Lecture Notes, Math 521, Fall 2018 Edition

Zachariah B. Etienne

June 25, 2019

The following is a set of lecture notes I compiled, wrote, and used over the course of the Fall 2015, 2016, 2017, and 2018 semesters of Math 521. All source material, including freely-available online lecture notes and encyclopedia articles, were very carefully chosen for strength in continuing the lecture narrative. Sources are cited, though the narrative is mine. I worked hard to distill and rewrite the material into small, easily-digested pieces, suitable for a 500-level Math class. It is my goal over the coming years to continue to improve and evolve the current material into a work that, in combination with original homework sets and exam problems already devised, will form the foundation of a textbook on Numerical Analysis.

Table of Contents

0.1 Preface: Preliminaries

To do well in this class, it is expected that in addition to good programming skills, you are also competent in basic high school and undergraduate mathematics, including

- Scientific notation
- Significant digits/figures; can you perform arithmetic keeping a fixed number of significant digits? It is critically important to know this, as it is fundamental to understanding how computers do floating point arithmetic. Understanding the limitations of this arithmetic is a central skill in this class.
- Basic algebra: Solving nonlinear equations & inequalities involving Logarithms, Exponentials, and Quadratics.
- Calculus I: Differentiation using Chain Rule, Product Rule. Basic exposure to Taylor Series, series convergence, radius of convergence.
- Calculus II: Integration by variable substitution, Integration by parts
- Linear Algebra: Computing determinants, Properties of determinants, Computing Eigenvalues/Eigenvectors
- Calculus IV/Ordinary Differential Equations (ODEs): Basic solution strategies: method of separation, etc. Exposure to Even/Odd functions, Fourier series, etc.

In terms of coding abilities, on coding assignments you will be expected to design, write, and debug codes that are roughly 200 lines long. It is important that you start on coding assignments immediately after receiving the homework, or bugs may prevent you from completing the assignment on time.

If you consider yourself a bit “rusty” on any of these or any other topics covered in this course, I have prepared a “Suggested/Additional Online Reading List”, which is linked to on the Course home page.

Now hand out the Skills Assessment. Give students 20 minutes to complete.

MATH 521, Fall 2018 Notes

Prof. Zachariah B. Etienne

Chapter 1: Introduction: Benefits and Pitfalls to Solving Mathematical Problems on the Computer

Computers are remarkable machines, capable of performing *billions* (i.e., of order 10^9) mathematical operations each second. So when we are faced with a mathematical problem—particularly a very challenging one—a computer may often be the most efficient or only means to solve it.

The central focus of this class is to provide a graduate-level introduction to the basic strategies used to solve a variety of mathematical problems on the computer. We call these strategies **algorithms**. It is assumed that you already know how to program computers, and this skill will be necessary to "code up" (i.e., program a computer to perform) these algorithms, which will be a major component to homework assignments. If you cannot understand how an algorithm works, you will quickly find yourself confused when the computer outputs the wrong result.

There is no "perfect algorithm" for solving mathematical problems. Instead, there are a multitude of algorithms for solving any general problem. Even an algorithm designed for a particular problem will have shortcomings. It is therefore *essential* that we understand how these algorithms work, which requires **a strong foundation in undergraduate mathematics**, including algebra, trigonometry, scientific notation, matrix algebra, differentiation, Taylor Series, Fourier Series, and integration. The strongest indicator of your final grade in this class will not be your ability to program computers (though this is a critical skill), but instead your proficiency in undergraduate mathematics.

Since computer algorithms we program can and will give the incorrect solution, someone who is skilled in solving mathematical problems on the computer must be *equally* skilled in understanding sources of error.

Continued on next page

1.1 The Four Horsemen of the Error-pocalypse

We classify errors into four categories when solving mathematical problems on the computer.

1. *Roundoff Error*: Computers have limited facilities for storing numbers, so to maximize efficiency, we usually represent numbers *approximately* in scientific notation with a *fixed number* of significant digits. As we will see in Chapter 3, error caused by storing only a fixed number of significant digits—i.e., **roundoff error**—can poison our numerical solution. We must therefore be able to determine how and where our algorithms lose precision.
2. *Undersampling Error / Insufficient Resources*: When solving mathematical problems on the computer, it is often essential that we first understand the *scale* of the solution—usually by computing a "back of the envelope" or "order of magnitude" estimate of the solution (Chapter 4). Without such an estimate, the algorithms we use may **undersample** the solution and yield the wrong answer, or we may find only after wasting a large amount of time writing a computer program that the required computing power to solve a mathematical problem far exceeds our available resources.
   Indeed, modern computers are wonders of technology, but for certain problems, they are simply not powerful enough. To determine the amount of computational resources needed to solve a problem on the computer, we can combine a back-of-the-envelope estimate of the solution with the computational *complexity* (**cost**) of the underlying algorithm itself (Chapter 5). If the needed resources exceeds our budget, we may need to choose or develop more efficient algorithms, optimize existing algorithms, or move to more powerful computational resources.
3. *Truncation / Approximation Errors*: Understanding the limitations and approximations made by our algorithms is critically important. Sometimes for example, our solution might depend on an infinite Taylor series. We cannot typically compute or store an infinite number of terms in this series, so we instead sum the series up to a finite term, ignoring all remaining terms. The error imposed by ignoring remaining terms in this way is called **truncation error**.
4. *Human Error*: Suppose we solve a mathematical problem on the computer based on some combination of algorithms taught in this class, and this problem is impossible to solve without a computer. How can we be confident that our solution is correct? To this end, we must always work to hone our skills in *code validation*—the process by which we eliminate **human errors** both in choice of algorithms and bug-free implementation of the algorithms—and gain confidence that the algorithms we have coded both work correctly and yield reliable solutions. To this end, every coding assignment will contain at least some component of code validation.

MATH 521, Fall 2018 Notes

Prof. Zachariah B. Etienne

Chapter 2: Scientific Notation, Significant Digits, and Relative Error

2.1 Scientific Notation

Proper Scientific Notation

Proper scientific notation rewrites real numbers in the form

$$a.bcdefghij \times 10^X,$$

where *a*–*j* denote individual digits of the **mantissa** or **significand**, such that $a \neq 0$ unless *all digits are zero*, and *b*–*j* are all decimal digits (e.g., $b = 2$, $c = 8$). X denotes the **exponent**, which can be any integer. In this class, the real number would generally correspond to some measure, like distance, time, pressure, energy, power, etc.
The total number of digits given in the significand in scientific notation must correspond to the number of **significant digits** in the number we are given.

The number `131,511,234` is written as 1.31511234×10^8 in scientific notation. While 0.131511234×10^9 would represent the same number, it is *not* in proper scientific notation (because $a \neq 0$ unless *all digits are zero*).

As another example, the number `0.000034100` would be 3.4100×10^{-5} in scientific notation. Notice that we kept the two zeroes after the 1 because these are assumed to be significant; the number `0.000034100` is assumed to have 5 significant digits.

Additionally, the number `1020000` would be 1.02×10^6 in scientific notation. If we wished to indicate that the zeroes after the 2 *were* significant, we would add a decimal point after the last zero; `1020000.` would be 1.020000×10^6 in scientific notation.

Finally, the number `1.5121` would be 1.5121×10^0 in scientific notation; notice the exponent is zero in this case.

In this class, we will make use of scientific notation both in the context of measurements, as well as in the context of performing fixed-precision arithmetic on the computer. We review the former in Sec. 2.1.1, and the latter in Sec. 2.1.2.

2.1.1 The Use of Scientific Notation for Measurements

Scientific notation is typically used to record measurements in science and engineering contexts, so that **the number of significant digits is related to the precision of our measuring device**. Suppose we have a standard ruler. With this ruler, we might be able to measure the thickness of a dictionary to a tolerance of 1 millimeter (mm), or equivalently, 0.1 cm. For example, our measurement with the ruler may indicate that the thickness is 11.0 cm.

Now suppose we wish to construct a bookshelf that can hold exactly 1,000 dictionaries in the standard position. How long must the bookshelf be? Because our measuring device is limited to the 1 millimeter tolerance, it would be *meaningless* and *misleading* to claim that, based on our measurement, the bookshelf must be exactly 11,000 cm in length. So it is very important that we keep track of our measurement tolerances, and **scientific notation** provides an easy means to accomplish this.

Being comfortable with scientific notation will be essential for succeeding as a professional in this field. **If you are rusty, you are strongly encouraged to next work through all material and quizzes on Scientific Notation & Significant Figures in** `https://tinyurl.com/sigdigitsreview`.

2.1.2 Scientific Notation on the Computer

As in our example with the books, the number of significant digits generally corresponds to the precision of our measuring device. We use computers in this class not to measure physical quantities directly, but to solve mathematical problems. Computers represent our numerical solutions and perform all arithmetic in scientific notation.

Computer scientific notation is unique in that a *fixed number* of significant digits is kept in all arithmetic operations, regardless of the number's relation to actual measurements. *Therefore we must be careful when interpreting the results our computer gives us, as the computer does not obey the standard rules of arithmetic with scientific notation.* For example, if our computer could only store three significant digits and we were to evaluate $7.13 \times 10^4 - 7.11 \times 10^4$, the result would be $2.XY \times 10^2$, where XY could be any two digit number. In other words, a computer that stores three significant digits will always give the result to three significant digits, despite the fact that in this case, the result is actually only valid to one significant digit.

This is quite different from the normal use of scientific notation for measurements, in which our measurement apparatus might be able to provide a precision of $\pm 0.01 \times 10^4$, meaning that the number 1.11×10^3 would contain a digit of significance *beyond* the capabilities of the measuring device.

In addition, instead of writing the exponent symbols $\times 10^X$ for all numbers, computers adopt the `eX` symbol. That is to say, instead of outputting 1.31041×10^{-21}, the number `1.31041e-21` will be output.

Examples of Computer Scientific Notation

Suppose we have a computer that can only store **three** significant digits in the mantissa in scientific notation. Assume for the purpose of this problem that there is no limit on the range of the exponent. Evaluate the following expressions, keeping the number of significant digits fixed at three. Also assume that the computer will completely ignore digits beyond the third significant digit; this is equivalent to always rounding down in any arithmetic operation. Your answer must be in proper computer scientific notation and correctly.

- `1.49e-22 + 3.11e-21`: The answer is `3.25e-21`; we were told to ignore the all significant digits beyond the third.
- `1.49e22 + 3.11e2`: The answer is `1.49e22`; we were told to ignore all significant digits beyond the third. Notice that `3.11e2` is completely insignificant and is therefore *ignored.*

2.2 Quantifying Numerical Error

When we solve mathematical problems on the computer, we typically do so in a way that *approximates* the exact solution to some number of significant digits, which might be impossible or impractical to compute by hand. Determining whether we can trust the computer's solution *is a central focus of this class*, and depends entirely on how much error we are willing to accept in our (typically) approximate solution. So to succeed in this class and as a professional in the field requires that we understand the error in our approximate numerical solution.

One quite useful strategy for measuring the error in our numerical solution is to force our numerical code to solve a problem that we can solve **exactly** by hand. Let's suppose our numerical solution gives the result 1.0229×10^3 seconds, and we find by hand the exact solution to be 1.0220×10^3 seconds.

So what is the error in the numerical solution?

Definition: The **Absolute Error**, corresponding to the absolute difference between numbers n_1 and n_2 is given by

$$E_{\text{Abs}} \equiv |n_1 - n_2|,$$

so for our example, the absolute error between 1.0220×10^3 seconds and 1.0229×10^3 seconds is

$$E_{\text{Abs}} = |1.0220 \times 10^3 - 1.0229 \times 10^3| \text{ seconds} = 0.0009 \times 10^3 \text{ seconds} = 9 \times 10^{-1} \text{ seconds}$$

Notice that the numbers going into this expression are accurate to *five* significant digits, but after the subtraction, we have a number that contains only *one* significant digit of precision. **Definition:** A reduction in the number of significant digits due to an addition or subtraction is known as a **loss of significance**, a **catastrophic cancellation**, or a **catastrophic subtraction**.

We must be very careful when a catastrophic cancellation occurs when doing arithmetic on the computer, because the computer will by default store a *fixed* number of significant digits, regardless of whether a catastrophic cancellation has occurred. For the example above, a computer might very well give us the answer 9.0129×10^{-1}. It is *not* the computer's responsibility to tell us that this number is only valid to a single significant digit; we must be able to figure this out on our own!

While absolute error can be a useful measure, we are often more interested in the *percent error* or *relative error*, which as we will find can give us the number of significant digits to which we can trust our answer. **Definition:** The **relative error** is the absolute difference, rescaled by the *magnitude* of the numbers $|n_1|$ and/or $|n_2|$, and the **percent error** is the relative error expressed in percentage.

There are at least two strategies for computing relative error, and both give similar results for this case, though only the second provides a means for directly and unambiguously computing relative error used when an exact solution is unknown. **Definition:** When the exact solution is unknown, the relative error might instead be referred to as the **relative difference**.

The first strategy for computing relative error compares two numbers n_1 and n_2 and scales the difference by the magnitude (absolute value) of n_2:

$$E_{\text{Rel},1} = \left| \frac{n_2 - n_1}{n_2} \right|.$$

Notice that this measure emphasizes n_2 as setting the magnitude, so if the exact value is known, we set n_2 to that value, and n_1 as the numerical solution. When n_2 and n_1 agree to more than a couple of significant digits, the choice of n_1 or n_2 in the denominator *will not significantly change the relative error.* You should be able to prove this to yourself.

The second strategy for computing relative error could be referred to instead as the *relative difference*, as this strategy is used primarily when an exact solution is unknown, and we are interested only in how well two approximate solutions agree. This second definition of relative error, i.e., relative difference, treats the magnitudes of n_1 and n_2 equally, by replacing the $|n_2|$ in the denominator by the average of $|n_1|$ and $|n_2|$:

$$E_{\text{Rel},2} = \frac{|n_1 - n_2|}{(|n_1| + |n_2|)/2}.$$

Applying both definitions to our two numbers (assuming 1.0220×10^3 is the "exact" value) yields results that agree to all significant digits:

$$\begin{aligned} E_{\text{Rel},1} &= \frac{0.0009 \times 10^3}{1.0220 \times 10^3} = 9 \times 10^{-4} = 9 \times 10^{-2}\% \\ E_{\text{Rel},2} &= \frac{0.0009 \times 10^3}{1.02245 \times 10^3} = 9 \times 10^{-4} = 9 \times 10^{-2}\%. \end{aligned}$$

Notice that we only kept one significant digit in our result, because 0.0009×10^3 is only known to one significant digit. Often Relative Error is used interchangeably with Percent Error; in this case, the Percent Error is $9 \times 10^{-2}\% = 0.09\%$.

Now consider a very basic question: To how many significant digits do two numbers agree? For example, suppose the numbers are 1.0220×10^3 and 1.0229×10^3? Clearly we would say they agree up to and including the fourth significant digit, so our answer is 4. But what about 1.0220×10^3 and 1.0221×10^3? Clearly they agree better, so it seems reasonable to say that the number of digits of agreement *does not need to be an integer*! In the former case, we'd say that the two numbers agree to *between 4 and 5 digits, but closer to 4*, and in the latter case, we'd say that the two numbers agree to *between 4 and 5 digits, but closer to 5.*

It is easy to count the number of digits of agreement between two numbers by eye, but how could we teach a computer to do this automatically? We could attempt a complicated algorithm that manually scans the digits of the number, seeing that, e.g., 100. minutes $= 1.00 \times 10^2$ minutes and 101 minutes$= 1.01 \times 10^2$ minutes agree to almost 3 significant digits. But notice that if we chose a different unit of time, say "ticks", where 100. minutes=99.0 ticks, then the two measurements would be 99.0 ticks and 100.0 ticks. Notice that the two numbers differ by only 1 part in 100, just like 100 and 101. Clearly we do not wish to define the number of digits of agreement based on our choice of unit, so this notion of counting digits of agreement is surely wrong, as it would depend on our number base and our choice of unit.

How do we solve this problem?

Consider the fact that the relative error between 1.0220×10^3 and 1.0229×10^3 is 9×10^{-4}, and these numbers agree to about 4 significant digits.

Similarly, the relative error between 1.02200×10^3 and 1.02209×10^3 is 9×10^{-5}, and these numbers agree to about 5 significant digits.

So it seems there is a pattern: look at the *exponent* of the relative error to get approximately the correct number of digits of agreement. What mathematical expression would convert 9×10^{-5} to a number that is nearly 5? If we can answer this question, then we can program a computer to give us the correct number of significant digits of agreement, SDA.

Based on this pattern, we can see that SDA must satisfy

$$SDA = -\log_{10} E_{\text{Rel}} + 1$$

Using this expression, we find that 1.01000 and 1.00000 yield an SDA of approximately 3, as expected. In addition, notice that 0.9900000 and 1.0000000 will give about 2.998 significant digits—i.e., almost 3 significant digits of agreement as well.

In this course, when you are asked to compute the logarithmic relative error between two numbers $\log_{10} E_{\text{Rel}}$, it is very important that you remember the very close relationship between this difference and the number of significant digits of agreement SDA.

Exercises:

1. How would you write the number `1020000` in proper scientific notation if the first *four* digits (starting with the 1 on the left) were significant?
2. Use scientific notation to write the thickness of the dictionary (given in the first example of the chapter) based on the 1 millimeter tolerance measurement. Based on the fact that digits after the least (rightmost) significant digit *are completely unknown*, write the range of widths that the bookshelf might require. Assume that the ruler always rounds *down* to the nearest millimeter; i.e., if the book's thickness falls between 11.00 cm and 11.09999... cm, we round to 11.0 cm.
3. Suppose we measure the thickness of a dictionary in our laboratory to be 11.0 cm, and our colleagues in Austria measure their lab notebook to be 4.5 cm thick. How many significant digits do we obtain in our measurement? How many significant digits do our colleagues measure? Assuming that we use the same ruler as described in the previous exercise, if we were to stack the dictionary and lab notebook atop one another, to how many significant digits do we know the total height, based only on our current measurements?
4. To how many significant digits do 1.023151×10^{-30} and $1.\bar{0} \times 10^{-30}$ (i.e., the exact number 10^{-30}) agree? Your answer must be in the form "Between N and $N + 1$ significant digits", where you must fill in the integers N and $N + 1$.
5. To how many significant digits do $n_1 = 1.023151 \times 10^{-30}$ and $n_2 = 0.000000 \times 10^{-30}$ agree?

Solutions:

1. 1.020×10^6
2. Thickness of dictionary: 1.10×10^1 cm. Since digits after the least significant digit *are completely unknown*, the width of the bookshelf will range from 1.10×10^4 cm to $1.10\bar{9} \times 10^4$ cm$= 1.11 \times 10^4$ cm. So to be safe, we should construct our bookshelf to be 1.11×10^4 cm in width.
3. *How many significant digits do we obtain in our measurement?* Answer: 3. *How many significant digits do our colleagues measure?* Answer: 2. *Assuming that we use the same ruler as described in the previous exercise, if we were to stack the dictionary and lab notebook atop one another, to how many significant digits do we know the total height, based only on our current measurements?* Answer: Between 2 and 3, but closer to 3. Can use our measure of significant digits to get a better idea.
4. Between 2 and 3 significant digits of agreement.
5. 0 significant digits of agreement. Must use $E_{\text{rel},2}$ here. Notice that we could simply swap n_1 and n_2 and use $E_{\text{rel},1}$ instead; there is nothing special about assigning zero to n_1 instead of n_2.

MATH 521, Fall 2018 Notes

Prof. Zachariah B. Etienne

Chapter 3: Number Storage and Arithmetic on the Computer

Great resource: `http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`

Computers store all information, including numbers, in a binary number base system consisting of ones and zeroes. So when doing arithmetic on the computer, we must be very cautious because computers allocate a limited number of binary digits to any given number.

In practice, it is rare that you would need to convert from decimal to binary format, but unless you understand *how* numbers are stored in binary format on the computer, you will sometimes be very confused by your computer's results. Thus the focus in this section will be on better understanding the limitations of finite precision arithmetic and how these limitations can sometimes be addressed.

In general, numbers on the computer are stored in one of two basic formats: *Integers* and *Floating Point.* Section 3.1 reviews the former and Sec. 3.2 the latter.

3.1 Integers on the Computer

In most modern programming languages, integers are a *type* of number that get exactly 32 bits (i.e., a set of 32 ones and zeroes) of storage space. This means that 2^{32} integers n can be represented using this data type, typically all numbers n satisfying $-2^{31} \leq n \leq 2^{31} - 1$ (using **two's complement**: `http://tinyurl.com/twoscomp`), meaning that integers from about -2.147×10^9 to $+2.147 \times 10^9$ can be represented exactly.

Integer arithmetic on computers will be *exact* as long as no part of the calculation exceeds the bounds of storage allocated to an integer in your given calculation. When this happens the results will usually be complete nonsense.

So we must either have foresight that no number will exist beyond these bounds before jumping in to a calculation using this data type, or regularly check that we are getting close. Otherwise our results will in general be untrustworthy.

Words of caution

Before doing integer arithmetic, first check how many bytes are allocated to the "integer type" in your chosen programming environment. Different programming environments adopt different standards for integer types. For example, in some implementations of C, an integer is defined as having 2 *bytes* or 16 bits of storage space (there are 8 bits in a byte). For 16 bits of storage, an integer n spans only the much more limited range $-2^{15} \leq n \leq 2^{15} - 1$, or $-32{,}768$ to 32,767.

16 or 32-bit integer calculations can be useful for very basic arithmetic. Unfortunately this is a very limited data type. For example, in the case of number theory we can contend with absolutely gigantic integers. What are we to do in this case? The answer is to use so-called "arbitrary precision arithmetic", which is built-in to Matlab and some symbolic calculators like Mathematica. For those of us using compiled languages like C/C++/FORTRAN, arbitrary precision arithmetic libraries exist and can be used for this purpose.

3.1.1 Exercises

Exercises:

1. Using two's complement, what range of unsigned integers n are stored in 6, 8, and 10 bits?
2. Using two's complement, what range of *signed* integers n can be stored in 6, 8, and 10 bits?
3. Assuming the following bits are ordered such that the rightmost bit is the 2^0 column, the second-to-rightmost bit is the 2^1 column, etc., convert the following 6-bit binary numbers to unsigned integers in base-10.
   (a) 010011
   (b) 100000
   (c) 011111
   (d) 111110
   (e) 011101
4. Assume that we have an integer format in which the leftmost bit sets the sign of the integer, where the number is negative if the bit is 1 and positive otherwise. Convert the following 6-bit binary numbers in this format to signed integers in base-10. For all bits except the leftmost bit, assume the same bit ordering as in the first problem.
   (a) 010011
   (b) 100000
   (c) 111111

Solutions to Exercises:

1. $0 \leq n \leq 63$, $0 \leq n \leq 255$, and $0 \leq n \leq 1023$, respectively
2. $-32 \leq n \leq 31$, $-128 \leq n \leq 127$, and $-512 \leq n \leq 511$, respectively.
3. (a) through (e): 19, 32, 31, 62, and 29, respectively.
4. (a) through (c): 19, -0 (signed zero), and -31, respectively.

3.2 Floating Point Numbers on the Computer

But what are we to do if our calculations involve rational, irrational, real, or complex numbers that may span many orders of magnitude and do not need to be exact? This is quite often the case in science and engineering applications for which we use computers to model systems for which we do not have exact measurements, and for which exactitude is therefore not a strict requirement. In this case, your best bet be to use numbers stored in floating point format, where the "point" denotes the decimal point, which "floats" according to the exponent of the number. Proper scientific notation, described in Chapter 2, is one example of a floating point number format.

How floating point numbers are stored is very easy to understand if you are familiar with scientific notation. For example, suppose we measure some event to be 154,132,000 nanoseconds in duration using a clock that can only measure to the nearest 1,000 nanosecond increment. We conclude that our clock measures the duration of this event to six significant digits. This measurement can be represented in scientific notation as 1.54132×10^8 nanoseconds.

Definitions: Notice that 1.54132×10^8 consists of two numbers; the number before the $\times$ and the number in the exponent above the 10. The former is called the **mantissa** or **significand**, which in this case is stored to 6 significant digits and the latter the **exponent**. So on a base-10 computer, storing such a number would require a total of 6+1 "deci-bits" (i.e., each "deci-bit" can store numbers zero through nine; notice that if the first "deci-bit" can be zero, in order to denote the number zero).

However, computers are base-2 machines, storing numbers as sets of ones and zeroes. We have reviewed how computers store integers, but how can they store floating point numbers?

Definition: In this class, we will make extensive use of **computer scientific notation**, which takes the form *mantissa*`e`*exponent*. So for example, the number 9.183×10^{-5} is written `9.183e-5` in computer scientific notation.

3.2.1 Floating-Point Arithmetic with Fixed Storage: Base-10 Example

Since we are more familiar with numbers written in *decimal* (base-10) and not *binary* (base-2), let us first imagine a computer that stores floating point numbers using a limited number of base-10 "deci-bits":

Example: Base-10 computer that stores 2 significant digits

Imagine you have a *base-10* computer number format that can only store 2 significant **decimal** digits of precision in the mantissa (with an added sign bit), and can only store integer exponents in the range $[-5, +4]$ (because there are 10 digits between -5 and 4, inclusive). In addition,

- **Definition**: **Overflow**: Numbers greater in magnitude than the largest representable number in a finite-precision format will be evaluated to `Infinity` if the number is positive or `-Infinity` if the number is negative. For this number format, expressions that — when evaluated — result in numbers greater than to `9.9e4` or less than `-9.9e4` are evaluated to `+Infinity` and `-Infinity`, respectively.
- **Definition**: **Underflow**: Positive numbers smaller than the smallest nonzero number will be evaluated as zero. Negative numbers larger than the smallest nonzero number will be evaluated as negative zero. For this number format, numbers smaller than `1.0e-5` but larger than zero will be evaluated to `0.0e0`. Numbers larger than `-1.0e-5` but smaller than zero will be evaluated to `-0.0e0`.

1. What is the largest number ϵ storable in a number format such that $1 + \epsilon$ will evaluate to 1? (In this class, we will define this ϵ as **machine epsilon**. Warning: There are multiple definitions of "machine epsilon" in the literature; this is the one we will use throughout this class.)
2. Next evaluate the following expressions using this computer number format; you must abide by the restrictions of the computer number format in your calculations, and your answers must be in the same number format:
   (a) `1.0e0 + 3.1e-1`
   (b) `4.5e-3 + 5.5e-1`
   (c) `1.0e-3*5.5e4`
   (d) `1.0e3/5.5e-5`
   (e) `1.0/(1.0e3/5.5e-5)`

Solutions to Exercises:

1. In any number system we will encounter in this class, machine epsilon as we have defined it can be found via the following two-step process:
   1. Find the first number representable in this number system that is greater than one. Call that number M. Clearly $M - 1$ is a representable number in the number system.
   2. Machine epsilon is simply the largest number in this number system that is less than $M - 1$. *Tip*: As you work through this process, always check that M, $M - 1$, and your machine epsilon are representable in your number system. Otherwise you are doing it wrong.

   Now let's apply this process to this problem. The smallest number greater than one in this number system is $M = 1.1e0$. Further, $M - 1 = 0.1 =$`1.0e-1` is a number in this number system. Thus finding machine epsilon is equivalent to finding the largest number less than $M - 1 =$`1.0e-1` in this number system. Clearly that number is $\epsilon =$`9.9e-2`, as there is no number larger than this but smaller than `1.0e-1` in this number system.
2.
   (a) `1.3e0`
   (b) `5.5e-1`. Let's rewrite the expression as `4.5e-3 + 5.5e-1 = 5.5e-1*(1 + 4.5e-3/5.5e-1`. Since `4.5e-3/5.5e-1`$< \epsilon_m$, this will evaluate to `5.5e-1`.
   (c) `5.5e1`. Note that `5.5e5` is representable in this number format.
   (d) `Infinity`. Note that `5.5e-5` is representable in this number format, and `1/5.5e-5`≈`1.8e4` is also representable. But when multiplying `1.8e4` by `1.0e3`, we get a number that exceeds the bounds of our number format.
   (e) `0`. The number in parentheses will evaluate to `Infinity`, and any finite number dividing `Infinity` wille evaluate to zero.

3.2.2 Double Precision Floating Point Arithmetic

Computers store floating point numbers in base-2, most commonly using either 32 or 64 bits, with the former being called *single precision* and the latter *double precision*. In this class, and in most scientific, mathematical, and engineering contexts, we will exclusively use double precision.

Source (vetted):
`https://en.wikipedia.org/w/index.php?title=Double-precision_floating-point_format&oldid=688044028`

Double Precision Floating Point Format

According to the IEEE 754 technical standard, double precision floating point numbers are stored in 64 bits on the computer as follows:

- Bit 63: Sign bit S
- Bits 62–52: exponent (11 bits) X.
- Bits 51–0: mantissa or significand (52 bits) M

With few exceptions (see box below), double precision numbers N are represented as follows, given mantissa bits b_{51}–b_0:

$$N = (-1)^S (1.b_{51}b_{50}...b_0) \times 2^{X-1023}, \quad \text{where } 1.b_{51}b_{50}...b_0 \text{ is a } \textit{binary} \text{ floating point number, } \textit{not} \text{ a decimal}$$

or equivalently,

$$N = (-1)^S \left(1 + \sum_{i=1}^{52} b_{52-i}2^{-i}\right) \times 2^{X-1023}, \text{ where } X = \sum_{i=0}^{10} b_{52+i}2^{+i}.$$

Exceptions to the above expressions

The exponent X, having 11 bits, can represent $2^{11} = 2048$ unique integers, of which only the 2046 numbers between -1022 ($X = 1$) to 1023 ($X = 2046$) are consistent the above definition for N. The exceptional cases $X = 0$ and $X = 2047$ are as follows:

- The exponent $X = 0$ and a mantissa of
  - all zeroes corresponds to $N = 0$. If the sign bit is set to 1, then $N = -0$, the signed zero.
  - anything other than zero typically corresponds to $N = \pm 0$ (depending on the sign bit), but this is implementation-dependent. Sometimes (rarely) these bits are used to store nonzero numbers smaller in absolute magnitude than roughly $2^{-1022} \approx 10^{-308}$.
- The exponent $X = 2047$ and a mantissa of
  - all zeroes corresponds to $\pm$Infinity, where the sign of the infinity is given by the sign bit.
  - any value except zero corresponds to `NaN` (Not a Number, which is output if an undefined arithmetic operation, like 1/0, is attempted).

3.2.2.1 Exactly Representable Numbers in Double Precision

In general we should expect a number to be represented to only 15–16 significant digits in double precision. For example, the decimal number $0.1 \equiv \frac{1}{10}$ is known exactly in base-10 floating point as

$$\begin{aligned} 0.1 &\equiv \frac{0}{10^0} + \frac{1}{10^1} + \frac{0}{10^2} + \sum_{n=3}^{\infty} \frac{0}{10^n} \\ &= 1.\bar{0} \times 10^{-1}. \end{aligned}$$

But when we convert this number to base-2 floating point, notice that we'll have to represent it as

$$0.1 = \frac{b_0}{2^0} + \frac{b_1}{2^1} + \frac{b_2}{2^2} + ...,$$

which happens to be a *repeating binary decimal!* And since double precision only stores 52 significant bits, the repeating binary decimal is artificially stopped at the 52nd bit, yielding an error that is about 1 part in 10^{16}.

But what numbers can we store exactly in double precision? Clearly powers of two like $\frac{1}{2} = 1 \times 2^{-1}$, or $\frac{1}{2^{1000}} = 1 \times 2^{-1000}$ are exactly representable. Let's dig further with an example:

Example #1

Consider the double precision floating point number `0:1[10 zeroes]:1[51 zeroes]`. To what does this correspond in decimal? Is the number exact?

- The sign bit is 0, so this is a positive number.
- Regarding the exponent, Bit 62 is 1, all others zero, so $X = 2^{10} = 1024$, and our exponent term is $2^{1024-1023} = 2^1 = 2$.
- Regarding the mantissa, Bit 51 is 1, all others zero, so $\sum_{i=1}^{52} b_{52-i} 2^{-i} = 2^{-1} = 1/2$.

So our number is $+(1 + 1/2) * 2 = 3$, which is an exact number.

Next notice that if we set *all* significand bits to zero, then the number becomes $+(1 + 0) * 2 = 2$. If we then "flip" all the bits in the exponent, we'll have `0[10 ones]`, which is the same number when counting in binary on our fingers as if we held up all of our fingers: 1023. Thus the result would be 1. We already determined that the number zero is represented in double precision (recall that +0 is represented by all bits being zero). So we have demonstrated that integers 0, 1, 2, and 3 are all *exactly* representable in double precision. But we also know that if we flip the sign bit, we can immediately get -3, -2, and -1.

Therefore, we have proven that the *contiguous* set of integers $N \in [-3 : 3]$ is exactly representable in double precision.

Example #2

What is the <u>largest range</u> of contiguous integers that can be stored in double precision?

First let's consider the **largest** integer we could possibly store exactly in double precision. Clearly we'd just need to maximize the exponent at 1023, and set all the bits in the mantissa to 1's. This will result in the integer N_{large}, where

$$\begin{aligned} N_{\text{large}} &= (1 + 2^{-1} + 2^{-2} + ... + 2^{-52}) \times 2^{1023} \\ &= 2^{1023} + 2^{1022} + 2^{1021} + ... + 2^{1023-52}. \end{aligned}$$

The result is nothing more than the sum of some very large integers, and the result is an integer.

But what is the next integer smaller than N_{large} that can be represented exactly in double precision?

The next smaller integer should be the same double-precision number, but with the least significant bit flipped to zero:

$$N_{\text{next largest}} = (1 + 2^{-1} + 2^{-2} + ... + 2^{-51} + 0 \times 2^{-52}) \times 2^{1023}.$$

Notice that the difference between the largest and second-largest integer exactly representable in double precision is $N_{\text{largest}} - N_{\text{next largest}} = 2^{1023-52} \gg 1$, so these two integers *are not contiguous.* So what would be the largest integer such that the next largest integer is just one less?

Since the significand is a number that is 52 bits, and we multiply by 2^L, where L ranges from -1022 to 1023, how about we multiply the largest possible significand by 2^{52}? Then we'd have

$$\begin{aligned} N_{52} &= (1 + 2^{-1} + 2^{-2} + ... + 2^{-52}) \times 2^{52} \\ &= (2^{52} + 2^{51} + 2^{50} + ... + 1) \\ &= 2^{53} - 1 \approx 9 \times 10^{15}. \end{aligned}$$

Ah, but we can also represent 1×2^{53} exactly as well! Notice also that the next integer smaller than N_{52} would be

$$\begin{aligned} N_{52,\text{ next largest}} &= (1 + 2^{-1} + 2^{-2} + ... + 2^{-51} + 0 \times 2^{-52}) \times 2^{52} \\ &= (2^{52} + 2^{51} + 2^{50} + ... + 2^1 + 0) \\ &= N_{52} - 1. \end{aligned}$$

Next, you should be able to show that $2^{53}+1$ is not an integer representable exactly in double precision. Thus we have found that $2^{53}-1$ and 2^{53} are the largest two consecutive integers that can be stored exactly in double precision. You may also find it instructive to prove that $N_{52}-2$ and $N_{52}-3$ can also be stored exactly, and that the pattern continues all the way to $N=0$. But we also know that if we flip the sign bit, we can get the negative versions of each of these integers.

Therefore double precision arithmetic can store the complete set of contiguous integers between -2^{53} ($\approx -9 \times 10^{16}$) and 2^{53} ($\approx 9 \times 10^{16}$), inclusive.

3.2.2.2 Summary: Limitations of Double Precision

Limitations of Double Precision

- **Definition**: **Machine epsilon** is the largest number ϵ representable in a given number format such that $1+\epsilon = 1$. In double precision, this number is 2^{-53} ≈`1.11e-16`. **Warning #1:** There is no IEEE 754 standard definition of machine epsilon; you fill find other texts define machine epsilon differently.[a] **Warning #2**: Different implementations of the IEEE 754 standard might behave differently near machine epsilon, yielding machine epsilons of `2.22e-16` or `4.44e-16` instead. **For simplicity, we will normally assume 16 decimal digits of significance for double precision.**
- **Magnitude of smallest nonzero number**: 2^{-1022} ≈`2.22e-308`. We will typically round to 10^{-308} in this class.
- **Magnitude of largest non-infinite number**:

$$2^{1023}(1+2^{-1}+2^{-2}+...+2^{-51}+2^{-52}) = 2^{1023}+2^{1022}+...+2^{1023-52} \approx \texttt{1.8e308}$$

  We will typically round to 10^{+308} in this class.
- **Largest set of contiguous integers**: From $-2^{53} \approx$ `-9e15` and 2^{53} ≈`9e15`.

[a]One common alternative definition is the smallest number in a given number format that, when added to 1 yields a number not equal to one. This is not the definition we will use in this class.

3.2.2.3 Loss of Significance

(References/Sources: `https://en.wikipedia.org/wiki/Loss_of_significance`
`http://math.stackexchange.com/questions/136634/rewriting-to-avoid-catastrophic-cancellation`)

Definition: Loss of significance or **catastrophic cancellation** occurs when significant digits are lost in an arithmetic operation. Suppose our number format stores 8 significant digits. Then

`1.1204002e-1 - 1.1204000e-1`

will yield `2.e-8`. However, because our number format stores 8 significant digits, *we can expect that for this case, the numbers after the first significant digit will be nonzero!*

Loss of significance can be a major problem when performing double precision arithmetic, because we typically assume that we can trust our double precision results to about 16 significant digits. Your computer will not warn you when a catastrophic cancellation has occurred, and doing so would greatly slow your calculation. We must therefore be able to identify when a loss of significance has occurred *by analyzing the expressions ourselves*. Typically this will involve adding `print()` statements to our code when we obtain results that are unusual. Numerical analysis can be hard work, but when we are successful, our efforts can be greatly rewarded!

Exercises:

You have added `print()` statements into your code to try and diagnose why your numerical solution (using double precision) appears to be strange. Below is a list of arithmetic operations that your code has performed, and you are to write the number of significant digits that remain after the following expressions are evaluated in double precision. You are to assume that double precision numbers input into all expressions are known to 16 significant digits.

1. `1.1204002e-1 - 1.1204000e-1` (same as example above, but in double precision)
2. `5.2101e-5 - 5.2101e-4`
3. `5.2111e-5 - 5.2101e-5`
4. `5.2111e-5 - 5.2101e-6`
5. `5.211104e-5 - 5.211101e-5`
6. `5.2101e-5 - 5.2101e-5`

Solutions to Exercises:

1. 7 significant digits are lost, so 9 remain.
2. No digits are lost, so 16 significant digits remain.
3. 3 significant digits are lost, so 13 remain.
4. No digits are lost, so 16 significant digits remain.
5. 6 significant digits are lost, so 10 significant digits remain.
6. There is exact cancellation in this case (each number is represented by exactly the same bits), so double precision will yield zero exactly, resulting in a *gain* of an *infinite* number of significant digits. In practice, this will happen rarely.

What happens to the lost significant digits?

Although digits of significance may be lost when performing double-precision arithmetic, the arithmetic will always produce a consistent result. For example, if you were to set

```
a = 2.150121e0 - 2.150120e0
b = 2.150121e0 - 2.150120e0
```

Even though we have lost about 6 significant digits of `a` and `b`, the result from these arithmetic operations will always be consistent. In this case, although we can trust the results of the subtractions up to about the tenth significant digit, all digits beyond the tenth in both `a` and `b` will be *identical*, as the bits representing `a` and `b` are identical. Therefore, if we were to evaluate, `a-b`, we would obtain the exact result *zero*.
Next consider

```
c = 8.924583e0-8.924582e0
```

Again, the exact result is `1e-6`, and we should expect our double-precision result to match this exact result to about 10 significant digits (and not 16, due to the loss of significance). However, in this case the double-precision numbers involved in the subtraction are different. Thus the digits beyond the tenth in `c` *will in general be completely different* than `a` or `b`. Therefore, we can have *no* expectation of consistency between `c` and `a`, and we should expect `c-a` to evaluate to a number that is nonzero.

3.2.2.4 Tips for determining how many significant digits of agreement SDA (see Chapter 2) you can expect between double-precision and exact arithmetic

Note that this neglects guard digits.

1. Evaluating the expression according to proper order-of-operation, check for arithmetic steps that go out of bounds (i.e., check for **underflow** or **overflow** errors) in double precision arithmetic. To this end, recall that the smallest nonzero number is roughly plus or minus `1e-308` and the largest non-infinite number has magnitude `1e+308`. If this happens, evaluate to zero or $\pm\infty$ as appropriate.
2. Check for catastrophic cancellation (a.k.a., loss of significance).
3. Check for numbers that are exactly representable by double precision (e.g., all integers between -2^{53} and 2^{53} (inclusive), as well as $\pm\frac{1}{2}$ to some power). If these exist, they are known to an infinite number of significant digits. If not, they are generally known to only 15–16 significant digits.
4. Dividing or subtracting two numbers that are guaranteed identical to all significant bits will yield *exactly* one or zero, respectively.
5. Transcendental functions like `sin()`, `cos()`, `log()`, and `log10()` are typically evaluated in double precision using a Taylor series approximation. So even if the input and result are both numbers exactly representable in double precision, you should generally expect only 16 significant digits of precision in the result.

3.2.3 Examples of Finite Precision Arithmetic Using Double Precision

Exercises:

When the following expressions are evaluated by the computer, **to how many significant decimal digits will the numerical result agree with the exact result?** Your answer will consist of a single integer (∞ is an acceptable answer). If the integer is finite and nonzero, your answer will be accepted if it is within 1 decimal digit of the exact answer.
We use computer scientific notation, such that, e.g., `5.63e22` $\equiv 5.63 \times 10^{22}$. For the purpose of this problem, apply IEEE 754 standard-compliant **double precision** arithmetic, assuming all given *integers* represented in integer or floating point format between -2^{53} and 2^{53} (i.e., integers between $\approx -9 \times 10^{15}$ and $\approx 9 \times 10^{15}$ inclusive) are **exact** (for example, `2.01e2` is exact), as well as powers of $\pm\frac{1}{2}$. Otherwise, assume that in double precision the number is only known to 16 significant digits, and that machine epsilon is `4e-16`.

1. `4 - 1`
2. `1e23/2.312e101 + 1e-20`
3. `1e20 - 1e200/1e-200`
4. `(1e-250/1e-250)`
5. `1.3512e-45 - 1.3512e-45`
6. `(1e55 - 1e35) + 1e55`
7. `1 - 1e-12`
8. `(1e120 * 1e120 * 1e120)`
9. `1.0/256.0`
10. `(2.51-2.50)/(2.51-2.50)`
11. `log10(1e-145) - 2e-200`
12. `5e-200*5e-2 + 2`
13. `7 + 2.4e-230/5e280`
14. `7 + 2.4e230/5e280`
15. `1/100`
16. `log10(2e-230) - 2e-200`
17. `(3.1415-3.14)/(3.1415-3.14)`
18. `1/1024`
19. `3.2e-200/5e-202 + 2`
20. `(5.0e-200 + 1)/(1.2e1+4e0)`

Additional problem set. Same instructions as previous problem. Solutions provided in class.

Additional Exercises:

1. `1e-1/1.0e-1`
2. `2e10 - 1e-100`
3. `4.34/2`
4. `1.0 + (5.0e33 - 1e-14)`
5. `8.0e200 + 1.0`
6. `(2.104e4 - 2.103e4)/1.0e1`

Solutions to Exercises:

1. `4 - 1`: 4 and 1 are exactly represented in double precision, and we have already proven that 3 is also represented exactly in double precision. The double-precision result agrees with the exact result to an infinite number of significant digits.
2. `1e23/2.312e101 + 1e-20`: Evaluating, we get about $0.5 \times 10^{23-101}$. $23 - 101 = -78$, so we get $0.5 \times 10^{-78} + 10^{-20}$. The first number in the sum is about 59 orders of magnitude less than the second. Additions in double precision are only sensitive to about 16 significant digits, so the first number is irrelevant and the result in double precision is 10^{-20}. *Ignoring the addition*, the number 10^{-20} cannot be exactly represented in double precision. Therefore this result is consistent with the exact solution to 16 significant digits.
3. `1e20 - 1e200/1e-200`: `1e200/1e-200=1e400`, which exceeds the bounds of double precision, thus we get Infinity, so the answer is zero digits of precision.
4. `(1e-250/1e-250)`: `1e-250/1e-250=1`. Note that even if `1e-250` is not exactly representable in double precision, its representation in bits is the same for both numerator and denominator. So imagine you have a string of bits divided by the same string of bits. You'll get 1 exactly, and 1 is exactly representable by double precision. Thus the answer is: the double-precision result is consistent with the exact result to ∞-many digits of precision.
5. `1.3512e-45 - 1.3512e-45`: Imagine you have a string of bits subtracting the same string of bits. You'll get 0 exactly, and 0 is exactly representable by double precision. Thus the answer is: the double-precision result is consistent with the exact result to ∞-many digits of precision.
6. `(1e55 - 1e35) + 1e55`: Notice the catastrophic cancellation that occurs within the parentheses. So
   `(1e55 - 1e35) + 1e55 = 1e55 + 1e55 = 2e55`,
   which is consistent with the exact result to 16 significant digits.
7. `1 - 1e-12`: Notice that 1 is exact, but `1e-12` is not. There is no catastrophic cancellation here; we get an answer that is consistent with the exact result to 16 significant digits.
8. `(1e120 * 1e120 * 1e120)`: The result, `1e360`, exceeds the bounds of double precision. Thus we get an overflow: Our answer will be correct to zero significant digits.
9. `1.0/256.0`: Both 1 and 256 are exact. Also, $1/256 = 1/2^8 = 2^{-8}$ is exactly representable in double precision as well. Thus the answer is: the double-precision result is consistent with the exact result to ∞-many digits of precision.
10. `(2.51-2.50)/(2.51-2.50)`: Let's look at the numerator first: `(2.51-2.50)`. Neither of these terms is an exact power of $\frac{1}{2}$ or an integer, and the result, 0.01 is also not an exact power of $\frac{1}{2}$. We will get 0.01 to 14 significant digits due to the loss of significance. But the denominator will result in exactly the same number: 0.01 to about 14 significant digits. Therefore, the numerator and denominator will be identical strings of bits dividing each other: 1 exactly. So the answer is ∞.
11. `log10(1e-145) - 2e-200`: Note that `log10(1e-145)` will be computed via a Taylor series on the computer, so despite the fact that the exact result is an integer, -145, which can be exactly represented in double precision, the Taylor series approximation can only be trusted to 16 significant digits! So we get `-145 - 2e-200`. Notice the catastrophic cancellation on the second term. So the answer is `-1.45e2` to 16 significant digits.
12. `5e-200*5e-2 + 2`: Notice the first term evaluates to `25e-202=2.5e-201`, so we get catastrophic cancellation. Since 2 is exactly represented in double precision, the double precision result is consistent with the exact result to about 201 significant digits.
13. `7 + 2.4e-230/5e280`: 7 is exactly representable, and the expression `2.4e-230/5e280` will yield an underflow in double precision, giving a double precision result of `7` *exactly*. The exact result is `7` plus a perturbation at the 511th digit. Thus the double precision and exact results agree to about 511 significant digits.
14. `7 + 2.4e230/5e280`: 7 is exactly representable, and the expression `2.4e230/5e280` will yield a number that is of order `1e-51`. Recall that machine epsilon is defined as the largest ϵ such that $1 + \epsilon = 1$. In double precision machine epsilon is approximately `4e-16`, so `7+1e-51 ~ 7 (1 + 1e-52)` is `7` exactly. Thus the double precision and exact results agree to about 51 significant digits.
15. `1/100`: The division is not a power of 2, so the answer is 16 significant digits of agreement between double precision and the exact answer.
16. `log10(2e-230) - 2e-200`: `log10(2e-230)` is computed via Taylor series, thus will yield a double precision result that matches the exact result to 16 significant digits. The `-2e-200` will not affect the result. The answer is 16.
17. `(3.1415-3.14)/(3.1415-3.14)`: The numerator and denominator will evaluate to exactly the same bits, despite being consistent with the exact result to 16 significant digits. Thus the answer in double precision is 1 exactly, matching the exact result to ∞ significant digits.
18. `1/1024`: $1024 = 2^{10}$, so the expression is 2^{-10}, an exactly representable number in double precision that matches the exact result to ∞ significant digits.
19. `3.2e-200/5e-202 + 2`: `3.2e-200/5e-202` is a number of order `1e1`. The result is *not* a number exactly representable in double precision, so will be consistent with the exact solution to 16 significant digits.
20. `(5.0e-200 + 1)/(1.2e1+4e0)`: `(5.0e-200 + 1)` in double precision yields `1` exactly, which differs from the exact result at the 200th significant digit. The denominator will evaluate to an exact integer since `1.2e1` and `4e0` are both less than 2^{52} and there is no loss of significance. The double precision result then is `1/16`, which is an exact power of 2, yielding the number `1/16` exactly. This is consistent with the exact result to about 199 significant digits.

3.2.3.1 Adjusting an Algorithm to Minimize Loss of Significance

Let us now consider an algorithm that suffers from a severe loss of significance in certain cases: the quadratic formula.

The quadratic formula solves for the roots x_1 and x_2 that solve the following quadratic polynomial

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

The quadratic formula is given by

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

In pseudocode, we could represent this algorithm as follows:

Algorithm for Solving Quadratic Polynomials:

```
function quadratic(a, b, c, x)
   x[1] = ( -b + sqrt(b^2 - 4*a*c) ) / (2*a)
   x[2] = ( -b - sqrt(b^2 - 4*a*c) ) / (2*a)
end function
```

Notice that when $b^2 \gtrsim 10^{16} \times 4ac$, $\sqrt{b^2 - 4ac} \equiv b$ in double precision. Thus for the first root we would get $x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} = \frac{-b+b}{2a} = 0$, which will in general be incorrect to *all* significant digits.

For example, consider the case in which $a = 1$, $b =$`1e10`, and $c =$`1e-10`. The exact roots are given by $x_1, x_2 = -10^{-20}, -10^{10}$, but the roots computed in double precision will be given by

$$\begin{aligned} x_1, x_2 &= \frac{-10^{10} \pm \sqrt{(10^{10})^2 - 4(1)(10^{-10})}}{2(1)} \\ &= \frac{-10^{10} \pm \sqrt{10^{20} - 4 \times 10^{-10}}}{2} \\ &= \frac{-10^{10} \pm \sqrt{10^{20}(1 - 4 \times 10^{-30})}}{2} \\ &= \frac{-10^{10} \pm \sqrt{10^{20}(1)}}{2} \\ &= 0,\ -10^{10}, \end{aligned}$$

where in the second-to-last step, we applied the fact that in double precision $(1 - 4 \times 10^{-30}) \equiv 1$, since $4 \times 10^{-30} < \epsilon_m$ (note that machine epsilon equally applies whether adding to *or subtracting from* 1). So we have found that x_1 is incorrect at *all* significant digits, while x_2 will be correct to about 16 significant digits. We conclude that was a *complete loss of significance* in computing x_1.

Sometimes, *but not always* there is an alternative strategy for getting around limitations of double precision arithmetic. In this case, it turns out that we can rewrite the quadratic formula for x_1 as follows:

$$\begin{aligned} x_1 &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \left(\frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} \right) \\ &= \frac{b^2 - (b^2 - 4ac)}{2a(-b - \sqrt{b^2 - 4ac})} \\ &= \frac{2c}{-b - \sqrt{b^2 - 4ac}} \end{aligned}$$

Now let's apply this example, but with this alternative strategy for computing the root of x_1:

$$\begin{aligned} x_1 &= \frac{2c}{-b-\sqrt{b^2-4ac}} \\ &= \frac{2(10^{-10})}{-(10^{10})-\sqrt{(10^{10})^2-4(1)(10^{-10})}} \\ &= \frac{2\times 10^{-10}}{-10^{10}-\sqrt{10^{20}-4\times 10^{-10}}} \\ &= \frac{2\times 10^{-10}}{-10^{10}-\sqrt{10^{20}(1-4\times 10^{-30})}} \\ &= \frac{2\times 10^{-10}}{-10^{10}-\sqrt{10^{20}(1)}} \\ &= \frac{2\times 10^{-10}}{-2\times 10^{10}} \\ &= -10^{-20}, \end{aligned}$$

which will be consistent with the exact result to 16 significant digits. Notice that while we did not eliminate the loss of significance, we did prevent it from influencing our result.

Exercise:

Rewrite the above algorithm for solving quadratic polynomials to make it more robust for cases in which $b^2 \gtrsim 10^{16} \times 4ac$.

Solution:

```
function quadratic(a, b, c, x)
  if(b^2 > 1e16*4*a*c)
    x[1] = (2*c) / ( -b - sqrt(b^2 - 4*a*c) )
  else
    x[1] = ( -b + sqrt(b^2 - 4*a*c) ) / (2*a)
  end if
  x[2] = ( -b - sqrt(b^2 - 4*a*c) ) / (2*a)
end function
```

3.2.3.2 Tips for minimizing finite precision arithmetic errors

Here are a few tips for minimizing finite precision arithmetic errors on the computer.

1. Avoid obvious addition or subtraction of numbers that are many orders of magnitude different. This may require rescaling your problem so that extremely large or small exponents do not appear. (Top priority)
2. **Definition**: Choose numerical algorithms that have small **unavoidable errors**; i.e., algorithms that naturally avoid loss of significance (like the adjusted quadratic equation formula given above). Using an algorithm that has small unavoidable errors *is no guarantee* that catastrophic cancellation will not occur for certain inputs! (High priority)
3. **Definition: Roundoff error** is the build-up of numerical errors due to loss of significance. Once the above problems are addressed, try to minimize the number of floating point operations (i.e., generally more efficient algorithms have lower roundoff errors). Roundoff errors will generally grow at a rate between $\sqrt{N}\epsilon$ (due to the expected distance traveled from the origin in a random walk after N steps) and $N\epsilon$ (a non-random walk, adding a single ϵ error at each step), *unless the numerical algorithm exhibits chaotic behavior.* If chaotic behavior is observed, tiny perturbations due to roundoff errors can grow at a rate proportional to e^N. (Medium priority)

Solutions to Additional Exercises:

1. `1e-1/1.0e-1`: The numerator and denominator are represented by exactly the same bits, so the double precision result is the number `1`, which is exactly representable and consistent with the exact result to Infinitely many significant digits.
2. `2e10 - 1e-100`: `2e10` $< 2^{52}$, so is exactly representable in double precision. `1e-100` is not exactly representable. There is a catastrophic cancellation in double precision, leading to the double precision result of `2e10`, which is exactly represented as an integer. The exact result is not an integer, and differs from the double precision result at the 110th significant digit. Thus the answer is 110.
3. `4.34/2`: `4.34` is not exactly representable in double precision, and it is known only to 16 significant digits. Thus dividing it by two will yield a number correct to only 16 significant digits.
4. `1 + (5.0e33 - 1e-14)`: `1` is exactly representable in double precision, but `5.0e20` and `1e-14` are represented only to 16 significant digits. `(5.0e20 - 1e-14)` will yield `5.0e20` to 16 significant digits due to catastrophic cancellation. Then `1 + (5.0e20)` will yield `5.0e20` to 16 significant digits, again due to loss of precision.
5. `8.0e200 + 1`: `8.0e200` is represented in double precision to 16 significant digits, and `1` is exact. The sum yields `8.0e200` to 16 significant digits due to loss of precision.
6. `(2.104e4 - 2.103e4)/1.0e1`: All numbers in the numerator and denominator are represented exactly in double precision. Evaluating numerator and denominator separately, we obtain `1.0e1/1.0e1`, an exactly representable number in double precision. Thus the double precision result agrees with the exact result to an infinite number of significant digits.

MATH 521, Fall 2018 Notes

Prof. Zachariah B. Etienne

Chapter 4: Determining the Scale of the Solution

We will find that many of the algorithms we encounter in this class *will not work properly* unless we first have some very rough idea (like, within a factor of a few) of the solution. Having some idea of the scale of the solution will also help us determine whether the computer's result is reasonable.

To get some sense of the scale of the solution without knowing the solution requires that we use all the information at our disposal to generate an estimate that is reasonably close. This is known as a "back-of-the-envelope" estimate or a "Fermi" problem, after Enrico Fermi—a very famous, Nobel-prize winning physicist—as he was a master at this sort of back-of-the-envelope calculations. Fermi worked on the Manhattan Project–the project that developed the first atomic bombs. During the first nuclear test, Fermi dropped pieces of paper from his viewing position to estimate the energy in the blast. His measurement came quite close to the official measurement, which made use of many sensors.

Being able to reason your way through back-of-the-envelope estimates make excellent job interview question for those of you who want to get really high-paying jobs in the financial or tech sectors. These sorts of question could be asked cold in an interview, meaning you have no access to books... just a whiteboard and your own numerical literacy. As you solve the problem in front of the interviewers, they will learn how you solve problems, as well as your basic numerical literacy.

4.1 Word Problems

The easiest Fermi problems boil down to basic word problems that require simple arithmetic to solve. Note that a typical Fermi problem will require basic numerical literacy (e.g., knowing the population of the United States), working knowledge of undergraduate mathematics (e.g., being able to integrate, differentiate, and compute Taylor expansions without help), and good analytical reasoning skills.

As a warm-up, let's suppose you are a graduate student who wishes to increase efficiency and productivity.

Example #1, Question #1

Graduate students are generally expected to work as a GTA or GRA 20 hours per week, spending another 20 hours on course work. This is of course the ideal scenario, assuming that there is not an urgent research project, midterm to grade, or exam to take. It also assumes that students are well prepared for their graduate level classes so that the amount of "brushing up" on previous material is minimized. If any of these assumptions are violated, the 40-hour week may become a 60-hour week or in some cases a 20 or 30-hour week.
Students on average take 3 courses per semester. Assume that each class is 150 minutes per week. If they are on average expected to spend 20 hours per week on classes, how many hours outside of class *per class* in an ideal situation are they expected to study?

Answer: First we compute the number of hours *in total* outside of class that should be spent studying. Notice we must first convert to a consistent unit of time (hours) so we can perform the necessary arithmetic:

```
20 hours/week - 150 minutes/course/week * (1 hour/(60 minutes)) * (3 courses)
   = (20 - 7.5) hours/week = 12.5 hours/week
```

Thus students are expected to spend 12.5 hours per week on their coursework outside of class, or about 12.5/3 hours per course, which comes to about 4 hours, 10 minutes studying outside of class for each class. Again, this is an ideal situation, as a student's level of preparation may be different for different classes, requiring additional time commitment for some classes.

Example #1, Question #2

Distracted Dan is not very focused on studying, keeping his cellphone out or the TV on. Each hour he studies is worth only 60% of the expected study time per week. Roughly how many hours would he need to study for each of his classes, if he is an otherwise average student? Give the answer as a simplified fraction and an approximate (i.e., within 10% of exact) decimal.

Answer: We determined in the previous problem that Dan must spend about 12.5 hours studying outside of class each week. Thus he must spend 12.5/3 hours per course per week, which amounts to a little over 4 hours per course. `12.5/3 = 25/6`, so the amount of time needed each week for studying outside of class per class for Distracted Dan will be `25/6 * 10/6 = 250/36 = 125/18 = (125*1.1)/(18*1.1)≈140/20 = 70/10 = 7` hours. The exact value is about 6h 57m.

Example #1, Question #3

Based on Question #2 above, about how many hours per week would Distracted Dan be expected to study on average outside of class to keep up with all of his classes?

Answer: Distracted Dan takes three courses each semester, and we just determined that the total study time outside of class required for Distracted Dan per course is nearly 7 hours, so the answer is that Distracted Dan must study about `3*7 = 21` hours outside of class to keep up with the average student.

Add this to the 7.5 hours spent inside of class, and Distracted Dan's 20 hour per week course load has ballooned to nearly 30 hours per week! The moral of the story is that students should not only work hard, but strive to work *efficiently* as well. Try to minimize distractions!

When we apply the concepts learned in this class to the real world, we will find that this sort of thinking is extremely valuable to determining the scale of physical systems as well.

4.2 Undersampling Error

Performing integrals, derivatives, interpolation, and extrapolation of functions on the computer is central to much of the work we do as when we solve mathematical problems on the computer. Unfortunately, performing these operations reliably on the computer usually requires that we first determine the basic scales over which the function we wish to integrate, differentiate, etc., varies.

For example, when evaluating a derivative approximately on the computer, we start from the definition of the derivative

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

to say that, *for Δx small enough*,

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

Approximations like this are made widely when solving mathematical problems on the computer, because

1. it is difficult to teach computers how to differentiate exactly;
2. sometimes we are only given the function values at a finite number of points; and
3. computers are designed to subtract, add, multiply, and divide extremely quickly.

The question we must answer is, *how small of a Δx will be necessary to obtain a reasonable approximation of the derivative?*. Or equivalently, *how finely must we sample the function to get a good approximation of its derivative at some arbitrary point x?*

The answer of course depends on the function. If x is in units of length for example, then we must first estimate the *lengthscale* over which $f(x)$ varies. If we chose to approximate the derivative of $f(x) = \sin(x)$ at any point x using the definition of derivative in this way, hopefully you would agree that if we chose $\Delta x = 10$, we should not expect the answer to be anywhere close to the correct result.[1] We refer to this type of error as **undersampling error**.

[1]After all, if x is related to units of length, then $\sin(x)$ varies over a lengthscale of $2\pi \approx 6.3$–its *wavelength*. If instead x is related to units of time, then we would say $\sin(x)$ varies over a timescale of $2\pi \approx 6.3$–its *period*.

Let's make this notion of undersampling just a little more rigorous by defining the minimal, or "Nyquist", sampling rate.

4.2.1 Minimal, or "Nyquist" Sampling

Suppose we have a function that is a wave with wavelength $\lambda = 1$ meter. We refer to λ as the *lengthscale* of the problem, and if we wish to interpolate the value of the function to any point, we will need to sample the function at a resolution of Δx that is some small fraction of λ.

In fact, the minimum sampling rate for a wave, called the *Nyquist sampling* frequency or rate, is $\Delta x = \lambda/2$. Figure 4.1 shows this graphically. This idea can be applied more generally, even to *non-periodic* functions, by simply replacing λ with *the scale over which the function varies.* The next section explores this more generic notion by analyzing the scale over which a solution will vary and applying it to estimate computational cost.

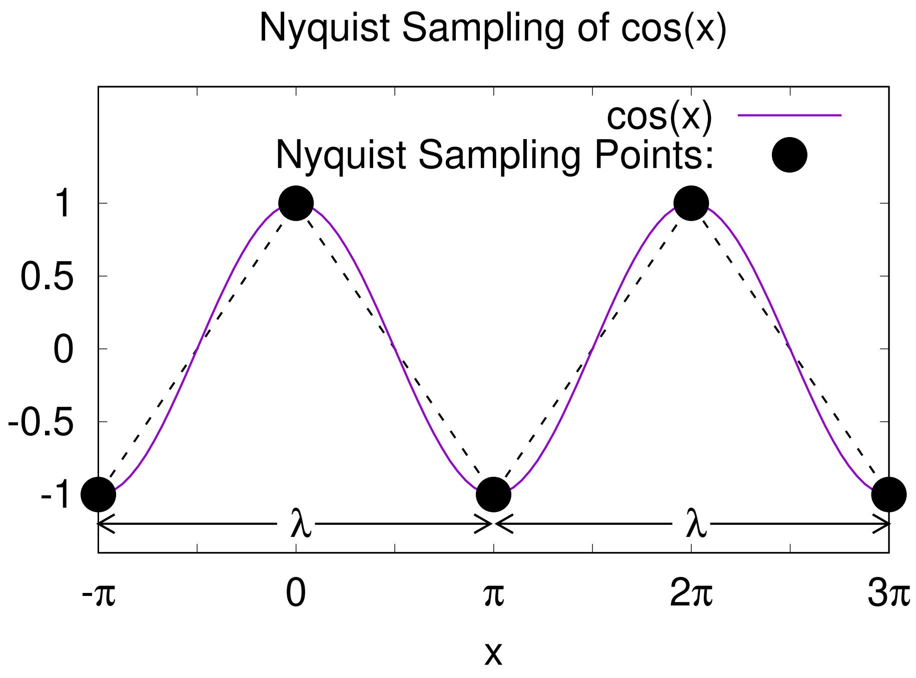

Figure 4.1: Minimal (Nyquist) sampling of a cosine function (solid curve). Nyquist-sampled points, at $\Delta x = \lambda/2$, are shown as black dots, which are connected by dashed lines.

4.3 Tying It All Together: Estimating the Cost of Weather Modeling

Example #2

Suppose we wish to produce a climate model that simulates cloud motion. Let's assume that the tiniest of clouds are negligible, so that our model is aiming to model large cloud movement. Provide an estimate for the minimum number of numerical points that we would need to Nyquist sample all the clouds in the Earth's atmosphere? Ignore the vertical direction.

One way of solving this problem is to estimate the number of distinct clouds we observe in the sky on average. Not every day we see clouds, so let's say 2 of 3 days are cloudy days, and when we consider that Morgantown is a bit cloudier than the average city (or desert for that matter), let's drop that to 1 of 2 days. Then on cloudy days, the average cloudiness is about 30%. So the average fraction of the local sky taken up by clouds is about 30% times $1/2 = 15\%$.

Now this is at our local position on Earth, so we need to know the fraction of the entire Earth's atmosphere we can view at a given time.

The Earth is roughly 6,000 km in radius (actual radius is around 6,300 km).

How do we figure out the fraction of the sky that we see? Well, this comes down to measuring our horizon. Our horizon is defined as the plane tangent to our location on the Earth. We obviously cannot see below the horizon (though there are some exceptions due to the way the atmosphere bends light passing through it). We can see clouds so long as they are above this tangent plane.

Suppose clouds are at 15,000 feet altitude on average. There are about 3 feet in a meter, so we have 5,000m or 5 km for the average altitude for clouds. So we get a right triangle: the line from us to the center of the Earth is about 6,000 km, and drawing another line from the center of the Earth to the cloud height on our horizon. Connecting these lines yields a right triangle, in which we stand at the right angle, with hypotenuse 6,005 km. Our goal is to find the subtended angle of our horizon from the center of the Earth.

What is this angle? Basic right-triangle geometry tells us that the angle θ satisfies

$$\cos(\theta) = \frac{6000}{6005}.$$

Of course this is very close to 1, which means that the angle must be very small. So let's Taylor expand $\cos(\theta)$ about $\theta = 0$:

$$\cos(\theta) = 1 - \frac{\theta^2}{2} + \frac{\theta^4}{4!} - ...$$

Since θ^2 is small, θ^4 is extremely small. so we can get a very good approximation of the angle by simply solving

$$1 - \frac{\theta^2}{2} = \frac{6000}{6005} \approx 1 - \frac{1}{1000}.$$

Thus the total horizon for us if we can see clouds at 15,000 feet is

$$\theta^2 \approx \frac{2}{1000} \implies \theta \approx \sqrt{\frac{2}{1000}} = \frac{1}{\sqrt{500}}.$$

What is the square root of 500? The square root of 100 is 10, so

$$\sqrt{100} = 10 \implies 2\sqrt{100} = 20 \implies \sqrt{400} = 20.$$

Also,

$$\sqrt{100} = 10 \implies 3\sqrt{100} = 30 \implies \sqrt{900} = 30.$$

So we're somewhere between 20 and 30, but closer to 20. Let's just keep 20 for our approximation.

Then the angle subtended by our horizon when looking in one direction is about 1/20 of a radian. Our goal then is to determine the area of the sky we can see as a total fraction of the overall surface area of the Earth.

Recall that the surface area of a sphere with radius R is derived from the integration formula

$$S_{\text{sphere}} = R^2 \int_0^{\pi} \int_0^{2\pi} \sin(\theta) d\theta = 4\pi R^2.$$

We are interested however in the surface area of some circular-like area subtended by an angle $\Delta\theta$ on the sphere, so

$$S_{\text{circ area}} = R^2 \int_0^{\Delta\theta} \int_0^{2\pi} \sin(\theta) d\theta = 2\pi[-\cos(\theta)]_0^{\Delta\theta} R^2.$$

We can use this formula, combined with the $\Delta\theta = 1/20$ to compute the surface area of the Earth's atmosphere over which clouds are visible to us:

$$\begin{aligned} S_{\text{area of our horizon}} &= 2\pi R^2[-\cos(\theta)]_0^{1/20} = 2\pi R^2 \left(-\cos\left(\frac{1}{20}\right) + 1\right) \\ &= 2\pi R^2 \left(1 - \cos\left(\frac{1}{20}\right)\right) = 2\pi R^2 \left(1 - \left(1 - \left(\frac{1}{20}\right)^2 \frac{1}{2!}\right)\right) = \pi R^2/400. \end{aligned}$$

From this we can compute what fraction of the Earth's atmosphere we can see at a given location on Earth:

$$\frac{S_{\text{area of our horizon}}}{E_{\text{sphere}}} = \frac{\pi/400}{4\pi} = 1/1600.$$

Notice the factors of R^2 canceled out.

Let's now use the assumed average fraction of the sky taken up by clouds from our perspective, of 15%. Let's assume the clouds are bunched up to take up 15% of our sky exactly. Then since our sky is only 1/1600 the total sky, this means that this one "fake" cloud takes up an angular fraction of the Earth's atmosphere of 15% of 1/1600, or $15/100 \times 1/1600 = 15/1600 \times 1/100 \approx 1/10,000$. So if we resolve one "cloud" by 3 points in each angular direction (Nyquist sampled), we need 9 points per cloud. Multiply that by 1,600 such "clouds", and we end up with about 15,000 numerical points required to Nyquist sample the cloud cover on Earth. Generally we'd like to resolve each feature in our solution by at least 10 points in each direction, so instead of 9 points, $10^2 = 100$ points per cloud would be far better. Also the clouds won't clump up as we assumed, so if there are a total of 20 clouds in the sky, we'll need 20 times more points. So a better model would use $20 \times 100 \times 15,000$, or about 30 million (3×10^7) points.

4.4 Other Back-of-the-Envelope Estimation Examples

You should be able to estimate based on your existing knowledge alone, without the help of any external sources, answers within an order of magnitude to the following questions.

Exercises:

1. What is the total number of miles driven in passenger vehicles each day within the United States? Based on this, how much gasoline is consumed each day within the United States in gallons? (There are 1.6 km per mile, and 3.8 liters per gallon.)
2. You are working as a digital artist and want to create a realistic 3D model of a healthy forest in the summer time, but before starting the project you smartly decide to do some calculations to determine whether your workstation might be overwhelmed by the task. To this end, you must first compute the number of leaves on a typical large, healthy, leafy (deciduous) tree, to within an order of magnitude. Provide this estimate.
3. You would like to know how much faster your computer code will run in one year. Moore's Law says that CPU speed will double every 18 months. Use the Rule of 72 to compute by what factor CPU speed increases each year, according to Moore's Law. Your final answer must be correct up to and including the second significant digit. *Hint: Recall the Rule of 72 says that for a growth rate of x% per unit time, the amount of time necessary for doubling be given by 72/x.*
4. The Rule of 72 can also be used to estimate the time needed to remove half of some exponentially decaying quantity. Given this, use the Rule of 72, along with the inflation rate of 3% per year to estimate the value of a United States dollar 20 years and 100 from today. I.e., you are to assume that the dollar lowers in value 3% each year.
5. To estimate the distance to an object, we often use parallax. Parallax distances depend on the use of small-angle approximations, which in turn are based on Taylor series expansion of trigonometric functions. Use the Taylor series of the sine function $\sin(x)$ about $x_0 = 0$ to compute by hand $\sin(0.01)$ to four significant digits.
   You may find the following useful:
   Defining $f^{(n)}(x)$ to be the nth derivative of $f(x)$, the Taylor series expansion of a function $f(x)$ about a point $x = x_0$ is given by:
   $$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)(x - x_0)^n}{n!}$$

Solutions:

1. Assuming each driver in the US drives on average 30 miles per day, and one out of three people in the US drive every day (i.e., one-third of 3×10^8 people), we get about 3×10^9 miles per day, within an order of magnitude or so. At 20 miles per gallon average fuel economy, each driver on average consumes 1.5 gallons of gas. Thus we should expect roughly 1.5×10^8 gallons of gas per day. Official figures here: `https://tinyurl.com/oomagnitude1`
2. Roughly 10^6 leaves: `https://tinyurl.com/oomagnitude2`
3. Solution:

$$\begin{aligned}(72/x)\text{years} &= 1.5\text{years}\\ (72/x)\text{years} &= (3/2)\text{years}\\ \implies x &= 72*2/3 = 48\end{aligned}$$

   Thus each year, CPU speed will increase by 48%.
4. The Rule of 72 says that the US dollar will decrease in value due to inflation by one-half after $72/3 = 24$ years. Thus after 20 years, each dollar will be able to purchase only about \$0.60 worth of goods, and after 100 years, each dollar will be able to purchase only about $\$1.00\times\frac{1}{2^4} \approx \$0.06$ worth of goods.
5. The Taylor series expansion of $\sin(x)$ about $x_0 = 0$ is

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

   Therefore $\sin(0.01)$ is

$$\begin{aligned}\sin(0.01) &= 0.01 - \frac{10^{-6}}{6} + \frac{10^{-10}}{120} - ...\\ &\approx 1\times10^{-2} - 1.7\times10^{-7} + 1\times10^{-12} - ...\\ &\approx 9.999\times10^{-3}\end{aligned}$$

Prof. Zachariah B. Etienne

Chapter 5: Computational Cost and Computational Complexity (Big-$\mathcal{O}$ Notation)

5.1 Computational Cost: Floating Point Operations (FLOPs)

Often, when solving mathematical problems on the computer for research problems, we must be careful not to overwhelm the computational resources available to us. The exact amount of time needed to solve a problem will depend on how many **FL**oating point **OP**erations (FLOPs) are required, and the number of FLOPs per second (FLOPS) our computer can evaluate. Being able to count FLOPs in an algorithm enables us to reliably estimate the computational cost, and provides us insights into how we might make our algorithms more computationally efficient.

A single floating point operation (FLOP) can be defined as a variable assignment (=), multiplication ($\times$), addition (+), or subtraction ($-$). On modern CPUs, A single division ($\div$) typically requires between 3–5 FLOPs (`http://tinyurl.com/estimateFLOPS`).

Applying this definition, for floating point numbers a, b, c, d, and F, the expression

$$F = a * b - c * d$$

requires a total of 4 FLOPs (one assignment, multiplication on $a * b$, multiplication on $c * d$, and subtraction of the result).

As another example, for floating point numbers $a, b, c, ..., h$ and G, the expression

$$G = -(a * b - c * d)/(e * f + g * h)$$

requires a total of 8 floating point operations to evaluate all but the division (one variable assignment, the overall minus sign (`-1*`) is a multiplication, 3 in the numerator, and 3 in the denominator). The division could add between 3–5 FLOPs, yielding a total cost of between 11–13 FLOPs.

5.2 Common Subexpression Elimination (CSE)

Consider the following expression:

```
x = b * sin(2*a) + c / sin(2*a).
```

Computation of this expression requires one assignment, three multiplications, one division, two `sin()` function calls, and one addition. Multiplications, additions, and subtractions typically require one FLOP on a modern-day CPU, and divisions can require roughly 3–5 FLOPs each. Transcendental functions are far worse—each computation of a sine or cosine can require about 20 FLOPs on a modern-day CPU.

Our goal in writing efficient codes should be to minimize the number of FLOPs in our mathematical expressions. One way of doing this is to implement **common subexpression elimination**, or CSE for short.

Automatic CSE algorithms do exist. Generally they search for common patterns within expressions and declare them as new variables, so they need not be computed again.

Consider the following simplification of the above expression:

```
tmp = sin(2*a)
x   = b * tmp + c / tmp
```

For the cost of only one additional assignment, we potentially saved about 21 FLOPs by not computing `sin(2*a)` twice. *Special note: Compiled languages like C++ or FORTRAN sometimes implement decent automatic CSE algorithms, so optimizing code by hand like this might not yield the expected performance gains, as the compiler may attempt its own CSE.*

Words of caution

Since division on most modern-day computers is significantly more expensive than addition, multiplication, subtraction, and variable assignment, when rearranging expressions to make your computer program run faster, be sure that minimizing the number of divisions is a high priority!

Modern CPUs possess multiple memory caches, typically with lower-numbered memory caches being closer to the CPU (e.g., memory in the L1 cache on Intel and AMD CPUs can be accessed *far* more quickly than L2 or L3 caches; i.e., accesses from L2 or L3 cache can waste up to hundreds of CPU cycles). When it comes to CPU cache, accessing memory in higher-numbered caches takes far longer than in low-numbered caches. Sometimes this means that declaring large numbers of temporary variables in a CSE optimization might overload the L1 cache, causing a significant slowdown. Therefore one must be generally careful to consider CPU cache sizes. Note also that RAM (outside CPU cache) is even more slowly accessed, and hard disk reading/writing can be slower than RAM by *orders of magnitude*.

Finally, most compiled languages like C, FORTRAN, or C++ will make use of (with suitable compiler options) Single Instruction-Multiple Data CPU instructions (e.g., SSE2 or AVX instruction sets) when compiling mathematical expressions. These instructions enable multiple additions, subtractions, multiplications, or fused-multiply-adds to be performed *each CPU clock cycle*! As a result, you may find when using these languages that your code runs far faster than you might otherwise expect. Writing codes with this—and the behavior of CPU caches—in mind can be very powerful ammunition to optimizing your algorithms *by orders of magnitude*!

5.3 Computational Complexity (Big-$\mathcal{O}$ Notation)

Some algorithms are more efficient than others, and big-$\mathcal{O}$ notation can be used as a measure of an algorithm's efficiency.

Imagine we have a cluster of N stars. Suppose we wished to calculate the gravitational force $\boldsymbol{F}_i$ on a given star i. The force would be equal to

$$\boldsymbol{F}_i = \sum_{i\neq j}^{N} \boldsymbol{F}_{i,j},$$

where $\boldsymbol{F}_{i,j}$ is simply the force on star i from some other star j. Clearly to calculate the gravitational pull on star i, we need to add a total of $N-1$ forces. In order to model the motions of all the stars in this star cluster, we must compute $\boldsymbol{F}_i$ for all N stars in the cluster. As you know, addition is an operation allowed by floating point numbers, so to add up all the forces on this star cluster will require $N(N-1) = N^2 - N$ floating point operations. Written in pseudocode, this translates to a nested loop:

```
do i=1,N
  F[i]=0
  do j=1,N
    if(i not equal to j) F[i] = F[i] + F(i,j)
  end do
end do
```

When you see nested loops, be careful: the number of nested loops adds directly to the *exponent* of the number of operations. And the largest exponent is what typically goes into the Big-$\mathcal{O}$ notation. Basically, Big-$\mathcal{O}$ notation is used to denote the *computational complexity* of a given algorithm, and we say that the algorithm just described is an $\mathcal{O}(N^2)$ operation.

Suppose we have chosen an algorithm that is $\mathcal{O}(N^3)$ due to a nested loop. If we need to run the $\mathcal{O}(N^3)$ loop in this algorithm three times as well as a $\mathcal{O}(N^2)$ loop once, many numerical analysis texts will write the computational complexity as $\mathcal{O}(3N^3 + N^2)$. Unless we are comparing the complexity of two algorithms with the same exponent on N (e.g., to find the most efficient algorithm) or are interested in the small-N limit, the constant coefficient on N^3 and the N^2 term will rarely be useful.

Thus we typically choose to write an algorithm that has computational complexity $\mathcal{O}(3N^3 + N^2)$ as $\mathcal{O}(N^3)$ because Big-$\mathcal{O}$ notation is typically used to estimate *how long* it will take a given algorithm to run on our computer if N (assumed $\gg 1$) is varied.

In particular, the time T required for our computer to evaluate an $\mathcal{O}(N^3)$ algorithm in the large-N limit will be given by

$$T \propto N^3,$$

which can be written

$$T = aN^3,$$

where a is a proportionality constant. a will not only depend on the fact that there are three loops, but also the speed of our computer! Different computers have different intrinsic speeds, so assessing the computational complexity of our algorithm will provide us an estimate of computational cost, modulo a proportionality constant that accounts for the speed of our computer.

Question:

If an $\mathcal{O}(N^2)$ algorithm requires 10 minutes to complete, how long will it take the algorithm to complete if N is doubled? Assume N is large.

Answer:

The cost in time of an $\mathcal{O}(N^2)$ algorithm will grow quadratically (as N^2) with N. We are given than $a(N_0)^2 = 10$ minutes for some value $N = N_0$, so the proportionality constant is 10 minutes / $(N_0)^2$. Therefore if we choose $N_1 = 2N_0$, then the cost in time is $a(N_1)^2 = a(2N_0)^2 = 4 \times a(N_0)^2 = 4 \times 10$ minutes $= 40$ minutes.

Definition: An algorithm with computational complexity $\mathcal{O}(1)$ will complete in a time *independent* of some size factor N. For example, the algorithm

```
x = N * 2 + 4
```

has computational complexity $\mathcal{O}(1)$, despite the fact that `x` depends on `N`, since as `N` increases, the computational cost remains the same.

5.4 Example: More Efficient Algorithms May Exist!

Suppose we play the game "higher or lower". In this game between two people – Person A and Person B, Person A picks a secret number (an integer) between 0 and N, and Person B attempts to guess the secret number. After each guess, Person A must say either "higher" or "lower" if the number Person B guessed is less than or greater than the secret number, respectively.

One algorithm Person B might choose would be to start at 0 and guess 1, then 2, then 3, etc., while Person A keeps saying "higher", until Person B reaches the secret number. This algorithm for guessing the correct number would be $\mathcal{O}(N)$, since Person B would need to make on average $N/2$ guesses before reaching the correct number.

If on the other hand, Person B chose $N/2$ as the first guess. For the second guess, if Person A said

- "higher", then Person B would pick the closest number to $3N/4$, or
- "lower", then Person B would pick the closest number to $1N/4$.

By continuing this pattern of bisecting the remaining interval on each guess, Person B would rule out fully *half* of the remaining possible secret numbers at each guess. You will find that (*exercise to student*) if $N = 4$, Person B would need at most 3 guesses. If $N = 8$, Person B would need at most 4 guesses. If $N = 64$, Person B would need at most 7 guesses. Continuing the pattern, Person B would need at most $1 + \log_2 N$ guesses for any N. Thus this "bisection" algorithm for finding the secret number has computational complexity of $\mathcal{O}(\log_2 N)$.

Now consider a very large N... say $N = 2^{100}$. Then the cost of the first algorithm would be of order $2^{100} \approx 1.3 \times 10^{30}$ guesses, while the second algorithm would require only 101 guesses at most... a savings of about *28 orders of magnitude.*

Therefore as a general tip, it is very important to seek out algorithms with lower computational complexity in the large-N limit if they exist — this tip might save you years of waiting in your research!

5.5 Big-$\mathcal{O}$ Notation, and Computational Cost Exercises

Exercises:

First write the computational complexity of each of the following algorithms, and then evaluate the total computational cost in FLOPs if $N = 10$. Use Big-$\mathcal{O}$ notation for computational complexity and assume variable assignments, multiplications, additions, subtractions, and divisions all require a single floating point operation.

1. `x = 2*N*N*N - 4*N*N + 5*N + 3`
2.
```
do i=1,N
  F[i]=0
end do
do i=1,N
  do j=1,N
    F[i] = F[i] + j
  end do
end do
```
3.
```
do i=1,N*N
  F[i]=0
  do j=1,N*N
    F[i] = F[i] + i*0.3 + j*0.1
  end do
end do
```
4.
```
do i=1,N*N*N
  F[i]=i+2
end do
do i=1,N
  do j=1,N*N
    F[i] = F[i] + j
  end do
end do
```

Solutions:

1. $\mathcal{O}(1)$. There are no loops, and only a single expression to evaluate. The expression itself, $x = 2*N*N*N - 4*N*N + 5*N + 3$, requires 10 FLOPs to compute: 6 multiplications, one subtraction, two additions, and one assignment.
2. The second loop is an $\mathcal{O}(N^2)$ operation. FLOPs for $N = 10$: each innermost loop requires 2 FLOPs per iteration, so we get 2×10^2 FLOPs for the innermost loop. Add this to the first loop of 10 FLOPs, and we get 210 FLOPs.
3. $\mathcal{O}(N^4)$, because there are two nested N^2 loops. FLOPs for $N = 10$: each innermost loop iteration requires 5 FLOPs, so a total of $5N^4 = 5 \times 10^4$ FLOPs are needed for the innermost loop. Add this to the N^2 assignments needed to initialize $F[i] = 0$, and we get $50,100$ FLOPs.
4. $\mathcal{O}(N^3)$; both the `do i=1,N*N*N` and `do i=1,N ; do j=1,N*N` loops have this computational complexity. FLOPs for $N = 10$: All loops require 2 FLOPs per iteration, and there are 10^3 evaluations per loop. Therefore we find the algorithm requires $4,000$ FLOPs.

MATH 521, Fall 2018 Notes

Prof. Zachariah B. Etienne

Chapter 6: Solving Square Matrix Equations on the Computer

In numerical analysis, we are often required to solve the matrix equation

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b},$$

where $\boldsymbol{A}$ is a square matrix, $\boldsymbol{b}$ is a known vector, and $\boldsymbol{x}$ is the unknown vector representing our solution. How do we solve for $\boldsymbol{x}$?

6.1 Gaussian Elimination

Gaussian Elimination is typically the strategy we are first taught in linear algebra to solve sets of linear equations. Consider the following set of equations:

$$\begin{aligned} 5x_1 - 2x_2 + x_3 &= -2 \\ 4x_1 - 4x_2 + 3x_3 &= 1 \\ x_1 + x_2 - x_3 &= -1 \end{aligned}$$

We first rewrite these in the form $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ as follows:

$$\boldsymbol{A}\boldsymbol{x} = \begin{bmatrix} 5 & -2 & 1 \\ 4 & -4 & 3 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2 \\ 1 \\ -1 \end{bmatrix} = \boldsymbol{b}$$

When performing Gaussian elimination we first write the **augmented matrix** corresponding to this linear system of equations:

$$\left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ 4 & -4 & 3 & 1 \\ 1 & 1 & -1 & -1 \end{array}\right].$$

Recall the goal now is to transform this matrix into echelon or **upper-triangle** form, where all elements below the diagonal are zero. The trick is to notice that the linear set of equations remain unchanged if

1. any row is multiplied by any nonzero, non-infinite constant,
2. rows are interchanged, or
3. any two rows can be added together, with the result replacing one of the rows.

By careful application of these rules, we can "zero-out" all elements below the diagonal, accomplishing our goal of rewriting the matrix in upper triangle form.

We know how to do this process by hand, but how can we teach a computer to do it? This is a question at the core of numerical analysis. Typically when solving the linear system of equations by hand, we try to avoid fractions in favor of integers.

For example, suppose we wish to zero-out the first column of the second row. We could start by multiplying the top row by -4 and then adding it to 5 times the second row, then replacing row 2 with the result.

In shorthand, $-4R_1 + 5R_2 \to R_2$:

$$\left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ -20+20 & 8-20 & -4+15 & 8+5 \\ 1 & 1 & -1 & -1 \end{array}\right] = \left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ 0 & -12 & 11 & 13 \\ 1 & 1 & -1 & -1 \end{array}\right]$$

Our computer is capable of performing floating-point arithmetic to many significant digits, and teaching our computer to perform only integer arithmetic in these cases would greatly complicate our algorithm for Gaussian elimination. In addition, the fact that we are dealing with an *integer* matrix is very artificial; normally the matrix elements will be populated by finite precision, floating point numbers.

So to zero-out the first column of the second row in a more automatic way, let's simply multiply the first column by $-\frac{4}{5}$, add the result to the second row, and then replace the second row.

In shorthand notation we have, $-\frac{4}{5}R_1 + R_2 \to R_2$:

$$\left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ 5\times -\frac{4}{5}+4 & -2\times -\frac{4}{5}-4 & 1\times -\frac{4}{5}+3 & -2\times -\frac{4}{5}+1 \\ 1 & 1 & -1 & -1 \end{array}\right] = \left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ 0 & -\frac{12}{5} & \frac{11}{5} & \frac{13}{5} \\ 1 & 1 & -1 & -1 \end{array}\right]$$

Next we zero out the lower-left component of the matrix with the same approach. To accomplish this, multiply the first row by $-\frac{1}{5}$, add it to the third row, and replace the third row. In shorthand notation: $-\frac{1}{5}R_1 + R_3 \to R_3$.

Analyzing our approach for zeroing-out the element in row i of the first column, we see the following pattern:

$$-c_i R_1 + R_i \to R_i, \quad \text{where} \quad c_i = \frac{\text{row } i\text{, column 1}}{\text{row 1, column 1}}. \tag{6.1}$$

When constructing an algorithm for the computer, it is critically important to standardize and simplify notation whenever possible; ambiguous or overcomplicated notation can make our jobs extremely difficult. So let's standardize and simplify the notation:

Notation:

For a matrix $\boldsymbol{A}$, $a_{i,j}$ will henceforth refer to the element at the ith row and the jth column.

Any 3×4 augmented matrix can be written in this notation as:

$$\boldsymbol{A} = \left[\begin{array}{ccc|c} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \end{array}\right]$$

With this notation, we can more compactly write the steps necessary for zeroing-out all elements in the first column ($a_{i,1}$) below the first row ($i > 1$): Multiply the entire row 1 (R_1) by $-a_{i,1}/a_{1,1}$, add the result to row i (R_i), and replace row i (R_i) with the result. In short-hand notation:

$$-\frac{a_{i,1}}{a_{1,1}} R_1 + R_i \to R_i. \tag{6.2}$$

Note that each row in the augmented matrix has a total of $N+1$ columns. This means that zeroing out each element in the matrix possesses $\mathcal{O}(N)$ complexity. Defining the 2-dimensional array `a[i][j]`$=a_{i,j}$, we might implement Eq. (6.2) to zero out the element `a[i][1]` for some row `i` is as follows (**Warning: There is a serious bug in this algorithm. See Exercise below**):

```
do j=1,N+1
   a[i][j] = -a[i][1]/a[1][1] * a[1][j]  + a[i][j]
end do
```

Exercise:

Find the serious bug in the above "zeroing-out the element `a[i][1]` for some row `i`" algorithm. *Hint: Add a* `print i,j,a[i][j]` *before the* `end do` *and carefully analyze what the output will be.*

Solution:

After the first iteration of the inner loop, a[i][1] has been set to zero. As a result, for j>1, a[i][1]/a[1][1]=0, meaning that the rescaling information c_i (Eq. 6.1) is lost after the first column of the row. To fix this, we store c_i before launching into the j loop:

```
c_i = a[i][1]/a[1][1]
do j=1,N+1
   a[i][j] = -c_i * a[1][j]  + a[i][j]
end do
```

This might be called a *data-dependency* bug, in that the buggy version modifies data that terms j>1 assume are unchanged.

Based on the Exercise above, the bug-fixed algorithm for zeroing out the first column in the ith row is given by

```
c_i = a[i][1]/a[1][1]
do j=1,N+1
   a[i][j] = -c_i * a[1][j]  + a[i][j]
end do
```

To fill in *all* rows $i > 1$ in this way requires that we repeat the above bug-fixed loop for all other rows:

```
do i=2,N
   c_i = a[i][1]/a[1][1]
   do j=1,N+1
      a[i][j] = -c_i * a[1][j]  + a[i][j]
   end do
end do
```

This algorithm will zero-out all elements in the first column below the first row of the matrix. However, we need to zero-out elements below the diagonal in all columns to get the matrix in echelon (upper-triangle) form!

Once we have zeroed-out all elements below the diagonal in the first column, we can then focus on the second column:

```
do i=3,N
   c_i = a[i][2]/a[2][2]
   do j=2,N+1
      a[i][j] = -c_i * a[2][j]  + a[i][j]
   end do
end do
```

Noticing the pattern, we can write the entire algorithm for decomposing the matrix into an upper-triangle (echelon form) matrix using Gaussian elimination as follows:

```
do k=1,N-1
   do i=k+1,N
      c_i = a[i][k]/a[k][k]
      do j=k,N+1
         a[i][j] = -c_i * a[k][j]  + a[i][j]
      end do
   end do
end do
```

This algorithm will output the following echelon or upper-triangle form of $\boldsymbol{A}|\boldsymbol{b}$, to double precision:

$$\left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ 0 & -12/5 & 11/5 & 13/5 \\ 0 & 0 & 1/12 & 11/12 \end{array}\right] \tag{6.3}$$

With the matrix in echelon form, x_1, x_2, and x_3 can be solved by **back substitution**. Let's work out the algorithm. The last row implies that

$$\frac{1}{12}x_3 = \frac{11}{12} \implies x_3 = 11.$$

In general the last row can be written in the form

$$\begin{aligned} & a_{N,N}x_N = a_{N,N+1} \\ \implies \quad & x_N = a_{N,N+1}/a_{N,N}. \end{aligned}$$

The second row then gives us

$$-\frac{12}{5}x_2 + \frac{11}{5}x_3(=11) = \frac{13}{5} \implies -12x_2 = 13 - 121 = -108 \implies x_2 = \frac{108}{12} = \frac{54}{6} = 9.$$

The general form of the second-to-last row is

$$\begin{aligned} & a_{N-1,N-1}x_{N-1} + a_{N-1,N}x_N = a_{N-1,N+1} \\ \implies \quad & x_{N-1} = \frac{a_{N-1,N+1} - a_{N-1,N}x_N}{a_{N-1,N-1}}, \end{aligned}$$

where all the terms on the right-hand side are known.

Then we apply back substitution on the first row:

$$5x_1 - 2x_2(=9) + x_3(=11) = -2 \implies 5x_1 = 2*9 - 11 - 2 = 18 - 13 = 5 \implies x_1 = 1.$$

The general form of the third-to-last row is

$$\begin{aligned} & a_{N-2,N-2}x_{N-2} + a_{N-2,N-1}x_{N-1} + a_{N-2,N}x_N = a_{N-2,N+1} \\ \implies \quad & x_{N-2} = \frac{a_{N-2,N+1} - a_{N-2,N-1}x_{N-1} - a_{N-2,N}x_N}{a_{N-2,N-2}}. \end{aligned}$$

Looking at the general forms of these expressions, we see that in the process of back-substitution, x_i will depend on $x_N, x_{N-1}, ..., x_{i-1}$, as well as the matrix elements $a_{i,i}, a_{i,i+1}, ..., a_{i,N}, a_{i,N+1}$.

Upon careful examination, we find that the expression for an arbitrary x_i in the process of back substitution will be given by

$$x_i = \frac{a_{i,N+1} - \sum_{j=i+1}^{N} a_{i,j}x_j}{a_{i,i}},$$

where all x_js on the right-hand side of this equation have already been computed. Note that for $i = N$, the sum becomes a sum from $N+1$ to N—no elements of the sum are computed, and the expression is consistent with the x_N expression above.

In pseudo-code, the back substitution algorithm takes the form

```
do i=N,1,-1 # Loop from N to 1 inclusive in increments of -1 (opposite normal loop order)
   x[i] = a[i][N+1]/a[i][i]
   do j=i+1,N
      x[i] = x[i] - a[i][j]*x[j]/a[i][i]
   end do
end do
```

Here is a summary of the Gaussian Elimination algorithm we have devised:

Summary: Simple Gaussian Elimination Algorithm

Warning: This algorithm is not very robust. See next section for details!

```
do k=1,N-1
   do i=k+1,N
      c_i = a[i][k]/a[k][k]
      do j=k,N+1
         a[i][j] = -c_i * a[k][j]  + a[i][j]
      end do
   end do
end do
do i=N,1,-1 # Loop from N to 1 inclusive in increments of -1 (opposite normal loop order)
   x[i] = a[i][N+1]/a[i][i]
   do j=i+1,N
      x[i] = x[i] - a[i][j]*x[j]/a[i][i]
   end do
end do
```

Exercises:

1. What is the computational complexity of the Gaussian elimination algorithm (including back substitution)?
2. What is the computational complexity of the back substitution algorithm?
3. Write the pseudocode for computing $\boldsymbol{AB}$, where $\boldsymbol{A}$ and $\boldsymbol{B}$ are $N \times N$ matrices. Store the result in matrix $\boldsymbol{C} = \boldsymbol{AB}$. What is the computational complexity? If our computer can multiply two 100×100 matrices in 1 second, how long can we expect it will need to multiply two $1,000 \times 1,000$ matrices? (Ignore cache effects.)

Solutions:

1. $\mathcal{O}(N^3)$
2. $\mathcal{O}(N^2)$—far less expensive than generating the upper triangle matrix!
3. The computational complexity of multiplication of two $N \times N$ matrices is $\mathcal{O}(N^3)$. If our computer can multiply two 100^2 matrices in 1 second, it will therefore need $(1,000/100)^3$ seconds to multiply two $1,000^2$ matrices, or 1,000 seconds.

```
do i=1,N
   do j=1,N
      sum = 0
      do k=1,N
         sum = sum + a[i][k]*b[k][j]
      end do
      c[i][j] = sum
   end do
end do
```

6.1.1 Toward a More Robust Gaussian Elimination Algorithm

In this section, we will explore cases in which the Gaussian elimination algorithm we have devised will fail, and work to make our algorithm more robust.

6.1.1.1 Zeroes Appearing along the Diagonal of $\boldsymbol{A}$ during Gaussian Elimination

Take a look at the line of code from our Gaussian elimination algorithm:

```
c_i = a[i][k]/a[k][k]
```

Notice that the algorithm will fail if `a[k][k]=0`. That is to say, if a zero appears along a diagonal of our matrix $\boldsymbol{A}$, then the algorithm will divide by zero, yielding `NaN`s (Not A Number) for our solution vector $\boldsymbol{x}$.

Consider the following set of linear equations:

$$\boldsymbol{A}|\boldsymbol{b} = \left[\begin{array}{ccc|c} 0 & -12 & 11 & 13 \\ 5 & -2 & 1 & -2 \\ 0 & 0 & 1 & 11 \end{array}\right]$$

Notice that this is equivalent to

$$\begin{aligned} -12x_2 + 11x_3 &= 13 \\ 5x_1 - 2x_2 + x_3 &= -2 \\ x_3 &= 11, \end{aligned}$$

which is precisely the same system of equations we found in the previous section (Eq. 6.3), just before back-substitution – all we did here was interchange the top two equations. We know the solution is $x_1 = 1$, $x_2 = 9$, and $x_3 = 11$, but if we were to plug this system of equations into our Gaussian elimination algorithm, it would fail because the upper-left diagonal component is zero, resulting in a division by zero in `c_i` when `k=1` and `i=1`.

If we flip the top two rows again, we get:

$$\left[\begin{array}{ccc|c} 5 & -2 & 1 & -2 \\ 0 & -12 & 11 & 13 \\ 0 & 0 & 1 & 11 \end{array}\right],$$

which does not contain a zero along the diagonal, and our standard Gaussian elimination algorithm will effectively do nothing until back-substitution, since the matrix is already in upper-triangle (i.e., echelon) form.

Definition: Flipping rows in this way is called **zero-avoidance pivoting**.

Let us now modify our Gaussian elimination algorithm to incorporate this simple zero-avoidance pivoting to avoid failures:

```
do k=1,N-1
   if(a[k][k] is 0)
      (flip row k with the next row down that does not have a zero in column k)
   end if
   do i=k+1,N
      c_i = a[i][k]/a[k][k]
      do j=k,N+1
         a[i][j] = -c_i * a[k][j]  + a[i][j]
      end do
   end do
end do
```

6.1.1.2 Addressing Roundoff Error Issues with Partial Pivoting

Even after installing the above zero-avoiding pivot algorithm, our Gaussian elimination algorithm can still fail. Consider the following system of equations:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

The exact solution to this equation is

$$\begin{aligned} x_1 &= -\frac{1}{1-10^{-20}} \\ x_2 &= \frac{1}{1-10^{-20}}. \end{aligned}$$

So in double precision, we might expect to get the answers ± 1, which are consistent with the exact results to about 19 significant digits. But this is not the case when using our Gaussian elimination algorithm!

Our Gaussian elimination algorithm will multiply the first row by -10^{20}, add the result to the bottom row, and replace the bottom row with this result. In shorthand notation: $-10^{20}R_1 + R_2 \rightarrow R_2$:

$$\left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & -10^{20}+1 & -10^{20} \end{array}\right]$$

Notice the loss of significance in the bottom row: $-10^{20} + 1 = -10^{20}$ in double precision. So to double precision, the augmented matrix in echelon (upper-triangle) form is written

$$\left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & -10^{20} & -10^{20} \end{array}\right]$$

Next the algorithm will solve for $\boldsymbol{x}$ using back substitution:

$$\begin{aligned} x_2 &= & a_{23}/a_{22} = (-10^{20})/(-10^{20}) & & = 1 \\ x_1 &= & a_{13}/a_{11} - x_2(a_{12}/a_{11}) = 1/10^{-20} - 1(1/10^{-20}) = 10^{20} - 10^{20} & & = 0 \end{aligned}$$

We conclude that back-substitution in double precision yields $x_2 = 1$, which is consistent with the exact result to about 19 significant digits. However, it also finds $x_1 = 0$, which is not even close to the correct answer! What happened here?

Recall our discussions of machine epsilon. It turns out that any $|x_1| \lesssim 10^4$ will satisfy the equation $10^{-20}x_1 + 1 = 1$ in double precision, since machine epsilon is $\epsilon_m \sim 10^{-16}$ in double precision. Notice that the correct value $x_1 = -1$ is in this inequality, but because the $\boldsymbol{U}$ matrix involves an equation that multiplies x_1 by a number 20 orders of magnitude smaller in magnitude than x_2, we have observed a catastrophic cancellation.

Definitions: To combat this loss of significance, we simply adjust our zero-pivoting algorithm to, instead of avoiding zeroes along the diagonal, to instead *maximize the magnitude of the elements along the diagonal.* This strategy is called **partial** or **row pivoting**. Usually partial pivoting is enough to minimize roundoff errors, but sometimes **complete pivoting** is necessary. Complete pivoting flips both columns and rows to maximize the diagonal elements. Note that unlike row pivoting, column pivoting changes the ordering of the components of the solution vector $\boldsymbol{x}$, which requires that one keep track of the order of the columns, usually by use of a **permutation matrix $\boldsymbol{P}$**. The permutation matrix is the identity matrix $\boldsymbol{I}$, but with the columns flipped consistent to the reordered solution vector $\boldsymbol{x}$. In this way, $\boldsymbol{Px}$ is the reordered solution vector.

The partial pivoting algorithm is as follows:

- As column j of the upper triangle (echelon) form of the matrix is computed, first find the largest magnitude element in this column at or below the main diagonal.
- Interchange that element's row with row j.

in pseudocode, the Gaussian elimination algorithm with partial pivoting is given by:

```
do k=1,N
   row_of_maximum_element = k
   do i=k+1,N
      if(|a[i][k]| > |a[row_of_maximum_element][k]|) then
         row_of_maximum_element = i
      end if
   end do
   (flip k and row_of_maximum_element rows)
   do i=k+1,N
      c_i = a[i][k]/a[k][k]
      do j=1,N+1
         a[i][j] = -c_i * a[k][j]  + a[i][j]
      end do
   end do
end do
```

Note that partial pivoting not only acts to minimize roundoff error, but also replaces zero elements along the diagonal. Thus Gaussian elimination with partial pivoting is *superior* to the original zero-avoiding pivot algorithm described above.

Let's apply partial pivoting strategy to our simple example:

$$\boldsymbol{A}|\boldsymbol{b} = \left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 1 & 1 & 0 \end{array}\right]$$

Partial pivoting first scans the first column for the row with the largest magnitude element in this column. Once it finds this element, it flips the rows. Clearly $|1| > |10^{-20}|$, so we must flip rows:

$$\left[\begin{array}{cc|c} 1 & 1 & 0 \\ 10^{-20} & 1 & 1 \end{array}\right]$$

Gaussian elimination then proceeds using our algorithm via $-10^{-20}R_1 + R_2 \rightarrow R_2$:

$$\left[\begin{array}{cc|c} 1 & 1 & 0 \\ 0 & 1-10^{-20} & 1-10^{-20} \end{array}\right] = \left[\begin{array}{cc|c} 1 & 1 & 0 \\ 0 & 1 & 1 \end{array}\right]$$

Back substitution yields

$$\begin{aligned} x_2 = a_{23}/a_{22} = 1/1 &= 1 \\ x_1 = a_{13}/a_{11} - x_2(a_{12}/a_{11}) = 0/1 - 1(1/1) &= -1, \end{aligned}$$

which is consistent with the exact solution to about 19 significant digits!

6.1.1.3 When Partial Pivoting Fails: Rescale maximum element in each row to unit magnitude

Consider the same example as in the previous section, except with the top row multiplied by 10^{20}:

$$\boldsymbol{A}|\boldsymbol{b} = \left[\begin{array}{cc|c} 1 & 10^{20} & 10^{20} \\ 1 & 1 & 0 \end{array}\right].$$

In this case, our row/partial pivoting strategy will be ineffective, because the elements in the first column under the upper-left diagonal are equal. The result will again be $x_1 = 0$ and $x_2 = 1$. To fix this, we only need to multiply each row by the inverse of the largest magnitude element, prior to partial pivoting. This will have computational complexity of $\mathcal{O}(N^2)$—again a small fraction of the total cost of Gaussian elimination in the large-N limit. An alternative to this approach is to apply a **scaled pivoting** algorithm, whereby rows are flipped based on their largest magnitude. This will have computational complexity of $\mathcal{O}(N^2)$ as well because all elements of the matrix must be analyzed.

6.1.1.4 No Solution

Sometimes we might encounter a case in which there exists zero or infinitely many solutions to $\boldsymbol{Ax} = \boldsymbol{b}$. This will happen if and only if $\det(\boldsymbol{A}) = 0$. In this case, a solution $\boldsymbol{x}$ obtained through Gaussian elimination is guaranteed to fail. To prevent this from happening, one could compute the determinant of $\boldsymbol{A}$, but this is also a $\mathcal{O}(N^3)$ operation using standard approach (cofactor, or "Laplace" expansion)! So if our output contains `NaN`s, then and only then do we check for a zero determinant.

6.1.1.5 Summary: A Robust Gaussian Elimination Algorithm

Caution: There are examples of matrices for which complete pivoting is necessary to guarantee a solution.

Summary: Gaussian Elimination Algorithm with Partial Pivoting and $\det(\boldsymbol{A}) = 0$ Checking

```
do i=1,N
   # First normalize maximum magnitude in each row to 1
   max_magnitude_in_row_i = 0
   do j=1,N+1
      if(|a[i][j]| > max_magnitude_in_row_i)
         max_magnitude_in_row_i = |a[i][j]|
      end if
   end do
   do j=1,N+1
      a[i][j] = a[i][j] / max_magnitude_in_row_i
   end do
end do
do k=1,N
   # Partial pivoting algorithm:
   row_of_maximum_element = k
   do i=k+1,N
      if(|a[i][k]| > |a[row_of_maximum_element][k]|) then
         row_of_maximum_element = i
      end if
   end do
   (flip k and row_of_maximum_element rows)
   # Algorithm for computing upper-triangle form of A|b
   do i=k+1,N
      c_i = a[i][k]/a[k][k]
      do j=1,N+1
         a[i][j] = -c_i * a[k][j]  + a[i][j]
      end do
   end do
end do
# Back-substitution algorithm:
do i=N,1,-1 # Loop from N to 1 inclusive in increments of -1 (opposite normal loop order)
   x[i] = a[i][N+1]/a[i][i]
   do j=i+1,N
      x[i] = x[i] - a[i][j]*x[j]/a[i][i]
   end do
end do
# Check for NaNs in solution vector. If found, print appropriate error message.
do i=1,N
   if(x[i] == NaN) then
      detA = (determinant of matrix A)
      if(detA == 0) then
         print("Error: Matrix A is singular. No solution can be found.")
      else
         print("Error: Matrix A is non-singular, yet no solution found.")
         print(" You may have chosen a matrix that requires complete pivoting")
      end if
   end if
end do
```

6.2 The LU Decomposition

Suppose we have M matrix equations to solve, which follow the pattern

$$\boldsymbol{A}\boldsymbol{x}_i = \boldsymbol{b}_i,$$

where $i \in [1, M]$. We could in principle apply the Gaussian Elimination algorithm we just derived to this equation M times to find all M solutions for $\boldsymbol{x}_i$, but this would be an $\mathcal{O}(MN^3)$ operation, and M could be very large.

Definition: When a matrix $\boldsymbol{A}$ can be written as a product of simpler matrices, we call it a **decomposition** of $\boldsymbol{A}$, so since $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$ we call this the approach the $\boldsymbol{L}\boldsymbol{U}$ decomposition of $\boldsymbol{A}$.

The LU decomposition decomposes $\boldsymbol{A}$ into a lower-triangle matrix $\boldsymbol{L}$ times an upper-triangular matrix $\boldsymbol{U}$, so that $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$. $\boldsymbol{U}$ is constructed via Gaussian elimination on $\boldsymbol{A}$, and $\boldsymbol{L}$ stores the information that encodes the Gaussian elimination, as we will soon see.

Once we have $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$, then

$$\begin{aligned} \boldsymbol{b}_i &= \boldsymbol{A}\boldsymbol{x}_i \\ &= \boldsymbol{L}\boldsymbol{U}\boldsymbol{x}_i \\ &= \boldsymbol{L}(\boldsymbol{U}\boldsymbol{x}_i) \end{aligned}$$

Define

$$\boldsymbol{y}_i = \boldsymbol{U}\boldsymbol{x}_i.$$

Notice that both $\boldsymbol{y}_i$ and $\boldsymbol{x}_i$ are unknowns. The original equation can then be written in terms of $\boldsymbol{y}_i$ as follows:

$$\begin{aligned} \boldsymbol{b}_i &= \boldsymbol{A}\boldsymbol{x}_i \\ &= \boldsymbol{L}\boldsymbol{U}\boldsymbol{x}_i \\ &= \boldsymbol{L}(\boldsymbol{U}\boldsymbol{x}_i) \\ &= \boldsymbol{L}\boldsymbol{y}_i. \end{aligned}$$

So once we have decomposed $\boldsymbol{A}$ into $\boldsymbol{L}$ and $\boldsymbol{U}$ via Gaussian elimination (an $\mathcal{O}(N^3)$ operation), we can then compute $\boldsymbol{y}_i$ via simple back substitution (an $\mathcal{O}(N^2)$ operation), and then $\boldsymbol{x}_i$ via back substitution as well (again, an $\mathcal{O}(N^2)$ operation).

This makes the LU decomposition quite useful for many engineering/scientific applications in which $\boldsymbol{A}$ stays fixed, while we must solve for many different $\boldsymbol{b}_i$. So if we have M different $\boldsymbol{b}_i$'s, then the LU decomposition can be leveraged to lower the total cost of this operation to a computatinoal complexity $\mathcal{O}(N^3) + \mathcal{O}(MN^2)$. Compare this to the complexity for Gaussian elimination: $\mathcal{O}(MN^3)$—an enormous cost savings when M is large!

6.2.1 Worked Example of LU Decomposition

Now I will go over the strategy for generating the $\boldsymbol{L}$ and $\boldsymbol{U}$ matrices, starting with the following 3×3 matrix:

$$\boldsymbol{A} = \begin{bmatrix} 1 & -2 & 4 \\ 2 & -6 & 10 \\ 1 & 2 & -1 \end{bmatrix}$$

The idea is, $\boldsymbol{L}$ and $\boldsymbol{U}$ encode all the information necessary to perform a Gaussian elimination solution of $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. This will be most obvious as we build the $\boldsymbol{L}$ matrix. We start by writing $\boldsymbol{L}$ as follows:

$$\boldsymbol{L} = \begin{bmatrix} 1 & 0 & 0 \\ - & 1 & 0 \\ - & - & 1 \end{bmatrix},$$

where the dashes indicate empty matrix elements that will be filled in with the Gaussian elimination information necessary to construct the upper-triangle matrix $\boldsymbol{U}$.

We now proceed with our Gaussian elimination on $\boldsymbol{A}$: $-2R_1 + R_2 \to R_2$

$$\boldsymbol{A_1} = \begin{bmatrix} 1 & -2 & 4 \\ 0 & -2 & 2 \\ 1 & 2 & -1 \end{bmatrix}$$

The idea is, we record this Gaussian elimination step by inserting the opposite of the multiplication factor we used to get rid of $a_{2,1}$, to the $l_{2,1}$ element of our matrix $\boldsymbol{L}$. The multiplication factor was -2, so we insert a $+2$ at element $l_{2,1}$ in the $\boldsymbol{L}$ matrix:

$$\boldsymbol{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ - & - & 1 \end{bmatrix},$$

We continue by zeroing the first column in the next row via $-R_1 + R_3 \to R_3$, and record the result in the lower-left corner of the $\boldsymbol{L}$ matrix:

$$\boldsymbol{A_2} = \begin{bmatrix} 1 & -2 & 4 \\ 0 & -2 & 2 \\ 0 & 4 & -5 \end{bmatrix}, \qquad \boldsymbol{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & - & 1 \end{bmatrix},$$

Next, to eliminate the 4 in the bottom row, we apply $2R_2 + R_3 \to R_3$. This completes not only our $\boldsymbol{U}$ matrix, but also our $\boldsymbol{L}$ matrix as well:

$$\boldsymbol{A_3} = \boldsymbol{U} = \begin{bmatrix} 1 & -2 & 4 \\ 0 & -2 & 2 \\ 0 & 0 & -1 \end{bmatrix}, \qquad \boldsymbol{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -2 & 1 \end{bmatrix}$$

We can confirm that $\boldsymbol{A} = \boldsymbol{LU}$:

$$\boldsymbol{LU} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} 1 & -2 & 4 \\ 0 & -2 & 2 \\ 0 & 0 & -1 \end{bmatrix} = \boldsymbol{A} = \begin{bmatrix} 1 & -2 & 4 \\ 2 & -6 & 10 \\ 1 & 2 & -1 \end{bmatrix}$$

Let us now apply the LU decomposition technique to solve for $\boldsymbol{x}$. Recall that our first step is to define $\boldsymbol{y} = \boldsymbol{Ux}$, so that

$$\boldsymbol{Ax} = \boldsymbol{LUx} = \boldsymbol{L}(\boldsymbol{Ux}) = \boldsymbol{Ly} = \boldsymbol{b}.$$

Then we use back-substitution to quickly solve for $\boldsymbol{y}$. Let us suppose we have $\boldsymbol{b}^T = (1, 2, 4)$. Then

$$\boldsymbol{Ly} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix}.$$

Then clearly, $y_1 = 1$, $2y_1 + y_2 = 2 \implies y_2 = 0$, and $y_1 - 2y_2 + y_3 = 1 + y_3 = 4 \implies y_3 = 3$.

Next we use the relation

$$\boldsymbol{Ux} = y = \begin{bmatrix} 1 & -2 & 4 \\ 0 & -2 & 2 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 3 \end{bmatrix}.$$

Thus we immediately get $x_3 = -3$, $-2x_2 + 2x_3 = -2x_2 - 6 = 0 \implies x_2 = -3$, and $x_1 - 2x_2 + 4x_3 = x_1 + 6 - 12 = 1 \implies x_1 = 7$.

Again, the power of the LU decomposition is that if we needed the solution $\boldsymbol{x}$ from $\boldsymbol{Ax} = \boldsymbol{b}$ now for a *different* $\boldsymbol{b}$, we could apply the LU decomposition of $\boldsymbol{A}$ we have already computed, along with some trivial back-substitution to get the new $\boldsymbol{x}$.

MATH 521, Fall 2018 Notes

Prof. Zachariah B. Etienne

Chapter 7: Function Approximation (Approximation Theory)

At their lowest levels, computers are generally capable of only multiplication, division, addition, and subtraction. If $f(x)$ is a low-order polynomial, this becomes an easy task, since polynomial functions may be evaluated at a given point using these operations alone. However, how are we to evaluate transcendental functions, like $\cos(x)$? Addressing this question is at the heart of Approximation Theory.

7.1 Taylor Series

The Taylor series should be our first approach for function approximation if the function is smooth and differentiable, and we only need to compute $f(x)$ over a limited range of x near a point x_0.

Definition: Defining $f^{(n)}(x)$ to be the nth derivative of $f(x)$, the **Taylor series expansion** of a function $f(x)$ about a point $x = x_0$ is given by:

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

Definition: Taylor series expansions are a type of **power series** representation of a function $f(x)$, where a power series satisfies the pattern:

$$f(x) = \sum_{n=0}^{\infty} a_n (x-x_0)^n.$$

Of course in the case of a Taylor series, the power series coefficient a_n satisfies $a_n = \frac{f^{(n)}(x_0)}{n!}$.

As we will find in Example #1 below, Taylor series can be used for evaluating a function at discrete points where it is smooth (i.e., multiple-times differentiable).

Example #1:

Evaluate $f(x) = \cos(x)$ at $x = 0.01$ via a Taylor series expansion about a neighboring point x_0 where $f(x)$ and its derivatives can be computed easily by hand.

Taylor series require that we evaluate derivatives of $f(x)$ at a point x_0. Let's take a cue from the problem statement and evaluate $\cos(x)$ at a nearby point that is easily evaluated by hand, say $x_0 = 0$. There are an infinite number of derivatives in the sum, so instead of spending our entire finite lives writing them out, let's instead find a pattern that these derivatives follow, so we can devise an algorithm for evaluating the Taylor series of $\cos(x)$ at $x = 0.01$, about $x_0 = 0$.

Note first that the zeroth derivative is defined as the function itself (i.e., taking the derivative of the function zero times). Thus

$$\begin{aligned}
f^{(0)}(0) &= \cos(0) = 1 \\
f^{(1)}(0) &= f'(0) = -\sin(0) = 0 \\
f^{(2)}(0) &= -\cos(0) = -1 \\
f^{(3)}(0) &= \sin(0) = 0 \\
f^{(4)}(0) &= \cos(0) = 1 \\
\vdots \quad & \quad \vdots \\
f^{(n)}(0) &= \begin{cases} 0 & \text{if } n \text{ odd} \\ (-1)^{n/2} & \text{if } n \text{ even.} \end{cases}
\end{aligned}$$

Based on this, we can immediately write an expression for computing the Taylor expansion of $f(x) = \cos(x)$ about $x_0 = 0$:

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

Let's now evaluate $\cos(x = 0.01)$ using this expansion:

$$\cos(0.01) = 1 - \frac{10^{-4}}{2!} + \frac{10^{-8}}{4!} - \frac{10^{-12}}{6!} - \dots$$

Notice that this evaluation of the transcendental function $\cos(x)$ at $x = 0.01$ requires only addition, multiplication, division, and subtraction, meaning that we have discovered an algorithm to evaluate $\cos(0.01)$ on the computer!

Exercise #1

How many terms do we need to include in this Taylor series expansion to guarantee double precision accuracy of $\cos(0.01)$?

Solution

The Taylor series expansion yielded

$$\cos(0.01) = 1 - \frac{10^{-4}}{2!} + \frac{10^{-8}}{4!} - \frac{10^{-12}}{6!} - \dots$$

The first term is 1, the second term adjusts the 4th significant digit of 1, the third term adjusts the 9th significant digit (since $\frac{10^{-8}}{4!} \sim 10^{-9}$), and the fourth term adjusts the 14th significant digit (since $\frac{10^{-12}}{6!} \sim 10^{-14}$). The fifth term, $10^{-16}/8!$ would adjust a number far below machine epsilon in double precision. Thus four terms in the expansion are necessary to guarantee double precision accuracy of the Taylor series expansion.

We have found that transcendental functions can be evaluated on the computer to double precision using a Taylor series that requires the sum of only a few terms. Let's take a look at another transcendental function:

Example #2:

Evaluate $\ln(x)$ at $x = 10$ via Taylor expansion about $x_0 = e$.

As before, we start by computing the derivatives of $\ln(x)$ evaluated at $x_0 = e$:

$$\begin{aligned}
f^{(0)}(e) &= \ln(e) = 1\\
f^{(1)}(e) &= 1/e\\
f^{(2)}(e) &= -1/e^2\\
f^{(3)}(e) &= 2/e^3\\
f^{(4)}(e) &= -3!/e^4\\
f^{(5)}(e) &= 4!/e^5\\
f^{(6)}(e) &= -5!/e^6\\
f^{(7)}(e) &= 6!/e^7\\
\vdots\quad &\quad \vdots\\
f^{(n)}(e) &= \begin{cases} 1 & \text{if } n = 0\\ (-1)^{n-1}(n-1)!/e^n & \text{if } n > 0. \end{cases}
\end{aligned}$$

Thus the general expression for the Taylor series of $f(x) = \ln(x)$ about $x_0 = e$ may be written:

$$\begin{aligned}
f(x) = \ln(x) &= 1 + \sum_{n=1}^{\infty} \frac{(x-e)^n(-1)^{n-1}(n-1)!}{n!e^n}\\
&= 1 + \sum_{n=1}^{\infty} \frac{(x-e)^n(-1)^n(-1)^{-1}}{ne^n}\\
&= 1 - \sum_{n=1}^{\infty} \frac{(e-x)^n}{ne^n}\\
f(10) = \ln(10) &= 1 - \sum_{n=1}^{\infty} \frac{(e-10)^n}{ne^n}\\
&= 1 + 2.68 - 3.59 + 6.41\\
&\quad -12.9 + 27.6 - 61.6 + 141 \dots
\end{aligned}$$

With the above pattern we can program a computer to evaluate the Taylor series sum to a finite number of terms N as N is increased. Define the Taylor series sum to a finite number of terms N to be $f_{\text{TS}}(x, N)$. Then our computer program computing the terms in the Taylor series will yield the following results:

$$f_{\mathrm{TS}}(x=10, N=10) \equiv 1 - \sum_{n=1}^{10} \frac{(e-10)^n}{ne^n} \quad \approx \quad \texttt{-1.344e3}$$
$$f_{\mathrm{TS}}(x=10, N=20) \equiv 1 - \sum_{n=1}^{20} \frac{(e-10)^n}{ne^n} \quad \approx \quad \texttt{-1.3e7}$$
$$f_{\mathrm{TS}}(x=10, N=30) \equiv 1 - \sum_{n=1}^{30} \frac{(e-10)^n}{ne^n} \quad \approx \quad \texttt{-1.7e11}.$$

It does not look like the series is converging to the exact value of $\ln(10) \approx 2.302$ at all. We will soon prove that this is a *divergent* series.

7.2 Series Convergence

Recall that a Taylor series is just one particular example of a power series. Now consider the *finite* series

$$f_{\mathrm{S}}(x, N) = \sum_{n=0}^{N} b_n(x)$$

Definition: We say a series **converges** at a point x if the limit $\lim_{N\to\infty} f_{\mathrm{S}}(x, N)$ exists and is finite for that x. Otherwise, we say the series **diverges**.

Definition: In general the Taylor series

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

may only be used to approximate a function $f(x)$ within a limited range of $|x - x_0| < \rho$, where ρ is called the **radius of convergence**.

Definition: We can sometimes compute the radius of convergence using the **Ratio Test**. For the power series

$$f(x) = \sum_{n=0}^{\infty} a_n (x-x_0)^n,$$

define

$$b_n(x) = a_n (x-x_0)^n.$$

Choosing $b_n \neq 0$, $b_{n+1} \neq 0$, and fixing x, compute the limit:

$$\lim_{n\to\infty} \left| \frac{b_{n+1}}{b_n} \right| = L.$$

- If $L < 1 \implies$ series converges absolutely.
- If $L > 1 \implies$ series diverges.
- If limit does not exist or $L = 1 \implies$ test inconclusive.

Example #3:

Apply the Ratio Test to determine whether the Taylor expansion of $\ln(x)$ about $x_0 = e$ is in fact a divergent series at $x = 10$.

Recall that we computed the general Taylor series of $\ln(x)$ about $x = e$ to be

$$\ln(x) = 1 - \sum_{n=1}^{\infty} \frac{(e-x)^n}{ne^n}$$

Let's subtract 1 from both sides of the equation first. This will have no effect on the divergence or convergence of the sum.

Then the Ratio Test gives:

$$\lim_{n\to\infty}\left|\frac{(e-x)^{n+1}ne^n}{(e-x)^n(n+1)e^{n+1}}\right| = |e-x|\lim_{n\to\infty}\frac{n}{(n+1)e} = |e-x|/e = L. \implies |e-x| = |x-e| = eL.$$

$L < 1$ for absolute convergence, so $|x-e|/e < 1$ for convergence and the radius of convergence is e. Thus we can only expect the Taylor series to converge if $|e-x| < e$. I.e., convergence will be guaranteed if $0 < x < 2e \approx 5.44$. We were trying to compute $\ln(x)$ at $x = 10$, which is far outside this inequality. Thus according to the Ratio Test, the Taylor series of $\ln(x)$ about $x_0 = e$ will be divergent at $x = 10$, consistent with what we found as we increased the number of terms in the sum.

Therefore, we must be careful when evaluating functions according to their Taylor series, as Taylor series in general will have a limited radius of convergence.

Example #4:

Compute the radius of convergence for the Taylor series expansion of $f(x) = \cos(x)$ about $x_0 = 0$ (see **Example #1**).

Recall that this Taylor series *appeared* to converge for $x = 0.01$, but can we prove that it is convergence with the Ratio Test?

Recall that $\cos(x)$ about $x_0 = 0$ is

$$\begin{aligned}\cos(x) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots \\ &= \sum_{n\text{ even}} (-1)^{n/2}\frac{x^n}{n!} \\ &= \sum_{n=0}^{\infty} b_n,\ \text{where } b_n = \begin{cases} 0 & \text{if } n \text{ is odd, and} \\ (-1)^{n/2}\frac{x^n}{n!} & \text{if } n \text{ is even.}\end{cases}\end{aligned}$$

The final expression is inconsistent with the Ratio Test's requirement that b_n and b_{n+1} be nonzero. So to be able to use the Ratio Test, we rewrite the series into the following form instead:

$$\cos(x) = \sum_{n=0}^{\infty}(-1)^n\frac{x^{2n}}{(2n)!}$$

Then the Ratio Test yields

$$\lim_{n\to\infty}\left|\frac{(x)^{2(n+1)}(2n)!}{(x)^{2n}(2(n+1))!}\right| = \lim_{n\to\infty}\left|\frac{(x)^{2(n+1)}(2n)!}{(x)^{2n}(2n+2)(2n+1)(2n)!}\right| = \lim_{n\to\infty} x^2\left|\frac{1}{(2n+2)(2n+1)}\right| = 0 < 1$$

The limit evaluates to a number less than one, so we conclude that the radius of convergence is infinite, so the series is guaranteed to converge for all real x!

7.3 Rate of Convergence

In addition to concerns about the radius of convergence, we must also be careful about series that are extremely slow to converge. You will recall that when we expanded $\cos(x)$ about $x_0 = 0$ to compute $\cos(0.01)$, our series converged quite rapidly:

$$\cos(0.01) = 1 - \frac{10^{-4}}{2!} + \frac{10^{-8}}{4!} - \frac{10^{-12}}{6!} + \frac{10^{-16}}{8!} - \dots$$

I.e., with just four terms, we obtained a result consistent with the exact value to double precision (**Exercise #1**).

Let's take a look at the same expansion to compute $\cos(1)$:

$$\cos(1) = 1 - \frac{1}{2!} + \frac{1}{4!} - \frac{1}{6!} + \frac{1}{8!} - \dots + \frac{(-1)^{n/2}}{n!}(n\text{ even})$$

It turns out that we will need to compute the first *eight* nonzero terms to achieve double precision (i.e., an error less than 1 part in `1e16`). Notice that this is twice the number of terms needed to compute $x = 0.01$.

Finally, let's apply the same Taylor expansion to evaluate cos(1000):

$$\cos(1000) = 1 - \frac{10^6}{2!} + \frac{10^{12}}{4!} - \frac{10^{18}}{6!} + \frac{10^{24}}{8!} - \dots$$

Notice the numerators are absolutely enormous. The Ratio Test guarantees that the series will be convergent, but

1. convergence will require possibly millions of terms,
2. the terms in the series will, within a small number of terms, reach the overflow limits of double precision, and
3. significant digits will be lost extremely rapidly, wiping out any hope of computing the sum in double precision (due to the terms in the sum varying over many orders of magnitude). Remember that the result of this sum must be between -1 and $+1$, yet individual terms in the sum can be hundreds of orders of magnitude larger than this!

Here's the bottom line: If we wish to evaluate a function at a given point x using a Taylor series, it is best to choose a known value x_0 that is

- *close* to the x we wish to compute (at least such that x is within the radius of convergence, but preferably as close to x_0 as possible) and
- a point at which $f^{(n)}(x_0)$ is known or very easily computed,

so that we may use the Taylor series to evaluate a function to double precision quickly and efficiently using only addition, subtraction, multiplication, and division.

7.3.1 Computing π with a Taylor Series

We can also use Taylor series to evaluate irrational numbers. E.g., π can be computed via a very clever trick. Let's look at the Taylor series expansion for $f(x) = 1/(1+x)$ about $x_0 = 0$:

$$\begin{aligned}
f^{(0)}(0) &= & \left.\tfrac{1}{(1+x)}\right|_{x=0} &= 1 &= (-1)^0 0! \\
f^{(1)}(0) &= & -\left.\tfrac{1}{(1+x)^2}\right|_{x=0} &= -1 &= (-1)^1 1! \\
f^{(2)}(0) &= & 2\left.\tfrac{1}{(1+x)^3}\right|_{x=0} &= 2 &= (-1)^2 2! \\
f^{(3)}(0) &= & -3!\left.\tfrac{1}{(1+x)^4}\right|_{x=0} &= -3! &= (-1)^3 3! \\
&\vdots & \vdots & & \vdots \\
f^{(n)}(0) &= & (-1)^n n!\left.\tfrac{1}{(1+x)^n}\right|_{x=0} & &= (-1)^n n!
\end{aligned}$$

The Taylor series can then be written

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

The radius of convergence is 1. Let's focus on the positive values of x. This series will definitely converge at $0 < x < 1$. Define $w^2 = x$. Then

$$f(w^2) = \sum_{n=0}^{\infty} (-1)^n w^{2n} = \frac{1}{1+w^2}$$

Integrate both sides of this equation from $w = 0$ to $w = y$. Then we get

$$\tan^{-1}(y) = y - \frac{y^3}{3} + \frac{y^5}{5} - \frac{y^7}{7} + \dots$$

Assume $\epsilon > 0$, where $\epsilon \ll 1$. The series will converge at $\bar{1} = 1 - \epsilon$, and will converge to $\pi/4$ as $\epsilon \to 0$, since $\lim_{\epsilon \to 0} \tan^{-1}(1-\epsilon) = \pi/4$. Thus we have

$$\pi/4 \approx \tan^{-1}(\bar{1}) \approx 1 - 1/3 + 1/5 - 1/7 + \ldots \implies \pi \approx 4 - 4/3 + 4/5 - 4/7 + \ldots$$

This series, known as the Gregory-Leibniz series, does converge to π, albeit extremely slowly: only 5 significant digits of π will be obtained after computing *500,000* terms in the sum. Perhaps we should not be surprised by this slow convergence; the Ratio Test for this series yields an indeterminate result.

Open Exercise

Assuming that $\sqrt{3}$ is known exactly, construct an algorithm based on the same approach as above for evaluating π using the Taylor series of $\tan^{-1}\left(\sqrt{3}/3\right) = \pi/6$. Your algorithm may only evaluate $\sqrt{3}$ *once*. How many terms in the sum are required to obtain a correct value of π to 8 and 16 significant digits? Is the convergence of this algorithm faster than the Gregory-Leibniz series? Why or why not?

In general, a Taylor series approximation is not the best way of computing irrational numbers on the computer. Often you will find other series or even sequence representations that work better. For example, the following series

$$\pi = 3 + \frac{4}{2 \times 3 \times 4} - \frac{4}{4 \times 5 \times 6} + \frac{4}{6 \times 7 \times 8} - \frac{4}{8 \times 9 \times 10} + \cdots$$

converges much faster, achieving 3 significant digits of pi after only 5 terms. This series was found in the 15th century.

7.4 Generic Bases, Fourier Series

Taylor series are but just one way of approximating functions via a series approximation on a computer. In general, we can approximate a function on the computer via

$$f(x) = \sum_{n=0}^{\infty} a_n \phi_n(x),$$

where $\phi_n(x)$ is called a *basis function*, and a_n is the coefficient. If $f(x)$ is simply a polynomial, then $\phi_n(x) = x^n$ and the sum will truncate at n equal to the polynomial order. We can also use polynomials for arbitrary functions, but as we saw, this is equivalent to a Taylor series representation, and the radius of convergence might be very limited or the convergence very slow in some cases. One reason for this is the fact that for large values of x, the values of x^n or $(x - x_0)^n$ become very large, requiring for convergence that a_n drop very rapidly with increasing n or, as we saw in the previous section, incredible cancellation for large values of x or $(x - x_0)$.

To address this limitation of polynomials and possibly obtain much faster convergence, suppose we adopt sines and cosines as basis functions. This **Fourier basis** is a very desirable one, because unlike the polynomial basis of a power or Taylor series, when x becomes large sines and cosines will always evaluate between -1 and $+1$.

The Fourier basis can reliably approximate an arbitrary, smooth function $f(x)$ on some interval $-L < x < L$ using the **Fourier series**:

$$f(x) = \frac{a_0}{2} + \sum_{m=1}^{\infty} \left[a_m \cos\left(\frac{m\pi x}{L}\right) + b_m \sin\left(\frac{m\pi x}{L}\right) \right]$$

Effectively, a Fourier series representation of a function is simply a sum of wiggles with different wavelengths; the *wavelength* of a given sine or cosine in the Fourier series is $\lambda_m = L/m$, so as we increase m, the wavelength of the underlying basis function shrinks. So smooth functions without sharp features should possess Fourier series that converge quite rapidly.

Next, notice that if we evaluate the Fourier series outside the original interval, at $x \to x + 2NL$, where N is any integer, we get

$$\begin{aligned}
f(x+2NL) &= \frac{a_0}{2} + \sum_{m=1}^{\infty} \left[a_m \cos\left(\frac{m\pi(x+2NL)}{L}\right) + b_m \sin\left(\frac{m\pi(x+2NL)}{L}\right)\right] \\
&= \frac{a_0}{2} + \sum_{m=1}^{\infty} \left[a_m \cos\left(\frac{m\pi x}{L} + 2mN\pi\right) + b_m \sin\left(\frac{m\pi x}{L} + 2mN\pi\right)\right] \\
&= \frac{a_0}{2} + \sum_{m=1}^{\infty} \left[a_m \cos\left(\frac{m\pi x}{L}\right) + b_m \sin\left(\frac{m\pi x}{L}\right)\right] \\
&= f(x).
\end{aligned}$$

That is to say, **the Fourier series of $f(x)$ is a periodic function outside the original interval** $x \in [-L, L]$ **with period** $2L$.

Corollary: If the underlying function $f(x)$ (i.e., the function we wish to represent as a Fourier series) is smooth but **is not periodic on this interval**, the Fourier series will need to represent the discontinuity in the function or its derivatives at those boundary points $x = \pm L$ and will generally be very slow to converge there. This is because sines and cosines are smooth, and approximating a discontinuous function with a smooth wavy one requires very short wavelength waves, requiring very short wavelength wiggles $\lambda_m = L/m$ to reach convergence (i.e., the number of terms in the sum would need to be very large to approach the correct value in the neighborhood of such discontinuities).

Useful Tip: A Fourier series of a function can be computed in an interval away from $x = 0$ by simply shifting the x coordinate. For example, if we wish to evaluate the Fourier series of $f(x)$ on an interval $x \in [0, 4]$, then this would be equivalent to defining $y(x) = x - 2$ and evaluating the Fourier series of $f(y(x)) = f(x-2)$ in the interval $x \in [-2, 2]$.

Fourier series are extremely useful on computers, and since our ears interpret only a limited frequency range of sound waves, we have Fourier series to thank for the very efficient MP3 audio format. Consider a sound wave as a function $f(t)$. MP3s saves space by removing coefficients a_m and b_m in the Fourier series of $f(t)$, so that when the audio is reconstructed, we can still identify the content of the sound. Removing coefficients is equivalent to removing wavelength from the sound, and sometimes these cut-out wavelengths correspond to wavelengths that are out of the range of human hearing, making MP3 audio indistinguishable from the original wave, despite the fact that it may have removed a very large fraction of the data. In the case when we simply wish to restrict the overall data bandwidth when streaming audio, we can truncate wavelengths within the range of human hearing, yet the sound will still be easily understood. This is typically what happens to the audio signal when you speak to someone over the phone.

Source: `http://dsp.stackexchange.com/questions/69/why-is-the-fourier-transform-so-important`:

During a call, when you press a button on your phone, notice how the different numbers produce slightly different sounds? Each of these sounds possesses a different Fourier spectrum that, e.g., an automated phone system can process and lead you to the correct department.

Fourier series are also useful for very accurately approximating functions when solving differential equations, when the solution cannot be solved by pencil and paper using known techniques. This is in part due to the fact that the derivative of the Fourier series of $f(x)$ is very simple to compute:

$$f'(x) = \sum_{m=1}^{\infty} (m\pi/L) \left[-a_m \sin\left(\frac{m\pi x}{L}\right) + b_m \cos\left(\frac{m\pi x}{L}\right)\right].$$

Definition: Even and odd functions.

- We say a function $f(x)$ is *even* if $f(x) = f(-x)$. $f(x) = \cos(x)$ falls into this category.
- We say a function $f(x)$ is *odd* if $f(x) = -f(-x)$. $f(x) = \sin(x)$ falls into this category.

Corollary #1: The integral of an even function over an interval that evenly straddles zero—i.e., $x \in [-L, L]$—will evaluate to twice the integral spanning $x \in [0, L]$:

$$\int_{-L}^{L} f(x)dx = 2\int_{0}^{L} f(x)dx, \quad \text{for } f(x) \text{ even.} \tag{7.1}$$

Corollary #2: The integral of an odd function over an interval that evenly straddles zero—i.e., $x \in [-L, L]$—will evaluate to zero, as the area under the curve for $x > 0$ will exactly cancel the area under the curve at $x < 0$ (odd functions evaluate to zero at $x = 0$):

$$\int_{-L}^{L} f(x)dx = 0, \quad \text{for } f(x) \text{ odd.} \tag{7.2}$$

Identities for even and odd functions:

1. even × even = even
2. even × odd = odd
3. odd × odd = even

Definition: The **Kronecker delta** symbol, $\delta_{m,n}$, is equal to 1 if $m = n$ and 0 otherwise.

To compute the coefficients in a Fourier series a_m and b_m, we make use of the *orthogonality* of sines and cosines. Namely, that for integers m and n:

$$\begin{aligned}
\int_{-L}^{L} \sin\left(\frac{m\pi x}{L}\right) \sin\left(\frac{n\pi x}{L}\right) dx &= \delta_{m,n} L \\
\int_{-L}^{L} \cos\left(\frac{m\pi x}{L}\right) \cos\left(\frac{n\pi x}{L}\right) dx &= \delta_{m,n} L \\
\int_{-L}^{L} \sin\left(\frac{m\pi x}{L}\right) \cos\left(\frac{n\pi x}{L}\right) dx &= 0
\end{aligned}$$

Proving the above orthogonality relations is greatly simplified by the following trigonometric angle sum identity (which itself can be derived from, e.g., Euler's formula):

$$\cos(\bar{\alpha} \pm \bar{\beta}) = \cos\bar{\alpha}\cos\bar{\beta} \mp \sin\bar{\alpha}\sin\bar{\beta},$$

where we are free to define $\bar{\alpha} = \alpha x$, and $\bar{\beta} = \beta x$. This implies that

$$\begin{aligned}
\cos(\alpha x - \beta x) - \cos(\alpha x + \beta x) &= (\cos(\alpha x)\cos(\beta x) + \sin(\alpha x)\sin(\beta x)) - (\cos(\alpha x)\cos(\beta x) - \sin(\alpha x)\sin(\beta x)) \\
&= 2\sin(\alpha x)\sin(\beta x).
\end{aligned}$$

Defining $\alpha = \frac{m\pi}{L}$ and $\beta = \frac{n\pi}{L}$, we get for the top integral:

$$\begin{aligned}
\int_{-L}^{L} \sin(\alpha x)\sin(\beta x)dx &= 2\int_{0}^{L} \sin(\alpha x)\sin(\beta x)dx \quad (\text{odd} \times \text{odd} = \text{even}) \\
&= 2 \times \frac{1}{2}\int_{0}^{L} \left[\cos\left((\alpha - \beta)x\right) - \cos\left((\alpha + \beta)x\right)\right] dx \\
&= \int_{0}^{L} \left[\cos\left((\alpha - \beta)x\right) - \cos\left((\alpha + \beta)x\right)\right] dx.
\end{aligned}$$

Now, for $m = n \neq 0$, we get $\alpha = \beta$, and

$$\begin{aligned}
\int_{-L}^{L} \sin(\alpha x)\sin(\beta x)dx &= \int_{0}^{L} \left[\cos(0) - \cos\left(\frac{2m\pi}{L}x\right)\right] dx \\
&= \int_{0}^{L} \left[1 - \cos\left(\frac{2m\pi}{L}x\right)\right] dx \\
&= L - \left[-\sin\left(\frac{2m\pi}{L}x\right)\right]_{0}^{L} \\
&= L.
\end{aligned}$$

Obviously for $m = n = 0$, the integral becomes $\int_0^L [\cos(0) - \cos(0)]dx = 0$, which is actually easier to see from the original expression of the integral.

For $m! = n$, we get $\alpha \neq \beta$, and

$$\begin{aligned}\int_{-L}^{L} \sin(\alpha x)\sin(\beta x)dx &= \int_{0}^{L} \left[\cos\left((\alpha-\beta)x\right) - \cos\left((\alpha+\beta)x\right)\right]dx \\ &= \left[\frac{\sin\left((\alpha-\beta)x\right)}{(\alpha-\beta)}\right]_0^L - \left[\frac{\sin\left((\alpha+\beta)x\right)}{(\alpha+\beta)}\right]_0^L \\ &= \frac{\sin\left((\alpha-\beta)L\right)}{(\alpha-\beta)} - \frac{\sin\left((\alpha+\beta)L\right)}{(\alpha+\beta)} \\ &= \frac{\sin\left(\frac{(m-n)\pi L}{L}\right)}{\frac{(m-n)\pi}{L}} - \frac{\sin\left(\frac{(m+n)\pi L}{L}\right)}{\frac{(m+n)\pi}{L}} \\ &= L\left[\frac{\sin\left((m-n)\pi\right)}{(m-n)\pi} - \frac{\sin\left((m+n)\pi\right)}{(m+n)\pi}\right] \\ &= 0,\end{aligned}$$

since $m - n \neq 0$ and $m + n \neq 0$ (thus we are not dividing by zero), and $\sin(\ell\pi) = 0$ for any integer ℓ (thus the numerators are both zero).

We conclude that

$$\int_{-L}^{L} \sin\left(\frac{m\pi x}{L}\right)\sin\left(\frac{n\pi x}{L}\right)dx = \delta_{m,n}L.$$

Open Exercise

Using the same approach as above, prove the remaining orthogonality relations:

$$\begin{aligned}\int_{-L}^{L} \cos\left(\frac{m\pi x}{L}\right)\cos\left(\frac{n\pi x}{L}\right)dx &= \delta_{m,n}L \\ \int_{-L}^{L} \sin\left(\frac{m\pi x}{L}\right)\cos\left(\frac{n\pi x}{L}\right)dx &= 0.\end{aligned}$$

Let's return to the original expression for Fourier series:

$$f(x) = \frac{a_0}{2} + \sum_{m=1}^{\infty}\left[a_m\cos\left(\frac{m\pi x}{L}\right) + b_m\sin\left(\frac{m\pi x}{L}\right)\right]$$

Multiply both sides by $\sin\left(\frac{n\pi x}{L}\right)$ and then take the integral $\int_{-L}^{L} dx$ of both sides:

$$\begin{aligned}\int_{-L}^{L} f(x)\sin\left(\frac{n\pi x}{L}\right)dx &= \int_{-L}^{L}\frac{a_0}{2}\sin\left(\frac{n\pi x}{L}\right)dx + \int_{-L}^{L}\sum_{m=1}^{\infty} a_m\sin\left(\frac{n\pi x}{L}\right)\cos\left(\frac{m\pi x}{L}\right)dx \\ &+ \int_{-L}^{L}\sum_{m=1}^{\infty} b_m\sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi x}{L}\right)dx.\end{aligned}$$

The a_0 term will evaluate to zero, since the integral of an odd function over an even interval across zero will be zero (**Corollary #2**, Eq. 7.2). As for the remaining terms, the sum of an integral is equal to the integral of a sum, so

$$\begin{aligned}\int_{-L}^{L} f(x)\sin\left(\frac{n\pi x}{L}\right)dx &= 0 + \sum_{m=1}^{\infty} a_m\int_{-L}^{L}\sin\left(\frac{n\pi x}{L}\right)\cos\left(\frac{m\pi x}{L}\right)dx \\ &+ \sum_{m=1}^{\infty} b_m\int_{-L}^{L}\sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi x}{L}\right)dx \\ &= 0 + 0 + \sum_{m=1}^{\infty} b_m\delta_{m,n}L \\ &= b_nL\end{aligned}$$

Thus we have found the expression for b_n in terms of an integral of our original function only. Next, let's compute a_n using a similar trick.

Multiply both sides of the original Fourier series expression by $\cos\left(\frac{n\pi x}{L}\right)$ and then take the integral $\int_{-L}^{L} dx$ of both sides:

$$\begin{aligned}
\int_{-L}^{L} f(x)\cos\left(\frac{n\pi x}{L}\right)dx &= \int_{-L}^{L}\frac{a_0}{2}\cos\left(\frac{n\pi x}{L}\right)dx + \int_{-L}^{L}\sum_{m=1}^{\infty} a_m\cos\left(\frac{n\pi x}{L}\right)\cos\left(\frac{m\pi x}{L}\right)dx \\
&\quad + \int_{-L}^{L}\sum_{m=1}^{\infty} b_m\cos\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi x}{L}\right)dx \\
&= \int_{-L}^{L}\frac{a_0}{2}\cos\left(\frac{n\pi x}{L}\right)dx + \sum_{m=1}^{\infty} a_m\int_{-L}^{L}\cos\left(\frac{n\pi x}{L}\right)\cos\left(\frac{m\pi x}{L}\right)dx \\
&\quad + \sum_{m=1}^{\infty} b_m\int_{-L}^{L}\cos\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi x}{L}\right)dx \\
&= \int_{-L}^{L}\frac{a_0}{2}\cos\left(\frac{n\pi x}{L}\right)dx + \sum_{m=1}^{\infty} a_m\delta_{m,n}L + 0.
\end{aligned}$$

Notice that when $n = 0$, $\cos\left(\frac{n\pi x}{L}\right) = 1$, so the second term vanishes and we get

$$\begin{aligned}
\int_{-L}^{L}\frac{a_0}{2}dx &= \int_{-L}^{L} f(x)dx \\
\implies a_0 &= \frac{1}{L}\int_{-L}^{L} f(x)dx
\end{aligned}$$

Now for $n > 0$ the the first term vanishes (can prove by taking the simple integral) and we get:

$$a_n = \frac{1}{L}\int_{-L}^{L} f(x)\cos\left(\frac{n\pi x}{L}\right)dx$$

7.4.1 Fourier Series Summary

Fourier Series Summary

A function $f(x)$ can be represented between $x = -L$ and $x = +L$ as a Fourier series

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty}\left[a_n\cos\left(\frac{n\pi x}{L}\right) + b_n\sin\left(\frac{n\pi x}{L}\right)\right],$$

where the Fourier coefficients a_n and b_n can be computed via

$$\begin{aligned}
a_n &= \frac{1}{L}\int_{-L}^{L} f(x)\cos\left(\frac{n\pi x}{L}\right)dx \\
b_n &= \frac{1}{L}\int_{-L}^{L} f(x)\sin\left(\frac{n\pi x}{L}\right)dx.
\end{aligned}$$

The Fourier series representation of the function $f(x)$ will be periodic with period of $2L$.

7.4.2 Fourier Series Examples

Example #1

Compute the Fourier series of $f(x) = 1 - x^2$, $x \in [-1, 1]$.

This boils down to computing the Fourier coefficients for this function, for which we just derived expressions:

$$\begin{aligned}
a_0 &= \frac{1}{L}\int_{-L}^{L}(1-x^2)dx \\
&= 2\frac{1}{L}\int_{0}^{L}(1-x^2)dx = 2\left[x - \frac{x^3}{3}\right]_0^L = 2\left[L - \frac{L^3}{3}\right]. \\
& L = 1 \implies a_0 = 2 - \frac{2}{3} = \frac{4}{3}
\end{aligned}$$

$$\begin{aligned}
a_m &= \frac{1}{L}\int_{-L}^{L}(1-x^2)\cos\left(\frac{m\pi x}{L}\right)dx \\
&= \frac{2}{L}\int_{0}^{L}(1-x^2)\cos\left(\frac{m\pi x}{L}\right)dx \\
&= \frac{2}{L}\left[\int_0^L \cos\left(\frac{m\pi x}{L}\right)dx - \int_0^L x^2\cos\left(\frac{m\pi x}{L}\right)dx\right] \\
&= \frac{2}{L}\left[\frac{L}{m\pi}\sin\left(\frac{m\pi x}{L}\right)\right]_0^L - \frac{2}{L}\left[\int_0^L x^2\cos\left(\frac{m\pi x}{L}\right)dx\right] \\
&= 0 - \frac{2}{L}\left[\int_0^L x^2\cos\left(\frac{m\pi x}{L}\right)dx\right] \\
& (L = 1 \implies) = -2\left[\frac{(\pi^2 m^2 x^2 - 2)\sin(\pi m x) + 2\pi m x\cos(\pi m x)}{\pi^3 m^3}\right]_0^1 \\
&= -4\frac{\cos(\pi m)}{\pi^2 m^2} = -\frac{4}{(m\pi)^2}(-1)^m
\end{aligned}$$

$$b_m = \frac{1}{L}\int_{-L}^{L}(1-x^2)\sin\left(\frac{n\pi x}{L}\right)dx = 0$$

Notice we used integration by parts, as well as properties of even and odd functions to evaluate the above integrals.

We conclude by writing the Fourier series:

$$\begin{aligned}
f(x) &= \frac{a_0}{2} + \sum_{m=1}^{\infty}\left[a_m\cos\left(\frac{m\pi x}{L}\right)\right] \\
&= \frac{2}{3} - \frac{4}{\pi^2}\sum_{m=1}^{\infty}\frac{(-1)^m}{m^2}\cos(m\pi x)
\end{aligned}$$

Notice that the Fourier coefficients converge as n^2, giving us an idea of how many terms we would need to compute the Fourier series of this function at a given desired precision. Note also that this example is quite artificial, as it would be a very rare situation in which we would wish to replace a simple polynomial with a Fourier basis.

Let's look at another, simpler example:

Example #2

Compute the Fourier series for the step function

$$f_{\text{step}}(x) = \begin{cases} -1, & \text{if } -1 \le x \le 0 \\ 1, & \text{if } 0 < x < 1 \end{cases} \tag{7.3}$$

Let's stop and analyze this function first before computing coefficients. Notice that it's an odd function. From that, we know immediately that all the cosine coefficients a_m in the Fourier expansion must be zero, including a_0. Thus we only need to compute the sine b_m coefficients.

Note also that $L = 1$, which we'll apply at the outset:

$$\begin{aligned}
b_m &= \int_{-1}^{0} (-1)\sin(m\pi x)dx + \int_{0}^{1} (+1)\sin(m\pi x)dx \\
&= \frac{1}{m\pi}\left(-[-\cos(m\pi x)]_{-1}^{0} + [-\cos(m\pi x)]_{0}^{1}\right) \\
(&= \frac{2}{m\pi}\left([-\cos(m\pi x)]_{0}^{1}\right)) \\
&= \frac{1}{m\pi}\left(+[1-\cos(-m\pi)] - [\cos(m\pi) - 1]\right) \\
&= \frac{1}{m\pi}\left(+[1-\cos(m\pi)] - [\cos(m\pi) - 1]\right) \\
&= \frac{2}{m\pi}\left([1-\cos(m\pi)]\right) \\
&= \frac{2}{m\pi}\left([1-(-1)^m]\right)
\end{aligned}$$

Thus the coefficients are zero for m even and for m odd, they are $4/(m\pi)$. Thus the Fourier series of the step function is simply

$$f_{\text{step,FS}}(x) = \frac{4}{\pi}\left(\sin(\pi x) + \frac{\sin(3\pi x)}{3} + \frac{\sin(5\pi x)}{5} + \dots\right) = \frac{4}{\pi}\sum_{n=1}^{\infty}\frac{\sin((2n-1)\pi x)}{2n-1} \tag{7.4}$$

Notice convergence of coefficients is very slow, dropping in magnitude at a rate proportional to $1/n$. How do we know the Fourier series will converge to the exact function values at all? To what value will the Fourier series converge where the function is discontinuous?

7.4.3 Fourier Convergence Theorem

The Fourier series of a piecewise continuous function $f(x)$ will converge to the exact value of that function at all points x where the function is continuous. At other points, the series will converge to $[f(x_+) + f(x_-)]/2$.

Extending the Fourier series *outside* the original interval of $x = -L$ to $x = +L$ will yield a function that is *periodic* with period $2L$. This means that to determine where the Fourier series converges outside the original interval, you simply periodically extend the function outside the interval.

To understand what it means to periodically extend a function, imagine making a stamp out of the function in the original interval. Now wet the stamp with ink, take a clean sheet of graph paper, and first stamp the function in the original interval. You now have the original function in the interval from $-L$ to $+L$. Next, without rotating or shifting the stamp up or down in any way, stamp the function again to the immediate right and left. This determines how the function will be defined from $-3L$ to $-L$, and from $+L$ to $+3L$, respectively. You can continue stamping to $x \to \infty$ and $x \to -\infty$.

Caution! Periodically extending the original function will usually result in discontinuities at $x = \pm L$. When this happens, you must be careful to apply the Fourier Convergence Theorem to properly know to what points Fourier series of the function will converge at these discontinuities. (Recall: it's just the average value of the function.)

Exercise #1

Sketch the following function's Fourier series representation from $x = -2$ to $x = +2$:

$$f(x) = \begin{cases} 0, & \text{if } -1 \leq x < 0 \\ x^2, & \text{if } 0 \leq x < 1 \end{cases}$$

Solution

The answer will be a plot of the function

$$f(x) = \begin{cases} 0, & \text{if } x = -2 \\ (x+2)^2, & \text{if } -2 < x < -1 \\ \frac{1}{2}, & \text{if } x = -1 \\ 0, & \text{if } -1 < x \leq 0 \\ x^2, & \text{if } 0 < x < 1 \\ \frac{1}{2}, & \text{if } x = 1 \\ 0, & \text{if } 1 < x \leq 2 \end{cases}$$

which is a periodic extension of the original function, which was defined on the interval $x \in [-1, 1)$.

Exercise #2

Given the following function $f(x)$:

$$f(x) = \begin{cases} -10x, & -1 \leq x < 0 \\ 10x + 4, & 0 \leq x < 1, \end{cases}$$

find to which value the Fourier series of $f(x)$, $f_{\text{FS}}(x)$, will converge at the following points (use the Fourier Convergence Theorem): $x = 0$, $x = 1$, $x = -1$, and $x = -\frac{1}{2}$.

Solution

$f_{\text{FS}}(0) = 2$, $f_{\text{FS}}(1) = 12$, $f_{\text{FS}}(-1) = 12$, and $f_{\text{FS}}(-1/2) = 5$.

Exercise #3

Given the following function $f(x)$:

$$f(x) = \begin{cases} 3x^3 - 2x^2, & -1 \leq x < -\frac{1}{2} \\ \ln(e^x), & -\frac{1}{2} \leq x < \frac{1}{2} \\ x^2, & \frac{1}{2} \leq x < 1, \end{cases}$$

find to which value the Fourier series of $f(x)$, $f_{\text{FS}}(x)$, will converge at the following points (use the Fourier Convergence Theorem): $x = 0$, $x = 1$, $x = -1$, $x = -\frac{1}{2}$, and $x = 23,101$.

Solution

$f_{\text{FS}}(0) = 0$, $f_{\text{FS}}(1) = -2$, $f_{\text{FS}}(-1) = -2$, $f_{\text{FS}}(-1/2) = -11/16$, and $f_{\text{FS}}(23,101) = -2$.

7.4.4 Gibbs Phenomenon

Although the Fourier convergence theorem guarantees that the function will converge everywhere, it does not say how rapidly this convergence will take place. Discontinuities can be seen as a zero-wavelength feature, requiring very large m values to approximate. Thus Fourier series will converge extremely slowly at discontinuities. In fact, Fourier series truncated at a finite M will exhibit the *Gibbs Phenomenon*, or "ringing artifacts" at discontinuities.

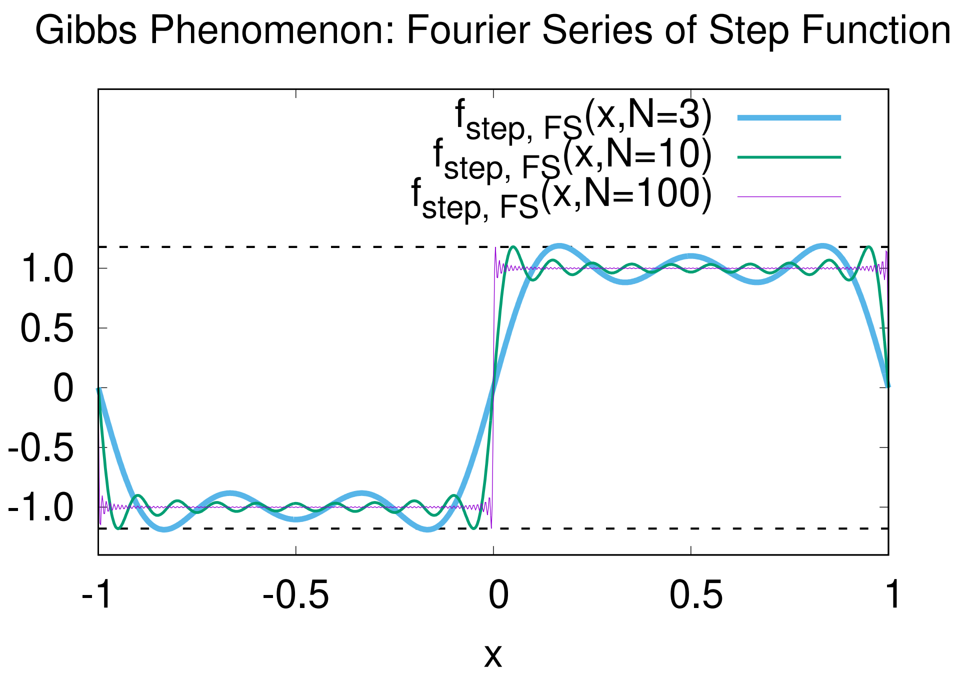

Figure 7.1: Gibbs phenomenon appearing from the evaluation of a finite Fourier series sum $f_{\text{step,FS}}(x, N)$ (Eq. 7.5) of the step function $f_{\text{step}}(x)$ (Eq. 7.3). Gibbs oscillations appear at points of discontinuity: $x = \pm 1$ and $x = 0$ in this plot. At each discontinuity, the expected Gibbs overshoot of about 1.2 ($= 1.09h$, where $h = 2$) is plotted as horizontal dashed lines. Finite sums with $N = 3$, $N = 10$, and $N = 100$ nonzero terms in the Fourier series are plotted.

As an example, consider the Fourier series representation of the step function $f_{\text{step,FS}}(x)$ (Eq. 7.4), except summed to a *finite* N:

$$f_{\text{step,FS}}(x, N) = \frac{4}{\pi} \sum_{n=0}^{N} \frac{\sin((2n-1)\pi x)}{2n-1}. \tag{7.5}$$

Given its incredibly sharp features, this function seems like a poor fit to the incredibly smooth Fourier basis, even for large values of N. As shown in Fig. 7.1, for values of $N = 3$, 10, and even 100, oscillations near the discontinuities at $x = 0$ and $x = \pm 1$ shrink in wavelength, but notice the overshoot at the discontinuities *does not appear to converge.*

Knowing that the Fourier convergence theorem guarantees the Gibbs phenomenon will be quenched at *infinite* terms in the sum, one might not be too concerned about this phenomenon. As it turns out, for *all finite* sums in a Fourier series representation of a discontinuity of total height $h = 2$ like in the example shown in Fig. 7.1, the discontinuity will be overshot on each side by $h \times 0.089392\ldots$, or about $1.09h$ on the left and $1.09h$ on the right of the discontinuity (this number is related to the Wilbraham-Gibbs constant).

Again, it must be stressed that this overshoot does not converge away with finite number of terms. At higher N in the Fourier series representation, the overshoot remains the same height but gets closer and closer to the discontinuity. Only in the limit $N \to \infty$ does the overshoot disappear; all finite values of N will yield this same overshoot.

Note that the Gibbs phenomenon is not restricted to Fourier series. In fact it occurs across all smooth basis functions, including polynomial bases.

MATH 521, Fall 2018 Notes

Prof. Zachariah B. Etienne

Chapter 8: Polynomial Interpolation & Extrapolation

Suppose we are given data for a function $f(x)$ at a discrete number of points $x_0 < x_1 < ... < x_{N-1} < x_N$ only. Then, assuming $f(x)$ is smooth, how do we estimate values for other points $x \in [x_0, x_N]$? How about $x \notin [x_0, x_N]$?

Definition: The process of estimating values of $f(x)$ at $x \in [x_0, x_N]$ based on known values of the function $\{f(x_0), f(x_1), ..., f(x_N)\}$ is known as **interpolation**.

Definition: The process of estimating values of $f(x)$ at $x \notin [x_0, x_N]$ based on known values of the function $\{f(x_0), f(x_1), ..., f(x_N)\}$ is known as **extrapolation**.

In the following sections, we will exclusively refer to *interpolation*, but the discussion could equally well be applied to *extrapolation*.

8.1 Lagrange Polynomial Interpolation

Example #1

Suppose we have values of a function at only two points $x_0 < x_1$: $f_0 = f(x_0)$ and $f_1 = f(x_1)$. How do we interpolate values of the function between x_0 and x_1? How do we extrapolate the function outside of this range of x?

The equation of a line (i.e., a first-order polynomial $P_1(x)$) between f_0 and f_1 can be written

$$P_1(x) = c_1 x + c_0,$$

where c_1 is the slope and c_0 is the y-intercept. c_0 and c_1 are unknown coefficients, but we are given values of the function at two points x_0 and x_1. We wish for the line $P_1(x)$ to pass through the function at these two points. Thus we obtain two equations and two unknowns:

$$\begin{aligned} P_1(x_0) = f(x_0) = f_0 &= c_1 x_0 + c_0 \\ P_1(x_1) = f(x_1) = f_1 &= c_1 x_1 + c_0. \end{aligned}$$

Definition: Writing this set of equations in matrix form, we obtain the **Vandermonde matrix $\boldsymbol{A}$** for computing the unknown linear interpolation coefficients.

$$\boldsymbol{A}\boldsymbol{c} = \begin{bmatrix} 1 & x_0 \\ 1 & x_1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \boldsymbol{f} = \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}.$$

Remember that x_0, x_1, f_0, and f_1 are known, and c_0 and c_1 are unknowns. Solving this linear system of equations for c_0 and c_1, we obtain

$$\begin{aligned} c_0 &= f_0 - x_0 \frac{f_1 - f_0}{x_1 - x_0} \\ c_1 &= \frac{f_1 - f_0}{x_1 - x_0}. \end{aligned}$$

Notice the second equation yields the slope of the function, and in the limit where $x_1 \to x_0$, this is precisely the definition of derivative!

Thus the equation that yields the interpolated or extrapolated value of the function $f(x)$ at any x based on this linear approximation is given by

$$\begin{aligned} P_1(x) &= \frac{f_1 - f_0}{x_1 - x_0} x + f_0 - x_0 \frac{f_1 - f_0}{x_1 - x_0} \\ &= \frac{f_1 - f_0}{x_1 - x_0}(x - x_0) + f_0 \\ &= (f_1 - f_0)\frac{x - x_0}{x_1 - x_0} + f_0. \end{aligned} \tag{8.1}$$

Example #2

Repeat **Example #1**, but if the function $f(x)$ is given at three points $x_0 < x_1 < x_2$.

The lowest degree polynomial with 3 coefficients is a quadratic:

$$P_2(x) = c_2 x^2 + c_1 x + c_0. \tag{8.2}$$

Knowing the value of $f(x)$ at x_0, x_1, and x_2 provides us with three equations and three unknowns:

$$\begin{aligned} P_2(x_0) = f(x_0) = f_0 &= c_2 x_0^2 + c_1 x_0 + c_0 \\ P_2(x_1) = f(x_1) = f_1 &= c_2 x_1^2 + c_1 x_1 + c_0 \\ P_2(x_2) = f(x_2) = f_2 &= c_2 x_2^2 + c_1 x_2 + c_0, \end{aligned}$$

where again we require that the polynomial fitting the function overlaps the function at these three points. Once we have these unknowns c_2, c_1, and c_0, we can immediately perform quadratic interpolation and extrapolation of the function.

Writing this set of equations in matrix form, we obtain the **Vandermonde matrix** for computing the unknown quadratic interpolation coefficients:

$$\boldsymbol{A}\boldsymbol{c} = \begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix} = \boldsymbol{f}.$$

The inverse of the Vandermonde matrix, $\boldsymbol{A}^{-1}$, yields the coefficients:

$$\begin{aligned} \boldsymbol{c} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \boldsymbol{A}^{-1}\boldsymbol{f} &= \begin{bmatrix} \frac{x_1 x_2}{(x_0-x_1)(x_0-x_2)} & \frac{x_0 x_2}{(x_1-x_0)(x_1-x_2)} & \frac{x_0 x_1}{(x_2-x_0)(x_2-x_1)} \\ -\frac{x_1+x_2}{(x_0-x_1)(x_0-x_2)} & -\frac{x_0+x_2}{(x_1-x_0)(x_1-x_2)} & -\frac{x_0+x_1}{(x_2-x_0)(x_2-x_1)} \\ \frac{1}{(x_0-x_1)(x_0-x_2)} & \frac{1}{(x_1-x_0)(x_1-x_2)} & \frac{1}{(x_2-x_0)(x_2-x_1)} \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix} \\ &= \begin{bmatrix} x_1 x_2 & x_0 x_2 & x_0 x_1 \\ -(x_1 + x_2) & -(x_0 + x_2) & -(x_0 + x_1) \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \frac{f_0}{(x_0-x_1)(x_0-x_2)} \\ \frac{f_1}{(x_1-x_0)(x_1-x_2)} \\ \frac{f_2}{(x_2-x_0)(x_2-x_1)} \end{bmatrix} \\ &= \begin{bmatrix} x_1 x_2 & x_0 x_2 & x_0 x_1 \\ -(x_1 + x_2) & -(x_0 + x_2) & -(x_0 + x_1) \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \tilde{f}_0 \\ \tilde{f}_1 \\ \tilde{f}_2 \end{bmatrix} \end{aligned}$$

Next, let's plug these coefficients into the original quadratic expression (Eq. 8.2) to obtain the expression for any interpolated value $f(x)$ for a quadratic polynomial:

$$\begin{aligned}
P_2(x) &= c_2x^2 + c_1x + c_0 \\
&= \left[\tilde{f}_0 + \tilde{f}_1 + \tilde{f}_2\right] x^2 - \left[(x_1+x_2)\tilde{f}_0 + (x_0+x_2)\tilde{f}_1 + (x_0+x_1)\tilde{f}_2\right] x + \left[x_1x_2\tilde{f}_0 + x_0x_2\tilde{f}_1 + x_0x_1\tilde{f}_2\right] \\
&= \tilde{f}_0\left[x^2 - (x_1+x_2)x + x_1x_2\right] + \tilde{f}_1\left[x^2 - (x_0+x_2)x + x_0x_2\right] + \tilde{f}_2\left[x^2 - (x_0+x_1)x + x_0x_1\right] \\
&= \tilde{f}_0\left[x(x-x_1) + x_2(x_1-x)\right] + \tilde{f}_1\left[x(x-x_0) + x_2(x_0-x)\right] + \tilde{f}_2\left[x(x-x_1) + x_0(x_1-x)\right] \\
&= \tilde{f}_0\left[x(x-x_1) - x_2(x-x_1)\right] + \tilde{f}_1\left[x(x-x_0) - x_2(x-x_0)\right] + \tilde{f}_2\left[x(x-x_1) - x_0(x-x_1)\right] \\
&= (x-x_2)(x-x_1)\tilde{f}_0 + (x-x_2)(x-x_0)\tilde{f}_1 + (x-x_0)(x-x_1)\tilde{f}_2 \\
&= \frac{\prod_{i\neq 0}(x-x_i)}{\prod_{i\neq 0}(x_0-x_i)} f_0 + \frac{\prod_{i\neq 1}(x-x_i)}{\prod_{i\neq 1}(x_1-x_i)} f_1 + \frac{\prod_{i\neq 2}(x-x_i)}{\prod_{i\neq 2}(x_2-x_i)} f_2 \\
&= \sum_{i=0}^{2} \ell_i(x) f(x_i), \quad \text{where} \quad \ell_i(x) = \prod_{\substack{0\le j\le 2,\\ i\neq j}} \frac{(x-x_j)}{(x_i-x_j)} \qquad (8.3)
\end{aligned}$$

Notice that the above expression holds true for first-order (linear) polynomials as well (cf. Eq. 8.1), if each two in the bottom expression is replaced by a one.

Looking at the inverse of the Vandermonde matrix for third-order polynomial interpolation,

$$\boldsymbol{A}^{-1}\boldsymbol{f} = \begin{bmatrix} -x_1x_2x_3 & -x_0x_2x_3 & -x_0x_1x_3 & -x_0x_1x_2 \\ x_1x_2+x_1x_3+x_2x_3 & x_0x_2+x_0x_3+x_2x_3 & x_0x_1+x_0x_3+x_1x_3 & x_0x_1+x_0x_2+x_1x_2 \\ -(x_1+x_2+x_3) & -(x_0+x_2+x_3) & -(x_0+x_1+x_3) & -(x_0+x_1+x_2) \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \tilde{f}_0 \\ \tilde{f}_1 \\ \tilde{f}_2 \\ \tilde{f}_3 \end{bmatrix},$$

where, extending the pattern for $N = 2$ order polynomials to $N = 3$ order, we define

$$\tilde{f}_i = f_i \Bigg/ \prod_{\substack{0\le j\le 3,\\ j\neq i}} (x_i - x_j)\ ,$$

we will find by following the same steps as with second order that at third order

$$\begin{aligned}
P_3(x) =& c_3x^3 + c_2x^2 + c_1x + c_0 \\
=& \sum_{i=0}^{3} \ell_i(x) f(x_i) \quad \text{where} \quad \ell_i(x) = \prod_{\substack{0\le j\le 3,\\ i\neq j}} \frac{(x-x_j)}{(x_i-x_j)} \qquad (8.4)
\end{aligned}$$

as well. As we have shown, the above formula holds for $P_N(x)$, where the upper limits on the sum and product are $n = 1$, 2, or 3. It can be proven that the pattern we found in Eqs. 8.1, 8.3, and 8.4 continues to hold for *all* polynomial degrees N, so that

$$P_N(x) = \sum_{i=0}^{N} c_i x^i = \sum_{i=0}^{N} \ell_i(x) f(x_i), \quad \text{where} \quad \ell_i(x) = \prod_{\substack{0\le j\le N,\\ i\neq j}} \frac{(x-x_j)}{(x_i-x_j)}. \qquad (8.5)$$

Of course, this assumes that we have evaluated $f(x)$ at $N+1$ points (because a sum running from 0 to N contains $N+1$ terms). Edward Warring may be the discoverer of this formula, but Joseph Louis Lagrange published it about 15 years after Warring's discovery, in 1795. Thus it is known as the **Lagrange interpolation formula**.

Example #3

Akin to example 1, pg. 88 of Kunz's Numerical Analysis, McGraw-Hill 1957, LCCCN 56-13395
Using Lagrange polynomial interpolation with a rank-4 polynomial, evaluate $\log_{10} 47$ from the following table:

$$\begin{aligned}\log_{10} 2 &\approx 0.3010299957\\ \log_{10} 3 &\approx 0.4771212547\\ \log_{10} 5 &\approx 0.6989700043\\ \log_{10} 7 &\approx 0.8450980400.\end{aligned}$$

What is the relative error in this interpolation?

Recalling that $\log_{10}(ab) = \log_{10} a + \log_{10} b$, we can quickly construct all the closest values where the logarithm is known:

$$\begin{aligned}
f_0 = \log_{10} 42 = &\quad \log_{10}(2\cdot 3\cdot 7) = \log_{10} 2 + \log_{10} 3 + \log_{10} 7 &&\approx 1.6232492904\\
f_1 = \log_{10} 45 = &\quad \log_{10}(3^2\cdot 5) = 2\log_{10} 3 + \log_{10} 5 &&\approx 1.6532125137\\
f_2 = \log_{10} 48 = &\quad \log_{10}(2^4\cdot 3) = 4\log_{10} 2 + \log_{10} 3 &&\approx 1.6812412375\\
f_3 = \log_{10} 49 = &\quad \log_{10} 7^2 = 2\log_{10} 7 &&\approx 1.6901960800\\
f_4 = \log_{10} 50 = &\quad \log_{10}(5^2\cdot 2) = 2\log_{10} 5 + \log_{10} 2 &&\approx 1.6989700043.
\end{aligned}$$

Thus we have found $x_0 = 42$, $x_1 = 45$, $x_2 = 48$, $x_3 = 49$, and $x_4 = 50$. The interpolation yields, for $x = 47$,

$$\begin{aligned}
P_4(47) \quad = \quad & \frac{x-x_1}{x_0-x_1}\frac{x-x_2}{x_0-x_2}\frac{x-x_3}{x_0-x_3}\frac{x-x_4}{x_0-x_4}f_0 + \frac{x-x_0}{x_1-x_0}\frac{x-x_2}{x_1-x_2}\frac{x-x_3}{x_1-x_3}\frac{x-x_4}{x_1-x_4}f_1 +\\
& \frac{x-x_0}{x_2-x_0}\frac{x-x_1}{x_2-x_1}\frac{x-x_3}{x_2-x_3}\frac{x-x_4}{x_2-x_4}f_2 + \frac{x-x_0}{x_3-x_0}\frac{x-x_1}{x_3-x_1}\frac{x-x_2}{x_3-x_2}\frac{x-x_4}{x_3-x_4}f_3 +\\
& \frac{x-x_0}{x_4-x_0}\frac{x-x_1}{x_4-x_1}\frac{x-x_2}{x_4-x_2}\frac{x-x_3}{x_4-x_3}f_4\\
= \quad & \frac{(2)(-1)(-2)(-3)}{(-3)(-6)(-7)(-8)}f_0 + \frac{(5)(-1)(-2)(-3)}{(3)(-3)(-4)(-5)}f_1 + \frac{(5)(2)(-2)(-3)}{(6)(3)(-1)(-2)}f_2 + \frac{(5)(2)(-1)(-3)}{(7)(4)(1)(-1)}f_3 + \frac{(5)(2)(-1)(-2)}{(8)(5)(2)(1)}f_4\\
= \quad & -\frac{1}{84}f_0 + \frac{1}{6}f_1 + \frac{5}{3}f_2 - \frac{15}{14}f_3 + \frac{1}{4}f_4 = 1.6720978814,
\end{aligned}$$

The exact value of $\log_{10} 47$ to 15 significant digits is `1.67209785793572`, meaning that the value obtained through interpolation has relative error of

$$E_{\text{Rel}} = \frac{|1.6720978814 - 1.672097857935|}{1.672097857935} \approx 1.40\times 10^{-8}.$$

8.2 Numerical Implementation

Suppose we wish to implement the Lagrange interpolation formula of Eq. 8.5 numerically. As input, we will need some function $f(x)$ evaluated at N points x_j, where $x_0 < x_1 < ... < x_j < x_{j+1} < ... < x_N$, and as output, we will be able to approximately evaluate $f(x)$ at any $x \in [x_0, x_N]$. This does not exclude use of this formula for *extrapolation*, however; it can be evaluated at any x outside the interval as well, and such an extrapolation can be quite accurate for some range of $x \notin [x_0, x_N]$ depending on the underlying function.

We might find this strategy quite useful if $f(x)$ is extremely expensive to compute, and we wish to evaluate an approximate expression for $f(x)$ at a very large number of points x in the neighborhood of $[x_0, x_N]$. It could also be useful if we cannot evaluate the function at points between x_i and x_{i+1} (e.g., because our measuring instrument does not provide the data), yet we have a reasonable expectation that the function $f(x)$ is smooth.

In pseudocode, the Lagrange interpolation formula can be written

Lagrange Polynomial Interpolation Pseudocode

```
output = 0.0
do i=0,N
   l_i = 1.0
   do j=0,N
      if(j not equal to i) then
         l_i = l_i * (x - x[j])/(x[i] - x[j])
      end if
   end do
   output = output + l_i * f[i]
end do
```

Exercise:

Suppose we need to interpolate M points, using Nth order Lagrange polynomial interpolation. We choose to use the algorithm given above in pseudocode. What is the computational complexity of such an algorithm?

Solutions:

$\mathcal{O}(MN^2)$.

8.3 Lagrange Interpolation in Barycentric Form

The above algorithm is *functional*, but *inefficient* for cases in which we must interpolate *many* points between x_0 and x_N, as each point requires $\mathcal{O}(N^2)$ operations for an $N-1$-order polynomial interpolation. Additional interpolations can be reduced to an $\mathcal{O}(N)$ operation as follows. First consider the Lagrange interpolation formula (Eq. 8.5):

$$P_N(x) = \sum_{i=0}^{N} f(x_i)\ell_i(x) = \sum_{i=0}^{N} f(x_i) \prod_{\substack{0\le j\le N,\\ j\ne i}} \frac{x-x_j}{x_i-x_j}.$$

Notice the **Barycentric weights**

$$w_i = \prod_{\substack{0\le j\le N,\\ j\ne i}} \frac{1}{x_i-x_j}$$

can be precomputed, and separated from the rest of the product appearing in $P_N(x)$:

$$P_N(x) = \sum_{i=0}^{N}\left(f(x_i) w_i \prod_{\substack{0\le j\le N,\\ j\ne i}} (x-x_j)\right).$$

The $\mathcal{O}(N)$ product on the right still depends on i, so this is still an $\mathcal{O}(N^2)$ algorithm. However, this dependency can be removed by defining a new quantity $\ell(x)$ that *does not* depend on i:

$$\ell(x) = \prod_{m=0}^{N}(x-x_m) = (x-x_i)\prod_{\substack{0\le j\le N,\\ j\ne i}}(x-x_j) \implies \prod_{\substack{0\le j\le N,\\ j\ne i}}(x-x_j) = \frac{\ell(x)}{x-x_i}.$$

We now have the **Lagrange interpolation formula in Barycentric form**:

$$P_N(x) = \ell(x)\sum_{i=0}^{N}\frac{f(x_i)w_i}{x-x_i}.$$

8.3.1 Implementation of Lagrange Interpolation in Barycentric Form

From the previous page, the **Lagrange interpolation formula in Barycentric form** is given by

$$P_N(x) = \ell(x) \sum_{i=0}^{N} \frac{f(x_i) w_i}{x - x_i}.$$

Here is the basic numerical approach for efficient implementation of the above expression.

Given some number of points M that need to be interpolated in the range $x \in (x_0, x_N)$, we first precompute and store an $N+1$-element array $f(x_i)w_i$, which is an $\mathcal{O}(N^2)$ operation, since computation of each of the N w_i's is an $\mathcal{O}(N)$ operation. Then evaluating $\ell(x)$ and the sum are *each* an $\mathcal{O}(N)$ operation, but *the operations are independent* so the total computational complexity to compute $P_N(x)$ is the *sum* of two $\mathcal{O}(N)$ operations—i.e., computing both is an $\mathcal{O}(N)$ operation.

In pseudocode,

Lagrange Polynomial Interpolation in Barycentric Form: Pseudocode

```
# Inputs: x[i] = x_i: Points at which underlying function f(x) is evaluated
#         f[i] = f_i = f(x_i): Values of function at points x_i
#         xx[i]: Points xx_i at which we evaluate P_N(xx)
# Output: P_N_xx[point]: P_N(xx[point])
do i=0,N
   w_tmp = 1.0
   do j=0,N
      if(j not equal to i)
         w_tmp = w_tmp / (x[i] - x[j])
      end if
   end do
   f_times_w[i] = f[i]*w_tmp
end do

do point=1,M
   P_N_xx[point] = 0.0
   l_of_x = 1.0
   do i=0,N
      # "l_of_x" in this loop over i is *independent* of "P_N_xx[point]".
      l_of_x = l_of_x * (xx[point] - x[i])
      P_N_xx[point] = P_N_xx[point] + f_times_w[i] / (xx[point] - x[i])
   end do
   P_N_xx[point] = P_N_xx[point] * l_of_x
end do
```

Notice that the computation of `f_times_w[i]` is a $\mathcal{O}(N^2)$ operation, but the computation of the interpolated function output (stored in variable `output`) requires $\mathcal{O}(N)$ per point x. Thus for a very large number of points to which we wish to interpolate, this results in an algorithm that costs $\mathcal{O}(MN + N^2) \approx \mathcal{O}(MN)$ for $M \gg N$.

8.4 Truncation Error in Interpolation

Note that the following discussion does not apply to extrapolation.

Lagrange's polynomial interpolation formula

$$P_N(x) = \sum_{i=0}^{N} c_i x^i = \sum_{i=0}^{N} \ell_i(x) f(x_i), \quad \text{where} \quad \ell_i(x) = \prod_{\substack{0 \le j \le N, \\ i \neq j}} \frac{(x - x_j)}{(x_i - x_j)}$$

fits a function $f(x)$, sampled at $N+1$ points, to a rank-N polynomial $P_N(x)$. If $f(x)$ is a smooth (analytic) function, then we know it can be exactly represented by a power series:

$$f(x) = \sum_{i=0}^{\infty} c_i x^i.$$

Thus in the case of such smooth functions, a polynomial interpolation can be seen as a truncated power series:

$$f(x) = \sum_{i=0}^{\infty} c_i x^i \approx P_N(x) = \sum_{i=0}^{N} d_i x^i = f(x) + E,$$

where E can be thought of as the *truncation* error. Intuitively, this error should depend on the points x_i where we sample the function and how slowly the function varies between these points. We will now proceed in deriving an expression for the error involved in fitting $f(x)$ to $P_N(x)$ at points $x_0 < x_1 < \ldots < x_i < \ldots < x_N$ and evaluating $P_N(x)$ instead of $f(x)$ at a point $x \in [x_0, x_N]$.

Definition: The **residual** or **remainder** function R_N for Lagrange polynomial interpolation at a point $x \in [x_0, x_N]$ is simply the difference between the Lagrange interpolating polynomial $P_N(x)$ and the original function $f(x)$:

$$R_N(x) = f(x) - P_N(x).$$

$P_N(x)$ is constructed by fitting a polynomial to sampled points x_i of the original function $f(x)$. Therefore, at all points where x overlaps a sampled point ($x = x_i$), the residual function *must* go to zero. So $R_N(x)$ must satisfy

$$R_N(x) = S_N(x) \times [(x - x_0)(x - x_1) \ldots (x - x_N)] = S_N(x) \prod_{i=0}^{N} (x - x_i),$$

where $S_N(x)$ is finite at all $x = x_i$. What is $S_N(x)$?

To find out, let's define a new function $W_N(t)$ such that

$$W_N(t) = R_N(t) - S_N(x) \prod_{i=0}^{N} (t - x_i).$$

Notice that S_N is marked as a function of x, so by the definition of the residual function $R_N(t)$, $W_N(t) = 0$ when $t = x$. Also, $W_N(t) = 0$ at $t = x_i$. **Therefore, $W_N(t)$ crosses zero a total of at least $N + 2$ times in the range $t \in [x_0, x_N]$.**

Assuming the $(N + 1)$st derivative $f(t)$ exists, we take the $(N + 1)$st derivative of $W_N(t)$ to get

$$\frac{d^{N+1}}{dt^{N+1}} W_N(t) = W_N^{(N+1)}(t) = R_N^{(N+1)}(t) - (N + 1)! S_N(x),$$

where the $(N+1)!$ term comes from taking $(N+1)$ derivatives of $\prod_{i=0}^{N}(t-x_i)$ with respect to t. *Exercise to reader: Prove this.*

$R_N^{(N+1)}(t)$ simplifies as well:

$$R_N^{(N+1)}(t) = f^{(N+1)}(t) - \frac{d^{N+1}}{dt^{N+1}} P_N(t) = f^{(N+1)}(t),$$

since the $(N + 1)$st derivative of a rank-N polynomial is zero.

Thus we have found that

$$W_N^{(N+1)}(t) = f^{(N+1)}(t) - (N + 1)! S_N(x).$$

Recall that $W_N(t)$ crosses zero a total of at least $N + 2$ times in the range $t \in [x_0, x_N]$. This means that $W_N'(t)$ must cross zero at least $N + 1$ times in the same range (Rolle's Theorem). Similarly, $W_N''(t)$ must cross zero at least N times. This means that $W_N^{(k)}(t)$ crosses zero at least $N + 2 - k$ times, and thus $W_N^{(N+1)}(t)$ must cross zero **at least once in the range** $t \in [x_0, x_N]$. Let's call this zero-crossing $t = \xi$.

We conclude that there exists at least one point ξ that satisfies

$$W_N^{(N+1)}(\xi) = 0 = f^{(N+1)}(\xi) - (N + 1)! S_N(x) \implies S_N(x) = \frac{f^{(N+1)}(\xi)}{(N + 1)!}.$$

Thus we can write the residual error as

$$R_N(x) = \frac{f^{(N+1)}(\xi)}{(N+1)!} \prod_{i=0}^{N} (x - x_i),$$

where $\xi \in [x_0, x_N]$. This formula is remarkably useful, as it indicates that

- Given a set of points x_i, the minimum and maximum error is given by the minimum and maximum of $|f^{(N+1)}(\xi)|$ (where $\xi \in [x_0, x_N]$), respectively.
- Increasing the order of interpolation N may decrease the residual error significantly, *provided* $f(\xi)$ is smooth (so that $f^{(N+1)}(\xi) < K$ for some constant K – i.e., $f^{(N+1)}(\xi)$ is bounded) and $\prod_{i=0}^{N}(x - x_i)$ increases more slowly than $(N+1)!$.
- **Definition**: There exists a set of points x_i such that $\prod_{i=0}^{N}(x - x_i)$, and thus the interpolation error, is minimized. This set of points is known as the **Chebyshev nodes**. For the case in which $x_0 = -1$ and $x_N = 1$, the Chebyshev nodes x_k are given by

$$x_k = \cos\left(\frac{(2N+1-2k)\pi}{2N+2}\right).$$

Example #4

Akin to example 2, pg. 90 of Kunz's Numerical Analysis, McGraw-Hill 1957, LCCCN 56-13395
Apply this error estimate strategy to **Example #3** and compare with the relative error computed in that example.

To estimate the error, first note that

$$\log_{10} x = y \implies x = 10^y \implies \ln x = y \ln 10 \implies y = \log_{10} x = \frac{\ln x}{\ln 10}.$$

Thus the derivative term can be computed as follows

$$\begin{aligned}
f(x) = \log_{10}(x) &= \frac{\ln x}{\ln 10} \\
f'(x) &= \frac{1}{\ln 10}\frac{1}{x} \\
f''(x) &= \frac{1}{\ln 10}(-1)\frac{1}{x^2} \\
f^{(3)}(x) &= \frac{1}{\ln 10}(-1)^2\frac{2!}{x^3} \\
\vdots &= \vdots \\
f^{(5)}(x) &= \frac{1}{\ln 10}(-1)^4\frac{4!}{x^5}.
\end{aligned}$$

Since $x_i \in [42, 50]$, $f^{(5)}(\xi)$ will be maximized at $\xi = 42$, yielding a value of $f^{(5)}(42) \approx$`7.98e-8`.

Next, we compute $\prod_{i=0}^{N}(x - x_i)$:

$$\prod_{i=0}^{4}(x - x_i) = (5)(2)(-1)(-2)(-3) = -60$$

Thus we expect our error to satisfy

$$R_4(47) < \frac{|-60 \times 7.98 \times 10^{-8}|}{5!} = 3.99 \times 10^{-8},$$

which is within a factor of 2 of the actual absolute error, $|1.6720978814 - 1.672097857935| = 2.35 \times 10^{-8}$.

8.5 Runge's Phenomenon

The analysis of the previous section indicates that we should be careful about the choice of points at which we sample our function, and that if the magnitude of $|f^{(N+1)}(x)|$ grows

Consider *Runge's function*:

$$f(x) = \frac{1}{1+25x^2}, \quad , \tag{8.6}$$

sampled uniformly over the interval $x \in [-1, 1]$.

It turns out that the magnitude of this function's derivatives grow more rapidly than $(N+1)!$ for $|x| \gtrsim 0.7$. As a result, errors grow without bound in this region (left panel of Fig. 8.1) as the degree of the interpolating polynomial is increased.

On the other hand, if Runge's function is sampled at Chebyshev nodes instead of uniformly throughout the interval, then the errors instead drop to zero (right panel of Fig. 8.1) with increased interpolating polynomial degree.

We conclude that for some functions, higher polynomial order may not necessarily result in a more accurate interpolation. This fact is encoded in the definition of the residual function, which depends on the magnitude of high-order derivatives of the function and the choice of sampling points.

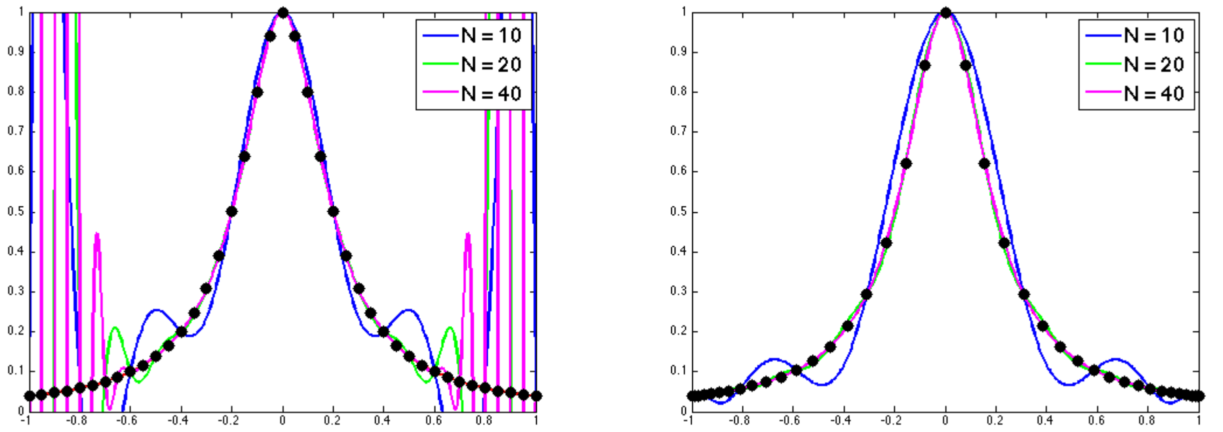

Figure 8.1: *Source:* `http://math.boisestate.edu/~calhoun/teaching/matlab-tutorials/lab_11/html/lab_11.html`. **Left panel**: Runge phenomenon: polynomials of increasing degree fit to Runge's function (Eq. 8.6), using uniform sampling of the function. **Right panel**: Same as left panel, except polynomial samples at Chebyshev nodes only.

MATH 521, Fall 2018 Notes

Prof. Zachariah B. Etienne

Chapter 9: Numerical Integration

Suppose we wish to calculate the definite integral

$$F = \int_a^b f(x)dx,$$

for some function $f(x)$. An exact expression for the integral *will not exist* for all functions $f(x)$, so we are forced to instead approximate the integral by evaluating $f(x)$ at only a finite number of points $x = x_i$. Numerical integration is the topic within numerical analysis whereby we not only devise algorithms for approximating F, but more importantly, provide error estimates for F, based on properties of $f(x)$ and our choice of sampling points x_i..

In this chapter, we will focus entirely on definite integration formulas for which the integrand is sampled uniformly; i.e., with unformly-sampled x_i.

Definition: Uniformly-sampled—also known as **equally-spaced**—x_i requires that $x_i = a + i\Delta x$, where i is an integer and $i \in [0, N]$, and $b = a + N\Delta x$, so $\Delta x = \frac{b-a}{N}$.

9.1 The Trapezoidal Rule

Let us first approximate the integral by drawing a line between $f(a)$ and $f(b)$ and computing the area under that line. That is to say, we wish to compute the area of a *trapezoid*:

- The area under $f(a)$ in the interval is $F_a = (b-a)f(a)$.
- Next we add the triangular area connected by the points $(x, f(x)) = (a, f(a))$, $(b, f(b))$, and $(b, f(a))$ to obtain the area of the right triangle (half the area of a rectangle): $F_\triangle = (b-a)[f(a) - f(b)]/2$
- Summing the two areas we get the area of the resulting trapezoid:

$$F_{\text{Trap}} = F_a + F_\triangle = (b-a)f(a) + (b-a)\frac{f(a)-f(b)}{2} = \frac{f(a)+f(b)}{2}(b-a). \tag{9.1}$$

  Definition: Definite integration by means of fitting a line to the value of the function at the endpoints of the integration interval is called the **Trapezoidal Rule**. I.e., the approximation of F using the Trapezoidal rule is given by Eq. 9.1.

Obviously the trapezoidal rule this is a not an exact expression for the integral of a generic function $f(x)$ from $x = a$ to $x = b$, so we can write it as

$$F = \frac{f(a)+f(b)}{2}(b-a) + E_{\text{Trap}}. \tag{9.2}$$

What is the nature of this error term E_{Trap}? For example, for what functions $f(x)$ would the error term be zero? (Any function that is linear between a and b.) If we know that the underlying function is smooth over our interval, it turns out that we can estimate this error term very accurately, based on the properties of the underlying function.

Next we derive an explicit expression for E_{Trap}.

9.1.1 The Trapezoidal Rule Error Term

Original Source: `http://math.ucsd.edu/~ebender/20B/77_Trap.pdf` WARNING: original source has correct result but presents a flawed argument for computing integration constant B, which actually cancels out, as shown in our derivation below. We get the same function for the Error.

Note that for one interval of the Trapezoidal rule we only evaluate the function at two points (we will later learn about integration rules that involve more points). So in this case, $N = 1$ and $\Delta x = (b - a)$

$$F = \int_a^b f(x)dx = \int_0^{\Delta x} f(x+a)dx.$$

Our first goal is to try to rewrite the integral in the form

$$F = F_{\text{Trap}} + E_{\text{Trap}} = \frac{f(a)+f(b)}{2}(b-a) + E_{\text{Trap}}.$$

To this end, notice that if we rewrite the integral on the right-hand side of the equation using integration by parts, we obtain:

$$\begin{aligned} u &= f(x+a) & dv &= dx \\ \implies u' &= f'(x+a) & v &= x + A \\ \implies F = \int_0^{\Delta x} f(x+a)dx &= [(x+A)f(x+a)]_0^{\Delta x} - \int_0^{\Delta x} (x+A)f'(x+a)dx. \end{aligned}$$

Let's expand the first term on the right-hand side:

$$[(x+A)f(x+a)]_0^{\Delta x} = [(\Delta x + A)f(\Delta x + a)] - [(0+A)f(0+a)] = \frac{f(a)+f(b)}{2}(b-a).$$

Recall that $\Delta x = (b-a)$, so

$$\begin{aligned} [(\Delta x + A)f(\Delta x + a)] - [(0+A)f(0+a)] &= [((b-a)+A)f((b-a)+a)] - Af(a) \\ &= [((b-a)+A)f(b)] - Af(a) \\ &= A(f(b)-f(a)) + (b-a)f(b) \end{aligned}$$

This looks very similar to the Trapezoidal Rule, and in fact we can choose A to make it *identical* to the Trapezoidal Rule! Note that in the Trapezoidal Rule, the coefficient in front of $f(a)$ must be $(b-a)/2$, so $A = (a-b)/2$:

$$\begin{aligned} A(f(b)-f(a)) + (b-a)f(b) &= (a-b)/2(f(b)-f(a)) + (b-a)f(b) \\ &= \frac{f(b)+f(a)}{2}(b-a) = F_{\text{Trap}}. \end{aligned}$$

Thus the expression we have derived can be written

$$F = \int_0^{\Delta x} f(x+a)dx = F_{\text{Trap}} - \int_0^{\Delta x} (x+A)f'(x+a)dx,$$

where again, $A = (a-b)/2$. Thus the error in the Trapezoidal Rule approximation is given by

$$E_{\text{Trap}} = -\int_0^{\Delta x} (x+A)f'(x+a)dx,$$

but what does this mean?

Let's analyze this expression for E_{Trap} more closely by integrating by parts again (expressions that cancel are $\underbrace{\text{underbraced}}$):

$$\begin{aligned}
u = f'(x+a) &\qquad dv = (x+A)dx = (x+A)d(x+A) \\
\implies u' = f''(x+a) &\qquad v = \frac{(x+A)^2}{2} + B
\end{aligned}$$

$$\begin{aligned}
\int_0^{\Delta x} f(x+a)dx &= F_{\text{Trap}} - \left[\left(\frac{(x+A)^2}{2} + B\right) f'(x+a)\right]_0^{\Delta x} + \int_0^{\Delta x} \left(\frac{(x+A)^2}{2} + B\right) f''(x+a)dx \\
&= F_{\text{Trap}} - \left[\frac{(x+A)^2}{2} f'(x+a)\right]_0^{\Delta x} \underbrace{-Bf'(x+a)|_0^{\Delta x}} + \int_0^{\Delta x} \left(\frac{(x+A)^2}{2}\right) f''(x+a)dx + \\
&\quad \underbrace{Bf'(x+a)|_0^{\Delta x}} \quad \text{[Fund. Thm. of Calc.]} \\
&= F_{\text{Trap}} - \left[\frac{(x+A)^2}{2} f'(x+a)\right]_0^{\Delta x} + \int_0^{\Delta x} \left(\frac{(x+A)^2}{2}\right) f''(x+a)dx \\
&= F_{\text{Trap}} - \left[\frac{x^2+2Ax+A^2}{2} f'(x+a)\right]_0^{\Delta x} + \int_0^{\Delta x} \left(\frac{x^2+2Ax+A^2}{2}\right) f''(x+a)dx \\
&= F_{\text{Trap}} - \left[\frac{x^2+2Ax}{2} f'(x+a)\right]_0^{\Delta x} \underbrace{-\frac{A^2}{2} f'(x+a)|_0^{\Delta x}} \\
&\quad + \int_0^{\Delta x} \left(\frac{x^2+2Ax}{2}\right) f''(x+a)dx + \underbrace{\frac{A^2}{2} f'(x+a)|_0^{\Delta x}} \quad \text{[Fund. Thm. of Calc.]} \\
&= F_{\text{Trap}} - \left[\frac{x^2+2Ax}{2} f'(x+a)\right]_0^{\Delta x} + \int_0^{\Delta x} \left(\frac{x^2+2Ax}{2}\right) f''(x+a)dx
\end{aligned}$$

Let's plug this value of $A = -(b-a)/2 = -\Delta x/2$ back into the rest of the expression we derived above:

$$\begin{aligned}
\int_0^{\Delta x} f(x+a)dx &= F_{\text{Trap}} - \left[\frac{x^2 - 2[(b-a)/2]x}{2} f'(x+a)\right]_0^{\Delta x} + \int_0^{\Delta x} \left(\frac{x^2 - 2[(b-a)/2]x}{2}\right) f''(x+a)dx \\
&= F_{\text{Trap}} - \left[\frac{x^2 - x\Delta x}{2} f'(x+a)\right]_0^{\Delta x} + \int_0^{\Delta x} \left(\frac{x^2 - x\Delta x}{2}\right) f''(x+a)dx \\
&= F_{\text{Trap}} - \left[\frac{\Delta x^2 - \Delta x^2}{2} f'(x+a)\right] + [0] + \int_0^{\Delta x} \left(\frac{x^2 - x\Delta x}{2}\right) f''(x+a)dx \\
&= F_{\text{Trap}} + \int_0^{\Delta x} \left(\frac{x^2 - x\Delta x}{2}\right) f''(x+a)dx
\end{aligned}$$

So the error term may be written:

$$E_{\text{Trap}} = \int_0^{\Delta x} \left(\frac{x}{2}(x - \Delta x)\right) f''(x+a)dx.$$

For smooth $f(x)$, it must be true that $|f''(x+a)| < K$ between $x = 0$ and $x = \Delta x$ for some constant K, so a suitable estimate for the magnitude of the error term is

$$\begin{aligned}
|E_{\text{Trap}}| &\lesssim K\left|\int_0^{\Delta x}\left(\frac{x}{2}(x-\Delta x)\right)dx\right| \\
&= \frac{K}{2}\left|\frac{x^3}{3} - \frac{x^2}{2}\Delta x\right|_0^{\Delta x} \\
&= \frac{K}{2}\left|\frac{(\Delta x)^3}{3} - \frac{(\Delta x)^2}{2}\Delta x\right| \\
&= \frac{K}{12}(\Delta x)^3 \\
&= \max\left(|f''_{\text{in interval}}|\right)\frac{(\Delta x)^3}{12}
\end{aligned}$$

This expression tells us that the error in approximating the integrand as a linear function across the integration interval should increase with the magnitude of the second derivative of the integrand over the integration bounds. It also indicates that, all else being fixed, the error should decrease proportionally to $(\Delta x)^3$.

9.1.2 The Extended Trapezoidal Rule

In practice we will use the Trapezoidal rule by dividing up the interval $x \in [a, b]$ over more than just the endpoints. For example, $F = \int_a^b f(x)dx$ can be divided into two intervals as follows:

$$F = \int_a^b f(x)dx = \int_a^{(a+b)/2} f(x)dx + \int_{(a+b)/2}^b f(x)dx$$

Then we apply the Trapezoidal rule to each of the integrals on the RHS. Let's now imagine we've divided the integral into N intervals and apply the Trapezoidal rule to each interval. Defining $x_N = b$ and $x_0 = a$, we can write:

$$\begin{aligned}
F = \int_a^b f(x)dx &\approx \frac{f(x_N)+f(x_{N-1})}{2}(x_N - x_{N-1}) + \frac{f(x_{N-1})+f(x_{N-2})}{2}(x_{N-1}-x_{N-2}) + ... \\
&\quad + \frac{f(x_2)+f(x_1)}{2}(x_2 - x_1) + \frac{f(x_1)+f(x_0)}{2}(x_1 - x_0) \\
&= f(x_N)\left(\frac{x_N - x_{N-1}}{2}\right) + f(x_{N-1})\left(\frac{x_N - x_{N-2}}{2}\right) + ... + f(x_1)\left(\frac{x_2 - x_0}{2}\right) + f(x_0)\left(\frac{x_1 - x_0}{2}\right)
\end{aligned}$$

Thus if the function is sampled evenly at points x_i, we can quickly rewrite this expression in terms of $x_i - x_{i-1} = \Delta x$:

$$F = \int_a^b f(x)dx \approx \Delta x\left(\frac{f(x_0)}{2} + f(x_1) + f(x_2) + ... + f(x_{N-2}) + f(x_{N-1}) + \frac{f(x_N)}{2}\right)$$

This is known as the **Extended Trapezoidal Rule**.

Based on our derivation of the error in the Trapezoidal Rule, we may define the error in each individual interval as

$$E_{\text{interval}} = ||f''_{\text{in single interval}}||\frac{(\Delta x)^3}{12},$$

where $||f''_{\text{in single interval}}||$ is a representative magnitude of the second derivative of the integrand over a single interval, we can write the error for the whole integral, divided up into N intervals as

$$E_{\text{tot,Trap}} \lesssim N\max\left(||f''_{\text{in single interval}}||\right)\frac{(\Delta x)^3}{12}.$$

Note that $\Delta x = (b-a)/N$, so we get

$$E_{\text{tot,Trap}} \lesssim \max\left(||f''_{\text{in single interval}}||\right) \frac{(b-a)^3}{12N^2}$$

for the Extended Trapezoidal Rule.

This means that as we increase the number of intervals, the error drops in proportion to $1/N^2$, or equivalently, in proportion to $N(\Delta x)^3$. We must be careful when estimating the error this way, as $|f''|$ will in general vary from interval to interval. However, in practice we will find that the f'' term will quickly asymptote to nearly a constant value as we increase N, meaning that $E_{\text{tot,Trap}}$ will, to good approximation, be proportional to $1/N^2$.

Exercise #1:

Evaluate the exact value of the integral

$$\int_0^1 \ln(1+y)dy,$$

using variable substitution and integration by parts. Using a calculator, evaluate the final expression you obtain to 15 significant digits.

Solution to Exercise #1

Let $x = 1+y$. Then this integral is equal to $\int_1^2 \ln(x)dx$.
Integration by parts is derived immediately from the Product Rule:

$$\begin{aligned}
\text{Product rule} : (u(x)v(x))' &= u'(x)v(x) + u(x)v'(x) \\
\Longrightarrow \int_a^b u(x)v'(x)dx &= \int_a^b (u(x)v(x))'dx - \int_a^b u'(x)v(x)dx \\
\Longrightarrow \int_a^b u(x)v'(x)dx &= [u(x)v(x)]_a^b - \int_a^b u'(x)v(x)dx
\end{aligned}$$

Choosing

$$\begin{aligned}
u(x) = \ln(x) &\qquad v'(x)dx = dx \\
u'(x) = 1/(x) &\qquad v(x) = x,
\end{aligned}$$

we get

$$\begin{aligned}
\int_1^2 \ln(x)dx &= [x\ln x]_1^2 - \int_1^2 (x)/(x)dx \\
&= [x\ln x]_1^2 - (2-1) = 2\ln(2) - 1\ln(1) - 1 = 0.386294361119890...
\end{aligned}$$

Exercise #2:

Compute the integral

$$\int_0^1 \ln(1+y)dy$$

using the Trapezoidal rule with 1, 2, and 4 evenly-spaced intervals. Compare the result with the exact solution computed in the previous exercise. Use $\ln(ab) = \ln a + \ln b$ to rewrite all logarithms as logarithms of integers.

Solution to Exercise #2

We start by evaluating the integral using the Trapezoidal rule with one interval:

$$\begin{aligned}\int_a^b f(y)dy &\approx \frac{f(a)+f(b)}{2}(b-a)\\ &= \frac{\ln(1)+\ln(1+1)}{2}(2-1)\\ &= \ln(2)/2 = 0.34657359027997...\end{aligned}$$

Two intervals now:

$$\int_a^b f(y)dy = \int_a^c f(y)dy + \int_c^b f(y)dy$$

$$\begin{aligned}&\approx \frac{f(a)+f(c)}{2}(c-a) + \frac{f(c)+f(b)}{2}(b-c)\\ &= \frac{\ln(1)+\ln(3/2)}{2}(3/2-1) + \frac{\ln(3/2)+\ln(2)}{2}(2-3/2)\\ &= \frac{\ln(3)-\ln(2)}{4} + \frac{\ln(3)-\ln(2)+\ln(2)}{4}\\ &= \frac{\ln(3)-\ln(2)}{2} + \frac{\ln(2)}{4} = 0.37601934919407...\end{aligned}$$

Four intervals now: Evaluate $f(y)$ at $y = 1, 1.25, 1.5, 1.75, 2$

$$\begin{aligned}\int_a^b f(y)dy &= \int_a^c f(y)dy + \int_c^d f(y)dy + \int_d^e f(y)dy + \int_e^b f(y)dy\\ \int_1^2 f(y)dy &= \int_1^{(5/4)} f(y)dy + \int_{(5/4)}^{(3/2)} f(y)dy + \int_{(3/2)}^{(7/4)} f(y)dy + \int_{(7/4)}^2 f(y)dy\\ &\approx \frac{\ln(1)+\ln(5/4)}{2}(1/4) + \frac{\ln(5/4)+\ln(3/2)}{2}(1/4)\\ &\quad + \frac{\ln(3/2)+\ln(7/4)}{2}(1/4) + \frac{\ln(7/4)+\ln(2)}{2}(1/4)\\ &= \frac{\ln(5)-\ln(4)}{8} + \frac{\ln(5)-\ln(4)+\ln(3)-\ln(2)}{8}\\ &\quad + \frac{\ln(3)-\ln(2)+\ln(7)-\ln(4)}{8} + \frac{\ln(7)-\ln(4)+\ln(2)}{8}\\ &= \frac{\ln(5)-\ln(4)}{4} + \frac{\ln(3)-\ln(2)}{4} + \frac{\ln(7)-\ln(4)}{4} + \frac{\ln(2)}{8}\\ &= \frac{\ln(5)-2\ln(2)}{4} + \frac{\ln(3)-\ln(2)}{4} + \frac{\ln(7)-2\ln(2)}{4} + \frac{\ln(2)}{8} = 0.38369950940944...\end{aligned}$$

Let's now analyze results from the previous Exercises in tabular form:

N	Result	Error	Error Improvement factor Relative to $N = 1$
1	0.34657359027997	10.3%	1
2	0.37601934919407	2.66%	3.9x
4	0.38369950940944	0.672%	15.3x
Exact	0.38629436111989	-	-

Thus we have shown that as N increases, the error drops nearly proportionally to $1/N^2$. Why is the proportionality not exact?

9.2 Simpson's Rule

Our goal in this chapter is to compute

$$F = \int_a^b f(x)dx$$

numerically. We have already examined the case in which the integral is split into many intervals, the function approximated in each by a line connected by the endpoints of each interval.

Definition: **Simpson's rule** samples $f(x)$ at points a, $(a+b)/2$, and b, fits the function evaluated at these three points to a parabola, and then approximates the integral F as the area under this parabola. Simpson's rule F_{Simp} can be written as:

$$\begin{aligned} F = \int_a^b f(x)dx &= F_{\text{Simp}} + E_{\text{Simp}} \\ &= (b-a)\left[\frac{1}{6}f(a) + \frac{2}{3}f\left(\frac{a+b}{2}\right) + \frac{1}{6}f(b)\right] + E_{\text{Simp}} \end{aligned}$$

where E_{Simp} is the error term. In this case it can be shown that

$$E_{\text{Simp}} \lesssim \max\left(||f^{(4)}_{\text{in single interval}}||\right)\frac{(\Delta x)^5}{90}.$$

In Simpson's rule, we evaluate the function at three points in the interval from a to b: at a, b, and the midpoint $(a+b)/2$. So Δx, the spacing between function evaluations, is simply equal to $(b-a)/2$. Thus Simpson's rule may be written as:

$$F = \int_a^b f(x)dx = \frac{\Delta x}{3}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] + E_{\text{Simp}}$$

Just as we did with the Trapezoidal rule, we can break up the integral

$$\int_a^b f(x)dx = \int_a^{(a+b)/2} f(x)dx + \int_{(a+b)/2}^b f(x)dx$$

into two equally-spaced pieces and apply the Simpson's rule to each piece. Here we have divided the integral into two integrals.

For a single interval, we must evaluate the function at 3 points using Simpson's rule. For two intervals, this increases to 5 (notice that we do not double-count the point at $x = (a+b)/2$). Continuing the pattern, for N Simpson's rule intervals, we will have a total of $(2N+1)$ function evaluations, at $x = x_0$,$x = x_1 = x_0 + \Delta x$,...$x = x_N$.

Let's use this form to now divide the integral up over an arbitrary number of intervals, giving us:

$$\begin{aligned} \int_a^b f(x)dx &= \frac{\Delta x}{3}[(f(x_0) + 4f(x_1) + f(x_2)) + (f(x_2) + 4f(x_3) + f(x_4)) + (f(x_4) + 4f(x_5) + f(x_6)) + ...] + NE_{\text{Simp}} \\ &= \frac{\Delta x}{3}(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + ...) + NE_{\text{Simp}} \end{aligned}$$

Notice the alternating pattern.

So long as we have minimally sampled the integrand $f(x)$ (i.e., we are not *under-samping* features of the function), the error over a single interval E_{Simp} will drop approximately in proportion to $(\Delta x)^5$. I.e., $E_{\text{Simp}} \overset{\propto}{\sim} (\Delta x)^5$. So the overall error in our integral when divided over N intervals using Simpson's rule, will satisfy $E_{\text{tot,Simp}} \overset{\propto}{\sim} (\Delta x)^5 \times N$. Since the distance between sampling points for N intervals of Simpson's rule is given by $\Delta x = (b-a)/(2N)$ (or equivalently $2N+1$ function evaluations), we find that $E_{\text{tot,Simp}} \overset{\propto}{\sim} 1/N^4$ for the extended Simpson's rule.

9.3 Code Validation Checks

If we are applying the Trapezoidal rule to a function $f(x)$ for which we have chosen a sufficiently large value of intervals N to avoid under-sampling $f(x)$, then the weighted average of $||f''(x)||$ over an interval can be approximated by some constant K and the exact value of the integral F at a given value of N is

$$F = F_a + K/N^2,$$

where F_a is the *approximate* value of the integral using the Extended Trapezoidal Rule with N intervals. Note that since $\Delta x = (b-a)/N$, we can rewrite this expression as

$$F = F_a + K'(\Delta x)^2,$$

where $K' = K/(b-a)^2$.

So after evaluating the integral, F_a and Δx are known, but K' and F are unknown. Thus we can solve for F by computing values of F_a with two values of Δx.

Suppose you have programmed an Extended Trapezoidal Rule numerical integration routine. You know that for smooth functions with any "bumps" resolved at least at the Nyquist sampling rate, your integral should converge to the exact value, approximately proportionally to $(\Delta x)^2$.

When solving mathematical problems on the computer, it is always our task to perform *code validation* tests, to demonstrate to ourselves and our colleagues that our codes do not have bugs, and in fact we have programmed the algorithms correctly. Thus it is your solemn duty to verify that indeed your error is very nearly proportional to $(\Delta x)^2$. If our code satisfies this test, we can say that "our code converged to second order in the interval spacing (Δx)".

After programming a routine that must converge to second order, we are obligated to solve

$$F = F_a + K'(\Delta x)^n$$

for unknowns F, K', and n. Remember that this equality is only approximate, as it assumes higher-order errors are negligible, and that truncation error dominates. To solve for these three unknowns, we must perform three numerical integrations to get known values F_a at three known Δx values. Let's solve for these three, using resolutions Δx_1, Δx_2, and Δx_3. Suppose also that, as in our example, $\Delta x_2 = \Delta x_1/2$ and $\Delta x_3 = \Delta x_1/4$:

$$\begin{aligned} F &= F_{a,1} + K'(\Delta x_1)^n && (9.3)\\ F &= F_{a,2} + K'(\Delta x_1/2)^n && (9.4)\\ F &= F_{a,3} + K'(\Delta x_1/4)^n && (9.5) \end{aligned}$$

First let's solve for n. Subtract Eq. 9.4 from Eq. 9.3:

$$\begin{aligned} 0 &= F_{a,1} - F_{a,2} + K'[(\Delta x_1)^n - (\Delta x_1/2)^n]\\ \Longrightarrow F_{a,1} - F_{a,2} &= -K'\Delta x_1^n[1-(1/2)^n] \end{aligned}$$

Similarly, subtracting Eq. 9.5 from Eq. 9.4:

$$\begin{aligned} 0 &= F_{a,2} - F_{a,3} + K'[(\Delta x_1/2)^n - (\Delta x_1/4)^n]\\ \Longrightarrow F_{a,2} - F_{a,3} &= -K'\Delta x_1^n[(1/2)^n - (1/4)^n]. \end{aligned}$$

Combining these expressions, we find

$$\begin{aligned} \frac{F_{a,1} - F_{a,2}}{F_{a,2} - F_{a,3}} &= \frac{1-(1/2)^n}{(1/2)^n - (1/4)^n}\\ &= \frac{1-(1/2)^n}{(1/2)^n(1-(1/2)^n)} = 2^n. \end{aligned}$$

So if we have correctly programmed a second-order method, then $n = 2$ and we should find that

$$\frac{F_{a,1} - F_{a,2}}{F_{a,2} - F_{a,3}} = 2^2 = 4$$

We will use this value as a check on the consistency of our method. If this ratio is found to be far from 4, there are *at least* three possible reasons:

1. There may be a bug in our code,
2. We are under-sampling the function (aliasing error), or
3. We chose a Δx so tiny that, e.g., $F_{a,1} - F_{a,2}$ results in a catastrophic cancellation (roundoff error/loss of significance).

All of these reasons can be easily checked via careful analysis of the code and a back-of-the-envelope estimate.

9.4 Richardson Extrapolation

Recall in our Extended Trapezoidal Rule example, as we increased N, the value we computed for the integral monotonically approached the exact value of the integral. *Richardson extrapolation* is the strategy of using known values of the integral with finite N and nonzero Δx and extrapolating the results to compute an estimate for the integral in the limits $N \to \infty$, or equivalently $\Delta x \to 0$. But how do we do this? In particular, how do we do this if we do not know the exact value?

Let's use the technique of the previous section to assume that our method is consistent with $n = 2$ and proceed to solve for F, the "exact" value of the integral:

Eq 1 gives:

$$K' = \frac{F - F_{a,1}}{(\Delta x_1)^2}$$

Plugging this into Eq 2 we get:

$$\begin{aligned}
F &= F_{a,2} + (F - F_{a,1})\frac{(\Delta x_1/2)^2}{(\Delta x_1)^2} \\
&= F_{a,2} + (F - F_{a,1})/4 \\
\implies \frac{3}{4}F &= F_{a,2} - F_{a,1}/4 \\
\implies F &= \frac{1}{3}(4F_{a,2} - F_{a,1})
\end{aligned}$$

Let's now compute this value for F in our example problem:

$$\begin{aligned}
F &= \frac{1}{3}(4F_{a,2} - F_{a,1}) \\
&= \frac{1}{3}(4 \times 0.37601934919407 - 0.34657359027997) \\
&= 0.38583460216544,
\end{aligned}$$

which is 0.119% from the exact value! Recall that the value we obtained using the Trapezoidal rule to evaluate the function at 5 points (4 intervals) had an error of 0.672%. Thus the value obtained from Richardson extrapolation of a total of three function evaluations, is significantly closer to the exact value than using the Trapezoidal rule to evaluate the function at five points!

This is because

$$\begin{aligned}
\frac{1}{3}(4F_{a,2} - F_{a,1}) &= \frac{1}{3}\left[4\frac{\Delta x}{2}\left(\frac{f(a)}{2} + f(c) + \frac{f(b)}{2}\right) - \Delta x\frac{f(a) + f(b)}{2}\right] \\
&= \frac{\Delta x}{3}\left[\frac{f(a)}{2} + 2f(c) + \frac{f(b)}{2}\right]
\end{aligned}$$

Which is exactly Simpson's rule! So by performing this Richardson extrapolation, we have effectively increased the order of our integration from $1/N^2$ to $1/N^4$. But remember that we assumed the exact value of the integral satisfies

$$F = F_{a,i} + K'(\Delta x_i)^n$$

So why did we not obtain the exact value of the integral? Recall from our derivation of the error in the Trapezoidal Rule, K' is not a constant; instead it is equal to an integral whose integrand contains the second derivative of the function over the integration interval. K' in a given interval will in general depend on f'' and Δx. What we've found here is that effectively we can rewrite K' as

$$K' = \tilde{K} + \hat{K}(\Delta x)^2,$$

so that we can rewrite (in the case of the Trapezoidal Rule) F as

$$\begin{aligned} F &= F_a + K'(\Delta x)^2 \\ &= F_a + \tilde{K}(\Delta x)^2 + \hat{K}(\Delta x)^4. \end{aligned}$$

A Richardson extrapolation effectively solves for $\tilde{K}$, giving us the exact value of F to error proportional to $(\Delta x)^4$

9.5 Romberg Integration

Richardson extrapolation can be used to obtain even more accurate values for the integral as the number of intervals is increased. The general algorithm is called **Romberg integration**, which is as follows.

Define

$$R(n = 0, m = 0) = \frac{1}{2}(b - a)(f(a) + f(b)).$$

Notice that $R(0, 0)$ is simply the expression for the integral as approximated by the Trapezoidal Rule. Suppose we divide the integral up into two equal intervals *a la*

$$\int_a^b f(x)dx = \int_a^{(a+b)/2} f(x)dx + \int_{(a+b)/2}^b f(x)dx.$$

Now repeat this n times. Then

$$(\Delta x)_n = \frac{b - a}{2^n}.$$

So $n > 0$ for the *Extended* Trapezoidal Rule (i.e., the Trapezoidal Rule such that the original integral is broken into multiple, equally-spaced and non-overlapping integrals), which may be written:

$$R(n, m = 0) = \frac{1}{2}R(n - 1, 0) + (\Delta x)_n \sum_{k=1}^{2^{n-1}} f(a + (2k - 1)(\Delta x)_n).$$

Defining N_f to be the number of function evaluations for the Extended Trapezoidal Rule, if we only evaluate the function at the endpoints a and b, then $N_f = 2$ and

$$\Delta x = (b - a).$$

Similarly, if we evaluate the function at the endpoints and the midpoint, then $N_f = 3$ and

$$\Delta x = (b - a)/2.$$

To establish the pattern, if we evaluate the function at the endpoints, $(b - a)/4$, $(b - a)/2$, and $3(b - a)/4$, then $N_f = 5$ and

$$\Delta x = (b - a)/4.$$

Clearly the pattern is that when we evaluate the function at N_f evenly-spaced sampling points Δx,

$$\Delta x = \frac{b - a}{N_f - 1}. \tag{9.6}$$

Although this discussion focuses on the $m = 0$ case, the distance between uniform sampling points Δx will satisfy Eq. 9.6 *regardless of the integration rule given by m.*

Recall that our numerical integration algorithms rely on first fitting a polynomial to the integrand $f(x)$. In Romberg integration, the degree of the polynomial is given by $m + 1$.

Therefore, m is related to the convergence order of the method as the number of intervals and function evaluations increases. Recall that the extended trapezoidal rule (corresponding to linear polynomial fits to the function at sample points x_i) possessed an error approximately proportional to $(\Delta x)^2$ and the extended Simpson's rule (corresponding to quadratic polynomial fits to the function at sample points x_i) $(\Delta x)^4$. Extending this to arbitrary order polynomial rules, the error of the Romberg integral approximation, $E_{\text{Romb}}(n, m)$, is approximately proportional to $(\Delta x_n)^{2m+2}$, provided that the integrand $f(x)$ is at least minimally (i.e., Nyquist) resolved.

Thus in the case of Romberg integration, the error $E_{\text{Romb}}(n, m)$ satisfies the approximate proportionality

$$E_{\text{Romb}}(n, m) \overset{\propto}{\sim} (\Delta x_n)^{2m+2} \propto 1/(N_f - 1)^{2m+2}.$$

Notice that $m = 0$ for the Trapezoidal Rule, and the number of trapezoidal intervals N is equivalent to $N_f - 1$, the error indeed is approximately proportional to $1/N^2$.

Exercise #3:

How are the number of intervals N related to N_f for the Extended Simpson's Rule?

Solution to Exercise #3

$N = 1$ corresponds to $N_f = 3$, $N = 2$ corresponds to $N_f = 5$, $N = 3$ corresponds to $N_f = 7$, so $N = x$ corresponds to $N_f = (2x + 1)$. Thus $N_f - 1 = 2N$ for the Extended Simpson's Rule.

As proven in Exercise #3, for the Extended Simpson's Rule $N_f - 1 = 2N$, so we find that

$$E_{\text{Simp,tot}} \overset{\propto}{\sim} (\Delta x_n)^{2m+2} \propto 1/(N_f - 1)^{2m+2} \propto 1/N^{2m+2} = 1/N^4,$$

consistent with what we found previously.

We wish to get the best, Richardson-extrapolated estimate. To complete our definition of Romberg integration, we simply combine the above definitions for $R(0, 0)$, $R(n, 0)$, with the formula:

$$R(n, m) = R(n, m - 1) + \frac{1}{4^m - 1}(R(n, m - 1) - R(n - 1, m - 1)),$$

where $n \geq m$. *Note that this can be rewritten in an equivalent form:*

$$R(n, m) = \frac{1}{4^m - 1}\left(4^m R(n, m - 1) - R(n - 1, m - 1)\right).$$

Exercise to reader: Prove this.

Exercise #4:

To what integration rule does $R(n, m) = R(1, 0)$ correspond? Apply the Romberg integration formula to write the full integration rule.

Solution to Exercise #4

$R(n,m) = R(1,0)$ corresponds to the Extended Trapezoidal Rule for three points:

$$\begin{aligned}
R(n=1,m=0) &= \frac{1}{2}R(0,0) + (\Delta x)_1 \sum_{k=1}^{2^{1-1}} f(a + (2k-1)(\Delta x)_1) \\
&= \frac{1}{4}(b-a)(f(a)+f(b)) + \frac{b-a}{2} f(a + (b-a)/2) \\
&= \frac{1}{4}(b-a)(f(a)+f(b)) + \frac{b-a}{2} f\left(\frac{a+b}{2}\right) \\
&= \frac{b-a}{2}\left[\frac{f(a)}{2} + f\left(\frac{a+b}{2}\right) + \frac{f(b)}{2}\right].
\end{aligned}$$

Exercise #5:

To what integration rule does $R(n,m) = R(1,1)$ correspond? Apply the Romberg integration formula to write the full integration rule.

Solution to Exercise #5

$R(1,1)$ is Simpson's Rule:

$$\begin{aligned}
R(1,1) &= R(1,0) + \frac{1}{4^1 - 1}(R(1,0) - R(0,0)) \\
&= \frac{b-a}{2}\left(\frac{f(a)}{2} + f\left(\frac{a+b}{2}\right) + \frac{f(b)}{2}\right) \\
&\quad + \frac{1}{3}\left[\frac{b-a}{2}\left(\frac{f(a)}{2} + f\left(\frac{a+b}{2}\right) + \frac{f(b)}{2}\right) - \frac{1}{2}(b-a)(f(a)+f(b))\right] \\
&= \frac{b-a}{2}\left[(f(a)+f(b))\left(\frac{1}{2} + \frac{1}{6} - \frac{1}{3}\right) + f\left(\frac{a+b}{2}\right)\left(1 + \frac{1}{3}\right)\right] \\
&= \frac{b-a}{2}\frac{1}{3}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]
\end{aligned}$$

We conclude that $R(n,0)$ is the Extended Trapezoidal Rule (for $n > 0$), $R(n,1)$ is the Extended Simpson's Rule (for $n > 1$), and $R(2,2)$ is a $1/N^6$-convergent approximation called Boole's Rule. Notice that each step in this sequence yields the highest-degree polynomial fit to the integrand.

MATH 521, Fall 2018 Notes

Prof. Zachariah B. Etienne

Chapter 10: Iterative Techniques, Numerical Root Finding

Iterative techniques in numerical analysis cover a large class of approaches for solving mathematical problems, which can be generally described with the formula

$$\boldsymbol{x}_{i+1} = \boldsymbol{g}(\boldsymbol{x}_i),$$

where $\boldsymbol{x}_i$ is the solution vector (i.e., the solution to our mathematical problem) at iteration i, and $\boldsymbol{g}$ is some function that reads the solution vector at iteration i and outputs the solution vector at iteration $i+1$. To implement iterative techniques, we first define $\boldsymbol{g}(\boldsymbol{x}_i)$, so that once an initial guess $\boldsymbol{x}_0$ for the solution is made, the solution at iteration $i = 1$ is given by evaluating $\boldsymbol{g}(\boldsymbol{x}_0)$. In the same way, the solution at iteration $i = 2$ is obtained by evaluating $\boldsymbol{g}(\boldsymbol{x}_1)$. The process repeats the same way for $i = 3$ and so forth until some algorithmic stopping condition (i.e., the condition under which we wish to no longer evaluate $\boldsymbol{x}_{i+1}$) is satisfied.

Sometimes the solution vector at finite i represents an approximate solution, where the exact solution will only be obtained in the $i \to \infty$ limit. This is generally the case for Numerical Root Finding—the topic of this chapter. In this case the algorithmic stopping condition might be the iteration at which $\boldsymbol{x}_{j+1}$ is the same as $\boldsymbol{x}_j$ to some prescribed number of significant digits.

Alternatively, $\boldsymbol{x}_i$ might represent the solution at a different point in space or time–for example solutions to ordinary differential equations can be written as an iterative scheme where $\boldsymbol{x}_i$ might represent the approximate solution at some time i. In such a case, the algorithmic stopping condition will be an iteration corresponding to some final time t_f beyond which we are no longer interested in evaluating the solution.

The **Root Finding** problem: Solve

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$$

for vector function $\boldsymbol{f}$ and vector $\boldsymbol{x}$, such that given a component of the vector x_i, there are N unknown components x_i and a total of N independent functions f_i's that relate the x_i's to one another (i.e., N independent equations of N unknowns).

While the familiar equation $\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b} = \boldsymbol{0}$ certainly fits this pattern, with $\boldsymbol{A}$ an $N \times N$ matrix and $\boldsymbol{b}$ a vector comprised of constants independent of x_i's, this is only one particular example of the overall pattern $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$. Namely, $\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b} = \boldsymbol{0}$ only holds for functions $\boldsymbol{f}(\boldsymbol{x})$ that consist of *linear combinations* of the x_i's. Thus Numerical Root Finding can be interpreted as a strategy for solving a wider class of equations beyond those we studied in the previous section on numerical linear algebra (i.e., solving $\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b} = 0$ for $\boldsymbol{x}$), including those sets of equations in which the f_i's depend *nonlinearly* on the x_i's.

There may be many solutions $\boldsymbol{x}$ to the set of equations $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$, and these solutions are called **roots**; our goal is to find them numerically. There are a variety of algorithms for finding roots of vector functions of multiple variables, but they generally build upon and extend strategies for finding roots of functions of a single variable $f(x) = 0$.

So we will focus first on a set of algorithms that solve $f(x) = 0$, where f is a function of a single variable x.

Exercise #1:

Consider

$$f(x) = 6x^6 - 4x^2 - 1.$$

How many roots will this function have and why?

Solution to Exercise #1

The fundamental theorem of algebra states that an Nth-degree polynomial will have N roots, where the roots are possibly a mixture of real and imaginary, and some may be repeated roots. Thus this sixth-order polynomial will have a total of 6 roots

In this section, we will generally focus on functions that are continuous everywhere, because if a function does not satisfy this criterion, the root may be very difficult or impossible to find. Consider for example the limiting case of a function $f(x)$ that maps x to a random number $R \in$`[-1e15:1e15]`. In this case, *no* strategy will be superior than a random search for roots along x, with only an infinitesimal chance of ever finding a true zero.

For continuous functions, we can do much better than a random search, and in general, the smoother the function, the faster we can converge on a root.

10.1 Bisection Method

The Bisection Method is a so-called "bracketing" method, in that you must first feed it two values of x, called, say a and b such that $a < b$, $(f(a)f(b)) < 0$. If the function is continuous, we are then guaranteed that there exists an $r \in (a, b)$ such that $f(r) = 0$.

Bisection finds the root by first evaluating $f(x)$ at $x_0 = (a+b)/2$. For example, if $f(x_0) < 0$ and $(f(a)f(x_0)) < 0$, then we know that $r \in (a, x_0)$, so next we evaluate $f(x)$ at $x_1 = (a + x_0)/2$. If for example, $f(x_1) > 0$ then we know $r \in (x_1, x_0)$. We continue this process until we have found an x that is within roundoff error of $f(x) = 0$. Note that roundoff error and machine epsilon are in general two different quantities! Roundoff error appears due to the growth of rounding in floating point operations. For example, if when evaluating $f(x)$, we have a catastrophic cancellation, then due to *roundoff error*, $f(x)$ might hover around `1e-8` without ever hitting zero. To avoid this, in the pseudocode below, we set `NMAX` to limit the number of iterations over which we bisect the interval.

The pseudocode for the Method of Bisection is as follows

Pseudocode for Bisection Method:

```
1  x1=a
2  x2=b
3  y1=f(x1)
4  y2=f(x2)
5  if(y1*y2)>=0) then
6    print "Bracketing error!"
7    exit program
8  end if
9  do i=1,NMAX
10   x_midpoint = (x1+x2)/2
11   y_midpoint = f( x_midpoint )
12   if(y_midpoint*y1 < 0) then
13     x2=x_midpoint
14     y2=y_midpoint
15   else
16     x1=x_midpoint
17     y1=y_midpoint
18   end if
19   if( | x2-x1 | < epsilon*(|x1|+|x2|*0.5) or y_midpoint == 0 ) then
20     exit loop
21   end if
22   if( i == NMAX ) then
23     print "Warning: Hit i=NMAX iterations. Result may not be converged!"
24   end if
25 end do
26 # The root lies somewhere between x1 and x2.
27 root=x_midpoint  # Let's choose x_midpoint. This way, if y_midpoint=0,
28                  # then we choose the exact location of the root.
```

Notice that in line 19, `epsilon` is defined so that the error in the root (variable `root`) is some small *percentage* of the actual root. This is most apparent if we rewrite line 19 as

$$\frac{|x_2 - x_1|}{(|x_2| + |x_1|)/2} < E_{\text{rel}} = \epsilon.$$

In this way, we see that `epsilon` is a measure of the *relative error*.

How fast does the Bisection Method converge to the actual root? In other words, after how many iterations over `i` can we expect to obtain the root?

We start with two points a and b such that the root r is guaranteed to exist within the interval $r \in (a, b)$, and we know that at each step, we are dividing the interval in two. Thus when we go from iteration i to iteration $i+1$, the maximum distance to the root from either endpoint of our bracket will at worst nearly halve. Defining the maximum distance to the root at iteration i to be ϵ_i, in the worst-case scenario we will find that:

$$\epsilon_{i+1} \approx \frac{\epsilon_i}{2}.$$

As an example, one possible worst-case (slowest-converging) scenario for the bisection method will be the case in which the root is at $r = a + \delta$, where $\delta > 0$ is vanishingly small.

In this worst-case scenario, $\epsilon_0 = (b-a) - \delta \approx (b-a)$ will be our initial maximum distance to the root. Then to find the root to a distance ϵ or less, how many iterations are needed? Clearly from the above expression $\epsilon_{i+1} \approx \frac{\epsilon_i}{2}$, we get $\epsilon_i \approx \frac{\epsilon_0}{2^i}$, and we want this to be less than or equal to ϵ:

$$\begin{aligned}\epsilon_i \approx \frac{\epsilon_0}{2^n} &\lesssim \epsilon \\ \implies \frac{\epsilon_0}{\epsilon} &\lesssim 2^n \\ \implies \log_2 \frac{\epsilon_0}{\epsilon} &\lesssim n.\end{aligned}$$

In the worst-case scenario $r = a + \delta$, $\epsilon_0 \approx (b-a)$, so in this case,

$$\log_2 \frac{(b-a)}{\epsilon} \lesssim i.$$

For bisection, the maximum distance to the root at iteration $i+1$ is given by $\epsilon_{i+1} \approx \frac{\epsilon_i}{2}$, which satisfies the pattern

$$\epsilon_{i+1} \approx \text{constant} \times (\epsilon_i)^m,$$

where $m = 1$. In cases for which $m = 1$, we say that the method converges *linearly.* This notation may seen a bit strange, since the distance to the root (and thus the error) drops by a factor of 2 at each iteration, but we will find that there exist root-finding methods that converge *quadratically*; i.e., with $m = 2$.

Other root finding methods that converge with $m > 1$ are generally less robust; i.e., more prone to convergence problems. When such problems arise, this bisection approach is often adopted as a backup algorithm. Thus the strength of bisection is its robustness; it only requires that the endpoints bracket the root and the function be continuous. If these conditions are met, then bisection will be guaranteed to find the root. The drawback of using bisection is its slowness to converge on the root, due to the fact that it converges linearly.

10.2 Secant Method

As our first example of a root-finding method with $m > 1$ convergence, let's explore the Secant Method, with convergence $m \approx 1.618$. As we will find, the Secant Method requires that the underlying function $f(x)$ be *smooth* (twice differentiable) in order to achieve convergence, and even if it is smooth, there is no guarantee of convergence.

Unlike the Bisection Method, the Secant Method *does not* require that we "bracket" a root. What follows is a possible implementation of the Secant Method algorithm:

1. Pick two arbitrary points $x = a$ and $x = b$ such that $a \neq b$ and $f(a) \neq f(b)$. Define $x_1 = a$ if $|f(a)| < |f(b)|$. Otherwise define $x_1 = b$.
2. Draw the line that connects $f(a)$ and $f(b)$. (The line that passes through two points on a curve is known as the *secant line.*) So long as $f(a) \neq f(b)$, this line is guaranteed to pass through zero, and where it passes through zero is our next guess x_2. Set `i=1`.
3. Draw the line connecting $f(x_i)$ and $f(x_{i+1})$. Define where this line crosses zero as $x = x_{i+2}$.
4. If $|x_{i+2} - x_{i+1}| \leq \epsilon(|x_{i+2}| + |x_{i+1}|)/2$, where ϵ is the maximum relative error we are willing to tolerate, then **exit the algorithm**. Otherwise set `i=i+1` and **Go to 3.**

The top plot in Fig. 10.1 illustrates the Secant Method.

Note that if x_i and x_{i+1} occur such that $f(x_{i+1}) \approx f(x_i)$ (i.e., the slope of the Secant Method line approaches zero), then the next step may result in the next guess for the root being very, very far away. It is therefore possible for this root finder to diverge to $x \to \pm\infty$ even for smooth functions. We must also be careful for situations where $f(x_i)$ and $f(x_{i-1})$ differ by many orders of magnitude; in this case catastrophic cancellation can occur. Generally, we will only observe $m \approx 1.618$ convergence in the neighborhood $x \in (a, b)$ where $f(x)$ is approximately linear.

10.2.1 Derivation of Secant Method

Based on the above algorithm, given $f(x)$ at two points x_i and x_{i-1}, x_{i+1} is the place where the line connecting $f(x_i)$ and $f(x_{i-1}$ crosses zero. Call the equation for this line $\ell(x)$. Then

$$\begin{aligned} \ell(x_i) &= f(x_i) = mx_i + b \\ \ell(x_{i-1}) &= f(x_{i-1}) = mx_{i-1} + b \\ \implies \ell(x_{i+1}) = 0 &= mx_{i+1} + b \implies x_{i+1} = -\frac{b}{m}. \end{aligned}$$

The slope of the line is trivial:

$$m = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}},$$

and the y-intercept can be immediately derived from the top equation above:

$$f(x_i) = mx_i + b \implies b = f(x_i) - mx_i.$$

Then the Secant Method is given by

$$x_{i+1} = -\frac{b}{m} = -\frac{f(x_i) - mx_i}{m} = x_i - \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} f(x_i)$$

10.2.2 Derivation of Secant Method Error

Our goal is to find m such that for the Secant Method, the error at iteration $i + 1$, ϵ_{i+1}, satisfies

$$\epsilon_{i+1} \approx \text{constant} \times (\epsilon_i)^m.$$

We derived above the Secant Method is given by

$$x_{i+1} = x_i - \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} f(x_i).$$

If we are near a root r, then we can define the error ϵ_i in our estimate x_i via

$$\epsilon_i = x_i - r.$$

This implies that

$$\begin{aligned} \epsilon_{i+1} &= x_{i+1} - r \\ &= \left(x_i - \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} f(x_i) \right) - r \\ &= \left(x_i - \frac{x_i - x_{i-1}}{f(r + \epsilon_i) - f(r + \epsilon_{i-1})} f(r + \epsilon_i) \right) - r \\ &= x_i - r - \frac{x_i - x_{i-1}}{f(r + \epsilon_i) - f(r + \epsilon_{i-1})} f(r + \epsilon_i) \\ &= \epsilon_i - \frac{x_i - x_{i-1}}{f(r + \epsilon_i) - f(r + \epsilon_{i-1})} f(r + \epsilon_i). \end{aligned}$$

Notice also that

$$x_i - x_{i-1} = (\epsilon_i + r) - (\epsilon_{i-1} + r) = \epsilon_i - \epsilon_{i-1},$$

so we have

$$\begin{aligned}
\epsilon_{i+1} &= \epsilon_i - \frac{\epsilon_i - \epsilon_{i-1}}{f(r+\epsilon_i) - f(r+\epsilon_{i-1})} f(r+\epsilon_i) \\
&= \epsilon_i \left(1 - \frac{f(r+\epsilon_i)}{f(r+\epsilon_i) - f(r+\epsilon_{i-1})}\right) + \epsilon_{i-1} \frac{f(r+\epsilon_i)}{f(r+\epsilon_i) - f(r+\epsilon_{i-1})} \\
&= \frac{\epsilon_i \left(f(r+\epsilon_i) - f(r+\epsilon_{i-1}) - f(r+\epsilon_i)\right) + \epsilon_{i-1} f(r+\epsilon_i)}{f(r+\epsilon_i) - f(r+\epsilon_{i-1})} \\
&= \frac{-\epsilon_i f(r+\epsilon_{i-1}) + \epsilon_{i-1} f(r+\epsilon_i)}{f(r+\epsilon_i) - f(r+\epsilon_{i-1})}
\end{aligned}$$

Assuming that $f(x)$ is twice-differentiable, we can expand the $f(r+\epsilon)$ terms about $\epsilon = 0$ using a Taylor series:

$$\begin{aligned}
f(r+\epsilon_i) &= f(r) + f'(r)\epsilon_i + \frac{1}{2!} f''(r)\epsilon_i^2 + ... \\
&= f'(r)\epsilon_i + \frac{1}{2!} f''(r)\epsilon_i^2 + ...
\end{aligned}$$

since r is by definition a root of f, so $f(r) \equiv 0$.

Define the Taylor expansion of $f(r+\epsilon_i)$ to second order as T_i:

$$T_i = f'(r)\epsilon_i + \frac{1}{2!} f''(r)\epsilon_i^2$$

Then our expression for ϵ_{i+1} can be written, to second order in ϵ:

$$\begin{aligned}
\epsilon_{i+1} &= \frac{-\epsilon_i T_{i-1} + \epsilon_{i-1} T_i}{T_i - T_{i-1}} \\
&= -\epsilon_i \frac{f'(r)\epsilon_{i-1} + \frac{1}{2!} f''(r)\epsilon_{i-1}^2}{T_i - T_{i-1}} + \epsilon_{i-1} \frac{f'(r)\epsilon_i + \frac{1}{2!} f''(r)\epsilon_i^2}{T_i - T_{i-1}} \\
&= \frac{-f'(r)\epsilon_i\epsilon_{i-1} - \epsilon_i \frac{1}{2!} f''(r)\epsilon_{i-1}^2 + f'(r)\epsilon_i\epsilon_{i-1} + \frac{1}{2!} f''(r)\epsilon_{i-1}\epsilon_i^2}{T_i - T_{i-1}} \\
&= \frac{\frac{1}{2!} f''(r)(\epsilon_{i-1}\epsilon_i^2 - \epsilon_i\epsilon_{i-1}^2)}{T_i - T_{i-1}} = \epsilon_i\epsilon_{i-1} \frac{\frac{1}{2!} f''(r)(\epsilon_i - \epsilon_{i-1})}{T_i - T_{i-1}}
\end{aligned}$$

Next we analyze the denominator:

$$\begin{aligned}
T_i - T_{i-1} &= f'(r)(\epsilon_i - \epsilon_{i-1}) + \frac{1}{2!} f''(r)(\epsilon_i^2 - \epsilon_{i-1}^2) \\
&= f'(r)(\epsilon_i - \epsilon_{i-1}) + \frac{1}{2!} f''(r)(\epsilon_i - \epsilon_{i-1})(\epsilon_i + \epsilon_{i-1}) \\
&= (\epsilon_i - \epsilon_{i-1}) \left(f'(r) + \frac{1}{2!} f''(r)(\epsilon_i + \epsilon_{i-1})\right)
\end{aligned}$$

Thus the $(\epsilon_i - \epsilon_{i-1})$ cancel in the numerator and denominator, leaving us with

$$\epsilon_{i+1} = \epsilon_i\epsilon_{i-1} \frac{\frac{1}{2!} f''(r)}{f'(r) + \frac{1}{2!} f''(r)(\epsilon_i + \epsilon_{i-1})} = \epsilon_i\epsilon_{i-1} \frac{\frac{1}{2!} f''(r)}{f'(r) + \mathcal{O}(\epsilon)} \approx \epsilon_i\epsilon_{i-1} \frac{f''(r)}{2f'(r)}$$

Let's return to our goal. Namely, we are to find m such that

$$|\epsilon_{i+1}| \approx |K \times (\epsilon_i)^m|,$$

where K is some constant.

In order for this to hold, we must have

$$\begin{aligned}
\left|\epsilon_i\epsilon_{i-1}\frac{f''(r)}{2f'(r)}\right| &\approx |K\times(\epsilon_i)^m| \\
|\epsilon_i\epsilon_{i-1}D| &\approx |K\times(\epsilon_i)^m| \\
|\epsilon_{i-1}| &\approx \left|\frac{K}{D}\right|\times(\epsilon_i)^{m-1}| \\
\implies |\epsilon_i| &\approx \left|\frac{K}{D}\right|\times(\epsilon_{i+1})^{m-1}| \\
\implies |\epsilon_{i+1}| &\approx \left|\left(\frac{D}{K}\right)^{1/(m-1)}\right|\times|(\epsilon_i)^{1/(m-1)}| \\
&\approx |K\times(\epsilon_i)^m|.
\end{aligned}$$

Since K and D are constants, the above expression can only hold if

$$m=\frac{1}{m-1}\implies m^2-m-1=0\implies m=\frac{1+\sqrt{5}}{2}=1.61803...,$$

the Golden Ratio!

10.3 False Position or *Regula Falsi* Method

The Method of False Position is, like the Bisection Method, a bracketing method. However, like the Secant Method, it makes use of Secant lines and thus requires that functions be second-differentiable. False Position is more robust than the Secant method, as it will never encounter the case in which the slope of the line is zero.

One implementation of the False Position algorithm is as follows:

1. Choose $x=a$ and $x=b$ such that $(f(a)f(b))<0$, $a<b$.
2. Draw the line that connects $f(a)$ and $f(b)$. So long as $f(a)\neq f(b)$, this line is guaranteed to pass through zero, and where it passes through zero is our next guess x_1.
   (a) If $(f(a)f(x_1))<0$ then set $b=x_1$.
   (b) If $(f(x_1)f(b))<0$ then set $a=x_1$.
3. If $|a-b|\leq\epsilon(|a|+|b|)/2$, where ϵ is the maximum relative error we are willing to tolerate (note that $|a-b|$ denotes the size of our interval), then **exit the algorithm**. Otherwise **Go to 2.**

So suppose $f(a)<f(x_1)<f(b)$ and $f(a)f(x_1)<0$. Then, so long as the function $f(x)$ is continuous, we know a root exists between a and x_1. The False Position method will always choose the next point such that the function is positive on one side of the bracket and negative on the other side. This choice will typically result in slower convergence than the Secant Method, and in fact False Position converges at a rate that is linear, but faster than bisection.

The bottom plot in Fig. 10.1 illustrates the False Position method.

10.4 Newton-Raphson Method

We'll find that the Newton-Raphson Method for finding roots is the most efficient method yet discussed, but requires that the function $f(x)$ and its derivative $f'(x)$ be evaluated at arbitrary x. Like the Secant method, it does not require bracketing, but unlike Secant, only requires one point for an initial guess.

Suppose our current guess for a root is x, and suppose our actual root is at $r=x+\delta$. What should our next guess be? The Newton-Raphson Method addresses this question using a Taylor expansion of $f(x+\delta)$:

$$f(x+\delta)=f(x)+f'(x)\delta+\frac{f''(x)}{2!}\delta^2+...$$

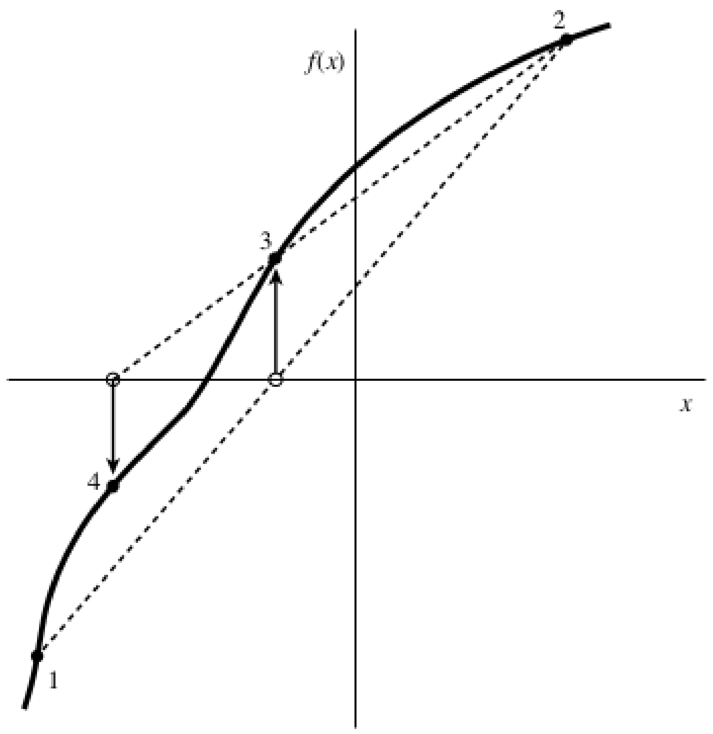

Figure 9.2.1. Secant method. Extrapolation or interpolation lines (dashed) are drawn through the two most recently evaluated points, whether or not they bracket the function. The points are numbered in the order that they are used.

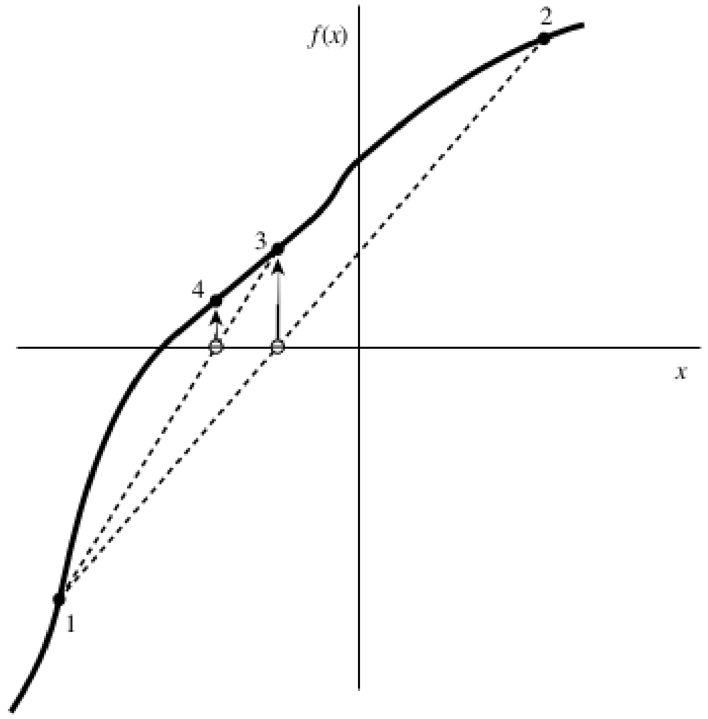

Figure 9.2.2. False position method. Interpolation lines (dashed) are drawn through the most recent points *that bracket the root*. In this example, point 1 thus remains "active" for many steps. False position converges less rapidly than the secant method, but it is more certain.

Figure 10.1: Secant/False Position. Figures from *Numerical Recipes in C, Second Edition*, Press, Teukolsky, Vetterling, and Flannery. Cambridge University Press, 1999.

For small δ, we can write

$$f(x+\delta) \approx f(x) + f'(x)\delta.$$

Thus if $f(x+\delta)$ is our root, then we can immediately get a good approximation for the distance we should move from some guess x nearby by evaluating the right-hand side of the below expression:

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

So Newton-Raphson starts with some guess x_0. $f(x_0)$ and $f'(x_0)$ are evaluated, and the next guess $x_1 = x_0 + \delta$, where δ is simply the distance $\delta = -\frac{f(x_0)}{f'(x_0)}$.

Source: `https://en.wikipedia.org/wiki/Newton%27s_method#Practical_considerations`

What is the rate of convergence for Newton-Raphson? Well, suppose we have a root at r (i.e., $f(r) = 0$) and f is twice differentiable in the neighborhood of r. Let's Taylor expand $f(r)$ about x_i, our guess for the root at iteration i:

$$f(r) = 0 = f(x_i) + f'(x_i)(r - x_i) + \frac{f''(x_i)(r-x_i)^2}{2!} + ...$$

Assume that the higher-order terms are negligible. Then divide both sides by $f'(x_i)$:

$$\frac{f(x_i)}{f'(x_i)} + (r - x_i) \approx \frac{-f''(x_i)}{2f'(x_i)}(r-x_i)^2$$

Recall that the Newton-Raphson formula is:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \implies (r - x_{i+1}) = (r - x_i) + \frac{f(x_i)}{f'(x_i)} \approx \frac{-f''(x_i)(r-x_i)^2}{2!},$$

so we have

$$\underbrace{r - x_{i+1}}_{\epsilon_{i+1}} \approx \frac{-f''(x_i)}{2f'(x_i)}(\underbrace{r - x_i}_{\epsilon_i})^2.$$

If we are close to the root, then $\frac{-f''(x_i)}{2f'(x_i)}$ will be nearly constant and we have shown that the error in the Newton-Raphson method drops as the square of the error term, so we say it converges *quadratically.*

10.4.1 Newton Raphson Example

Exercise #2

Source: `http://www3.ul.ie/~mlc/support/CompMaths2/files/NewtonExample.pdf`

Write the Newton-Raphson algorithm necessary to find the roots of

$$f(x) = x^3 - x - 1.$$

Your answer will be in the form

$$x_{i+1} = x_i + \ell(x_i)$$

I.e., find $\ell(x_i)$. Next fully simplify this expression so that it may be written as a simple fraction of polynomials $\alpha(x_i)$ and $\beta(x_i)$:

$$x_{i+1} = \frac{\alpha(x_i)}{\beta(x_i)}.$$

Then compute the factor R_i such that

$$\epsilon_{i+1} \approx R_i \epsilon_i^2.$$

Solution to Exercise #2

The Newton-Raphson algorithm is given by

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

Thus if $f(x) = x^3 - x - 1$, the algorithm can be written

$$\begin{aligned} x_{i+1} &= x_i - \frac{x_i^3 - x_i - 1}{3x_i^2 - 1} \\ &= \frac{2x_i^3 + 1}{3x_i^2 - 1}. \end{aligned}$$

Next we compute the factor R_i such that

$$\epsilon_{i+1} \approx R_i \epsilon_i^2.$$

Based on the expression we already derived for $r - x_{i+1}$, we find

$$\underbrace{r - x_{i+1}}_{\epsilon_{i+1}} \approx \frac{-f''(x_i)}{2f'(x_i)}(\underbrace{r - x_i}_{\epsilon_i})^2.$$

Plugging $f'(x_i)$ and $f''(x_i)$ into this expression we get:

$$R_i \approx -\frac{3x_i}{3x_i^2 - 1}$$

Exercise #3

Suppose we have found that the Newton-Raphson method has begun to converge on a root, with $|x_{i-1} - x_i| = 0.1$ and $\frac{-f''(x_i)}{2f'(x_i)} \approx 1$. About how many more iterations should we expect to get $|\epsilon| < 10^{-10}$?
Finally, if the bracket size shrinks by a factor of $1/2$ each iteration, as in the Bisection Method, approximately how many more iterations would we need to achieve a bracket width $|x_{i-1} - x_i| < 10^{-10}$? You may use the fact that $\log_{10}(2) \approx 0.3$.

Solution to Exercise #3

By definition, $\epsilon_i = r - x_i$. Also, for Newton-Raphson, $\epsilon_{i+1} \approx K(\epsilon_i)^2$. We are told to assume that $K \approx 1$. So if $|x_{i-1} - x_i| = 0.1$, then by these facts, $|x_{i-1} - x_i| = 0.1 \approx |\epsilon_{i-1} - K\epsilon_{i-1}^2|$. If the Newton-Raphson method has begun to converge on a root, then the $K\epsilon_{i-1}^2$ is negligible in magnitude compared to ϵ_{i-1}. Thus we may conclude that $\epsilon_{i-1} \approx 0.1$.
We are to find how many iterations we would need to get $\epsilon_i < 10^{-10}$.
We are given that $\epsilon_{i-1} \approx 0.1$. Based on this fact and the derived convergence rate of Newton-Raphson, we get:

$$\begin{aligned} \epsilon_i &\approx \epsilon_{i-1}^2 \approx 10^{-2} \\ \epsilon_{i+1} &\approx 10^{-4} \\ \implies \epsilon_{i+2} &\approx 10^{-8} \\ \implies \epsilon_{i+3} &\approx 10^{-16} \end{aligned}$$

So the answer is **3 more iterations**.
Finally, if $R_i = 1/2$, as in the Bisection Method, approximately how many more iterations would we need for a method that converged linearly to the solution, using the fact that $\log_{10}(2) \approx 0.3$?
For the linear convergence case with $R_i = 1/2$, we have:

$$\begin{aligned} \epsilon_{i+1} &\approx \frac{1}{2}\epsilon_i \\ \implies \epsilon_{i+2} &\approx \frac{1}{2}\epsilon_{i+1} = \frac{1}{2^2}\epsilon_i \\ \implies \epsilon_{i+m} &\approx \frac{1}{2^m}\epsilon_i \end{aligned}$$

Thus if $\epsilon_i = 0.1$, and $\epsilon_{i+m} =$`1e-10`, then we get:

$$\begin{aligned} 10^{-10} &\approx 0.1/2^m \\ \implies -10 &\approx -1 - m\log_{10} 2 \\ \implies 9/\log_{10}(2) &\approx m \approx 9/0.3 = 90/3 = 30 \end{aligned}$$

So the answer is approximately **30 more iterations**. Given how much more quickly Newton-Raphson will converge when compared to Bisection, it is often the case that the highly-robust but slowly-converging Bisection Method will be used only as a backup in case Newton-Raphson does not converge.

10.5 Fixed-Point Iteration (FPI) without Derivatives

Source: `http://wwwf.imperial.ac.uk/metric/metric_public/numerical_methods/iteration/fixed_point_iteration.html`

Newton's method is one example of a fixed-point iteration (FPI) method. In short, FPI methods for solving $f(x) = 0$ for x take the form

$$x_{i+1} = g(x_i),$$

where $g(x_i)$ may depend on $f(x)$ and/or its derivatives.

For the purposes of this section, we will focus exclusively on FPI methods for solving $f(x) = 0$ without derivatives. I.e., we solve $f(x) = 0$ such that

1. $f(x) = 0$ is rewritten as $x = g(x)$, where $x - g(x) = \pm f(x)^\alpha$, $g(x)$ does *not* depend on any derivatives of $f(x)$, and
2. the iteration strategy $x_{i+1} = g(x_i)$ is applied, starting with an initial guess for the root x_0.

In other words, suppose we wish to solve $f(x) = 0$, where $f(x)$ is a continuous function in the neighborhood of the root $x = r$, and that $f(x) = 0$ can be rewritten as

$$x = g(x).$$

Let's pick a guess for the solution x, and to get the next guess, we simply plug this value of x into $g(x)$. I.e.,

$$x_{i+1} = g(x_i).$$

The benefit of the Fixed-Point Iteration method without derivatives is its simplicity: just rewrite $f(x) = 0$ as $x = g(x)$; i.e., algebraically extract an x from $f(x) = 0$ and place it on the left-hand side of the expression, defining the remainder as $g(x)$. Then choose the iteration as $x_{i+1} = g(x_i)$. Finally, see if the method converges to a root. Given the ease of coding, this strategy is often used as a first attempt for finding a root. One can even easily implement it within spreadsheet software!

10.5.1 Fixed-Point Iteration without Derivatives: Convergence

How quickly does the Fixed-Point Iteration method without derivatives converge? Suppose we pick a point x that is close to the root r. By Taylor's theorem,

$$g(x) = g(r) + (x - r)g'(r) + ...$$

However, we know that $g(r) = r$, and x is close to r, so

$$g(x) - r \approx (x - r)g'(r)$$

Since

$$x_{i+1} = g(x_i)$$

for a guessed root x_i, we have

$$\begin{aligned} g(x) - r &\approx (x - r)g'(r) \\ \implies g(x_i) - r &\approx (x_i - r)g'(r) \\ x_{i+1} - r &\approx (x_i - r)g'(r) \\ \epsilon_{i+1} &\approx \epsilon_i g'(r). \end{aligned}$$

Thus this method will converge so long as $|g'(r)| < 1$ in the vicinity of the root $x = r$. In the case that x_i is very far from a root, then this analysis may not hold, because we have thrown away $(x_i - r)^2$ and higher-order terms, which may diverge.

10.5.2 Example of FPI Methods without Derivatives

Exercise #4:

Suppose you are given $f(x) = x^5 - e^x$. Explain why the following FPI method is ill-formed:

$$\begin{aligned} f(x) &= x^5 - e^x - 1 = 0 \\ \implies x^6 - xe^x - x &= 0 \\ \implies x(x^5 - e^x) &= x \\ \implies x_{i+1} &= \frac{x_i}{x_i^5 - e^{x_i}}. \end{aligned}$$

Solution to Exercise #4

Multiplying both sides of $f(x) = 0$ by x:

$$x^6 - xe^x - x = 0$$

adds an additional root, $x = 0$, which does not exist in the original equation.

Source: `https://mat.iitm.ac.in/home/sryedida/public_html/caimna/transcendental/iteration%20methods/fixed-point/iteration.html`

Example

Note that there is often no unique way of rewriting $f(x) = 0$ as $x = g(x)$. For example, consider $f(x) = x^4 - 10 - x = 0$. Let's construct four independent FPI methods without derivatives and analyze how well these FPI methods can find the root at $x = r \approx 1.85558$, with initial guess of neighboring point $x_0 = 2$.

Let's analyze four different FPI methods for solving $f(x) = x^4 - 10 - x = 0$

1. First rewrite $f(x) = 0$ as
$$f(x) = (x^4 - 10) - x = 0 = g(x) - x,$$
to make a choice of $g(x)$ more transparent: $g(x) = (x^4 - 10)$. Then our FPI scheme is given by
$$x_{i+1} = g(x_i) = x_i^4 - 10.$$
Clearly $|g'(x_i)| = 4|x_i^3| > 1$ for all $|x_i| > \frac{1}{4^{1/3}} \approx 0.63$, and it turns out there are *no* roots (real or otherwise) with $|x_i| < 1$. Thus this FPI method will *not* converge to any roots for this method.

2. But there are many other ways to rewrite this $f(x)$. It just requires that we recognize that finding $g(x)$ depends only on us rewriting $f(x) = x^4 - x - 10 = 0$ so that x appears by itself on one side of the equation.

   Of course we can add x to both sides and this is the $g(x)$ we just found above. Let's instead try adding $x + 10$ to both sides and then taking the fourth root of both sides. Then we get $g(x) = (x + 10)^{1/4}$ and
$$x_{i+1} = (x_i + 10)^{1/4}.$$
Notice that $g'(x) = (x + 10)^{-3/4}/4$. For $x = 2$, $|g'(x)| = 0.0388$, and in fact this method does converge quite rapidly to the root at $x = r \approx 1.85558$ even for, e.g., $x_0 = 200$.

3. Let's try another strategy: Add $x + 10$ to both sides, then take the square root of both sides, then divide by x: $g(x) = \sqrt{x + 10}/x$, so the FPI method may be written
$$x_{i+1} = \sqrt{x_i + 10}/x_i.$$
In this case, $|g'(x)| = 0.793$, which converges extremely slowly to the desired root.

4. One more: Add 10 to both sides, then divide both sides by $(x^3 - 1)$: $g(x) = 10/(x^3 - 1)$, yielding the FPI method:
$$x_{i+1} = 10/(x_i^3 - 1).$$
In this case, $|g'(2)| = 2.449$, which diverges!

Thus we have found, of four FPI methods, one very rapidly converges to the root (the second one), one converges slowly (the third one), and two do not converge at all. Despite its hit-or-miss character, FPI without derivatives is widely used because it is by far the most trivial method to program, and can exhibit very rapid convergence for many functions.

10.6 Root-Finding Algorithms: Summary

Below is a table summarizing advantages & disadvantages of the various root-finding algorithms we explored.

Root-finding method	Bracketing Method?	Easy to Code?	How Robust?	How Fast to Converge?
Bisection	Yes	Easy	**Most Robust**; $f(x)$ need only be continuous in bracket	**Slow**: linear ($m = 1$) convergence
Secant Method	No	Easy–Moderate	**Less Robust**; $f''(x)$ needs to exist everywhere; problems near $f'(x) = 0$	**Moderate**: $m \approx 1.612$
False Position	Yes	Easy–Moderate	**Robust**; $f''(x)$ needs to exist in bracket	**Slow–Moderate**: linear ($m = 1$) convergence, but usually faster than Bisection
Newton-Raphson	No	Easy–Moderate	**Less Robust**; Problems near $f'(x) = 0$	**Fast**: quadratic ($m = 2$) convergence
Fixed-Point Iteration (FPI) without Derivatives	No	Very Easy	**Least robust**; depends on whether $g'(x_i) < 1$ as $n \to \infty$	**Varies**, depending on choice of $g(x)$.

10.7 A Two-Dimensional Bisection Method

The Bisection Method is known as a **bracketing method**, requiring that we first find two points $x = a$ and $x = b$ such that $f(a)f(b) < 0$. This requires that the underlying function be continuous and *pass through* zero at the root, instead of just touching zero, like $f(x) = x^2$ at $x = 0$.

For two functions $\mathbf{f} = \{f_1, f_2\}$ of two variables $\mathbf{x} = (x_1, x_2)$, function f_1 will possess its own set of points $\{\mathbf{x}\}_1$ such that $f_1(\{\mathbf{x}\}_1) = 0$. Similarly, f_2 will possess its own set of points $\{\mathbf{x}\}_2$ such that $f_2(\{\mathbf{x}\}_2) = 0$. If each function is continuous and *passes through* zero at its roots, this set of points will be infinite and create a continuous curve $\mathcal{C}_i$ on the (x_1, x_2) plane. Our goal is to find a point where curves $\mathcal{C}_1$ and $\mathcal{C}_2$ intersect, as this will be a point at which both functions share a root.

To construct an algorithm that will enable us to find this point of intersection, we must first define a bracket in two dimensions. In one dimension, we needed only find $x = a$ and $x = b$ such that $f(a)f(b) < 0$. Thus we only needed to sample the (continuous) function twice to guarantee that the two points bracket the root.

In two dimensions, we are trying to find the intersection of the curves $\mathcal{C}_1$ and $\mathcal{C}_2$. We know that on one side of $\mathcal{C}_1$, $f_1 > 0$ and on the other side $f_1 < 0$. Similarly, on one side of $\mathcal{C}_2$, $f_2 > 0$ and on the other side $f_2 < 0$. Now consider the point where the two curves $\mathcal{C}_1$ and $\mathcal{C}_2$ intersect. As this point will appear where two curves intersect, near the point of intersection, the curves create four regions:

1. Region A, where $f_1 > 0$ and $f_2 > 0$
2. Region B, where $f_1 < 0$ and $f_2 > 0$
3. Region C, where $f_1 > 0$ and $f_2 < 0$, and
4. Region D, where $f_1 < 0$ and $f_2 < 0$.

If we pick a single point in each of these four regions, so that they form the edges of a convex polygon, then the polygon should provide a starting bracket for our two-dimensional Bisection Method.

Then the two-dimensional Bisection Method is as follows:

1. Evaluate the functions at the four midpoints of each edge of the convex polygon. Then draw the two line segments that connecting midpoints from opposite sides. Evaluate the function where these line segments intersect. If both functions are zero at any of these five points, then **end** the algorithm.
2. Where before you had a single convex polygon, the new line segments have broken it into a total of four convex polygons. For the next iteration, pick the one polygon out of these four whose vertices represent all four regions. **Stop if convergence criterion is met**, otherwise **go to Step 1**.

MATH 521, Fall 2018 Notes

Prof. Zachariah B. Etienne

Chapter 11: Optimization: Finding Minima and Maxima of Functions

Optimization is a challenging problem; sometimes we look for *global* minima or maxima, but our algorithm can get stuck in *local* minima or maxima. Similar to the fact that there exists no root-finding method for finding *all* zeroes of a function, there is no algorithm that guarantees finding *global* function extrema for all possible functions.

To make matters worse, the precision at which extrema might be found is typically orders of magnitude less than root finding methods. To understand why, consider the fact that when $f(x)$ is differentiable and $f'(x) \neq 0$ at a root, $f(x)$ can be well-approximated by a line with nonzero slope near a root. This fact lies at the heart of the Secant, False Position, and Newton-Raphson methods. Thus near the root, $f(x) \approx ax + c$, where $a \neq 0$ and c are constants. So when we are *near* a root but $f(x_i) \neq 0$,

$$E_{\rm rel}(f(x_i)) = \left|\frac{f(x_i) - f(x_{i-1})}{f(x_i)}\right| \approx E_{\rm rel}(x_i) = \left|\frac{x_i - x_{i-1}}{x_i}\right|.$$

This implies that *when we are searching for roots*, if $f(x_i)$ and $f(x_{i-1})$ are the same to 15 significant digits, then we can typically expect x_i and x_{i-1} to be the same to within 15 significant digits as well. Thus we can generally expect to find roots (x such that $f(x) = 0$) to 15–16 significant digits in double precision.

But it comes to function minimization, the same does not hold true! When we have converged to a minimum in double precision, we will have found values x_i and x_{i-1} such that $f(x_i)$ and $f(x_{i-1})$ are the same to 15–16 significant digits (i.e., $|f(x_i) - f(x_{i-1})|/|f(x_i)| \sim 10^{-15}$). But near the minimum, the function *cannot* be approximated by a line with nonzero slope, so $|f(x_i) - f(x_{i-1})|/|f(x_i)| \sim 10^{-15}$ does *not* imply that x_i and x_{i-1} are the same to 15 significant digits! So how many digits should we expect?

We know that near a minimum at $x = \alpha$,

$$f(x) \approx f(\alpha) + \frac{f''(\alpha)}{2!}(x-\alpha)^2, \tag{11.1}$$

where $f''(\alpha) > 0$ for a minimum.

So when we have reached found the function's minimum to within machine epsilon; i.e.,

$$\left|\frac{f(x) - f(\alpha)}{f(\alpha)}\right| = \epsilon_m$$

(assuming $f(\alpha) \neq 0$ and $\alpha \neq 0$; can always shift $f(x)$ or x to avoid this eventuality), then, using Eq. (11.1), we get

$$\begin{aligned}
\left|\frac{f(x) - f(\alpha)}{f(\alpha)}\right| &= \epsilon_m \\
&\approx \frac{f''(\alpha)}{2|f(\alpha)|}(x-\alpha)^2 \\
\implies |x-\alpha| &\approx \sqrt{\epsilon_m \frac{2|f(\alpha)|}{f''(\alpha)}} \\
\implies \frac{|x-\alpha|}{|\alpha|} &\approx \sqrt{\epsilon_m}\sqrt{\frac{2|f(\alpha)|}{\alpha^2 f''(\alpha)}}
\end{aligned}$$

The right-most term will typically be of order 1, since most functions vary over a characteristic or "curvature" scale $x_c \sim \sqrt{|f(\alpha)/f''(\alpha)|}$, and a minimum will typically appear within a few characteristic scales ($x_c \sim \alpha$), which implies $\sqrt{|f(\alpha)/[\alpha^2 f''(\alpha)]|} \sim 1$. This means that we should only expect to find x such that $f(x)$ is a minimum, to about 8 significant digits in double precision! In other words, don't set your tolerance below about 1 part in 10^8 when finding maxima and minima using double-precision arithmetic.

Next we will review basic algorithms for finding local minima of functions $f(x)$. To find local maxima, one may simply look for minima of $-f(x)$.

11.1 Minimization through Bisection

We are familiar with the Bisection Method for root finding. How might we extend this method to find, say, minima of a function?

Suppose we have a function $f(x)$ that is continuous and we have found three evenly-spaced points

$$x_1 < x_2 = \frac{x_1 + x_3}{2} < x_3$$

such that $f(x_2) < f(x_1)$ and $f(x_2) < f(x_3)$. The bisection method for minimization evaluates f at point x_4 midway between x_1 and x_2 and point x_5 midway between x_2 and x_3. We then choose the interval that gets us closer to the minimum:

- If $f(x_4) < f(x_2)$ then we choose the leftmost three points, $x \in \{x_1, x_4, x_2\}$, for the start of our next bisection step.
- If $f(x_5) < f(x_2)$, we choose the rightmost three points, $x \in \{x_2, x_5, x_3\}$, for the start of our next bisection step.
- Otherwise, we choose the center three points, $x \in \{x_4, x_2, x_5\}$, for the start of our next bisection step.

At the end of each iteration, we check if our convergence criterion is satisfied. Typically if x_i corresponds to the current minimum value of the function and x_{i-1} is the next smallest function value, the criterion would be written $|x_i - x_{i-1}| < \epsilon(|x_i| + |x_{i-1}|)/2$, where $\epsilon \sim \sqrt{\epsilon_m} \sim 10^{-8}$. If it is satisfied, we have found the minimum and exit the loop. Otherwise, we repeat the method, re-defining our leftmost point to be x_1, center point x_2, and rightmost point x_3.

Although we are guaranteed to find a *local* minimum between $x = x_1$ and $x = x_3$ via bisection, notice that each iteration requires that we evaluate the function at two points. Usually, the most expensive part of a minimization algorithm is the evaluation of the function. Let's next examine a minimization algorithm that requires only *one* function evaluation per iteration, the Golden Section Search.

11.2 Golden Section Search

Unlike the bisection method, the Golden Section Search (GSS) requires only one function evaluation per iteration. Here's how it works.

Source: `https://en.wikipedia.org/w/index.php?title=Golden_section_search&oldid=688437219`

GSS, like Bisection Minimization, is a bracketing method. So first we must choose three points x_1, x_2, x_3 such that $x_1 < x_2 < x_3$, $f(x_2) < f(x_1)$, and $f(x_2) < f(x_3)$.

We define $f_1 = f(x_1)$, $f_2 = f(x_2)$, and $f_3 = f(x_3)$.

The idea with Golden Section search is to re-use previous function evaluations so that we only need a *single* function evaluation to reduce the size of the interval. The key to this method is ensuring that the center point of our bracket is never in the middle of the interval, instead placing it off-center by a set amount so that the bracketing interval reduces in size by a fixed factor at each iteration.

As shown in Fig. 11.1 (Figure from: `https://en.wikipedia.org/w/index.php?title=Golden_section_search&oldid=688437219`), define $a = x_2 - x_1$, $b = x_3 - x_2$, and $c = x_4 - x_2$. We want the bracketing interval to reduce in size by the same factor, a/b, at each iteration. To accomplish this requires that $a \neq b$, unlike the bisection method. Our intervals will decrease in size by the same factor a/b at each iteration, *regardless* of whether the value $f(x_4) = f_4$ is less than or greater than f_2.

In the case that, $f_2 < f_4$ we choose the next bracketing interval to be $x \in (x_1, x_4)$

$$\frac{c}{a} = \frac{a}{b}.$$

In case $f_2 > f_4$, we choose the next bracketing interval to be $x \in (x_2, x_3)$, so the ratio of distances a/b becomes

$$\frac{c}{b-c} = \frac{a}{b}.$$

Remember that we wish the bracketing interval to decrease in size by a/b at each iteration, *regardless of whether* $f_2 < f_4$ *or* $f_2 > f_4$, so we set

$$\frac{c}{a} = \frac{a}{b} = \frac{c}{b-c}.$$

The first equation implies that $c = a^2/b$, so the second equation yields

$$\begin{aligned} \frac{a}{b} &= \frac{a^2/b}{b - a^2/b} \\ &= \frac{a^2}{b^2 - a^2} \\ \Longrightarrow \frac{b}{a} &= \frac{b^2 - a^2}{a^2} = \frac{b^2}{a^2} - 1 \\ \Longrightarrow \frac{b}{a} &= \frac{1 \pm \sqrt{5}}{2} \end{aligned}$$

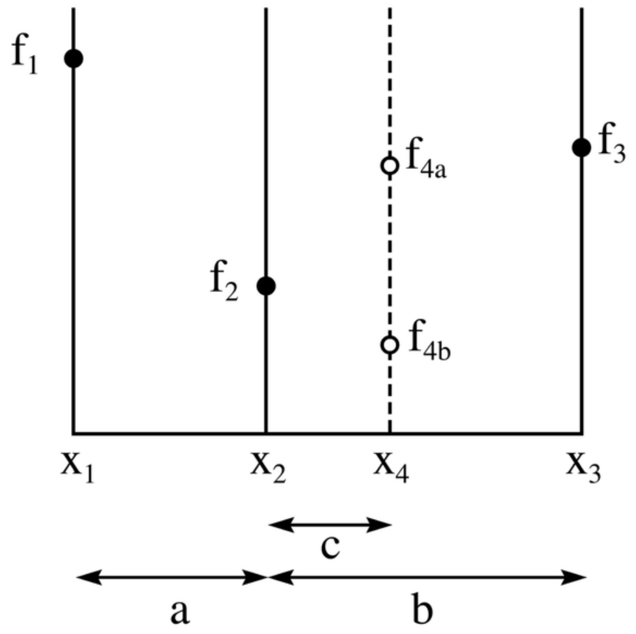

Figure 11.1: Golden Section Search illustration. (See text for source.)

The only value for which this ratio is positive is called the Golden Ratio ϕ:

$$\frac{b}{a} = \frac{1 + \sqrt{5}}{2} = \phi \approx 1.618$$

Thus the interval will decrease in size each iteration by a factor of $a/b = 1/\phi \approx 0.618$, which is exactly one over the Golden Ratio. Note that since the golden ratio $\phi = \frac{b}{a}$ solves the equation

$$\frac{b}{a} = \frac{b^2}{a^2} - 1,$$

it also has the unique property (multiplying the above equation by $\frac{a}{b}$):

$$\begin{aligned} 1 &= \frac{b}{a} - \frac{a}{b} \\ &= \phi - \frac{1}{\phi} \\ \Longrightarrow \frac{1}{\phi} &= \phi - 1. \end{aligned}$$

That is to say, the Golden Ratio ϕ has the property that its reciprocal $\frac{1}{\phi}$ is equivalent to the difference between itself and 1. Hence it naturally relates division and subtraction.

Here's a summary of the Golden Section Search algorithm.

0. Define $\Phi = (\sqrt{5} - 1)/2$.

1. Evaluate f_1, f_2, and f_3, where $x_1 < x_3$ and $x_2 = x_3 + \Phi(x_1 - x_3)$, such that f_1 and f_3 is known to bracket a local minimum.
2. Let $x_4 = x_1 + \Phi(x_3 - x_1)$. Evaluate f_4.
3. If:
   (a) $f_2 < f_4$, leave x_1 and f_1 fixed, and make replacements $\{x_3, f_3\} \leftarrow \{x_4, f_4\}$, $\{x_4, f_4\} \leftarrow \{x_2, f_2\}$. Then set $x_2 = x_3 + \Phi(x_1 - x_3)$ and evaluate f_2. **Go to Step 4.**
   (b) $f_2 \geq f_4$, leave x_3 and f_3 fixed, and make replacements $\{x_1, f_1\} \leftarrow \{x_2, f_2\}$, $\{x_2, f_2\} \leftarrow \{x_4, f_4\}$, Then set $x_4 = x_1 + \Phi(x_3 - x_1)$ and evaluate f_4. **Go to Step 4.**
4. We now have $[x_1, x_2, x_4, x_3]$ bracketing our minimum. Check for convergence, using method along same lines as Bisection Minimization method. If we have not yet converged, **go to Step 3**.

Often when we are presented with an algorithm, there are equations that we do not immediately understand, and it is up to us to derive them to ensure that a typo was not made by the authors. In light of this, let's next try to relate the equation $x_2 = x_3 + \Phi(x_1 - x_3)$ with the Golden Section Search algorithm as outlined previously.

The equation in Steps 2 and 3a relates x_1, x_2, x_3, and ϕ as follows:

$$\begin{aligned} x_2 &= x_3 + \Phi(x_1 - x_3) \\ \implies x_2 - x_3 &= \Phi(x_1 - x_3). \end{aligned}$$

In terms of the distances a and b we have defined previously, this equation can be written

$$\implies -b = -\Phi(a+b) \implies \Phi = \frac{b}{a+b}.$$

From our previous discussion, we know that $b/a = \phi$, so let's evaluate the reciprocal of the expression:

$$\frac{1}{\Phi} = \frac{a+b}{b} = \frac{a}{b} + 1 = \frac{1}{\phi} + 1.$$

Thus we have found that

$$\frac{1}{\phi} = \frac{1}{\Phi} - 1. \qquad (11.2)$$

We know that the Golden Ratio has the property $\frac{1}{\phi} = \phi - 1$, so in order for Eq. 11.2 to be true, ϕ must be equal to $\frac{1}{\Phi}$. Alternatively we need only prove that $\Phi = \frac{1}{\phi}$:

$$\begin{aligned} \Phi &= \frac{\sqrt{5}-1}{2} \\ &= \frac{\sqrt{5}-1}{2}\left[\frac{\sqrt{5}+1}{\sqrt{5}+1}\right] \\ &= \frac{5-1}{2(\sqrt{5}+1)} \\ &= \frac{2}{\sqrt{5}+1} = \frac{1}{\phi} \quad \checkmark \end{aligned}$$

So we have shown that the equation in Steps 2 and 3a is consistent with the Golden Section Search algorithm we derived previously.

Similarly, the equation in Step 3b is given by

$$\begin{aligned} x_4 &= x_1 + \Phi(x_3 - x_1) \\ \implies x_4 - x_1 &= \Phi(x_3 - x_1) \\ c + a &= \Phi(a+b), \end{aligned}$$

but we know that $c/a = a/b = \frac{1}{\phi}$ and $\Phi = \frac{1}{\phi}$, so dividing both sides by a, we get

$$\begin{aligned} \frac{c+a}{a} &= \Phi\frac{a+b}{a} \\ \frac{1}{\phi}+1 &= \Phi(1+\phi) \\ \frac{1}{\phi}+1 &= \frac{1}{\phi}(1+\phi) \quad \checkmark \end{aligned}$$

Thus we have demonstrated that the algorithm given above for the Golden Section Search indeed reduces the interval by a factor of $a/b = 1/\phi \equiv \Phi \approx 0.618$ at each iteration.

11.3 Brent's Method

R.P. Brent, Algorithms for Minimization without Derivatives (Englewood Cliffs, NJ: Prentice Hall), Ch. 5.

Knowing three values of the function that bracket a minimum, why don't we fit a parabola to these three values of the function and choose the point x where the parabola is minimized as our next guess?

The extremum of a parabola $f(x) = \alpha x^2 + \beta x + \gamma$ occurs when $2\alpha x + \beta = 0 \implies x = -\beta/(2\alpha)$. Thus we'll need to solve

$$\begin{aligned} f_1 &= \alpha x_1^2 + \beta x_1 + \gamma \\ f_2 &= \alpha x_2^2 + \beta x_2 + \gamma \\ f_3 &= \alpha x_3^2 + \beta x_3 + \gamma \end{aligned}$$

for α and β. We could just use the Lagrange polynomial interpolation formula to find α and β, or just work through the arithmetic:

$$\begin{aligned} f_2 - f_1 &= \beta(x_2 - x_1) + \alpha(x_2^2 - x_1^2) \\ \implies \beta &= \frac{f_2 - f_1 - \alpha(x_2^2 - x_1^2)}{(x_2 - x_1)} \\ \implies f_2 - f_3 &= \beta(x_2 - x_3) + \alpha(x_2^2 - x_3^2) \\ &= \frac{f_2 - f_1 - \alpha(x_2^2 - x_1^2)}{(x_2 - x_1)}(x_2 - x_3) + \alpha(x_2^2 - x_3^2) \\ \implies \frac{f_2 - f_3}{x_2 - x_3} &= \frac{f_2 - f_1}{x_2 - x_1} - \alpha(x_2 + x_1) + \alpha(x_2 + x_3) \\ &= \frac{f_2 - f_1}{x_2 - x_1} + \alpha(x_3 - x_1) \\ \implies \alpha &= \frac{1}{x_3 - x_1}\left(\frac{f_2 - f_3}{x_2 - x_3} - \frac{f_2 - f_1}{x_2 - x_1}\right) \\ &= \frac{1}{(x_2 - x_1)(x_3 - x_1)(x_2 - x_3)}[(f_2 - f_3)(x_2 - x_1) - (f_2 - f_1)(x_2 - x_3)] \end{aligned}$$

Now that we've found α, let's see if we can write β so that $-\beta/(2\alpha)$ can be made as simple as possible:

$$\begin{aligned}
\implies \beta &= \frac{f_2 - f_1 - \alpha(x_2^2 - x_1^2)}{(x_2 - x_1)} \\
&= \frac{f_2 - f_1}{(x_2 - x_1)} - \alpha(x_2 + x_1) \\
&= \frac{f_2 - f_1}{(x_2 - x_1)} - \frac{x_2 + x_1}{x_3 - x_1}\left(\frac{f_2 - f_3}{x_2 - x_3} - \frac{f_2 - f_1}{x_2 - x_1}\right) \\
&= -f_1\left(\frac{1}{(x_2 - x_1)} + \frac{(x_2 + x_1)}{(x_3 - x_1)(x_2 - x_1)}\right) \\
&\quad +f_2\left(\frac{1}{(x_2 - x_1)} - \frac{(x_2 + x_1)}{(x_3 - x_1)(x_2 - x_3)} + \frac{(x_2 + x_1)}{(x_3 - x_1)(x_2 - x_1)}\right) \\
&\quad +f_3\left(\frac{(x_2 + x_1)}{(x_3 - x_1)(x_2 - x_3)}\right) \\
&= \frac{-f_1\left((x_3 - x_1)(x_2 - x_3) + (x_2 - x_3)(x_2 + x_1)\right)}{(x_2 - x_1)(x_3 - x_1)(x_2 - x_3)} \\
&\quad + \frac{f_2\left((x_3 - x_1)(x_2 - x_3) - (x_2 - x_1)(x_2 + x_1) + (x_2 - x_3)(x_2 + x_1)\right)}{(x_2 - x_1)(x_3 - x_1)(x_2 - x_3)} \\
&\quad + \frac{f_3\left((x_2 - x_1)(x_2 + x_1)\right)}{(x_2 - x_1)(x_3 - x_1)(x_2 - x_3)} \\
&= \frac{1}{(x_2 - x_1)(x_3 - x_1)(x_2 - x_3)}[-f_1\left((x_2 + x_3)(x_2 - x_3)\right) \\
&\quad +f_2\left((x_1 + x_3)(x_1 - x_3)\right) \\
&\quad +f_3\left((x_2 - x_1)(x_2 + x_1)\right)]
\end{aligned}$$

Recall the expression for α has the same denominator:

$$\alpha = \frac{1}{(x_2 - x_1)(x_3 - x_1)(x_2 - x_3)}[(f_2 - f_3)(x_2 - x_1) - (f_2 - f_1)(x_2 - x_3)]$$

Thus

$$\begin{aligned}
-\beta/(2\alpha) &= -\frac{-f_1\left((x_2 + x_3)(x_2 - x_3)\right) + f_2\left((x_1 + x_3)(x_1 - x_3)\right) + f_3\left((x_2 - x_1)(x_2 + x_1)\right)}{2[(f_2 - f_3)(x_2 - x_1) - (f_2 - f_1)(x_2 - x_3)]} \\
&= x_2 - \frac{1}{2}\frac{(x_2 - x_1)^2[f_2 - f_3] - (x_2 - x_3)^2[f_2 - f_1]}{(x_2 - x_1)[f_2 - f_3] - (x_2 - x_3)[f_2 - f_1]}
\end{aligned}$$

Exercise to reader: Prove the last step.

Here's a possible "parabolic fitting" algorithm (this is *not* Brent's method):

1. Choose three points x_1, x_2, and x_3, such that $x_1 < x_2 < x_3$ and both $f_2 < f_1$ and $f_2 < f_3$.
2. Fit a parabola to the function values at these three points. The minimum value of the parabola will appear at $x_4 = -\beta/(2\alpha)$ as computed above.
3. If
    (a) $f_4 < f_2$ then choose the two nearest points to x_4, including x_4 as the new interval. Go to Step 4.
    (b) $f_4 \geq f_2$ then choose the two nearest points to x_2, including x_2 as the new interval. Go to Step 4.
4. Check for convergence, using a similar approach as the Bisection Minimization algorithm.

So why not use this parabolic fitting method for finding the minimum? Here's why:

1. Suppose the initial x_1 is far from x_2 and x_3, and x_4 happens to appear between x_2 and x_3. In this case, regardless of whether $f_4 < f_2$ or otherwise, the next three points in the bracket will be x_2, x_4, and x_3 (the closest two points to x_2 are x_4 and x_3, and the closest two points to x_4 are x_2 and x_3). At this point we have lost our bracket if the function is monotonic between x_2, x_4, and x_3.

2. Losing the bracket means the points may become collinear, causing the minimum to be infinitely far away. This is easily seen from our expression for $-\beta/(2\alpha)$; the denominator is zero zero if the points are collinear. To see why this is the case, if κ is the slope of the line, then collinearity between the three points implies:

$$\begin{aligned} \frac{f_2 - f_3}{x_2 - x_3} &= \kappa = \frac{f_2 - f_1}{x_2 - x_1} \\ \implies (f_2 - f_3)(x_2 - x_1) &= (f_2 - f_1)(x_2 - x_3). \end{aligned}$$

   Even where functions are nearly collinear, we may be brought very far from our original bracketing interval.
3. Losing the bracket will cause the method to also latch on to local maxima.
4. Functions with discontinuous first or second derivatives in the interval will often be poorly fit by parabolas, so it is better to use Golden Section Search to find minima of these functions.

To fix these shortcomings, we adopt Brent's method, which combines a more careful parabolic interpolation method with the Golden Section Search, so that when interpolation fails to quickly converge on to a minimum, we revert to the Golden Section Search.

Brent's method works as follows:

1. Find three points x_1, x_2, and x_3 inside a and b, such that $f(x_1) < f(x_2) < f(x_3)$. Thus $f(x_1)$ is the smallest function evaluation so far, followed by $f(x_2)$ and $f(x_3)$.
2. Find the extremum point y of the parabola through $f(x_1)$, $f(x_3)$, and $f(x_2)$. We accept the new point y if both of the following conditions are satisfied:
   (a) x_4 is within a and b, and
   (b) the distance between x_1 and x_4 is less than half the distance between x_2 and x_3, to ensure that we're converging.

   When x_4 is accepted, we then relabel our points x_1, x_2, and x_3 such that $f(x_1) < f(x_2) < f(x_3)$ are the function evaluations with the three smallest values.
3. If the new point x_4 is not accepted, apply a single step of the Golden Section Search to x_1, x_2, and x_3 to find a new set of x_1, x_2, and x_3, then go to the third step. Since x_1, x_2, and x_3 may not be spaced according to the Golden Ratio, choose the x_4 trial point to be at the location $1 - 1/\phi$ times the distance from the central point in the direction of the farthest point from the central point.
4. Check for convergence.

11.4 Survey of Other Optimization Algorithms

It would require an entire semester just to scratch the surface of the literature on optimization algorithms, clearly more time than we have available to us. So instead of devoting more time to specific algorithms (we have covered only the very basics), we will now review different optimization algorithms you may encounter in your research or careers.

11.4.1 1D Methods without Derivatives

We covered some subset of these, including bisection minimization, golden section search, and Brent's method for minimization, so let's ove on to multi-D methods without derivatives.

11.4.2 Multi-D Methods without Derivatives

Nedler-Mead algorithm: simple, "walking amoeba". For an N-dimensional space, create a geometrical object that has a total of N+1 points (vertices) and all connecting line segments. This "amoeba" "walks" along the N-dimensional space through basic geometric transformations: reflection, expansion, and contraction, and each time any points (at the ends of the "arms") move, the function is evaluated at the updated points. The Nedler-Mead algorithm governs the order and magnitude of these transformations, and generally only requires of order 100 lines of code.

Powell's method starts with a set of basis vectors in an N-dimensional space. Starting with a guess for the minimum represented by vector $\boldsymbol{g}$, pick one basis vector $\boldsymbol{e_i}$ and find the value of the scalar λ that minimizes the function $f(\boldsymbol{g} + \lambda \boldsymbol{e_i})$. Replace $\boldsymbol{g} \leftarrow \boldsymbol{g} + \lambda \boldsymbol{e_i}$. This "conjugate direction" minimization is one-dimensional and can be accomplished using Brent's method for minimization. Next, pick the next basis vector $\boldsymbol{e_{i+1}}$ and minimize in that direction using exactly the same strategy. Cycle over the whole set of basis vectors until the function stops decreasing.

Genetic algorithms are based on the principles of evolutionary biology. The reason our species and many other species are so robust to our environments is because nature optimizes our DNA for reproductive fitness. That is to say, those who produce the most healthy offspring that survive and in turn themselves produce the most offspring will dominate any population. Hence their genes will carry on and are in some sense "optimal" for the environment.

The core of genetic or evolutionary algorithms is to first generate a population that represents possible minima $\boldsymbol{x}$ of some function $f(\boldsymbol{x})$. Then evaluate the fitness of that population selecting the best-fit individuals for reproduction (parents). Then generate the next population using crossover and/or mutation operations, then evaluate the fitness of the offspring to pick the parents of the next generation, etc.

Evolutionary or genetic algorithms are incredibly useful, particularly in the case where the dimensionality of the problem or space of possible minima is enormous. There are many examples outside of Nature where such algorithms have been applied, with some success. For example, to predict the price of some stock, one could represent its price over time as some sum of functions that depend on time. Each function can have some arbitrary number of input parameters, e.g., $f_i(t, a, b, c) = e^{-at^2-bt+c}$, and one can use evolutionary algorithms to find the set of input parameters that minimizes the difference between the actual stock price and the unknown function $f(t)$. Then to predict future prices of the stock, one only needs to extrapolate $f(t)$ to a future time.

Linear programming: Suppose you want to minimize or maximize a linear, multidimensional function subject to a number of constraints. This is the domain of Linear Programming. Check out the graphing calculator example from the following link:

Source: `http://www.purplemath.com/modules/linprog3.htm`

LP lecture notes: `http://www.math.ucla.edu/~tom/LP.pdf`

11.4.3 Methods with Derivatives

It is not generally the case that we can evaluate derivatives of the functions of which we wish to find extrema, but when we can, the derivative information can indeed be useful in speeding up convergence. One example is provided below.

Method of steepest descent: Approximate the function you wish to minimize by $f(\boldsymbol{x}) = \boldsymbol{x^T A x}/2 - \boldsymbol{b^T x} + c$. Start with some initial guess $\boldsymbol{g}_0$. The next value $\boldsymbol{g}_1$ will be the minimum value of $f(\boldsymbol{x})$ along the line from $\boldsymbol{g}_0$ in the direction of the local downhill gradient $-\nabla f(\boldsymbol{g}_0)$. This basically involves matrix multiplications and is a state-of-the-art algorithm for solving $\boldsymbol{Ax} = \boldsymbol{b}$ as well.

11.4.4 Application of Optimization Algorithms: Neural Networks

Neural networks are also modeled on biology, based on our understanding of how our brains function. For example, our brains have remarkable pattern-recognition capabilities, and this has to do with the fact that our brains exist as an enormously complex network of neurons—electrically excitable cells that process and transmit information both electrically and chemically. The process works by feeding sensory neurons with information, which passes into the enormous neural network. Patterns are recognized by these networks via how they are connected, and the output of the pattern recognition can be very useful. For example, when you write a check, the bank has software with a very sophisticated neural network to recognize the written numbers and translate them into floating-point numbers.

Where optimization comes into the process is in training the neural network to recognize patterns. For example, when you deposit a check, the bank has neural network algorithms that read the numbers you wrote by hand and convert them to floating point numbers. These algorithms consist of three parts: input, neural network, and output. To generate a useful neural network typically requires the use of some optimization algorithm and some form of training.

MATH 521, Fall 2018 Notes

Prof. Zachariah B. Etienne

Chapter 12: Solving Ordinary Differential Equations (ODEs)

12.1 Euler's Method

Consider the first-order ODE:

$$\frac{dy(t)}{dt} = y'(t) = f(y,t)$$

Our goal is to solve this ODE for $y(t)$, subject to some condition $y(t_0) = y_0$, which is equivalent to setting the integration constant, if this ODE were integrable. Note that $f(y,t)$ can be a function of both $y(t)$ and the independent variable t itself.

How might we go about solving this ODE on the computer? Let's start with the derivative dy/dt. By the definition of derivative, we know that

$$\frac{dy(t)}{dt} \equiv \lim_{\Delta t \to 0} \frac{y(t+\Delta t) - y(t)}{\Delta t}.$$

So when Δt is small, we know that

$$y'(t) = f(y,t) \approx \frac{y(t+\Delta t) - y(t)}{\Delta t}$$

So one strategy for solving this ODE would be to start with the "initial" or boundary condition $y(t_0) = y_0$ and use this expression to approximate the value of $y(t_1 = t_0 + \Delta t)$, as follows:

$$\begin{aligned} y(t_0 + \Delta t) = y_1 &\approx y(t_0) + \Delta t f(y,t) \\ &= y(t_0) + \Delta t \frac{dy(t)}{dt} \end{aligned}$$

Then to obtain the value at $t_0 + 2\Delta t$, and then $t_0 + 3\Delta t$, etc., we simply iterate using a fixed-point iteration method. Defining $t_i = t_0 + i\Delta t$ and $y_i = y(t_0 + i\Delta t)$, the method can be written as

$$y_{i+1} = y_i + \Delta t f(y_i, t_i).$$

This iterative strategy for solving ODEs is known as **Euler's method**, which will in fact yield the correct solution $y(t)$ as $\Delta t \to 0$. But what sort of error will be added at each step?

Taylor expanding about $t = t_0$, we get (starting with the definition of Taylor series)

$$\begin{aligned} y(t) &= \sum_{n=0}^{\infty} \frac{y^{(n)}(t_0)}{n!}(t-t_0)^n \\ \implies y(t_0+\Delta t) &= \sum_{n=0}^{\infty} \frac{y^{(n)}(t_0)}{n!}(\Delta t)^n \\ &= y(t_0) + \Delta t \frac{y'(t_0)}{1!} + (\Delta t)^2 \frac{y''(t_0)}{2!} + ... \\ \implies y(t_0+\Delta t) &= y(t_0) + \Delta t \frac{dy(t_0)}{dt} + \mathcal{O}\left((\Delta t)^2\right). \end{aligned}$$

Thus the error term goes like the second derivative of $y(t_0)$ times $(\Delta t)^2$. Note that this is the error on a single iteration, also known as the **local truncation error**.

This error is not particularly useful, as we normally need to solve the ODE from t_0 to some *fixed* T, which will be many iterations away from t_0. To reach $t = T$ at a given step size Δt, we will need to step forward a total of $(T - t_0)/(\Delta t)$ iterations. Thus the **total accumulated error** associated with truncating the Taylor expansion goes like the error of a single iteration, $(\Delta t)^2$, multiplied by the total number of iterations, which is proportional to $1/(\Delta t)$. Thus the total accumulated error is proportional to Δt, which is why we typically refer to Euler's method as a *first-order method.*

As you might imagine we can reduce the truncation error by simply including more terms from the function's Taylor expansion. Let's now work out a method for solving ODEs that with total accumulated error proportional to $(\Delta t)^2$.

12.2 Explicit Runge-Kutta Second-Order Methods

Source: `http://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node5.html`

Runge-Kutta Second-Order methods are a type of second-order **predictor-corrector** method. Such methods perform an estimate step from iteration i to $i+1$, using e.g., Euler's method, to get a prediction of the solution at step $i+1$. This is the "predictor" step. Then it uses this prediction to perform at least one more "corrector" step, which increases the accuracy of the solution.

Let's return to the simple ODE:

$$y'(t) = f(y,t).$$

A general two-step Runge-Kutta method is as follows,

$$\begin{aligned} k_1 &= \Delta t f(y_i, t_i) \\ k_2 &= \Delta t f(y_i + \beta k_1, t_i + \alpha \Delta t) \end{aligned}$$

where k_1 is the predicted value of the solution (simply the Euler step), k_2 is the corrected value of the solution, and α and β are unknowns that must be chosen such that the method is locally third-order convergent in Δt. Then to get our solution at the next iteration, we simply sum a linear combination of the k_1 and k_2:

$$y_{i+1} = y_i + ak_1 + bk_2 + \mathcal{O}\left((\Delta t)^3\right),$$

where a and b are additional unknowns that must be chosen so that the method is locally third-order convergent in Δt.

Our goal now is to find these values of a, b, α, and β so that our scheme possesses local truncation error that converges to zero at third order in Δt (i.e., local truncation error proportional to $(\Delta t)^3$).

We proceed by relating $y(t_i)$, the solution at some step t_i, to the solution at the next step $t_{i+1} = t_i + \Delta t$ using the Taylor expansion of $y(t_i + \Delta t)$ to third order in Δt:

$$y(t_i + \Delta t) = y(t_i) + \Delta t y'(t_i) + \frac{(\Delta t)^2}{2!} y''(t_i) + \mathcal{O}\left((\Delta t)^3\right)$$

Note that

$$y''(t_i) = \frac{df(y_i, t_i)}{dt}$$

Using the chain rule for partial derivatives, we get

$$\begin{aligned} y''(t_i) &= \frac{\partial f(y_i, t_i)}{\partial t} + \frac{\partial f(y_i, t_i)}{\partial y}\frac{dy}{dt} \\ &= \partial_t f(y_i, t_i) + f(y_i, t_i)\partial_y f(y_i, t_i) \end{aligned}$$

Thus we get:

$$y(t_i + \Delta t) = y(t_i) + \Delta t y'(t_i) + \frac{(\Delta t)^2}{2!}\left(\partial_t f(y_i, t_i) + f(y_i, t_i)\partial_y f(y_i, t_i)\right) + \mathcal{O}\left((\Delta t)^3\right).$$

Let's simplify the notation a bit

$$y_{i+1} = y_i + \Delta t f_i + \frac{(\Delta t)^2}{2!}\left([\partial_t f]_i + f_i[\partial_y f]_i\right) + \mathcal{O}\left((\Delta t)^3\right),$$

where we have defined $f_i = f(y_i, t_i)$, and the partial derivatives of f with respect to variable z at $(y,t) = (y_i, t_i)$ as $[\partial_z f]_i = [\partial_z f]_{y=y_i,\ t=t_i}$.

Let's return to the definition of k_2, which can be Taylor expanded as well (using the standard Taylor series expression for a function of two variables), at iteration n:

$$\begin{aligned} k_2 &= \Delta t f(y_i + \beta k_1, t_i + \alpha\Delta t) \\ &= \Delta t\left([f(y,t)]_i + \alpha\Delta t[\partial_t f]_i + \beta k_1[\partial_y f]_i\right) + \mathcal{O}\left((\Delta t)^3\right) \end{aligned}$$

Let's substitute all of this back into the expression for y_{i+1}. For notational simplicity, let's agree that $f = f(y_i, t_i)$:

$$\begin{aligned} y_{i+1} &= y_i + ak_1 + bk_2 + \mathcal{O}\left((\Delta t)^3\right) \\ &= y_i + a\Delta t f + b\Delta t\left(f + \alpha\Delta t\partial_t f + \beta k_1\partial_y f\right) + \mathcal{O}\left((\Delta t)^3\right) \\ &= y_i + a\Delta t f + b\Delta t\left(f + \alpha\Delta t\partial_t f + \beta\Delta t f\partial_y f\right) + \mathcal{O}\left((\Delta t)^3\right) \\ &= y_i + (a+b)\Delta t f + b(\Delta t)^2\left(\alpha\partial_t f + \beta f\partial_y f\right) + \mathcal{O}\left((\Delta t)^3\right) \end{aligned}$$

We can compare this to the Taylor expansion of y_{i+1} to obtain values for our coefficients:

$$\begin{aligned} a + b &= 1 \\ \alpha b &= \frac{1}{2} \\ \beta b &= \frac{1}{2} \end{aligned}$$

Notice we have 3 equations and 4 unknowns! This means there are infinitely many ways we could write this stepping to give us a second-order accurate scheme, so let's choose to write the most general scheme with free parameter α:

$$\begin{aligned} \alpha b = \frac{1}{2} \text{ and } \beta b = \frac{1}{2} \quad &\Longrightarrow \quad \alpha = \beta \\ &\Longrightarrow \quad b = \frac{1}{2\alpha} \quad (\alpha \neq 0) \\ a + b = 1 \quad &\Longrightarrow \quad a = 1 - \frac{1}{2\alpha}. \end{aligned}$$

Thus the generic, second-order Runge-Kutta method may be written as

$$\begin{aligned} k_1 &= \Delta t f(y_i, t_i) \\ k_2 &= \Delta t f(y_i + \alpha k_1, t_i + \alpha\Delta t) \\ y_{i+1} &= y_i + \left(1 - \frac{1}{2\alpha}\right)k_1 + \frac{1}{2\alpha}k_2 + \mathcal{O}\left((\Delta t)^3\right), \end{aligned}$$

for any $\alpha \neq 0$. *Open exercise: What does it mean if $\alpha < 0$?*

Picking the following specific values for α results in the corresponding methods listed below:

- $\alpha = \frac{1}{2}$: The **midpoint method**.
- $\alpha = 1$: **Heun's method**.
- $\alpha = \frac{2}{3}$: The **Ralston method**. This method was derived with a goal of minimizing the error associated with the $\mathcal{O}\left((\Delta t)^3\right)$ term. We will explore the impact of this choice below in Example #2.

12.2.1 Moving from t_0 to t_1: Worked Examples of Heun's Method

Example #1

Consider the ODE

$$y'(t) = -ty(t), \quad y(0) = y(t_0) = 1.$$

- Solve this ODE exactly, then Taylor expand the solution about $t = 0$ to approximate the solution at $y(t = \Delta t)$ to fifth order in Δt.
- Next solve this ODE using the second-order-accurate Heun's method *by hand* with a step size of Δt to find $y(\Delta t)$. Confirm that the solution obtained when using Heun's method has an error term that is at worst $\mathcal{O}\left((\Delta t)^3\right)$. If the dominant error is proportional to a higher power of Δt, explain the discrepancy.

This ODE is separable, and has the solution of a normal, Gaussian distribution:

$$y(t) = ce^{-t^2/2}$$

Consider initial conditions $y(0) = 1$ Then the solution is

$$y(t) = e^{-t^2/2}$$

We know $y(0) = y(t_0) = y_0 = 1$, so let's now find $y_1 = y(\Delta t)$ using Heun's method (i.e., the generic RK2 method with $\alpha = 1$):

$$\begin{aligned} k_1 &= \Delta t f(y_n, t_n) \\ k_2 &= \Delta t f(y_n + k_1, t_n + \Delta t)) \\ y_{n+1} &= y_n + \frac{1}{2}(k_1 + k_2) + \mathcal{O}\left((\Delta t)^3\right). \end{aligned}$$

Here, $f(y, t) = -ty$, so

$$\begin{aligned} k_1 &= \Delta t f(y_0, t_0) \\ &= \Delta t \times 0 \\ &= 0 \\ k_2 &= \Delta t f(y_0 + k_1, 0 + \Delta t) \\ &= \Delta t f(y_0 + 0, 0 + \Delta t) \\ &= \Delta t(y_0)(-\Delta t) \\ &= -\Delta t^2 \\ y(\Delta t) &= y(0) + \frac{1}{2}(k_1 + k_2) \\ &= 1 - (\Delta t)^2/2 \end{aligned}$$

Notice that the exact solution is

$$y(\Delta t) = e^{-(\Delta t)^2/2} = 1 - (\Delta t)^2/2 + (\Delta t)^4/8 + \mathcal{O}\left((\Delta t)^6\right)$$

Thus Heun's method yields the correct solution at the end of the first iteration to *fourth order* in the error, in this case. This is in part because Heun's method preserves the even-ness of the function (i.e., $f(-t) = f(t)$). In general you should only expect the solution at the end of the first iteration to be accurate to third-order in the error.

Example #2

Consider the ODE

$$y' = y - 2te^{-2t}, \quad y(0) = y(t_0) = 0.$$

- Solve this ODE exactly, then Taylor expand the solution about $t = 0$ to approximate the solution at $y(t = \Delta t)$ to fifth order in Δt.
- Next solve this ODE using Heun's method *by hand* with a step size of Δt to find $y(\Delta t)$. Confirm that the solution obtained when using Heun's method has an error term that is at worst $\mathcal{O}\left((\Delta t)^3\right)$. If the dominant error is proportional to a higher power of Δt, explain the discrepancy.
- Finally solve this ODE using the Ralston method *by hand* with a step size of Δt to find $y(\Delta t)$. Is the coefficient on the dominant error term closer to the exact solution than Heun's method?

We can solve this equation via the method of integrating factors, which states that ODEs of the form:

$$y'(t) + p(t)y(t) = g(t)$$

are solved via

$$y(t) = \frac{1}{\mu(t)}\left[\int \mu(s)g(s)ds + c\right],$$

where the integrating factor $\mu(t)$ is given by

$$\mu(t) = \exp\left(\int p(t)dt\right)$$

Here, $p(t) = -1$ and $g(t) = -2te^{-2t}$. Then

$$\mu(t) = \exp\left(-\int dt\right) = e^{-t+c} = ke^{-t}$$

and

$$\begin{aligned} y(t) &= e^t/k\left[\int ke^{-s}(-2se^{-2s})ds + c\right] = -2e^t\left[\int se^{-3s}ds + c'\right] \\ &= -2e^t\left[e^{-3t}\left(-\frac{t}{3} - \frac{1}{9}\right) + c'\right] = -2e^{-2t}\left(-\frac{t}{3} - \frac{1}{9}\right) - 2c'e^t \\ &= e^{-2t}\left(2\frac{t}{3} + \frac{2}{9}\right) + c''e^t \end{aligned}$$

If $y(0) = 0$ then we can compute the integration constant c'', and $y(t)$ becomes

$$y(t) = \frac{2}{9}e^{-2t}\left(3t + 1 - e^{3t}\right).$$

The Taylor Series expansion of the exact solution about $t = 0$ evaluated at $y(\Delta t)$ yields

$$y(\Delta t) = -(\Delta t)^2 + (\Delta t)^3 - \frac{3(\Delta t)^4}{4} + \frac{23(\Delta t)^5}{60} - \frac{19(\Delta t)^6}{120} + O\left((\Delta t)^7\right).$$

Next we evaluate $y(\Delta t)$ using Heun's method. We know $y(0) = y_0 = 0$ and $f(y,t) = y - 2te^{-2t}$, so

$$\begin{aligned}
k_1 &= \Delta t f(y(0), 0) \\
&= \Delta t \times 0 \\
&= 0 \\
k_2 &= \Delta t f(y(0) + k_1, 0 + \Delta t) \\
&= \Delta t f(y(0) + 0, 0 + \Delta t) \\
&= \Delta t(-2\Delta t e^{-2\Delta t}) \\
&= -2(\Delta t)^2 e^{-2\Delta t} \\
y(\Delta t) &= y_0 + \frac{1}{2}(k_1 + k_2) + \mathcal{O}\left((\Delta t)^3\right) \\
&= 0 - (\Delta t)^2 e^{-2\Delta t} \\
&= -(\Delta t)^2 (1 - 2\Delta t + 2(\Delta t)^2 + ...) \\
&= -(\Delta t)^2 + 2(\Delta t)^3 + \mathcal{O}\left((\Delta t)^4\right).
\end{aligned}$$

Thus the coefficient on the $(\Delta t)^3$ term is wrong, but this is completely consistent with the fact that our stepping scheme is only third-order accurate in Δt.

Let's see if we can improve them with the Ralston method:

$$\begin{aligned}
k_1 &= \Delta t f(y_n, t_n) \\
k_2 &= \Delta t f(y_n + 2k_1/3, t_n + 2\Delta t/3)) \\
y_{n+1} &= y_n + \left(\frac{1}{4}k_1 + \frac{3}{4}k_2\right) + \mathcal{O}\left((\Delta t)^3\right).
\end{aligned}$$

Applying the Ralston method to our ODE, we get:

$$\begin{aligned}
k_1 &= \Delta t f(y(0), 0) \\
&= \Delta t \times 0 \\
&= 0 \\
k_2 &= \Delta t f(y(0) + 2k_1/3, 0 + 2\Delta t/3) \\
&= \Delta t f(y(0) + 0, 0 + 2\Delta t/3) \\
&= \Delta t(-2(2\Delta t/3) e^{-4\Delta t/3}) \\
&= -4/3(\Delta t)^2 e^{-4\Delta t/3} \\
y(\Delta t) &= y_0 + \frac{1}{4}k_1 + \frac{3}{4}k_2 + \mathcal{O}\left((\Delta t)^3\right) \\
&= 0 - (\Delta t)^2 e^{-4\Delta t/3} \\
&= -(\Delta t)^2 \left(1 - \frac{4\Delta t}{3} + \frac{1}{2!}\left(\frac{4\Delta t}{3}\right)^2 + ...\right) \\
&= -(\Delta t)^2 + 4/3(\Delta t)^3 + \mathcal{O}\left((\Delta t)^4\right)
\end{aligned}$$

Notice that 4/3 is closer to the exact Taylor Series expansion coefficient of 1 than the 2 we found with Heun's method. Recall that α in the Ralston method was chosen to minimize truncation error, and indeed the Ralson method possesses lower truncation error than Heun's method in this case.

12.2.2 Demonstration that Total Accumulated Error is $\mathcal{O}\left((\Delta t)^2\right)$ with Heun's Method

Example #3

Consider the simple ODE:

$$y' = -y, \quad y(0) = y_0 = 1.$$

- Solve this ODE exactly, then Taylor expand the solution about $t = 0$ to approximate the solution at $y(t = \Delta t)$, $y(t = 2\Delta t)$, and $y(t = 3\Delta t)$ to third order in Δt (i.e., up to and including the $(\Delta t)^3$ term).
- Next solve this ODE using Heun's method *by hand* with a step size of Δt to find $y(\Delta t)$, $y(2\Delta t)$, and $y(3\Delta t)$.
- Compute the absolute error between the exact solution and the numerical solution for the $(\Delta t)^3$ term. Notice that the constant coefficient increases linearly with the number of steps taken.
- Next suppose we wish to solve the ODE from $t_0 = 0$ to some final t, t_f, taking N steps. Derive an expression relating t_f, N, and Δt to prove that after N steps, the dominant error is proportional to $(\Delta t)^2$ and not $(\Delta t)^3$.

This is an exact equation, with solution $y = e^{-t}$. The Taylor expansion about $t = \Delta t$ is given by

$$y(\Delta t) = y_1 = 1 - \Delta t + \frac{\Delta t^2}{2!} - \frac{\Delta t^3}{3!} + \mathcal{O}\left((\Delta t)^4\right)$$

Similarly, the exact solution at $t = 2\Delta t$ is given by the Taylor expansion:

$$\begin{aligned} y(2\Delta t) &= 1 - 2\Delta t + \frac{(2\Delta t)^2}{2!} - \frac{(2\Delta t)^3}{3!} + ... \\ &= 1 - 2\Delta t + 2(\Delta t)^2 - \frac{8}{6}(\Delta t)^3 + ... \end{aligned}$$

Finally, the exact solution at $t = 3\Delta t$ is given by the Taylor expansion:

$$\begin{aligned} y(3\Delta t) &= 1 - 3\Delta t + \frac{(3\Delta t)^2}{2!} - \frac{(3\Delta t)^3}{3!} + ... \\ &= 1 - 3\Delta t + \frac{3}{4}(\Delta t)^2 - \frac{9}{2}(\Delta t)^3 + ... \end{aligned}$$

Next we focus on Heun's method. For $y_1 = y(\Delta t)$, this method yields:

$$\begin{aligned} k_1 &= \Delta t f(y_0, t_0) \\ &= -\Delta t \times 1 \\ &= -\Delta t \\ k_2 &= \Delta t f(y_0 + k_1, t_0 + \Delta t) \\ &= \Delta t f(1 - \Delta t, 0 + \Delta t) \\ &= -\Delta t(1 - \Delta t) = -\Delta t + (\Delta t)^2 \\ y_1 &= y_0 + \frac{1}{2}(k_1 + k_2) + \mathcal{O}\left((\Delta t)^3\right) \\ &= 1 + \frac{1}{2}(-\Delta t - \Delta t + (\Delta t)^2) \\ &= 1 - \Delta t + \frac{(\Delta t)^2}{2}, \end{aligned}$$

which indeed matches the exact solution up to and including the second-order term in the Taylor expansion. Thus the absolute error of the dominant error term at the end of iteration 1 is

$$E_1^{\text{dom}} = |\text{num} - \text{exact}| = \frac{1}{6}(\Delta t)^3.$$

Let's now compute y_2:

$$\begin{aligned}
k_1 &= \Delta t f(y_1, t_1) \\
&= -\Delta t \left(1 - \Delta t + \frac{(\Delta t)^2}{2}\right) \\
&= -\Delta t + (\Delta t)^2 - \frac{(\Delta t)^3}{2} \\
k_2 &= \Delta t f(y_1 + k_1, t_1 + \Delta t) \\
&= -\Delta t \left(1 - \Delta t + \frac{(\Delta t)^2}{2} - \Delta t + (\Delta t)^2 - \frac{(\Delta t)^3}{2}\right) \\
&= -\Delta t + 2(\Delta t)^2 - \frac{3}{2}(\Delta t)^3 + \frac{1}{2}(\Delta t)^4 \\
y_2 &= y_1 + \frac{1}{2}(k_1 + k_2) \\
&= 1 - \Delta t + \frac{(\Delta t)^2}{2} + \frac{1}{2}\left(-\Delta t + (\Delta t)^2 - \frac{(\Delta t)^3}{2} - \Delta t + 2(\Delta t)^2 - \frac{3}{2}(\Delta t)^3 + \frac{1}{2}(\Delta t)^4\right) \\
&= 1 - 2\Delta t + 2(\Delta t)^2 - (\Delta t)^3 + \frac{1}{4}(\Delta t)^4.
\end{aligned}$$

Thus the dominant error at iteration 2, which occurs in the 3rd-order term, is

$$E_2^{\text{dom}} = |\text{num} - \text{exact}| = \left|-1 + \frac{8}{6}\right| (\Delta t)^3 = \frac{2}{6}(\Delta t)^3.$$

If you proceed and compute y_3, y_4, etc., you will find that the error in the dominant-order (i.e., 3rd-order in Δt) term at iteration N follows the pattern

$$E_i^{\text{dom}} = \frac{N}{6}(\Delta t)^3.$$

Thus the dominant error is indeed proportional to $N(\Delta t)^3$, so that if we wish to measure the dominant error at some fixed $t = T = N\Delta t$, it will be given by

$$E_i^{\text{dom}}(t) = \frac{1}{6}N(\Delta t)^3 = \frac{T}{6}\frac{(\Delta t)^3}{\Delta t} \propto (\Delta t)^2.$$

So if we increased our sampling rate, going from Δt to $\Delta t/2$, the number of iterations required to evaluate the solution at $t = T$ would double. But we have just shown that the dominant error term at $t = T$ would double as well, meaning that at some fixed $t = T = N\Delta t$, the error E_i^{dom} should grow proportionally to $(\Delta t)^3/(\Delta t) = (\Delta t)^2$.

12.2.3 Additional Example: Heun versus Ralston

Example #4

Consider the simple ODE:

$$y' = y + t, \quad y(0) = y_0 = 1.$$

- Solve this ODE exactly, then Taylor expand the solution about $t = 0$ to approximate the solution at $y(t = \Delta t)$, $y(t = 2\Delta t)$, and $y(t = 3\Delta t)$ to third order in Δt (i.e., up to and including the $(\Delta t)^3$ term).
- Next solve this ODE using Heun's method *by hand* with a step size of Δt to find $y(\Delta t)$.
- Finally solve this ODE using the Ralston method *by hand* with a step size of Δt to find $y(\Delta t)$.
- Explain the similarities or differences between Heun's method and the Ralston method when computing $y(\Delta t)$.

The ODE takes the general form

$$y'(t) + p(t)y(t) = g(t),$$

which is solved via the Method of Integrating Factors:

$$y(t) = \frac{1}{\mu(t)} \int \mu(s)g(s)ds,$$

where the integrating factor $\mu(t)$ is given by

$$\mu(t) = \exp\left(\int p(t)dt\right).$$

Here, $p(t) = -1$ and $g(t) = t$. Then

$$\mu(t) = \exp\left(-\int dt'\right) = e^{-t+c} = ke^{-t}$$

and

$$y(t) = e^t/k \int^t kse^{-s}ds = e^t \int^t se^{-s}ds.$$

Integration by parts is based on the product rule:

$$(u(s)v(s))' = u'v + v'u \implies \int u'vds = \int (uv)'ds - \int v'uds,$$

so defining $u' = e^{-s}$ and $v = s$, we get $u = -e^{-s}$ and $v' = 1$. Thus the solution becomes

$$\begin{aligned} y(t) &= e^t \int^t se^{-s}ds = e^t\left[-te^{-t} - \int^t e^{-s} + C_0\right] \\ &= e^t\left[-te^{-t} - e^{-t} + C\right] = Ce^t - t - 1. \end{aligned}$$

Plugging in our initial condition, $y(t=0) = 1$, we get $y(0) = 1 = C - 1$, so $C = 2$, and our solution is given by

$$y(t) = 2e^t - t - 1.$$

Taylor expanding this about $t = 0$, we get

$$y(\Delta t) = 2\left(1 + \Delta t + \frac{(\Delta t)^2}{2!} + \frac{(\Delta t)^3}{3!} + ...\right) - \Delta t - 1 = 1 + \Delta t + (\Delta t)^2 + \frac{(\Delta t)^3}{3}.$$

Next we apply Heun's method. Recall our ODE implies that $f(y,t) = y + t$, so Heun's method for the zeroth iteration ($i = 0$) gives:

$$\begin{aligned} k_1 &= \Delta t f(y_0, t_0) = \Delta t(y_0 + t_0) = \Delta t \\ k_2 &= \Delta t f(y_0 + k_1, t_0 + \Delta t) = \Delta t\left((y_0 + k_1) + (t_0 + \Delta t)\right) = \Delta t\left((1 + \Delta t) + (0 + \Delta t)\right) = \Delta t + 2(\Delta t)^2 \\ y(\Delta t) = y_1 &= y_0 + \frac{1}{2}(k_1 + k_2) = 1 + \frac{1}{2}\left(\Delta t + \Delta t + 2(\Delta t)^2\right) = 1 + \Delta t + (\Delta t)^2, \end{aligned}$$

which is consistent with the exact solution up to and including the second-order term in powers of Δt!

Next we apply the Ralston method:

$$\begin{aligned} k_1 &= \Delta t f(y_0, t_0) = \Delta t(y_0 + t_0) = \Delta t \\ k_2 &= \Delta t f\left(y_0 + \frac{2}{3}k_1, t_0 + \frac{2}{3}\Delta t\right) = \Delta t\left(\left(y_0 + \frac{2}{3}k_1\right) + \left(t_0 + \frac{2}{3}\Delta t\right)\right) = \Delta t\left(\left(1 + \frac{2}{3}\Delta t\right) + \left(0 + \frac{2}{3}\Delta t\right)\right) \\ &= \Delta t + \frac{4}{3}(\Delta t)^2 \\ y(\Delta t) = y_1 &= y_0 + \frac{1}{4}k_1 + \frac{3}{4}k_2 = 1 + \frac{1}{4}\Delta t + \frac{3}{4}\left(\Delta t + \frac{4}{3}(\Delta t)^2\right) = 1 + \Delta t + (\Delta t)^2, \end{aligned}$$

which is consistent with the exact solution up to and including the second-order term in powers of Δt!

In fact, we have found that for this ODE, both the Ralston and Heun's method yield exactly the same result for $y(\Delta t)$. This contrasts with the earlier example, largely because of terms like $e^{-2\Delta t}$ that appeared due to the explicit e^{-2t} in $f(y,t)$, which we needed to Taylor expand in that example. *Open question: Will both methods be equivalent for all $y(t)$ for $t > \Delta t$ as well?*

12.3 Runge-Kutta Fourth Order (RK4)

Thus far, we have explored first- and second-order Runge-Kutta (RK) methods. These methods guarantee that the total accumulated error will be proportional to $(\Delta t)^1$ and $(\Delta t)^2$, respectively. Recall that the first-order (Euler's) RK method can be derived directly from keeping terms up to and including $(\Delta t)^2$ in the Taylor series expansion of $y(t+\Delta t)$. The generic second-order RK method was then derived by expanding the Taylor series to the next higher order.

A similar procedure can be applied to derive higher-order RK methods. Like the generic second-order RK method, these higher-order methods are not unique.

The most widely known Runge-Kutta technique guarantees total accumulated truncation error proportional to $(\Delta t)^4$ for smooth functions $y(t)$. This technique is known as RK4, or sometimes **the Runge-Kutta method**. RK4 won popularity by being both highly robust and rapidly convergent.

Definition of RK4: Given the generic ODE

$$y'(t) = f(y,t),$$

The RK4 method obtains the solution $y(t+\Delta t) = y_{i+1}$ at time t_{i+1} from y_i and t_i via:

$$\begin{aligned}
k_1 &= \Delta t f(y_i, t_i), \\
k_2 &= \Delta t f(y_i + \tfrac{1}{2}k_1, t_i + \tfrac{\Delta t}{2}), \\
k_3 &= \Delta t f(y_i + \tfrac{1}{2}k_2, t_i + \tfrac{\Delta t}{2}), \\
k_4 &= \Delta t f(y_i + k_3, t_i + \Delta t), \\
y_{i+1} &= y_i + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + \mathcal{O}\left((\Delta t)^5\right).
\end{aligned}$$

Notice that for each timestep, RK4 requires four evaluations of the RHS of the equation. Recall second-order RK methods only required two such evaluations. Thus RK4 will only be superior to second-order methods if you are also able to double your timestep while maintaining at worst the same level of error. This depends on the problem, but usually RK4 is superior to second-order methods in this sense, which is why so many people depend on it. The bottom line is, a higher-order scheme often, but *does not always* yield smaller errors more efficiently than a lower-order scheme.

Now let's return to the first example:

Example #1, Revisited: Validating RK4

Consider the ODE

$$y'(t) = -ty(t) \quad y(0) = 1$$

Again, this ODE is separable, and has the solution of a normal, Gaussian distribution:

$$y(t) = e^{-t^2/2}.$$

- Taylor expand the solution about $t = 0$ to approximate the solution at $y(t = \Delta t)$ up to and including the seventh-order term in Δt.
- Next solve this ODE using RK4 *by hand* with a step size of Δt to find $y(\Delta t)$. Verify that the solution obtained when using RK4 has an error term that is at worst $\mathcal{O}\left((\Delta t)^5\right)$.

The exact solution of $y(\Delta t)$ up to and including the $\mathcal{O}\left(\Delta t^7\right)$ term is given by

$$y(\Delta t) = e^{-(\Delta t)^2/2} = 1 - (\Delta t)^2/2 + (\Delta t)^4/8 - (\Delta t)^6/48 + \mathcal{O}\left(\Delta t^8\right).$$

(Notice that since this is an even function, the $\mathcal{O}\left(\Delta t^7\right)$ is zero.)

We know $y(0) = y(t_0) = y_0 = 1$, so $y(\Delta t)$ using RK4 is as follows. Note that $f(y,t) = -ty$, so

$$\begin{aligned}
k_1 &= \Delta t f(y_0, t_0) = \Delta t f(1,0) \\
&= \Delta t \times (-0 \times y_0) \\
&= 0,
\end{aligned}$$

$$\begin{aligned}
k_2 &= \Delta t f\left(y_0 + \frac{k_1}{2}, t_0 + \frac{\Delta t}{2}\right) \\
&= \Delta t f(y_0 + 0, t_0 + \Delta t/2) \\
&= \Delta t(y_0)(-\Delta t/2) \\
&= -(\Delta t)^2/2,
\end{aligned}$$

$$\begin{aligned}
k_3 &= \Delta t f\left(y_0 + \frac{k_2}{2}, t_0 + \frac{\Delta t}{2}\right) \\
&= \Delta t f(y_0 + (-(\Delta t)^2/2)/2, 0 + \Delta t/2) \\
&= \Delta t f(y = 1 - (\Delta t)^2/4, t = \Delta t/2) \\
&= \Delta t\left[-(1 - (\Delta t)^2/4)(\Delta t/2)\right] \\
&= -(\Delta t)^2/2 + (\Delta t)^4/8,
\end{aligned}$$

and

$$\begin{aligned}
k_4 &= \Delta t f(y_0 + k_3\Delta t, t_0 + \Delta t) \\
&= \Delta t f(1 + (-(\Delta t)^2/2 + (\Delta t)^4/8)\Delta t, 0 + \Delta t) \\
&= \Delta t\left[-(1 + (-(\Delta t)^2/2 + (\Delta t)^4/8)\Delta t)(\Delta t)\right] \\
&= (\Delta t)^2\left[-(1 + (-(\Delta t)^2/2 + (\Delta t)^4/8)\Delta t)\right] \\
&= -(\Delta t)^2 + (\Delta t)^4/2 - (\Delta t)^6/8,
\end{aligned}$$

so that

$$\begin{aligned}
y_1 &= y_0 + \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \\
&= 1 + \tfrac{1}{6}\left[0 + 2(-(\Delta t)^2/2) + 2(-(\Delta t)^2/2 + (\Delta t)^4/8) + (-(\Delta t)^2 + (\Delta t)^4/2 - (\Delta t)^6/8)\right] \\
&= 1 + \tfrac{1}{6}\left[-3(\Delta t)^2 + 2((\Delta t)^4/8) + ((\Delta t)^4/2 - (\Delta t)^6/8)\right] \\
&= 1 + \tfrac{1}{6}\left[-3(\Delta t)^2 + 3(\Delta t)^4/4 - (\Delta t)^6/8\right] \\
&= 1 - \tfrac{1}{2}(\Delta t)^2 + (\Delta t)^4/8 - (\Delta t)^6/48
\end{aligned}$$

Comparing this to the exact solution, we find that although RK4 can only be expected to yield local truncation error proportional to $(\Delta t)^5$, RK4 manages to get the next order term exactly correct in this case!

12.4 The Family of Explicit Runge-Kutta-Like Schemes

"Explicit" Runge-Kutta methods, including all the RK methods discussed in this chapter, can be written:

$$y_{i+1} = y_i + \sum_{\ell=1}^{s} b_\ell k_\ell,$$

where

$$\begin{aligned}
k_1 &= \Delta t f(y_i, t_i) \\
k_2 &= \Delta t f(y_i + [a_{21}k_1], t_i + c_2\Delta t) \\
k_3 &= \Delta t f(y_i + [a_{31}k_1 + a_{32}k_2], t_i + c_3\Delta t) \\
&\vdots \\
k_s &= \Delta t f(y_i + [a_{s1}k_1 + a_{s2}k_2 + \cdots + a_{s,s-1}k_{s-1}], t_i + c_s\Delta t).
\end{aligned}$$

Whereas "explicit" methods generally relate y_{i+1} to possibly complicated functions of y_i, "implicit" methods generally relate y_i to more complicated functions of y_{i+1} requiring a matrix inversion to evaluate y_{i+1}.

Both explicit and implicit methods are usually communicated in the form of so-called "Butcher tableaus", or "Butcher tables". A Butcher table has the general form:

$$
\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \dots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \dots & b_{s-1} & b_s
\end{array}
$$

The Butcher table for Heun's method is:

$$
\begin{array}{c|cc}
0 & & \\
1 & 1 & \\
\hline
 & \frac{1}{2} & \frac{1}{2}
\end{array}
$$

We say that a Runge-Kutta method is consistent if

$$\sum_{j=1}^{i-1} a_{ij} = c_i \text{ for } i = 2, \dots, s$$

Similarly, the Butcher table for the Ralston method is:

$$
\begin{array}{c|cc}
0 & & \\
2/3 & 2/3 & \\
\hline
 & 1/4 & 3/4
\end{array}
$$

Finally, the RK4 scheme's Butcher table is given by:

$$
\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{1}{2} & 0 & \frac{1}{2} & & \\
1 & 0 & 0 & 1 & \\
\hline
 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6}
\end{array}
$$

12.5 Higher-Order ODEs

We can solve higher-order ODEs by applying a trick to reduce them to a set of coupled, first-order ODEs. Consider the ODE

$$y''(t) - ty'(t) + \cos(t)y(t) = 0,$$

with $y'(0) = y(0) = 0$.

Let's define a variable $v(t) = y'(t)$. Then we can rewrite this second order equation as the first-order equations:

$$
\begin{aligned}
y'(t) &= v(t) \\
v'(t) &= tv(t) - \cos(t)y(t),
\end{aligned}
$$

$v(0) = y(0) = 0$.

We can then solve this equation using Euler's method via:

$$
\begin{aligned}
y_{i+1} &= y_i + \Delta t v_i \\
v_{i+1} &= v_i + \Delta t(t_i v_i - \cos(t_i)y_i)
\end{aligned}
$$

Notice that Euler's method generalizes quite simply to coupled first-order ODEs.

Heun's method also easily generalizes:

$$\begin{aligned}
k_1^y &= \Delta t v_i \\
k_1^v &= \Delta t(t_i v_i - \cos(t_i) y_i) \\
k_2^y &= \Delta t(v_i + k_1^v) \\
k_2^v &= \Delta t((t_i + \Delta t)(v_i + k_1^v) - \cos(t_i + \Delta t)(y_i + k_1^y) \\
y_{i+1} &= y_i + \frac{1}{2}(k_1^y + k_2^y) \\
v_{i+1} &= v_i + \frac{1}{2}(k_1^v + k_2^v).
\end{aligned}$$

MATH 521, Fall 2018 Notes

Prof. Zachariah B. Etienne

Chapter 13: Final Exam Review Topics/Example Problems

Your final exam will take place at the sanctioned time (see the Registrar's website or the syllabus) in our usual course meeting room.

Note that this exposition of topics and example problems is *not comprehensive*. However, your Final Exam *will be comprehensive*. In other words, you will be expected to be sufficiently familiar with all topics covered in class, such that you are able to quickly and competently answer questions on these topics.

In addition to example problems below, please study notes, midterms, & homeworks, as problems similar to those you have encountered may appear on the Final Exam. The Additional/Supplementary Reading Material List on the course website may also be useful.

13.1 Short-Answer Problems

You can expect to see short-answer questions on the Final Exam. Your answer to these problems must both fit in the number of sentences specified, and be clear and unambiguous, adopting vocabulary used in class whenever possible.

Short-Answer Exercises

Your answers for the following short-answer problems must be clear and unambiguous, adopting vocabulary used in class whenever possible.

1. When performing a numerical integration over a fixed interval in the x-direction using the Extended Trapezoidal Rule, you find that whether the sample rate is set to $\Delta x/2$, Δx, or $2\Delta x$, you consistently find the relative error in the result to be of order `1e-16`. In one or two sentences, explain the source of this error.
2. Many methods in numerical analysis guarantee convergence of errors to zero provided certain conditions are met. In two or three sentences, describe under what circumstances **undersampling error** will prevent numerical errors from converging to zero, and provide two examples of numerical methods that can be influenced by undersampling error.
3. To solve a math problem on the computer, you implement a numerical analysis algorithm with computational complexity $\mathcal{O}(N^3)$. The algorithm takes too long to run, so you implement an alternative algorithm with computational complexity $\mathcal{O}(N^2)$. Frustratingly, the $\mathcal{O}(N^2)$ algorithm is slower than the $\mathcal{O}(N^3)$ algorithm! Assuming that you programmed both algorithms correctly and as efficiently as possible, in up to two sentences explain why the $\mathcal{O}(N^3)$ algorithm can be faster.

Solutions

1. As the relative error is of order `1e-16`, the *SDA* between the numerical and exact results is approximately 16, corresponding to the maximum number of significant decimal digits storable in double-precision arithmetic. Thus our results from the numerical integration are dominated by roundoff error.
2. Many methods in numerical analysis guarantee convergence of errors to zero with an increased sampling of some function (e.g., numerical integration, solving ODEs, interpolation etc.). For such convergence, these methods require the function be sampled at or above some minimum sampling rate, known as the Nyquist sampling rate. If the function is sampled below this rate, undersampling error will dominate and convergence may not be observed in the numerical method.
3. The total computational cost (in FLOPs) of an algorithm with $\mathcal{O}(N^3)$ complexity might be nN^3, and the cost of an algorithm with $\mathcal{O}(N^2)$ complexity might be mN^2. If $m \gg n$ and N is sufficiently small, then nN^3 may be smaller than mN^2.

13.2 Scientific Notation, Significant Digits, and Relative Error

Be sure to study Midterm Exams, homeworks, and notes. There are additional practice scientific notation problems linked to from the course homepage that you might find useful: simply click the "Suggested/Additional Reading" link.

13.3 Number Storage and Arithmetic on the Computer

In addition to Midterm Exams, homeworks, and notes, you may find the following exercises useful.

Double Precision Arithmetic Exercises

When the following expressions are evaluated by the computer, **to how many significant decimal digits will the numerical result agree with the exact result?** Your answer will consist of a single integer (∞ is an acceptable answer). If the integer is finite and nonzero, your answer will be accepted if it is within 1 decimal digit of the exact answer.
We use computer scientific notation, such that, e.g., `5.63e22` $\equiv 5.63 \times 10^{22}$. For the purpose of this problem, apply IEEE 754 standard-compliant **double precision** arithmetic, assuming all given *integers* represented in integer or floating point format between -2^{53} and 2^{53} (i.e., integers between $\approx -9 \times 10^{15}$ and $\approx 9 \times 10^{15}$ inclusive) are **exact** (for example, `2.01e2` is exact), as well as powers of $\frac{1}{2}$. Otherwise, assume that in double precision the number is only known to 16 significant digits, and that machine epsilon is `4e-16`.
Answers will receive full credit so long as they are correct within 1 decimal digit. You will not receive credit for showing work.

1. `7 + 2.4e-230/5e280`
2. `7 + 2.4e230/5e280`
3. `1/100`
4. `log10(2e-230) - 2e-200`
5. `(3.1415-3.14)/(3.1415-3.14)`
6. `1/1024`
7. `3.2e-200/5e-202 + 2`

Solutions

1. `7 + 2.4e-230/5e280`: `7` is exactly representable, and the expression `2.4e-230/5e280` will yield an underflow in double precision, giving a double precision result of `7` *exactly*. The exact result is `7` plus a perturbation at the 511th digit. Thus the double precision and exact results agree to about 511 significant digits.
2. `7 + 2.4e230/5e280`: `7` is exactly representable, and the expression `2.4e230/5e280` will yield a number that is of order `1e-51`. Recall that machine epsilon is defined as the largest ϵ such that $1 + \epsilon = 1$. In double precision machine epsilon is approximately `4e-16`, so `7+1e-51 ~ 7 (1 + 1e-52)` is `7` exactly. Thus the double precision and exact results agree to about 51 significant digits.
3. `1/100`: The division is not a power of `2`, so the answer is 16 significant digits of agreement between double precision and the exact answer.
4. `log10(2e-230) - 2e-200`: `log10(2e-230)` is computed via Taylor series, thus will yield a double precision result that matches the exact result to 16 significant digits. The `-2e-200` will not affect the result. The answer is 16.
5. `(3.1415-3.14)/(3.1415-3.14)`: The numerator and denominator will evaluate to exactly the same bits, despite being consistent with the exact result to 16 significant digits. Thus the answer in double precision is 1 exactly, matching the exact result to ∞ significant digits.
6. `1/1024`: $1024 = 2^{10}$, so the expression is 2^{-10}, an exactly representable number in double precision that matches the exact result to ∞ significant digits.
7. `3.2e-200/5e-202 + 2`: `3.2e-200/5e-202` will be evaluated to a precision of 16 significant digits and is of order `1e2`. Thus `2` will perturb it at the second significant digit. The result is *not* a number exactly representable in double precision, so will be consistent with the exact solution to 16 significant digits.

13.4 Determining the Scale of the Solution

Be able to combine estimates in "word problem" format to estimate the scale of a solution to a problem. Study course notes, Midterm 1, and homeworks. Such problems on the Final Exam will not require that you base any estimates on your own experiences. Rather, all numbers required to solve problems like these will be provided; your task will be to synthesize any given numbers into a final answer.

13.5 Computational Cost and Computational Complexity; Big-$\mathcal{O}$ Notation

In addition to Midterm Exams, homeworks, and notes, you may find the following exercises useful.

Exercises

Write the computational complexity of the following two algorithms using Big-$\mathcal{O}$ notation. You are to assume that N is large, as we are only using this notation to estimate how much longer it will take for a given computer to evaluate the given algorithms as N increases. For example, for problem 0 below, the correct answer is $\mathcal{O}(N^2)$.

0.
```
f=N*N
do i=1,N
   do j=1,N
      f = f + N*N
   end do
end do
```
1.
```
f=N*N
do i=1,N*N*N
   f = f + i*i
end do
```
2.
```
f=N*N*N
do i=1,N
  do j=1,sqrt(N) # Assume sqrt(N) is an integer
    f = f + i*i
  end do
end do
```
3.
```
f=N*N*N
do i=1,N
  do j=i,i+1
    do k=1,j
      f = f + i*i + j + k
    end do
  end do
end do
```
4.
```
f=N*N*N
do i=1,N
  do j=-N,N
    if(i+1 is equal to j)
      f = f + i*i
    end if
  end do
end do
```

Solutions

1. $\mathcal{O}(N^3)$
2. $\mathcal{O}(N^2)$, since the j loop only covers two elements at a time, and each of the nested i and k loops individually have complexity $\mathcal{O}(N)$.
3. $\mathcal{O}(N^{3/2})$
4. $\mathcal{O}(N)$, as $i+1=j$ a total of $\mathcal{O}(N)$ times.

13.6 Basic Coding Knowledge

Be sure to study Midterm Exams, homeworks, and course notes. The following example problems may also be helpful to your preparation.

Exercises

What is the output from the `print` statements in each of the following pseudocode snippets?

1.
```
result=0
N=4
do i=1,N
  do j=1,sqrt(N)
    result = result + 1
    print i,j,result
  end do
end do
```
2.
```
result=0
N=4
do i=1,N
  result = result * 2
  print i,result
end do
```
3.
```
result=5
N=4
do i=1,N
  result = result - 1
  print i,result
end do
```

Solutions

```
1. 1 1 1
   1 2 2
   2 1 3
   2 2 4
   3 1 5
   3 2 6
   4 1 7
   4 2 8
2. 1 0
   2 0
   3 0
   4 0
3. 1 4
   2 3
   3 2
   4 1
```

13.7 Numerical Linear Algebra: Solving Square Matrix Equations on the Computer

Must be familiar with basic algorithms for solving $\boldsymbol{Ax} = \boldsymbol{b}$ analyzed in class. Study Midterm 1, homeworks, as well as your notes.

13.8 Function Approximation: Taylor Series

In addition to Midterm Exams, homeworks, and notes, you may find the following exercises useful.

If asked to compute a Taylor series, you will be given the formula for the Taylor series:

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

Taylor Series Exercise #1:

Expand the following function as Taylor series about $x = x_0$, up to and including the second derivative term:

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

Solution

The definition of Taylor series requires that we evaluate derivatives of the underlying function, and we are asked to evaluate the Taylor series up to and including the second derivative term. Let's first evaluate these derivatives:

$$\begin{aligned} f(x) &= e^{\sin(x)} \\ f'(x) &= e^{\sin(x)}\cos(x) \\ f''(x) &= e^{\sin(x)}\cos^2(x) - e^{\sin(x)}\sin(x) \end{aligned}$$

Thus the Taylor series is given by

$$\begin{aligned} f(x) = e^{\sin x} &= \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)(x-x_0)^n}{n!} \\ &= e^{\sin(x_0)} + (x-x_0)e^{\sin(x_0)}\cos(x_0) - \frac{1}{2}(x-x_0)^2\left(e^{\sin(x_0)}\left(\sin(x_0) - \cos^2(x_0)\right)\right) + \mathcal{O}\left((x-x_0)^3\right) \end{aligned}$$

Taylor Series Exercise #2:

Expand the following function as Taylor series about $x = x_0$, up to and including the second derivative term:

$$f(x) = \ln(\ln(x^3)), \quad \text{Assume that } x_0 > 1.$$

Solution

First we evaluate derivatives of the function:

$$\begin{aligned} f(x) &= \ln(\ln(x^3)) \\ f'(x) &= \frac{3}{x \ln(x^3)} \\ f''(x) &= -\frac{9}{x^2 \ln^2(x^3)} - \frac{3}{x^2 \ln(x^3)} \end{aligned}$$

Thus the Taylor series up to and including the second-derivative term is given by

$$\begin{aligned} f(x) = \ln(\ln(x^3)) &= \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)(x-x_0)^n}{n!} \\ &= \ln\left(\ln\left(x_0^3\right)\right) + \frac{3(x-x_0)}{x_0 \ln\left(x_0^3\right)} - \frac{3(x-x_0)^2\left(\ln\left(x_0^3\right)+3\right)}{2\left(x_0^2 \ln^2\left(x_0^3\right)\right)} + \mathcal{O}\left((x-x_0)^3\right) \end{aligned}$$

Taylor Series Exercise #3:

Use the Ratio Test to determine the radius of convergence of the Taylor expansion of $\ln(x)$ about $x_0 = 1$.

Solution

We start by computing the derivatives of $\ln(x)$ evaluated at $x_0 = 1$:

$$\begin{aligned} f^{(0)}(1) &= \ln(1) = 0 \\ f^{(1)}(1) &= 1/1 \\ f^{(2)}(1) &= -1/1^2 \\ f^{(3)}(1) &= 2/1^3 \\ f^{(4)}(1) &= -3!/1^4 \\ f^{(5)}(1) &= 4!/1^5 \\ f^{(6)}(1) &= -5!/1^6 \\ f^{(7)}(1) &= 6!/1^7 \\ \vdots & \quad \vdots \\ f^{(n)}(1) &= \begin{cases} 0 & \text{if } n = 0 \\ (-1)^{n-1}(n-1)! & \text{if } n > 0. \end{cases} \end{aligned}$$

Thus the general expression for the Taylor series of $f(x) = \ln(x)$ about $x_0 = 1$ may be written:

$$\begin{aligned} f(x) = \ln(x) &= \sum_{n=1}^{\infty} \frac{(x-1)^n(-1)^{n-1}(n-1)!}{n!} \\ &= \sum_{n=1}^{\infty} \frac{(x-1)^n(-1)^n(-1)^{-1}}{n} \\ &= -\sum_{n=1}^{\infty} \frac{(1-x)^n}{n} \end{aligned}$$

The Ratio Test yields

$$\lim_{n\to\infty}\left|\frac{b_{n+1}}{b_n}\right| = \lim_{n\to\infty}\left|\frac{(1-x)^{n+1}n}{(1-x)^n(n+1)}\right| = |1-x| \lim_{n\to\infty}\frac{n}{(n+1)} = |1-x| = |1-x| = L.$$

This means that $L < 1$ (absolute convergence) when $-1 < 1 - x < 1$. Thus $-2 < -x < 0 \implies 0 < x < 2$, so the radius of convergence is 1.

13.9 Function Approximation: Fourier Series

In addition to example problems below, please study notes, midterms, & homeworks, as problems similar to those you have encountered are fair game for the Final Exam. Additional/Supplementary Reading Material List on the course website may also be useful.

Fourier Series Example #1

Consider the function $g(x) = \sin(x^2)$.
In this problem, you will compute the coefficients of the Fourier series of $g(x)$ in the interval $x \in [-\pi/2, \pi/2)$. Recall the formula for the Fourier series of a function $f(x)$ defined in the interval $x \in [-L, L)$ is given by

$$f(x) = \frac{a_0}{2} + \sum_{m=1}^{\infty} \left[a_m \cos\left(\frac{m\pi x}{L}\right) + b_m \sin\left(\frac{m\pi x}{L}\right) \right].$$

You are to evaluate all coefficients a_m and b_m in the Fourier series, where

$$\begin{aligned} a_0 &= \frac{1}{L}\int_{-L}^{L} f(x)dx \\ a_m &= \frac{1}{L}\int_{-L}^{L} f(x)\cos\left(\frac{m\pi x}{L}\right)dx \\ b_m &= \frac{1}{L}\int_{-L}^{L} f(x)\sin\left(\frac{m\pi x}{L}\right)dx. \end{aligned}$$

You may find the following integrals useful in computing the Fourier coefficients (as is common in many integral tables, the integration constant is left out).

$$\begin{aligned} 2^{3/2}\frac{1}{\sqrt{\pi}}\int \sin(x^2)\cos(ax)dx &= \cos\left(\frac{a^2}{4}\right)\left(S\left(\frac{2x-a}{\sqrt{2\pi}}\right) + S\left(\frac{2x+a}{\sqrt{2\pi}}\right)\right) \\ &\quad -\sin\left(\frac{a^2}{4}\right)\left(C\left(\frac{2x-a}{\sqrt{2\pi}}\right) + C\left(\frac{2x+a}{\sqrt{2\pi}}\right)\right) \\ 2^{3/2}\frac{1}{\sqrt{\pi}}\int \sin(x^2)\sin(ax)dx &= \cos\left(\frac{a^2}{4}\right)C\left(\frac{2x-a}{\sqrt{2\pi}}\right) - \cos\left(\frac{a^2}{4}\right)C\left(\frac{2x+a}{\sqrt{2\pi}}\right) \\ &\quad \sin\left(\frac{a^2}{4}\right)\left(S\left(\frac{2x-a}{\sqrt{2\pi}}\right) - S\left(\frac{2x+a}{\sqrt{2\pi}}\right)\right), \end{aligned}$$

where

$$S(x) = \int_0^x \sin(t^2)\,dt, \quad C(x) = \int_0^x \cos(t^2)\,dt,$$

are the *Fresnel integrals*, which are transcendental functions.
Write all nonzero Fourier coefficients in terms of the Fresnel integrals evaluated at the appropriate endpoints, using the above integral expressions as needed.

Solution

$g(x) = \sin(x^2)$ is an even function, so we immediately know that all the b_m coefficients must be zero. Thus we only need to compute all a_m coefficients.
First we evaluate a_0. Since $L = \pi/2$ and $g(x)$ is even, we have

$$a_0 = \frac{2}{\pi} 2S(\pi/2) = \frac{4}{\pi} S(\pi/2).$$

Next we evaluate a_m. Since $L = \pi/2$, the definition gives us

$$a_m = \frac{2}{\pi} \int_{-\pi/2}^{\pi/2} \sin(x^2)\cos(2mx)dx.$$

The integrand is an even function, so

$$a_m = \frac{4}{\pi} \int_{0}^{\pi/2} \sin(x^2)\cos(2mx)dx.$$

Comparing this expression with the one in the integral table gives us $a = 2m$. Thus we get

$$\begin{aligned} a_m &= \frac{4}{\pi} \int_0^{\pi/2} \sin(x^2)\cos(2mx)dx \\ &= \frac{4}{\pi}\frac{\sqrt{\pi}}{2^{3/2}}\left[\cos\left(\frac{(2m)^2}{4}\right)\left(S\left(\frac{2x-(2m)}{\sqrt{2\pi}}\right) + S\left(\frac{(2m)+2x}{\sqrt{2\pi}}\right)\right)\right. \\ &\qquad \left. -\sin\left(\frac{(2m)^2}{4}\right)\left(C\left(\frac{2x-(2m)}{\sqrt{2\pi}}\right) + C\left(\frac{(2m)+2x}{\sqrt{2\pi}}\right)\right)\right]_0^{\pi/2}. \end{aligned}$$

Notice that, with the exception of the constant out front simplifying to $\sqrt{2/\pi}$, the expression does not simplify further when $x = 0$ or $x = \pi/2$, so this is our final answer.

Fourier Series Example #2

To greatly simplify integrals, particularly in the context of Fourier series, it is critically important to identify even and odd functions. **In your test booklet**, label the following functions as **even**, **odd**, or **neither**. Recall that an even function $f(x)$ satisfies $f(-x) = f(x)$ and an odd function $f(x)$ satisfies $f(-x) = -f(x)$. You will receive partial credit for identifying important pieces of the function as even, odd, or neither.

1. $f(x) = xe^{\sin^2(x)}$
2. $f(x) = e^x - e^{-x}$
3. $f(x) = e^x + e^{-x}$
4. $f(x) = (e^x)^2 - (e^{-x})^2$
5. $f(x) = \sin(\cos(e^x))$

Solutions

1. $f(x) = xe^{\sin^2(x)} \leftarrow$ odd
2. $f(x) = e^x - e^{-x} = 2\sinh(x) \leftarrow$ odd
3. $f(x) = e^x + e^{-x} = 2\cosh(x) \leftarrow$ even
4. $f(x) = (e^x)^2 - (e^{-x})^2 = e^{2x} - e^{-2x} = 2\sinh(2x) \leftarrow$ odd
5. $f(x) = \sin(\cos(e^x)) \leftarrow$ neither

Fourier Series Example #3

To what value will the Fourier series of the following functions converge at the given points?

1. $f(x) = x$, Fourier series in interval $x \in [-1, 5]$.
   (a) $f(-1) =?$
   (b) $f(5) =?$
   (c) $f(10) =?$
2. $f(x) = \log_{10}(x)$, Fourier series in interval $x \in [0.1, 10.1]$. Your answer may not include logarithms unless otherwise specified.
   (a) $f(0) =?$
   (b) $f(-9.9) =?$ (Your answer may include logarithms.)
   (c) $f(10) =?$
   (d) $f(10.1) =?$ (Your answer may include logarithms.)

Solutions

1. $f(x) = x$, Fourier series in interval $x \in [-1, 5]$.
   (a) $f(-1) = \frac{(-1)+5}{2} = 2$
   (b) Since the period is 6 (i.e., the Fourier series representation of the function repeats every 6), $f(5) = f(-1) = 2$
   (c) $f(10) = f(4) = 4$.
2. $f(x) = \log_{10}(x)$, Fourier series in interval $x \in [0.1, 10.1]$. Your answer may not include logarithms unless otherwise specified.
   (a) The function has a period of 10 (i.e., the Fourier series representation of the function repeats every 10) $f(0) = f(10) = \log_{10}(10) = 1$
   (b) $f(-9.9) = f(0.1) = \frac{\log_{10}(0.1)+\log_{10}(10.1)}{2} = \frac{-1+\log_{10}(10.1)}{2}$.
   (c) $f(10) = \log_{10}(10) = 1$.
   (d) $f(10.1) = f(-9.9)$, computed above.

13.10 Polynomial Interpolation and Extrapolation

Study course notes and homeworks.

13.11 Numerical Integration

Study course notes and homeworks.

13.12 Numerical Root Finding and Optimization

In addition to homeworks and course notes, you may find the following exercises useful.

Example Problem #1

After N iterations, you are distance d from a root, where d is 1,000 times machine epsilon. Using the bisection method, how many more iterations will be necessary to guarantee that you will be within machine epsilon of the root? You may assume that the function is such that this is possible, and any answer within 1 iteration of the correct answer will be accepted.

Solution

After N iterations, you are a distance of $1000\epsilon_m$ from the root. Since we are using the bisection method, at iteration $N+1$ we will be guaranteed to be a distance of at most $500\epsilon_m$ from the root. Thus we have

1. 1 more iteration: $500\epsilon_m$
2. 2 more iteration: $250\epsilon_m$
3. 3 more iteration: $125\epsilon_m$
4. 4 more iteration: $\approx 63\epsilon_m$
5. 5 more iteration: $\approx 32\epsilon_m$
6. 6 more iteration: $\approx 16\epsilon_m$
7. 7 more iteration: $\approx 8\epsilon_m$
8. 8 more iteration: $\approx 4\epsilon_m$
9. 9 more iteration: $\approx 2\epsilon_m$
10. 10 more iteration: $\approx 1\epsilon_m$

Thus the answer is 10, and answers of 9 and 11 will also be accepted.

Example Problem #2

Write three unique fixed-point iteration methods for finding a root of the following function $f(x)$: $f(x) = x^3 - e^x + x$.

Solution

Many answers are possible, and here are some examples of correct answers:

1. $0 = x + x^3 - e^x \implies x_{n+1} = \exp(x_n) - x_n^3$
2. $0 = x(x^2+1) - e^x \implies x_{n+1} = \exp(x_n)/(x_n^2+1)$
3. $0 = x^3(1+1/x^2) - e^x \implies x_{n+1} = \Big(\exp(x_n)/(1+x_n^{-2})\Big)^{1/3}$
4. $0 = x^3 + x - e^x \implies x_{n+1} = \ln(x_n^3 + x_n)$

Example Problem #3

Given the initial guess $x_0 = \pi/2$, what is the value x_1 using the Newton-Raphson root finding method for $f(x) = \sin(x)$?

Solution

The Newton-Raphson method finds roots of the smooth function $f(x)$ via the fixed-point iteration method: $x_{n+1} = x_n - f(x_n)/f'(x_n)$. Thus we have

$$x_{n+1} = x_n - \frac{\sin(x_n)}{\cos(x_n)} \implies x_1 = \frac{\pi}{2} - \frac{\sin(\pi/2)}{\cos(\pi/2)} = \frac{\pi}{2} - \frac{1}{0}.$$

Thus the answer is `NaN`. This is not surprising since the slope of $f(x) = \sin(x)$ at $x = \pi/2$ is zero, meaning that the Newton-Raphson method will choose its next guess an infinite distance away.

Example Problem #4

Our derivation of the Golden Section Search algorithm depended sensitively on the fact that

$$\frac{\sqrt{5}-1}{2} = \frac{2}{1+\sqrt{5}}$$

Prove this.

Solution

This was shown in the notes. The basic strategy is to multiply numerator and denominator in the right expression by $(1-\sqrt{5})$ and then simplify to get the expression on the left of the equals sign.

13.13 Ordinary Differential Equations

Be able to solve ODEs using explicit Runge-Kutta methods, and be able to (as discussed in the notes) verify that these methods produce solutions that are convergent to the expected order. You will not need to solve any ODEs by hand, unless they are separable.

In addition to homeworks and course notes, you may find the following exercise useful.

Example Problem

Demonstrate that Heun's method will reproduce the exact solution to

$$y'(t) = te^t,$$

$y(0) = 0$, to the expected order in Δt after one iteration. The exact solution is given by $y(t) = c + te^t - e^t$, where c is an integration constant you must find by applying the initial condition. You must expand this exact solution to the appropriate order Δt and demonstrate its identity with the output from Heun's method after one iteration. (*Hint: What is the expected order of convergence for Heun's method after one iteration?*)

Solution

The exact solution is $y(t) = c + e^t(t-1)$. When we apply the boundary condition, we find that $c = 1$. Thus $y(t) = 1 + e^t(t-1)$. The Taylor series of this function requires that derivatives be evaluated. Since we're comparing with Heun's method, we will need to expand the Taylor series up to and including the second derivative term:

$$\begin{aligned} y(t_0 = 0) &= 0 \\ y'(t_0 = 0) &= [e^t + e^t(t-1)]_{t=0} = [te^t]_{t=0} = 0 \\ y''(t_0 = 0) &= [te^t]'_{t=0} = [e^t + te^t]'_{t=0} = 1 \end{aligned}$$

Thus the Taylor series about $t_0 = 0$ is given by

$$y(t) = \frac{t^2}{2} + \mathcal{O}(\Delta t)^3$$

Heun's method yields

$$\begin{aligned} k_1 &= \Delta t f(y_0, t_0) = \Delta t[te^t]_{y=0,t=0} = 0 \\ k_2 &= \Delta t f(y_0 + k_1, t_0 + \Delta t) = \Delta t[te^t]_{y=0,t=\Delta t} = (\Delta t)^2 e^{\Delta t} \\ y_1 = y(\Delta t) &= y_0 + \frac{1}{2}(k_1 + k_2) + \mathcal{O}\left((\Delta t)^3\right) \\ &= 0 + \frac{1}{2}(\Delta t)^2 e^{\Delta t} + \mathcal{O}\left((\Delta t)^3\right) \\ &= \frac{1}{2}(\Delta t)^2(1 + \Delta t + ...) + \mathcal{O}\left((\Delta t)^3\right) \\ &= \frac{(\Delta t)^2}{2}(\Delta t)^2 + \mathcal{O}\left((\Delta t)^3\right), \end{aligned}$$

which indeed matches the exact solution to second order in Δt.

13.14 Final Exam: Free Problem

Example Problem

Write three unique fixed-point-iteration-without-derivatives algorithms that may conceivably be used to find the real roots of

$$f(x) = x\ln(x) - x^3 - 1.$$

Each answer must be in the form

$$x_{i+1} = g(x_i),$$

where $g(x)$ is the unique function for your given fixed-point iteration algorithm.

Solution

Our goal is to find an iterative method to compute $f(x) = 0$. This will be the case when

$$x\ln(x) - x^3 - 1 = 0,$$

so one iterative method would be solving for x as follows:

$$\begin{aligned} x\ln(x) - x^3 - 1 &= 0 \\ \implies x &= \frac{1+x^3}{\ln(x)} \\ \implies x_{i+1} &= \frac{1+x_i^3}{\ln(x_i)} \end{aligned}$$

Another would involve solving for $\ln(x)$ and then exponentiating both sides:

$$\begin{aligned} x\ln(x) - x^3 - 1 &= 0 \\ \implies \ln(x) &= \frac{1+x^3}{x} \\ \implies x_{i+1} &= \exp\left(\frac{1+x_i^3}{x_i}\right) \end{aligned}$$

And another would involve factoring out an x from $x\ln(x) - x^3$ and solving for that x:

$$\begin{aligned} x\ln(x) - x^3 - 1 &= 0 \\ \implies x(\ln(x) - x^2) = 1 \\ \implies x = \frac{1}{\ln(x) - x^2} \\ \implies x_{i+1} = \frac{1}{\ln(x_i) - x_i^2} \end{aligned}$$

MATH 521, Fall 2018 Notes

Prof. Zachariah B. Etienne

Chapter 14: The Fast Fourier Transform (FFT)

14.1 Fast Fourier Transform (FFT): Motivation

Suppose we wish to compute the Fourier series coefficients of some even function $f(x)$ in the interval $x \in [-L, L)$:

$$a_n = 2\int_0^L f(x)\cos(n\pi x/L)dx$$

Notice that at each point x in the interval, we multiply $f(x)$ by some oscillatory function $\cos(n\pi x/L)$. This function has some natural wavelength λ_n that will depend on n, such that

$$\cos(n\pi(x+\lambda_n)/L) = \cos(n\pi x/L)$$

But we also know that over one wavelength the cosine argument varies by a total of 2π:

$$\cos(n\pi x/L) = \cos(n\pi x/L + 2\pi).$$

So the wavelength of coefficient n in the Fourier series is given by:

$$\begin{aligned} n\pi x/L + 2\pi &= n\pi(x+\lambda_n)/L \\ nx/L + 2 &= n(x+\lambda_n)/L \\ 2 &= n\lambda_n/L \\ \implies \lambda_n &= 2L/n \end{aligned}$$

Thus the wavelength of oscillation in the nth term of a Fourier series decreases as $1/n$.

Knowing this, let's think for a minute how this integral will evaluate with different functions $f(x)$ and different n's. If $f(x)$ is nearly constant, then a_n will evaluate to a very small number for $n > 0$ because in the limit $f(x) = c$ (a constant), $a_n = c\int_0^L \cos(n\pi x/L)dx = 0$. Now let's imagine what would happen if $f(x)$ is smooth & oscillates gently two times on the interval, and we choose a very large value of n, say $n = 100$ with a correspondingly small wavelength. Then if we split the integral to be a sum of integrals over each wavelength, each integral will evaluate to a number very close to zero again, because over each oscillation period, the function is nearly constant. This is the essence of convergence of a Fourier series; oscillatory terms that vary on the same scales as $f(x)$ varies will contribute the most to the Fourier sum.

So if we wish to evaluate the Fourier series of some function $f(x)$ that varies on the scale of Δx_f (e.g., "curvature scale" of f, given by $\sqrt{\frac{f(x)}{f''(x)}}$), we must be sure to compute the Fourier coefficients at least up to n such that

$$\lambda_n \leq \Delta x_f.$$

This implies that $n \geq 2L/\Delta x_f$ to start seeing convergence of a Fourier series.

Recall that to minimally sample an oscillatory function with wavelength λ requires that we choose $\Delta x \leq \Delta x_{\rm Nyq} = \lambda/2$. So for $\cos(n\pi x/L)$,

$$\lambda_n = 2L/n \implies \Delta x \leq \Delta x_{\rm Nyq} = \lambda_n/2 = L/n.$$

So suppose $L = 1$ and the function varies on the order of $\Delta x_f = 0.1$. Further, suppose we choose $n = 100$, which easily satisfies our inequality $\lambda_n \leq \Delta x_f$. Then to resolve a wavelength of the $n = 100$ term of $\cos(n\pi x/L)$ will require uniform sampling of the function $f(x)$ at spacing

$$\Delta x \leq \Delta x_{\rm Nyq} = L/n = 1/100$$

So over the interval from 0 to L, we will need to sample the (even) function $f(x)$ and $\cos(n\pi x/L)$ to at least 100 points to compute the $n = 100$ term. If the function were neither even nor odd, this number would increase to 200 points (since we would double the sampling interval to from $x \in [-L, +L)$).

Extending the discussion, we see that to compute the nth Fourier coefficient, we will need to sample $f(x)$ and the oscillatory Fourier sine and cosine functions to at least $2n$ points before we will start to see convergence in the integral.

Thus the computation of all Fourier coefficients up to and including the nth coefficient will be an $\mathcal{O}(n^2)$ operation.

This is quite an expensive algorithm; if we were to double the number of points where the function is evaluated, we would need four times as many floating point operations. Does a better algorithm exist?

The answer is yes: the Fast Fourier Transform, or simply, the "FFT".

14.2 Fast Fourier Transform (FFT): Discrete Fourier Transform

To understand better how the FFT works, we first need to review some basics about complex numbers. Recall the Euler identity:

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

Recall also the definition of Fourier series:

$$f(x) = \frac{a_0}{2} + \sum_{m=1}^{\infty} \left[a_m \cos\left(\frac{m\pi x}{L}\right) + b_m \sin\left(\frac{m\pi x}{L}\right) \right] \tag{14.1}$$

Let's now define

$$\begin{aligned} A_m &= \sqrt{{a_m}^2 + {b_m}^2} \\ \phi_m &= \arctan(\frac{b_m}{a_m}) \\ c_n &= \begin{cases} \frac{A_n}{2i} e^{i\phi_n} = \frac{1}{2}(a_n - ib_n) & \text{for } n > 0 \\ \frac{1}{2}a_0 & \text{for } n = 0 \\ c^*_{|n|} & \text{for } n < 0. \end{cases} \end{aligned}$$

Then we can show

$$f(x) = \sum_{n=-\infty}^{\infty} c_n e^{i\frac{n\pi x}{L}}$$

is equivalent to the definition of Fourier series (Eq. 14.1).

Proof:

$$\begin{aligned}
f(x) &= \sum_{n=-\infty}^{\infty} c_n e^{i\frac{n\pi x}{L}} \\
&= \frac{a_0}{2} + \sum_{n>0} c_n e^{i\frac{n\pi x}{L}} + \sum_{n<0} c_{|n|}^* e^{i\frac{n\pi x}{L}} \\
&= \frac{a_0}{2} + \sum_{n>0} c_n e^{i\frac{n\pi x}{L}} + \sum_{n>0} c_n^* e^{-i\frac{n\pi x}{L}} \\
&= \frac{a_0}{2} + \sum_{n>0} (c_n e^{i\frac{n\pi x}{L}} + c_n^* e^{-i\frac{n\pi x}{L}}) \\
&= \frac{a_0}{2} + \sum_{n>0} \left(\frac{1}{2}(a_n - ib_n) e^{i\frac{n\pi x}{L}} + \frac{1}{2}(a_n + ib_n) e^{-i\frac{n\pi x}{L}} \right) \\
&= \frac{a_0}{2} + \sum_{n>0} \left(a_n \frac{e^{i\frac{n\pi x}{L}} + e^{-i\frac{n\pi x}{L}}}{2} + ib_n \frac{-e^{i\frac{n\pi x}{L}} + e^{-i\frac{n\pi x}{L}}}{2} \right) \\
&= \frac{a_0}{2} + \sum_{n>0} \left(a_n \frac{e^{i\frac{n\pi x}{L}} + e^{-i\frac{n\pi x}{L}}}{2} + b_n \frac{e^{i\frac{n\pi x}{L}} - e^{-i\frac{n\pi x}{L}}}{2i} \right) \\
&= \frac{a_0}{2} + \sum_{n>0} \left[a_n \cos\left(\frac{n\pi x}{L}\right) + b_n \sin\left(\frac{n\pi x}{L}\right) \right].
\end{aligned}$$

We know how to compute the a_n and b_n's, but how do we compute this new quantity c_n? Recall that we found that only $n/2$ a_n and b_n's minimally resolve the oscillatory term. So what does this mean for the sum? Suppose we have the function at all points x_i, $i \in [0, N)$. Let's now do the reverse Fourier transform, to compute c_n:

$$\begin{aligned}
f(x) &= \sum_{n=-\infty}^{\infty} c_n e^{i\frac{n\pi x}{L}} \\
f(x) e^{-i\frac{m\pi x}{L}} &= \sum_{n=-\infty}^{\infty} c_n e^{i\frac{(n-m)\pi x}{L}} \\
\int_{-L}^{L} f(x) e^{-i\frac{m\pi x}{L}} dx &= \sum_{n=-\infty}^{\infty} c_n \int_{-L}^{L} e^{i\frac{(n-m)\pi x}{L}} dx \\
&= \sum_{n=-\infty}^{\infty} c_n \int_{-L}^{L} (\cos(\frac{(n-m)\pi x}{L}) + i\sin(\frac{(n-m)\pi x}{L}))dx \\
&= 2\sum_{n=-\infty}^{\infty} c_n [\sin(\frac{(n-m)\pi x}{L})]_0^L \\
&= 2\sum_{n=-\infty}^{\infty} c_n [\sin((n-m)\pi)] = 0, \quad \text{unless } n = m. \\
\text{if } n = m \text{, we have:} \quad \int_{-L}^{L} f(x) e^{-i\frac{m\pi x}{L}} dx &= \sum_{n=-\infty}^{\infty} c_n \delta_{n,m} \int_{-L}^{L} dx \\
&= 2Lc_m. \\
\implies \frac{1}{2L} \sum_{n=-\infty}^{\infty} c_n \delta_{n,m} \int_{-L}^{L} dx &= c_m
\end{aligned}$$

To compute the Fourier coefficient integrals, let's use the Trapezoidal rule. To do this, first note that the Fourier series representation of $f(x)$ is periodic and in the continuum limit, the integral is over the interval $x \in [-L, L]$. Since $f(x_0) = f(x_N)$, $f(x_0)/2 + f(x_N)/2 = f(x_0)$. Thus

$$\frac{2L}{2L} \sum_{n=0}^{N-1} f(x_n) e^{-i\frac{m\pi x_n}{L}} = c_m.$$

Sources: *https://en.wikipedia.org/wiki/Discrete-time_Fourier_transform*, *http://home.engineering.iastate.edu/~julied/classes/ee524/LectureNotes/l5.pdf*

Similarly, the discrete, finite Fourier series of $f(x)$ evaluated at N points, for $x \in [-L, L)$ can be written

$$f(x_n) = \sum_{m=0}^{N-1} c_m e^{i\frac{m\pi x_n}{L}},$$

Thus we have found

$$\begin{aligned} f(x_n) &= \sum_{m=0}^{N-1} c_m e^{i\frac{m\pi x_n}{L}} \\ c_m &= \sum_{n=0}^{N-1} f(x_n) e^{-i\frac{m\pi x_n}{L}} \end{aligned}$$

14.2.1 Fast Fourier Transform (FFT): Cooley-Tukey Algorithm

We start from the discrete Fourier Transform for a function that is sampled at $N = 2^M$ points, where $M > 0$ is an integer.

$$f(x_n) = \sum_{m=0}^{N-1} c_m e^{i\frac{m\pi x_n}{L}}.$$

Notice that $x_n = -L + n\Delta x$, where $\Delta x = 2L/N$. Then $x_n = -L + 2Ln/N$ $x_n/L = (-1 + 2n/N)$. Thus,

Source: `https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey\_FFT\_algorithm`*, with slight modifications*

$$\begin{aligned} f(x_n) &= \sum_{m=0}^{N-1} c_m e^{i\frac{m\pi x_n}{L}} \\ &= \sum_{m=0}^{N-1} c_m e^{im\pi(2n/N-1)} \\ &= \sum_{m=0}^{N-1} d_m e^{im2\pi n/N}, \end{aligned}$$

where we have absorbed the $e^{-im\pi}$ term into c_m, to define d_m. Let's now rewrite this as a sum of even and odd terms:

$$\begin{aligned} f(x_n) &= \sum_{m=0}^{N-1} d_m e^{im2\pi n/N} \\ &= \sum_{m=0}^{N/2-1} d_{2m} e^{-\frac{2\pi i}{N}(2m)n} + \sum_{m=0}^{N/2-1} d_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)n} \\ &= \underbrace{\sum_{m=0}^{N/2-1} d_{2m} e^{-\frac{2\pi i}{N/2}mn}}_{\text{DFT of even}-\text{indexed part of } d_m} + e^{-\frac{2\pi i}{N}n} \underbrace{\sum_{m=0}^{N/2-1} d_{2m+1} e^{-\frac{2\pi i}{N/2}mn}}_{\text{DFT of odd}-\text{indexed part of } d_m} = E_n + e^{-\frac{2\pi i}{N}n} O_n. \end{aligned}$$

Thanks to the periodicity of the DFT, we know that

$$\begin{aligned} E_{n+\frac{N}{2}} &= E_n \\ O_{n+\frac{N}{2}} &= O_n \end{aligned}$$

Proof:

$$\begin{aligned}
E_{n+\frac{N}{2}} &= \sum_{m=0}^{N/2-1} d_{2m} e^{-\frac{2\pi i}{N/2}m(n+\frac{N}{2})} \\
&= \sum_{m=0}^{N/2-1} d_{2m} e^{-\frac{2\pi i}{N/2}mn} e^{-\frac{2\pi i}{N/2}m\frac{N}{2}} \\
&= \sum_{m=0}^{N/2-1} d_{2m} e^{-\frac{2\pi i}{N/2}mn} e^{-2\pi i m} \\
&= \sum_{m=0}^{N/2-1} d_{2m} e^{-\frac{2\pi i}{N/2}mn} (\cos(-2\pi m) + i\sin(-2\pi m)) \\
&= \sum_{m=0}^{N/2-1} d_{2m} e^{-\frac{2\pi i}{N/2}mn} (1 + i \times 0) \\
&= \sum_{m=0}^{N/2-1} d_{2m} e^{-\frac{2\pi i}{N/2}mn} \\
&= E_n.
\end{aligned}$$

For $O_{n+\frac{N}{2}} = O_n$, the proof follows precisely the same reasoning.

Therefore, we can rewrite the above equation as

$$f(x_n) = \begin{cases} E_n + e^{-\frac{2\pi i}{N}n} O_n & \text{for } 0 \le n < N/2 \\ E_{n-N/2} + e^{-\frac{2\pi i}{N}n} O_{n-N/2} & \text{for } N/2 \le n < N. \end{cases}$$

We also know that the "twiddle factor" $e^{-2\pi i n/N}$ obeys the following relation:

$$\begin{aligned}
e^{\frac{-2\pi i}{N}(n+N/2)} &= e^{\frac{-2\pi i n}{N} - \pi i} \\
&= e^{-\pi i} e^{\frac{-2\pi i n}{N}} \\
&= -e^{\frac{-2\pi i n}{N}}
\end{aligned}$$

Thus, for $0 \le n < \frac{N}{2}$, we can write

$$\begin{aligned}
f(x_n) &= E_n + e^{-\frac{2\pi i}{N}n} O_n \\
f(x_{n+\frac{N}{2}}) &= E_n - e^{-\frac{2\pi i}{N}n} O_n
\end{aligned}$$

So where is the speed-up coming from? Clearly, the cost to compute the sums E_n and O_n is $\mathcal{O}(\mathcal{N})$, but if we exploit this simple relationship given in Eq. (14.2.1), we can compute $f(x_n)$ by just analyzing $0 \le n < \frac{N}{2}$ terms. So if we were to apply Eq. (14.2.1) once, our Fourier Transform would require $\mathcal{O}(\mathcal{N} * \mathcal{N}/\in)$ operations. **The key to the FFT is that we can continue to apply** *Eq.* (14.2.1) $\log_2 N = M$ **times for** $N = 2^M$**, hence reducing the overall computational complexity to** $\mathcal{O}(\mathcal{N} \log \mathcal{N})$ **operations.**

Note that one can also perform FFT in $N \log(N)$ without the requirement that $N = 2^M$, $M > 0$ an integer, but this is outside the scope of our class.