & \implies & \quad f'''(0) = 6 = 3! \\
f^{(4)}(x) &= 24(1-x)^{-5} & \implies & \quad f^{(4)}(0) = 24 = 4! \\
f^{(5)}(x) &= 120(1-x)^{-6} & \implies & \quad f^{(5)}(0) = 120 = 5! \\
&\;\;\vdots & & \\
f^{(n)}(x) &= (-1)^n n! (1-x)^{-n-1} & \implies & \quad f^{(n)}(0) = n!
\end{aligned}
$$

Therefore the Taylor series for $f(x) = \frac{1}{1-x}$ is

$$\frac{1}{1-x} = \sum_{n=0}^{\infty} \frac{n!}{n!} x^n$$

which can clearly simplify to

$$\frac{1}{1-x} = \sum_{n=0}^{\infty} x^n = 1 + x + x^2 + x^3 + x^4 + \cdots$$

(a beautifully simple Taylor series for kind of a complicated function).

To find the domain of convergence we note that the absolute value of the ratio of successive terms in the series is
$$\left| \frac{x^{n+1}}{x^n} \right| = |x|.$$

Hence
$$\lim_{n \to 0} \left| \frac{x^{n+1}}{x^n} \right| = |x|$$

and we see from the ratio test that $|x| < 1$ is the domain of convergence. In other words, the Taylor series only makes sense for x values on the inverval $-1 < x < 1$.

A deeper look at this function reveals a deeper insight: The domain of convergence is the distance from the center of the series, $a = 0$, to the neareest vertical asymptote at $x = 1$.

Problem 1.40. Write Python code to show successive approximations of the function $f(x) = e^x$ on the domain $-1 < x < 1$ using a Taylor series centered at $a = 0$. Write your

code so that it animates through the approximations. Once your code is working, modify it to do the same for $f(x) = \sin(x)$ centered at $a = 0$ and for $f(x) = \cos(x)$ centered at $a = 0$.
▲

Problem 1.41. Consider the function $f(x) = \frac{1}{1+x^2}$.

(a) Build the Taylor Series for this function centered at $a = 0$.

(b) Using the previous example as a guide find the domain of convergence for the Taylor Series.

(c) Modify your Python code from the previous problem to demonstrate the fact that $f(x) = \frac{1}{1+x^2}$ does not have an infinite domain of convergence.

▲

1.5   Truncation Error with Taylor Series

The great thing about the Taylor Series is that it allows for the approximation of smooth functions as polynomials – and polynomials are easily dealt with on a computer since they involve only addition, subtraction, multiplication, and integer powers. The down side is that the sum 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.

Definition 1.42 (Truncation Error). If $f(x) = \sum_{n=0}^{\infty} c_n(x-a)^n$ is the Taylor Series expansion for $f(x)$ at the point $x = a$ then $f_N(x) = \sum_{n=0}^{N} c_n(x-a)^n$ is an approximation of $f(x)$ using only N terms of the Taylor Series where $N < \infty$. The function $f_N(x)$ is only the same as $f(x)$ if N goes to infinity. The **truncation error** at the point $x = x_0$ made when truncating the infinite series is the absolute difference between $f(x_0)$ and $f_N(x_0)$
$$T_{x_0} = |f(x_0) - f_N(x_0)|.$$

Problem 1.43. What is the absolute truncation error when calculating e^1 with the Taylor Series
$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \cdots$$
using the following values for n? Recall that $e^1 \approx 2.718281828459045$.

n	Value from the Taylor Series	Absolute Error
0	1	$|e^1 - 1| \approx 1.718281828459045\ldots$
1	1+1=2	$|e^1 - 2| \approx 0.718281828459045\ldots$
2	$1 + 1 + 1^2/2 = 2.5$	$|e^1 - 2.5| \approx 0.218281828459045\ldots$
3	$1 + 1 + 1^2/2 + 1^3/6 = 2.\bar{6}$	$|e^1 - 2.\bar{6}| \approx$
4		
5		
6		

▲

Problem 1.44. Based on your answers to the previous problem, how many terms do you need in the Taylor Series for e^x to approximation e to two decimal places? ▲

Remember that when using a computer we cannot ever use an infinite series. Thus, every time we use a Taylor Series approximation we will naturally be using a truncated version of the infinite series. This means that we need a good tool to measure the error that we make when doing so.

To set the stage, let's say that we truncate a Taylor series at the nth term. Therefore there is some remainder left over in the infinity of terms from the $(n+1)^{st}$ term on and we can write the series as

$$f(x) = P_n(x) + R_n(x)$$

where $P_n(x)$ is just the nth order polynomial coming from the 0th to the nth terms of the Taylor Series. The remainder is the subject of the next theorem.

Theorem 1.45 (Taylor's Theorem). Let $f, f', f'', \ldots, f^{(n)}$ be continuous *near a* and let $f^{(n+1)}(x)$ exist for all *x near* $x = a$. Then there is a number ξ between x and a such that

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + R_n(x) \tag{1.1}$$

where the remainder function $R_n(x)$ is given as

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-a)^{n+1}. \tag{1.2}$$

Often times we are using Taylor series that are centered at $a = 0$ so for simplicity we restate Taylor's theorem here with $a = 0$.

Corollary 1.46 (Taylor's Theorem at $a = 0$). Let $f, f', f'', \ldots, f^{(n)}$ be continuous *near a* and let $f^{(n+1)}(x)$ exist for all *x near* $x = 0$. Then there is a number ξ between x and 0 such that

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f'''(0)}{3!}x^3 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + R_n(x) \tag{1.3}$$

where the remainder function $R_n(x)$ is given as

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}x^{n+1} \tag{1.4}$$

Example 1.47. The first three terms of the Taylor series for $f(x) = e^x$ centered at $x = 0$ are

$$f(x) = e^x \approx 1 + x + \frac{x^2}{2}.$$

Use Taylor's theorem to approximate the error in this approximation when $x \approx 1$.
Solution: The remainder function gives us that there exists a number ξ such that $0 < \xi < 1$ and the remainder in the Taylor series is

$$R_3(x) = \frac{f^{(3)}(\xi)}{3!}(x - 0)^3 = \frac{e^\xi}{3!}x^3.$$

Therefore $R_3(x) \leq \frac{e^1}{3!} \cdot 1^3 = \frac{e}{6} \approx 0.45$. In Figure 1.1 we see that the error is indeed less than this. Indeed, $f(1) = 2.718281828459045\cdots$ and $g(1) = 2.5$ so the actual error is about $0.218 < 0.45$.

   If we extend this example a bit we can see that the absolute error and the approximated error (as found with Taylor's Theorem) converge to each other rather quickly.

n	Value from the Taylor Series	Absolute Error	Approximate Error
0	1	1.71828	2.71828
1	2	0.71828	1.35914
2	2.5	0.21828	0.45305
3	2.66666	0.05162	0.11326
4	2.70833	0.00995	0.02265
5	2.71806	0.00023	0.00054
6	2.71825	0.00003	0.00006

Example 1.48. The Taylor series for $g(x) = \frac{1}{1-x}$ is given in Example 1.39. What is the maximum error that can occur when approximating $f(c)$ for $c \in (-1, 1)$ with a fifth-order polynomial?
Solution: We know that $g(x) \approx \frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + x^5$ for $x \in (-1, 1)$ and from Taylor's remainder theorem we know that $R_5(x) = g^{(6)}(\xi)x^{5+1}/(5+1)!$. The sixth derivative of g is $g^{(6)}(x) = 6!(1-x)^{-7}$ and therefore the largest that $g^{(6)}\xi$ can be for $\xi \in (-1, 1)$ is $6!$. Therefore the remainder term simplifies to $R_5(x) = x^6$. Finally, the largest that $R_5(x)$ can be on $x \in (-1, 1)$ is 1 so the maximum error is 1.

   Long story short: Taylor's theorem gives a way to bound the amount of error that you can make when using a truncated Taylor series.

Problem 1.49. The *engineer's approximation* to the sine function is:

$$\text{For } x \text{ close to } 0, \sin(x) \approx x.$$
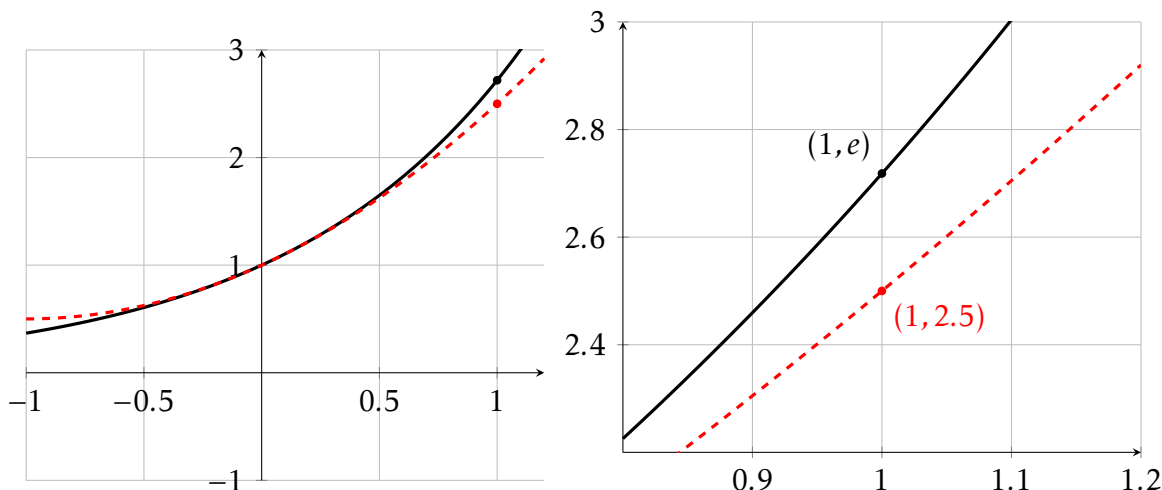
Figure 1.1. The function $f(x) = e^x$ and a second order Taylor approximation. The solid black curve is $f(x) = e^x$ and the dashed red curve is the Taylor approximation. The right-hand plot shows a zoomed in view near the point $x = 1$.

Obviously the word *close* is relative. Use Taylor's theorem to determine how much error is being made with the *engineer's approximation* if you want to calculate $\sin(0.5)$? See Figure 1.2. ▲
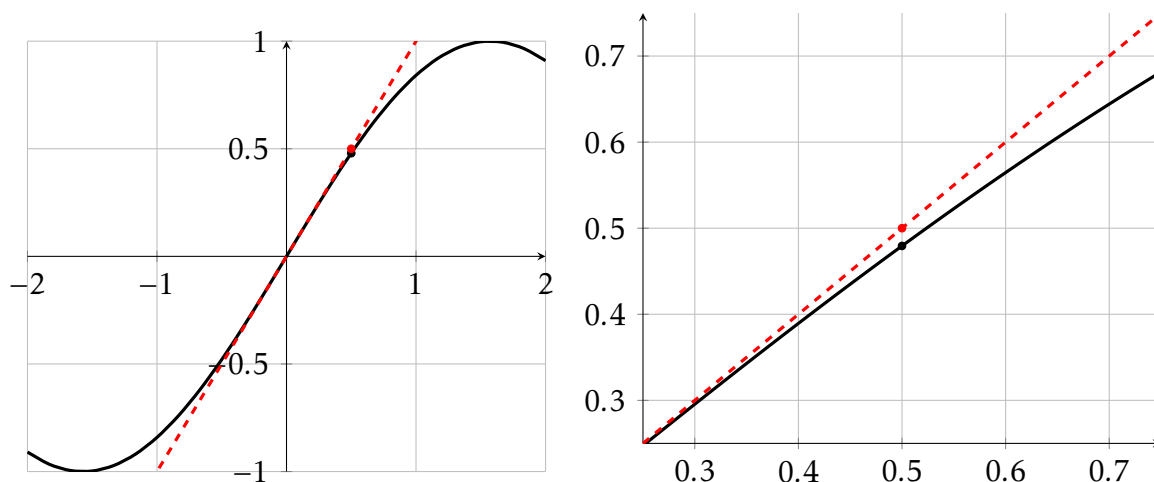
Figure 1.2. The function $f(x) = \sin(x)$ and a second order Taylor approximation. The solid black curve is $f(x) = \sin(x)$ and the dashed red curve is the Taylor approximation. The right-hand plot shows a zoomed in view near the point $x = 0.5$.

Problem 1.50. No computational software actually *knows* functions like the exponential function or the sine function. Instead, they have a way to calculate values for these functions based on Taylor series. If we want to calculate a value for $e^{0.5}$ on a computer,

how many terms in the Taylor series do we need so that the truncation error is less than machine precision?   ▲

Example 1.51. How many terms of the Taylor Series for e^x (centered at $a = 0$) are needed to calculate e to within 10 decimal places?
Solution:
Recall that if $f(x) = e^x$ then

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

and this Taylor Series representation converges for all $x \in \mathbb{R}$. From Taylor's Theorem we know that the remainder term takes the form

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} x^{n+1}$$

where $\xi \in (0, 1)$. For the exponential function we know that $f^{(n+1)}(\xi) = f(\xi) = e^\xi$. Furthermore, we are trying to approximate the error at $x = 1$ so the remainder term is

$$R_n(1) = \frac{e^\xi}{(n+1)!}.$$

We want to find n such that $R_n(1) < 10^{-10}$. Observe that since $\xi \in (0, 1)$ we know that $e^\xi < e$. Hence we need n such that $e \cdot 10^{10} < (n+1)!$. Examining the orders of magnitude for the factorial sequence we see that $n \geq 13$ since $(13 + 1)! \approx 9 \times 10^{10} > e \cdot 10^{10}$.

Indeed, if we use 13 terms in the Taylor Series for $f(x) = e^x$ we get

$$f(1) \approx 2.71828182844676,$$

and this approximation is correct to the 10th decimal digit.

Example 1.52. Calculate e to within machine precision.
Solution:
Modifying the previous example we see that we need n such that

$$R_n(1) = \frac{e^\xi}{(n+1)!} < 10^{-16}.$$

Examining the orders of magnitude of the factorial sequence we see that we need $n \geq 18$ terms in the Taylor Series to have a value of e that is accurate to within machine precision.

1.6 Exercises

1.6.1 Coding Exercises

The first several exercises here are meant for you to practice and improve your coding skills.

Problem 1.53. 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. Consult some of the examples in Appendix A if you're stuck with the coding.   ▲

Problem 1.54. 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:

$$1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

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. Consult some of the examples in Appendix A if you're stuck with the coding.   ▲

Problem 1.55. Write computer code that will draw random numbers from the unit interval [0,1], distributed uniformly, 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 draw until your sum is larger than 1?
Hint #1: In Python you should use the `numpy.random.rand(1,1)` 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!   ▲

Problem 1.56 (Jenny's Phone Number)**.** My favorite prime number is 8675309. Yep. Jenny's phone number is prime! Write a Python script that verifies this fact. Consult some of the examples in Appendix A if you're stuck with the coding.   ▲

Problem 1.57. Write a function called `MyPrimeChecker` that accepts an integer and returns a binary variable: 0 = not prime, 1 = prime. Next write a Python script to find the sum of all of the prime numbers less than 1000. Consult some of the examples in Appendix A if you're stuck with the coding.   ▲

Problem 1.58. 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 sum of the squares of the first ten natural numbers and the square of the sum is $3025 - 385 = 2640$.

Write code to find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum. Your code needs to run error free and output only the difference.   ▲

Problem 1.59. 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.   ▲

Problem 1.60. 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?   ▲

Problem 1.61. 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.

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

Problem 1.62. For each of the following commands tell what the command is asking Python to do and why the answer is *wrong*.

(a) `sqrt(2)^2 == 2`

(b) `(1/49)*49 == 1`

(c) `exp(log(3)) == 3`

                                                                      ▲

Problem 1.63. Use the ratio test to determine the radius of convergence for the functions $\sin(x)$, $\cos(x)$, $\ln(1 + x)$, and $\arctan(x)$. Show all of your work and support your answer with plots demonstrating your findings graphically.   ▲

Problem 1.64 (modified from [1]). 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 following two lines of Python code do.

```python
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?

▲

Problem 1.65 (modified from [1]). 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).

▲

Problem 1.66 (Approximating π). In this problem we will use Taylor Series to build approximations for the irrational number π.

(a) Write the Taylor series centered at $a = 0$ for the function $f(x) = \frac{1}{1+x}$. You may want to refer back to Examples 1.38 and 1.39 to get started.

(b) Use the ratio test to determine the domain of convergence for this Taylor series.

(b) Substitute t^2 for x to get a Taylor series for $g(t) = \frac{1}{1+t^2}$.

(c) Integrate both sides from $t = 0$ to $t = y$ to get a Taylor series for $h(y) = \arctan(y)$.

(d) Use the fact that $\arctan(1) = \pi/4$ along with your answer to part (c) to approximate π to 10 decimal digits of accuracy. Use Taylor's theorem to prove that you have the correct accuracy.

▲

Problem 1.67. 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 the Taylor expansion.

(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 $e^{i\pi} + 1 = 0$.

▲

Problem 1.68. Create a Python function that accepts an Lambda Function $f(x)$ and an integer N and gives as an output an animation of successive Taylor approximations of the function up to the Nth term.
Write a test script that calls your function on several different infinitely differentiable functions. Your test script will look something like the following.

```
1  import numpy as np
2  f = lambda x: np.sin(x)
3  N = 50;
4  TaylorAnimation(f,N)
```

▲

Problem 1.69. Find the domain of convergence for the Taylor series for each of the following Taylor series. Center each Taylor series at $a = 0$.

$$f_1(x) = \sin(x)$$
$$f_2(x) = \cos(x)$$
$$f_3(x) = \frac{1}{1+x}$$
$$f_4(x) = \frac{1}{1-x}$$
$$f_5(x) = \frac{1}{1+x^2}$$
$$f_6(x) = \ln(1+x)$$

▲

Chapter 2

Numerical Algebra

2.1 A Root Finding Game

Let's play a game!

Problem 2.1. Find a partner in the room to play a two-person game. Choose someone to go first and follow these steps to play.

Step #1: The first player secretly chooses a positive real number. For simplicity let's choose something less than 100.

Step #2: The first player chooses non-negative real numbers a and b so that the secret number is between a and b.

Step #3: The first player tells the second player "*my number is between a and b*".

Step #4: The second player makes a guess at the secret number. We'll call that number c.

Step #5: The first player responds with one of the following three responses.

- If c is within 0.01 of the secret number then say
  "*you win!*"
- If the secret number is between a and c then say
  "*the secret number is between a and c.*"
- If the secret number is between c and b then say
  "*the secret number is between c and b.*"

Step #6: Repeat steps 4 and 5 until the second player guesses the secret number to within 0.01 or you have repeated more than 20 times. The second player wins if he/she guesses the number in 20 or fewer steps.

▲

Problem 2.2. Discuss with your partner what the optimal strategy for the second player is for the game in Problem 2.1. ▲

Problem 2.3. Write Python code so you can play the game from Problem 2.1 against the computer. The following incomplete code should get you started. Fill in the missing pieces of the code and play the game several times to be sure that it works. What strategy do you find yourself using to play the game?

```python
from numpy import *
secret = 100*rand(1,1) # get the secret random number
secret = secret[0,0]
tolerance = 0.01

# initial lower bound integer below secret number
a = float(randint(low=0, high=floor(secret)))
# initial lower bound integer below secret number
b = float(randint(low=ceil(secret),high=101))

guess = 1000 # initial guess
n = 1 # initialize the game counter
while abs( secret - guess ) > tolerance and n < 20:
    print('Guess Number',n)
    print('The secret number is between',a,' and ',b)
    guess = float(input('What is your guess for the secret number?'))
    if abs(secret - guess) < tolerance:
        print("Close enough!  The random number was",secret)
        break
    elif guess < (secret - tolerance):
        ... # write appropriate code here in this case
    elif guess > (secret + tolerance):
        ... # write appropriate code here in this case
    n += 1 # increment the game counter
```

▲

2.2 Introduction to Root Finding

In this chapter we want to solve algebraic equations using a computer. Consider the equation $\ell(x) = r(x)$ (where ℓ and r stand for left and right respectively). To solve this equation we can first rewrite it by subtracting the right-hand side from the left to get

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

For example, if we want to solve $3\sin(x) + 9 = x^2 - \cos(x)$ then this is the same as solving $(3\sin(x) + 9) - (x^2 - \cos(x)) = 0$. Hence, we can define a function $f(x)$ as $f(x) = \ell(x) - r(x)$ and observe that <u>every</u> algebraic equation can be written as: "if $f(x) = 0$, find x". We illustrate this idea in Figure 2.1. 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 of Figure 2.1 we see the solutions to the equation $(3\sin(x) + 9) - \left(x^2 - \cos(x)\right) = 0$. From the plots it is apparent that the two equations have the same solutions: $x_1 \approx -2.55$ and $x_2 \approx 2.88$.
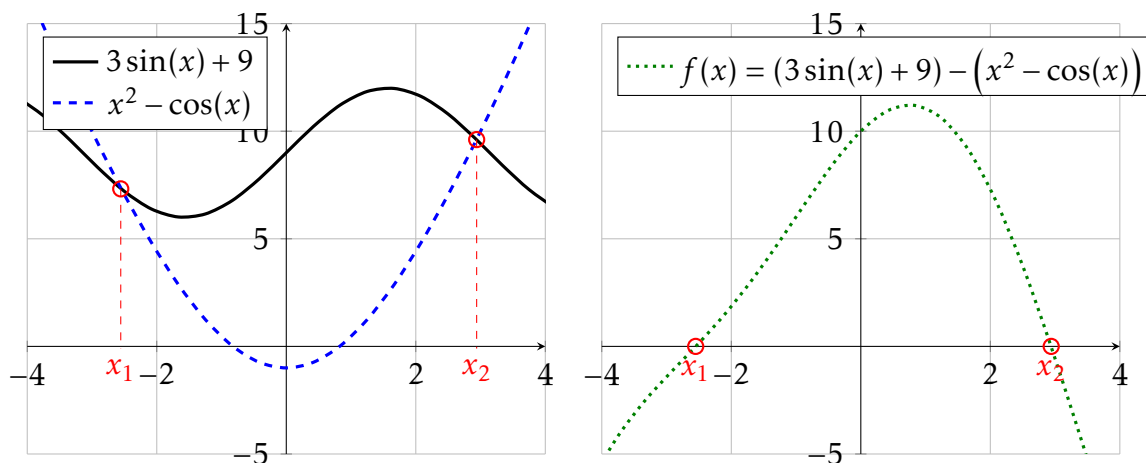
Figure 2.1. The left-hand plot shows two nonlinear functions, $\ell(x) = 3\sin(x) + 9$ and $r(x) = x^2 - \cos(x)$, with their intersection points marked. The right-hand plot shows the equivalent problem formed by solving $\ell(x) - r(x) = 0$.

We now have one way to view every algebraic 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 algebraic equations of the form $f(x) = 0$.

Problem 2.4. With a partner make three different plots just like Figure 2.1. Do so by picking an algebraic root-finding problem, plotting the left and right sides on top of each other (for the left-hand plot) and then plotting the difference between the left and right sides of your equation (for the right-hand plot). Visually verify that the intersections you see on the left-hand plot and the same as the roots on the right-hand plot each time. ▲

2.3   The Bisection Method

We'll start the mathematical discussion with a theorem from Calculus.

> **Theorem 2.5** (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 $f(x_*) = y_*$.

Problem 2.6. Draw a picture of what the intermediate value theorem says graphically. ▲

Problem 2.7. If $y_* = 0$ the intermediate value theorem gives us important information about solving equations. What does it tell us?   ▲

> **Corollary 2.8.** 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 _____.

   Theorem 2.5 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. Let's play a game that might at least tell us how to find the root.

Problem 2.9. Let's make a modification to the game from Problem 2.1. In groups of 2 follow these rules to play a root-finding game.

Step #1: Player 1 chooses a <u>continuous</u> function $f(x)$ that has a root somewhere on the real line.

Step #2: Player 1 chooses real numbers $x = a$ and $x = b$ so that the root of $f(x)$ is between those values.

Step #3: Player 1 tells Player 2 *"the root of my function is between a and b"*.

Step #4: Player 2 makes a guess of the root. We'll call that number m.

Step #5: Player 1 reponds with one of the following three responses.

- If m is within 0.01 of the root of $f(x)$ then say
  "you win!"
- If the root of $f(x)$ is between a and m then say
  "the root is between a and m"
- If the root of $f(x)$ is between m and b then say
  "the root is between m and b"

Step #6: Repeat steps 4 and 5 until Player 2 has a reasonable guess of the root or until you have repeated these steps more than 20 times.

▲

Problem 2.10. Consider the game that you played in the previous problem.

(a) Where was the Intermediate Value Theorem used in the game? Which player was using it and why were they using it?

(b) Why was it important that the function $f(x)$ was continuous in this game?

(c) Propose an optimal strategy for finding the root with the information given.

(d) Write an outline of computer code (called pseudo-code) that will play this game with a given function and given initial values of a and b.

▲

The pseudo-code that you wrote in the previous problem is likely very similar to the "bisection method" for finding roots of an algebraic function. In the following algorithm there are several questions you need to answer.

Algorithm 2.11 (The Bisection Method). 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.

Problem 2.12. Draw a picture illustrating what the Bisection Method does to approximate solutions to the algebraic equation $f(x) = 0$.   ▲

Problem 2.13. 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 Python file. Use the template below to complete your pseudo-code.

```python
1  def Bisection(f , a , b , tol):
2  # The input parameters are
3  # f is a function
4  # a is the lower guess
5  # b is the upper guess
6  # tol is an optional tolerance for the accuracy of the root
7
8  # if the user doesn't define a tolerance we need code to create a default
9
10 # check that there is a root between a and b
11 # if not we should return an error and break the code
12
13 # next calculate the midpoint m = (a+b)/2
14
15 # start a while loop
16 #    # in the while loop we need an if statement
17 #    # if ...
18 #    # elif ...
19 #    # elif ...
20
21 #    # we should check that the while loop isn't running away
22
23 # end the while loop
24 # define and return the root
```

▲

Problem 2.14. Now use the pseudo-code as structure to complete a Python function for the Bisection Method. Also write a test script that verifies that your function works properly. Be sure that it can take a Lambda Functionas 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.

def Bisection(f , a , b , tol):   ▲

Problem 2.15. Test your Bisection Method code on the following algebraic equations.

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

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

3. $(5 - x)e^x = 5$ on $x \in [0, 5]$

▲

Problem 2.16. 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 δ 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 δ then how many iterations of the bisection method do we need to take? Hint: Set the nth approximation error from the table equal to δ. What should you solve for from there?

▲

Problem 2.17. 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.   ▲

Problem 2.18. Create a second version of you 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?   ▲

Problem 2.19. How many iterations of the bisection method are necessary to approximate $\sqrt{3}$ to within $10^{-3}, 10^{-4}, \ldots, 10^{-15}$ using the initial interval $[a,b] = [0,2]$?   ▲

2.4   The Regula Falsi Method

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

Problem 2.20. 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 following three pictures 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?   ▲
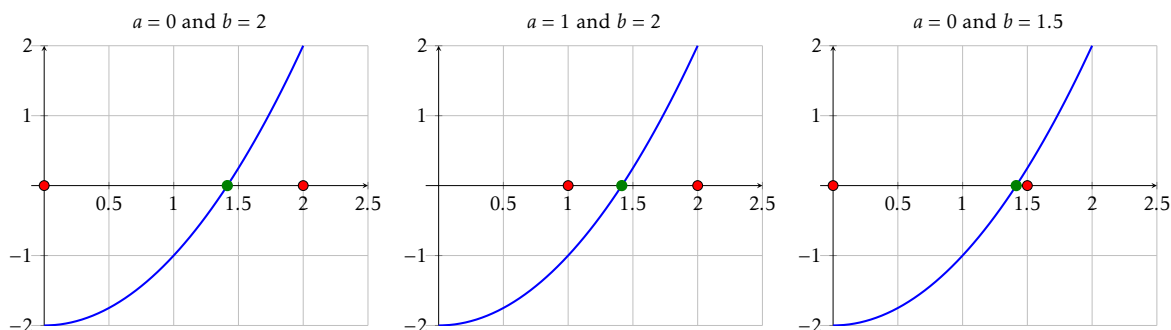
Problem 2.21. Now let's modify the Bisection Method approach. Instead of always using the midpoint (which as you saw in the previous problem could take a little while to converge) let's draw a line between the endpoints and use the x-intercept as the updated guess. If we use this method can we improve 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?

▲
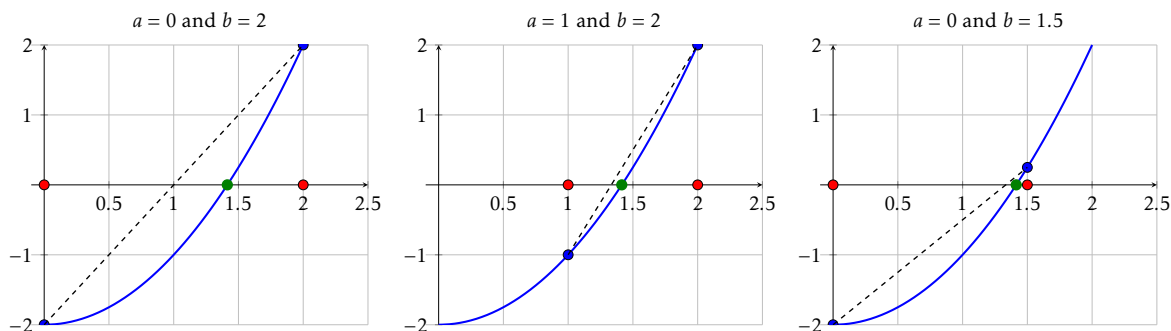
The algorithm that you played with graphically in the previous problem is known as the Regula Falsi (false position) algorithm. It is really just a minor tweak on the Bisection method. After all, the algorithm is still designed to use the Intermediate Value Theorem and to iteratively zero in on the root of the function on the given interval.

Algorithm 2.22 (The Regula Falsi Method)**.** Assume that $f(x)$ is continuous on the interval $[a, b]$. To make approximations of the solutions to the equation $f(x) = 0$, do the following:

1. Check to see if $f(a)$ and $f(b)$ have opposite signs 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?

   $$m = \underline{\hspace{3cm}}$$

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

   $$y - \underline{\hspace{1.5cm}} = \underline{\hspace{1.5cm}} \cdot (x - \underline{\hspace{1.5cm}})$$

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 $x = c$.

   $$c = \underline{\hspace{5cm}}$$

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.

Problem 2.23. Draw a picture of what the Regula Falsi method does to approximate a root. ▲

Problem 2.24. 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. ▲

Problem 2.25. Create a new Python function called `RegulaFalsi` and write comments giving pseudo-code for the Regula-Falsi method. ▲

Problem 2.26. Use your pseudo-code to create a Python function that implements the Regula Falsi method, and write a test script that verifies that your function works properly. Your function should accept a Lambda Functionas 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.
```
def RegulaFalsi(f, a, b, tol):
```
▲

2.5   Newton's Method

Problem 2.27. We are still interested in solving the equation $f(x) = 0$, but now let's leverage some Calculus. 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?

▲
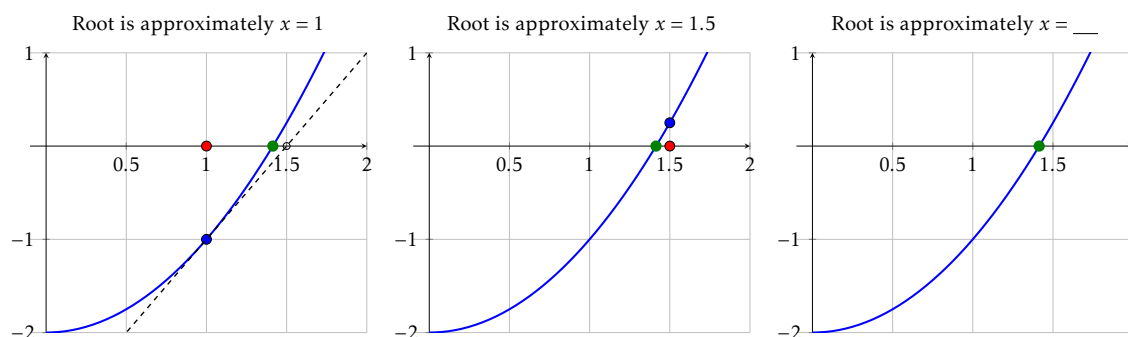
Problem 2.28. If we had started at $x = 0$ in the previous problem what would have happened? Would this initial guess have worked to eventually approximate the root?   ▲

Problem 2.29. 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 solving the algebraic 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.

Algorithm 2.30 (Newton's Method). The Newton-Raphson method for solving algebraic 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 $(x_0, f(x_0))$.

- What is the slope of the tangent line to the function $f(x)$ at the point $(x_0, f(x_0))$?

$$m_{tangent} = \underline{\qquad}$$

- 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 $(x_0, f(x_0))$.

$$y - \underline{\qquad} = \underline{\qquad} \cdot (x - \underline{\qquad})$$

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.

$$x_1 = \underline{\hspace{5cm}}$$

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.

$$x_{n+1} = \underline{\hspace{5cm}}$$

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

Problem 2.31. Draw a picture of what Newton's method does graphically.   ▲

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 iteration formula

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

Problem 2.32 (Failings of Newton's Method). There are several reasons why Newton's method could fail. Work with your partners to come up with a list of all of the reasons. Support each of your reasons with a sketch or an algebraic example.   ▲

Problem 2.33 (An Unexpected Newton Failure). 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	0	$f(0) = 2$
1	$0 - \frac{f(0)}{f'(0)} = 1$	$f(1) = 1$
2	$1 - \frac{f(1)}{f'(1)} =$	
3		
4		

$f(x) = x^3 - 2x + 2$

▲
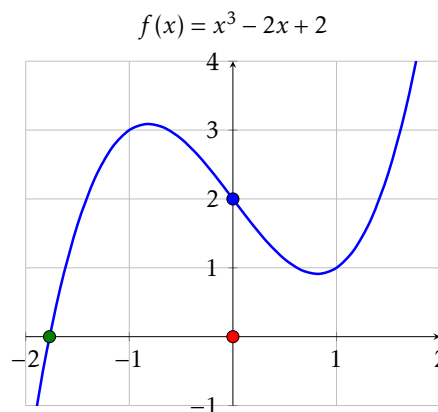
Problem 2.34 (Another Unexpected Failure of Newton's Method). Now let's consider the function $f(x) = \sqrt[3]{x}$. This function has a root $x = 0$. Furthermore, it is differentiable everywhere except at $x = 0$ since

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

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.

$$f(x) = x^3 - 2x + 2$$

n	x_n	$f(x_n)$
0	-1	$f(-1) = -1$
1	$-1 - \frac{f(-1)}{f'(-1)} =$	
2		
3		
4		
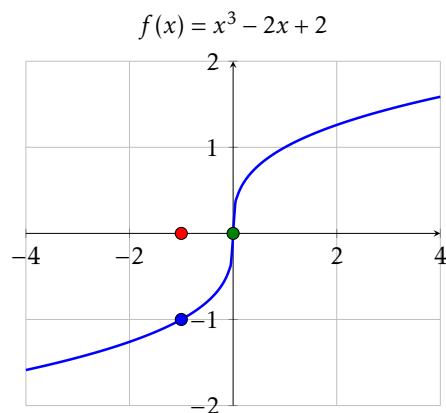
(b) Now let's look at the Newton iteration in a bit more detail. Since $f(x) = x^{1/3}$ and $f'(x) = \frac{1}{3}x^{-2/3}$ the Newton iteration can be simplified as

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

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?

▲

Problem 2.35. Create a new Python function called `Newton` and write comments giving pseudo-code for Newton's method. This version of Newton's method will accept the function and the first derivative so you don't need to set aside any code for calculating the derivative.   ▲

Problem 2.36. Write a Python function for Newton's method. Your function needs to accept a Lambda Function, the derivative of $f(x)$ as a Lambda Function, an initial guess, and an optional error tolerance. 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.

```
def Newton(f , df , x0 , tol):
```
                                                                                      ▲

Problem 2.37. The previous problem required that you calculate the derivative ahead of time for Newton's method. While this is relatively easy with a computer algebra system it might be easier just to have your function compute the derivative for you. Write a new Python function for Newton's method that accepts the function symbolically,, the initial point, and an optional error tolerance. Your function must compute the derivative symbolically behind the scenes.

```
def NewtonSymbolic(f, x0, tol):
```
                                                                                      ▲

Problem 2.38. In Problem 2.32 you constructed several examples of where Newton's method will fail. Modify your Newton's method code to catch these special cases and warns the user before the method fails. Test your new code with several of the special cases showing that indeed you were able to catch them properly and give the user appropriate feedback.   ▲

Problem 2.39. Newton's Method is known to have a *quadratic convergence rate*. This means that there is some constant C such that

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

where x_* is the root that we're hunting for. To simplify the notation a bit we define $\varepsilon_k = |x_k - x_*|$ as the error in the Newton iteration at step k. Therefore, there must be a constant C such that

$$\varepsilon_{k+1} \le C\varepsilon_k^2.$$

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

$$\log(\varepsilon_{k+1}) = \log(C) + 2\log(\varepsilon_k),$$

and we see that this is a linear function (on a log-log plot) and the slope is 2.

In this problem we're going to build a numerical experiment to verify that Newton's method indeed has quadratic convergence. Modify your Newton's method code so that it outputs all of the iterations instead of just the final root. Once you have the iterations compute the error between the approximations and the exact root. For simplicity let's solve the equation $x^2 - 2 = 0$. Plot the sequence of error approximations with the iterate ε_k on the x-axis and the iterate ε_{k+1} on the y-axis of a log-log plot. Quadratic convergence means that at every iteration the error should decrease by roughly 2 orders of magnitude. How can you see this in your plot?   ▲

2.6 Quasi-Newton Methods

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 obvious way to approximate the derivative is just to use a secant line instead of 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.

Algorithm 2.40 (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.

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.)

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.

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

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

Problem 2.41. Draw several pictures showing what the Secant method does pictorially. ▲

Problem 2.42. Write Python code for solving algebraic equations of the form $f(x) = 0$ with the Secant method. You should ALWAYS start by writing pseudo-code as comments in your Python file. Your function should accept a Lambda Function, two starting points, and an optional error tolerance. Also write a test script that clearly shows that your code is working.

```
def SecantMethod(f, x0, x1, tol):
```
▲

Problem 2.43. Choose a non-trivial algebraic equation for which you know the solution and write a script to empirically determine the convergence rate of the Secant method. You may want to look back at 2.39. ▲

2.7 Exercises

2.7.1 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" them method works, but give all of the mathematical details for "why" it works.

Problem 2.44. 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 $[a, b]$.   ▲

Problem 2.45. 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 $[a, b]$.   ▲

Problem 2.46. 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 $f(x)$.   ▲

Problem 2.47. 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 $f(x)$.   ▲

2.7.2 Applying What You've Learned

Problem 2.48. In this problem you will demonstrate that all of your root finding codes work. At the beginning of this chapter we proposed the algebraic equation solving problem

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

Write a Python 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. Clearly all of your root finding routines will be stored as Python functions so be sure to turn in all of these functions along with this problem.   ▲

Problem 2.49 (Order of Convergence)**.** 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 kth iteration of the root finding technique, and x_{k+1} is the $(k+1)^{st}$ iteration of the root finding technique. We will call the value $\varepsilon_k = |x_k - x_*|$ the "old" error since it is the error in the method at the old step. We will call the value $\varepsilon_{k+1} = |x_{k+1} - x_*|$ the "new" error since it is the error in the method at the new step.

(a) If we consider the equation

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

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

$$\log\left(|x_{k+1} - x_*|\right) \le \underline{\hspace{2cm}} + \underline{\hspace{2cm}}$$

(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 three different log-log plots of the new error to the old error for different root finding techniques. What is the order of the convergence rate for each of these methods?

(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.
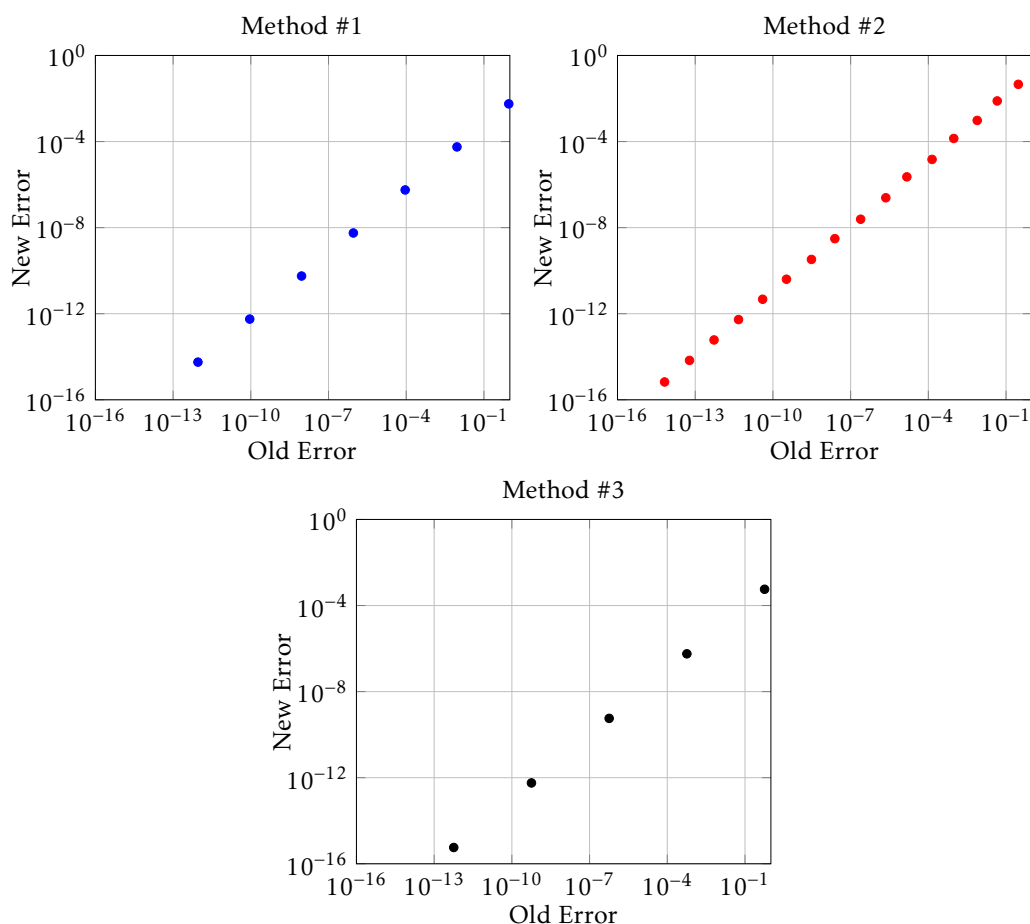
Problem 2.50.   (a) Write code and create plots to demonstrate that the Bisection Method has approximately first order convergence. Demonstrate this result on several differen equation-solving problems.

(b) Write code and create plots to demonstrate that Newton's Method has second order convergence. Demonstrate this result on the same equation-solving problems from part (a).

▲

Problem 2.51. Multiple Choice: For every step of Newton's method the new estimate of the error between the Newton iteration and the exact answer is approximately:

(a) the same as it was on the previous step.

(b) ten times smaller than it was on the previous step

(c) one hundred times smaller than it was on the previous step

(d) one thousand times smaller than it was on the previous step

Defend your answer with a sentence or two.   ▲

Problem 2.52. Write Python code that produces a loglog plot that compares the convergence rates for the bisection method, the regula-falsi method, Newton's method, and the secant method on a problem where we know the exact answer.   ▲

Problem 2.53. Show several plots that demonstrates that the Chebyshev Method,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{1}{2}\left(\frac{f(x_n)}{f'(x_n)}\right)^2 \frac{f''(x_n)}{f'(x_n)},$$

exhibits a third-order convergence rate. That is, create a log-log plot of the successive errors for several different equation-solving problems. What does this "third order convergence" mean about the successive approximations in the Chebyshev iterations? What are the pro's and con's to using Chebyshev's method instead of Newton's method?   ▲

Problem 2.54. Compare the number of iterations necessary for convergence to within 10^{-8} for both the bisection method and the regula falsi method on several test problems. Find example problems where bisection converges faster and examples where regula falsi converges faster. Write a test script that clearly indicates to the user which equation was being solved, which endpoints were used, and which root finding technique performed faster.   ▲

Problem 2.55. 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 $m = 1$kg, $g = 9.8$m/s^2, $k = 0.1$, and $s_0 = 100$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?

▲

Problem 2.56 (Modified from [1]). 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. Follow the steps below to arrive at an approximate answer.

(a) Solve the simple differential equation $v_y'(t) = -g$ by hand where $v_y(t)$ is the vertical velocity of the canon and gravity is given as $g \approx 9.8$m/s^2. We don't know the initial velocity so just use $v(0) = v_0$ and hence $v_y(0) = v_0 \sin(\theta)$. Note: your answer will have a "v_0" and a "$\sin(\theta)$" in it.

(b) Solve the differential equation $s_y'(t) = v_y(t)$ by hand for the position function $s_y(t)$. Assume that $s_y(0) = 0$. Your answer will still have a "v_0" and a "$\sin(\theta)$" in it.

(c) Solve $s_y(t) = 0$ by hand for t in terms of v_0 and θ to find a function for the amount of time the projectile takes to reach the ground.

(d) In the absence of air resistance the projectile will have a constant velocity in the horizontal direction. Solve the differential equation $s_x'(t) = v_0 \cos(\theta)$ for the horizontal position function $s_x(t)$.

(e) The range function $R(v_0, \theta)$ can be found by substituting the time from part (c) into the horizontal position function in part (d). Find the function $R(v_0, \theta)$.

(f) For a certain projectile and canon the initial velocity is $v_0 = 126$m/s. We want to give the artillery officer a distance, d, and have them calculate the angle to hit the target. Write Python code to approximate θ in the equation

$$R(126, \theta) = d.$$

(Hint: remember that we can rewrite the equation $R(126, \theta) = d$ in the form $f(\theta) = 0$ for some function f so that we can actually use our numerical root finding techniques. See the discussion at the very beginning of this chapter.)

Report a table of values of the form shown below and provide an appropriate plot showing your results. Clearly some distances will be out of range so be sure to clearly indicate the range of the weapon.

Distance (d meters)	Angle (θ)
0	
25	
50	
100	
$\vdots$	

<div align="right">▲</div>

Problem 2.57. Consider our four primary root finding methods: Bisection, Regula Falsi, Newton's method, and the Secant method.

(a) For each method give a mathematical situation where you might want to use the method and give a mathematical situation where you might not want to use the method.

(b) For each of these methods give one example where the method will fail.

Support your arguments with proper mathematics. You are welcome to argue graphically but be sure to fully explain your graphical reasoning.   ▲

Problem 2.58. 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 do the algebraic 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 $f(x) = x^3(x-3)(x-6)^4$ and explicitly demonstrate, without the use of a plot, that you have found all of the local extrema.   ▲

Problem 2.59. 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 $f(x) = \sqrt{\frac{8}{x+6}}$.

<div align="right">▲</div>

2.8   Projects

In this section we propose several ideas for projects related to numerical 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.

2.8.1   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.

(a) $f(x) = (x - 4)(x + 1)$

(b) $g(x) = (x - 1)(x + 3)$

(c) $h(x) = (x - 4)(x - 1)(x + 3)$

(d) Find a single-variable function of your own that has an interesting picture of the basins of attraction.

(e) 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.

(f) 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.

Chapter 3

Numerical Calculus

In this chapter we build some of the common techniques for approximating the two primary computations in calculus: taking derivatives and evaluating definite integrals.

3.1   Why Do Calculus Numerically?

Recall the typical tools of 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.

Problem 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.   ▲

Problem 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.

Problem 3.3. In statistical 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.   ▲

Problem 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.   ▲

Problem 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

Problem 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. 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 this section we'll build several approximation of first and second derivatives. The idea for each of these approximation is:

- Partition the interval $[a,b]$ into N points.

- Approximate the derivative at the point $x \in [a,b]$ 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 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.
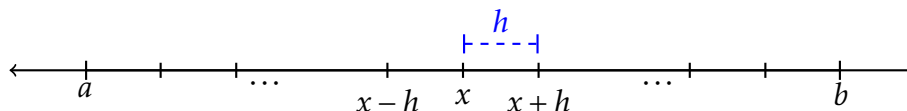
Figure 3.1. A partition of an interval on the real line.

If we recall that the definition of the first derivative of a function is

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{3.1}$$

our first approximation for the first derivative is naturally

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x)}{h}. \tag{3.2}$$

In (3.2) 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*. The linear combination that we spoke about before is

$$\frac{df}{dx} \approx \frac{1}{h}f(x+h) - \frac{1}{h}f(x).$$

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.

Problem 3.7. From Taylor's Theorem we know that for an infinitely differentiable function $f(x)$,

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

What do we get if we replace x with $x+h$ and a 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 $x+h$.

$$f(x+h) = \rule{6cm}{0.4pt}$$

▲

Problem 3.8. 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)

$$f'(x) = \frac{f(x+h) - ???}{??} + \rule{3cm}{0.4pt}$$

▲

Problem 3.9. If we were to drop everything after the fraction in the previous problem we know that we would be introducing error into our derivative computation. According to Taylor's Theorem, how can we quantify the error of the approximation?   ▲

Definition 3.10 (Order of a Numerical Derivative). The **order** of a numerical derivative is the power of the step size in the remainder term. For example, a first order method will have "h^1" in the remainder term. A second order method will have "h^2" in the remainder term.

Theorem 3.11 (First Order Approximation of the First Derivative). In problem 3.8 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.

In the previous definition, "$\mathcal{O}(h)$" (read: big-O of h) states that the method is first order. This means that the maximum error that you're making with this method is on the order of the size of the step. Not surprisingly, if we let h get arbitrarily small then the error in this method gets arbitrarily small. More formally we have the following definition.

> **Definition 3.12** (Big $\mathcal{O}$ Notation)**.** We say that the error in a differentiation method is "big O of h", $E = \mathcal{O}(h)$, if and only if there is a positive constant M such that
>
> $$|Error| \leq M|h|.$$
>
> This is equivalent to saying that a differentiation method is first order.

Problem 3.13. Explain what the phrase

> "*The approximation of $f'(x)$ in Theorem 3.11 is $\mathcal{O}(h)$*"

into your own words.   ▲

Problem 3.14. 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 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?

▲

Problem 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	
0.02	
0.002	

▲

Problem 3.16. Explain the phrase:

The first derivative approximation

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

is first order.

▲

Now that we have a handle on how the technique of approximating the first derivative works, let's build some code that will take in a function and output the approximate first derivative at a collection of points.

Problem 3.17. Write Python code that takes a function and a domain $(xmin, xmax)$ and returns a numerical approximation to the derivative on the interval $(xmin, xmax)$. You should ALWAYS start by writing pseudo-code as comments in your Python file. Your function should accept a Lambda Function, the bounds on the domain, and the number of interior points used for approximation within the domain. Your function should output the x values and y values associated with the derivative.

```
def FirstDerivFirstOrder(f,xmin,xmax,num_interior_pts):
```
▲

The only two ways to really check a numerical derivative are to plot the numerical approximation and to do the derivative by hand and to plot the error. Be warned, however, that the numerical derivative that we have built from Theorem 3.11 should have one less value than the original list of x and y values. Think about why this must be true. Also double check your code from the previous problem and make sure that you can plot `newx` vs `yprime` without having to change their size.

Example 3.18 (Efficient Coding with Vectors). Let's pause now for a few words on efficient vector-based coding. We can absolutely write code for the first derivative using a `for` loop as you can see in the code below. Notice that I have resized the x vector so that it is the same length as the y' vector.

Code:

```
1  def FirstDerivFirstOrder(f,xmin,xmax,num_interior_pts):
2      import numpy as np
3      x = np.array(np.linspace(xmin,xmax,num_interior_pts))
4      y = f(x)
5      yprime = np.zeros(y.size-1)
6      newx = np.zeros_like(yprime)
7      for n in range(0,y.size-1):
8          newx[n] = x[n]
9          yprime[n] = (y[n+1]-y[n]) / (x[n+1] - x[n])
10     return(newx, yprime)
```

Implementing this code we can use the following script.

Code:

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  %matplotlib inline # for use in Jupyter Notebooks
4
5  f = lambda x: np.sin(x) # define the function of interest
6  xmin = 0
7  xmax = 2*np.pi
8  numpts = 100
9  x = np.linspace(xmin,xmax,numpts)
10 y = f(x)
11 dx, dy = FirstDerivFirstOrder(f,xmin,xmax,numpts)
12 plt.plot(x,y,'b--',newx,yprime,'r--')
13 plt.grid()
```

The output of this code will give the sine function along with the derivative, the

cosine function as seen in Figure 3.2.

Now let's build the first derivative function in a much smarter way. Instead of looping over all of the elements we can take advantage of the fact that every thing is stored in vectors. Hence we can just do vector operations and do all of the subtractions and divisions at once.

```
1  def FirstDerivFirstOrder(f,xmin,xmax,num_interior_pts):
2      import numpy as np
3      x = np.array(np.linspace(xmin,xmax,num_interior_pts))
4      y = f(x)
5      yprime = (y[1:]-y[:-1]) / (x[1:] - x[:-1])
6      newx = x[:-1]
7      return(newx, yprime)
```
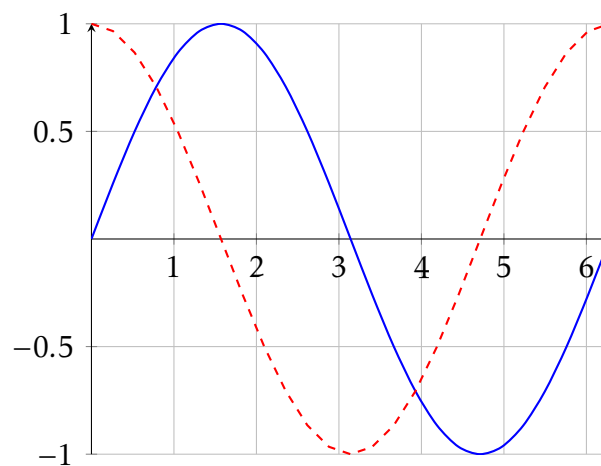

Figure 3.2. The result of using the first derivative code with 100 interior point on the function $f(x) = \sin(x)$.

Problem 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).

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

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  %matplotlib inline
4
5  f = lambda x: np.sin(x) - x*np.sin(x)
6  xmin = 0
7  xmax = 15
```

```
8  numpts = 1000
9  x = np.linspace(xmin,xmax,numpts)
10 y = f(x)
11 newx, yprime = FirstDerivFirstOrder(f,xmin,xmax,numpts)
12
13 fig, ax = plt.subplots(1,2)
14 ax[0].plot(x,y,'b',newx,yprime,'r--')
15 ax[0].grid()
```

2. 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.

```
1  df = lambda x:   # write a function for the exact derivative
2  dy = df(newx)
3  ax[1].semilogy(newx,abs(dy - yprime))
4  ax[1].grid()
```

Discuss how you can see the fact that this is a first order method. You may want to put one (or both) of the plots on a log-log scale or a semi-log scale ... I'll let you figure out which one makes the most sense.   ▲

Let's return to the mathematics:
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.

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

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

(a) This time, replace x with $x - h$ and a with x and simplify.

$$f(x-h) = \underline{\hspace{6cm}}$$

(b) You should already have the Taylor series for $f(x+h)$. Subtract $f(x+h)$ and $f(x-h)$ and simplify. Be very careful of you signs.

$$f(x+h) - f(x-h) = \underline{\hspace{6cm}}$$

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

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

▲

Problem 3.21. If we were to drop the terms after the fraction in the last result then we would be making an error in the derivative computation. According to Taylor's Theorem quantify the error that is being made.   ▲

Theorem 3.22 (Second Order Approximation of the First Derivative).

$$f'(x) = \underline{\hspace{5cm}} + \mathcal{O}(h^2)$$

Problem 3.23. 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 Problem 3.14 (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 contant M?

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

▲

Problem 3.24. 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	

▲

Problem 3.25. Write a Python function that takes a function and a domain and returns a second order numerical approximation to the first derivative on the interval. You should ALWAYS start by writing pseudo-code as comments in your Python file. Your function should accept a function, the bounds on the domain, and the number of interior points used for approximation within the domain. Your function should output the x values and y values associated with the derivative.

```
def FirstDerivSecondOrder(f,xmin,xmax,num_interior_pts):
```

You should try to write this code without using any `for` loops.   ▲

Now we'll search for an approximation of the second derivative. Again, the game will be the same: play with the Taylor series and some algebra and hope that the second derivative pops out. 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.

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

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

Problem 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 of the **second derivative**.

(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.

$$f(x+h) + f(x-h) = \underline{\hspace{6cm}}.$$

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

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

▲

Problem 3.27. 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. According to Taylor's Theorem, how can we quantify the order of the error?
▲

Problem 3.28. Again consider the function $f(x) = \sin(x)(1 - x)$.

(a) Calculate the <u>second</u> 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 Problem 3.26 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 $h = 0.005$? Be able to defend your answers.

▲

Problem 3.29. Write a Python function that takes a 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 Python file. Your function should accept a function, the bounds on the domain, and the number of interior points used for approximation within the domain. Your function should output the x values and y values associated with the derivative.
`def SecondDerivSecondOrder(f,xmin,xmax,num_interior_pts):` Again, you should write your code without using any `for` loops.   ▲

Problem 3.30. 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.

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

Table 3.1. First and second derivatives.

(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?

▲

Table 3.1 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.

Problem 3.31. 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 abolute 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 differnce 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

Problem 3.32. Consider the shaded area of the region under the function plotted in Figure 3.3 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?

(c) Work with your partner to give an estimate of the area and provide an estimate for the amount of error that you're making.

▲
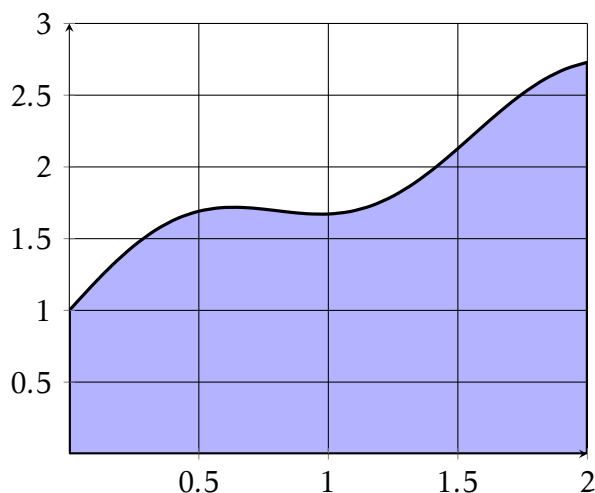
Figure 3.3. Approximate the area under the curve between $x = 0$ and $x = 2$.

In this section we will build methods for approximating integrals. Recall from Calculus that the definition of the Riemann integral is

$$\int_a^b f(x)dx = \lim_{\Delta x \to 0} \sum_{j=1}^{N} f(x_j)\Delta x \tag{3.3}$$

where N is the number of subintervals 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

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

You are likely familiar with this approximation of the integral from Calculus. The value of x_j can be chosen anywhere within the subinterval and three common choices are to use the left endpoint, the midpoint, and the right endpoint. We see a depiction of this in Figure 3.4.
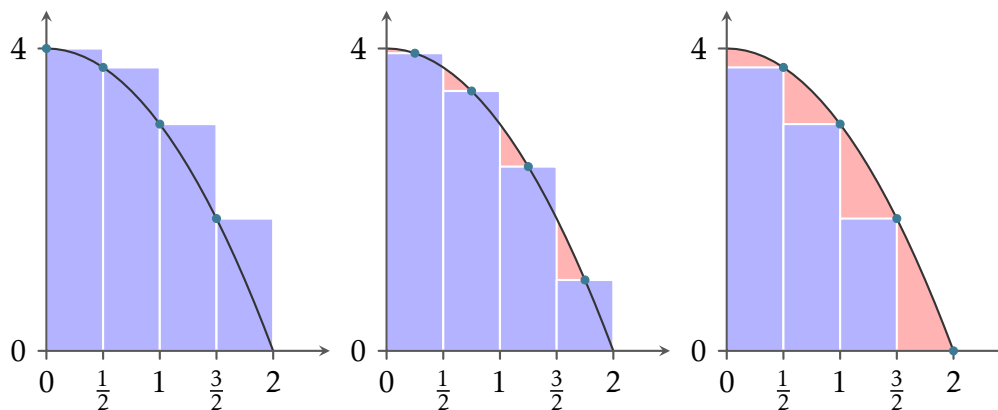
Figure 3.4. Riemann sum to approximate an integral with left, midpoint, and right rectangles on the function $f(x) = 4 - x^2$ on the interval x $in[0, 2]$.

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

Problem 3.33. Write Python code approximate an integral with Riemann sums. You should ALWAYS start by writing pseudo-code as comments in your Python file. Your Python function should accept a Lambda 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.

```
def MyRiemannSum(f, a, b, num_subintervals, type):
```
Test your code on several functions for which you know the integral. You should write your code without any `for` loops.   ▲

Problem 3.34. 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
$2^{-2} = 0.25$			
$2^{-3} = 0.125$			
2^{-4}			
2^{-5}			
2^{-6}			
2^{-7}			
2^{-8}			
2^{-9}			

(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.

▲

Problem 3.35. Repeat Problem 3.34 using right-justified rectangles. ▲

Theorem 3.36. 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 _____.

The previous theorem could be stated in an equivalent way.

Theorem 3.37. 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 _____.
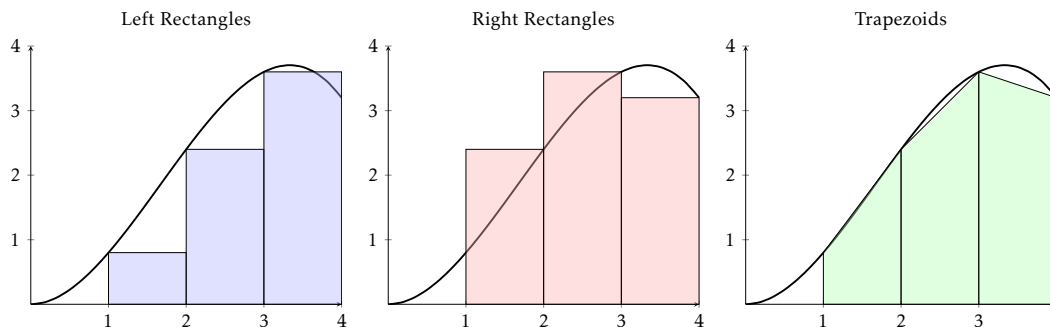
Figure 3.5. Using rectangles and trapezoids with $\Delta x = 1$ for approximating the integral $\int_1^4 f(x)dx$.

Problem 3.38. Create a plot with the width of the subintervals on the horizontal axis and the absolute error between your (left) Riemann sum calculation 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?   ▲

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 3.5. 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.

Problem 3.39. Consider a single trapezoid approximating the area under a curve. From geometry we recall that the area of a trapezoid is

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

where b_1, b_2 and h are marked on the picture below. 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

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

Trapezoids

$\blacktriangle$
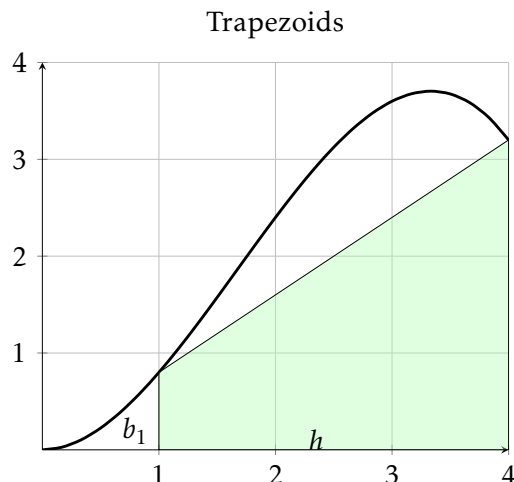
Problem 3.40. Now use the same idea with $h = \Delta x = 1$ from figure 3.5 to approximate the area under the function $f(x) = (1/5)x^2(5 - x)$ between $x = 1$ and $x = 4$ using three trapezoids. $\blacktriangle$

Problem 3.41. Consider the function $f(x) = \frac{1}{5}x^2(5 - x)$ again.

(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			
$\vdots$	$\vdots$	$\vdots$	$\vdots$

(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?

$\blacktriangle$

Technique 3.42. We want to approximate $\displaystyle\int_a^b f(x)dx$. 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:

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

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

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

3. Approximate the integral as

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

Problem 3.43. 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 the area by examining a plot (i.e. you have a visual estimate of the area). ▲

Problem 3.44. 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 I calculate the definite integral with a fixed Δx and get an absolute percent error, P, then what absolute percent error will I get if I use a width of $\Delta x/M$ for some positive number M? ▲

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.

Problem 3.45. We want to find constants $A1$, $A2$, and $A3$ 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

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

is <u>exact</u> 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.

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

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

▲

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.

Problem 3.46. 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

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

▲

Problem 3.47.   (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?

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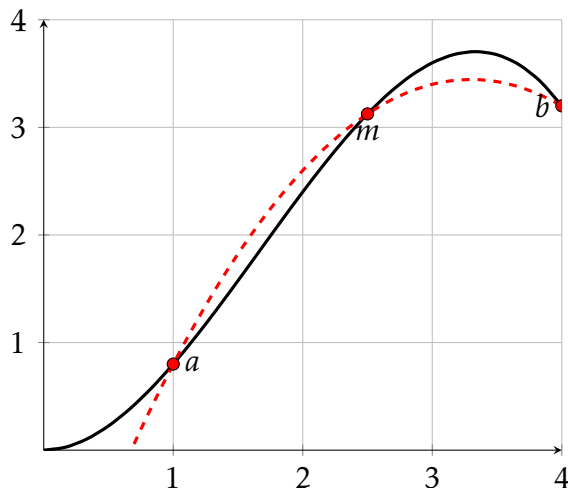
To make the punchline of the previous problems a bit more clear, using the formula

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

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 approxmiate 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.

Technique 3.48 (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.

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

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

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

3. Approximate the integral as

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

Problem 3.49. 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 $\Delta x/M$? ▲

Problem 3.50. Write Python functions that implement both the trapezoidal rule and Simpson's rule. You should ALWAYS start by writing pseudo-code as comments in your Python file. Keep in mind that Python (numpy arrays, specifically)deals with vectors and iteration in very nice ways. You shouldn't need a `for` loop in your function.
Test both Python functions 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 3.51 (Numerical Integration Techniques). 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: $\displaystyle \int_a^b f(x)dx \approx \sum_{j=1}^N f(x_j)\Delta x$

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

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

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

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

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.

Problem 3.52. Theorem 3.51 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. ▲

Example 3.53. In Figure 3.6 you'll see the error for our three primary integration techniques on the problem

$$\int_0^1 \sin(2x)dx$$

which has an exact answer of

$$\int_0^1 \sin(2x)dx = \frac{1}{2} - \frac{\cos(2)}{2}.$$

You'll see that Simpson's rule gains 4 orders of accuracy for every order of magnitude that the Δx is decreased. Similarly you'll see that Trapezoidal method gains 2 orders of magnitude of accuracy for every order of magnitude that Δx is decreased, and Riemann sums only gain 1 order of magnitude. Plots such as this can reveal the performance of a numerical method and give you a sense of how it is going to behave on a problem where you don't know the exact answer.
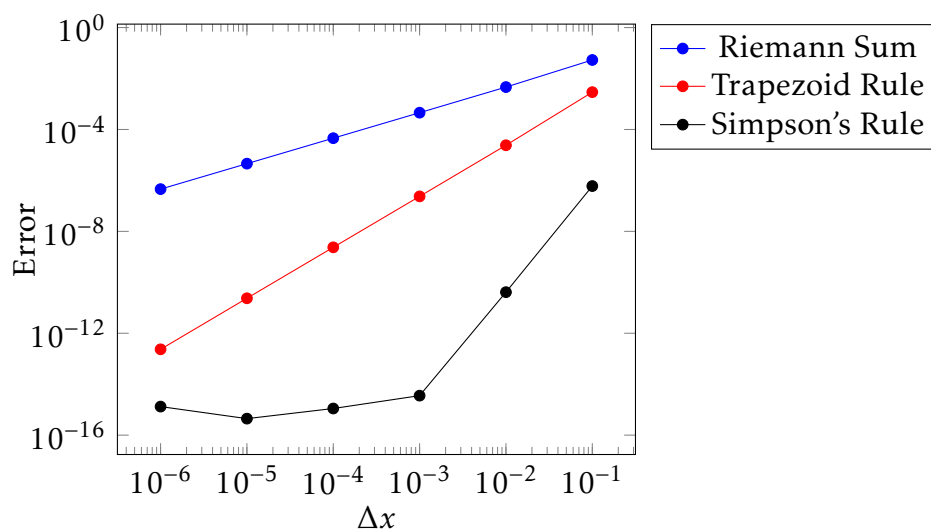
Figure 3.6. Errors with integration techniques.

3.4 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.

Problem 3.54 (The Classic Cardboard Box Problem). A piece of cardboard measuring 20cm by 20cm is to be cut so that it can be folded into a box without a lid. We want to find the size of the cut, x, that maximizes the volume of the box.
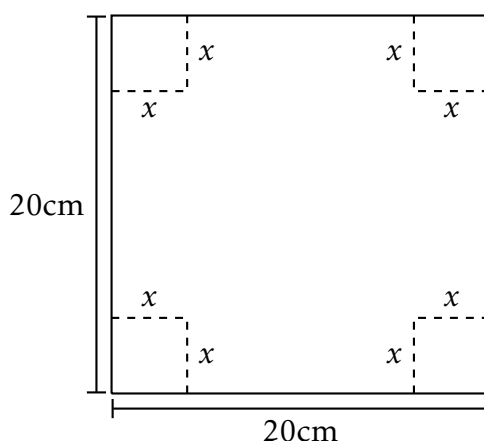
(a) Write a function for the volume of the box resulting from a cut of size x. What is the domain of your function?

(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.

▲

Now we'll propose a few numerical techniques that can approximate the solution to these types of problems. The basic ideas are simple!

Problem 3.55. 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.   ▲

Problem 3.56. 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).

Problem 3.57. Let's turn your intuition into an algorithm. If $f(x)$ is the function that you are trying to maximize then turn your algorithm from Problem 3.55 into a step-by-step algorithm which could be coded. Then try out your code on the function

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

to see if your algorithm can find the local maximum near $x \approx 1.14$.   ▲

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.

Algorithm 3.58 (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)

Problem 3.59. Write code to implement the 1D derivative free optimization algorithm and use it to solve Problem 3.54. Compare your answer to the analytic solution.   ▲

Algorithm 3.60 (Gradient Ascent/Descent)**.** 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)

Problem 3.61. Write code to implement the 1D gradient descent algorithm and use it to solve Problem 3.54. Compare your answer to the analytic solution.   ▲

Algorithm 3.62 (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

Problem 3.63. Write code to implement the 1D monte carlo search algorithm and use it to solve Problem 3.54. Compare your answer to the analytic solution.   ▲

Algorithm 3.64 (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).

Problem 3.65. Write code to implement the 1D numerical root finding optimization algorithm and use it to solve Problem 3.54. Compare your answer to the analytic solution. ▲

Problem 3.66. 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?

▲

Problem 3.67. The Gradient Ascent/Descent algorithm is the most geometrically inter-esting 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. ▲

Problem 3.68. (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. ▲

Problem 3.69. (This problem is modified from [6])
Reconsider the pig problem 3.68 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. ▲

Problem 3.70. Numerical optimization is often seen as quite challenging since the algo-rithms that we have introduced here could all get "stuck" at local extrema. To illustrate this see the function shown in Figure 3.7. 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? ▲
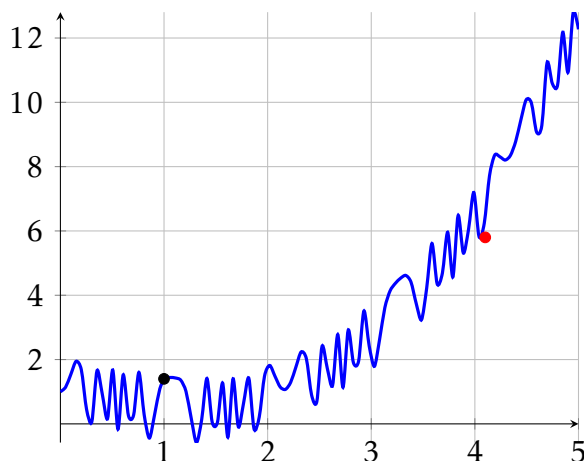
Figure 3.7. A challenging numerical optimization problem. If we start at the black point then how will any of our algorithms find the local minimum at the red point?

3.5 Curve Fitting – The Least Squares Problem via Numerical Optimization

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. Instead of finding the roots of an algebraic equation, what 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?

Problem 3.71. 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?

▲

Problem 3.72. 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.

(c) Now let's be a bit more systematic about things. 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.

    (i) Pick an arbitrary starting value for a. For each of the four points find the error between the predicted y value and the actual y value.

   (ii) Square all four of your errors and add them up. (Stop and think: why are we squaring the errors before we sum them ... have a discussion here)

  (iii) 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.

  (iv) 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.

▲

Problem 3.73. 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. Have your loop stop when you think that it has reached the best value of a. ▲

Technique 3.74 (Nonlinear Least Squares Curve Fitting). 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 $a_1, a_2, \ldots$ then

1. Pick initial values for the parameters $a_1, a_2, \ldots$ 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)

2. Calculate the square error between the data point and the prediction from the function $f(x)$
$$\text{error for the point } x_i: e_i = (y_i - f(x_i))^2.$$

   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.

3. 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.)

> 4. Change the parameters $a_1, a_2, \ldots$ so as to minimize the sum of the square errors.

In Technique 3.74 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. In this particular section of this text we'll look at two of them: (1) Solver in MS Excel and (2) `scipy.optimize.fmin` in Python's `scipy` package. Before we get into how those techniques work let's look at an example of how the minimization step works graphically.

Let's return to the data set

$$S = \{(0, 1.07), (1, 3.9), (2, 14.8), (3, 26.8)\}$$

under the assumption that the curve we want to use is quadratic: $f(x) = ax^2 + bx + c$.*
In Figure 3.8 we see the points in red along with several guesses for the values of the parameters. In these figures we are just changing one of the parameters while keeping the other two fixed for easier illustration. In reality we should note that all three of the parameters need to be changed to achieve the desired minimization.

In Figure 3.8 we saw that for $b = 2$ and $c = 0.7$ then the quadratic function $f(x) \approx 2.5x^2 + 2x + 0.7$ is a pretty good approximation of the data $S = \{(0, 1.07), (1, 3.9), (2, 14.8), (3, 26.8)\}$.
If, however, we take finer increments between our guesses for a we can get a refined estimate of the value of a that minimizes the sum of the squared errors. In Figure 3.9 we see the values of the sum of the squared errors for many estimates of a. In 3.9 we see that $a \approx 2.3$ gives a slightly better value of the sum of the square errors and is (visually) near the minimum of the error curve.

Problem 3.75. For each of the plots in Figure 3.10 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 $f(x) = ax^2 + bx + c$).
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. Your instructor will make the data available to you.   ▲

It should be clear to the reader at this point that curve fitting is actually an optimization problem. For every collection of parameters in your chosen function there is a unique sum of square errors. The goal, of course, is to pick parameters that minimize the sum of square errors. It is beyond the scope of this book to write a gradient descent or derivative free optimization routine to do this optimization. Instead we will lean on several pre-built tools that can do this optimization for us: MS Excel's Solver and Python's `scipy.optimize.fmin` to name two of them. If you are curious about how the optimization occurs see a course in mathematical optimization or machine learning.

Problem 3.76. In Problem 3.75 you made hypotheses about the types of functions that fit the data shown in 3.10. Use Excel's Solver function to minimize the sum of the square residuals for between your parameterized function and the data.   ▲

---

*Take note that since we have four points then we could find an exact interpolating fit if we used a cubic function instead, but that isn't the point. In fact, using higher order polynomials has the disadvantage of potentially fitting the curve to measurement noise.
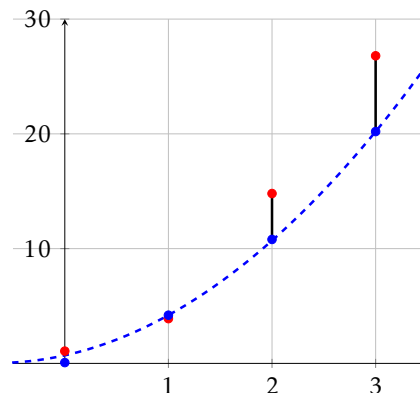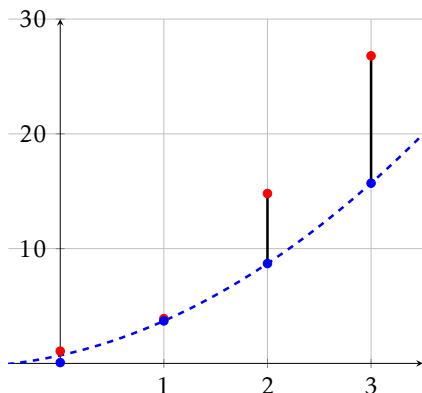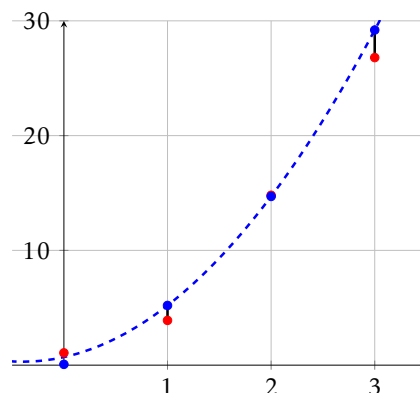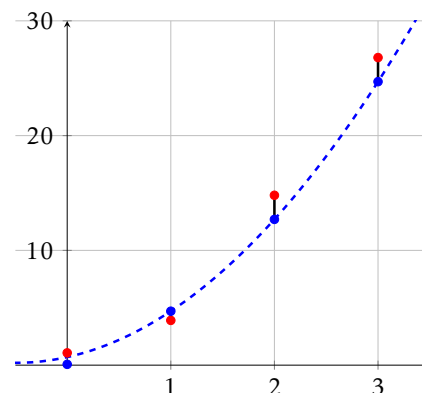
Figure 3.8. Demonstration of the curve fitting technique. The vertical black bars between the points show the point-wise error in the estimate. In these four pictures we are only modifying the value of a while leaving $b = 2$ and $c = 0.7$ fixed for simplicity. In reality we need to change all three parameters in the minimization process - hence the actual minimization takes place in three dimensions.
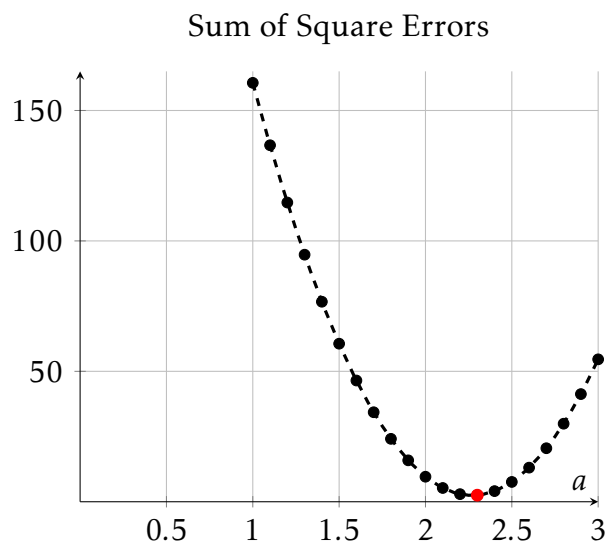
Figure 3.9. The sum of the squared errors for the function $f(x) = ax^2 + bx + c$ for the data set $S = \{(0, 1.07), (1, 3.9), (2, 14.8), (3, 26.8)\}$. For $b = 2$ and $c = 0.7$ the minimum occurs on this curve at approximately $a \approx 2.3$ with the sum of the square errors of approximately 2.51. (To build this plot we took increments of 0.1 between different guesses for a from 1 to 3 – a very brute-force way of approximating a minimum.

Problem 3.77. Repeat the previous problem with Python's `scipy.optimize.fmin` function.   ▲

Problem 3.78. With your partner come up with an algebraic function governed by several parameters. Add some random noise to your function and then save the data (without any formulas). Give your data to a different group in the class and have them reverse engineer your function from your data. The second group should report the function, the parameters, and the sum of the square errors.   ▲
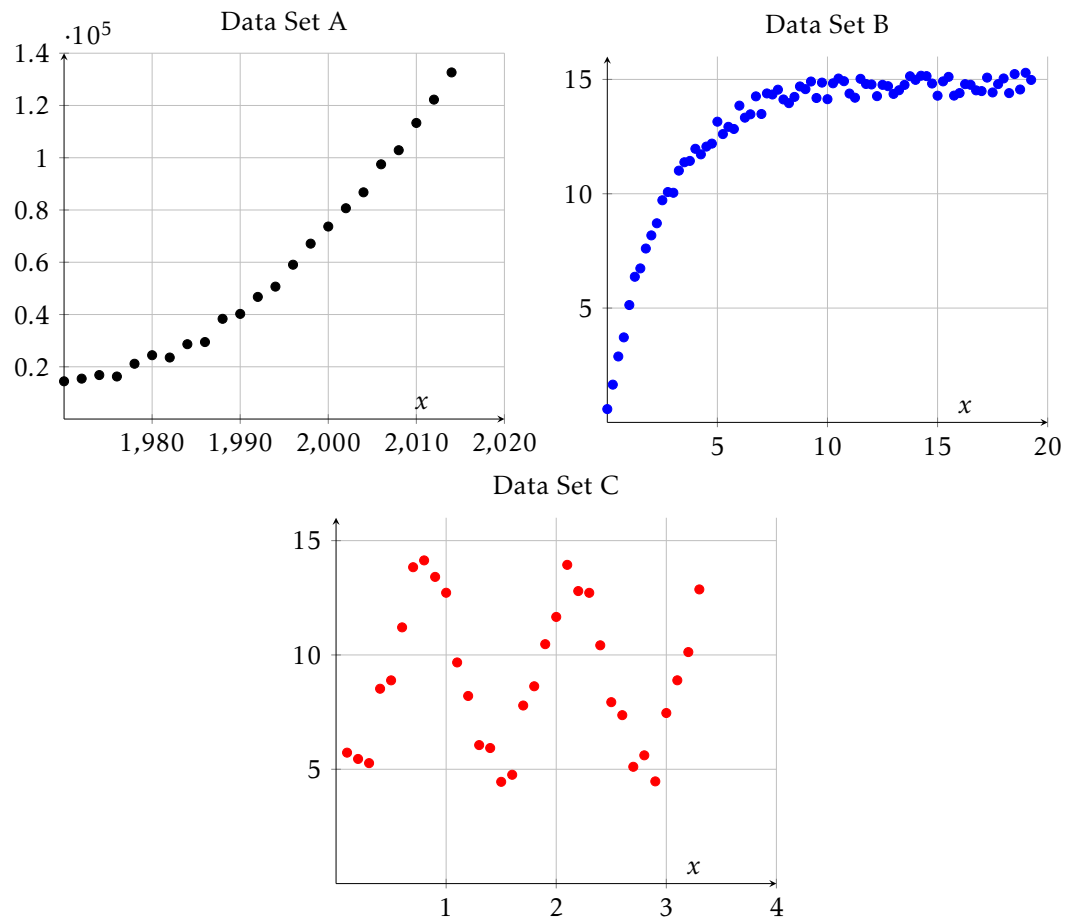
Figure 3.10. Several data plots for Problem 3.75.

3.6   Exercises

3.6.1   Algorithm Summaries

Problem 3.79. 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.   ▲

Problem 3.80. 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.   ▲

Problem 3.81. 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.   ▲

Problem 3.82. 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. ▲

Problem 3.83. 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.   ▲

Problem 3.84. 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.   ▲

Problem 3.85. Explain in clear language how the derivative free optimization method works on a single-variable function.   ▲

Problem 3.86. Explain in clear language how the gradient descent/ascent optimization method works on a single-variable function.   ▲

Problem 3.87. Explain in clear language how the Monte Carlo search optimization method works on a single-variable function.   ▲

Problem 3.88. Explain in clear language how you find the optimal set of parameters given a set of data and a proposed general function type.   ▲

3.6.2 Applying What You've Learned

Problem 3.89. 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. It would be very wise to redo problems 3.7 - 3.8, 3.20, and 3.26 before beginning this problem.

(a) $f'(x) \approx \dfrac{\frac{1}{12}f(x-2h) - \frac{2}{3}f(x-h) + \frac{2}{3}f(x+h) - \frac{1}{12}f(x+2h)}{h}$

(b) $f'(x) \approx \dfrac{-\frac{3}{2}f(x) + 2f(x+h) - \frac{1}{2}f(x+2h)}{h}$

(c) $f''(x) \approx \dfrac{-\frac{1}{12}f(x-2h) + \frac{4}{3}f(x-h) - \frac{5}{2}f(x) + \frac{4}{3}f(x+h) - \frac{1}{12}f(x+2h)}{h^2}$

(d) $f'''(x) \approx \dfrac{-\frac{1}{2}f(x-2h) + f(x-h) - f(x+h) + \frac{1}{2}f(x+2h)}{h^3}$

▲

Problem 3.90. Write a Python function that accepts a list of (x,y) ordered pairs from an Excel 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 Excel file and a test script that have graphical output showing that your Python function is finding the correct derivative. ▲

Problem 3.91. Write a Python function that accepts a list of (x,y) ordered pairs from an Excel spreadsheet 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 Excel file and a test script that have graphical output showing that your Python function is finding the correct derivative. ▲

Problem 3.92. Write a Python 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 an Excel file. Create a test Excel file and a test script showing that your Python function is finding the correct integral. ▲

Problem 3.93. 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

(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.

▲

Problem 3.94. 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 the Python's `scipy.integrate.quad` command. (What we're really doing here is comparing our algorithms to Python 's built in algorithm). ▲

Problem 3.95. 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 your code from Problem 3.90 to evaluate and plot the derivative. If your data appears to be subject to significant noise then use the Excel curve fitting tools first to smooth the data; then do 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 your code from Problem 3.92 to evaluate the definite integral. If your data appears to be subject to significant noise then use the Excel curve fitting tools first to smooth the data; then do 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. ▲

Problem 3.96. Integrate each of the functions over the interval $[-1, 2]$ 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) $f(x) = \dfrac{x}{1 + x^4}$

(b) $g(x) = (x-1)^3(x-2)^2$

(c) $h(x) = \sin\left(x^2\right)$

▲

Problem 3.97. 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

▲
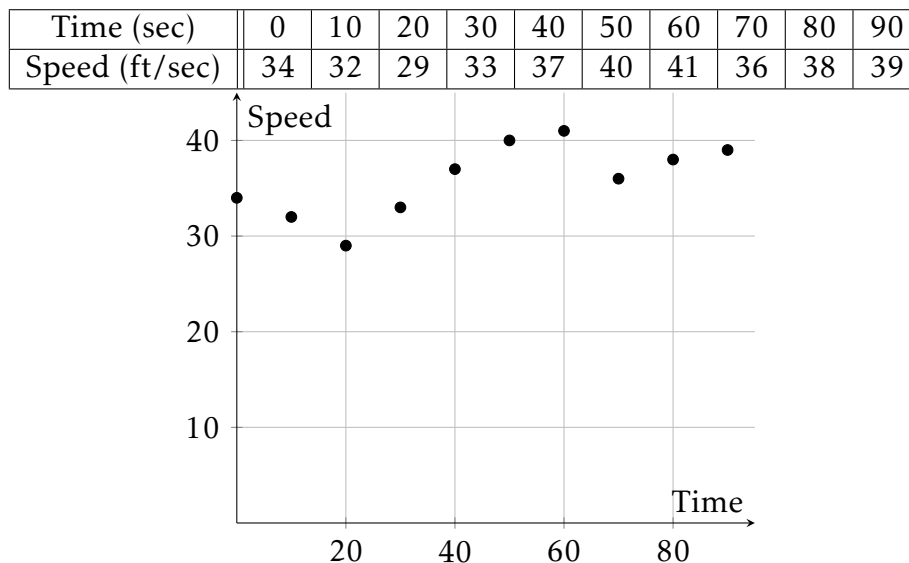
Problem 3.98. For each of the following functions write code to find the local maximum or minimum that is closest to $x = 0$. You may want to start with a plot of the function just to get a feel for where the local extreme value might be.

(a) $f(x) = \dfrac{x}{1+x^4} + \sin(x)$

(b) $g(x) = (x-1)^3 \cdot (x-2)^2 + e^{-0.5 \cdot x}$

▲

Problem 3.99. 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. ▲

Problem 3.100. The plots in Figure 3.11 show noisy data from measurements of elementary single variable functions.

(a) Make a hypothesis about which 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.

Your instructor will provide you with the raw data. Note that in the pictures the points are connected with line segments simply for visual effect. The functions that you propose must be continuous functions.   ▲
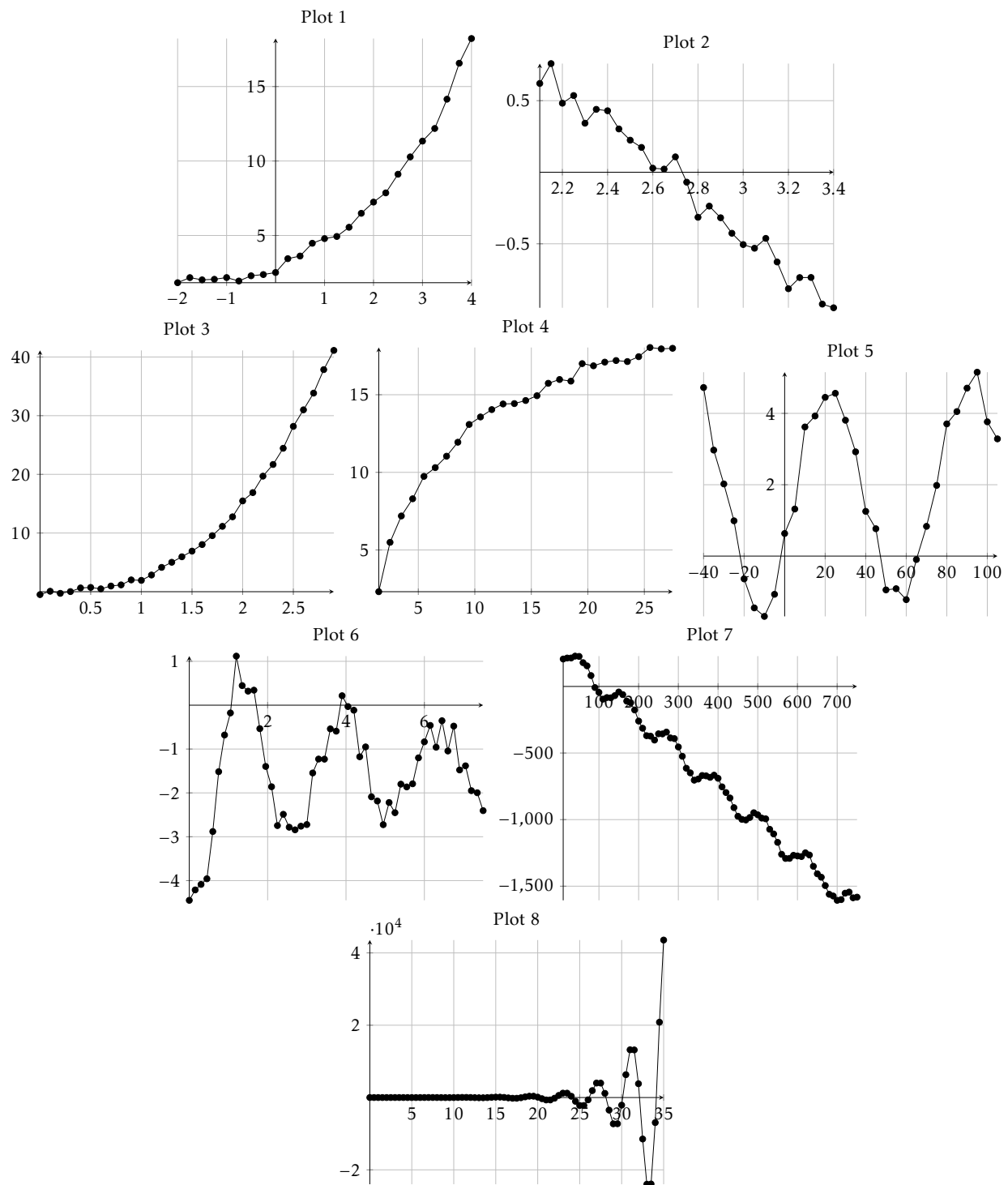
Figure 3.11. Plots for Problem 3.100.

3.7 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.

3.7.1 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".[†] In the Excel file associated with this project 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 integrated 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

---

[†]www.newscientist.com/article/dn14573-galactic-spaghetti-monster-powered-by-magnetic-fields/

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.

3.7.2   Higher Order Integration

Riemann sums can be used to approximate integrals and they do so by using piecewise constant functions to approximate the function. The trapezoidal rule uses piecewise 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 piecewise 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 piecewise 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.

3.7.3   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 dropdown 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.

[‡]I would be surprised if you didn't do some statistics along the way.

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).

Chapter 4

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 and build. 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 and under-determined systems of equations, the eigenvalue-eigenvector problem, and the singular value problem. 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.

4.1 Matrix Operations

We start this chapter 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. You'll find in numerical linear algebra that the indexing and the housekeeping in the codes is the hardest part, so why don't we start "easy".

Problem 4.1. Recall that the dot product of two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ is

$$\mathbf{u} \cdot \mathbf{v} = \sum_{j=1}^{n} u_j v_j.$$

Without summation notation,

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n.$$

Write a Python function that accepts two vectors and returns the dot product.
```
def MyDotProduct(u, v):
```
It would be wise to put an error check in your code to make sure that the vectors are the same size. You should be able to write this code without any loops. ▲

Problem 4.2. Write a test script for the dot product of two random $n \times 1$ vectors. Your script should output the absolute error between your dot product code and Python 's `numpy.dot` command.   ▲

Problem 4.3. Recall that 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 moments reflection reveals that each entry in the matrix product is actually a dot product,

$$(AB)_{ij} = (\text{Row } i \text{ of matrix } A) \cdot (\text{Column } j \text{ of matirx } B).$$

Write a Python function that accepts two matrices and returns the matrix product.
```
def MatrixMultiply(A, B):
```
You should be able to write this code with only two loops; one for i and one for j. The rest of the work should be done using dot products.   ▲

Problem 4.4. Write a test script for the matrix product of two random matrices of appropriate sizes. Your script should output the normed error between your matrix product code and Python 's matrix multiplication. (Remember to cast your numpy arrays as matrices in order to get proper matrix multiplication in Python.)   ▲

If you're having trouble seeing that matrix multiplication is just a bunch of dot products then let's examine a fairly simple example. You are welcome to use this example to test your code in the previous problem, but be sure that your code is robust enough to accept matrices of any size.

Example 4.5. Find the product of matrices A and B using dot products.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \qquad B = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

Solution:
The product AB will clearly be a 3×3 matrix since $A \in \mathbb{R}^{3 \times 2}$ and $B \in \mathbb{R}^{2 \times 3}$. To build the matrix product we'll first write matrix A as a matrix filled with row vectors and matrix B as a matrix filled with column vectors. Let $\mathbf{a}_1 = \begin{pmatrix} 1 & 2 \end{pmatrix}$, $\mathbf{a}_2 = \begin{pmatrix} 3 & 4 \end{pmatrix}$, and $\mathbf{a}_3 = \begin{pmatrix} 5 & 6 \end{pmatrix}$ so that we can write A as

$$A = \begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix}.$$

Similarly, write $\mathbf{b}_1 = \begin{pmatrix} 7 \\ 10 \end{pmatrix}$, $\mathbf{b}_2 = \begin{pmatrix} 8 \\ 11 \end{pmatrix}$, and $\mathbf{b}_3 = \begin{pmatrix} 9 \\ 12 \end{pmatrix}$ so that we can write B as

$$B = \begin{pmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \mathbf{b}_3 \end{pmatrix}.$$

Now to build the matrix multiplication we see that AB is given as

$$AB = \begin{pmatrix} \mathbf{a}_1 \cdot \mathbf{b}_1 & \mathbf{a}_1 \cdot \mathbf{b}_2 & \mathbf{a}_1 \cdot \mathbf{b}_3 \\ \mathbf{a}_2 \cdot \mathbf{b}_1 & \mathbf{a}_2 \cdot \mathbf{b}_2 & \mathbf{a}_2 \cdot \mathbf{b}_3 \\ \mathbf{a}_3 \cdot \mathbf{b}_1 & \mathbf{a}_3 \cdot \mathbf{b}_2 & \mathbf{a}_3 \cdot \mathbf{b}_3 \end{pmatrix}$$

Therefore,

$$AB = \begin{pmatrix} 27 & 30 & 33 \\ 61 & 68 & 75 \\ 95 & 106 & 117 \end{pmatrix}$$

The following Python code gives the matrix product between two matrices, but when you build your own code you need to have sufficient catches that avoid inappropriately sized matrices.

```
1  AB = np.zeros( (A.shape[0], B.shape[1]) )
2  for i in range(0,A.shape[0]): # row index
3      for j in range(0,B.shape[1]): # column index
4          AB[i,j] = np.dot( A[i,:] , B[:,j])
```

Problem 4.6. In matrix arithmetic there are two types of multiplication: matrix-matrix multiplication and scalar multiplication. Modify your code written in problem 4.3 so that if one of the two matrices is entered as a scalar (a 1×1 matrix) then your `MatrixMultiply` code gives the correct result. You should be able to do this with no loops.   ▲

4.2   Efficiently Solving Systems of Linear Equations

One of the many classic problems of linear algebra is to solve the linear system $A\mathbf{x} = \mathbf{b}$. In this chapter we will talk about efficient ways to have the computer solve these systems. You likely recall row reduction (AKA Gaussian Elimination or RREF) from previous linear algebra courses, but the algorithm that you used is actually slow an cumbersome for computer implementation. Even so, let's blow the dust off of what you recall with a small practice problem.

Problem 4.7. Solve the following problem by hand using Gaussian Elimination (row reduction).

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}$$

Hint: Start by augmenting the coefficient matrix and the right-hand column vector to build the augmented system

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 0 & 2 \end{array} \right).$$

Then perform row operations to get to the reduced row echelon form

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & \star \\ 0 & 1 & 0 & \star \\ 0 & 0 & 1 & \star \end{array} \right)$$

▲

4.2.1   Lower Triangular Systems

Row reduction works well on dense (or nearly dense) matrices but if the coefficient matrix has special structure then we can avoid row reduction in lieu of faster algorithms. Furthermore, the process that you know as Gaussian Elimination is really just a collection of sneaky matrix operations. In the following two problems you will devise algorithms for triangular matrices. After we know how to work with triangular matrices we'll build a general tool for doing Gaussian Elimination that is easily implemented in a computer.

Problem 4.8. Outline a fast algorithm (without formal row reduction) for solving the lower triangular system

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}.$$

As a convention we will always write our lower triangular matrices with ones on the main diagonal. Your outline should be a list of explicit steps to solve the system. The most natural algorithm that most people devise here is called *forward substitution*.   ▲

Technique 4.9 (Forward Substitution: `LSolve`). The following code solves the problem $L\mathbf{y} = \mathbf{b}$ using forward substitution. The matrix L is assumed the be lower triangular with ones on the main diagonal.

Code:

```
1  def LSolve(L, b):
2      L = np.matrix(L) # make sure L is the correct data type
3      n = b.size
4      y = np.matrix( np.zeros( (n,1)) )
5      for i in range(0,n):
6          y[i] = b[i]
7          for j in range(0,i):
8              y[i] = y[i] - L[i,j] * y[j]
9      return(y)
```

Take note in the code that we are leveraging the fact that Python will not enter a `for` loop if the ending value is less than the starting value.

Problem 4.10. Consider the lower triangular system

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}.$$

Work the code from Technique 4.9 by hand to solve the system. Keep track of all of the indices as you work through the code. ▲

Problem 4.11. Copy the code from Technique 4.9 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 $\mathbf{b}$ and solve for $\mathbf{y}$. Your code needs to work on systems of arbitrarily large size. ▲

4.2.2 Upper Triangular Systems

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.

Problem 4.12. Outline a fast algorithm (without formal row reduction) for solving the upper triangular system

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & -9 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -4 \\ 3 \end{pmatrix}$$

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. ▲

Technique 4.13 (Backward Substitution: `USolve`)**.** The following code solves the problem $U\mathbf{x} = \mathbf{y}$ using backward substitution. The matrix U is assumed the be upper triangular. You'll notice that most of this code is incomplete. It is your job to complete this code.

Code:

```
1  def USolve(U, y):
2      U = np.matrix(U)
3      n = y.size
4      x = np.matrix( np.zeros( (n,1)))
5      for i in range(? , ? , ?):      # what should we be looping over?
6          x[i] = y[i] / ???           # what should we be dividing by?
7          for j in range(? , ? , ?): # what should we be looping over:
8              x[i] = x[i] - U[i,j] * x[j] / U[i,i]
9      return(x)
```

Problem 4.14. Consider the upper triangular system

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & -9 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -4 \\ 3 \end{pmatrix}$$

Work the code from Technique 4.13 by hand 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. ▲

Problem 4.15. Copy the code from Technique 4.13 into a Python function but in your code write a comment on every line stating what it is doing. Write a test script that creates an upper triangular matrix of the correct form and a right-hand side **y** and solve for **x**. Your code needs to work on systems of arbitrarily large size. ▲

4.2.3 The LU Factorization

In the next few problems we will solve the system of equations

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}$$

using upper and lower triangular matrices. We have already solved this problem with your Gaussian Elimination algorithm at the beginning of this chapter – now let's improve upon that algorithm and reveal some amazing underlying structure.

Throughout the following several problems, let A and b be defined as

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}.$$

Problem 4.16. In Python enter the matrix A (Remember to use `A = np.matrix()` to define a Python matrix). Observe what happens when we do the following in sequence:

- left multiply A by the matrix

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

  This gives the matrix L_1A. (Look at the resulting matrix. What happened here?)

- left multiply L_1A by the matrix

$$L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -7 & 0 & 1 \end{pmatrix}$$

  This gives the matrix L_2L_1A. (Look at the resulting matrix. What happened here?)

- left multiply L_2L_1A by the matrix

$$L_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix}$$

  (Look at the resulting matrix. What happened here?)

▲

Problem 4.17. Make a conjecture: 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 lower triangular matrix? ▲

Problem 4.18. After the process from the previous problem you should notice that you now have an upper triangular matrix. Hence, in general, we have done this:

$$L_3L_2L_1A = U,$$

so if you solve for A we see that A can be written as

$$A = L_1^{-1}L_2^{-1}L_3^{-1}U.$$

Therefore, we could rewrite the problem $A\mathbf{x} = \mathbf{b}$ as the problem $LU\mathbf{x} = \mathbf{b}$ where L is which matrix? ▲

Problem 4.19. In the previous problem you likely found that $L = L_1^{-1}L_2^{-1}L_3^{-1}$. Use Python to find L_j^{-1} for each j and discuss general observations about how to find inverses of lower triangular matrices. ▲

Problem 4.20. Now for the punch line: If we want to solve $A\mathbf{x} = \mathbf{b}$ then if we can write it as $LU\mathbf{x} = \mathbf{b}$ we can

1. Solve $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$ using forward substitution. Then,

   2. solve $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$ using backward substitution.

For our running example, write down L and U and solve the system. ▲

Problem 4.21. Try the process again on the 3×3 system of equations

$$\begin{pmatrix} 3 & 6 & 8 \\ 2 & 7 & -1 \\ 5 & 2 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -13 \\ 4 \\ 1 \end{pmatrix}$$

That is: Find matrices L and U such that $A\mathbf{x} = \mathbf{b}$ can be written as $LU\mathbf{x} = \mathbf{b}$. Then do two triangular solves to determine $\mathbf{x}$.

Notice that this time there isn't a "1" in the top left corner to begin with. Be careful. ▲

Technique 4.22 (LU Factorization). The following Python function takes a square matrix A and outputs the matrices L and U such that $A = LU$. Partial code is given below. Complete the code.

Code:

```
1  def MyLU(A):
2      n = A.shape[0]
3      if A.shape[0] != A.shape[1]:
4          print('Error: The matrix A is not square.')
5      L = np.matrix( np.identity(n) )
6      U = A
7      for j in range(0,n-1):
8          for i in range(j+1,n):
9              mult = A[i,j] / A[j,j]
10             A[i, j+1:n] = A[i, j+1:n] - mult * A[j,j+1:n]
11             U[i, j+1:n] = A[i, j+1:n]
12             L[i,j] = mult
13             U[i,j] = 0
14     return(L,U)
```

Theorem 4.23 (LU Factorization Algorithm). Let A be a square matrix in $\mathbb{R}^{n \times n}$ and let $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$. To solve the problem $A\mathbf{x} = \mathbf{b}$,

   1. Factor A into lower and upper triangular matrices $A = LU$.

      `L, U = MyLU(A)`

   2. The system can now be written as $LU\mathbf{x} = \mathbf{b}$. Substitute $U\mathbf{x} = \mathbf{y}$ and solve the system $L\mathbf{y} = \mathbf{b}$ with forward substitution.

      `y = LSolve(L, b)`

   3. Finally, solve the system $U\mathbf{x} = \mathbf{y}$ with backward substitution.

      `x = USolve(U, y)`

Problem 4.24. Test your `MyLU`, `LSolve`, and `USolve` 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. Discuss where your code will fail. Use the following partial code to test your functions.

Code:

```
1  import numpy as np
2  A = np.matrix( ) # put the matrix A here
3  b = np.matrix( ) # put the vector b here
4  ExactSoln = np.linalg.solve(A,b) # get the exact answer from numpy's linalg.solve()
5
6  L, U = MyLU(A) # get the L and U matrices
7  y = LSolve(L,b) # do the lower solve
8  x = USolve(U,y) # do the upper solve
9
10 Error = np.linalg.norm(ExactSoln - x) # check the size of the difference
11 print(Error)
```

▲

Problem 4.25. For this problem we are going to run a numerical experiment to see how the process of solving the equation $A\mathbf{x} = \mathbf{b}$ using the *LU* factorization performs.
Create a loop that does the following
(Note to Python Users: You will need to run the import command:
`from numpy.random import rand`
to get the `rand` command.)

- Build a random matrix of size $n \times n$. You can do this with the code:
  `A=rand(n,n)`

- Build a random vector in $\mathbb{R}^n$. You can do this with the code:
  `b = rand(n,1)`

- Find Python 's exact answer to the problem $A\mathbf{x} = \mathbf{b}$ using the `numpy.linalg.solve` command
  `Xexact = numpy.linalg.solve(A,b)`

- Write code that uses your three *LU* functions (`MyLU`, `Lsolve`, `Usolve`) to find a solution to the equation $A\mathbf{x} = \mathbf{b}$.

- Find the error between your answer and the exact answer using the code:
  `error = numpy.linalg.norm(x - Xexact)`

- Make a plot 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.

▲

Problem 4.26. Write a summary of how the `LU` factorization solve a square system of equations.   ▲

4.3 The QR Factorization

There are several instances where the LU factorization will not perform very fast. In fact, there are several ways that the LU factorization (the way that we have stated it) will fail. In this section we propose a new factorization that improves the performance of solving a linear system in some instances.

Let's say that we want to find $\mathbf{x}$ such that $A\mathbf{x} = \mathbf{b}$ but where $\mathbf{b}$ is not in the column space of A. Strictly speaking there is no solution to this type of equation – the column vector $\mathbf{b}$ cannot be written as a linear combination of the columns of A. If we multiply both sides of this equation by A^T we build what are called the **normal equations:**

$$A^T A\mathbf{x} = A^T \mathbf{b}.$$

In this equation, the right-hand side is a collection of the projections of $\mathbf{b}$ onto the column space of A. The nicest type of projections occur when the columns are perpendicular to each other. Hence, our goal is to take A and factor it into $A = QR$ where the columns of Q are orthonormal (orthogonal and normalized) and R is an upper triangular matrix. In this case we get the following advantages:

- If $A\mathbf{x} = \mathbf{b}$ then we can write $QR\mathbf{x} = \mathbf{b}$.

- Multiplying by Q^T on both sides gives $R\mathbf{x} = Q^T\mathbf{b}$.

- If R is upper triangular then the resulting solve can be done very quickly.

Problem 4.27. If $Q \in \mathbb{R}^{m \times n}$ is an orthonormal matrix then what are the products $Q^T Q$ and QQ^T? ▲

Problem 4.28. If $A = QR$ where Q is an orthonormal matrix and R is upper triangular then how would we go about solving the equation $A\mathbf{x} = \mathbf{b}$ and where would the QR factorization help us? ▲

Problem 4.29. First we'll do a problem by hand:

Consider the matrix $A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$. We want to factor A into $A = QR$ where the columns of Q are orthonormal and R is upper triangular. Here is the algorithm (run every step by hand!). For notation purposes, $\mathbf{a}_j$ will be the jth column of A.

1. Define $\mathbf{q}_1 = \dfrac{\mathbf{a}_1}{\|\mathbf{a}_1\|}$. This will be the first column of Q.

2. Define $\mathbf{q}_2 = \mathbf{a}_2 - (\mathbf{a}_2 \cdot \mathbf{q}_1)\mathbf{q}_1$. Once you've done this calculation normalize your result so $\mathbf{q}_2 = \dfrac{\mathbf{q}_2}{\|\mathbf{q}_2\|}$. This is the second column of Q.
   Explain the geometry of this step (DRAW A PICTURE!)

3. Define $\mathbf{q}_3 = \mathbf{a}_3 - (\mathbf{a}_3 \cdot \mathbf{q}_1)\mathbf{q}_1 - (\mathbf{a}_3 \cdot \mathbf{q}_2)\mathbf{q}_2$ and then redefine $\mathbf{q}_3 = \dfrac{\mathbf{q}_3}{\|\mathbf{q}_3\|}$. This is now the third column of Q.

4. The matrix R is formed as follows:

$$
R = \begin{pmatrix}
\mathbf{a}_1 \cdot \mathbf{q}_1 & \mathbf{a}_2 \cdot \mathbf{q}_1 & \mathbf{a}_3 \cdot \mathbf{q}_1 \\
0 & \mathbf{a}_2 \cdot \mathbf{q}_2 & \mathbf{a}_3 \cdot \mathbf{q}_2 \\
0 & 0 & \mathbf{a}_3 \cdot \mathbf{q}_3
\end{pmatrix}
$$

5. Write down Q and observe that the process is just begging for several loops on a computer to implement this on bigger matrices.

6. Use Python to check that $A = QR$.

7. Finally let's look at the utility of the result:

   (a) Since Q is an orthonormal matrix $Q^T Q$ is the identity matrix! (Explain why.)

   (b) If we want to solve $A\mathbf{x} = \mathbf{b}$ and we can write $A = QR$ then

   $$
   A\mathbf{x} = \mathbf{b} \quad \implies \quad QR\mathbf{x} = \mathbf{b} \quad \implies \quad R\mathbf{x} = Q^T\mathbf{b}
   $$

   (c) Since R is upper triangular we can use the `Usolve` code we have from our work with the LU-factorization to solve for $\mathbf{x}$. THIS IS REALLY FAST!!

   (d) Compare this to the number of computer operations needed so solve the normal equations $A^T A\mathbf{x} = A^T\mathbf{b}$ with an LU-solver.

▲

Problem 4.30. Write Python code that takes a matrix A (not necessarily square) and outputs both the Q matrix and the R matrix. Pseducode for this function is as follows:

- Set up the function call:
  ```
  def MyQR(A):
  ```

- Set up `zeros` matrices for both Q and R. Remember that A may not be square so assume that the columns are in $\mathbb{R}^m$ and the rows are in $\mathbb{R}^n$. Hence Q will be $m \times n$ and R will be $n \times n$.

- Start a loop that counts across the columns:
  ```
  for j in range(0,n):
  ```

  - define a temporary variable: $\hat{\mathbf{q}} = \mathbf{a}_j$
  - start a loop that will do all of the subtractions and build some of the $R's$:
    ```
    for i in range(0,j-2):
    ```
    * build one of the $R's$: $R_{ij} = \mathbf{a}_j \cdot \mathbf{q}_i$
    * Do one of the subtractions: $\hat{\mathbf{q}} = \hat{\mathbf{q}} - R_{ij}\mathbf{q}_i$

- – end the loop for `i`
- – normalize the jth column in Q: $\mathbf{q}_j = \hat{\mathbf{q}}/\|\hat{\mathbf{q}}\|$
- – build the $R's$ on the diagonal: $R_{jj} = \mathbf{a}_j \cdot \mathbf{q}_j$

- end the loop for `j`

▲

Problem 4.31. Write a script that tests your `MyQR` function on randomly generated $m \times n$ matrices with randomly generated right-hand sides. Compare the time that it takes to solve the least squares problems using QR to the time necessary to solve with LU via the normal equations. When is LU more efficient? When is QR more efficient?   ▲

4.4   Curve Fitting – The Least Squares Problem via Linear Algebra

Problem 4.32. Grab the data sets from the Google Sheet HERE. Copy the data into MS Excel for the following exercise. We're going to develop a way to match the data sets using linear algebra. Before doing the linear algebra versions of this problem though we'll use Excel to match the data sets. Our goal is to make a guess at the type of polynomial that models the function (given in this case) and to minimize the error between our guess and the data. Follow these steps in Excel to find best fitting curves. We'll start with the linear data.

1. In column C set up a function for your guess of the model based on the x data in column A. You will need to set up cells for the polynomial parameters (for the linear case these are the slope and the y-intercept).

2. Fill in your guesses based on initial approximations for the slope and y-intercept.

3. Use column D to calculate the residual value for each data point.

$$\text{Residual} = \text{Actual } y \text{ Value} - \text{Approximate } y \text{ Value}$$

4. Use column E to calculate the square of the residual for each data point.

5. Our goal is the minimize the sum of the squares of the residuals.

$$\min\left(\sum_{j=1}^{n} \left(y_j - \hat{y}_j\right)^2\right)$$

   For this we can use the `Excel Solver` to minimize the sum of the square residuals.

Now repeat the process for the quadratic data.   ▲

At this point we need to discuss the quality of the fits that we are building. You are likely familiar with the R and R^2 values from statistics, but let's just recap here.

> **Definition 4.33** (Correlation Coefficient, R). The **correlation coefficient**, R, is a measure of the quality of a linear fit.
>
> - If $R = +1$ then the data represent a perfect linear fit with a positive slope.
>
> - If $R = -1$ then the data represent a perfect linear fit with a negative slope.
>
> - If $R = 0$ then there is no linear relationship between the two variables.

Definition 4.34 (Coefficient of Determination, R^2). The **coefficient of determination**, R^2, is the proportion of variance in the dependent (y) variable that is explained by the independent (x) variable.

- If $R^2 = 1$ then 100% of the variance in y is explained by x and the fit is perfect.

- If $R^2 = 0$ then 0% of the variance in y is explain by x and there is no correlation between the two variables.

In the problems that we've studied thus far we have calculated the sum of the squares of the residuals as a measure of how well our function fits the data. The trouble with the sum of the squares of the residuals is that it is context dependent. Hence, there is no way to tell at the outset what a *good* sum of squares of residuals is.

Theorem 4.35. To calculate the R^2 value we can use the following equations where y_j is the jth data point, $\bar{y}$ is the mean y-value in the data, and $\hat{y}_j$ is the jth predicted output from our model.

$$\text{Total Sum of Squares: } TSS = \sum_j \left(y_j - \bar{y}\right)^2 \tag{4.1}$$

$$\text{Residual Sum of Squares: } RSS = \sum_j \left(y_j - \hat{y}_j\right)^2 \tag{4.2}$$

$$\text{Coeff. of Determination: } R^2 = 1 - \frac{RSS}{TSS} \tag{4.3}$$

Problem 4.36. Compute the R^2 value for each of the data sets from Problem 4.32.   ▲

Problem 4.37. Use equation (4.3) to explain why $R^2 = 1$ implies a perfect fit.   ▲

There are MANY statistics packages out there that will do least squares regression for us. Python has the `polyfit` command, R has the `lm` command, and Excel has the data analysis toolpack. Behind the scenes in all of these is actually linear algebra. It is informative pedagogically to do the least squares problems using Excel a few times (as we did in Problem 4.32), but in reality there is some beautifully simple linear algebra behind the scenes.

Problem 4.38. Now we'll use Linear Algebra to complete the same types of problems. Set up a Python script the reads the linear data from the Excel sheet. We are assuming that the data are linear so for each x-value x_j we assume that the equation

$$\hat{y}_j = \beta_0 + \beta_1 x_j \approx y_j.$$

The values of β_0 and β_1 are the intercept and slope that best predict the y values from the data.

The equation $\beta_0 + \beta_1 x_j \approx y_j$ gives rise to a system of linear equations

$$
\begin{pmatrix}
1 & x_1 \\
1 & x_2 \\
1 & x_3 \\
\vdots & \vdots \\
1 & x_n
\end{pmatrix}
\begin{pmatrix}
\beta_0 \\
\beta_1
\end{pmatrix}
\approx
\begin{pmatrix}
y_1 \\
y_2 \\
y_3 \\
\vdots \\
y_n
\end{pmatrix}
$$

when we consider all n of the data points. We'll call the left-hand matrix A and the right-hand vector $\mathbf{y}$.

1. For each data set get A and $\mathbf{y}$ into Python so we can use them.

2. This is not a square system. In fact, it is *overdetermined* (more rows than columns). Consider that the column space of A ($Col(A)$) is a subspace of $\mathbb{R}^n$ and that $\mathbf{y} \in \mathbb{R}^n$ is likely not in the column space of A. If we project $\mathbf{y}$ onto $Col(A)$ we should be able to find the best approximation of $\mathbf{y}$ that lies in $Col(A)$.

   Projections onto a subspace can be achieved with matrix multiplication, but which matrix shall we multiply by . . .

   $$(??)A\begin{pmatrix}\beta_0 \\ \beta_1\end{pmatrix} = (??)\mathbf{y}.$$

   Once you are satisfied that the right-hand side is a projection of $\mathbf{y}$ onto $Col(A)$ you have formed the **normal equations**. If you've done everthing right then you also now have a square system (2×2 in the case of the linear data).

3. Now that this is a square matrix you can solve for β_0 and β_1 using your *LU* or *QR* code.

4. Now that you have β_0 and β_1 you can form the linear approximation. Plot the approximation along with your data. Also write code to plot the residuals (subplot would be great here).

5. Repeat the process for the quadratic. For the quadratic data you are assuming that

   $$\beta_0 + \beta_1 x_j + \beta_2 x_j^2 \approx y_j.$$

▲

Definition 4.39 (The Normal Equations)**.** Let $A \in \mathbb{R}^{n \times m}$, $\beta \in \mathbb{R}^m$, and $\mathbf{y} \in \mathbb{R}^n$ where $n \gg m$. The system of equations

$$A\beta = \mathbf{y}$$

is over determine since there are more equations than unknowns. Multiply both sides of the equation by A^T yields the **normal equations**

$$A^T A\beta = A^T \mathbf{y}.$$

> We note here that $A^T A \in \mathbb{R}^{m \times m}$ is square and much smaller than A. The right-hand side of the normal equations is the projection of **y** onto the columns space of A.

You should take careful note of something here. The process that we just formulated is called *linear regression* even though we were fitting a quadratic function to data. This may, at first, seem like an unfortunate or inappropriate naming convention, but stop and think more carefully about what we did. …Good. Now that you've though about it, I'll give you my take.

For the quadratic data fit we are trying to create the function $\hat{y}_j = \beta_0 + \beta_1 x_j + \beta_2 x_j^2$. Notice that this function is really just a multiple regression that is, indeed, linear in the coefficients β_0, β_1, and β_2. The term *linear regression* does <u>not</u> have anything to do fitting lines to data – a common misconception. Instead, it pertains to the relationship between the coefficients. Since the coefficients of a polynomial function are linearly related to each other, we can use linear regression to fit polynomial models for any order polynomial.

The reader should be further warned that polynomial regression comes with some down sides. If the order of the polynomial grows the matrix $A^T A$ arising from the normal equations formulation gets closer and closer to being singular. That is to say that as the order of the polynomial increases it becomes less and less desirable to solve the problem through matrix inversion (or Gaussian Elimination).

Problem 4.40. In this problem we are going to follow up on the claim in the previous paragraph:

> *When doing polynomial regression via the normal equations, as the order of the polynomial grows the matrix $A^T A$ gets close and closer to being singular.*

(a) Create a data set from a third or fourth order polynomial. Then introduce some random noise onto your data set.

(b) Write a loop in Python that fits polynomials of increasing order to the data set (starting with linear and increasing by 1 each time). Create two plots. In the first plot put the order of the polynomial on the horizontal axis and the R^2 value on the vertical axis. On the second plot put the order of the polynomial on the horizontal axis and the ratio $|\lambda_{max}|/|\lambda_{min}|$ on the vertical axis, where λ_{max} and λ_{min} are the maximum and minimum eigenvalues respectively. You'll probably want to use a `semilogy` scale for the second plot.

(c) What do you notice about the R^2 value in the first plot as you increase the order of the polynomial? Is this surprising? Discuss.

(d) What does the right-hand plot tell you about the matrix $A^T A$?

(e) Why is the ratio $\lambda_{max}/\lambda_{min}$ a measure of how close to singular (non-invertible) the matrix $A^T A$ is?

▲

Problem 4.41. Explain what the previous problem tells you about fitting polynomials to data.   ▲

4.5   The Eigenvalue-Eigenvector Problem

Recall that the eigenvectors, $\mathbf{x}$, and the eigenvalues, λ of a square matrix satisfy the equation $A\mathbf{x} = \lambda\mathbf{x}$. Geometrically, the eign-problem is the task of finding the special vectors $\mathbf{x}$ such that multiplication by the matrix A only produces a scalar multiple of $\mathbf{x}$. Thinking about matrix multiplication, this is rather peculiar since matrix-vector multiplication usually results in a scaling and a rotation of the vector. Therefore, in some sense the eigenvectors are the only special vectors which avoid geometric rotation under matrix multiplication. For a graphical exploration of this idea see:
https://www.geogebra.org/m/JP2XZpzV.

Recall that to solve the eigen-problem for a square matrix A we complete the following steps:

1. First rearrange the definition of the eigenvalue-eigenvector pair to

$$(A\mathbf{x} - \lambda\mathbf{x}) = \mathbf{0}.$$

2. Next, factor the $\mathbf{x}$ on the right to get

$$(A - \lambda I)\mathbf{x} = \mathbf{0}.$$

3. Now observe that since $\mathbf{x} \neq 0$ the matrix $A - \lambda I$ must NOT have an inverse. Therefore,

$$\det(A - \lambda I) = 0.$$

4. Solve the equation $\det(A - \lambda I) = 0$ for all of the values of λ.

5. For each λ, find a solution to the equation $(A - \lambda I)\mathbf{x} = \mathbf{0}$. Note that there will be infinitely many solutions so you will need to make wise choices for the free variables.

Problem 4.42. Find the eigenvalues and eigenvectors of $A = \begin{pmatrix} 1 & 2 \\ 4 & 3 \end{pmatrix}$ by hand.   ▲

Problem 4.43. In the matrix $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ one of the eigenvalues is $\lambda_1 = 0$.

1. What does that tell us about the matrix A?

2. What is the eigenvector $\mathbf{v}_1$ associated with $\lambda_1 = 0$?

3. What is the null space of the matrix A?

▲

Problem 4.44. Find matrices P and D such that $A = \begin{pmatrix} 1 & 2 \\ 4 & 3 \end{pmatrix}$ can be written as $A = PDP^{-1}$ where P is a dense 2×2 matrix and D is a diagonal matrix. Once you have this factorization of A, use it to determine A^{10}.   ▲

Problem 4.45. Let A be an $n \times n$ matrix with n distinct eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ and let $\mathbf{x} \in \mathbb{R}^n$ be a vector such that $\mathbf{x} = \sum_{j=1}^n c_j \mathbf{v}_j$. Find expressions for $A\mathbf{x}, A^2\mathbf{x}, A^3\mathbf{x}, \ldots$ ▲

Problem 4.46. In this problem we first describes the mathematical idea for the **power method** for computing the largest eigenvalue / eigenvector pair. Then we write an algorithm for find the largest eigen-pair numerically.

1. Assume that A has n linearly independent eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ and choose $\mathbf{x} = \sum_{j=1}^n c_j \mathbf{v}_j$. From the previous problem,

$$A^k\mathbf{x} = \underline{\hspace{5cm}}$$

2. Factor the right-hand side so that

$$A^k\mathbf{x} = \lambda_1^k \left(c_1\mathbf{v}_1 + c_2\left(\frac{\lambda_2}{\lambda_1}\right)^k \mathbf{v}_2 + c_3\left(\frac{\lambda_3}{\lambda_1}\right)^k \mathbf{v}_3 + \cdots + c_n\left(\frac{\lambda_n}{\lambda_1}\right)^k \mathbf{v}_n \right)$$

3. If $\lambda_1 > \lambda_2 \geq \lambda_3 \geq \cdots \geq \lambda_n$ then what happens to each of the $(\lambda_j/\lambda_1)^k$ terms as $k \to \infty$? Using this answer, what is $\lim_{k \to \infty} A^k\mathbf{x}$?

▲

Technique 4.47 (The Power Method Algorithm). This algorithm 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 $\mathbf{x}$, set $\mathbf{v}^{(1)} = \mathbf{x}/\|\mathbf{x}\|$. (Here the superscript indiates the iteration number)

Step #2: For $k = 2, 3, \ldots$

    Step #2a: Compute $\tilde{\mathbf{v}}^{(k)} = A\mathbf{v}^{(k-1)}$ (this gives a non-normalized version of the next estimate of the dominant eigenvector.)

    Step #2b: Set $\lambda^{(k)} = \left\langle \tilde{\mathbf{v}}^{(k)}, \mathbf{v}^{(k-1)} \right\rangle$. (this gives an approximation of the eigenvalue since if $\mathbf{v}^{(k-1)}$ was the actual eigenvector we would have $\lambda = \left\langle A\mathbf{v}^{(k-1)}, \mathbf{v}^{(k-1)} \right\rangle$)

    Step #2c: Normalize $\tilde{\mathbf{v}}^{(k)}$ by computing $\mathbf{v}^{(k)} = \tilde{\mathbf{v}}^{(k)}/\|\tilde{\mathbf{v}}^{(k)}\|$. (This guarantees that you will be sending a unit vector into the next iteration of the loop)

Problem 4.48. Write a Python function to implement the power method for finding the eigenvalue of largest absolute value and the associated eigenvector. Test it on a matrix where you know the eigenvalue of interest. ▲

4.6 The Singular Value Decomposition

Our overarching goal of this section is to discuss an analogue to the eigenvalue-eigenvector problem for non-square matrices. That is, we would like to take a matrix A that is $m \times n$ and find vectors are values that behave similarly to how eigenvectors and eigenvalues behave for square matrices. The key to this discussion is the matrix $A^T A$, so let's start there.

Problem 4.49. Let A be an $m \times n$ matrix. What is the size of $A^T A$? Prove that $A^T A$ must be a symmetric matrix (a matrix B is symmetric if $B_{ij} = B_{ji}$). Finally, what bearing does the next Theorem have on the matrix $A^T A$? ▲

Theorem 4.50. An $n \times n$ matrix A has n orthogonal eigenvectors if and only if A is a symmetric matrix.

Definition 4.51. The **singular values** of an $m \times n$ matrix A are the square roots of the eigenvalues of $A^T A$. They are typically denoted as $\sigma_1, \sigma_2, \ldots, \sigma_n$ where $\sigma_j = \sqrt{\lambda_j}$ and λ_j is an eigenvalue of $A^T A$.

Definition 4.52. The **singular value decomposition** of an $m \times n$ matrix A with rank r is a factorization of A into the product of three matrices, U, Σ, and V, such that

$$A = U\Sigma V^T.$$

In the singular value decomposition, U ($m \times m$) and V ($n \times n$) have orthogonal columns and Σ ($m \times n$) is a block diagonal matrix

$$\Sigma = \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix}$$

where D is an $r \times r$ diagonal matrix containing the r singular values of A in rank order (largest to smallest).
   To build the singular value decomposition:

1. Form $A^T A$ and find the eigenvalues and eigenvectors (guaranteed to exist by Theorem 4.50).

2. Form Σ

3. The columns of V are the eigenvectors of $A^T A$.

4. The columns of U are the normalized vectors obtained by

$$\mathbf{u}_1 = \frac{1}{\sigma_1} A\mathbf{v}_1 \,, \mathbf{u}_2 = \frac{1}{\sigma_2} A\mathbf{v}_2 \,, \ldots, \mathbf{u}_m = \frac{1}{\sigma_m} A\mathbf{v}_m$$

Problem 4.53. Use Python to find the singular value decomposition of

$$A = \begin{pmatrix} 4 & 11 & 14 \\ 8 & 7 & -2 \end{pmatrix}$$

Continue altering the coding sections starting here
Some practical MATLAB tips follow:

1. Define A

2. Define the sizes: `m=size(A,1); n=size(A,2)`

3. Find the rank of A: `r = rank(A);`

4. Define the matrices `Sigma` and `U` to be zero matrices with the right size.

5. Have MATLAB calculate the eigenvectors and eigenvalues of $A^T A$:
   `[vectors,values]=eig(A'A,'vector');`
   The `'vector'` command spits out the eigenvalues as a vector instead of a diagonal matrix. This will be helpful in the next step.

6. Have MATLAB sort the eigenvalues and strip any negative *approximate zero* eigenvalues that arise from numerical approximation of zero.
   `values = abs(values);`
   `[values,indices] = sort(values,'descend')`

7. Sort the columns of V using the indices coming out of the sort command:
   `V = vectors(:,indices);`

8. Build the singular values from the eigenvalues of $A^T A$ (remember the square root!):
   `singularvalues=...`

9. Build non-zero diagonal entries of the Σ matrix with a loop. Also build a temporary matrix B the same size as Σ but with the diagonal entries $1/\sigma_j$. We'll need B in the next step.

```
1  B=zeros(size(Sigma));
2  for j=1:r}
3  Sigma(j,j) = ...
4  B(j,j) = ...
5  end
```

10. Observe that since V has orthonormal columns we can write $AV = U\Sigma$. Now, Σ is not square, but we know that it has diagonal entries only so we have a *pseudo-inverse* B^T already built. Hence, $U = AVB^T$. Build U.

11. Check that $A = U\Sigma V^T$

▲

Problem 4.54. Create a Python function that accepts a matrix A and outputs the three matrices for the singular value decomposition. Test your function on a large random rectangular matrix.

```
1  import numpy as np
2  A = np.random.rand(500,300)
3  U, S, V = MySVD(A)
4  error = np.linalg.norm(A − U*S*np.transpose(V))
5  print(error)
```

▲

4.7   Exercises

4.7.1   Algorithm Summaries

Problem 4.55. Explain in clear language how to efficiently solve an upper triangular system of linear equations.   ▲

Problem 4.56. Explain in clear language how to efficiently solve a lower triangular system of linear equations.   ▲

Problem 4.57. Explain in clear language how to solve the equation $A\mathbf{x} = \mathbf{b}$ using an LU decomposition.   ▲

Problem 4.58. Explain in clear language how to solve an overdetermined system of linear equations (more equations than unknowns) numerically.   ▲

Problem 4.59. Explain in clear language the algorithm for finding the columns of the Q matrix in the QR factorization. Give all of the mathematical details.   ▲

Problem 4.60. Explain in clear language how to find the upper triangular matrix R in the QR factorization. Give all of the mathematical details.   ▲

Problem 4.61. Explain in clear language how to solve the equation $A\mathbf{x} = \mathbf{b}$ using a QR decomposition.   ▲

Problem 4.62. 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.
▲

Problem 4.63. Explain in clear language how to find the Singular Value Decomposition of a matrix.   ▲

4.7.2   Applying What You've Learned

Problem 4.64. 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.
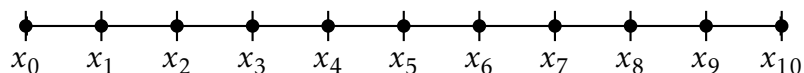
How many unknowns are there in this picture?

(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.

▲

Problem 4.65. Write code to solve the following systems of equations via both LU and QR decompositions.

(a)

$$
\begin{aligned}
x + 2y + 3z &= 4 \\
2x + 4y + 3z &= 5 \\
x + y &= 4
\end{aligned}
$$

(b)

$$
\begin{aligned}
2y + 3z &= 4 \\
2x + 3z &= 5 \\
y &= 4
\end{aligned}
$$

(b)

$$
\begin{aligned}
2y + 3z &= 4 \\
2x + 4y + 3z &= 5 \\
x + y &= 4
\end{aligned}
$$

▲

Problem 4.66. Find a least squares solution to the equation $A\mathbf{x} = \mathbf{b}$ in two different ways with

$$
A = \begin{pmatrix} 1 & 3 & 5 \\ 4 & -2 & 6 \\ 4 & 7 & 8 \\ 3 & 7 & 19 \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} 5 \\ 2 \\ -2 \\ 8 \end{pmatrix}.
$$

▲

Problem 4.67. 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);
```

Build the t vector and the y vector (these are your data). We need to set up the least squares problems $A\mathbf{x} = \mathbf{b}$ by setting up the matrix A as we did in the other least squares curve fitting problems and by setting up the $\mathbf{b}$ vector using the y data you just built.

(a) Solve the normal equations $A^T A\mathbf{x} = A^T\mathbf{b}$ using your LU code.

(b) Solve the system $A\mathbf{x} = \mathbf{b}$ by first transforming A to $A = QR$ and then solving $R\mathbf{x} = Q^T\mathbf{b}$.

(c) Use Python to find the sum of the square errors between the polynomial approximation and the function $f(t) = \cos(4t)$ for both the QR and the LU approaches.

(d) Build Python code that does parts (a) - (c) several hundred times and complies results comparing which method gives the better approximation (smaller sum of square error).

▲

Problem 4.68. Find the largest eigenvalue of the matrix A WITHOUT using the built in "`eig`" command in Python .

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 0 & 1 & 2 \\ 3 & 4 & 5 & 6 \end{pmatrix}$$

▲

Problem 4.69. Find a least square cubic function that best fits the following data. Solve this problem with Excel and with Python using the normal equations.

x	y
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

▲

> **Theorem 4.70** (Eigen-Structure of Symmetric Matrices). 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.

Problem 4.71. 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 4.70 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.

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 Python function `MyPower2` to find the second largest eigenvalue and eigenvector pair by putting the deflation method into practice. Test your code on a symmetric matrix A and compare against Python 's `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 Python 's eigenvector.

To guarantee that you start with a symmetric matrix you can use the following code.

```
1  import numpy as np
2  N = 40
3  A = np.random.rand(N,N)
4  A = np.matrix(A)
5  A = np.transpose(A) * A
```

▲

4.8   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.

4.8.1   Applications of the Singular Value Decomposition

The singular value decomposition (SVD) is a matrix factorization that can be thought of as the generalization of the eigenvalue problem for non-square matrices. We wish to take a matrix A that has size $m \times n$ and factor it into three matrices, U, Σ, and V such that

$$A = U\Sigma V^T$$

where U is an $m \times m$ orthonormal matrix, Σ is an $m \times n$ rectangular diagonal matrix, and V is and $n \times n$ orthonormal matrix. Recall that an orthonormal matrix is a matrix where the transpose is the inverse. That is, $UU^T = I_{m \times m}$ and $VV^T = I_{n \times n}$. The columns of U are called the left singular vectors of A and are the eigenvectors of the matrix AA^T. The columns of V are called the right singular vectors of A and are the eigenvectors of the matrix $A^T A$. The diagonal entries of Σ are the square roots of the non-zero eigenvalues of both AA^T and $A^T A$.

It is probably best to have a look at an example (modified from [*]). Consider the matrix

$$A = \begin{pmatrix} 3 & 2 & 1 \\ 2 & 3 & -2 \end{pmatrix}.$$

We would like to factor A into $A = U\Sigma V^T$. First we find the eigenvalues and eigenvectors of AA^T. It is left to the reader to verify[†] that the eigenvalues are $\lambda_1 = 25$ and $\lambda_2 = 9$. Hence, the singular values are $\sigma_1 = 5$ and $\sigma_2 = 3$. Next we find the right singular vectors (the columns of V) by finding an orthonormal set of eigenvectors of $A^T A$. This too is left to the reader to verify that the matrix V is

$$V = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{18} & 2/3 \\ 1/\sqrt{2} & -1/\sqrt{18} & -2/3 \\ 0 & 4/\sqrt{18} & -1/3 \end{pmatrix}.$$

The reader should also observe that the columns of V are all orthogonal to each other as well as being unit vectors.

We now know that

$$A = U\Sigma V^T = U\begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix}\begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 1/\sqrt{18} & -1/\sqrt{18} & 4/\sqrt{18} \\ 2/3 & -2/3 & -1/3 \end{pmatrix}$$

---

[*]http://www.d.umn.edu/ mhampton/m4326svd_example.pdf
[†]That means that you should stop what you're doing and actually verify this result!

Since V is an orthogonal matrix and the singular values σ_1 and σ_2 are non-zero we can see that

$$U = AV\Sigma^\dagger$$

where

$$\Sigma^\dagger = \begin{pmatrix} 1/5 & 0 \\ 0 & 1/3 \\ 0 & 0 \end{pmatrix}.$$

After a bit of computation we finally find that

$$U = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}$$

and hence

$$A = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix} \begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 1/\sqrt{18} & -1/\sqrt{18} & 4/\sqrt{18} \\ 2/3 & -2/3 & -1/3 \end{pmatrix}$$

We leave it to the reader to check that this is correct.

Obviously this is not a computation that you typically do by hand on larger matrices. You should have SVD code handy that does this entire algorithm. Python also has a built-in svd code that is optimized to run faster than the one that we built in class. You are welcome to use the built-in svd Python routine in this project.

Problem Statement

There are many applications of the singular value decomposition. In fact, the SVD is considered one of the most useful and widely used matrix decompositions in the linear algebra arsenal. Your job for this project will be to explore two of those applications. Below you will find a brief list of some of the primary uses for the singular value decomposition. Some of them are rather intricate and some are fairly straight forward to implement, but I leave it to you to choose applications that are of interest to you.

Your Tasks

You and your partner will choose two applications and do the following:

(a) implement non-trivial examples of the application

(b) write a brief technical report detailing how each application works (including all of the mathematical details), and

(c) write a 1 paragraph non-technical summary describing each application and how it works. The audience for this non-technical summary is any reasonably educated, but not necessarily mathematically sophisticated, adult.

Note: All code for this project must be uniquely yours or your partners.

List of Applications

Each of the following applications of the singular value decomposition is followed by a brief description (paraphrased from various web sources linked in the footnotes). Use this to guide your choice.

Principal Component Analysis: (PCA) is a statistical tool that allows you to convert a set of possibly correlated data observations into uncorrelated components. This is a method of data reduction that, in some sense, reduces how much data is actually influential in your data set. PCA takes a potentially large data set and reduces it down to its *principal components* that are the most influential on the data. PCA itself has many applications ranging from statistics to signal processing. If you choose this application you will need to implement PCA on a non-trivial data set of your choosing. There are several sample data sets on the web but the ones that are used for explanation are often quite trivial in size and scope. [‡] Absolutely do not just make up your own data set.

Low Rank Image Approximations: There are literally hundreds of ways to trim information from images while retaining the most important parts of the image. One of the simplest is to apply the singular value decomposition to the rectangular image (either in grayscale or separately for each RBG matrix). Then you can zero out the lower value singular values and vectors and then rebuild the image. The low rank approximation (really just another type of data reduction) does obviously lead to some loss of information but up to a certain point this loss of information is un-noticeable to the human eye. If you choose this application you need to supply your own images as well as give several examples of how low rank approximation changes the image. You should also explore how low rank approximations can be used as a method of image compression. and give some sort of mathematical measure defining how much data reduction was performed along with the re-generated images so we can see the reduction.[§]

Recommender Systems: You run in the *recommender systems* all the time. Amazon seems to know what other books you might like, NetFlix seems to know what other movies you might like, and the list goes on. One way to perform the matching behind the scenes in a recommender system is to use the SVD along with data reduction techniques (just like with PCA and image compression). If you choose this application you need to provide a non-trivial example of a recommender system preferably of real data released from one of the large common sites (NetFlix, Amazon, etc.).[¶] [‖]

The Kabsh Algorithm: This is an algorithm for calculating the optimal rotation matrix that minimizes the error between two sets of points. This is a useful method in

---

[‡]PCA Wiki page: en.wikipedia.org/wiki/Principal_component_analysis
[§]A Sample of the Image Compression Technique: math.arizona.edu/~brio/VIGRE/ThursdayTalk.pdf
[¶]The idea of SVD for Recommender Systems: http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/svd.
[‖]Paper about Recommender Systems: http://robotics.stanford.edu/ ronnyk/WE-BKDD2000/papers/sarwar.pdf

comparing molecular structures (including proteins) but is not just used in mathematical biology. If you choose this application then you need to provide several non-trivial examples of how the method works and what it is that you are finding. **

Choose Your Own Adventure: If you happen upon an interesting and non-trivial application of the singular value decomposition then you are welcome to use this as one of your two applications. Just be sure that you can implement it and that you can fully explain what you're doing. A decent place to start is the applications section of the SVD Wiki page. ††

** Wiki page for Kabsh Algorithm: https://en.wikipedia.org/wiki/Kabsch_algorithm
†† SVD Wiki: https://en.wikipedia.org/wiki/Singular_value_decomposition

4.8.2   The Google Page Rank Algorithm

In this project you will discover how the PageRank 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 *PageRank 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 *inlinks*, and the ones pointing out of a page are called *outlinks*. 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 4.1 (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 *inlink* and an arrow pointing out of a node is an *outlink*. In Figure 4.1, node 3 has three outlinks (to nodes 1, 2, and 5) and 1 inlink (from node 1).
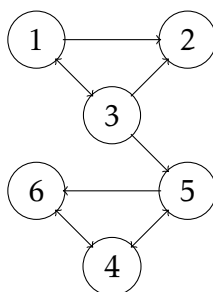
Figure 4.1. Sample graph of a web with six pages.

We will first define some notation in the PageRank algorithm:

- $|P_i|$ is the number of outlinks 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.

- $\mathbf{x}$ is a vector that contains all of the PageRanks for the individual pages.

The PageRank 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 4.1 we would take $\mathbf{x}_0$ to be

$$\mathbf{x}_0 = \begin{pmatrix} 1/6 \\ 1/6 \\ 1/6 \\ 1/6 \\ 1/6 \\ 1/6 \end{pmatrix}.$$

2. Build the hyperlink matrix.
   As an example we'll consider node 3 in Figure 4.1. There are three outlinks 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 $\mathbf{x}_{n+1} = H\mathbf{x}_n$ 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	$\mathbf{x}_0$
1	$\mathbf{x}_1 = H\mathbf{x}_0$
2	$\mathbf{x}_2 = H\mathbf{x}_1 = H^2\mathbf{x}_0$
3	$\mathbf{x}_3 = H\mathbf{x}_2 = H^3\mathbf{x}_0$
4	$\mathbf{x}_4 = H\mathbf{x}_3 = H^4\mathbf{x}_0$
$\vdots$	$\vdots$
k	$\mathbf{x}_k = H^k\mathbf{x}_0$

4. When a steady state is reached we sort the resulting vector $\mathbf{x}_k$ 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 PageRank process.

Theorem 4.72. If A is an $n{\times}n$ matrix with n linearly independent eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3,$ $\ldots, \mathbf{v}_n$ and associated eigenvalues $\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_n$ then for any initial vector $\mathbf{x} \in \mathbb{R}^n$ we can write $A^k \mathbf{x}$ as

$$A^k \mathbf{x} = c_1 \lambda_1^k \mathbf{v}_1 + c_2 \lambda_2^k \mathbf{v}_2 + c_3 \lambda_3^k \mathbf{v}_3 + \cdots c_n \lambda_n^k \mathbf{v}_n$$

where $c_1, c_2, c_3, \ldots, c_n$ are the constants found by expressing $\mathbf{x}$ as a linear combination of the eigenvectors.
Note: We can assume that the eigenvalues are ordered such that $\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq \cdots \geq \lambda_n$.

Proof. (Prove the preceding theorem) $\square$

Definition 4.73. A **probability vector** is a vector with entries on the interval $[0,1]$ that add up to 1.

Definition 4.74. A **stochastic matrix** is a square matrix whose columns are probability vectors.

Theorem 4.75. 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 always 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 out loud.

Problem 4.76. Finish writing the hyperlink matrix H from Figure 4.1.   ▲

Problem 4.77. Write Python code to implement the iterative process defined previously. Make a plot that shows how the rank evolves over the iterations. ▲

Problem 4.78. What must be true about a collection of n pages such that an $n \times n$ hyperlink matrix H is a stochastic matrix. ▲

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 4.79. If A is an $n \times n$ stochastic matrix and $\mathbf{x}_0$ is some initial vector for the difference equation $\mathbf{x}_{n+1} = A\mathbf{x}_n$, then the steady state vector is

$$\mathbf{x}_{equilib} = \lim_{k \to \infty} A^k \mathbf{x}_0 = \underline{\hspace{2cm}}.$$

Proof. First note that A is an $n \times n$ stochastic matrix so from Theorem 4.75 we know that there are n linearly independent eigenvectors. We can then substitute the eigenvalues from Theorem 4.75 in Theorem 4.72. Noting that if $0 < \lambda_j < 1$ we have $\lim_{k \to \infty} \lambda_j^k = 0$ the result follows immediately. ☐

Problem 4.80. Discuss how Theorem 4.79 greatly simplifies the PageRank iterative process described previously. In other words: there is no reason to iterate at all. Instead, just find \underline{\hspace{2cm}}. ▲

Problem 4.81.

Now use the previous two problems to find the resulting PageRank vector from the web in Figure 4.1? Be sure to rank the pages in order of importance. Compare your answer to the one that you got in problem 2. ▲

Problem 4.82. Consider the web in Figure 4.2.

(a) Write the H matrix and find the initial state $\mathbf{x}_0$,

(b) Find steady state PageRank vector using the two different methods described: one using the iterative difference equation and the other using Theorem 4.79 and the dominant eigenvector.

(c) Rank the pages in order of importance.

▲

Problem 4.83. 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 4.2 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
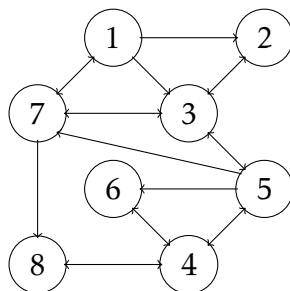
Figure 4.2. Graph of a web with eight pages.

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 4.2. Compare your ranking to the non-random surfer results from the previous problem. ▲

Chapter 5

Numerical Ordinary Differential Equations

"The mathematical discipline of differential equations furnishes the explanation of all those elementary manifestations of nature which involve time."
–Norwegian Mathematician Sophus Lie

5.1   Recalling the Basics of Ordinary Differential Equations

Problem 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 be as a function of time?

(c) A pollutant has entered the 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?

▲

All of the functions that you played with in the previous problem can easily be modeled with differential equations. There are few tools in the mathematician's arsenal that are more useful than modeling with differential equations. We will focus the next two

chapters of the book on these types of equations and their numerical solutions, but first: what is a differential equation?

> **Definition 5.2.** A **differential equation** is an equation that relates the derivative (or derivatives) of an unknown function to itself.

> **Definition 5.3.** A **solution to a differential equation** is a function which, when substituted into the differential equation, creates a true statement.

> **Definition 5.4.** A **numerical solution to a differential equation** is a list of ordered pairs that gives a point-wise approximation to the actual solution.

These ideas should be familiar to you from previous classes, but just in case, 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 – you're done if you have an analytic solution. The fact of the matter is, however, that the techniques for finding analytic solutions to differential equations are rather limiting, and when the differential equations get complicated we will only have numerical approximations to lean back on.

Let's get started with some review.

Problem 5.5. Identify which of the following problems are *differential* equations and which are *algebraic* equations.

$$x^2 + 5x = 7x^3 - 2$$
$$x'' + 5x = 7x''' - 2$$
$$y' + 5 = -3y$$
$$y''y'y = 8$$
$$y^2 \cdot y = 8$$

(Do not try to solve any of these equations)   ▲

Problem 5.6. Consider the differential equation $y' = 3y$ with an initial condition $y(0) = 4$. Which of the following functions is a solution to this differential equation, and what is the value of the constant in the function?

(a) $y(t) = C\sin(3t)$

(b) $y(t) = Ce^{3t}$

(c) $y(t) = Ct^3$

(d) $y(t) = t^3 + C$

(e) $y(t) = e^{3t} + C$

(f) $y(t) = \sin(3t) + C$

▲

Problem 5.7. Consider the differential equation $y' = 3y + t$ with an initial condition $y(0) = 4$. Which of the following functions is a solution to this differential equation, and what are the values of the constants?

(a) $y(t) = C_0 \sin(\sqrt{3}t) + C_1 t + C_2$

(b) $y(t) = C_0 e^{3t} + C_1 t + C_2$

(c) $y(t) = C_0 t^3 + C_1 t + C_2$

(d) $y(t) = C_3 t^3 + C_2 t^2 + C_1 t + C_0$

(e) $y(t) = e^{3t} + C_1 t + C_2$

(f) $y(t) = \sin(3t) + C_1 t + C_2$

▲

Problem 5.8. 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 $x(0) = 3$. ▲

Technique 5.9 (Separation of Variables). To solve a differential equation of the form

$$\frac{dy}{dt} = f(y)g(t)$$

we can separate the variables and rewrite the problem as

$$\int \frac{dy}{f(y)} = \int g(t)dt.$$

Integrating both sides and solving for $y(t)$ gives the solution.

Problem 5.10. Use separation of variables to solve the differential equation

$$\frac{dy}{dt} = y\sin(t)$$

with the initial condition $y(0) = 1$. ▲

Problem 5.11. Solve the differential equation $y' = -2y + 12$ with $y(0) = 2$ using separation of variables.   ▲

Problem 5.12. Consider the differential equation

$$\frac{dy}{dt} = -\frac{1}{4}y + 4$$

with the initial condition $y(0) = 7$.

(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.

5.2 Euler's Method

Problem 5.13. Consider the differential equation $y' = -0.5y$ with the initial condition $y(0) = 6$.

(a) Since we know that $y(0) = 6$ and we know that $y'(0) = -0.5 \cdot y(0)$ we can approximate the value of y at some future time step. Let's go 1 unit forward in time. That is, approximate $y(1)$ knowing that $y(0) = 6$ and $y'(0) = -3$.
Hint: We know a y-value, a slope, and the size of the step that we would like to move in the t direction . . .

$$y(1) \approx \underline{\hspace{3cm}}$$

(b) Use your answer from part (a) for time $t = 1$ to approximate the y 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 y 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 y values at the appropriate times.

t	0	1	2	3	4	5	6	7	8	9	10
Approximation of y	6										
Exact value of y	6										

(c) The "approximations of y" that you found in part (b) are a **numerical approximation** of the solution to the differential equation. You should notice that your numerical solution is pretty far off from the actual solution for most values of t. Why? What could be the sources of this error and how could we fix it? Once you have an idea of how to fix it, put your idea into action and devise some measurement of error to analyze your results.

(d) Draw a clear picture of what this method is doing in order to approximate the slope at each individual step.

▲

The notion of approximating solutions to differential equations is simple: make a discrete approximation to the derivative and step forward through time as a difference equation. The fun part is making the approximation to the derivative(s). There are many methods for approximating derivatives, and that is exactly where we'll start.

Technique 5.14 (Euler's Method). You're probably already familiar with Euler's method for approximating the solution to a differential equation. We want to approximate a solution to $y'(t) = f(t, y(t))$. Recall from Problem 3.8 that

$$y'(t) = \frac{y(t + h) - y(t)}{h} + \mathcal{O}(h)$$

so the differential equation $y'(t) = f(y(t), t)$ becomes

$$\frac{y(t+h) - y(t)}{h} \approx f(y(t), t).$$

Rewriting as a difference equation, letting $y_{n+1} = y(t_n + h)$ and $y_n = y(t_n)$, we get

$$y_{n+1} = y_n + hf(y_n, t_n) \tag{5.1}$$

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 5.1 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 $h \to 0$ theoretically gives the exact solution at the tradeoff of needing infinite computational resources.
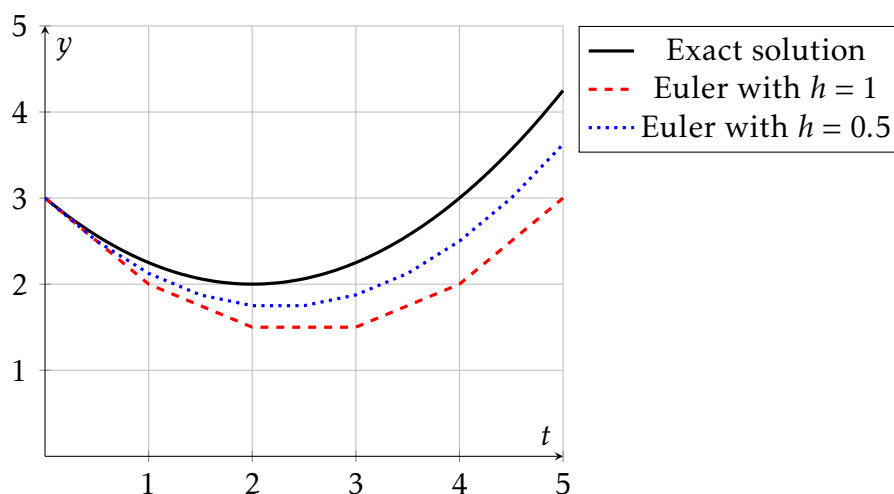
Figure 5.1. A depiction of Euler's method with step size $h = 1$ (red) and $h = 0.5$ (blue).

Problem 5.15. Write code to implement Euler's method for initial value problems. Your Python function should accept as input: $f(y, t)$, `tmin`, `tmax`, the number of grid points (the value of $h = \Delta t$ should be calculated within your code), and an initial condition. The output should be vectors for t and y.

`def MyEuler1D(f,tmin,tmax,num_pts,IC):`

Test your code on a first order differential equation where you know the answer and then test your code on the differential equation

$$y' = -\frac{1}{3}y + \sin(t) \quad \text{where} \quad y(0) = 1.$$

▲

Problem 5.16. The differential equation $y' = -\frac{1}{3}y + \sin(t)$ with $y(0) = 1$ has an analytic solution

$$y(t) = \frac{1}{10}\left(19e^{-t/3} + 3\sin(t) - 9\cos(t)\right).$$

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?

◀

   We can solve differential equations with multiple unknowns. For example: the populations of two competing species modeled over time. These problems are typically very challenging to solve by hand, but numerically the idea is exactly the same as with one-dimensional Euler's method. The problem is to simultaneously solve the equations

$$x' = f(x,y,t)$$
$$y' = g(x,y,t)$$

where $x(t)$ and $y(t)$ are the functions we want to know, and f and g are functions that model the system of interest. Consider the following example of a system of differential equations

$$x' = 0.5xy - y$$
$$y' = 2y - 0.1xy^2 - t.$$

In this example, $f(x,y,t) = 0.5xy - y$ and $g(x,y,t) = 2y - 0.1xy^2 - t$. The changes in the functions $x(t)$ and $y(t)$ are coupled together – as one changes then so must the other. While this system of equations might be impossible to solve analytically, we can approximate both of the derivatives in the same way that we did with Euler's method in 1D and get a reasonable way to step this differential equation forward in time. Indeed,

$$x' \approx \frac{x_{n+1} - x_n}{\Delta t} = 0.5x_n y_n - y_n$$
$$y' \approx \frac{y_{n+1} - y_n}{\Delta t} = 2y_n - 0.1x_n y_n^2 - t_n.$$

After simplifying a bit we get

$$x_{n+1} = x_n + \Delta t\left(0.5x_n y_n - y_n\right)$$
$$y_{n+1} = y_n + \Delta t\left(2y_n - 0.1x_n y_n^2 - t_n\right).$$

As you can see, this is now a system of difference equations that can be coded with relative ease.

Technique 5.17 (2D Euler's Method)**.** If

$$x' = f(x, y, t)$$
$$y' = g(x, y, t)$$

then we can build a system of difference equations to approximate the solution by taking a first-order approximation of the derivatives and rewriting the system as

$$x_{n+1} = x_n + \underline{\hspace{2cm}}$$
$$y_{n+1} = y_n + \underline{\hspace{2cm}}.$$

Problem 5.18. Write a Python function for 2D Euler's method. Test your code on a problem where you know the qualitative solution and can easily graphically check the answer. ▲

Problem 5.19. Test your code from the previous problem on the following by showing a time evolution plot (time on x and populations on y) as well as a phase plot (x on the x and y on the y with time understood implicitly):
The Lotka-Volterra Predator-Prey Model:
Let $x(t)$ denote the number of rabbits (prey) and $y(t)$ 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:

$$\begin{cases} x' &= \alpha x - \beta xy \\ y' &= -\delta y + \gamma xy \end{cases}$$

where α, β, γ, and δ are positive constants. For this problems take $\alpha \approx 1$, $\beta \approx 0.05$, $\gamma \approx 0.01$, and $\delta \approx 1$. Be sure to explain the meaning of each of the parameters and each of the components of the model. ▲

5.3 The Midpoint Method

Problem 5.20. Let's return to the simple differential equation $y' = -0.5y$ with $y(0) = 6$ that we saw in Problem 5.13. Now we'll propose a slightly different method for approximating the solution.

(a) At $t = 0$ we know that $y(0) = 6$. 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 $t = 0$? $y'(0) =$ _____
  - Use this slope to step a half step forward and find the y value: $y(0.5) \approx$ _____
  - Now use the differential equation to find the slope at time $t = 0.5$. $y'(0.5) =$ _____
  - Now take your answer from the previous step, and go one full step forward from time $t = 0$. What y value do you end up with?
  - Your answers to the previous bullets should be: $y'(0) = -3$, $y(0.5) \approx 4.5$, $y'(0.5) = -2.25$, so if we take a full step forward with slope $m = -2.25$ starting from $t = 0$ we get $y(1) \approx 3.75$.

(b) Repeat the process outlines 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 y	6										
Exact value of y	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 Problem 5.13?

▲

> **Technique 5.21** (The Midpoint Method). Now we begin the journey of creating better solvers than Euler's method. The midpoint method is defined by first taking a half step with Euler's method to approximate a solution at time $t_{n+1/2} \equiv (t_n + t_{n+1})/2$ and

then taking a full step using the value of f at $t_{n+1/2}$ and the approximate $y_{n+1/2}$.

$$y_{n+1/2} = y_n + \frac{h}{2} f(y_n, t_n)$$
$$y_{n+1} = y_n + h f(y_{n+1/2}, t_{n+1/2})$$

Note: Indexing by 1/2 in a computer is nonsense. Instead, we implement the midpoint method with:

$$y_{temp} = y_n + \frac{h}{2} f(y_n, t_n)$$
$$y_{n+1} = y_n + h f\left(y_{temp}, \frac{t_n + t_{n+1}}{2}\right)$$

Problem 5.22. Write Python code to implement the midpoint method

```
function [t,y]=MyMidpointMethod(f,tmin,tmax,num_pts,IC)
```
▲

Problem 5.23. Repeat Problem 5.16 with the midpoint method. Compare your results to what you found with Euler's method. ▲

Problem 5.24. Test your midpoint method code against your Euler1D code on the same single variable ODE as before. You will likely see very little difference on a very small step size (equivalently, a large number of points), but for a smaller number of points there will be a remarkable difference. ▲

We have studied two methods thus far: Euler's method and the Midpoint method. In Figure 5.2 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 as

Euler's Method	Midpoint Method
1. Get the slope at time t_n	1. Get the slope at time t_n
2. Follow the slope for time Δt	2. Follow the slope for time $\Delta t/2$
	3. Get the slope at the point $t_n + \Delta t/2$
	4. Follow the new slope from time t_n for time Δt

Problem 5.25. 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? ▲
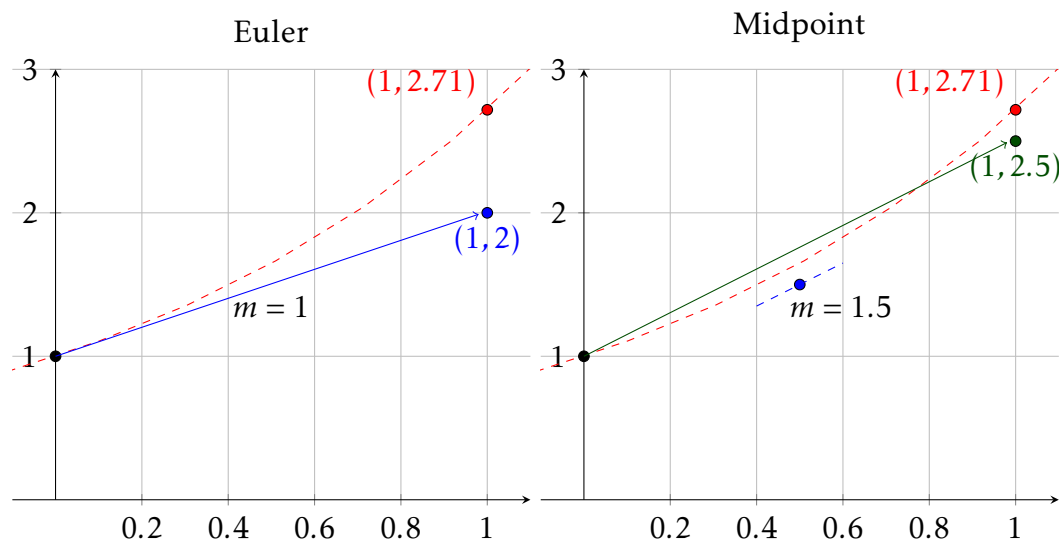
Figure 5.2. Graphical depictions of two numerical methods: Euler (left) and Midpoint (right). Here we use the simple differential equation $y' = y$ with $y(0) = 0.5$ and $\Delta t = 1$. The exact solution is shown in red.

5.4   The Runge-Kutta Method

Problem 5.26. Again we return to the differential equation $y' = -0.5y$ with $y(0) = 6$ that we saw in Problem 5.13. This time we'll walk through a slightly more challenging algorithm for approximating the slope solution to the differential equation. In this algorithm we will make four approximations of the slope at a point and then use a weighted average of these four to take a full step forward. The four approximations will be called k_1, k_2, k_3, and k_4.

(a) At $t = 0$ we know that $y(0) = 6$. Therefore, what is the slope at $t = 0$? $y'(0) = $ _____.
    We will call this value k_1.

(b) To calculate k_2 we take a half-step forward using the slope k_1. Using this slope, at
    time $t = 0.5$ we find that $y'(0.5) = $ _____. We will call this value k_2.

(c) To calculate k_3 we take another half-step forward from the starting point but this
    time using the slope k_2. Using this slope, at the $t = 0.5$ we find that $y'(0.5) = $ _____.
    We will call this value k_3.

(d) To calculate k_4 we now take a full step forward using the slope k_3. Using this slope
    we find that at $t = 1$ we have $y'(1) = $ _____. Call this slope k_4.

(e) To check your work you should now verify that

$$y(0) = 6 \quad \implies y'(0) = -0.5 \cdot 6 = -3 \quad \implies k_1 = -3$$

$$y(0.5) \approx 6 - \frac{1}{2} \cdot 3 = 4.5 \quad \implies y'(0.5) = -0.5 \cdot 4.5 = -2.25 \quad \implies k_2 = -2.25$$

$$y(0.5) \approx 6 - \frac{1}{2} \cdot 2.25 = 4.875 \quad \implies y'(0.5) = -0.5 \cdot 4.875 = -2.4375 \quad \implies k_3 = -2.4375$$

$$y(1) \approx 6 - 2.4375 = 3.5625 \quad \implies y'(1) = -0.5 \cdot 3.5625 = -1.78125 \quad \implies k_4 = -1.78125$$

If your answers don't match then go back to parts (a) - (d) and check your thinking and arithmetic.

(f) We now have four approximations of the slope at the point $t = 0$. To move forward take a weighted average of these four approximations

$$\text{slope} \approx \frac{1k_1 + 2k_2 + 2k_3 + 1k_4}{6}.$$

Calculate this new slope and step forward one full step. You should find that your approximate slope is -2.359375 and this gives an approximate value of $y(1)$ as $y(1) \approx 6 - -2.359375 = 3.640625$. Compare this to the exact value of the solution to the differential equation at $t = 1$. You should see that you have some pretty amazing accuracy!!

(g) Push this process forward one more step by hand. Use the table below to summarize your results.

t	0	1	2
k_1	-3		$\times$
k_2	-2.25		$\times$
k_3	-2.4375		$\times$
k_4	-1.78125		$\times$
Approximation of slope	-2.359375		$\times$
Approximation of y	6	3.640625	
Exact Value of y	6	3.639184	

▲

The method that you stepped through in the previous problem is called the Runge-Kutta 4 method for approximating the solution to an ordinary differential equation. It is one of a family of many different approximation methods. The "4" in the name is not simply the fact that we are making four approximations of the slope, but in that it enjoys fourth-order accuracy – far better than both the Euler and Midpoint methods.

Technique 5.27 (The Runge-Kutta 4 (RK4) Method)**.** The Runge-Kutta 4 method is one (of many) such methods. In this method, each k_j is an approximation of the slope

and we combine them in as a weighted average in the end.

$$k_1 = f(y_n, t_n)$$

$$k_2 = f(y_n + \frac{h}{2}k_1, t_n + \frac{h}{2})$$

$$k_3 = f(y_n + \frac{h}{2}k_2, t_n + \frac{h}{2})$$

$$k_4 = f(y_n + hk_3, t_n + h)$$

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Before we write code to implement the RK4 method we will examine it graphically just as we did with Euler's method and the Midpoint method in Figure 5.2. For simplicity we will examine the differential equation $y' = y$ with initial condition $y(0) = 1$ and $\Delta t = 1$. In Figure 5.3 the red dashed line is the exact solution $y(t) = e^t$. In this example, $k_1 = 1$, $k_2 = 1.5$, $k_3 = 1.75$, and $k_4 = 2.75$. Hence the final slope propogating forward with $\Delta t = 1$ is

$$\frac{1}{6}(1 + 2(1.5) + 2(1.75) + 2.75) = 1.708.$$

Propogating this forward from the point $(0,1)$ gives the new point $(1, 2.708)$. Knowing that $e \approx 2.718$ we see a very high level of accuracy even with a really large time step!

Runge-Kutta 4 Method
1. k_1 is the slope evaluated at time t_n Project this slope half a step forward from time t_n to the point y_1
2. k_2 is the slope evaluated at y_1 Project the slope k_2 half a step forward from time t_n to the point y_2
3. k_3 is the slope evaluated at the point y_2 Project the slope k_3 a full step foward from time t_n to the point y_3
4. k_4 is the slope evaluated at the point y_3
5. Project forward with slope $\frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ from time t_n

Problem 5.28. Write a Python function that implements the Runge-Kutta 4 method in one dimension.

```
def MyRk4(f,tmin,tmax,num_pts,IC):
```

Test the problem on a known differential equation. ▲

Problem 5.29. Repeat Problem 5.16 with the Runge Kutta method. Compare your results to what you found with Euler's method and with the midpoint method. ▲

Problem 5.30. Modify your Runge-Kutta 4 code to work for two dependent variables. I'll get you started:
We want to solve

$$\begin{cases} x' & = f(x,y,t) \\ y' & = g(x,y,t) \end{cases}$$
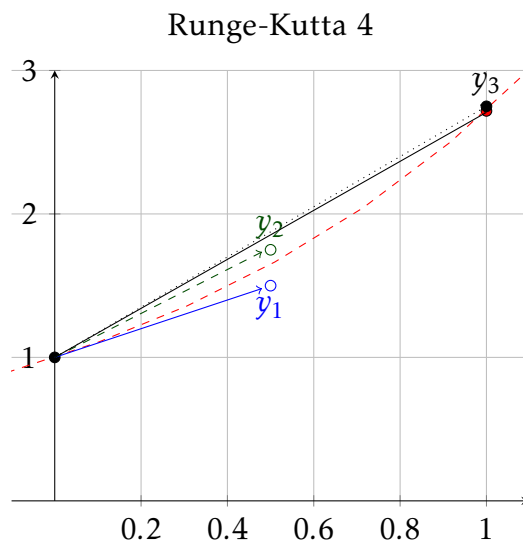
Figure 5.3. Graphical depiction of the RK4 method. The red dashed curve gives the exact solution to the differential equation $y' = y$ with initial condition $y(0) = 1$. The solid black line gives the final projection.

and to do so we extend the Runge Kutta method as

$$k_1 = f(x_n, y_n, t_n)$$

$$q_1 = g(x_n, y_n, t_n,)$$

$$k_2 = f(x_n + \frac{h}{2}k_1, y_n + \frac{h}{2}q_1, t_n + \frac{h}{2})$$

$$q_2 = g(x_n + \frac{h}{2}k_1, y_n + \frac{h}{2}q_1, t_n + \frac{h}{2})$$

$$k_3 = \ldots$$

$$q_3 = \ldots$$

$$k_4 = \ldots$$

$$q_4 = \ldots$$

$$x_{n+1} = x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$y_{n+1} = y_n + \frac{h}{6}(q_1 + 2q_2 + 2q_3 + q_4)$$

Test your code on the predator prey model in problem 5.19. ▲

Problem 5.31. Solving <u>systems</u> of ordinary differential equations would become challenging if we were to <u>continue</u> coding in the same way as in the previous problem – modifying your code to account for the number of differential equations. Write a Python function that accepts any number of right-hand sides from a system of differential equations and then leverages the fact that Python works very well with vectors to create Euler

and Runge-Kutta solutions to these systems. Devise several systems to test your code (including 1D and 2D).

One particular nonlinear system of differential equations that you can test on is

$$\begin{cases} y_1' &= -0.1y_2y_1 - y_1 & y_1(0) = 2 \\ y_2' &= -y_1 + 0.9y_2 & y_2(0) = 1 \\ y_3' &= \sin(t) + \cos(y_2) & y_3(0) = -2 \end{cases} .$$

Note that we would probably never try to solve this problem by hand so a numerical method is warranted.

A test script to check your General Euler and General RK code is as follows. You can use my notation here to possibly reverse engineer the construction of your general ODE solver codes.

Code:

```
1  numpts = 1000;
2  f = @(t,y) [-0.1*y(2)*y(1)-y(1) ;
3      -y(1)+0.9*y(2) ;
4      sin(t) + cos(y(2))];
5
6  tmax = 5.5;
7
8  [t,yeuler]=MyGeneralEuler(f,0,tmax,numpts,[2,1,-2]);
9  plot(t,yeuler(1,:),'b--',t,yeuler(2,:),'r--',t,yeuler(3,:),'k--')
10 hold on
11
12 [t,yrk]=MyGeneralRK4(f,0,tmax,numpts,[2,1,-2]);
13 plot(t,yrk(1,:),'k-.',t,yrk(2,:),'m-.',t,yrk(3,:),'c--')
14 grid on
15 legend('y_1''(t)' , 'y_2''(t)' , 'y_3''(t)')
```

Note: Build some sample code for this problem in the Python version   ▲

5.5   The Backwards Euler Method

Problem 5.32. The major trouble with the RK (and midpoint) methods is that they take many more function evaluations than Euler's method. We can improve upon Euler's method in the following way:
We want to solve $y' = f(t, y)$ so:

1. Approximate the derivative by looking forward in time(!)

$$\frac{y_{n+1} - y_n}{h} \approx f(y_{n+1}, t_{n+1})$$

2. Rearrange to get the difference equation

$$y_{n+1} = y_n + hf(y_{n+1}, t_{n+1}).$$

3. Notice that we <u>will</u> have t_{n+1} but we <u>do not have</u> y_{n+1}. The major trouble is that y_{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!!!

4. This method is called the **backward Euler** method and is known as an **implicit method** since you need to solve a nonlinear equation at each step. The advantage (usually) is that you can take far fewer steps with reasonably little loss of accuracy.

▲

Problem 5.33. Let's take a few steps through the backward Euler method on a problem that we know well: $y' = -0.5y$ with $y(0) = 6$.
Let's take $h = 1$ for simplicity, so the backward Euler iteration scheme for this particular differential equation is

$$y_{n+1} = y_n - \frac{1}{2}y_{n+1}.$$

Notice that y_{n+1} shows up on both sides of the equation. A little bit of rearranging gives

$$\frac{3}{2}y_{n+1} = y_n \quad \implies \quad y_{n+1} = \frac{2}{3}y_n.$$

(a) Complete the following table.

t	0	1	2	3	4	5	6	7	8	9	10
Euler Approx. of y	6	3	1.5	0.75							
Back. Euler Approx.of y	6	4	2.667	1.778							
Exact value of y	6	3.64	2.207	1.339							

(b) Compare now to what we found for the midpoint method on this problem as well.

▲

Problem 5.34. The previous problem could potentially lead you to believe that the backward Euler method will always result in some other nice difference equation. That isn't true! Let's consider a slightly more complicated differential equation and see what happens

$$y' = -\frac{1}{2}y^2 \quad \text{with} \quad y(0) = 0.$$

(a) Recall that the backward Euler approximation is

$$y_{n+1} = y_n + hf(y_{n+1}, t_{n+1}).$$

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 y_{n+1}. That is to say, if you are given a value of y_n then you need to solve a quadratic polynomial equation to get y_{n+1}. Let's be more explicit:
We know that $y(0) = 6$ so in our numerical solutions, $y_1 = 6$. In order to get y_2 we consider the equation $y_2 = y_1 - \frac{1}{2}y_2^2$. Rearranging we see that we need to solve $\frac{1}{2}y_2^2 + y_2 - 6 = 0$ in order to get y_2. Doing so gives us $y_2 = \sqrt{13} - 1 \approx 2.606$.

(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.

▲

The complications with the backward Euler's method are that you have a nonlinear equation to solve at every time step

$$y_{n+1} = y_n + hf(y_{n+1}, t_{n+1}).$$

Notice that this is the same as solving the equation

$$y_{n+1} - hf(y_{n+1}, t_{n+1}) - y_n = 0.$$

You know the values of h, t_{n+1} and y_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. One seed for your Newton's method could be the value from the previous time step since you likely don't expect your solution to change wildly from step to step. Following this logic, if you expect a solution at least *close* to where you currently are, then only a few steps of Newton's method should suffice (perhaps only 5 or 10!). You may want to experiment with how many steps of Newton's method you need to solve and still maintain some accuracy in your method.

Problem 5.35. Write Python code to implement the backward Euler's method for a 1D initial value problem.

```
def MyBackwardEuler(f, tmin, tmax, num_pts, IC):
```
▲

Problem 5.36. Write a Python 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. ▲

5.6 Exercises

5.6.1 Algorithm Summaries

Problem 5.37. Consider the first-order differential equation $y' = f(y, t)$. 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. ▲

Problem 5.38. Explain in clear language what Euler's method does geometrically. ▲

Problem 5.39. Consider the first-order differential equation $y' = f(y, t)$. 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. ▲

Problem 5.40. Explain in clear language what the Midpoint method does geometrically. ▲

Problem 5.41. Consider the first-order differential equation $y' = f(y, t)$. What is the Runge Kutta method for approximating the solution to this differential equation? What is the order of accuracy of the Runge Kutta method? Explain the meaning of the order of the method in the context of solving a differential equation. ▲

Problem 5.42. Explain in clear language what the Runge Kutta method does geometrically. ▲

5.6.2 Applying What You've Learned

Problem 5.43. Test the Euler, Midpoint, and Runge Kutta methods on the differential equation
$$y' = \lambda (y - \cos(t)) - \sin(t) \quad \text{with} \quad y(0) = 1.5.$$
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 $\lambda = -1$, $\lambda = -10$, $\lambda = -10^2$, ..., $\lambda = -10^6$. 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. ▲

Problem 5.44. 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. ▲

Problem 5.45. Write code to solve the boundary valued differential equation
$$y'' = \cos(t)y' + \sin(t)y \quad \text{with} \quad y(0) = 0 \quad \text{and} \quad y(1) = 1.$$

▲

Problem 5.46. 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:

1. First implement this 2D system with $x(0) = 2$, $y(0) = 0$, $\alpha = 0.2$, $\beta = 0.8$, and $\gamma = 0.1$ for $t \in [0, 60]$. What is the fate of this pair's love under these assumptions?

2. Write Python 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 and also submit your Python file (along with any support files) demonstrating how it works.
   Hint: One way to do this problem is very similar to a bisection method for root finding.

   - Shoot two solutions with two different parameters.

   - Compare their results at the desired time against the result of shooting with the average value of the parameter.

   - Use the logic of the bisection method to make a new estimate of the parameter.

▲

Problem 5.47 (Orbiting Bodies Problem)**.** 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

$$x' = \underline{\hspace{5cm}}$$
$$v_x' = \underline{\hspace{5cm}}$$
$$y' = \underline{\hspace{5cm}}$$
$$v_y' = \underline{\hspace{5cm}}$$

(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 $(0, 0.5]$.

▲

Problem 5.48 (Pursuit and Evasion Problem)**.** 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

$$\|P'(t)\| = k\|E'(t)\| \tag{5.2}$$

and from the second assumption we have

$$\frac{P'(t)}{\|P'(t)\|} = \frac{E(t) - P(t)}{\|E(t) - P(t)\|}. \tag{5.3}$$

Solving for $P'(t)$ and using 5.2 the differential equation that we need to solve becomes

$$P'(t) = k\|E'(t)\|\frac{E(t) - P(t)}{\|E(t) - P(t)\|}. \tag{5.4}$$

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.

(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.

(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. ▲

Problem 5.49 (Whales and Krill Problem)**.** 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.

▲

Problem 5.50 (Drone Path Problem)**.** 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 drones 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?

▲

Problem 5.51 (Trebuchet Problem)**.** A trebuchet catapult throws a cow vertically into the air. The differential equation describing its acceleration is

$$\frac{d^2y}{dt^2} = -g - c\frac{dy}{dt}\left|\frac{dy}{dt}\right|$$

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.   ▲

Problem 5.52 (Classical SIR Model). When a virus is introduced into a small homogeneously mixed population the people in the population can be split into three categories: susceptible to the virus (S), infected with the virus (I), and recovered from the virus (R). Assume the following:

- a susceptible person becomes infected at a rate proportional to the product of the number of susceptible people and the number of infected people,

- the recovery rate is constant,

- the recovered people are immune to re-infection,

- the virus is not fatal so the total population stays fixed.

For this problem we will assume that there are $N = 1000$ people in the population with $I(0) = 1$ person initially infected. Your Tasks:

(a) Write a differential equation for the susceptible population assuming that the infection rate is proportional to the product of the sizes of the susceptible and infected populations

$$\frac{dS}{dt} = \underline{\qquad\qquad}.$$

(let the proportionality constant be α)

(b) Write a differential equation for the infected population knowing that susceptible people are becoming infected and that infected people are recovering at a constant rate

$$\frac{dI}{dt} = \underline{\qquad\qquad}.$$

(let the proportionality constant be β)

(c) Since the total population is fixed in size and only contains the three categories we know that $S + I + R = N$ and $\frac{dN}{dt} = 0$. Hence, the differential equations that you wrote in parts (a) and (b) are sufficient for modeling the three populations. Write numerical code that generates a plot of the three populations over time. Fully explore the parameters α and β and provide several plots that show the different dynamics of the problem.

▲

5.7 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.

5.7.1 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:

$$\frac{dP}{dt} = -(k_0 + k_1 D_1 + k_2 D_2 + k_3 D_3)P + k_0 u$$

$$\frac{dD_1}{dt} = -k_{D_1} D_1 + \sum_{j=1}^{N_1} \delta(t - \tau_{1,j})$$

$$\frac{dD_2}{dt} = -k_{D_2} D_2 + \sum_{j=1}^{N_2} \delta(t - \tau_{2,j})$$

$$\frac{dD_3}{dt} = -k_{D_3} D_3 + \sum_{j=1}^{N_3} \delta(t - \tau_{3,j})$$

where

- P is a patient's subjective pain level on a 0 to 10 scale,

- D_i is the amount of the ith 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 ith drug on the relaxation rate,

- u is the patient's baseline (unmitigated) pain,

- k_{D_i} is the elimination rate of the ith drug from the bloodstream,

- N_i is the total number of the ith drug doses taken, and

- $\tau_{i,j}$ are the time times the patient takes the ith drug.

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 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?

5.7.2   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.

5.7.3 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:

$$\frac{dv_x}{dt} = 0 \quad \text{and} \quad \frac{dv_z}{dt} = -g \tag{5.5}$$

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 equation 5.5 to *easily* show that the resulting trajectory is parabolic. Once we know this we can easily* calculate the initial speed v_0 and angle θ_0 above the horizontal necessary for the shell to reach the target. We will undoubtedly find that the maximum range will always result from an angle of $\theta_0 = 45°$.

The effects of air resistance are significant when the shell must travel a large distance or when the speed is large. If we modify (5.5) to include a simple model of air resistance the governing equations become

$$\frac{dv_x}{dt} = -cv_x\sqrt{v_x^2 + v_z^2} \quad \text{and} \quad \frac{dv_z}{dt} = -g - cv_z\sqrt{v_x^2 + v_z^2} \tag{5.6}$$

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

$$s_x(t) = \int_0^t v_x(\tau)d\tau \quad \text{and} \quad s_z(t) = \int_0^t v_z(\tau)d\tau.$$

(Note that you will have to implement this integration numerically. I suggest looping over t to build a vector of positions.)

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.

1. Develop a method for estimating v_0 and θ_0 with reasonable accuracy given the exact range to the target, Δx. Test the effectiveness of your method by comparing it to the

---

*You should now pause and actually verify that it is indeed an easy mathematical task to verify that the trajectory is parabolic and to calculate the initial speed and angle necessary to reach a target.

numerical solution of (5.6). Your method needs to be simple enough to use in real time on a historic (Nepoleon-era) battle field without the aid of a computer. (Be sure to persuade me that your numerical solution is accurate enough.)

2. Discuss the sensitivity in your solutions to variations in the constant c.

3. Extend this problem to make it more realistic. A few possible extensions are listed below but please do not restrict yourselves just to this list and do not think that you need to do everything on the list.

   - You could consider the effects of targets at different altitudes Δz.
   - You could consider moving targets.
   - You could consider headwinds and/or tailwinds.
   - You could consider winds coming from an angle outside the xz-plane.
   - You could consider shooting the cannon from a boat with the target on shore (the waves could be interesting!).
   - . . . You could consider any other physical situation which I 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.

Final note: Remember that every time you use and discuss numerical solutions to differential equations, numerical differentiation, or numerical integration in a technical paper you need to discuss the error in your method. All of our approximation methods come along with error and we have used Taylor series to derive the approximation error for most of them (the rest are easily Googleable). When you present your findings to the reader you could say something like the following.

> *We use the Runge-Kutta 4 (RK4) method to approximate the solutions to (5.6). Since the RK4 method is fourth-order and we took $\Delta t < 0.1$ seconds we know that the maximum error in our solution at any given time is on the order of $(\Delta t)^4 < 0.0001$.*

Similar statements need to be made <u>every</u> time you discuss the use of an approximation method.

5.7.4   The HIV Problem

In a normal, HIV-free body, uninfected T-cells are introduced into the system at a constant rate λ. The T-cells in the system have a finite life span, and hence a proportion μ of the T-cells die. In an HIV infected body, there is a rate k at which the T-cells become infected. This rate depends both on the presence of T-cells and free HIV-1 particles to infect the T-cells. Putting this together, we get the following equation for the healthy T-cells:

$$\frac{dT}{dt} = \lambda - \mu T - kTV$$

where

- $T(t)$ is the number of uninfected and activated T-cells at time t

- $L(t)$ is the number of latently infected T-cells at time t

- $I(t)$ is the number of actively infected T-cells at time t

- $V(t)$ is the number of free HIV-1 particles at time t.

Once a T-cell becomes infected, a proportion p of them will become latently infected, while the remainder immediately be actively infected. A latently infected T-cell can later become actively infected, and this happens at a rate α. Latently infected cells die at the same rate μ as uninfected T-cells. Actively infected T-cells will be assumed to die at a different rate a. We can now write the equations for the infected T-cells:

$$\frac{dL}{dt} = kpTV - \mu L - \alpha L$$
$$\frac{dI}{dt} = k(1-p)TV + \alpha L - aI$$

Finally, free HIV-1 particles are manufactured inside actively infected T-cells at a rate c. The particles die at a rate γ. The particles are also used in the process of infecting healthy T-cells at the rate k. This gives us the following equation for HIV-1 particles:

$$\frac{dV}{dt} = cI - \gamma V - kTV.$$

Numerically solve system of differential equations and plot the time evolution of all four components. Write a thorough mathematical and biological description of the evolution of the system. The table below gives values for the initial conditions and parameters.

Parameters	Value
$T(0)$	200 T-cells / mm^3
$L(0)$	0 T-cells / mm^3
$I(0)$	0 T-cells / mm^3
$V(0)$	4×10^{-7} HIV-1 cells / mm^3
λ	0.272 / mm^3 day
μ	1.36×10^{-3} / day
k	2.7×10^{-4} mm^3/day
p	0.1
α	3.6×10^{-2} / day
a	0.33 / day
c	100 / day
γ	2 / day

Chapter 6

Numerical Partial Differential Equations

"The mathematical discipline of differential equations furnishes the explanation of all those elementary manifestations of nature which involve time."
–Norwegian Mathematician Sophus Lie

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 particular 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 is likely governed by a PDE model.

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. Since PDEs require a strong background in the notions of vector calculus we'll start there.

6.1   Quick Review – Main Ideas from Vector Calculus

Let's start with some basic review of multivariable calculus.

Problem 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 $F(x,y) = \left\langle \sin(xy), x^2 + y^2 \right\rangle$ then what is $\nabla \cdot F$?

(d) If u is a function of x, y, and z then what is $\nabla \cdot \nabla u$?

(e) What is the divergence theorem? (ok . . . go ahead and use the internet for this one) Be able to explain what you find.

▲

Now that you've realized that you don't recall most of your multivariable calculus, let's simply recap.

Definition 6.2 (Definitions from Multivariable Calculus). The following are a few of the primary definitions and theorems from multivariable calculus.

- The **del** or **grad** operator, ∇, is a vector operator

$$\nabla = \left\langle \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right\rangle.$$

- The **gradient** of a multivariable function $u(x, y, z)$ is the vector

$$\nabla u = \left\langle \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z} \right\rangle.$$

- The **divergence** of a vector valued function $F = \langle F_1, F_2, F_3 \rangle$ is the scalar

$$\nabla \cdot F = \frac{\partial F_1}{\partial x} + \frac{\partial F_2}{\partial y} + \frac{\partial F_3}{\partial z}.$$

- The **curl** of a vector valued function $F = \langle F_1, F_2, F_3 \rangle$ is the vector

$$\nabla \times F = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ F_1 & F_2 & F_3 \end{vmatrix} = \left\langle \frac{\partial F_3}{\partial y} - \frac{\partial F_2}{\partial z}, -\left(\frac{\partial F_3}{\partial x} - \frac{\partial F_1}{\partial z} \right), \frac{\partial F_2}{\partial x} - \frac{\partial F_1}{\partial y} \right\rangle.$$

- The **Laplacian** of a multivariable function $u(x, y, z)$ is the scalar

$$\nabla \cdot \nabla u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}.$$

- The **divergence theorem** states that the flux of a vector field out of closed body is the same as the integral of the divergence of the vector field within the body.

$$\iint F \cdot n \, dA = \iiint \nabla \cdot F \, dV.$$

6.2   An Intuitive Introduction to some Common PDEs

To build intuition for partial differential equations we'll first start with your intuition with ordinary differential equations. Let's consider the really simple ODE $y' = y$. We can verbalize this problems with the phrase *the rate of change of y is equal to the current value of y*, and we can use this phrase, along with our understanding of linear approximation from Calculus, to build intuition about how the value of y propogates in time. If we start with a value of $y(0) = 0.5$ then after 1 unit of time passes we have a reasonable guess for the value of y based on our understanding of slope:

$$y(1) \approx 0.5 + (1)(0.5) = 1.$$

Similarly we can propogate forward 1 unit of time again to get

$$y(2) \approx 1 + 1(1) = 2,$$

and we can continue this to get

Time	0	1	2	3	4	$\cdots$
y	0.5	1	2	4	8	$\cdots$

We also know that this isn't quite correct since the solution to the differntial equation is $y(t) = 0.5e^t$. In Figure 6.1 we can see, however, that our intuitive guesses get us somewhat close to the basic behavior of the differential equation but clearly miss the finer details associated with the exact curvature.
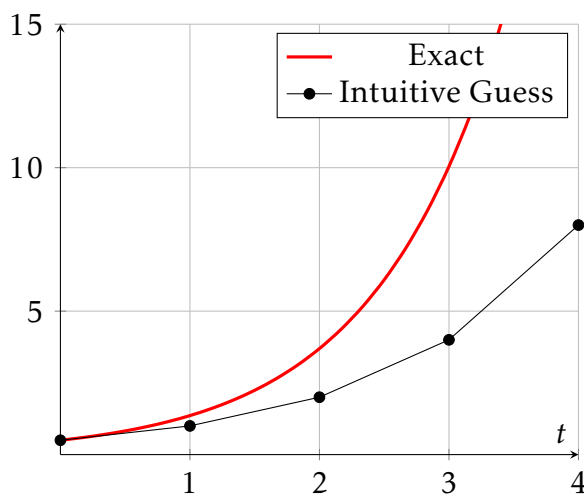
Figure 6.1. The analytic solution to $y' = y$ and the points given by calculus intuition.

We're going to use this same idea to build intuition for some of the most basic partial differential equations. In these partial differential equations we have two variables: $t =$ time and $x =$ a single spatial dimension.

Problem 6.3. Let $u(t,x)$ be the concentration of a quantity at time t and spatial location x. The quantity might be something like heat or chemical concentration. We will let $x \in [0,1]$ and we assume that at time $t = 0$ we have $u(0,x) = \sin(2\pi x)$ as shown in the plot below. Use the phrase

the time rate of change of concentration is equal to the concavity of the concentration function

to give several plots showing how the concentration evolves in time. The initial condition function $u(0,x)$ is shown in each plot for reference. Think of this as creating four frames in an animation.

▲
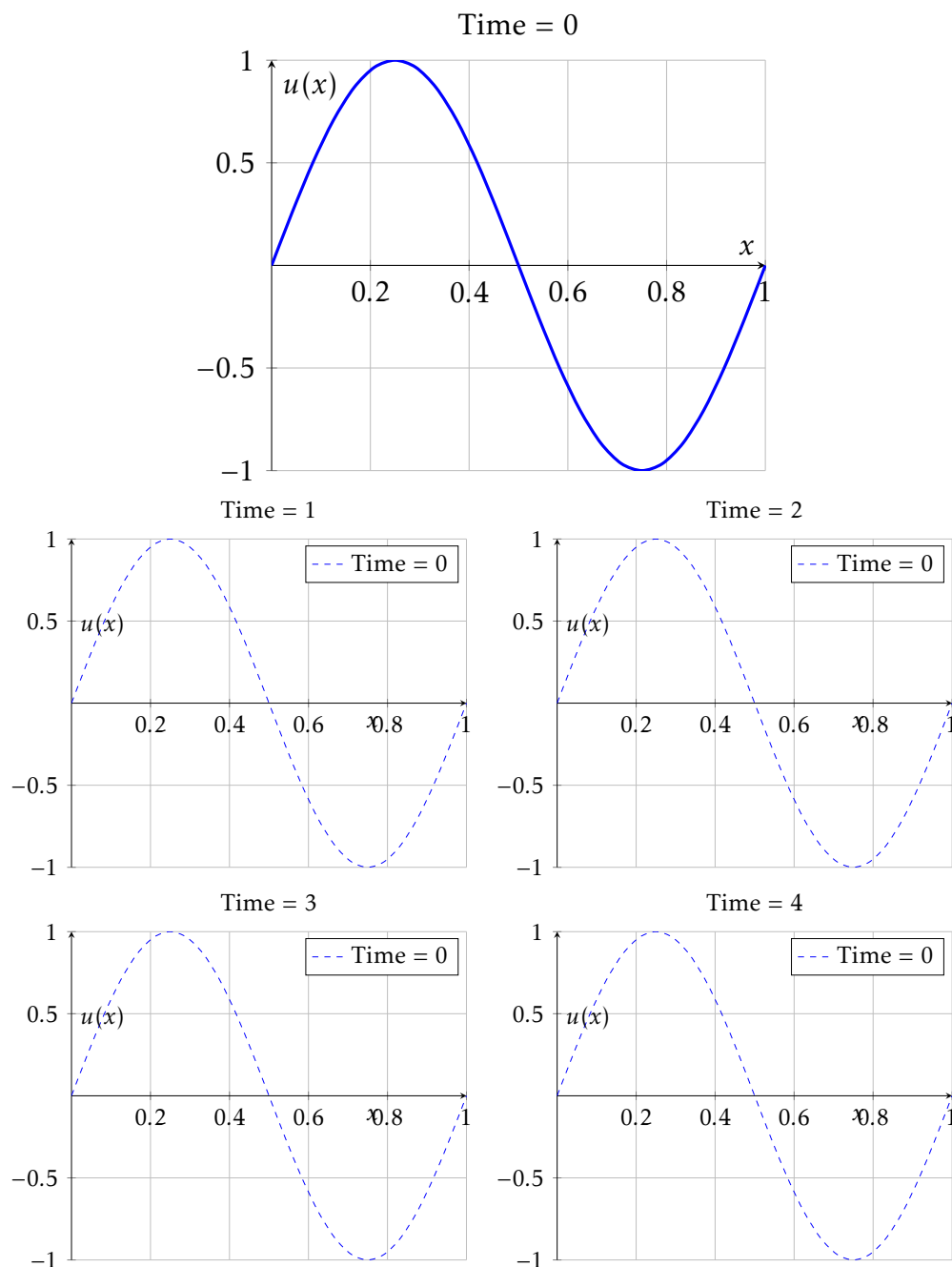
Problem 6.4. Which of the following differential equations corresponds to the phrase *"the time rate of change of concentration is equal to the concavity of the concentration function"*?

$$\frac{\partial u}{\partial t} = -\frac{\partial u}{\partial x}, \qquad \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \quad \text{or} \quad \frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}$$

Explain your reasoning clearly. ▲

Problem 6.5. Based on your answer to Problem 6.3 you should now have an intuitive sense for the behavior of the solution to the partial differential equation that you identified in Problem 6.4. This equation is called the **Heat Equation** or the **Diffusion Equation**. Give context to the physical process that is being described by this equation. ▲

Problem 6.6. How would your answer to Problem 6.3 change if we were to modify the heat equation to

$$\frac{\partial u}{\partial t} = 0.5\frac{\partial^2 u}{\partial x^2}?$$

What about

$$\frac{\partial u}{\partial t} = 2\frac{\partial^2 u}{\partial x^2}?$$

▲

Problem 6.7. The 1D Heat Equation is given as

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}.$$

What does the parameter D control in terms of the physics of the problem? ▲

Problem 6.8. Let $u(t,x)$ be the position of a string or cable at time t and spatial location x. We will let $x \in [0,1]$ and we assume that at time $t = 0$ we have $u(0,x) = \sin(2\pi x)$ as shown in the plot below. Use the phrase

the acceleration of each point on the string is equal to the concavity of the string

to give several plots showing how the concentration evolves in time. The initial condition function $u(0,x)$ is shown in each plot for reference. Think of this as creating four frames in an animation.
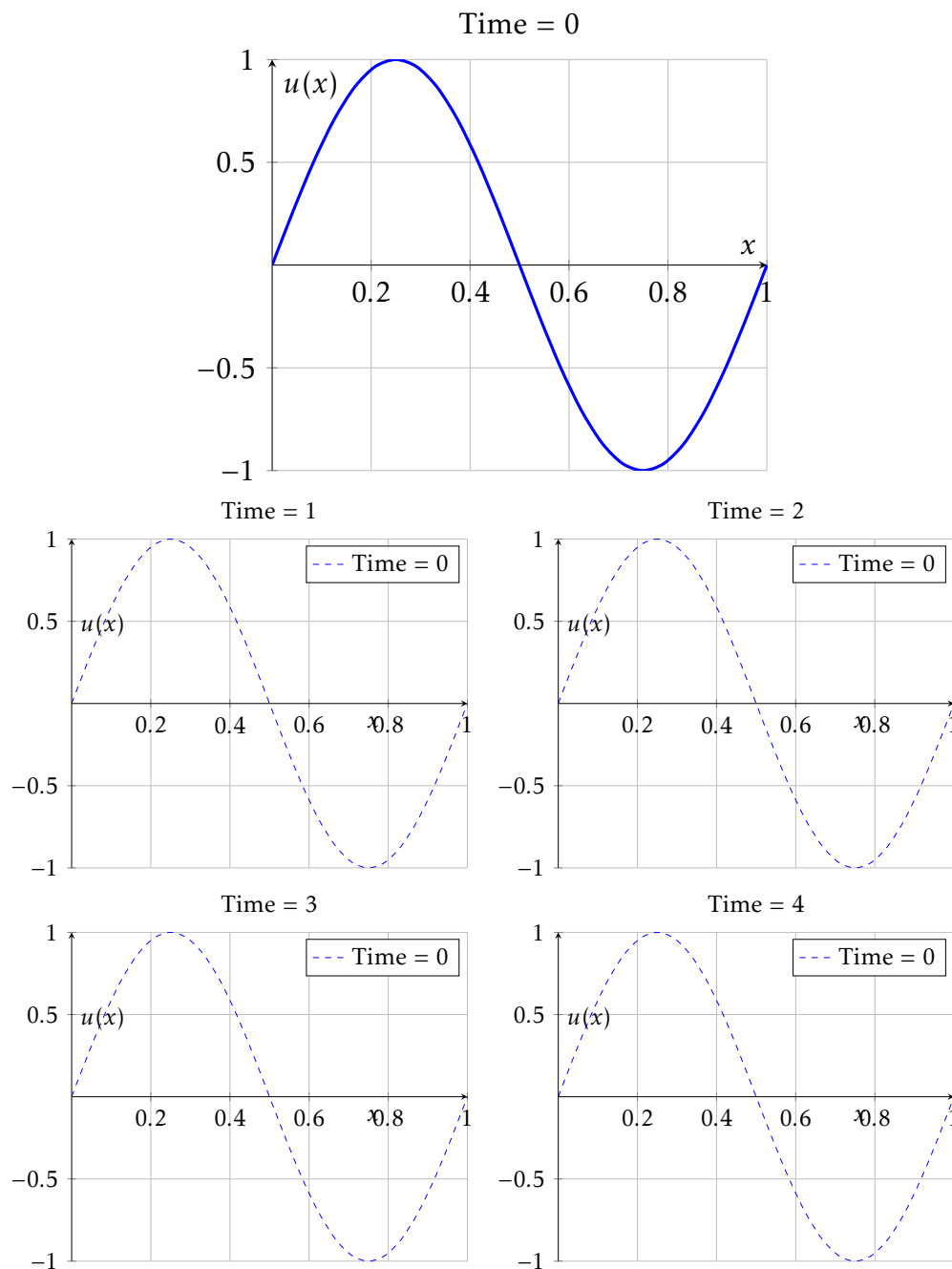
Problem 6.9. Which of the following differential equations corresponds to the phrase *"the acceleration of each point on the string is equal to the concavity of the string"*?

$$\frac{\partial u}{\partial t} = -\frac{\partial u}{\partial x}, \qquad \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \quad \text{or} \quad \frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}$$

Explain your reasoning clearly. ▲

Problem 6.10. Based on your answer to Problem 6.8 you should now have an intuitive sense for the behavior of the solution to the partial differential equation that you iden-

tified in Problem 6.9. This equation is called the **Wave Equation**. Give context to the physical process that is being described by this equation. ▲

Problem 6.11. How would your answer to Problem 6.8 change if we were to modify the wave equation to

$$\frac{\partial^2 u}{\partial t^2} = 0.5 \frac{\partial^2 u}{\partial x^2}?$$

What about

$$\frac{\partial^2 u}{\partial t^2} = 2 \frac{\partial^2 u}{\partial x^2}?$$

▲

Problem 6.12. The 1D Wave Equation is given as

$$\frac{\partial^2 u}{\partial t^2} = \alpha^2 \frac{\partial^2 u}{\partial x^2}.$$

What does the parameter α^2 control in terms of the physics of the problem? ▲

6.3   Analytic Solutions to Linear PDEs

If I were to claim that $y = f(t) = 7e^{3t}$ is a solution to the ordinary differential equation $\frac{dy}{dt} = 3y$ with $y(0) = 7$ then you could easily check that my claim were true by checking the initial condition

$$f(0) = 7e^{3\cdot 0} = 7 \quad \checkmark$$

and checking that the function satisfies the differential equation

$$y' = 3 \cdot 7e^{3t} = 3y \quad \checkmark.$$

Let's do the same for some partial differential equations.

Example 6.13. Consider the diffusion equation $u_t = ku_{xx}$. This is known as the "heat" or "diffusion" equation as we saw in the previous section, and the function u can be thought of as the temperature of some heated object.
Question: Is the function $u(x,t) = x^2 - t^3$ a valid solution for this differential equation? If so, what is the value of the constant k?
Solution:
We will check this by taking a few derivatives.

$$\frac{\partial u}{\partial t} = -3t^2$$
$$\frac{\partial^2 u}{\partial x^2} = 2.$$

As we can see, there is no constant value of k that makes the equation $-3t^2 = 2k$ valid. Therefore, the function $u(x,t) = x^2 - t^2$ is not a solution to the heat equation.

Problem 6.14. Consider the diffusion equation $T_t = kT_{xx}$ where $T(x,t)$ is the temperature of a long thin metal rod at time t (in seconds) and spatial location x (in meters).

(a) What are the units of the constant k?

(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. If $k = 3$, which of these functions is a solution? Be able to defend your answer.

    (1) $T(x,t) = 4x^3 + 6t^2$

    (2) $T(x,t) = 7x + 5$

    (3) $T(x,t) = 8x^2 t$

    (4) $T(x,t) = e^{3t+x}$

    (5) $T(x,t) = 6e^{3t+x} + 5x - 2$

(6) $T(x,t) = e^{-3t} + \sin(x)$

(7) $T(x,t) = e^{3t} \sin(x)$

(8) $T(x,t) = e^{-3t} \sin(x)$

(9) $T(x,t) = 5e^{-3t} \sin(x) + 6x + 7$

(10) $T(x,t) = -4e^{-3t} \sin(x) + 3t + 2$

(11) $T(x,t) = e^{-2t} \sin(3x)$

(12) $T(x,t) = e^{-12t} \cos(3x)$

(13) $T(x,t) = e^{-12t} \cos(3x) + 4x^2 + 8$

(14) $T(x,t) = e^{-75t} \cos(5x)$

(15) $T(x,t) = 9e^{-75t} \cos(5x) + 2x + 7$

▲

Problem 6.15. Consider the diffusion equation $T_t = kT_{xx}$, and suppose that $k = 4$. 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) $T(x,t) = 6e^{-8t} \sin(ax)$

(b) $T(x,t) = -5e^{38t} \cos(ax)$

(c) $T(x,t) = 3e^{at} \sin(5x)$

(d) $T(x,t) = 7e^{at} \cos(2x)$

(e) $T(x,t) = ae^{-36t} \cos(3x)$

(f) $T(x,t) = ae^{-4t} \cos(6x)$

▲

In your study of ordinary differential equations you likely ran into the dissatisfying realization that you could simply *guess* the answer by knowing a little bit of calculus. As much as it might make you uncomfortable, this is a totally legitimate technique: guess the form of the answer, put it into the differential equation, and see what has to happen in order for your guess to be valid.

Problem 6.16. Consider the diffusion equation

$$\frac{\partial T}{\partial t} = k\frac{\partial^2 T}{\partial x^2}.$$

(a) Reading from left-to-right, the partial differential equation says that the 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?

(b) 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?

(c) Based on your guesses from parts (a) and (b), what type of function would think is a reasonable solution for the differential equation? Why?

▲

Problem 6.17. Find an analytical solution to the diffusion equation that satisfies the following:

(a) Our domain is $-10 \le x \le 10$ and $-20 \le y \le 20$, with both measured in meters.

(b) The thermal diffusivity is a constant k.

(c) The temperature around the edges is 20 degrees C.

(d) At the first instant of time, the temperature reaches a minimum at the center of 5 degrees C.

▲

   Based on what you have explored in the previous few problems you should now have a reasonable guess for the functional forms of solutions to the diffusion equation are.

Theorem 6.18. If the function $u(x,t)$ solves the diffusion equation $u_t = ku_{xx}$ in one spatial dimension then $u(x,t)$ likely takes the form

$$u(x,t) = \underline{\quad} e^{\left(\underline{\quad}\right)} \sin\left(\underline{\quad}\right)$$

or

$$u(x,t) = \underline{\quad} e^{\left(\underline{\quad}\right)} \cos\left(\underline{\quad}\right).$$

Problem 6.19. Prove that your hypotheses in the previous theorem are indeed solutions to the diffusion equation $u_t = ku_{xx}$ for some constant k.   ▲

Problem 6.20. As a sanity check, pick reasonable values for your constants and create an animation showing how your function evolves in time. Does your function show the behavior expected of the 1D diffusion equation?
Hint: Think of this as a cooling metal rod ... does your solution look plausible for this physical scenario?   ▲

Theorem 6.21. If the function $u(x,y,t)$ solves the diffusion equation $u_t = k(u_{xx} + u_{yy})$ in two spatial dimensions then $u(x,y,t)$ likely takes the form

$$u(x,y,t) = \underline{\hspace{5cm}}$$

Problem 6.22. Prove that your hypotheses in the previous theorem are indeed solutions to the 2D diffusion equation $u_t = k(u_{xx} + u_{yy})$ for some constant k.   ▲

Problem 6.23. As a sanity check, pick reasonable values for your constants and create an animation showing how your function evolves in time. Does your function show the behavior expected of the 2D diffusion equation?
Hint: Think of this as a cooling metal plate ... does your solution look plausible for this physical scenario?   ▲

Problem 6.24. Consider the wave equation

$$\frac{\partial^2 W}{\partial t^2} = c^2 \frac{\partial^2 W}{\partial x^2}.$$

(a) 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?

(b) 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?

(c) Based on your guesses from parts (a) and (b), what type of function would think is a reasonable solution for the differential equation? Why?

▲

Problem 6.25. Consider the wave equation $W_{tt} = c^2(W_{xx} + W_{yy})$ where $W(x,y,t)$ 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?

    (1) $W(x,y,t) = 3x + 2y + 5t - 6$

    (2) $W(x,y,t) = 3x^2 + 2y^2 + 5t^2 - 6$

    (3) $W(x,y,t) = \sin(2x) + \cos(3y) + \sin(4t)$

    (4) $W(x,y,t) = \sin(2x)\cos(3y)\sin(4t)$

    (5) $W(x,y,t) = \sin(3x)\cos(4y)\sin(10t)$

    (6) $W(x,y,t) = -6\sin(3x)\cos(4y)\sin(10t) + 2x - 3y + 9 - 12$

    (7) $W(x,y,t) = \cos(7x)\cos(3y)\cos(12t)$

(8) $W(x, y, t) = \cos(5x)\sin(12y)\cos(26t)$

▲

Problem 6.26. Find an analytical solution to the wave equation that satisfies the following

(a) Our domain is $0 \leq x \leq 5$ and $-6 \leq y \leq 6$, with both measured in meters.

(b) The wave speed is a constant c.

(c) The function equals zero at the edges.

(d) The initial velocity is zero everywhere.

(e) At the first instant of time, the height reaches a maximum at the center of 2 cm.

▲

Theorem 6.27. If the function $u(x, t)$ solves the 1D wave equation $u_{tt} = c^2 u_{xx}$ then $u(x, t)$ likely has the functional form

$$u(x, t) = \underline{\hspace{4cm}}.$$

Problem 6.28. Prove your hypotheses from the previous theorem. ▲

Problem 6.29. As a sanity check, pick reasonable values for your constants and create an animation showing how your function evolves in time. Does your function show the behavior expected of a wave equation?
Hint: Think of this as a vibrating guitar string ...does your solution look plausible for this physical scenario? ▲

Theorem 6.30. If the function $u(x, y, t)$ solves the 2D wave equation $u_{tt} = c^2 \left(u_{xx} + u_{yy} \right)$ then $u(x, y, t)$ likely has the functional form

$$u(x, y, t) = \underline{\hspace{4cm}}.$$

Problem 6.31. Prove your hypotheses from the previous theorem. ▲

Problem 6.32. As a sanity check, pick reasonable values for your constants and create an animation showing how your function evolves in time. Does your function show the behavior expected of a wave equation?
Hint: Think of this as a vibrating drum head ...does your solution look plausible for this physical scenario? ▲

6.4 Boundary Conditions

When we were solving ODEs we typically needed initial conditions to tell us where the solutions starts at time 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 6.33. Let's say that we want to solve the 1D heat equation $u_t = ku_{xx}$ 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 problem. They state that the value of the temperature is fixed at these points.

    - If the value of the solution u depends on the rate of change 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 for this problem. 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.

Problem 6.34. Consider solving the heat equation $u_t = u_{xx}$ in 1 spatial dimension. It will be helpful to reconsider Problem 6.3 for this problem.

(a) In Problem 6.3 we didn't explicitly state the boundary conditions. What type of boundary conditions were they? How can you tell?

(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 $\frac{\partial u}{\partial x}\Big|_{x=0} = 0$ and $u(t,1) = 1$. What types of boundary conditions are these? Draw a collection of pictures showing the expected evolution of the heat equation with these boundary conditions.

▲

Problem 6.35. Consider solving the wave equation $u_{tt} = u_{xx}$ in 1 spatial dimension. It will be helpful to reconsider Problem 6.8 for this problem.

(a) In Problem 6.8 we didn't explicitly state the boundary conditions. What type of boundary conditions were they? How can you tell?

(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 heat 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.

Problem 6.36. 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 $u_t = u_{xx}$. What are the appropriate boundary 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 $u_t = u_{xx}$. 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. We model the heat transport within the column with the 1D heat equation $u_t = u_{xx}$. Wat 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 $u_t = u_{xx} + u_{yy}$. What are the appropriate boundary conditions?

▲

Problem 6.37. 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 $u_{tt} = u_{xx}$. 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 $u_{tt} = u_{xx}$. 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 $u_{tt} = u_{xx} + u_{yy}$. What are the appropriate boundary conditions?

▲

6.5   Numerical Solutions of The Heat Equation

In this section we'll use a technique called *the finite difference method* to find numerical approximations to the heat equation

$$u_t = D\nabla \cdot \nabla u + f(x).$$

Recall that this equation governs the diffusive process of heat diffusion.

   In one spatial dimension the heat equation can be written as $u_t = ku_{xx} + f(x)$ and in two spatial dimensional 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.

6.5.1   1D Heat Equation

Problem 6.38. Now we would like to consider the time dependent heat equation

$$u_t = D\nabla \cdot \nabla u$$

in 1 spatial dimension. Note that D is 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.

   In 1 spatial dimension, the heat equation is simply

$$u_t = Du_{xx}$$

and we can approximate the derivatives with an Euler-type approximation of the time and a central difference in space:

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = D\left(\frac{U_{i+1}^n - 2U_i^n + U_{i-1}^n}{\Delta x^2}\right).$$

Here we are taking $U_i^n \approx u(t_n, x_i)$ (superscripts represent time step and subscripts represent spatial steps). Rearranging we see that

$$U_i^{n+1} = U_i^n + \frac{D\Delta t}{\Delta x^2}\left(U_{i+1}^n - 2U_i^n + U_{i-1}^n\right). \tag{6.1}$$

   Implement (6.1) in Python 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 $u(t,1) = 0$.

▲

Problem 6.39. You may have noticed in the previous problem that you will have terribly unstable solutions for certain choices of Δx and Δt. Set $D = 1$ in the previous problem and experiment with choices for Δx and Δt to find where (6.1) gives a stable numerical solution to the heat equation. For each choice of Δx and Δt report the value of $\frac{\Delta t}{\Delta x^2}$.   ▲

> **Theorem 6.40** (Stability of Finite Differences for the Heat Equation). Consider the 1D Heat Equation $u_t = Du_{xx}$. In the finite difference scheme for the 1D heat equation
>
> $$U_i^{n+1} = U_i^n + \frac{D\Delta t}{\Delta x^2}\left(U_{i+1}^n - 2U_i^n + U_{i-1}^n\right)$$
>
> the solution will be stable for
>
> $$\frac{D\Delta t}{\Delta x^2} < \underline{}$$

Proof. For a detailed proof of this fact we need to use a method called *Von Neumann Analysis*. See a detailed proof HERE. □

Problem 6.41. Modify your 1D heat equation code to solve the following problems. For each be sure to classify the type of boundary conditions given.

(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 $x \in (0,1)$, $u(0,x) = x^2$, $u_x(t,0) = 0$ 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 $u(t,1) = 0$.

▲

6.5.2   Stabilized 1D Heat Equation – The Crank Nicolson Method

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 partial differential equations, and surprisingly enough we can never completely beat these issues. That is to say that no matter how sophisticated of a method you use there will always be some region where the parameters of the problem give rise to instability.

Problem 6.42. 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:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} = \frac{1}{2}\left[D\left(\frac{U_{j+1}^n - 2U_j^n + U_{j+1}^n}{\Delta x^2}\right) + D\left(\frac{U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j+1}^{n+1}}{\Delta x^2}\right)\right].$$

Letting $r = D\Delta t/(2\Delta x^2)$ we can rearrange to get

$$-rU_{j-1}^{n+1} + (1+2r)U_j^{n+1} - rU_{j+1}^{n+1} = rU_{j-1}^n + (1-2r)U_j^n + rU_{j+1}^n.$$

This can now be viewed as a system of equations. Let's build this system carefully and then write Python 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 j.

$$(j = 2): \quad -rU_1^{n+1} + (1 + 2r)U_2^{n+1} - rU_3^{n+1} = rU_1^n + (1 - 2r)U_2^n + rU_3^n$$

$$(j = 3): \quad -rU_2^{n+1} + (1 + 2r)U_3^{n+1} - rU_4^{n+1} = rU_2^n + (1 - 2r)U_3^n + rU_4^n$$

$$(j = 4): \quad -rU_3^{n+1} + (1 + 2r)U_4^{n+1} - rU_5^{n+1} = rU_3^n + (1 - 2r)U_4^n + rU_5^n$$

$$\vdots \qquad \vdots$$

$$(j = N - 1): \quad -rU_{N-2}^{n+1} + (1 + 2r)U_{N-1}^{n+1} - rU_N^{n+1} = rU_{N-2}^n + (1 - 2r)U_{N-1}^n + rU_N^n$$

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 equtions.

(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 so in MATLAB this code looks like

```
1  Un = U(n,2:end-1)'; % note the transpose
2  RHS = B*Un + D;
3  temp = A \ RHS; % note the \ doing the work of the inverse
4  U(n+1, 2:end-1) = temp;
```

Note: Write similar Python code

(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.

▲

6.5.3   2D Heat Equation

For the 2D heat equation we notice that the only new part of the PDE is the $\nabla \cdot \nabla u$ term in place of the 1 dimensional u_{xx} term. Recall that $\nabla \cdot \nabla u = u_{xx} + u_{yy}$ so we can use what we know about approximating second derivatives to approximate the right-hand side of the heat equation with

$$D\nabla \cdot \nabla u \approx D\left[\frac{U_{j+1,k}^n - 2U_{j,k}^n + U_{j-1,k}^n}{\Delta x^2} + \frac{U_{j,k+1}^n - 2U_{j,k}^n + U_{j,k-1}^n}{\Delta y^2}\right].$$

Assuming that $\Delta x = \Delta y$ and simplifying we can write the right-hand side of the 2D heat equation as

$$D\nabla \cdot \nabla u \approx \frac{D}{\Delta x^2}\left[U_{j+1,k}^n + U_{j,k+1}^n - 4U_{j,k}^n + U_{j-1,k}^n + U_{j,k-1}^n\right].$$

Notice that we had to invent a bit of notation in the process. The superscript, just as before, stands for the time step. The subscripts stand for the two spatial indices. More specifically, $U_{j,k}^n \approx u(t_n, x_j, y_k)$.

Problem 6.43. In Figure 6.2 you will see a schematic of the domain $\Omega = (0,1) \times (0,1)$ with homogeneous Dirichlet boundary conditions. Write code to solve the 2D Dirichlet problem

$$\text{Solve: } u_t = D\nabla \cdot \nabla u \quad \text{in} \quad x \in \Omega$$

with

$$u(0,x,y) = \sin(\pi x)\sin(\pi y)$$

$$u(t, x, 1) = 0)$$

$$u(t, 0, y) = 0 \qquad \Omega \qquad u(t, 1, y) = 0$$

$$u(t, x, 0) = 0$$
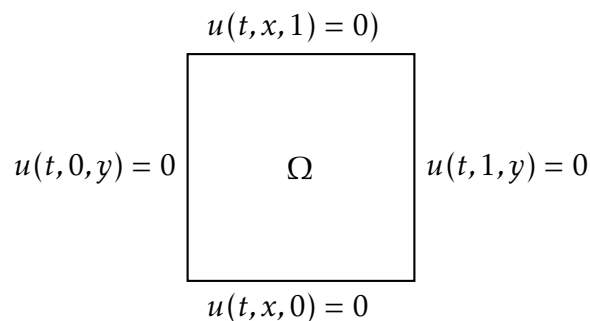
Figure 6.2. Dirichlet boundary conditions for a 2D Poisson equation.

subject to the boundary conditions in the figure.

For simplicity we suggest that you take $\Delta x = \Delta y$. You should also be very careful of the stability conditions for the heat equation. ▲

Problem 6.44. Repeat the previous exercise with different boundary conditions (both Dirichlet and Neumann) and with different domains (rectangular instead of square). For a rectangular domain it will likely be necessary to have different values for Δx and for Δy. Be prepared to present your solutions to your classmates. ▲

6.6 Numerical Solutions of The Wave Equation

Problem 6.45. 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

$$u_{tt} = \alpha^2 u_{xx} \tag{6.2}$$

where α is the stiffness of the wave. With Dirichlet boundary conditions we can think of this as the behaviour of a guitar string after it has been plucked.

Let's write Python code to numerically solve this problem:
Consider $u_{tt} = 2u_{xx}$ in $x \in (0, 1)$ with $u(0, x) = x(1 - x)$, $u_t(0, x) = 0$, and $u(t, 0) = u(t, 1) = 0$ and $\alpha = 5$. We can discretize the derivatives as

$$u_{tt}(t_{n+1}, x_j) \approx \frac{U_j^{n+1} - 2U_j^n + U_j^{n-1}}{\Delta t^2}$$

$$u_{xx}(t_n, x_j) \approx \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{\Delta x^2}$$

▲

Problem 6.46. There is a natural stability question waiting to be asked about the discretization of the 1D wave equation. Ask and answer this question. ▲

Problem 6.47. In Figure 6.3 you will see a schematic of the domain $\Omega = (0, 1) \times (0, 1)$ with homogeneous Dirichlet boundary conditions. Write code to solve the 2D Dirichlet problem

Solve: $u_{tt} = \alpha^2 \nabla \cdot \nabla u$ in $x \in \Omega$ with $u(0, x, y) = \sin(2\pi(x - 0.5))\sin(2\pi(y - 0.5))$

subject to the boundary conditions in the figure.

$$u(t, x, 1) = 0$$

$$u(t, 0, y) = 0 \quad \Omega \quad u(t, 1, y) = 0$$
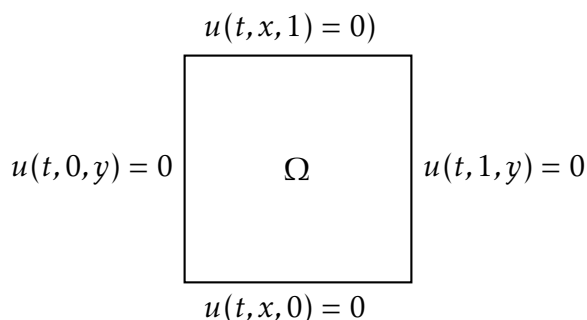
$$u(t, x, 0) = 0$$

Figure 6.3. Dirichlet boundary conditions for a 2D Poisson equation.

For simplicity we suggest that you take $\Delta x = \Delta y$. You should also be very careful of the stability conditions for the heat equation. ▲

6.7   Traveling Waves

Problem 6.48. A traveling wave can be modeled by the PDE

$$u_t + v u_x = 0$$

where v is the speed of the wave propogation and $u(t,x)$ is the height of the wave. Write a numerical solve for the traveling wave problem on $x \in (0,\infty)$ with initial condition $u(0,x) = \exp\left(-\frac{(x-1)^2}{0.1}\right)$, boundary condition $u(t,0) = 0$, and $a = 1$.

Solve this problem with and Euler-type time step and

1. centered differences in space, and

2. upwind differences in space.

where the "wind" is from left to right. Plot both solutions on top of each other. What do you notice about the behavior of the solutions? Neither of these solutions should actually give a traveling wave, but that is what is expected out of the solution.

Note about the analytic solution: If $\eta(x)$ is the initial condition for $u_t + v u_x = 0$ then $u(t,x) = \eta(x - vt)$ is the analytic solution to the PDE. You should check this using the chain rule.   ▲

Problem 6.49. Three ways to fix the issues seen in the previous problem are the "Leapfrog" scheme, the "Lax-Friedrichs" scheme, and the "Lax-Wendroff" scheme:

$$\text{Leapfrog: } \frac{U_j^{n+1} - U_j^{n-1}}{2\Delta t} = -v \frac{U_{j+1}^n - U_{j-1}^n}{2\Delta x} \tag{6.3}$$

$$\text{Lax-Friedrichs: } \frac{U_j^{n+1} - \frac{1}{2}\left(U_{j+1}^n + U_{j-1}^n\right)}{\Delta t} = -v \frac{U_{j+1}^n - U_{j-1}^n}{2\Delta x} \tag{6.4}$$

$$\text{Lax-Wendroff: } U_j^{n+1} = U_j^n - \frac{v\Delta t}{2\Delta x}\left(U_{j+1}^n - U_{j-1}^n\right) + \left(\frac{v^2 \Delta t^2}{2\Delta x^2}\right)\left(U_{j-1}^n - 2U_j^n + U_{j+1}^n\right) \tag{6.5}$$

Implement all of these schemes and discuss stability and consistency.   ▲

6.8 The Laplace and Poisson Equations – Steady State PDEs

Problem 6.50. 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 $u(0) = 1$,

- the temperature of the right end is held fixed at $u(1) = 0$, and

- the temperature has reached a steady state.

(assume that the temperatures are *reference temperatures* instead of absolute temperatures).

Since there are no external sources of heat we model the steady-state heat profile with Laplace's equation

$$u_{xx} = 0. \tag{6.6}$$

Write this equation in terms of 1-dimensional spatial derivatives and solve for the temperature profile by hand. ▲

Problem 6.51. Devise a way to approximate the temperature profile from the previous problem numerically. Recall that we already know how to build numerical second derivatives. Your method will eventually involve solving a system of linear equations. ▲

Problem 6.52. Now we will solve the steady state temperature profile problem assuming that there is an external source of heat. This means that we need to solve the 1D Poisson equation

$$u_{xx} = f(x). \tag{6.7}$$

Take $f(x) = 5\sin(2\pi x)$, $u(0) = 2$ and $u(1) = 0.5$.

$$\text{Solve:} \quad u_{xx} = -5\sin(2\pi x) \quad \text{on} \quad x \in (0,1) \quad \text{with} \quad u(0) = 2 \quad \text{and} \quad u(1) = 0.5$$

First do so by discretizing the domain with very few points so we can write the system of equations by hand. Write your code with the number of points as a parameter so you can later change it to several hundred points. ▲

Problem 6.53. Generalize the previous problem with a Python function that solves the 1D Poisson boundary valued equation:

$$\text{Solve:} \quad u_{xx} = -f(x) \quad \text{on} \quad x \in (x_0, x_n) \quad \text{with} \quad u(x_0) = \alpha \quad \text{and} \quad u(x_n) = \beta.$$

```
1  def Poisson1D(f , xmin , xmax , num_interior_pts , BCleft , BCright):
```

Test your code with a known function $f(x)$.

Note: when we are using fixed values for the boundary conditions these are called "Dirichlet boundary conditions." ▲

Problem 6.54. The previous problems 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:

$$\text{Solve: } u_{xx} = -f(x) \quad \text{on} \quad x \in (x_0, x_n) \quad \text{with} \quad \left.\frac{du}{dx}\right|_{x_0} = \alpha \quad \text{and} \quad u(x_n) = \beta.$$

Write a function to solve this problem:

```
1 def Poisson1D_Neumann(f , xmin , xmax , num_interior_pts , NBC , DBC):
```

Write a Python script to solve the Neumann problem with $f(x) = 2e^{-0.1x}$, $u'(0) = 0$, and $u(1) = 0$. ▲

Problem 6.55. Now let's ramp up our discussion of the Poisson equation to two spatial dimensions. In Figure 6.4 you will see a schematic of the domain $\Omega = (0,1) \times (0,1)$ with Dirichlet boundary conditions. With the help of your instructor, write code to solve the 2D Dirichlet problem

$$\text{Solve: } \nabla \cdot \nabla u = -f(x) \quad \text{in} \quad x \in \Omega \quad \text{with} \quad f(x,y) = 20\exp\left(-\frac{(x-0.5)^2 + (y-0.5)^2}{0.05}\right)$$

subject to the boundary conditions in the figure.

$$u(x,1) = \sin(2\pi x)$$

$$u(0,y) = 0 \qquad \Omega \qquad u(1,y) = 0$$

$$u(x,0) = \sin(\pi x)$$

Figure 6.4. Dirichlet boundary conditions for a 2D Poisson equation.

▲

6.9 Exercises

6.9.1 Algorithm Summaries

Problem 6.56. Explain in clear language what it means to check an analytic solution to a differential equation.   ▲

Problem 6.57. Explain in clear language what Dirichlet boundary conditions are.   ▲

Problem 6.58. Explain in clear language what Neumann boundary conditions are.   ▲

Problem 6.59. 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   ▲

Problem 6.60. 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   ▲

6.9.2 Applying What You've Learned

Problem 6.61. For every one of the scenarios described in Problem 6.36, propose a sensible initial condition and solve the problem numerically. Notice that the diffusion coefficient, D, is set to 1 for all of these models. You are welcome to change D if you see that it is necessary.   ▲

Problem 6.62. For every one of the scenarios described in Problem 6.37, propose a sensible initial condition and solve the problem numerically. Notice that the tension coefficient, α^2, is set to 1 for all of these models. You are welcome to change α^2 if you see that it is necessary.   ▲

Problem 6.63. 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 Python 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?

▲

Problem 6.64 (The Diffusing Logo). 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 `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.   ▲

Problem 6.65 (The Wiggling Logo). Repeat the previous exercise but this time solve the wave equation with the logo as the initial condition.   ▲

Problem 6.66. 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.   ▲

Problem 6.67. 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.

▲

6.10 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.

6.10.1 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 other's 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

$$\frac{\partial u}{\partial t} = ru\left(1 - \frac{u}{K}\right) - hu + D\nabla \cdot \nabla u.$$

(a) Use any of your ODE codes to solve the ordinary differential equation with harvesting. Give a complete description of the parameter space.

(b) 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.

(c) 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.

6.10.2 Heating Adobe Houses

Adobe houses, typically built in desert climates, are known for their great thermal efficiency. The heat equation

$$\frac{\partial T}{\partial t} = \frac{k}{c_p \rho} \nabla \cdot \nabla T,$$

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:

(a) 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.

(b) Do some research on the cost of building adobe walls and find approximations for the parameters in the heat equation.

(c) 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.

6.10.3   The River Contamination Problem

In this project you will be exploring a particular Partial Differential Equation: the Advection-Diffusion equation. In the following subsections of this document we will derive the equation from first principals. You will then be given a choice of applications of the advection diffusion equation. Your job will be to numerically solve the equation with physically realistic boundary and initial conditions, thus allowing you to study the possible behaviors of your system.

Derivation of the Advection-Diffusion Equation

The mathematical formulation for a balance law is

$$\frac{\partial u}{\partial t} + \nabla \cdot \mathbf{q} = f(t, \mathbf{x}) \tag{6.8}$$

where u is the quantity of interest (e.g. temperature, mass concentration, momentum, etc), $\mathbf{q}$ is the flux of the quantity, and f is any external source of the quantity. For example, in the heat equation we took Fourier's law for heat conduction which assumes that $\mathbf{q} = -\kappa \nabla T$ and arrived at the classic heat equation $\frac{\partial T}{\partial t} - \nabla \cdot (\kappa \nabla T) = f$. Similarly, if we are studying mass transport we might take Fick's law for molecular diffusion which assumes that $\mathbf{q} = -D \nabla C$ and arrive at the classic diffusion equation $\frac{\partial C}{\partial t} - \nabla \cdot (D \nabla C) = f$. These are, in effect, the exact same equation since the PDEs take the exact same functional form. The values of κ and D are the diffusivity constants and, in part, describe how easy it is for the conserved quantity to diffuse through the medium of interest. For example, in heat transport κ is related to the specific heat. One primary physical disadvantage to the heat equation (and it's analogous brethren) is that it does not allow for advection.

> **Definition 6.68. Advection** is the transport of conserved quantity or substance.

> **Definition 6.69. Diffusion** is the net movement of a conserved quantity or substance from regions of high concentration to regions of low concentration.

> **Definition 6.70. Convection** is the movement of a fluid, possibly in response to heat (e.g. convection currents from a heater). Often convection is used to describe the combined effects of advection and diffusion.

In the presence of a moving fluid, the flux term in (6.8) can be taken as the sum of a diffusive and an advective vector,

$$\mathbf{q} = -D \nabla u + \mathbf{v} u, \tag{6.9}$$

where D is the diffusivity parameter, u is the quantity of interest (e.g. concentration, temperature, etc.), and $\mathbf{v}$ is the flow velocity vector for the fluid. In more concrete terms,

if we are modeling die expanding in a flowing pipe, u might be the concentration of dye (maybe in ml/L) and $\mathbf{v}$ might be the constant flow rate of the pipe. Substituting (6.9) into (6.8) gives

$$\frac{\partial u}{\partial t} + \nabla \cdot (-D\nabla u + \mathbf{v}u) = f(t, \mathbf{x}). \tag{6.10}$$

Expanding the divergence on the left-hand side of (6.10) gives

$$\frac{\partial u}{\partial t} - \nabla \cdot (D\nabla u) + \nabla \cdot (\mathbf{v}u) = f(t, \mathbf{x}). \tag{6.11}$$

When a fluid is incompressible it can be shown that $\nabla \cdot \mathbf{v} = 0$. The impact of this is that by the product rule we have $\nabla \cdot (\mathbf{v}u) = \mathbf{v} \cdot \nabla u + u\nabla \cdot \mathbf{v}$ and since $\nabla \cdot \mathbf{v} = 0$ we get $\nabla \cdot (\mathbf{v}u) = \mathbf{v} \cdot \nabla u$. Hence, when modeling an incompressible fluid we can finally write the advection diffusion equation as

$$\boxed{\frac{\partial u}{\partial t} - \underbrace{\nabla \cdot (D\nabla u)}_{\text{Diffusion}} + \underbrace{\mathbf{v} \cdot \nabla u}_{\text{Advection}} = f(t, \mathbf{x}).} \tag{6.12}$$

Other simplifications include a non-spatially dependent diffusivity coefficient, implying that $\nabla \cdot (D\nabla u) = D\nabla \cdot \nabla u$, and problems that are free of external sources so $f(t, \mathbf{x}) = 0$. Keep in mind that these simplification are problem dependent. Without a source we can rearrange (6.12) to get

$$\boxed{\frac{\partial u}{\partial t} = \underbrace{\nabla \cdot (D\nabla u)}_{\text{Diffusion}} - \underbrace{\mathbf{v} \cdot \nabla u}_{\text{Advection}}.} \tag{6.13}$$

River Contamination Problem

Contaminant transport in a river is naturally modeled with the advection-diffusion equation. Let u be the concentration of a contaminant in the river and let $\mathbf{v}$ be the flow rate of the river. Equation (6.12) models the concentration of the contaminant as the river flows. You get to choose the river, the contaminant, the flow rate, the river's velocity profile[*], etc. A primary simplifying assumption is that the river is straight and two dimensional, but don't hesitate to include a 3D model if you feel it necessary.

A few words of caution: (1) The advection-diffusion equations are not designed to model turbulent flow so don't expect to see eddies in your solutions, (2) start with simple geometry and build up from there, and (3) there is a wealth of literature out there for the advection diffusion equation . . . just enough to get yourself lost and confused. Start simple and build from there.

Remember the most important part of this project: Have Fun!!

---

[*]You may want to look up what a pipe flow velocity profile looks like.

Project Goals

There are many areas of application for the advection diffusion equation which could span several majors, several courses, and possibly several undergraduate and graduate degrees. For this project we are only studying one broad area of application so as to narrow the scope of the project. You get to choose exactly what direction you want to take it. For your chosen direction you will need to do the following.

1. Clearly state what system you are trying to model.

2. Clearly state appropriate initial and boundary conditions for the system[†].

3. Clearly give reasonable physical values for all of the parameters (citing appropriate research).

4. Clearly explain the numerical methods used in the solution of the system. Your numerical methods should at least be 2D, but I won't stop you from doing 3D. You can also allow the river to be straight or allow it to bend. If you allow it to bend be sure to very carefully explain your numerical method. You should do at least two numerical methods and do a comparison study.

5. Include a verification that the time steps Δt and spatial steps Δx, Δy (and possibly Δz) are of the appropriate size to guarantee convergence. This means that you must compare the results for different choices of $\Delta t, \Delta x, \Delta y$, and Δz to verify that they produce the same result.[‡]

6. Present graphs and results exploring the behaviors that result from a wide variety of initial conditions, boundary conditions, changes in parameters, etc. Again, your project should at least be in two spatial dimensions. If you need to use animations (in addition to the images in your document) you can upload those along with your project and cite them accordingly.

7. Discuss the meaning of your work with the intention that the audience for your paper is mathematically sophisticated.

8. Finally, you will need to submit a zipped file containing Python codes that run select simulations and animations. The main driver file that you turn in needs to contain enough information so that your instructor can run your code by changing as few things as possible.[§]

[†]The boundary conditions for these problems take a great deal of care. You WILL need to look back to our homework from numerical differentiation to figure out how to discretize the derivatives near the boundaries. In particular, look at problem 6 on page 226 of the text. You WILL need this.

[‡]The best way to show convergence is some sort of error plot.

[§]Realistically, your instructor should just have to tell the code to run and it will do the rest. Cumbersome, broken, and hard-to-use code will not be taken lightly.

Appendix A

Python Basics

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.

Why Python?

We are going to be using Python in this class 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 develope 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/,

- 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.1 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. 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.

The environment in which you code is largely up to personal preference (or perhaps instructor preference). Jupyter Notebooks seem to be a good modern choice for coding environments. For more information and for installation of Jupyter see https://jupyter.org/. The beauty of Jupyter Notebooks is that you can mix your code and your writing together in a nice and seamless way. A slightly modern touch on Jupyter Notebooks is an interface called a Jupyter Lab. This is the exact same as a Jupyter Notebook, but it also gives you an interface for selecting files, changing directories, building new files, and other such basic functions. From now on, we'll use the terms "Jupyter Notebook" and "Jupyter Lab" as synonyms.

In the rest of this chapter we will assume that you have a working version of Python along with (most likely) a working version of Jupyter Notebooks to work in. Throughout this chapter all code will be highlighted in boxes so you, the reader, can easily tell that it is supposed to be code. The output of the code may also be in a box. Lastly, the code and the output have line numbers so the reader can more easily keep track of the commands, discuss with their peers, and discuss with their instructor. The problems in this appendix are meant to get you going with the Python and 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.2 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.

Code:

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

Output:

```
1  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 and you'll see your output.

Problem A.1. In a Jupyter Notebook have Python print `Hello, world!` to the screen. ▲

Problem A.2. Write code to print your name to the screen. ▲

Python is a general purpose programming language that can be used for all sorts of programming tasks. In this book we will focus on uses of Python that are more computational and mathematical in nature. It is expected that you already know a bit a programming from another math class that required some coding, possibly from an introductory computer science class, or from some other exposure to coding. If not then don't fret – not all hope is lost. A good place to start is to ask your instructor for additional programming materials.

We also recommend that you 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's hard drive

- change the working directory for a Jupyter Notebook.

- write text in a Jupyter Notebook

- use the keyboard to switch between writing text and writing code.

A.3   Python Programming Basics

A.3.1   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!).

Warning: When defining numerical variables you don't always get floating point numbers like in MATLAB. In MATLAB 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 <u>integer</u> (with no decimal digits) but if you assign x=1.0 it is assigned as a floating point <u>number</u>.

Example A.3 (Number Types in Python). **Code:**

```
1 # assign some variables
2 x = 7 # integer assignment of the integer 7
3 y = 7.0 # floating point assignment of the decimal number 7.0
4 print(x, type(x))
5 print(y, type(y))
6
7 # multiplying by a float will convert an integer to a float
8 print(1.0*x , type(1.0*x))
```

Output:

```
1 7 <class 'int'>
2 7.0 <class 'float'>
3 7.0 <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 powers.

Example A.4 (Powers in Python). Keep in mind that the caret key is not an exponent in Python.
Code:

```
1 x = 7.0
2 y = x**2 # square the value in x
3 print(y)
```

Output:

```
1 49.0
```

Problem A.5. What happens if you type 7^2 into Python? What does it give you? Can you figure out what it is doing? ▲

Problem A.6. Write code to define the variables a, b, and c of your own choosing. Then calculate a^2, b^2, and c^2. When you have all three computed, check to see if your three values form a Pythagorean Triple so that $a^2 + b^2 = c^2$ and have Python simply say True or False to verify that you do, or do not, have a Pythagorean Triple defined.

Hint: You will need to use the == Boolean check just like in other programming languages.   ▲

A.3.2   Indexing and Lists

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

Warning to MATLAB users: Python indexing starts at 0 whereas MATLAB indexing starts at 1. We just have to keep this in mind.

We can extract a part of a list using the syntax [start:stop] which extracts elements between index start and stop-1.

NOTE: Python stops reading at the second to last index.

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.7. Let's look at a few examples of indexing from lists. In this example we will use the list of numbers 1 through 8.

- Create the list of numbers 1 through 8 and then print only the element with index 0.

  Code:
  ```
  1  MyList = [1,2,3,4,5,6,7,8]
  2  print(MyList[0])
  ```

  Output:
  ```
  1  1
  ```

- Print all elements up to, but not including, the third element.

  Code:
  ```
  1  print(MyList[:2])
  ```

  Output:
  ```
  1  [1, 2]
  ```

- Print the last element (this is a handy trick!).

  Code:

  ```
  1  print(MyList[-1])
  ```

  Output:

  ```
  1  8
  ```

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

  Code:

  ```
  1  print(MyList[1:5])
  ```

  Output:

  ```
  1  [2, 3, 4, 5]
  ```

Example A.8. 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.

Code:

```
1  MySecondList = list(range(4,20)) # range is a handy command for creating a sequence of
2  print(MySecondList) # notice that it didn't create the last element!
3  print(MySecondList[0]) # print the first element ... the one with index 0
4  print(MySecondList[-5]) # print the fifth element from the end
5  print(MySecondList[-1:0:-1]) # print the last element to the one indexed by 1 counting
6  print(MySecondList[::2]) # print every other element starting at the beginning
```

Output:

```
1  [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
2  4
3  15
4  [19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5]
5  [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.9 (Lists with elements of mixed type)**.** 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.

Code:

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

Output:

```
1  [1.0, 7, 'Bob', (1-1j)]
2  <class 'float'>
3  <class 'int'>
4  <class 'str'>
5  <class 'complex'>
```

Problem A.10.   (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.

(d) Print the last element of the list.

▲

A.3.3   List Operations

Python is awesome about allowing you to do things like appending items to lists, removing items from lists, and inserting items into lists. Note in all of the examples below that we are using the code
`variable.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.11 (Appending To Lists)**.** The `.append` command can be used to append an element to the end of a list.
Code:

```
1  MyList = [0,1,2,3]
2  print(MyList)
3  MyList.append('a') # append the string 'a' to the end of the list
4  print(MyList)
5  MyList.append('a') # do it again ... just for kicks
6  print(MyList)
7  MyList.append(15) # append the number 15 to the end of the list
8  print(MyList)
```

Output:

```
1  [0, 1, 2, 3]
2  [0, 1, 2, 3, 'a']
3  [0, 1, 2, 3, 'a', 'a']
4  [0, 1, 2, 3, 'a', 'a', 15]
```

Example A.12 (Removing From Lists)**.** The `.remove` command can be used to remove an element from a list.

Code:

```
1  MyList.remove('a') # remove the first instance of the string `a` from the list
2  print(MyList)
3  MyList.remove(3) # now let's remove the 3
4  print(MyList)
```

Output:

```
1  [0, 1, 2, 3, 'a', 15]
2  [0, 1, 2, 'a', 15]
```

Example A.13 (Inserting Into Lists)**.** The `.insert` command can be used to insert an element at a location in a list.

Code:

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

Output:

```
1  ['A', 0, 1, 'B', 2, 'a', 15]
```

Problem A.14.   (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 (how do you suppose you should do this?)

(f) Do a few other list operations to this list and report your findings.

▲

A.3.4 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.

Example A.15 (Defining Tuples)**.** We can define the tuple of numbers $(10, 20)$ in Python as follows.
Code:

```
1  point = 10, 20 # notice that I don't need the parenthesis
2  print(point, type(point))
```

Output:

```
1  (10, 20) <class 'tuple'>
```

We can also define the type with parenthesis if we like. Python doesn't care.
Code:

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

Output:

```
1  (10, 20) <class 'tuple'>
```

Example A.16 (Unpacking Tuples)**.** The cool thing is that we can then unpack the tuple into components if we wish. **Code:**

```
1  x, y = point
2  print("x = ", x)
3  print("y = ", y)
```

Output:

```
1  x =  10
2  y =  20
```

A.3.5 Control Flow: Loops and If Statements

Just like in other programming languages we can do loops and conditional statements in very easy ways. The thing to keep in mind is that Python 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 (like MATLAB) 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.

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.17. Print the first 6 perfect square.
Code:

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

Output:

```
1  1.0
2  4.0
3  9.0
4  16.0
5  25.0
6  36.0
```

Example A.18. Print the names in a list.
Code:

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

Output:

```
1  Alice
2  Billy
3  Charlie
4  Dom
5  Enrique
```

```
6  Francisco
```

You can use a more compact notation sometimes. This takes a bit of getting used to, but is super slick!

Example A.19. Create a list of the perfect squares from 1 to 9.
Code:

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

Output:

```
1  [1, 4, 9, 16, 25, 36, 49, 64, 81]
2  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.20. In the following code we write a for loop that outputs a list of the first 10 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.
Code:

```
1  x=[3.0]
2  for n in range(0,9):
3      x.append(-0.5*x[n] + 1)
4  print(x)
```

Output:

```
1  [3.0, -0.5, 1.25, 0.375, 0.8125, 0.59375, 0.703125, 0.6484375,
2      0.67578125, 0.662109375]
```

As an alternative to the code immediately above 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.

Example A.21. Now we'll build the sequence with pre-allocated memory.
Code:

```
1  x = [0] * 10
2  x[0] = 3.0
3  for n in range(0,9):
4      x[n+1] = -0.5*x[n]+1
```

```
5      print(x) # This print statement shows x at each iteration
```

Output:

```
1  [3.0, −0.5, 0, 0, 0, 0, 0, 0, 0, 0]
2  [3.0, −0.5, 1.25, 0, 0, 0, 0, 0, 0, 0]
3  [3.0, −0.5, 1.25, 0.375, 0, 0, 0, 0, 0, 0]
4  [3.0, −0.5, 1.25, 0.375, 0.8125, 0, 0, 0, 0, 0]
5  [3.0, −0.5, 1.25, 0.375, 0.8125, 0.59375, 0, 0, 0, 0]
6  [3.0, −0.5, 1.25, 0.375, 0.8125, 0.59375, 0.703125, 0, 0, 0]
7  [3.0, −0.5, 1.25, 0.375, 0.8125, 0.59375, 0.703125, 0.6484375, 0, 0]
8  [3.0, −0.5, 1.25, 0.375, 0.8125, 0.59375, 0.703125, 0.6484375, 0.67578125, 0]
9  [3.0, −0.5, 1.25, 0.375, 0.8125, 0.59375, 0.703125, 0.6484375, 0.67578125, 0.662109375
```

Problem A.22. We want to sum the first 100 perfect cubes. Let's do this in two ways.

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

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

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

Problem A.23. 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.   ▲

While Loops

A while loop repeats some task (or sequence of tasks) until a logical condition is met. The structure in Python is the same as with for loops.

Example A.24. Print the numbers 0 through 4 and then the word "done". We'll do this by starting a counter variable, i, at 0 and incrementing it every time we pass through the loop.
Code:

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

Output:

```
1  0
2  1
3  2
4  3
5  4
6  done
```

Example A.25. 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

Code:

```
1  Fib = [1,1]
2  while Fib[-1] <= 1000:
3      Fib.append(Fib[-1] + Fib[-2])
4  Fib
```

Output:

```
1  [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597]
```

Problem A.26. Write a while loop that sums the terms in the Fibonacci sequence until the sum is larger than 1000   ▲

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.27. Code:

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

Output:

```
1  Hello, Alice.  Isn't it a lovely day to learn Python?
```

Code:

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

Output:

```
1  You're not Alice.  Where is Alice?
```

Example A.28. 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.

Code:

```
1  import numpy as np
2  x = np.random.rand(1,1) # get a random 1x1 matrix using numpy
3  x = x[0,0] # pull the entry from the first row, first column of the random matrix
4  if x < 0.5:
5      print(x," is less than a half")
6  else:
7      print(x, "is NOT less than a half")
```

Output:

```
1  0.654697487883 is NOT less than a half
```

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

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.

Example A.29. This is the same code as last time except we will make the decision at 0.33 and 0.67 **Code:**

```
1  import numpy as np
2  x = np.random.rand(1,1) # get a random 1x1 matrix using numpy
3  x = x[0,0] # pull the entry from the first row, first column of the random matrix
4  if x < 0.33:
5      print(x," is less than one third")
6  elif x < 0.67:
7      print(x, "is less than two thirds but greater than or equal to one third")
8  else:
9      print(x, "is greater than or equal to two thirds")
```

Output:

```
1  0.654697487883 is less than two thirds but greater than or equal to one third
```

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

Problem A.30. 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 to 1 so your code should stop when the sequence reaches 1.

▲

A.3.6 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.31. The following code defines the polynomial $f(x) = x^3 + 3x^2 + 3x + 1$ and then evaluates the function at a point $x = 2.3$.
Code:

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

Output:

```
1  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. Other scientific programming languages will allow you to define mathematical functions in this way, but Python will not.

- 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.

Problem A.32. 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 $n = 41$? ▲

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.

Example A.33. 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!
Code:

```
1  def Fact(n):
2      if n==0:
3          return(1)
4      else:
5          return( n*Fact(n-1) ) # we are calling the same 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.
Code:

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

Output:

```
1  [1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

Example A.34. 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.
Code:

```
1   # Define the function
2   def MySeq(xn):
3       if xn <= 0.5:
4           return(2*xn)
5       else:
6           return(2*xn-1)
7
8   # Now build a sequence with this function
9   x = [0.125] # arbitrary starting point
10  for n in range(0,5): # Let's only build the first 5 terms
11      x.append(MySeq(x[-1]))
12  print(x)
```

Output:

```
1   [0.125, 0.25, 0.5, 1.0, 1.0, 1.0]
```

Example A.35. Here is another cool example.

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 two. 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.

Code:

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

Output:

```
1  approximation         exact          abs error
2  1.459090909090909 1.4142135623730951 0.04487734671781385
3  1.414903709997168 1.4142135623730951 0.0006901476240728233
4  1.4142137306897584 1.4142135623730951 1.6831666327377093e-07
5  1.4142135623731051 1.4142135623730951 9.992007221626409e-15
```

Problem A.36. 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.3.7   Lambda Functions

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

The basic idea of a Lambda Function is that we just want to state what the variable is and what the rule is for evaluating the function. This is the most like the way that we write mathematical functions. For example, let's define the mathematical function $f(x) = \sin(x^2) + 3$ in two different ways.

- As a Python function with `def`:

```
1  def f(x):
2      return(np.sin(x**2)+3)
```

- As a Lambda Function:

```
1  f = lambda x: np.sin(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: $f(1.5)$

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

where the result is exactly the floating point number we were interested in: $f(1.5 \approx 3.778073196887921$.

The distinct mathematical advantange 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, but the code for evaluating a Lambda Function is *exactly* what we would write mathematically. Additionally, there is less coding overhead than for defining function with the `def` command.

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

```
1  f = lambda x, y: x**2 + y * np.cos(x*y)
```

If we wanted the value $f(2, \pi)$ we could now write the code
Code:

```
1  f(2,np.pi)
```

Output:

```
1  7.141592653589793
```

Example A.37. You may recall Euler's Method from your differential equations train-
ing. 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 Func-
tion 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 partic-
ular attention to how we use the Lambda Function.

Code:

```
1  RightSide = lambda y: -0.25*y + 2 # define the right-hand side
2  dt = 0.125 # define the delta t in Euler's method
3  t = [0] # initial time
4  y = [1.1] # initial condition
5  for n in range(0,10):
6      t.append(t[n] + dt) # increment the time
7      y.append(y[n] + dt*RightSide(y[n])) # find approx soln at next pt
8  print(t) # print the times
9  print(np.round(y,4)) # round the approx y values to 4 decimal places
```

Output:

```
1  [0,    0.125,  0.25,    0.375,  0.5,     0.625]
2  [1.1, 1.3156, 1.5245, 1.7269, 1.9229, 2.1128]
```

Problem A.38. Go back to Problem A.32 and repeat this exercise using a Lambda Func-
tion instead of a Python function.   ▲

Problem A.39. Go back tp problem A.36 and repeat this exercise using a Lambda Func-
tion instead of a Python function.   ▲

A.3.8 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 dif-
   ferentiation command comming 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 uses the same name for multiple purposes

---

[*]If you are familiar with the statistical programming language, R, then you might be used to the idea of
importing packages or libraries for certain tasks.

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.40 (Importing the `math` Package)**.** The code below imports the `math` package into your instance of Python and calculates the cosine of $\pi/4$.
Code:

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

Output:

```
1  0.7071067811865476
```

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

You might already see a potential disadvantage: 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.41. Here we import the entire math package so we can use every one of the functions therein without having to use the `math` prefix.
Code:

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

Output:

```
1  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.42 (Importing Multiple Packages)**.** 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.
Code:

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

Output:

```
1  0.7071067811865476 0.707106781187
```

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.

Once you have a package imported you can see what is inside of it using the `dir` command.

Example A.43 (Looking Inside a Package)**.** The following block of code prints a list of all of the functions inside the `math` package.
Code:

```
1  import math
2  print(dir(math))
```

Output:

```
1  ['__doc__', '__file__', '__loader__', '__name__', '__package__',
2  '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
3  'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf',
4  'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
5  'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
6  'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
7  'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin',
8  'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

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.

Example A.44 (Help). Here we get help on the arc cosine function in the math package.
Code:

```
1 help(math.acos)
```

Output:

```
1 Help on built-in function acos in module math:
2 acos(...)
3     acos(x)
4     Return the arc cosine (measured in radians) of x.
```

Problem A.45. 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.4 Numerical Python with numpy

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

Example A.46 (Horrible Numerical Operations). 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 as you can see below.
Code:

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

Output:

```
1 TypeError: must be real number, not list
```

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

```
1 SineList = [sin(n) for n in MyList]
2 print(SineList)
```

Output:

```
1 [0.0, 0.49999999999999994, 0.7071067811865475, 0.8660254037844386,
```

```
2 1.0, 0.8660254037844388, 0.7071067811865476, 0.49999999999999994,
3 1.2246467991473532e-16]
```

Code:

```
1 SineList = [ ]
2 for n in range(0,len(MyList)):
3     SineList.append( sin(MyList[n]) )
4 SineList
```

Output:

```
1 [0.0,
2  0.49999999999999994,
3  0.7071067811865475,
4  0.8660254037844386,
5  1.0,
6  0.8660254037844388,
7  0.7071067811865476,
8  0.49999999999999994,
9  1.2246467991473532e-16]
```

The package numpy is used in almost all numerical computations 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. A brief glimpse through the following 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)

Code:

```
1 import numpy as np
2 # Uncomment the next line of code and run it.
3 # print(dir(np))
```

A.4.1   Numpy Arrays, Array Operations, and Matrix Operations

In the previous section you worked with Python lists. As we pointed out, the shortcoming of Python lists is that they 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 example. 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.47 (Defining Matrices and Vectors). The first thing to note is that a matrix is a list of lists (each row is a list).
Code:

```
1  A = np.array([[1,2],[3,4]])
2  print(A)
3  v = np.array([[5],[6]]) # this creates a column vector
4  print(v)
5  w = np.array([5,6]) # this creates a row vector
6  print(w)
```

Output:

```
1  [[1 2]
2   [3 4]]
3  [[5]
4   [6]]
5  [5 6]
```

Example A.48 (Shape Matrices and Vectors). 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,).
Code:

```
1  print(A.shape) # Shape of the matrix A
2  print(v.shape) # Shape of the column vector v
3  print(w.shape) # Shape of the row vector w
```

Output:

```
1  (2, 2)
2  (2, 1)
3  (2,)
```

Example A.49 (Size of Matrices and Vectors)**.** 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.

Code:

```
1  print(A.size) # Size (number of elements) of A
2  print(v.size) # Size (number of elements) of v
3  print(w.size) # Size (number of elements) of w
```

Output:

```
1  4
2  2
3  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.50 (Reading Entries from a Matrix)**.** Let's read the top left and bottom right entries of the matrix *A*.

Code:

```
1  print(A)
2  print(A[0,0]) # top left
3  print(A[1,1]) # bottom right
```

Output:

```
1  [[1 2]
2   [3 4]]
3  1
4  4
```

Example A.51 (Reading a Row from a Matrix)**.** Let's read the first row from that matrix *A*.
Code:

```
1 print(A)
2 print(A[0,:])
```

Output:

```
1 [[1 2]
2  [3 4]]
3 [1 2]
```

Example A.52 (Reading a Column from a Matrix)**.** Let's read the second column from the matrix *A*.
Code:

```
1 print(A)
2 print(A[:,1])
```

Output:

```
1 [[1 2]
2  [3 4]]
3 [2 4]
```

Notice when we read the column it was displayed as a column. Be careful.

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.53 (Element-wise Matrix Multiplication)**.** 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.
Code:

```
1 print(A)
2 A * A # Notice that this is NOT the same as A*A with matrix multiplication
```

Output:

```
1 [[1 2]
2 [3 4]]
3 array([[ 1,  4],
4        [ 9, 16]])
```

Example A.54 (Element-wise Matrix-Vector Multiplication). If we write A * v Python will do element-wise multiplication across each column since v is a column vector.
Code:

```
1  print(A)
2  print(v)
3  A * v # This will actually do element wise multiplication on each column
```

Output:

```
1  [[1 2]
2   [3 4]]
3  [[5]
4   [6]]
5  array([[ 5, 10],
6         [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.55 (Matrix-Matrix and Matrix-Vector Multiplication). Recasting the numpy arrays as matrices allows you to use multiplication as we would expect from linear algebra.
Code:

```
1  A = np.matrix(A)
2  v = np.matrix(v)
3  w = np.matrix(w)
4  print(A*A)
5  print(A*v)
6  print(w*A)
```

Output:

```
1  [[ 7 10]
2   [15 22]]
3  [[17]
4   [39]]
5  [[23 34]]
```

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

Example A.56 (Transpose of a Matrix). 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.).
Code:

```
1  print(A)
2  A.T # The transpose is relatively simple
```

Output:

```
1  [[1 2]
2   [3 4]]
3  array([[1, 3],
4         [2, 4]])
```

Example A.57 (Inverse of a Matrix). The inverse of a square matrix is done with A.I.
Code:

```
1  print(A)
2  Ainv = A.I # Taking the inverse is also pretty simple
3  print(Ainv)
4  print(A * Ainv) # check that we get the identity matrix back
```

Output:

```
1  [[1 2]
2   [3 4]]
3  [[-2.   1. ]
4   [ 1.5 -0.5]]
5  [[  1.00000000e+00    0.00000000e+00]
6   [  8.88178420e-16    1.00000000e+00]]
```

Example A.58 (Determinant of a Matrix). The determinant command is hiding under
the linalg subpackage inside numpy. Therefore we need to call it as such.
Code:

```
1  np.linalg.det(A) # The determinant is hiding inside the linear algebra package under n
```

Output:

```
1  -2.0000000000000004
```

Example A.59 (Trace of a Matrix). The trace is done with matrix.trace()
Code:

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

Output:

```
1  matrix([[5]])
```

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

Problem A.60. Now that we can do some basic linear algebra with 'numpy' it is your turn. Define the matrix B and the vector u as below. 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

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

▲

A.4.2 `arange`, `linspace`, `zeros`, `ones`, and `mgrid`

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` (linear space) 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.

- `mgrid` (mesh grid) is the same as the `meshgrid` command in MATLAB. This builds two arrays that when paired make up the ordered pairs for a 2D (or higher D) mesh grid of points.

Note: Just like with all Python lists, the "stop" number is the one immediately after where you intended to stop.

Example A.61 (Arange). The numpy arange command is great for building sequences.
Code:

```
1  x = np.arange(0,5,0.1)
2  print(x)
```

Output:

```
1  [ 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
2    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
3    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    4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9]
```

Example A.62 (Linspace). The linspace command builds a list with equal (linear) spacing between the starting and ending values.
Code:

```
1  y = np.linspace(0,5,10)
2  print(y)
```

Output:

```
1  [ 0.         0.55555556  1.11111111  1.66666667  2.22222222  2.77777778
2    3.33333333  3.88888889  4.44444444  5.         ]
```

Example A.63 (Zeros). The zeros command builds an array of zeros. This is handy for pre-allocating memory.
Code:

```
1  z = np.zeros((3,5)) # create a 3x5 matrix of zeros
2  z
```

Output:

```
1  array([[ 0.,   0.,   0.,   0.,   0.],
2         [ 0.,   0.,   0.,   0.,   0.],
3         [ 0.,   0.,   0.,   0.,   0.]])
```

Example A.64 (Ones). The ones command builds an array of ones.
Code:

```
1  u = np.ones((3,5)) # create a 3x5 matrix of ones
2  u
```

Output:

```
1  array([[ 1.,   1.,   1.,   1.,   1.],
2         [ 1.,   1.,   1.,   1.,   1.],
```

```
3              [ 1.,   1.,   1.,   1.,   1.]])
```

Example A.65 (Mgrid). The `mgrid` 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.

Code:

```
1  x, y = np.mgrid[0:5, 0:5]
2  print("x = ", x)
3  print("y = ", y)
```

Output:

```
1  x =  [[0 0 0 0 0]
2   [1 1 1 1 1]
3   [2 2 2 2 2]
4   [3 3 3 3 3]
5   [4 4 4 4 4]]
6  y =  [[0 1 2 3 4]
7   [0 1 2 3 4]
8   [0 1 2 3 4]
9   [0 1 2 3 4]
10   [0 1 2 3 4]]
```

Problem A.66.   (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, change the third column to all 7's, and then find the 5th power of this matrix.

▲

A.5   Mathematical 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. The main advantage to `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.5.1 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.

Example A.67 (Plotting with `plt.plot()`)**.** 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`).

Code:

```
1  import numpy as np
2  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.

Code:

```
1  %matplotlib inline
```

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.

Now we'll build a `numpy` array of x values (using the `linspace` command) and a `numpy` array of y values for the sine function.

Code:

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

- Finally, build the plot with `plt.plot()`. This function behaves just like MATLAB's `plot` command:

```
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 (unlike MATLAB).

- 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. You can get by without this line of code, but it is handy to have in there when you're doing complicated plots.

Code:

```
1  plt.plot(x,y, 'green', label='The Sine Function')
2  plt.grid()
3  plt.legend()
4  plt.xlabel("x axis")
5  plt.ylabel("y axis")
6  plt.show()
```

The plot that is produced can be seen in Figure A.1.
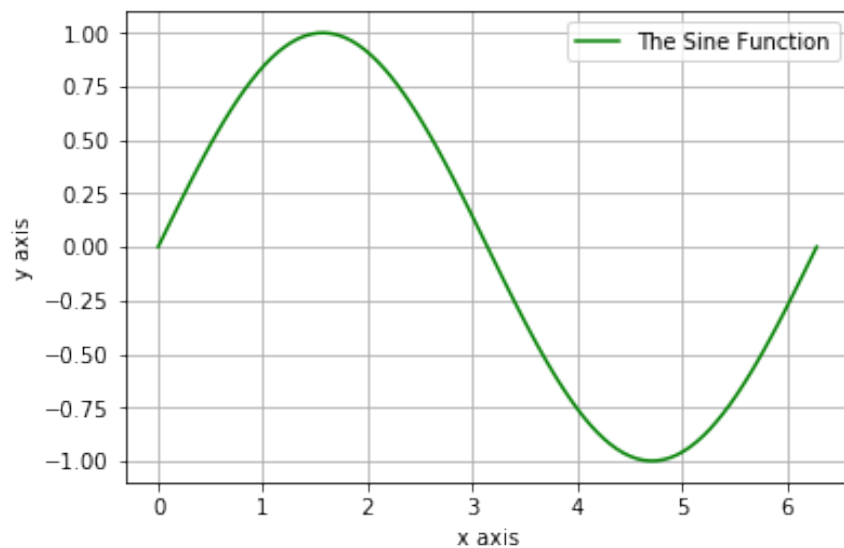
Figure A.1. The sine function on $x \in [0, 2\pi]$, colored greed, with a grid, and with a legend.

Example A.68 (Overlayed Plots with MatPlotLib). 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. 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.

Code:

```python
1  # We really don't need to import the packages again,
2  # but for completeness of the example let's do it anyway.
3  import numpy as np
4  import matplotlib.pyplot as plt
5  %matplotlib inline
6
7  # build the x and y values
8  x = np.linspace(0,1,100)
9  y0 = np.sin(2*np.pi*x)
10 y1 = np.cos(2*np.pi*x)
11 y2 = y0 + y1
12 y3 = y0 - y1
13
14 # plot each of the functions (notice that they will be on the same axes)
15 plt.plot(x, y0, 'b-.', label=r"$y_0 = \sin(2\pi x)$")
16 plt.plot(x, y1, 'r--', label=r"$y_1 = \cos(2\pi x)$")
17 plt.plot(x, y2, 'g:', label=r"$y_2 = y_0 + y_1$")
18 plt.plot(x, y3, 'k-', label=r"$y_3 = y_0 - y_1$")
19
20 # put in a grid, legend, title, and axis labels
21 plt.grid()
22 plt.legend()
23 plt.title("Awesome Title")
24 plt.xlabel('x axis label')
25 plt.ylabel('y axis label')
26 plt.show()
```

The resulting plot can be seen in Figure A.2

Notice in Figure A.2 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.69 (Same Plot, Different Code). Now let's create the same plot as in Figure A.2 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
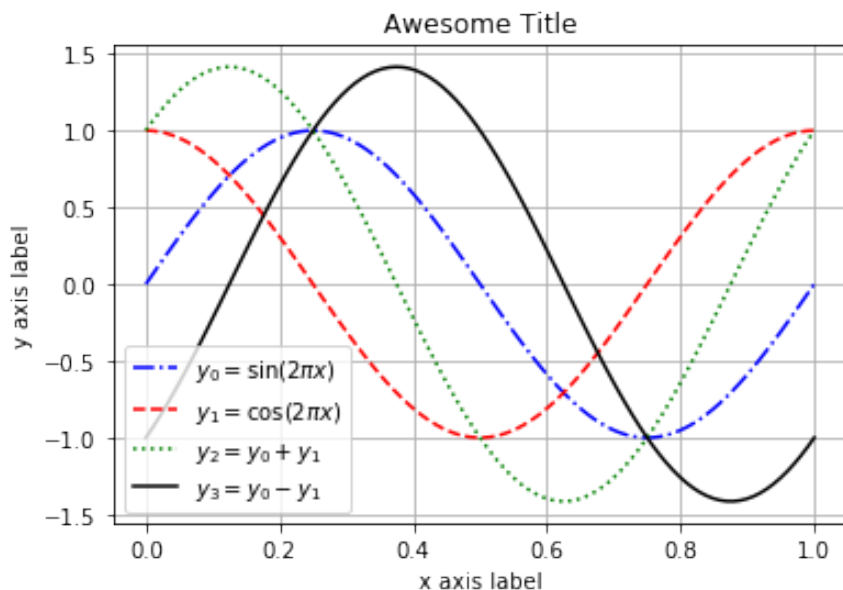
Figure A.2. Several trigonometric functions on the same axes.

do when plotting several functions on top of each other.
Code:

```
1  # The next line of code does all of the plotting of all of the functions.
2  # Notice the order: x, y, color and line style, repeat
3  plt.plot(x, y0, 'b-.', x, y1, 'r--', x, y2, 'g:', x, y3, 'k-')
4
5  plt.grid()
6  plt.legend([r"$y_0 = \sin(2\pi x)$",r"$y_1 = \cos(2\pi x)$",\
7              r"$y_2 = y_0 + y_1$",r"$y_3 = y_0 - y_1$"])
8  plt.title("Awesome Title")
9  plt.xlabel('x axis label')
10 plt.ylabel('y axis label')
11 plt.show()
```

Problem A.70. 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.5.2 Subplots

It is often very handy to place plots side-by-side or as some array of plots. The subplots command allows us that control. The main idea is that we are setting up a matrix of blank plot and then populating the axes with the plots that we want.

Example A.71 (First Subplot Example). Let's repeat the previous exercise (which produced Figure A.2), 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 high-light 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.

Code:

```
1  # set up the blank matrix of plots
2  fig, axes = plt.subplots(nrows = 2, ncols = 2, figsize = (10,5))
3
4  # Build the first plot
5  axes[0,0].plot(x, y0, 'b-.')
6  axes[0,0].grid()
7  axes[0,0].set_title(r"$y_0 = \sin(2\pi x)$")
8  axes[0,0].set_ylim(-2,2)
9  axes[0,0].set_xlabel("x")
10 axes[0,0].set_ylabel("y")
11
12 # Build the second plot
13 axes[0,1].plot(x, y1, 'r--')
14 axes[0,1].grid()
15 axes[0,1].set_title(r"$y_1 = \cos(2\pi x)$")
16 axes[0,1].set_ylim(-2,2)
17 axes[0,1].set_xlabel("x")
18 axes[0,1].set_ylabel("y")
19
20 # Build the first plot
21 axes[1,0].plot(x, y2, 'g:')
22 axes[1,0].grid()
23 axes[1,0].set_title(r"$y_2 = y_0 + y_1$")
24 axes[1,0].set_ylim(-2,2)
25 axes[1,0].set_xlabel("x")
26 axes[1,0].set_ylabel("y")
27
28 # Build the first plot
29 axes[1,1].plot(x, y3, 'k-')
30 axes[1,1].grid()
31 axes[1,1].set_title(r"$y_3 = y_0 - y_1$")
```

```
32  axes[1,1].set_ylim(-2,2)
33  axes[1,1].set_xlabel("x")
34  axes[1,1].set_ylabel("y")
35
36  fig.tight_layout()
```

The resulting plot can be seen in Figure A.3. The `fig.tight_layout()` command makes the plot labels a bit more readable in this instance (again, something you can play with).
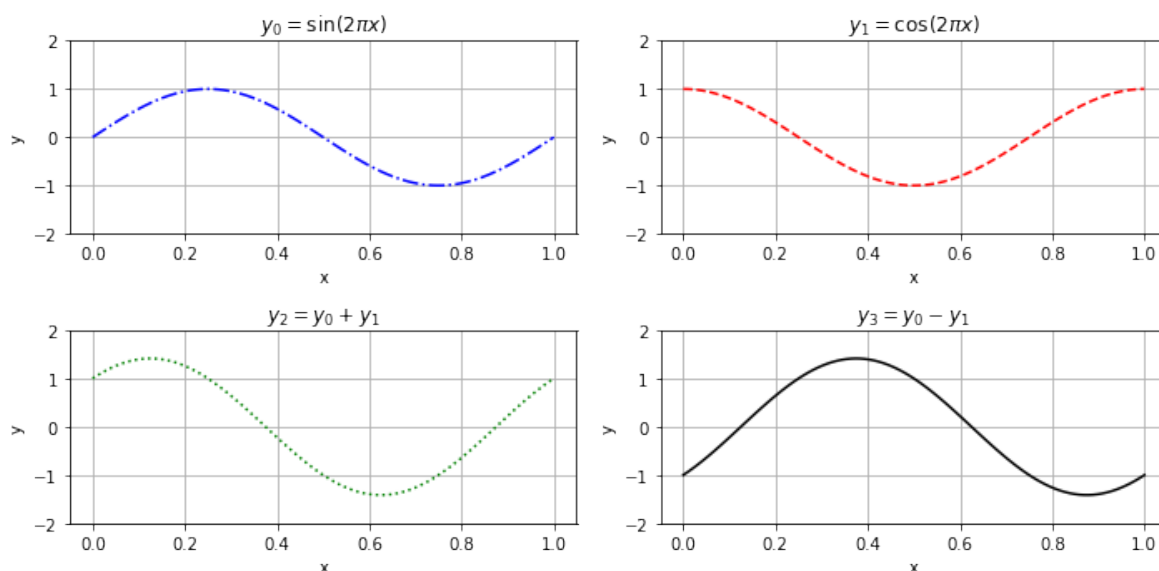

Figure A.3. The trigonometric functions with a 2×2 matrix of subplots.

Problem A.72. 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.5.3 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.

Example A.73. 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 $[0, 500]$ as a domain.
Code:

```
1  x = np.linspace(0,500,1000)
2  y = 10**(-0.01*x)
3  fig, axis = plt.subplots(1,2, figsize = (10,5))
4
5  axis[0].plot(x,y, 'r')
6  axis[0].grid()
7  axis[0].set_title("Linearly scaled y axis")
8  axis[0].set_xlabel("x")
9  axis[0].set_ylabel("y")
10
11 axis[1].semilogy(x,y, 'k--')
12 axis[1].grid()
13 axis[1].set_title("Logarithmically scaled y axis")
14 axis[1].set_xlabel("x")
15 axis[1].set_ylabel("Log(y)")
```

The output for this code can be seen in Figure A.4.

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.

Code:

```
1  x = np.linspace(0,500,1000)
2  y = 10**(-0.01*x)
3  fig, axis = plt.subplots(1,2, figsize = (10,5))
4
5  axis[0].plot(x,y, 'r')
6  axis[0].grid()
7  axis[0].set_title("Linearly scaled y axis")
8  axis[0].set_xlabel("x")
9  axis[0].set_ylabel("y")
10
11 axis[1].plot(x,y, 'k--') # <----- Notice the change here
12 axis[1].set_yscale("log") # <----- And we added this line
13 axis[1].grid()
14 axis[1].set_title("Logarithmically scaled y axis")
15 axis[1].set_xlabel("x")
16 axis[1].set_ylabel("Log(y)")
```

Problem A.74. 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.6 Symbolic Algebra and Calculus 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
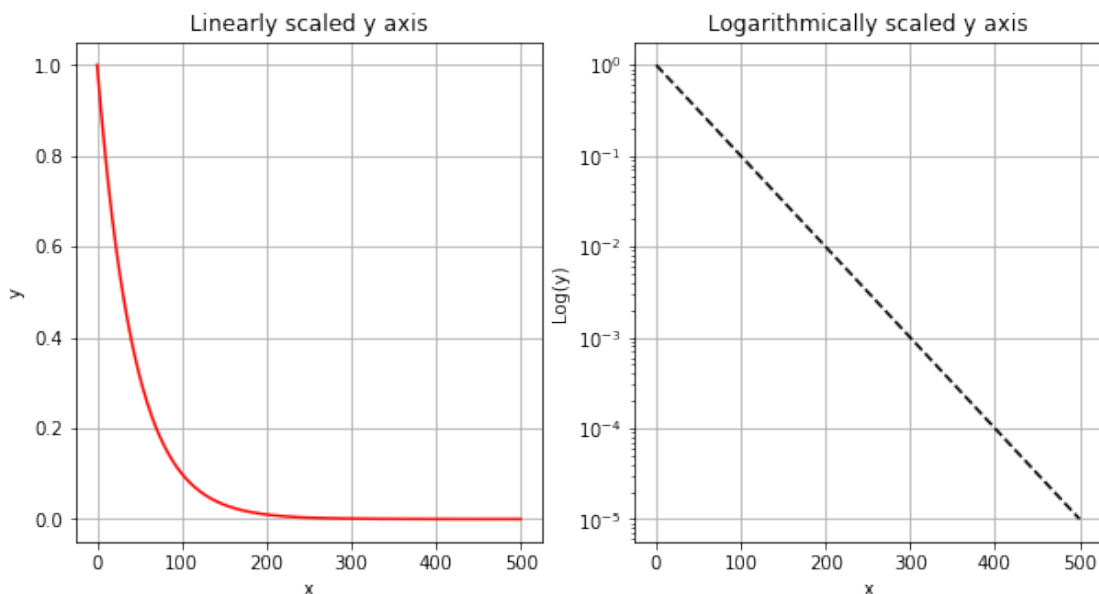
Figure A.4. The left-hand plot shows the exponential function $y = 10^{-0.01x}$ and the right-hand side shows the same plot with the y axis scaled logarithmically.

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 . . .

For the sake of convenience we will load matplotlib so we can plot a few things along the way.

Code:

```
1  import matplotlib.pyplot as plt
2  %matplotlib inline
```

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.

Code:

```
1  import sympy as sp
2  # The directory of sympy is full of stuff.
3  # Uncomment the next line and run the code to see what you're getting.
4  # print(dir(sp))
5
6  sp.init_printing() # This allows for a more LaTeX style output
```

The last line of the previous block of code is again just for Jupyter Notebooks. We want to allow for some beautiful styling of the output of these symbolic calculations and this is the line that allows Jupyter to do it.

A.6.1 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.

Let's define the variable x as a symbolic variable. Then we'll define a few symbolic expressions that use x as a variable.

Code:

```
1  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.

Code:

```
1  f = (x+2)**3 # A symbolic function
2  f
```

The output of these lines of code is the nicely formatted symbolic expression $(x+2)^3$.

Be careful that you are using symbolically defined function along with your symbols. For example, see the code below:

Code:

```
1  # g = sin(x) # this line gives an error since it doesn't know which "sine" function to u
2  g = sp.sin(x) # this one works
3  g
```

The output of these lines of code is the symbolic expression $\sin(x)$.

A.6.2 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.75 (Expand a Polynomial)**.** 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.

Code:

```
1  f = (x+2)**3
2  sp.expand(f) # do the multiplication to expand the polynomial
```

The output of these lines of code is the expression $x^3 + 6x^2 + 12x + 8$.

Example A.76 (Factor a Polynomial)**.** We will factor the polynomial $h(x) = x^2 + 4x + 3$.

Code:

```
1  h = x**2 + 4*x + 3
```

```
2  sp.factor(h) # factor this polynomial
```

The output of this function is the expression $(x+1)(x+3)$.

Example A.77 (Trig Expansion). The sympy package knows how to work with trigonometric identities. In this example we show how sympy expands $\sin(a+b)$.
Code:

```
1  a, b = sp.symbols('a b')
2  j = sp.sin(a+b)
3  sp.expand(j, trig=True) # Trig identities are built in!
```

The output of these linse of code is the expression $\sin(a)\cos(b) + \sin(b)\cos(a)$

Example A.78 (Simplifying Algebraic Expressions). In this example we will simplify the function $g(x) = x^3 + 5x^3 + 12x^2 + 1$.
Code:

```
1  g = x**3 + 5*x**3 + 12*x**2 + 1
2  sp.simplify(g) # Simplify some algebraic expression
```

The output of these lines of code is the expression $6x^3 + 12x^2 + 1$.

Example A.79 (Simplifying Trig Expressions). In this example we'll simplify an expression that involves trigonometry.
Code:

```
1  sp.simplify( sp.sin(x) / sp.cos(x)) # simplify a trig expression.
```

The output of this line of code is the expression $\tan(x)$ as expected.

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 the following example gives a simple example of how it works in sympy.

Example A.80 (Simple Equation Solving). We want to solve the equation $x^2 + 4x + 3 = 0$ for x.
Code:

```
1  h = x**2 + 4*x + 3
2  sp.solve(h,x)
```

Output:

```
1  [−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.6.3   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 symolic 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.

The following code is what the mathematicians in us would like to do: **Code:**

```
1  f = (x+2)**3
2  f(5) # This gives an error!
```

...but this is how it should be done: **Code:**

```
1  f.subs(x,5) # This actually substitutes 5 for x in f
```

Output:

```
1  343
```

A.6.4   Symbolic Calculus

The `sympy` package has routines to take symbolic derivatives, antiderivatives, limits, and Taylor series just like other computer algebra systems.

Derivatives

The `diff` command in `sympy` does differentiation.

```
1  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.81 (Symbolic Differentiation)**.** In this example we'll differentiate the function $f(x) = (x+2)^3$.
Code:

```
1  import sympy as sp # we already did this above, but let's do it again for clarity
2  x = sp.Symbol('x') # Define the symbol x
3  f = (x+2)**3 # Define a symbolic function f(x) = (x+2)^3
4  df = sp.diff(f,x) # Take the derivative of f and call it "df"
5  print("f(x) = ", f)
```

```
6  print("f'(x) = ",df)
7  print("f'(x) = ", sp.expand(df))
```

Output:

```
1  f(x) =   (x + 2)**3
2  f'(x) =   3*(x + 2)**2
3  f'(x) =   3*x**2 + 12*x + 12
```

Now let's get the first, second, third, and fourth derivatives of the function f.

Code:

```
1  df = sp.diff(f,x,1) # first derivative
2  ddf = sp.diff(f,x,2) # second deriative
3  dddf = sp.diff(f,x,3) # third deriative
4  ddddf = sp.diff(f,x,4) # fourth deriative
5  print("f'(x) = ",df)
6  print("f''(x) = ",sp.simplify(ddf))
7  print("f'''(x) = ",sp.simplify(dddf))
8  print("f''''(x) = ",sp.simplify(ddddf))
```

Output:

```
1  f'(x) =   3*(x + 2)**2
2  f''(x) =   6*x + 12
3  f'''(x) =   6
4  f''''(x) =   0
```

Example A.82 (Partial Derivatives). 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.

Code:

```
1  x, y = sp.symbols('x y') # Define the symbols
2  f = sp.sin(x*y) + sp.cos(x**2) # Define the function
3  fx = sp.diff(f,x)
4  fy = sp.diff(f,y)
5  print("f(x,y) = ", f)
6  print("f_x(x,y) = ", fx)
7  print("f_y(x,y) = ", fy)
```

Output:

```
1  f(x,y) =   sin(x*y) + cos(x**2)
2  f_x(x,y) =   -2*x*sin(x**2) + y*cos(x*y)
3  f_y(x,y) =   x*cos(x*y)
```

Example A.83 (LATEX from Symbolic Expressions). 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.

Code:

```
1  sp.latex(f)
```

Output:

```
1  '\\sin{\\left(x y \\right)} + \\cos{\\left(x^{2} \\right)}'
```

Integrals

For integration, the `integrate` tool is the command for the job

```
1  sp.integrate(function, variable) # antiderivative
2  sp.integrate(function, (variable, lower, upper)) # 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.84 (An Antiderivative). Find the antiderivative of the function $f(x) = (x+2)^3$.

Code:

```
1  import sympy as sp
2  x = sp.Symbol('x')
3  f = (x+2)**3
4  F = sp.integrate(f,x)
5  F
```

The output of these lines of code is the expression $\frac{x^4}{4} + 2x^3 + 6x^2 + 8x$ which is indeed the antiderivative. We can ask sympy to factor the result with

Code:

```
1  sp.factor(F)
```

which results in the expression $\dfrac{x(x+4)\left(x^2 + 4x + 8\right)}{4}$

Example A.85 (Another Antiderivative). Consider the multivariable antiderivative

$$\int \sin(xy) + \cos(x) dx.$$

The sympy package deals with the second variable just as it should.

Code:

```
1  g = sp.sin(x*y) + sp.cos(x)
2  G = sp.integrate(g,x)
```

```
3  G
```

These lines of code result in the output

$$\begin{cases} -\frac{\cos(xy)}{y} & \text{for } y \neq 0 \\ 0 & \text{otherwise} \end{cases} + \sin(x)$$

so it is apparent that sympy was sensitive to the fact that there was trouble at $y = 0$ and took care of it with a piecewise function.

Example A.86 (A Definite Integral). 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).
Code:

```
1  sp.integrate( sp.sin(x), (x,0,sp.pi))
```

This line of code results in 2; the correct definite integral.

Example A.87 (A Tougher Definite Integral). 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.
Code:

```
1  sp.integrate( sp.exp(-x**2) , (x, -sp.oo, sp.oo))
```

which results in the value $\sqrt{\pi}$ ... which is simply a fun integral.

Limits

The limit command in sympy takes symbolic limits

```
1  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.88 (Symbolic Limit). Let's take the limit

$$\lim_{x \to 0} \frac{\sin(x)}{x}.$$

Code:

```
1  sp.limit( sp.sin(x)/x, x, 0)
```

which gives us the value 1.

Example A.89 (Difference Quotient). 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!

Code:

```
1  f = (x+2)**3
2  print(sp.diff(f,x))
3  h = sp.Symbol('h')
4  df = sp.limit( (f.subs(x,x+h) - f) / h , h , 0 )
5  print(df)
6  print(df == sp.diff(f,x)) # notice that these are not "symbolically" equatl
7  print(df == sp.expand(sp.diff(f,x))) # but these are
```

Output:

```
1  3*(x + 2)**2
2  3*x**2 + 12*x + 12
3  False
4  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.

Problem A.90. Define the function $f(x) = 3x^2 + x\sin(x^2)$ symbolically and then do the following:

1. Evaluate the function at $x = 2$ and get symbolic and numerical answers.

2. Take the first and second derivative

3. Take the antiderivative

4. Find the definite integral from 0 to 1

5. Find the limit as x goes to 3

▲

Taylor Series

The `sympy` package has a tool for expanding Taylor Series of symbolic functions.

```
1  sp.series( function, variable, [center], [num terms])
```

The center defaults to 0 and the number of terms defaults to 5.

Example A.91 (A Taylor Series Expansion)**.** Find the Taylor series for $f(x) = e^x$ centered at $x = 0$ and centered at $x = 1$.
Code:

```
1  sp.series( sp.exp(x),x)
```

gives the output $1 + x + \dfrac{x^2}{2} + \dfrac{x^3}{6} + \dfrac{x^4}{24} + \dfrac{x^5}{120} + \mathcal{O}(x^6)$.
Code:

```
1  sp.series( sp.exp(x), x, 1) # expand at x=1
```

gives the output

$$e + e(x-1) + \frac{e(x-1)^2}{2} + \frac{e(x-1)^3}{6} + \frac{e(x-1)^4}{24} + \frac{e(x-1)^5}{120} + \mathcal{O}\left((x-1)^6; x \to 1\right)$$

Finally, if we want more terms then we can send the number of desired terms to the `series` command.
Code:

```
1  sp.series( sp.exp(x), x, 0, 10) # expand at x=0 and give 10 terms
```

gives the output

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \frac{x^8}{40320} + \frac{x^9}{362880} + \mathcal{O}\left(x^{10}\right).$$

A.6.5 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.

```
1  sp.solve( equation, variable )
```

The equation doesn't actually need to be the whole equation. For any equation-solving problem we can always reframe 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.92 (Solve Equation Symbolically). 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.
Code:

```
1  sp.solve( x**2 - 2, x)
```

gives the output $\left[-\sqrt{2}, \sqrt{2}\right]$.

Example A.93 (A Slightly More Challenging Equation Solving Problem). 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!)
Code:

```
1  sp.solve( x**4 - x**2 - 1, x)
```

which gives the output

$$\left[-i\sqrt{-\frac{1}{2}+\frac{\sqrt{5}}{2}},\ i\sqrt{-\frac{1}{2}+\frac{\sqrt{5}}{2}},\ -\sqrt{\frac{1}{2}+\frac{\sqrt{5}}{2}},\ \sqrt{\frac{1}{2}+\frac{\sqrt{5}}{2}}\right]$$

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.94 (Converting Symbolic Answers to Floating Point). 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.
Code:

```
1  soln = sp.solve( x**4 - x**2 - 1, x)
2  for j in range(0, len(soln)):
3      print(sp.N(soln[j]))
```

Output:

```
1  -0.786151377757423*I
2  0.786151377757423*I
```

```
3  -1.27201964951407
4  1.27201964951407
```

The `N` command gives a numerical approximation for a symbolic expression (this is taken straight from Mathematica!).

Problem A.95. 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.

```
1  sp.nsolve( equation, variable, intial guess )
```

Example A.96 (Numerical Solve on a Symbolically Defined Equation). 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 $x = 1$.

Code:

```
1  sp.solve(x**3 - x**2 - 2, x, 1) # symbolic solution
```

gives the exact solution

$$
\left[\frac{1}{3} + \left(-\frac{1}{2} - \frac{\sqrt{3}i}{2} \right) \sqrt[3]{\frac{\sqrt{87}}{9} + \frac{28}{27}} + \frac{1}{9\left(-\frac{1}{2} - \frac{\sqrt{3}i}{2} \right)\sqrt[3]{\frac{\sqrt{87}}{9} + \frac{28}{27}}} , \right.
$$

$$
\frac{1}{3} + \frac{1}{9\left(-\frac{1}{2} + \frac{\sqrt{3}i}{2} \right)\sqrt[3]{\frac{\sqrt{87}}{9} + \frac{28}{27}}} + \left(-\frac{1}{2} + \frac{\sqrt{3}i}{2} \right) \sqrt[3]{\frac{\sqrt{87}}{9} + \frac{28}{27}} ,
$$

$$
\left. \frac{1}{9\sqrt[3]{\frac{\sqrt{87}}{9} + \frac{28}{27}}} + \frac{1}{3} + \sqrt[3]{\frac{\sqrt{87}}{9} + \frac{28}{27}} \right]
$$

which is rather challenging to read. We can give all of the floating point approximations with the following code.

Code:

```
1  print("First Solution: ",sp.N(ExactSoln[0]))
2  print("Second Solution: ",sp.N(ExactSoln[1]))
3  print("Third Solution: ",sp.N(ExactSoln[2]))
```

Output:

```
1  First Solution:  -0.347810384779931 - 1.02885225413669*I
```

```
2  Second Solution:   -0.347810384779931 + 1.02885225413669*I
3  Third Solution:   1.69562076955986
```

If we were only looking for the floating point real solution near $x = 1$ then we could just use nsolve.

Code:

```
1  sp.nsolve(x**3 - x**2 - 2, x, 1)
```

Output:

```
1  1.69562076955986
```

Problem A.97. Solve the equation

$$x^3 \ln(x) = 7$$

and give your answer both symbolically and numerically.   ▲

A.6.6   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.

```
1  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.

In numerical analysis we don't 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.98 (A Symbolic Plot)**.** Let's get a quick plot of the function $f(x) = (x + 2)^3$ on the domain $x \in [-5, 2]$.

Code:

```
1  import sympy as sp
2  x = sp.Symbol('x')
3  f = (x+2)**3
4  sp.plot(f,(x,-5,2))
```

The output for which can be seen in Figure A.5.

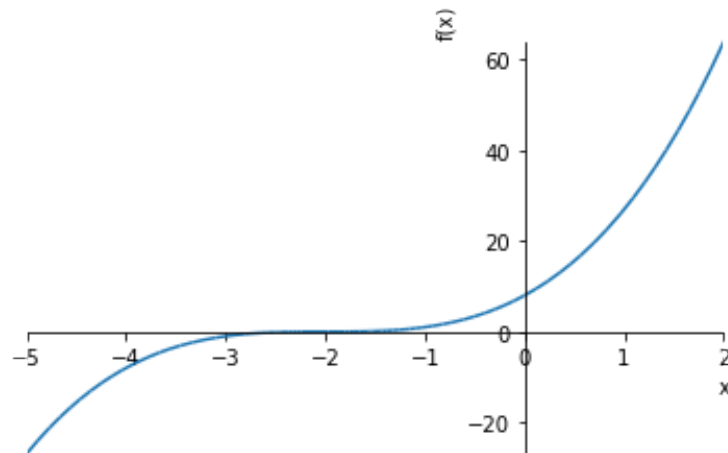Multiple plots can be done at the same time with the sympy.plot command.

Figure A.5. A plot of $f(x) = (x+2)^3$ on the domain $x \in [-5, 2]$ created with `sympy.plot`.

Example A.99. Plot $f(x) = (x+2)^3$ and $g(x) = 20\cos(x)$ on the same axes on the domain $x \in [-5, 2]$.
Code:

```
1  import sympy as sp
2  x = sp.Symbol('x')
3  f = (x+2)**3
4  g = 20*sp.cos(x)
5  sp.plot(f, g, (x,-5,2))
```
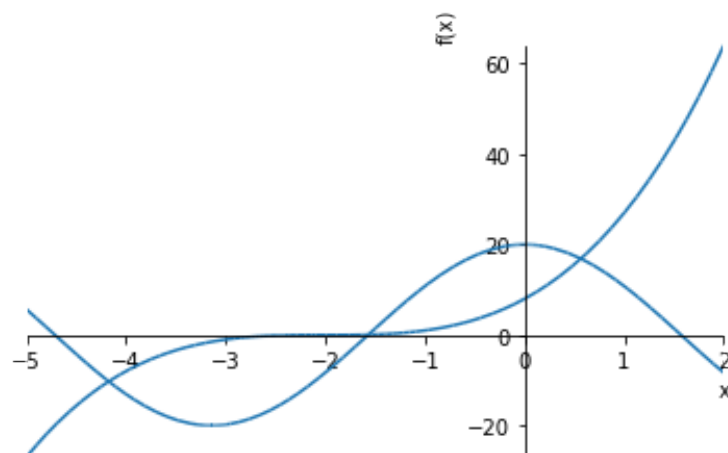
The output for which can be seen in Figure A.6

Figure A.6. Two plots on top of eachother created with `sympy.plot`.

Problem A.100. Make a symbolic plot of the function $f(x) = x^3 \ln(x) - 7$ on the domain $[0, 3]$. ▲

Appendix B

Writing and Projects

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 Mathematics

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 $3x = 6$:

$$\textbf{Incorrect!} \qquad 3x = 6 = \underbrace{\frac{3x}{3}}_{\text{false!}} = \frac{6}{3} = x = 2$$

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 $x = 2$.

"$\implies$" means "implies": The double arrow "$\implies$" 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.

$$\textbf{Incorrect!} \qquad 20 \cdot 0.07 \underbrace{=}_{false!} 1.4 + 20 \underbrace{=}_{false!} \$21.4.$$

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 $\overline{G}$ 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 $\overline{G}$ 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

$$f(x) = \sum_{j=1}^{\infty} \frac{x^n}{n!}. \tag{B.1}$$

In (B.1) 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 youre 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

$$S = \left| \left(\frac{\Delta x}{x} \right) \middle/ \left(\frac{\Delta k}{k} \right) \right|.$$

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.6   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[*]

$$S = \left| \left(\frac{T_d(k-\delta) - T_d(k)}{\delta} \right) \cdot \left(\frac{k}{T_d(k)} \right) \right|$$

$$= \left| \left(\frac{\frac{\ln(2)}{k-\delta} - \frac{\ln(2)}{k}}{\delta} \right) \cdot \left(\frac{k}{\frac{\ln(2)}{k}} \right) \right|$$

$$= \left| \left(\frac{k\ln(2) - (k-\delta)\ln(2)}{\delta k (k-\delta)} \right) \cdot \left(\frac{k^2}{\ln(2)} \right) \right|$$

$$= \left| \frac{k}{k-\delta} \right|$$

$$\approx \frac{0.009}{0.009 - 1.96 \times 0.001} = \frac{0.009}{0.00704} \approx 1.2784.$$

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*.

---

[*]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

LaTeX

If you want your mathematical writing to look like real math then the modern mathematician uses LaTeX for all of their writing.

LaTeX is a powerful mathematical typesetting language. We encourage students to use a tool such as Overleaf (https://www.overleaf.com/) to do their LaTeX editing. Overleaf is an online platform much like Google Docs where you can write and share dynamically in real time. In this appendix we give the basics of mathematical writing with LaTeX.

C.1 Equation Environments and Cross Referencing

When working with equations is it often times convenient and necessary to cross-reference the equations that you're talking about. A simple example is:

Example C.1. Recall the Pythagorean Theorem: If a and b are the legs of a right triangle and c is the hypotenuse, then

$$a^2 + b^2 = c^2. \tag{C.1}$$

Let $a = 3$ and $b = 4$ in equation (C.1). If that is the case then ...
The LaTeX code for this is

```
Recall the Pythagorean Theorem: If $a$ and $b$ are the legs of a
right triangle and $c$ is the hypotenuse, then
\begin{flalign}
    a^2 + b^2 = c^2
    \label{eqn:pythag}
\end{flalign}

Let $a=3$ and $b=4$ in equation \eqref{eqn:pythag}.
If that is the case then \dots
```

Note in Example C.1 that the equations and the equation reference are part of the sentence. In fact, these are always part of the grammatical structure of your writing.

Other numbered environments include `align`, `flalign`, `eqnarray`, `equation` and several others. The modern convention for LATEX is to use `align` or `flalign` for all equations. If you want to use one of these environments without numbers then use the *. In otherwords `align*` will align in the same way without numbering the equations. If you only want a number on one line then you can use `\notag` at the beginning of that line.

To align equations use the "align" enviroment, which requires the amsmath package. Align supersedes eqnarray. The ampersands control the vertical alignment:

```
\begin{align}
\frac{\partial{x}}{\partial{s}}&=zx  & \text{with} && x(0)=1\\
\frac{\partial{y}}{\partial{s}}&=x^2y & \text{with} && y(0)=t\\
\frac{\partial{z}}{\partial{s}}&=xyz & \text{with} && z(0)=t^2
\end{align}
```

$$\frac{\partial x}{\partial s} = zx \qquad \text{with} \qquad x(0) = 1 \tag{C.2}$$

$$\frac{\partial y}{\partial s} = x^2 y \qquad \text{with} \qquad y(0) = t \tag{C.3}$$

$$\frac{\partial z}{\partial s} = xyz \qquad \text{with} \qquad z(0) = t^2 \tag{C.4}$$

A few more math-related typesetting examples are included below:

- Inline math with and without numbering

```
\[ \sum_{j=1}^\infty \frac{1}{j^2} = \frac{\pi^2}{6} \]
```

$$\sum_{j=1}^{\infty} \frac{1}{j^2} = \frac{\pi^2}{6}$$

```
\begin{flalign}
    \sum_{j=1}^\infty \frac{1}{j^2} = \frac{\pi^2}{6}
    \label{eqn:sample_equation}
\end{flalign}
```

$$\sum_{j=1}^{\infty} \frac{1}{j^2} = \frac{\pi^2}{6} \tag{C.5}$$

```
\begin{subequations}
    \begin{eqnarray}
        \sin \left( \frac{\pi}{6} \right) &=& \frac{\sqrt{3}}{2}
        \label{eqn:sine} \\
        \cos \left( \frac{\pi}{6} \right) &=& \frac{1}{2}
        \label{eqn:cosine}
    \end{eqnarray}
    \label{eqn:trig}
\end{subequations}
```

$$\sin\left(\frac{\pi}{6}\right) \;=\; \frac{\sqrt{3}}{2} \tag{C.6a}$$

$$\cos\left(\frac{\pi}{6}\right) \;=\; \frac{1}{2} \tag{C.6b}$$

This second example allows you to cross reference equations like (C.5) using (\ref{eqn:sample_equation}) or, more simply, (C.5) using \eqref{eqn:sample_equation}. The third set of equations allows for multiple types of references. Like: The sine equation, (C.6a) (\eqref{eqn:sine}), and the cosine equation, (C.6b) (\eqref{eqn:cosine}), are grouped together via equation (C.6) (\eqref{eqn:trig}).

- Matrices

```
\[ \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \]
```
$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

```
\[ \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \]
```
$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

```
\[ \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} \]
```
$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix}$

- Including basic graphics
  Be sure that the graphics file is in the same directory as your TeX file. Your picture should be a *.eps or *.pdf file. If not then some TeX compilers will complain (plus, *.jpg usually looks horrible).

```
\begin{center}
    \includegraphics[width=0.9\columnwidth]{filename.eps}
\end{center}
```

OR

```
\begin{center}
    \includegraphics[height=3in]{filename.eps}
\end{center}
```

There are many options for \includegraphics, but these two work for many pictures. Sometimes, though, it is desired to trim an image that you've saved from elsewhere. The basic syntax for trim and clip is

```
\begin{center}
    \includegraphics[trim = 1cm 2cm 1cm 2cm, clip=true,
        width=0.9\columnwidth]{filename.eps}
\end{center}
```

The four measurements after the trim command are the amount to trim from the left, bottom, right, and top (in that order).

- Leaving white space:

  - horizontal Spacing: \hspace{0.5in}
  - vertical Spacing: \vspace{2in}

C.2 Tables, Tabular, Figures, Shortcuts, and Other Environments

C.2.1 Tables and Tabular Environments

Tables can be rather annoying in LATEX, but it is important to get the basics down before moving on.

Example C.2. In this example we want the table to be place *here* [h*], the first column left justified, the middle column centered, and the last column right justified with vertical bars between each column.

```
\begin{center}
    \begin{tabular}[h*]{|l|c|r|}
        \hline
        Title 1 & Title 2 & Title 3 \\ \hline \hline
        Hello & Ni Hao & Bonjour \\ \hline
        good bye & zia jian & adieux \\ \hline
    \end{tabular}
\end{center}
```

Title 1	Title 2	Title 3
Hello	Ni Hao	Bonjour
good bye	zia jian	adieux

Title 1	Title 2	Title 3
Hello	Ni Hao	Bonjour
good bye	zia jian	adieux

Table C.1. This is the amazing table of doom

In Example C.2 we used the tabular environment. This builds the table. If you want to build a table where there is a caption and the environment *floats* to various parts of the page then you need to use the table command.

Example C.3. In this example we build the same table as in Example C.2 but this time we allow it to float and we want a caption. The code is:

```
\begin{table}
    \centering
        \begin{tabular}[h*]{|l|c|r|}
            \hline
            Title 1 & Title 2 & Title 3 \\ \hline \hline
            Hello & Ni Hao & Bonjour \\ \hline
            good bye & zia jian & adieux \\ \hline
        \end{tabular}
    \caption{This is the amazing table of doom}
    \label{tab:MyLabel}
\end{table}
```

C.2.2 Excel To LATEX

One tool that is often overlooked is the ExcelToLaTeX macro for Excel. I'm leaving this one up to you. Google ExcelToLaTeX , download it, add it to the macros for your version of Excel, and have fun with it. This tool will allow you to convert Excel-based tables to LATEX tables.

C.2.3 Figures

The figure environment in LATEX is almost identical to that for table. For example:

Example C.4. This figure simply shows a Python plot of the sine and cosine functions together in all of their shared glory. The file type was eps, which is notoriously hard to handle on Windows machines and on Overleaf. Be sure to use the epstopdf package if you're using epsfile types.

```
\begin{figure}[ht!]
```

```
    \centering
    \includegraphics[width=0.7\columnwidth]{SampleFigure.eps}
    \caption{Figure for Example \ref{ex:C3:fig}
    \label{fig:C3:fig}
\end{figure}
```
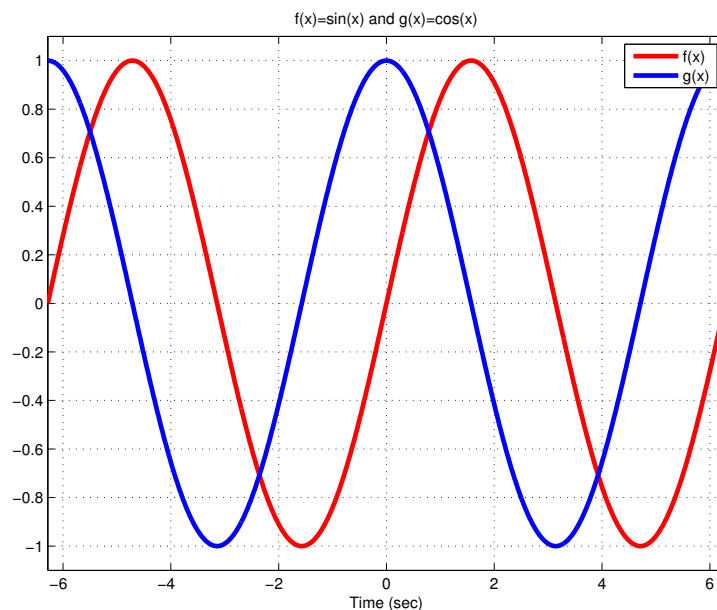

Figure C.1. Figure for Example C.4

C.2.4   New Commands: Shortcuts are AWESOME!

You can save yourself a vast amount of typing by defining new commands which meet your specific need. It is easy. The newcommand command goes in the preamble (before the \begin{document}). The examples that follow are a few handy ones that I've used in the past. The world is your oyster here, so make any shortcut for a LATEX command that is cumbersome to type.

- Derivatives

```
\newcommand{\dd}[2] {\frac{d #1}{d #2}}
\newcommand{\ddd}[2] {\frac{d^2 #1}{d #2^2}}

\dd{y}{x}
\ddd{y}{x}
```

The results are:

$$\frac{dy}{dx}$$

$$\frac{d^2y}{dx^2}$$

- Partial derivatives

```
\newcommand{\pd}[2] {\frac{\partial{#1}}{\partial{#2}}}
\newcommand{\pdd}[2] {\frac{\partial^2{#1}}{\partial{#2}^2}}
\newcommand{\pddm}[3] {\frac{\partial^2{#1}}{\partial{#2}\partial{#3}}}

\pd{y}{x}
\pdd{y}{x}
\pddm{y}{x}{z}
```

The results are:

$$\frac{\partial y}{\partial x}$$

$$\frac{\partial^2 y}{\partial x^2}$$

$$\frac{\partial^2 y}{\partial x \partial z}$$

- Some of the common number sets

```
\newcommand{\cc}{\mathbb{C}}
\newcommand{\rr}{\mathbb{R}}
\newcommand{\nn}{\mathbb{N}}
\newcommand{\qq}{\mathbb{Q}}
\newcommand{\zz}{\mathbb{Z}}

\cc \hspace{1cm} \nn \hspace{1cm} \qq \hspace{1cm} \rr \hspace{1cm} \zz
```

The result is:

$$\mathbb{C} \qquad \mathbb{N} \qquad \mathbb{Q} \qquad \mathbb{R} \qquad \mathbb{Z}$$

- Grouping symbols (parentheses, brackets, etc)

```
\newcommand{\lp}{\left(}
\newcommand{\rp}{\right)}
\newcommand{\lb}{\left[}
\newcommand{\rb}{\right]}
```

```
\lp\pdd{F}{x}\rp
compared to
(\pdd{F}{x})
```

The result is:

$$\left(\frac{\partial^2 F}{\partial x^2}\right)$$

compared to

$$(\frac{\partial^2 F}{\partial x^2})$$

- Common conjunctions

```
\newcommand{\andd}[1]{\quad\text{and}\quad}
\newcommand{\orr}[1]{\quad\text{or}\quad}
\newcommand{\forr}[1]{\quad\text{for}\quad}
\newcommand{\st}[1]{\quad\text{such that}\quad}
\newcommand{\conj}[1]{\quad\text{#1}\quad}

% Implies
\newcommand{\ra}{\quad\Rightarrow\quad}

A\ra B\st C\ne D
```

The result is:

$$A \quad \Rightarrow \quad B \quad \text{such that} \quad \neq D$$

C.3   Graphics in LATEX

In this chapter we will focus on several tools that extend your knowledge of figures beyond just `includegraphics` and move you toward the domain of professional publications. The tools that we'll cover are:

1. The `tikz` package,

2. The `pgfplots` package,

3. Using `GeoGebra` to generate `tikz` code, and

4. Using `MatLab` to generate `tikz` code.

These tools take a lot of work, but the end result is well worth it.

There is nothing worse or more distracting than a poorly done figure.

There is more to these packages than we could possibly cover in a few days. It is imperative that you use the internet to its fullest extent with these packages. You can get yourself into a pickle with some of the internet-based examples, but starting with someone else's code for these packages is über helpful sometimes!

What I'll present here are simply a few examples to get you going.

C.3.1   The Tikz and PGFPlots Packages

The Tikz package is made for doing line drawings. The simplest mode of operation with Tikz is to do point-by-point drawings on a Cartesian grid.

> **Example C.5.** Say we want to draw a coordinate plane with a few geometric shapes. Inside the figure environment we include a tikzpicture environment around the code for the picture. Be sure to end every Tikz line with a semicolon; Figure C.2 shows the results.
>
> ```
> \begin{tikzpicture}
>     \draw[color=gray] (-3,-3) grid (3,3);
>     \draw[thick, <->] (-3,0) -- (3,0) node[anchor=west]{$x$};
>     \draw[thick, <->] (0,-3) -- (0,3) node[anchor=south]{$y$};
>     \draw[very thick, blue, fill=blue!50] (0,0) --
>         (2,1) -- (1,3) -- cycle;
>     \draw[very thick, dashed, color=red,
>         fill=red!20!blue, opacity=0.5] (-2,0) circle(1cm);
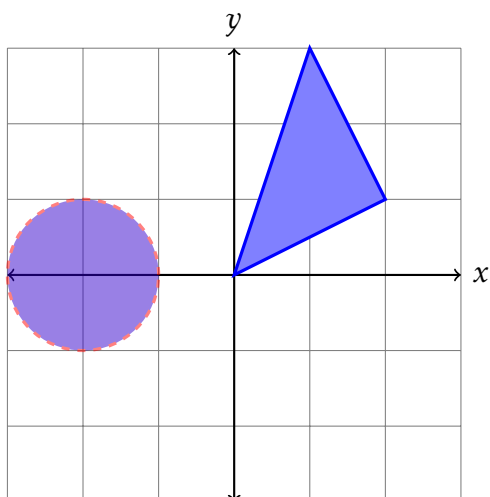> \end{tikzpicture}
> ```



Figure C.2. A simple Tikz picture

For more examples about the Tikz package, see http://www.texample.net/tikz/examples/ ...texample is your new best friend.

You don't have to plot in MATLAB, Python, Excel, or any other tool when writing a technical document! Say this to yourself 100 times and be sure that you're sitting down.

Example C.6. This first example shows a simple way to plot functions.

```
\begin{tikzpicture}
    \begin{axis}[axis lines=center, xlabel={x},
            title={My Awesome Plot},
            domain=-2*pi:2*pi, ymin=-1.5, ymax=2, grid]
        \addplot[blue, thick, smooth] {sin(deg(x))};
        \addlegendentry{$f(x)=\sin(x)$};
        \addplot[red, thick, smooth] {cos(deg(x))};
        \addlegendentry{$g(x)=\cos(x)$};
        \addplot[black, dashed, thick, smooth] {0.1*exp(-x)};
        \addlegendentry{$h(x)=0.1\text{exp}(-x)$};
    \end{axis}
\end{tikzpicture}
```
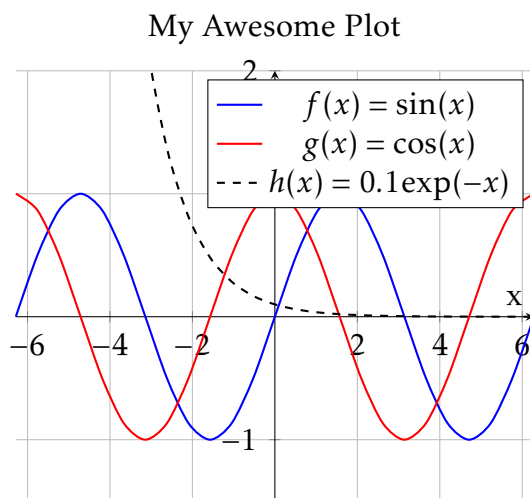

Figure C.3. A figure drawn with the `tikzpicture` and `axis` commands (leveraging the `pgfplots` package in the backgroud).

Next we'll follow with several more examples. Some of them are very advanced and some are beautifully simple.

Example C.7. Draw a bar chart for the following table of the world's largest producers of gem-quality diamonds in 2010. The solution is shown in Figure C.4.

Country	Millions of Carats
Botswana	25.0
Russia	17.8
Angola	12.5
Canada	11.8
Congo	5.5

Souce: USGS Mineral Commodity Summaries.

```
\usetikzlibrary{patterns}
\pgfplotsset{width=12cm,height=8cm}
\begin{tikzpicture}
    \begin{axis}[
            ybar,
            bar width=10mm,
            enlargelimits=0.15,
            xlabel={\Large{Country}},
            ylabel={\Large{Millions of Carats}},
            title={\Large{World's Largest Diamond Producers 2010}},
            xtick=data,
            symbolic x coords={Botswana,Russia,Angola,Canada,Congo},
            nodes near coords,
            axis lines*=left
        ]
        \addplot [pattern=crosshatch dots,pattern color=red!80!white,
            draw=red] coordinates {(Botswana,25)
            (Russia,17.8) (Angola,12.5) (Canada,11.8) (Congo,5.5)};
    \end{axis}
\end{tikzpicture}
```

C.4   Bibliography Management

There are two primary ways to manage a bibliography file in LATEX. In both ways you need to remember that (as usual) you have full control over everything! Two rules of thumb:

1. If you are using a short bibliography or if this paper stands alone then you probably want to use an embedded bibliography.

2. If you have a collection of references that will be used for several papers then you should consider using a BibTeX database.

Both types of bibliographies will save huge amounts of time and allow for very simple citation formats.
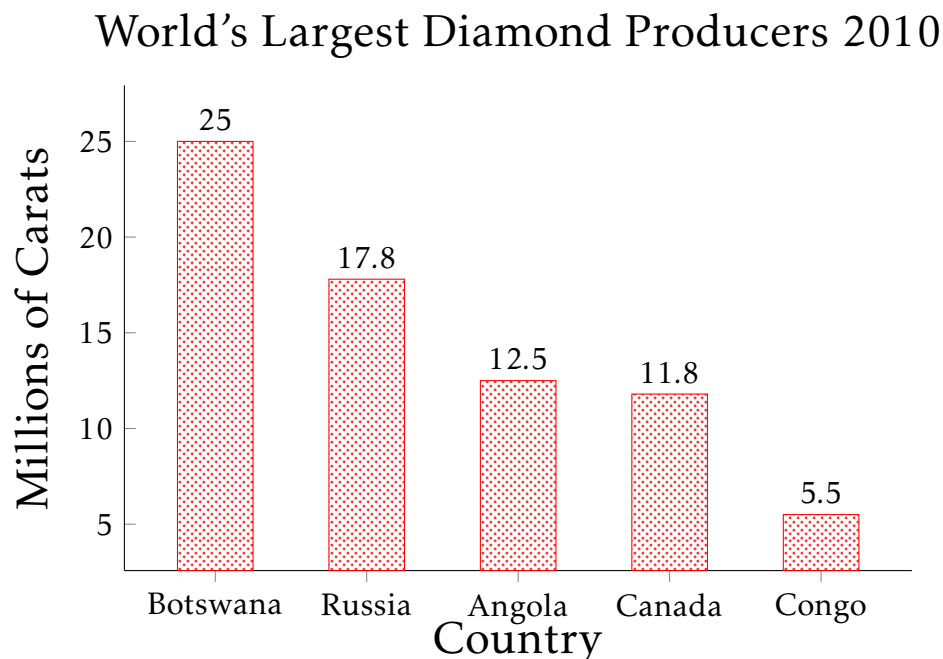
World's Largest Diamond Producers 2010

Figure C.4. Figure for Example C.7

As usual, there is MUCH more to writing a good bibliography than what can possibly be listed here. A really good source is the wiki page for the latex bibliography: http://en.wikibooks.org/wiki/LaTeX/Bibliography_Management.

C.4.1   Embedded Bibliography

If you're using an embedded bib for a stand-alone paper then just before the \end{document} you include all of the bibliography information. A simple example (with 1 paper) is included here:

```
\begin{thebibliography}{9}

\bibitem{lamport94}
  Leslie Lamport,
  \emph{\LaTeX: a document preparation system},
  Addison Wesley, Massachusetts,
  2nd edition,
  1994.

\end{thebibliography}
```

Use the \cite{ } command to cite items that are listed labeled inside the curly braces after \bibitem. For example, if we type \cite{lamport94} then we get a citation like this: [?].

C.4.2   Bibliography Database: BibTeX

BibTeX is a way for you to keep all of your bibliography materials in one place. The basic idea is as follows:

1. Start a file called `MyBib.bib` and follow the instructions from the link below to build your bibliography:
   http://ccm.ucdenver.edu/wiki/How_to_write_BibTeX_files

2. In your LATEX file you can cite bib items with the `\cite{ }` command. As you cite works and compile you will build the bibliography automatically. You will need to compile MANY times to get all of the cross referencing and citations to appear.

3. Be sure that the `*.bib` file is in the same working directory as your LATEX document (or at least give a path).

   The primary utility of a bibtex file is that you can simply build it once when you're working on a large project and the citations will draw only the parts that are necessary for the current paper.

Appendix D

Optional Material

This Appendix contains a few sections that are considered optional for the course as we teach it. There are only a few exercises and minimal editing has been doing on these sections. Instructors may be interested in expanding upon what is here for their classes.

D.1 Multivariable Numerical 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. (In Section D.3 we will build a multivariable version of Newton's method if you want to look ahead.)

Let's see if you can extend your intuition from single variable to multivariable.

Problem D.1. 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.

▲

Problem D.2. 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. ▲

Problem D.3. How does the Monte Carlo algorithm extend to a two-variable optimization problem? Clearly write your algorithm. ▲

Problem D.4. Try out the three algorithms that you just wrote on the function $f(x,y) = \sin(x)\cos(y) + 0.1x^2$ which has many local extrema and no global maximum. ▲

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.

Algorithm D.5 (Gradient Descent). We want to solve the problem

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

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

2. We are going to define a difference equation that gives successive guesses for the optimal value:
$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha \nabla f(\mathbf{x}_n).$$
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?).

3. Repeat the iterative process in step 2 until two successive points are *close enough* to each other.

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* and follow the positive gradient instead of the negative gradient.

Problem D.6. Write generic code to implement the Gradient Descent algorithm in as many dimensions as necessary and test your code on a multivariable function where you know the location and value of a local minimum. ▲

Problem D.7. The functions like $f(x,y) = \sin(x)\cos(y)$ have many local extreme values which makes optimization challenging. Impelement 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? ▲

D.2 Interpolation

The least squares problem that we studied in Chapter 4 seeks to find a best fitting function that is *closest* (in the 2-norm sense) to a set of data. What if, instead, we want to match the data points exactly 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 fits these points.

D.2.1 Vandermonde Interpolation

We have technically already seen Vandermonde interpolation. In Section 4.4 we built a system of equations to solve the least squares problem. In the least squares problem we had more data points than unknown parameters resulting in an over-determined system (look back to Section 4.4 to remind yourself). If, however, we choose a polynomial model that has the same number of unknown parameters as data points then the resulting system if not over-determined.

For example, let's say that we have the data set

$$S = \{(0, 1), (1, 2), (2, 5), (3, 10)\}$$

and we want to fit a polynomial then we can use a cubic function (which has 4 parameters) to match the data perfectly. Indeed, 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

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 5 \\ 10 \end{pmatrix}.$$

Notice that the system of equations is square, and solving using any method discussed in this chapter results in $\beta_0 = 1$, $\beta_1 = 0$, $\beta_2 = 1$, and $\beta_3 = 0$. Hence, the interpolating function is $p(x) = 1 + 0x + 1x^2 + 0x^3 = 1 + x^2$, and we know that $p(x)$ matches this data set perfectly as seen in Figure D.1.

Problem D.8. Write a Python function that accepts a list of ordered pairs (where each x value is unique) and builds a Vandermonde interpolation polynomial. Test your function on the simple example listed above and then on several larger problems. It may be simplest to initially test on functions that we know. ▲

Problem D.9. Build a Vandermonde interpolation polynomial to interpolate the function $f(x) = \cos(2\pi x)$ with 10 points that are linearly spaced on the interval $x \in [0, 2]$. ▲
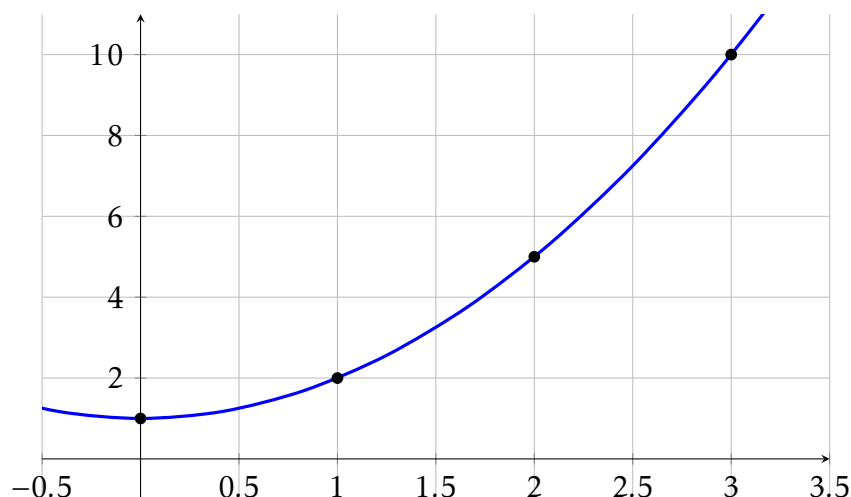
Figure D.1.   A simple Vandermonde interpolation for the data set $S =$ $\{(0,1),(1,2),(2,5),(3,10)\}$ resulting in the interpolating function $p(x) = 1 + x^2$.

Definition D.10 (The Vandermonde Matrix). Let $S = \{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$ be a list of ordered pairs where the x values are all unique. Using Vandermonde interpolation we arrive at the system of equations

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}. \tag{D.1}$$

The matrix on the left-hand side of (D.1) is called the **Vandermonde Matrix**.

Problem D.11. Vandermonde matrix is relatively easy to conceptualize and code, but there is an inherent problem. Use your code from Problem D.8 to create a plot on a `semilogy` scale. The horizontal axis of the plot is the order of the interpolating polynomial and the vertical axis is the ratio $|\lambda_{max}|/|\lambda_{min}|$ where λ_{max} and λ_{min} are the maximum and minimum eigenvalues of the Vandermonde matrix respectively. What does this plot tell you about Vandermonde interpolation for high-order polynomials?   ▲

D.2.2   Lagrange Interpolation

Lagrange interpolation is a rather clever interpolation scheme where we build up the polynomial from simpler polynomials. For interpolation we want to build a polynomial $p(x)$ such that $p(x_j) = y_j$. If we can find a polynomial $\phi_j(x)$ 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.

Problem D.12. 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

$$\phi_0(x) = \frac{(x-1)(x-2)(x-3)}{(0-1)(0-2)(0-3)}.$$

(c) Verify that $\phi_1(x)$ can be defined as

$$\phi_1(x) = \frac{(x-0)(x-2)(x-3)}{(1-0)(1-2)(1-3)}.$$

(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.

▲

Technique D.13 (Lagrange Interpolation). To build an interpolating 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 D.14. 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.

Problem D.15. 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 the examples that we've presented in this section. ▲

D.2.3 Interpolation at Chebyshev Points

Problem D.16. 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?
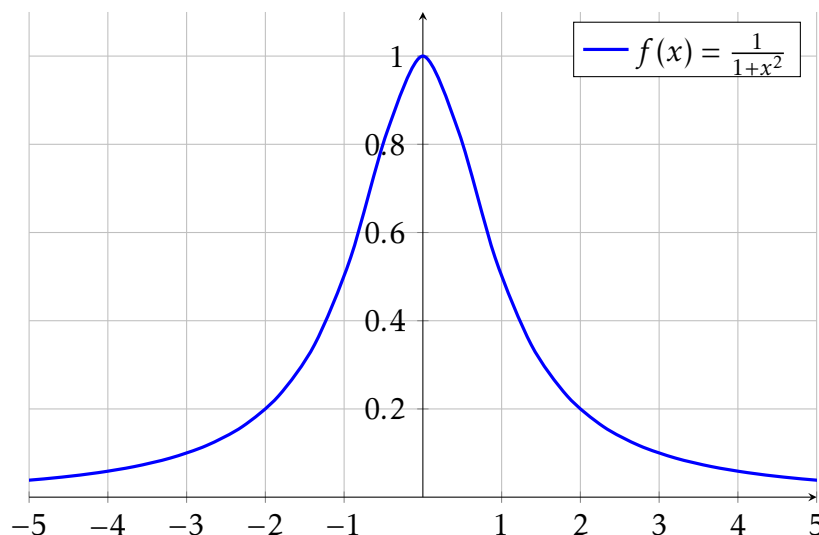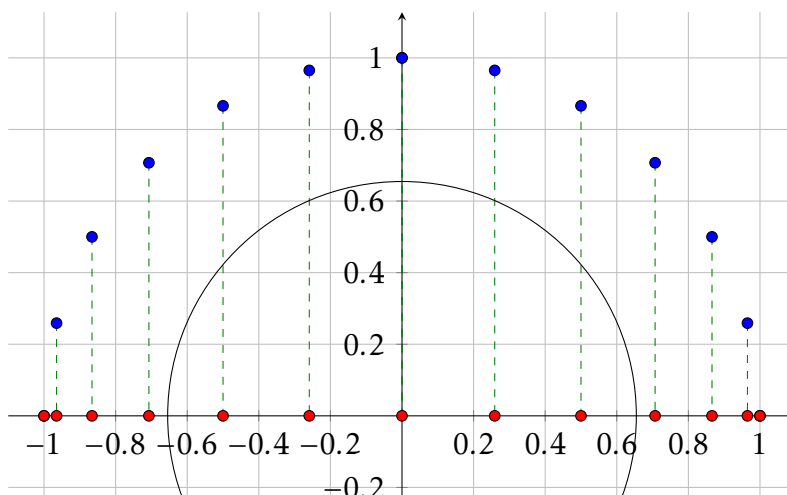
Figure D.2. Chebyshev interpolation nodes for the interval $[-1, 1]$. In this case each node is separated by $\pi/8$ radians giving 13 interpolations points including the endpoints.

▲

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 (shown in black in Figure D.2).

- Pick n equally spaced points along the semicircle (i.e. same arc length between each point). (shown in blue in Figure D.2)

- Project the points on the semicircle down to the interval. Use these projected points for the interpolation. (shown in red in Figure D.2)

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 from the formula

$$x_j = \cos\left(\frac{\pi j}{n}\right), \quad \text{for} \quad j = 0, 1, \ldots, n \tag{D.2}$$

on the interval $[-1, 1]$.

To transform the Chebyshev points from the interval $[-1, 1]$ (found with (D.2)) to the interval $[a, b]$ we can apply a linear function 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 $[-1, 1]$.

Problem D.17. 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 an interpolation with Chebyshev nodes. Give plots for polynomial of order $n = 2, 3, 4, \ldots$. Be sure to show the original function on your plots as well. ▲

Problem D.18. Demonstrate that the Chebyshev interpolation nodes will improve the stability of the Vandermonde matrix over using linearly spaced nodes. ▲

D.3   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

$$f(x,y) = 0$$
$$g(x,y) = 0.$$

In the present problem this amounts to solving the nonlinear system of equations

$$x\sin(y) = 0$$
$$\cos(x) + \sin(y^2) = 0.$$

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.

Problem D.19. 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),

$$J(x,y) = \begin{pmatrix} \sin(y) & \underline{\quad\quad} \\ \underline{\quad\quad} & \underline{\quad\quad} \end{pmatrix}.$$

(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 $(x_0, f(x_0))$

$$f(x) - f(x_0) \approx f'(x_0)(x - x_0)$$

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 ... it is a matrix after all!!

   (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

$$\boxed{\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} x_n \\ y_n \end{pmatrix} - J^{-1}(x_n, y_n)F(x_n, y_n).}$$

Writing this in full matrix-vector form we get

$$\boxed{\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} x_n \\ y_n \end{pmatrix} - \begin{pmatrix} f_x & f_y \\ g_x & g_y \end{pmatrix}^{-1} \begin{pmatrix} f(x_n, y_n) \\ g(x_n, y_n) \end{pmatrix}.}$$

(c) Write down the Newton iteration formula for the system

$$x\sin(y) = 0$$
$$\cos(x) + \sin(y^2) = 0.$$

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 $\mathbf{b} = J^{-1}(x_n, y_n)F(x_n, y_n)$.

- The vector $\mathbf{b}$ is the same as the solution to the equation $J(x_n, y_n)\mathbf{b} = F(x_n, y_n)$ at each iteration of Newton's Method.

- Therefore we can so a relatively fast linear solve (using any technique from this chapter) to find $\mathbf{b}$.

- The Newton iteration becomes

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} x_n \\ y_n \end{pmatrix} - \mathbf{b}.$$

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. Be prepared to share your answer and your technique with your peers. 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.

▲

Problem D.20. Test your code from the previous problem on the system of nonlinear equations

$$1 + x^2 - y^2 + e^x \cos(y) = 0$$
$$2xy + e^x \sin(y) = 0.$$

Note here that $f(x,y) = 1 + x^2 - y^2 + e^x \cos(y)$ and $g(x,y) = 2xy + e^x \sin(y)$. ▲

Problem D.21. 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 $F : \mathbb{R}^k \to \mathbb{R}^k$.

Let $\mathbf{x}$ be a vector in $\mathbb{R}^k$, let

$$F(\mathbf{x}) = \begin{pmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_k(\mathbf{x}) \end{pmatrix}$$

be a vector valued function, and let J be the Jacobian matrix

$$J(\mathbf{x}) = \begin{pmatrix} \partial f_1/\partial x_1(\mathbf{x}) & \partial f_1/\partial x_2(\mathbf{x}) & \cdots & \partial f_1/\partial x_k(\mathbf{x}) \\ \partial f_2/\partial x_1(\mathbf{x}) & \partial f_2/\partial x_2(\mathbf{x}) & \cdots & \partial f_2/\partial x_k(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \partial f_k/\partial x_1(\mathbf{x}) & \partial f_k/\partial x_2(\mathbf{x}) & \cdots & \partial f_k/\partial x_k(\mathbf{x}) \end{pmatrix}$$

By analogy, the multi-dimensional Newton's method is

$$\mathbf{x}_{n+1} = \mathbf{x}_n - J^{-1}(\mathbf{x}_n)F(\mathbf{x}_n)$$

where $J^{-1}(\mathbf{x}_n)$ is the inverse of the Jacobian matrix evaluated at the point $\mathbf{x}_n$.

(a) Write code that accepts any number of functions and an initial vector guess and returns an approximation to the root for the problem $F(\mathbf{x}) = \mathbf{0}$.

(c) Use Newton's method to find an approximate solution to the system of equations

$$x^2 + y^2 + z^2 = 100$$
$$xyz = 1$$
$$x - y - \sin(z) = 0$$

▲

Technique D.22 (Multi-Dimensional Newton's Method). The iteration for Newton's Method in multiple dimensions is

$$\mathbf{x}_{n+1} = \mathbf{x}_n - J^{-1}(\mathbf{x}_n)F(\mathbf{x}_n)$$

where $\mathbf{x}_n \in \mathbb{R}^k$, J is the k-dimensional Jacobian matrix, and $F : \mathbb{R}^k \to \mathbb{R}^k$. Take note that numerically you do not want to directly compute the inverse of the Jacobian. Instead you solve the sub-problem

$$J(\mathbf{x}_n)\mathbf{b} = F(\mathbf{x}_n)$$

for the vector $\mathbf{b}$ using any technique from linear algebra (e.g. an LU or QR decomposition) and iterate the equation

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{b}.$$

Problem D.23. 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. ▲

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.

Problem D.24. Find the equilibrium point(s) for the system of differential equations

$$x' = \alpha x - \beta xy$$
$$y' = \delta y + \gamma xy$$

where $\alpha = 1, \beta = 0.05, \gamma = 0.01$ and $\delta = 1$. You will play more with this differential equation system later. ▲

Problem D.25. Find the equilibrium point(s) for the system of differential equations

$$x' = -0.1xy - x$$
$$y' = -x + 0.9y$$
$$z' = \cos(y) - xz$$

if they exist. ▲

Problem D.26. (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 418 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. ▲

Bibliography

[1] A. Greenbaum and T. Charier. *Numerical Methods: Design, Analysis, and Computer Implementation of Algorithms* Princeton University Press. 2012.

[2] R. Burden, D, Faires, and A. Burden. *Numerical Analysis, 10ed* Cengage Learning. 2016.

[3] D. Kindaid and W. Cheney. *Numerical Analysis, 2ed.* Brooks/Cole Publishing, 1996.

[4] R. Haberman. *Applied Partial Differential Equations, 4ed.* Pearson Education Inc. Upper Saddle River, New Jersey, 2004

[5] D. Lay. *Linear Algebra 4ed.* Pearson Education Inc. Upper Saddle River, New Jersey, 2012.

[6] M. Meerschaert. *Mathematical Modeling 4ed.* Academic Press Publications, 2013.

[7] Holistic Numerical Methods http://nm.mathforcollege.com/
The Holistic Numerical Methods book is probably the most complete free reference that I've found on the web. This should be your source to look up deeper explanations of problems, algorithms, and code.

[8] Scientific Computing with MATLAB http://gribblelab.org/scicomp/scicomp.pdf

[9] Tea Time Numerical Analysis http://lqbrin.github.io/tea-time-numerical/