Introduction to Numerical Analysis

Charles C. Johnson

April 27, 2019

Contents

Introduction to the Course

> *On two occasions I have been asked, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.*
>
> Charles Babbage

Overview

Welcome to Math M-371, Introduction to Numerical Analysis, at Indiana University. The goal of this course is to give an in-depth introduction to the field of numerical analysis, which is essentially the study of the nitty-gritty details of performing mathematics on a computer. As we will discuss, "most" numbers can not be represented exactly in a computer. This means whenever we perform a calculation with such numbers, we are forced to use an approximation of the numbers we want and not the true values. In other words, "most" computations we perform have some inherent error. One of our primary goals will be to understand this error very precisely and develop ways to minimize the error. We will also discuss numerical algorithms for estimating mathematical quantities which we can not directly compute; for example, how can we program a computer to find the solution to some complicated equation for us?

This material, by its very nature, is a mix of computer science and mathematics. However, there is no prerequisite for familiarity with computer science or computer programming for this course, so we will introduce the necessary portions computer programming as we go. Any time we write a computer program we have to make a choice of what *programming language* we want to use – how we will tell the computer what to do. In this course we will use Matlab, which is available for free to all students at IU. This will require you to download and install Matlab on your computer, and instructions for doing this are posted to Canvas.

We will spend the first few weeks of the semester learning how to write code in general, and how to use Matlab in particular. After learning a bit about Matlab, we will then discussing how numbers are represented

in a computer, and begin developing algorithms for various mathematical computations. This will require that we make an excursion through some basic linear algebra, since many of the algorithms we develop will require us to solve various systems of linear equations. Computing interpolating splines, for example, requires that we solve a system of equations which comes from requiring a function to have certain known values at prescribed points and requiring the function's derivatives to be certain values; the solution to the system of equation will tell us coefficients of polynomials that appear in the spline.

All of the material discussed in class appears in the official textbook, *Elementary Numerical Analysis*, 3rd edition, by Atkinson and Han. However, we will discuss the material in a different order than that book and may sometimes dive into the details more thoroughly than the book. In particular we will try to derive any equations or formulas we use in the course, and when appropriate we will prove any necessary theorems (or at least sketch the proof).

Part I

Introduction to Programming in Matlab

1 Introduction to Matlab

Computer science inverts the normal. In normal science, you're given a world, and your job is to find out the rules. In computer science, you give the computer the rules, and it creates the world.

Alan Kay

Numerical analysis is essentially about the nitty gritty details of doing mathematics on a computer, ranging from technical issues about how numbers are represented in a computer, how error accumulates as computations are performed, and developing algorithms for performing various mathematical calculations.

To actually do any of this, though, we need to know how to write code. This chapter provides an introduction to programming using Matlab, and assumes no previous knowledge of computer programming.

1.1 Getting started with Matlab

Throughout this chapter we assume that you have access to Matlab. If you have not yet installed Matlab, see the Appendix A or the notes instructions posted to Canvas.

When you first start Matlab, you will see a window that is broken into three subwindows. The window in the center, called the *Command Window*, is where you can enter commands that you wish for Matlab to perform. This window has a prompt that looks like `>>`, and when you start typing in Matlab your text will appear next to this prompt. For our purposes right now, we will mostly just talk about the Command Window and will wait to introduce the other windows later.

1.2 Using Matlab as a calculator

Basic Calculations and Formatting

Matlab's most basic functionality is evaluating arithmetic expressions: basically just a calculator. You can enter these expressions into Matlab like you would basically guess: using `+` for addition, `-` for subtraction, `*` for

multiplication, and / for division, entering an arithmetic expression and pressing Enter results in Matlab doing the calculation and presenting you with the result of that output immediately below. For example, to add two plus two, we simply enter `2 + 2` at the `>>` prompt inside the command window and hit enter. After entering this, you should see the following in your command window:

```
>> 2 + 2

ans =

      4
```

Notice that beneath the command you entered Matlab prints `ans =` and then the result of the calculation below that. We'll say more about `ans` later, but for right now you can think of that as simply saying that the answer of the previous calculation is four.

Of course, Matlab evaluates expressions according to the usual order of operations where different types of operations have different precedence with expressions in parentheses having highest precedence, followed by expressions that occur in exponents, followed by multiplication and division, then lastly addition and subtraction.

```
>> 2 + 3 * 5

ans =

    17

>> (2 + 3) * 5

ans =

    25
```

It's worth pointing out that you *must* enter an asterisk, `*` (Shift-8 on a standard US keyboard), to perform multiplication: Matlab will complain if you try to simply put two expressions next to one another without an asterisk or some other operator:

```
>> 2(3 + 5)
 2(3 + 5)

Error: Unbalanced or unexpected parenthesis or bracket.
```

To exponentiate, we use the caret symbol, `^` which is Shift-6 on standard US keyboards.

```
>> 2^(18/6)

ans =

    8
```

Exercise 1.1.
Use Matlab to evaluate the following expressions:

(a) $\frac{12^3}{4^2}$

(b) $2 + 3 * 5/7$

(c) $2 + 3/5 * 7$

This all works like you would probably expect, except for a couple of key points. The first thing to notice is that, by default, Matlab only displays

the first four digits of a calculation. For example, the fraction $\frac{22}{7}$, when represented as a decimal requires infinitely-many decimal places:

$$\frac{22}{7} = 3.142857142857142857142857142857...$$

the 142857 above repeats forever. However, if we ask Matlab to compute 22/7 it will tell us the following:

```
>> 22/7

ans =

    3.1429
```

Not only did we only see four digits, but the last digit was rounded up since the fifth (unseen) digit was a five.

If we want to see more digits, we need to enter the command `format long` which tells Matlab to give us fifteen decimal places instead.

```
>> format long
>> 22/7

ans =

   3.142857142857143
```

If we want or need to go back to the default four decimal places, we can evaluate the command `format short`.

```
>> 12/97

ans =

    0.1237
```

```
>> format long
>> 12/97

ans =

   0.123711340206186

>> format short
>> 12/97

ans =

    0.1237
```

Another "oddity," but one that is not exclusive to Matlab, is the following: suppose you asked Matlab to evaluate $2.24 \cdot 10 - 22.4$. Of course, $2.24 \cdot 10 = 22.4$, so this expression should be zero, but we ask Matlab to evaluate this we have

```
>> 2.24*10 - 22.4

ans =

   3.5527e-15
```

Let's first notice that this number is not zero. Let's also note that `e-15` is Matlab's notation for telling us to multiply by 10^{-15}. I.e., `3.5527e-15` really means

$$3.5527 \times 10^{-15} = 0.0000000000000035527$$

So, for this calculation that should be zero, Matlab is telling us the result is this small (but non-zero) number. This isn't really a problem with Matlab *per se*, but a problem with the way that numbers are represented in a computer. We will have a lot more about this to say in the future, but it's good to at least go ahead and be aware of this.

Built-in Functions

Many standard mathematical functions are available in Matlab, such as roots, logarithms, and the trig functions. To compute square roots we use the function `sqrt`:

```
>> sqrt(1024)

ans =

32
```

To compute a higher root, like a cubed root, we can either raise to the $\frac{1}{3}$ power,

```
>> 27^(1/3)

ans =

    3
```

Or we could use the `nthroot` function to compute the n-th root of a number, $\sqrt[n]{x}$. To calculate the n-th root of x with `nthroot`, we first say what number we want to calculate the root of, and which root we want. That is, to compute $\sqrt[n]{x}$ we would use something like `nthroot(x, n)`. For the seventh root of 16384, for example, we would enter the following:

```
>> nthroot(16384, 7)

ans =

     4
```

The usual trig functions are available in Matlab under the names you would expect: `sin` for sine, `cos` for cosine, `tan` for tangent, etc. However, if you start entering values in degrees into these commands you will quickly see that they do not give you back what you were hoping:

```
>> sin(90)

ans =

    0.8940
```

Of course, the reason for this is that these commands assume the input they are given is in terms of radians. In order to use radians it would be helpful if we could enter multiples of π, but this is easy to do in Matlab: we simply use `pi` for π.

```
>> sin(pi/4)

ans =

    0.7071

>> cos(3*pi)

ans =

    -1
```

We can also take logarithms with `log`, although this is actually the logarithm base e, not base 10:

```
>> log(10)

ans =
```

```
    2.3026
```

If you want to calculate the logarithm with another base, you can rewrite it in terms of natural logs:

$$\log_b(x) = \frac{\ln(x)}{\ln(b)}$$

So, for example, to calculate $\log_2(32)$ we could compute

```
>> log(32)/log(2)

ans =

     5
```

Since `log` is the logarithm base e, it might be convenient to know how to enter e in Matlab. Unfortunately, `e` *is not* the mathematical constant e in Matlab. However, Matlab does provide us with a function `exp` for computing powers of e. That is, `exp(x)` is e^x:

```
>> log(exp(17))

ans =

    17
```

Thus e in Matlab is given by `exp(1)` since $e = e^1$.

Very long expressions, whitespace

If you want to enter a very, very long expression, it can be helpful to break it up over multiple lines: it's usually easier to read something broken up into smaller chunks than it is enter one huge command. In order to do this

in Matlab, we must end a line with an ellipsis, ..., and hit Enter, then continue entering the command on the next line.

```
>> 2 + 3 + 5 + 7 + ...
11 + 13 + 17 + 19

ans =

    77
```

Matlab also ignores ***whitespace*** (e.g., spaces and tabs). Thus you can insert extra space between characters in your command if you'd like, and this does not affect the calculation.

```
>> 2   +     3      +     5    + ...
   4   +     9      +    25    + ...
   8   +    27      +   125    + ...
  16   +    81      +   625

ans =

   930
```

1.3 Variables

Storing and using the results of previous calculations

It is often helpful to save the values of previous calculations for future use so that you don't have to re-compute them later. We can easily do this in Matlab by assigning those values to a variable. To create a variable and assign it a value, we simply type the name we want to give the variable, followed by an equals sign, followed by the value (or a calculation resulting in some value we want to hold on to).

```
  favoritePrime = log(exp(17 * (2 + 3)) * ...
                  exp(1 + 1 + 2 + 3))/log(exp(46))

favoritePrime =

     2
```

This creates a variable called `favoritePrime` and sets to the value of whatever expression is on the right, in this case that is 2.

Notice that after executing the above command, there is now an entry in the *Workspace* window to the right of the Command Window for the variable we have created. In general the Workspace shows you the names and values of all currently assigned variables. Notice that `ans` is in here: the `ans` we have been seeing is actually the name of a variable. Every time you evaluate an expression in Matlab, Matlab assigns the result of that expression to the variable `ans` – unless you're already assigning that value to some other variable.

```
>> 5 + 2

ans =

     7

>> ans * 3

ans =

    21

>> nextPrime = sqrt(9)

nextPrime =

     3

>> ans + 2
```

```
ans =

    23
```

Notice that when we create a variable and assign it a name, Matlab prints out the variable we have created and its value. This can be a bit redundant and annoying, but we can tell Matlab to not print anything out after assigning a variable (or performing any other command) by ending the command with a semicolon.

```
>> x = 4;
>> y = 7;
```

If you later want to display the value of a variable later, you can use the `disp` function.

```
>> disp(x)
     4
```

It is often helpful when creating variables to use names that describe whatever the value in the variable is supposed to represent. Using vague, generic names like `x` and `y` is usually a good way to forget what those variables were supposed to represent.

```
>> secondsPerMinute = 60;
>> minutesPerHour = 60;
>> hoursPerDay = 24;
>> daysPerYear = 365;
>> secondsPerYear = secondsPerMinute * ...
                    minutesPerHour * ...
                    hoursPerDay * ...
                    daysPerYear
```

```
secondsPerYear =

    31536000
```

When you name a variable, there are a few rules you have to follow: variable names can not be any arbitrary thing, but instead have to satsify the following conditions:

1. Contain only letters, numbers, and the underscore (_).
2. Must begin with either a letter or an underscore.

If you would like to give a variable a multiword name, you can capitalize the letter of each word to make the words easier to read (e.g., `minutesPerHour` or `favoritePrime`) above, or use the underscore instead of a space.

```
>> age_of_IU = 2019 - 1820

age_of_IU =

   199
```

It is sometimes desirable to "get rid" of variables; to have Matlab pretend you had never assigned anything to a variable. You can delete a variable by using the `clear` command followed by the variable name, which makes Matlab forget you had ever created the variable.

```
>> willForget = 12;
>> 2 * willForget

ans =

    24

>> clear willForget
```

```
>> 2 * willForget
Undefined function or variable 'willForget'.
```

You can also clear *all* of the variables by just entering `clear`.

```
>> a = 2;
>> b = 3;
>> c = 5;
>> a * b * c

ans =

    30

>> clear
>> a * b * c
Undefined function or variable 'a'.
```

1.4 Data types

Every expression entered in Matlab has a corresponding ***data type***. Internally, everything in a computer is ones and zeros, and a sequence of ones and zeros has no intrinsic value: it has to be interpreted as something particular. For example, the sequence `01100001` represents the number 97 when interpreted as an integer; it represents the letter `A` when interpreted as an ASCII character; it represents the number 18.0 when represented as an 8-bit floating point number (under some assumptions). Don't worry if you don't know what all of the previously mentioned terms mean: the important thing is to understanding ones and zeros have no intrinsic meaning and we have to make a choice of how to interpret those ones and zeros.

A ***data type*** is basically just a choice of interpretation for those ones and zeros. For our purposes in this class there are six basic data types we will care about: integers, floating-point numbers, strings, vectors, cell arrays, and logical values. There are also "subtypes" of these data types which determine how many bits Matlab will actually use to represent that

quantity. We don't need to worry too much about all of this right now. A significant portion of the class will be devoted to understanding the integer and floating-point types, so we'll talk about those letter. We'll also discuss the logical type soon when we get more involved in writing code. For now, let's just go ahead and talk about strings and vectors.

Strings

Sometimes we will want Matlab to format the output of our calculations in a nice way. Instead of just printing out numbers, for example, we may want to print out a description of what the number means. This is helpful if we are performing several calculations in a loop and want an easy way to keep track of which values we print out correspond to which input. (We'll talk about loops later, I'm just saying sometimes it's convenient to print a description of our calculation instead of just a number.)

Before we can discuss how to display things in a nice way, we need to discuss strings. A ***string*** is collection of text. In Matlab strings always occur between quotation marks, but they can be either single quotes or double quotes. For example, `'This is a string'` is a string in single quotes, while `"Ceci n'est pas une string."` is a string in double quotes.

Whenever we start a string with a quotation mark, single or double, we *must* end it with the same quotation mark. Inbetween those single or double quotation marks we may place the other quotation mark and it does not prematurely end the string.

```
>> disp('In Scrabble, "syzygy" is worth 21 points.')
In Scrabble, "syzygy" is worth 21 points.
```

Since our string above started with a single quote, it doesn't end until we get to another single quote. This lets us put double quotes in the middle of the string without any problem. Likewise, we can put single quotes in a string easily if we start and end the string with double quotes:

```
>> disp("Short words like 'id' are also useful.")
Short words like 'id' are also useful.
```

We can assign strings to variables and then print them out with `disp`, just like for numerical values.

```
>> shakespeare = "Alas, poor Yorick.";
>> disp(shakespeare)
Alas, poor Yorick.
```

Remark.
There is actually a difference between strings created with single quotes and double quotes in Matlab. For our purposes in this class, 99.9% of the time that distinction won't matter. The one place it does matter is when we want to have a vector of strings, which we'll discuss soon.

Vectors

One of the nice things about Matlab is that it is built for doing calculations with "multidimensional" data with vectors. In Matlab, a ***vector*** is a finite list of values. We specify this finite list by using square brackets, `[` and `]`, and putting a comma separated list of values between the brackets. Usually we want to assign this list to a variable. For example, a list of the first few primes would be given by

```
>> somePrimes = [2, 3, 5, 7, 11, 13, 17, 19];
```

If we want to access a particular element of the list, we give the list's name and then in parentheses specify the ***index*** (aka position) of the value we want. For example, the first thing in our list `somePrimes` above is `somePrimes(1)`, and the fourth thing in our list is `somePrimes(4)`.

```
>> somePrimes(1)

ans =

     2

>> somePrimes(4)

ans =

     7
```

(If you're used to writing code in other languages like C or Java, notice that this is slightly different from what you might be expecting: indices in Matlab start at 1, not 0.)

If you want to know how many things are in a vector, you can use `length` to determine the length of the vector.

```
>> length(somePrimes)

ans =

     8
```

There are lots of operations we can perform to vectors. For example, we can multiply every element in a vector by a constant:

```
>> myVector = [-2, 5, 3, 7];
>> doubled = 2 * myVector

doubled =

    -4    10     6    14
```

We can also add a constant to every element of a vector:

```
>> plusTwo = 2 + myVector

plusTwo =

     0     7     5     9
```

We can also subtract and divide like you would expect.

```
>> myVector / 2

ans =

   -1.0000    2.5000    1.5000    3.5000

>> 3 - myVector

ans =

     5    -2     0    -4
```

We can also exponentiate, but this is slightly different than what you might think. If we tried to do something like `myVector^2`, Matlab will complain:

```
>> squared = myVector^2
Error using  ^
One argument must be a square matrix and the other must be
a scalar. Use POWER (.^) for elementwise power.
```

The reason for this error message is that Matlab can also work with matrices (which we won't talk about just yet) and there the caret operator means

something a little different. If you want to square every element of a vector, you must instead use `.^`:

```
>> squared = myVector .^ 2

squared =

     4    25     9    49
```

We can also do calculations with several vectors at a time, provided the vectors have the same lengths. For example, we can easily add and subtract two vectors component-by-component.

```
>> vector1 = [1, 2, 3, 4, 5, 6];
>> vector2 = [0, -1, 3, 8, -2, 3];
>> vector1 + vector2

ans =

     1     1     6    12     3     9

>> vector1 - vector2

ans =

     1     3     0    -4     7     3
```

This can be useful, by the way, if you have lots of things you want to add to or subtract from one another. If we store those values as vectors, we can just add or subtract the vectors instead of all of the elements individually.

We can also multiply vectors component-by-component, but here we have to use `.*` instead of just `*` (again, because `*` means matrix multiplication which is a completely different operation).

```
>> vector1 * vector2
Error using  *
Inner matrix dimensions must agree.

>> vector1 .* vector2

ans =

     0    -2     9    32   -10    18
```

We can use the functions `max` and `min` to find the largest and smallest values in a vector.

```
>> vector1 = [7, 2, 34, -2, 483, 1384, 18742, -234897, 239];
>> max(vector1)

ans =

       18742

>> min(vector1)

ans =

     -234897
```

Sometimes it is convenient to not only know what the largest or smallest value in a vector is, but also know the index of that value. The `max` and `min` functions can tell us this information as well. That is, these functions can return multiple values instead of just one value. By default Matlab only gives us the first value unless we tell it we want to store multiple outputs. To do this we can create a vector of variables, say `maxVal` for the maximum value in the vector and `maxIdx` for the index of the maximum entry, to which we assign the output of the `max` function.

```
>> [maxVal, maxIdx] = max(vector1);
>> maxVal

maxVal =

       18742
>> maxIdx

maxIdx =

     7
>> vector1(maxIdx)

ans =

       18742
```

In the example above, we created two variables at once with the left-hand side of the first line. Matlab will store the maximum value of the vector `vector1` into `maxVal` and the index of that maximum value in `maxIdx`. In the other commands we are ismply verifying that these really are the maximum value and the index of the maximum value.

Vectors of strings

We can also create vectors of strings: a list of strings, just like we have a list of numbers. However, there are some oddities we need to be aware of. If you tried to create a vector of strings contained in single quotes, something a little strange happens. Consider the following code which we might use to try to create a vector of strings containing three names.

```
>> names = ['Alice', 'Bob', 'Charlene']

names =

    'AliceBobCharlene'
```

Notice that Matlab came back and said our supposed vector of strings was just one long string. Trying to index individual elements of the `names` confirms this is one long string:

```
>> names(1)

ans =

    'A'

>> names(2)

ans =

    'l'

>> names(6)

ans =

    'B'
```

If we were to repeat the above putting our strings in double quotes, however, things work like you would expect:

```
>> names = ["Alice", "Bob", "Charlie"]

names =

  13 string array

    "Alice"    "Bob"    "Charlie"

>> names(1)

ans =
```

```
    "Alice"

>> names(2)

ans =

    "Bob"

>> names(3)

ans =

    "Charlie"
```

What's going on? Why does Matlab take the strings in single quotes and concatenate them together, but keeps the strings in single quotes separate? This is kind of a weird historical accident. In older versions of Matlab you could *only* create strings using single quotes, and internally Matlab represented these strings as vectors of characters. Most of the time this doesn't really matter, but if you try to create a vector of strings (with single quotes), Matlab gets "confused" and thinks you really have one long string, one long vector of characters. This is annoying, so newer versions of Matlab have a separate `string` data type, what you create with double quotes, which you can have vectors of without getting everything lumped into one giant vector of characters.

For the majority of what we're going to do in this class, the distinction between a vector of characters and a string doesn't matter, so we can use single quotes or double quotes for our strings as we'd like. The one exception to this is when we actually want a vector of strings.

As an example of where this might be useful, let's suppose that we have a vector that represents student's grades on an exam. We might not want to know simply what the highest or lowest grades were, but which students received those grades. To keep track of the students in the class, we might like to have a vector of names.

So let's suppose we have a vector of student names and a vector grades on an exam, where the grades are listed in the same order as the students in the vector of names we create. We'll then use `max` and `min` to not only determine the highest and lowest grades, but which students received those grades.

```
>> students = ["Alice", "Bob", "Chloe", ...
               "Daniel", "Eric", "Felice"];
>> grades = [82, 85, 94, 78, 67, 92];
>> [maxGrade, maxIdx] = max(grades);
>> [minGrade, minIdx] = min(grades);
>> disp(maxGrade)
    94

>> disp(students(maxIdx))
Chloe
>> disp(minGrade)
    67

>> disp(students(minIdx))
Eric
```

So the maximum grade of 94 was obtained by Chloe, and the minimum grade of 67 was obtained by Eric.

1.5 Formatting strings

We end this chapter by discussing how we can format strings to print out information in a nice way. Continuing with the example of grades on an exam above, how could we print out a string that says something like

`The highest grade was 94, obtained by Chloe.`

Obviously we could just type the above into a `disp`, but this would have the "94" and "Chloe" *hardcoded* – it would always print the 94 and Chloe, even if we were to change the grades or names of students. It would be nice if we had some code that, regardless of what the grades were or who got the highest grade, printed a single string of text saying what the highest graded was and who obtained it. We know how to write code to determine the highest grade and the name of the student that received that grade, but can we find some way of formatting the string we want to print out so that it prints out those values?

Matlab provides us with two special functions that do exactly this, called `fprintf` and `sprintf`. Both functions take a single string which has some

"placeholders" in it, and then a list of variables which it will put in for those placeholders.

Let's see an example before we try to describe `fprintf` and `sprintf` in general.

```
>> fprintf("Two plus two is %d.", 2 + 2)
Two plus two is 4.>>
```

The `fprintf` command above has a string which has the special code `%d` inside of it. These two characters form a placeholder for a decimal number (which, confusingly, means a whole number). This is followed by a comma, and then an expression which evaluates to a decimal number (whole number). Matlab takes the we gave it, finds the placeholder, and replaces that placeholder with the value of the expression we passed.

Notice that after printing the string but with 4, the value of $2 + 2$, in place of the `%d`, Matlab instantly puts the command prompt next to the string. Usually we won't want this to happen, and so we can tell Matlab to print a new line by putting `\n` at the end of the string.

```
>> fprintf("Two plus two is %d.\n", 2 + 2)
Two plus two is 4.
>>
```

Anywhere we put a `\n`, inside a string passed to `fprintf`, Matlab will insert a new line into the string it's printing.

```
>> fprintf("Line numero uno\nLine dos \n Line three.\n\n")
Line numero uno
Line dos
 Line three.

>>
```

Notice that the first \n immediately gives us a new line: no additional whitespace or anything, just a new line. If we place a space before or after the \n, Matlab prints those spaces as well. (Of course, we can't really see the space before the newline, but we definitely see the space after the newline: that's why the third line has a space just before `Line three`.) In the example above we ended with two newlines, which is why the command prompt got moved down further than you might expect. If there was one newline, the >> prompt would occur immediately below the text which says `Line three`. An additional newline moves the prompt down one more space.

We can have multiple of these placeholders inside our single format string, but for each placeholder that appers we must have an expression whose value will be placed where that placeholder is:

```
>> sum = 1 + 2 + 3 + 4 + 5;
>> product = 1 * 2 * 3 * 4 * 5;
>> fprintf("The sum is %d, and the product is %d.\n", ...
           sum, product)
The sum is 15, and the product is 120.
```

Instead of passing one variable for each placeholder, we can actually pass one vector that contains all of the values.

```
>> years = [1977, 1980, 1983];
>> fprintf("Star Wars movies came out in %d, %d, and %d.\n", ...
           years)
Star Wars movies came out in 1977, 1980, and 1983.
```

In the examples above we used %d for our placeholder, but there are other placeholders for other types of data. For fractional numbers we use %f, or %e if we want to use scientific notation. Consider the following two lines of code for printing out an approximation of π^9:

```
>> fprintf("pi^9 = %f.\n", pi^9)
pi^9 = 29809.099333.
>> fprintf("pi^9 = %e.\n", pi^9)
pi^9 = 2.980910e+04.
```

We actually have some control over how many decimals are printed with `%f`, which for our purposes in this class will occasionally be useful, so we'll go ahead and mention it. By using `%.3f`, for example, we can only print out the first three digits after the decimal place; `%.4f` would print out four digits; `%0.9f` would print out nine digits; and so on.

```
>> fprintf("Two digits: %.2f\n", pi)
Three digits: 3.14
>> fprintf("Three digits: %.3f\n", pi)
Three digits: 3.142
>> fprintf("Four digits: %.4f\n", pi)
Four digits: 3.1416
>> fprintf("Nine digits: %.9f\n", pi)
Nine digits: 3.141592654
```

Sometimes we may want to may want to put other strings inside our string, and `%s` is the placeholder to use for that.

```
>> teacher = "Gauss";
>> pupil = "Riemann";
>> fprintf("%s was a student of %s.\n", pupil, teacher)
Riemann was a student of Gauss.
```

The function `fprintf` can also print text to files, which is what the first `f` refers to. Sometimes we may not want to print anything out just yet, but instead save the formatted string. For this we can use `sprintf` which works just like `fprintf`, except it returns a string instead of printing it out.

```
>> subject = "Alice";
>> verb = "plays";
>> object = "basketball";
>> str1 = sprintf("%s %s %s.", subject, verb, object);
>> subject = "Bob";
>> verb = "sings";
>> object = "folk songs";
>> str2 = sprintf("%s %s %s.", subject, verb, object);
>> disp(str1)
Alice plays basketball.
>> disp(str2)
Bob sings folk songs.
```

2 Scripts and Functions

Science is what we understand well enough to explain to a computer. Art is everything else we do.

Donald Knuth

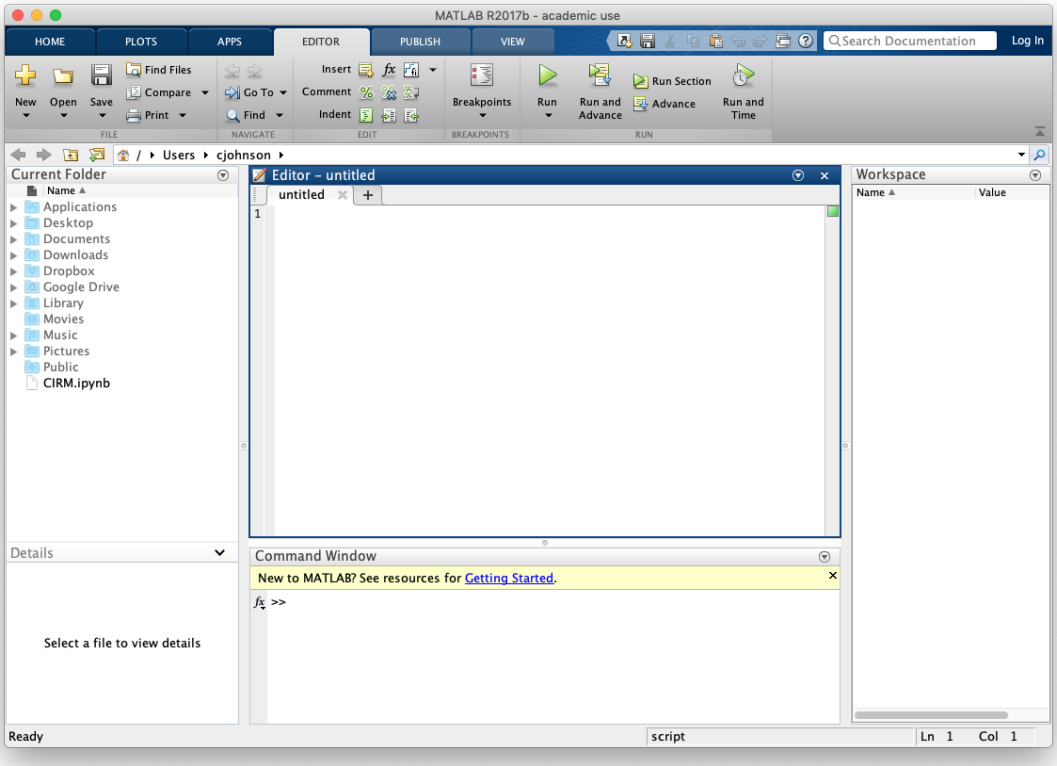
Entering commands directly into Matlab's command window at the `>>` prompt is convenient for certain tasks, especially when you're just learning Matlab and want to play around with various commands to immediately see what happens. For some tasks, however, it's better to save a sequence of commands in a file, and then have Matlab execute all of those commands in order. There are two related ways to do this in Matlab: scripts and functions.

2.1 Scripts

A ***script*** is a file with a `.m` extension which, as stated above, is simply a list of commands that Matlab executes one after the other. Matlab starts at the first line in the file, executes any commands on that line; then moves to the second line, executes any commands on that line; then moves to the third line, executes any commands on that line; and so on until we reach the end of the file.

As a silly example, let's create a script which prints the phrase *Hello, world.* in four different languages each time it is ran.

In Matlab, click on the *New Script* button to create a new file. The region which contained the Command Window should now be split into two parts: the Editor window is on top showing the contents of a new empty file, and the command window is on the bottom.

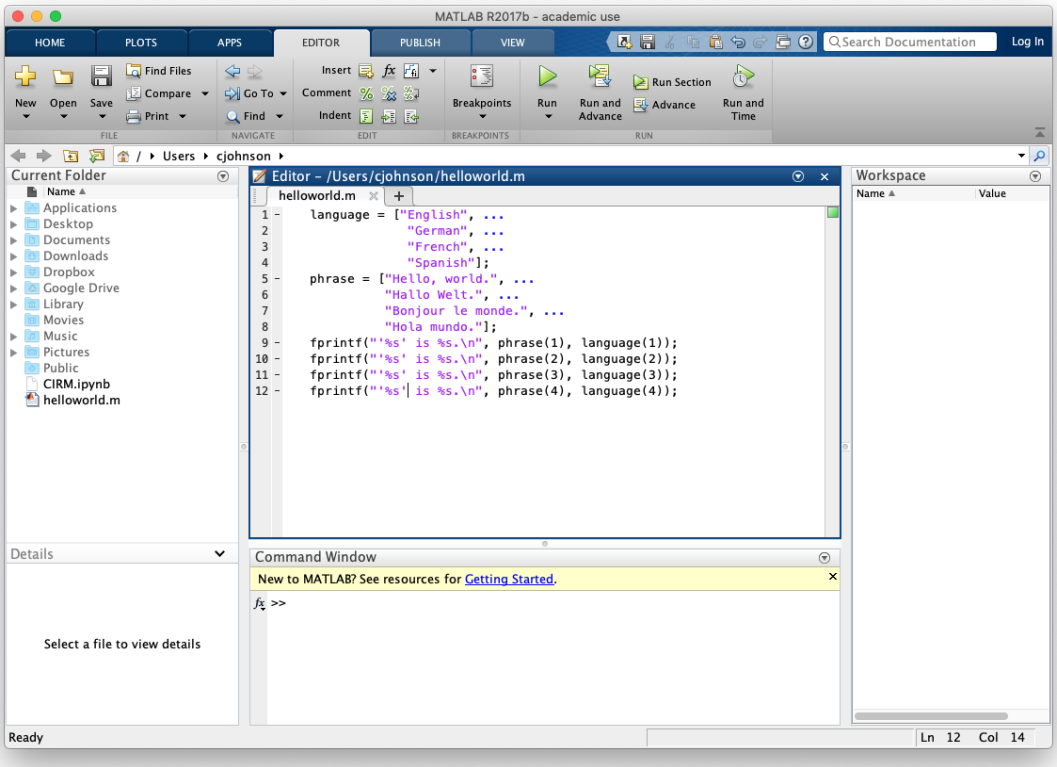


Inside this new file, add the following commands:

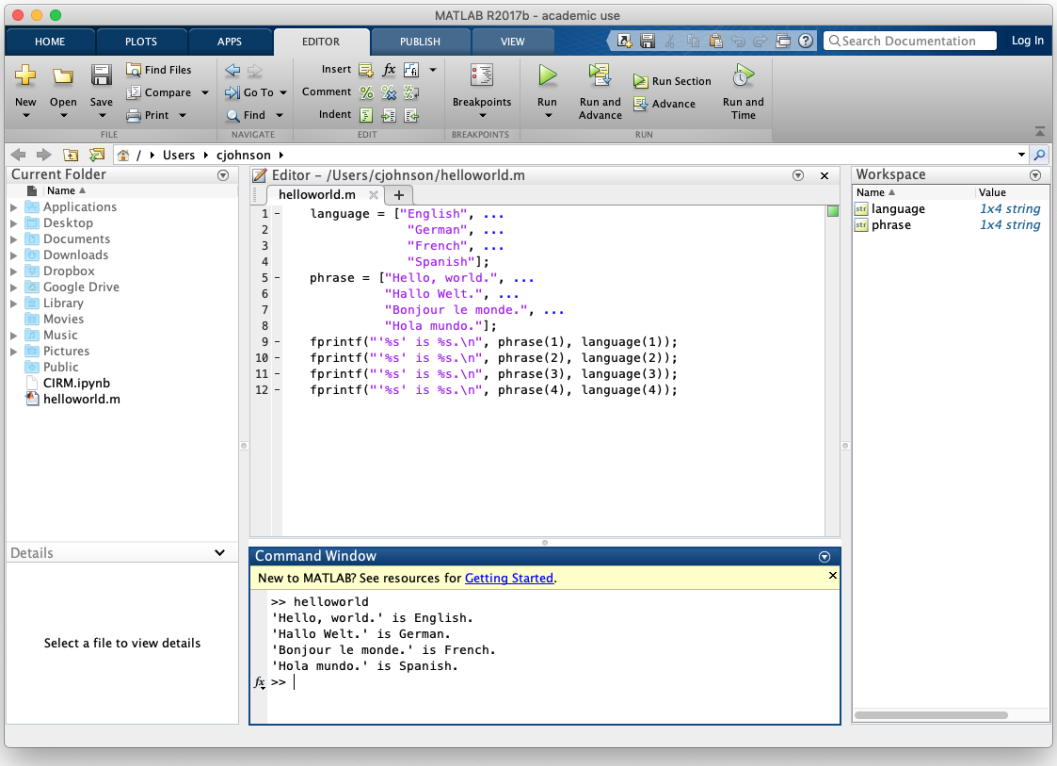
```
language = ["English", ...
            "German", ...
            "French", ...
            "Spanish"];
phrase = ["Hello, world.", ...
          "Hallo Welt.", ...
          "Bonjour le monde.", ...
          "Hola mundo."];

fprintf("'%s' is %s.\n", phrase(1), language(1));
fprintf("'%s' is %s.\n", phrase(2), language(2));
fprintf("'%s' is %s.\n", phrase(3), language(3));
fprintf("'%s' is %s.\n", phrase(4), language(4));
```

Now save the file we've created as `helloworld.m`. Your window should look like the following:



In your Command Window, you can now enter the command `helloworld`, hit enter, and the lines of code in `helloworld.m` will be executed.



2.2 Functions

A ***function*** in Matlab is very similar to a script, except for two key differences: a function can take arguments, and it can return a value. This means that when we call the function, we can pass it values which the function can use in any calculations it performs, and it can also give us back the result of any calculations as something we could store in a variable. This will make more sense if we have some examples.

Let's start off with a very silly function: we'll create a function called `triple` which takes one argument, and returns three times that number. We'll present the code for such a function below, and then talk about what's going on. First we create a new file for our function by selecting *New -¿ Function* from the toobar. This will create a new tab in the Editor window which will contain the contents of our file. By default, Matlab will give us a template for our function, but we can just delete all of this text and put the following in the file:

```
function product = triple(num)
    product = 3 * num;
end
```

If we save this file as `triple.m`, then we will have access to a new command called `triple`. In the Command Window we can then enter something like `triple(7)`, and Matlab will come back and say the result of executing this command is `21`.

```
>> triple(7)

ans =

    21
```

So, what's going on with the code above that we put in the `triple.m` file? The first line,

```
function product = triple(num)
```

has three key ingredients:

- The first word, `function`, tells Matlab we are creating a function. This must always appear when we want to create a new function.
- The next part, `product =`, tells Matlab that we're going to have a variable called `product` that appears in the definition of our function, and this variable is special. Whatever `product` is equal to at the end of the function is the value that will be returned. There's nothing special about the name `product` here – we're just using this name since we're going to multiply two numbers, and the result of a multiplication is called a *product*. We could just as easily called the variable `result` or `returnValue` or `gandalfTheGrey`; it's just a variable name.
- The `triple(num)` tells us two things: the name of the function is `triple`, and `triple` takes one argument which we are calling `num` since it's just some number.

Below the first line we have the ***body*** of the function: these are the commands that make up the definition of the function, the commands that Matlab will execute whenever we call the function. In this particular example the function consists of one command, which just sets `product` to be three times `num`. You'll notice the body of the function is indented. This is not strictly required, but is considered good practice. We'll see several instances later when some of the lines below one line of code are special, and it makes everything a little easier to read and comprehend when these special lines are indented. Don't worry too much if this seems a little strange or you can't imagine why this is useful right now: as we do more involved examples we'll see why indentation is nice, even if not strictly required.

Finally, the function ends by the keyword `end` at the end of the function. This is always required: any function we start we must end, and `end` is the way we do that.

Now, if you have this file `triple.m` saved on your computer and enter something like `triple(7)` into the Command Window, what exactly happens? First, Matlab looks in the current directory for a file named `triple.m` which would contain the contents of a `triple` function. Then, intuitively, what happens is that Matlab takes the contents of that file, but everywhere `num` appears Matlab replaces `num` with the value we gave it in parenthesis, `7` in this case. (That isn't *literally* what happens, but that's how you should think about it.) Now, Matlab executes the lines we have between the line containing `function` and `end`, with `num` replaced by `7`. Once all the commands are executed, Matlab records the value of `product` at the end and says that's the result of our function. That is why Matlab set `ans` to be `21`.

We can now use the result of this function in other calculations or save it to a variable.

```
>> x = triple(7);
>> y = x + 2 * triple(5)

y =

    51
```

Before we see more involved examples, let's consider what would happen if we changed the contents of the `triple.m` file to be

```
function product = triple(num)
    product = 19
    product = 3 * num;
    product = 0;
end
```

Notice in particular there is no semicolon on the first line in the body of the function. If we try to use this function now, by say calling `triple(7)` and saving the result as `x`, then displaying the value of `x`, we would get the following:

```
>> x = triple(7);

product =

    19

>> disp(x)
     0
```

There are a few things to notice here. First note that Matlab printed out the line that said `product` was 19. This is because we left a semicolon off of this line. Usually we will want semicolons at the ends of lines in the body of a function to suppress the output associated to any sort of intermediate or incomplete calculations we're performing.

Also notice that the value of `x` is zero. This is because the value returned by the function is the *last* value that the special variable `product` in our function is set to. The previous values of `product` are lost and forgotten: only the last value matters for the result of the function.

Let's move on to a slightly more involved example: let's create a function that gives the Euclidean distance between two points (x_1, y_1) and (x_2, y_2) in the plane. Of course, the distance between these two points is computed by

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

so let's create a function which returns this value.

Let's name our function `eucldist`, so we'll create a file named `eucldist.m` which will contain the definition of this function.

This time, since we need to give four numbers (two x values and two y values), we will have a comma delimited list of four numbers representing our x and y values as the list of arguments. Our `eucldist.m` file might then contain the following:

```
function dist = eucldist(x1, y1, x2, y2)
    deltaX = x2 - x1;
    deltaY = y2 - y1;
    dist = sqrt(deltaX^2 + deltaY^2);
end
```

While the entire command for calculating the distance between our points could be put on one line, here we elected to split the command up a little. While this might seem silly in this particular example, it's usually good practice to try to take complicated commands and split them up into simpler pieces that are individually easier to understand. Here we created variable `deltaX` and `deltaY` represent the change in the x coordinates and the change in the y coordinates. We then computed the square root of the sum of the squares of these changes in x and y values. This represents the distance we are trying to calculate and want the function to return, so we set the variable `dist` to be the result of this calculation.

Now we could use our `eucldist` function to compute distances between points in the plane. The distance between points $(1, -2)$ and $(3, 7)$, for example, can be computed as

```
>> eucldist(1, -2, 3, 7)

ans =

    9.2195
```

So the distance between $(1, -2)$ and $(3, 7)$ is about 9.2195.

Note the order of the arguments we pass to a function matters. If we were to permute the arguments a little bit, we would get a different answer:

```
>> eucldist(1, 3, 7, -2)

ans =

    7.8102
```

The way we made our function, we give it the x value of the first point, followed by the y value of the first point, then followed by the x value of the second point, and finally the y value of the second point appears at the end. That is, the command `eucldist(1, 3, 7, -2)` above is giving us the distance between $(1, 3)$ and $(7, -2)$.

Let's give one more simple function: instead of computing Euclidean distance between points, let's compute the hyperbolic distance. Just as a quick referesher (or introduction) to hyperbolic geometry, the hyperbolic plane, denoted $\mathbb{H}$, is represented by the upper half-plane. That is, the hyperbolic plane is one half of the Euclidean plane: it only consists of points whose y coordinates are positive. Given any two points (x_1, y_1) and (x_2, y_2) in the upper half-plane (so, by assumption y_1 and y_2 are both positive numbers), the distance between the points is measured in a little bit of a weird way. The distance between points is cooked up so one specific thing happens: the parallel postulate of Euclidean geometry fails. To make this happen we have to change the way we measure distance pretty significantly, and it turns out the formula we want for our distances is

$$2 \ln \left(\frac{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} + \sqrt{(x_2 - x_1)^2 + (y_2 + y_1)^2}}{2\sqrt{y_1 y_2}} \right)$$

You don't need to worry if you've never seen this before, the point is just that it's a kind of complicated formula that people care about. Even if you're the kind of person that cares about hyperbolic geometry, you might not want to try to commit the function above to memory. In such a situation it might be convenient to create a file that does the calcuation for you.

To do this, we will create a function `hypdist` similar to our `eucldist` function before, except for two key differences: of course, we'll compute hyperbolic distance instead of Euclidean distance, but let's also represent our points as two vectors of coordiantes instead of four individual numbers.

That is, instead of having arguments `x1`, `y1`, `x2`, `y2` which give the x and y coordinates of our points, let's just have two arguments, say `pt1` and `pt2`. We'll think of these as being vectors whose first entry is the x coordinate of our point, and whose second entry is the y coordinate. For example, the point $(2, 5)$ would be represented as the vector $[2, 5]$.

The contens of our `hypdist.m` file might then be

```
function dist = hypdist(pt1, pt2)
    deltaX = pt2(1) - pt1(1);
    deltaY = pt2(2) - pt1(2);
    sumY = pt2(2) + pt1(2);
    prodY = pt1(2) * pt2(2);

    numerator = sqrt(deltaX^2 + deltaY^2) + sqrt(deltaX^2 + sumY^2);
    denominator = 2 * sqrt(prodY);

    dist = 2 * log(numerator / denominator);
end
```

Notice that we broke the big, ugly calculation for hyperbolic distance and broken it up into several simpler pieces. If it turned out that we happened to have a mistake somewhere in our calculation, it's usually a lot easier to find that mistake and fix it when we break things up into simpler steps.

The `hypdist` function we created tells us, for example, the hyperbolic distance between $(1, 2)$ and $(4, 2)$ is about 1.3863,

```
>> hypdist([1, 2], [4, 2])

ans =

    1.3863
```

And the hyperbolic distance between $(0, 1)$ and $(0, 2)$ is about 0.6931

```
>> hypdist([0, 1], [0, 2])

ans =

    0.6931
```

Let's now end our discussion of functions by noting that the definition of hyperbolic distance above really requires that the y-coordinates of our points are both positive. If either is zero or negative we'll have a problem since we'll be trying to take the square root of a negative value, or possibly divide by zero. However, Matlab is okay with this: Matlab doesn't complain if we try to divide by zero or take the square root of a negative number:

```
>> 1 / 0

ans =

   Inf

>> sqrt(-1)

ans =

   0.0000 + 1.0000i
```

The above shows that Matlab thinks $\frac{1}{0}$ is infinity, and that $\sqrt{-1}$ is i (which it's writing as $0 + 1i$). Sometimes these are things we like, but sometimes they're not. For example, if we try to find the hyperbolic distance between $(0, -1)$ and $(0, 1)$ Matlab gives us the following:

```
>> hypdist([0, -1], [0, 1])

ans =
```

```
    0.0000 - 3.1416i
```

This is not okay: distances, even hyperbolic distances, should always be non-negative real numbers – what is an imaginary distance? So, what we like is for Matlab to first determine if the y-coordinates we give it are positive, do our calculation if that's the case, but then complain if our y-coordinates are not positive. We will see how to get Matlab to do something like this in the next chapter when we talk about *conditionals*.

3 Conditionals

Everyone should know how to program a computer, because it teaches you how to think.

Steve Jobs

3.1 Motivating example

The code we have written thus far always performed the same operations without regard to any kind of context. Our `hypdist` function in the last chapter, for example, always did the same sequence of computations even for values that would give non-sensical answers. In this chapter we will see how to write code that executes different sequences of commands depending on the context.

We'll begin with a very mathematical example: the absolute value function. Recall that the absolute value of a number x is a piecewise function: $|x|$ is simply x when $x \geq 0$, but it is $-x$ when $x < 0$. For instance, $|3| = 3$, but $|-2| = -(-2) = 2$. How could we create such a function in Matlab?

Remark.
Matlab already has the absolute value function built in, and it's called `abs`. For example, `abs(-2)` will compute the absolute value of -2. Our goal right now is not to create an absolute value function because we need, but just to illustrate the idea of having code that executes differently depending on the context.

Based on what we've done thus far, it's not possible for us to create a command that would give us back x when $x \geq 0$, but gives us back $-x$ when $x < 0$. Instead, we need to introduce a new idea called a ***conditional***, which is sometimes also called a ***if-then statement***.

The idea is that we will check to see if some condition is true or false, and then execute one sequence of commands if the sequence is true, and another sequence of commands if the condition is false. The general format for such a command in Matlab is

```
if (condition)
  (code to execute if condition is true)
else
  (code to execute if condition is false)
end
```

Where `(condition)` is some expression that can be either true or false. For example, $x \geq 0$ is either true or it's false. Many times the condition we'll be interested will be of this form: comparing one number to see if it's less than, greater than, or equal to another number. As you might expect, the condition $x > 0$ is written in Matlab as `x > 0`, but what about if you allow equality: how do we enter $x \geq 0$? We don't have a $\geq$ key on the keyboard that we can type, so we instead use `>=` as our greater-than-or-equal-to operation. For example, the function (saved in a file named `nonnegative.m`) prints out one line if `x` is non-negative, and print out something else if `x` is negative:

```
function nonnegative(x)
    if x >= 0
        fprintf("%f is non-negative\n", x);
    else
        fprintf("%f is negative\n", x);
    end
end
```

We can now execute our `nonnegative` function and it will tell us if the value we gave it was non-negative or not.

```
>> nonnegative(3)
3.000000 is non-negative
>> nonnegative(-pi)
-3.141593 is negative
```

Let's go ahead and make a few observations about the function above. The statements we execute if the condition is true or false are indendented. While this is not strictly necessary, indenting the commands that we will execute if the condition is true or if it is false makes the code much easier to read. Notice too that our function does not have a return value. All of functions we have created up until this point performed some calculation and they gave us back the result of that calculation. This was the value of the special variable created on the first line of the function. E.g., for a function that cubes a given value such as

```
function product = cube(x)
    product = x^3;
end
```

there was a special variable `product` and the value of the function is whatever `product` equals when the function ends.

In our `nonnegative` function above we are just printing some text to the screen, and not performing any calculation. There's nothing to return, so we don't need a special variable.

Let's now use our knowledge of if-then statements to create our own absolute value function, which we might call `absval`:

```
function val = absval(x)
    if x >= 0
        val = x;
    else
        val = -x;
    end
end
```

Saving this in a file `absval.m`, we can use this function to calculate absolute values:

```
>> absval(3)

ans =

     3

>> absval(-2)

ans =

     2
```

3.2 Logical values, comparisons, and/or

When using an if-statement, the condition given must evaluate to either true or false. Values which are either true or false are referred to in Matlab as ***logical values***, and we usually arrive at logical values by using comparison operators such as `>=`, though there are other ways to get logical values we should be aware of.

Let's first go ahead and mention the logical operators that we'll use most often. Above we say that `>=` was how the mathematical operation $\geq$ is entered in Matlab, and so it shouldn't be too surprising that `<=` represents $\leq$ in Matlab. Similarly, `>` and `<` represent strict greater-than and less-than. For example `3 < 3` will evaluate to `false`, while `3 <= 3` evaluates to `true`.

What about equality? You might guess that we would use `=` to test for equality in Matlab, but keep in mind that `=` is *already* used in Matlab to represent assignment. For example, when you enter `number = 7`, Matlab creates a variable called `number` in which it stores the value `7`. That is, a single equals sign *does not* return true or false, it creates a variable and gives it a value.

To test for equality in Matlab we instead have to use the operator `==` (two equals signs). The silly function below tells us if the given value is equal to zero or not:

```
function iszero(x)
    if x == 0
        disp("The value is zero.");
    else
        disp("The value is not zero.");
    end
end
```

The values `true` and `false` can also be returned from a function. For example, we might create a function called `iseven` which checks to see if a given number is even, and returns `true` if it is, but `false` if it is not. One way we might determine if a number is even or not would be to take the number, call it x for the moment, divide by two, and see if the result is an integer or not. For example 18 is even since $\frac{18}{2} = 9$ is an integer, where as 17 is not even since $\frac{17}{2} = 8.5$ is not an integer. How do we know if a number is an integer or not, though? That is, if we're given a variable `x` in Matlab which could potentially be any number, how can we determine if `x` was an integer? One way to do this would be to take the ***floor*** of `x` and see if it equals x or not.

Remark.
Recall that the floor of a number x, usually denoted $\lfloor x \rfloor$ is the largest integer which is no larger than x. Another way to say this, is that $\lfloor x \rfloor$ is what you get if you chop off everything to the right of the decimal when writing the number's decimal expansion. E.g., $\lfloor 13.234 \rfloor = 13$ and $\lfloor 4.99999 \rfloor = 4$.

We can compute the floor of a number in Matlab using the `floor` function:

```
>> floor(pi)

ans =
```

```
     3
```

Our function to test if a given number is even or not would then be the following:

```
function result = iseven(x)
    y = x / 2;

    if y == floor(y)
        result = true;
    else
        result = false;
    end
end
```

So, we're given a number `x`, we set `y` to be half of this number, and then we test to see if `y` is an integer or not by seeing if `y` is equal to its floor. If the number is an integer, then the `x` we were given must have been an even number and so we set `result` to be true. Otherwise we do not have an even number and so we set `result` to be false.

Remark.
If you actually use the `iseven` function above in Matlab, you might find the results it gives you slightly strange. For example, testing to see if four is an even number gives the following:

```
>> iseven(4)

ans =

  logical

   1
```

Following what happens in the body of `iseven`, you'd expect this to be `true` not `1`. What's going on here?

Matlab, and many other programming languages, actually use `1` to represent true and `0` to represent false. Usually it's simpler for us, as humans, to actually use "true" and "false" because we don't want to think of this one or zero as a number – in the sense it's not something we're going to want to do arithmetic with. This is why we used `true` instead of `1`, and `false` instead of `0`, in our definition of the `iseven` function above.

Let's play around with our `iseven` function a little to see if we can make it a little simpler. First note that we set `result` to be `true` if `y == floor(y)` is `true`, and we set `result` to be `false` when `y == floor(y)` is `false`. I.e., `result` is just the value of the expression `y == floor(y)`, so why don't we just set `result` to be the value of this expression? We can then make the `iseven` function a bit shorter:

```
function result = iseven(x)
    y = x / 2;
    result = y == floor(y);
end
```

The last line might look a little weird, so let's take a second to decipher what's going on. The line begins with `result =` which means we are going to set the variable `result` to be the value of whatever is on the right-hand side of that equals sign. The right-hand side is the expression `y ==`

`floor(y)`, which compares `y` to `floor(y)` and evaluates to `true` if these values are equal and `false` otherwise. The end result is that `result` gets set to whatever the value of the expression `y == floor(y)` is.

We can actually make our `iseven` function more succinct by taking advantage of a special function in Matlab called `mod`. The `mod` function takes two arguments, say `x` and `y`, and it returns the remainder obtained by trying to divide `y` into `x`. For example, `mod(13, 5)` evalutes to `3` since the remainder of 13 divide by 5 is three: $13 = 2 \cdot 5 + 3$.

```
>> mod(13, 5)

ans =

     3
```

Notice that if x is divisible by y, then the remainder of x divided by y is zero. For example, 24 is divisible by 3, so the remainder is zero: $24 = 8 \cdot 3 + 0$. In Matlab this means `mod(24, 3)` will evaluate to zero. In general, if `x` is divisible by `y`, then `mod(x, y)` will equal zero. A number is even if and only if it is divisible by two, so we could write our `iseven` function as

```
function result = iseven(x)
    result = mod(x, 2) == 0;
end
```

Now imagine we wanted a function which checked if a number was divisible by *either* two or three. We can check if either of two logical values is true using the operator `||` (two pipe characters; the pipe is obtained by holding Shift and then hitting the blackslack, \, key on the keyboard). This operator occurs inbetween two logical expressions and returns `true` if either of the expressions is true (or if both expressions are true), and returns false only if both expressions are false. For example, `true || false`, `false || true`, and `true || true` are all `true` since at least one of the operands is true, but `false || false` is false.

If we wanted a function which determined if a given number `x` was divisible by either two or three we would check the condition

```
mod(x, 2) == 0 || mod(x, 3) == 0
```

This would evalute to `true` if either `mod(x, 2) == 0` is true (so `x` is divisible by two) or if `mod(x, 3) == 0` is true (so `x` is divisible by three). The function below, for example, prints out a statement if the given number is divisible by two or by three.

```
function divByTwoOrThree(x)
    if mod(x, 2) == 0 || mod(x, 3) == 0
        fprintf("%d is divisible by either two or three.\n", x);
    else
        fprintf("%d is divisible by neither two nor three.\n", x);
    end
end
```

Similarly, there is a way to see if a number is divisible by both two *and* three: the logical operation "and" in Matlab is represented by `&&`. This is similar to the "or" operation, `||`, in that we have

```
(condition 1) && (condition 2)
```

where `(condition 1)` and `(condition 2)` are two logical (true or false) conditions. The "and" of two such conditions will be `true` if and only if *both* conditions are true; the expression is false if either (or both) conditions are false. For example, the expression

```
1 < 2 && 4 < 3
```

evaluates to `false` since the second condition, `4 < 3`, is false.

A function which checks to see if a given number is divisible by both four and six, for instance, would be the following:

```
function result = divByFourAndSix(x)
  if mod(x, 4) == 0 && mod(x, 6) == 0
    result = true
  else
    result = false
```

```
end
```

We could of course also simplify this function down to one line:

```
function result = divByFourAndSix(x)
  result = mod(x, 4) == 0 && mod(x, 6) == 0
end
```

3.3 elseif

Based on what we've done, we now see how we could create a piecewise function in Matlab. For example, suppose we wanted to create the following function Matlab:

$$f(x) = \begin{cases} x^2 - 3x & \text{if } x \leq 2 \\ -x^3 & \text{if } x > 2 \end{cases}$$

Such a function could easily be created with an if-statement

```
function result = f(x)
    if x <= 2
        result = x^2 - 3*x;
    else
        result = -x^3;
    end
end
```

But what if the function was more involved? E.g., what if the function had four different pieces such as

$$f(x) = \begin{cases} x^2 & \text{if } x < 0 \\ x & \text{if } 0 \leq x \leq 1 \\ -x^2 & \text{if } 1 < x < 3 \\ x^3 - 2x & \text{if } x \geq 3 \end{cases}$$

One approach would be to think of this function as having two pieces, say and $x < 0$ piece, which evaluates to x^2, and a $x \geq 0$ piece which is itself a piecewise function. That "new" piecewise function is itself piecewise, but with three pieces. We could try to think of this as a piecewise function with two pieces, where one piece is itself piecewise. Treating each of these "piecewise with two pieces" functions as corresponding to an if-else statement, we could represent our piecewise function above as follows:

```
function result = f(x)
    if x < 0
        result = x^2;
    else
        if x <= 1
            result = x
        else
            if x < 3
                result = -x^2
            else
                result = x^3 - 2*x
        end
    end
end
```

If you look at the code above and feel that it's a bit hard to decipher what exactly is going on, you're certainly not alone. Code like this is ugly, hard to read, hard to understand, and is prone to errors. However, piecewise functions are certainly something we care about, and more generally we may want code that has more than two "branches" – that is, code that can respond in multiple different ways, not limited to just two ways.

In situations like this we want to augment our if-else statement with an `elseif`. The general format of an if-else statement with an `elseif` is as follows:

```
if (condition 1)
    (code that executes if condition 1 is true)
elseif (condition 2)
```

```
    (code that executes if condition 1 is false,
     but condition 2 is true)
elseif (condition 3)
    (code that executes if condition 1 is false,
     and condition 2 is false, but condition 3 is true)
...
elseif (condition n)
    (code that executes if all of the previous
     conditions were false, but condition n is rue)
else
    (code that executes if all of the preceeding
     conditions were false)
end
```

What happens is that Matlab looks down this list of conditions we have given, checks them one-by-one, and once it finds one that is true, it executes that code. *It then skips all of the remaining conditions!* That is, Matlab first checks if `(condition 1)` is true. If so, it executes the code that we have indented above, and then skips down to the `end` at the end our list of conditions. If `(condition 1)` is false, Matlab then moves on to check if `(condition 2)` is true or not. If the condition is true, Matlab executes the code we have indenteded below condition 2, *then skips to* ***end***. This process continues until we get to the `else` at the very end: if none of the conditions above were true, then this last bit of code is what gets executed.

You have to be careful attention to this when you're writing code, otherwise you can write code that will never execute. Consider the function below:

```
function never(x)
    if x < 3
        fprintf("%f < 3\n", x);
    elseif x < 2
        fprintf("This will never print.");
    else
        fprintf("%f >= 3\n", x);
    end
end
```

The claim is that the middle condition, the one which would print `This will never print.` will never be executed. Why? The first condition Matlab checks is whether `x < 3` is true or not. If it is true, then Matlab prints out a statement that the given number is less than three, and then immediately jumps to the `end` that appears after our `else`. That is, Matlab is done once it finds a condition is true and it doesn't bother checking any more conditions. So, if `x < 3` is false, then certainly $x \geq 3$, and so the middle condition, that `x < 2`, is necessarily false, and the

```
fprintf("This will never print.");
```

will never have any opportunity to execute.

Just to verify this, consider what happens if we pass 1.5 to the function:

```
>> never(1.5)
1.500000 < 3
```

Again, notice Matlab didn't check any more conditions once it finds one that is true.

Using the `elseif`, our piecewise function from before becomes much simpler and easier to understand:

```
function result = f(x)
    if x < 0
        result = x^2;
    elseif 0 <= x && x <= 1
        result = x;
    elseif 1 <= x && x < 3
        result = -x^2;
    else
        result = x^3 - 2*x;
    end
end
```

This is much easier to decipher: it's a lot easier to glance at this code and see there are four different conditions being tested. In our earlier imple-

mentation of the piecewise function we have to stop and think for a minute to see what's going on.

The earlier example of a piecewise function where we had if-else statements inside if-else statements inside if-else statements is sometimes said to have "nested if's" – if's inside of if's. While there are times when we can't avoid such a construction, we usually want to avoid having nexted if's as much as possible because they're hard to understand and error prone.

You might notice that our most recent implementation of the piecewise function above actually has some redundancy. Consider the second condition, for example, the one that has

```
elseif 0 <= x && x <= 1
```

The first part, `0 <= x`, is actually unnecessary. Our very first condition in the function, the line containing `if x < 0`, must have been false: otherwise the Matlab would have executed the command `result = x^2` and then jumped to the `end` below our last `else` statement. Since `x < 0` was false, it must be that `x >= 0` is true. That is, we don't need to explicitly check `x >= 0` since it must be true if Matlab gets to our second condition.

Removing these redundancies, our code then becomes

```
function result = f(x)
    if x < 0
        result = x^2;
    elseif x <= 1
        result = x;
    elseif x < 3
        result = -x^2;
    else
        result = x^3 - 2*x;
    end
end
```

3.4 `if` without `else`, and errors

In describing if-else conditions above, our code has had two "branches" (or more if we used `elseif`), but sometimes we will want our code to only

have the one main branch, and occassionally do something extra. One common example of this has to do with creating errors. For most functions we'll create, we will assume the arguments to the function are of a certain type. For example, we may assume the given values are positive, or are integers, or something else. However, Matlab doesn't force someone using our function from giving us some non-sensical values, so we may want to check the arguments to make sure they are values we can work with. If they are not, then we may want Matlab to create an error to tell the user the value they gave us won't work.

One example of this is our `hypdist` function for calculating hyperbolic distance in the last chapter. The formula we have only makes sense if the y-values of the points given are positive. If someone gives non-positive y-values, the results we calculate won't make sense: we'll get things like complex numbers for the distance between two points. So, we may want to check the values we are given have positive y-coordinates. If not, we'll create an error with the `error` function that informs the user they must give us positive y-coordinates. To create the error, we simply call `error` with one argument which is a string describing the error.

```
function dist = hypdist(pt1, pt2)
    if pt1(2) <= 0 || pt2(2) <= 0
       error("y-coordinate must be positive.")
    end

    deltaX = pt2(1) - pt1(1);
    deltaY = pt2(2) - pt1(2);
    sumY = pt2(2) + pt1(2);
    prodY = pt1(2) * pt2(2);

    numerator = sqrt(deltaX^2 + deltaY^2) + ...
                sqrt(deltaX^2 + sumY^2);
    denominator = 2 * sqrt(prodY);

    dist = 2 * log(numerator / denominator);
end
```

Now if someone tries to use our `hypdist` function with negative y-values, they get an error:

```
>> hypdist([0, 1], [0, -1])
Error using hypdist (line 3)
y-coordinate must be positive.
```

Notice that none of the code after the `error` executed: whenever `error` is called, nothing else in that function will ever execute. Consider the function `never` containing the following code:

```
function never()
    error("You silly person...");
    disp("This will not display.");
end
```

If we try to call `never`, we see the following:

```
>> never()
Error using never (line 2)
You silly person...

>>
```

Notice the text `"This will not display."` did not display because it occurs after the call to `error`.

Also notice that our `if` statement in `hypdist` above does not have an `else`. In general, conditions *must* start with an `if`, and then they *may* have one or more `elseif` clauses, they *may* then have an `else`, and then they *must* end with `end`.

4 *Iteration*

Computer science is no more about computers than astronomy is about telescopes.

Edsgar Dijkstra

4.1 `while` loops and not-equals

One of the most convenient things about computers is that they are much, much faster at computations than a human. In particular, some types of calculations we might be interested in may require hundreds, thousands, or millions of individual calculations. While in principle we can do each of those calculations, in reality it's not very realistic for a human being to do millions of calculations. A mathematician, for instance, may be interested in determining if a given number is prime or not. In principle this is the kind of thing *we can* do by hand, but reality it's not something we really want to do by hand for most large numbers because it requires lots of work.

Remark.
Recall that a ***prime number*** is a positive integer greater than 1 which is not divisible by any number other than 1 and itself. For example, the number 13 is prime since it is not divisible by any number less than it (except 1), but the number 12 is not prime since it is divisible by some values less than itself, namely 2, 3, 4, and 6 are all divisors of 12.

Consider the following silly example: suppose we wanted to determine if a given number, say 713 just for the sake of argument, was prime or not. How could we determine this? Well, we would need to actually sit down and check if 2 divided 713, or if 3 divided 713, or if 4 divided 713, or if 5 divided 713, ..., or if 712 divided 713. That is, there are 711 different checks we have to perform. Each check by itself is pretty straight-forward in Matlab: we can see if a given number `n` divides `713` or not by seeing if `mod(713, n)` is equal to zero or not. So, we just need to compute `mod(713,`

`2)`, and `mod(713, 3)`, and `mod(713, 4)`, and ..., and `mod(713, 712)`. We really don't want to write each of these by hand in Matlab, though.

Luckily, there's a convenient way to get Matlab to perform some sort of calculation multiple times – possibly millions of times – with only a few lines of code. The simplest way to do this is with a `while` loop, whose general form is as follows:

```
while (condition)
    (code that executes as long as condition is true)
    (after executing the code, the condition is checked again)
    (if the condition is true, the code executes again)
end
```

A `while` loop evaluates a condition (like a condition that occurs in an `if` statement) to see if it's true or false. If the condition is false, the body of the `while` statement (the code between `while` and `end`) is ignored and Matlab moves on. If the condition is true, however, Matlab executes the code in the body of the `while` loop, just like in an `if` statement. After executing the code, however, Matlab then checks the condition again. If the condition is true, the code executes again, and again checks the condition at the end. This process repeats over and over until the condition finally becomes false.

Remark.
Notice that if you have a condition that never changes, that has no way of becoming false, then the code will execute forever. This is usually not what you want to have happen, so be sure your loop has some way of ending!

In the case of determining if 713 is prime or not, we can use a `while` loop like the following which we'll place in a script `primalityOf713.m`.

```
n = 2;
```

```
while mod(713, n) ~= 0
    n = n + 1;
end

if n < 713
    fprintf("713 is not prime; %d divides it\n", n);
else
    disp("713 must be prime as nothing else divided it")
end
```

Let's take a moment to carefully walk through what the above code will do. First we create a variable `n` which we assigned the value two to. In the `while` loop below, our condition is `mod(713, n) ~= 0`. This uses an operator we had not yet discussed, `~=`. This is the not-equals-to operator: it returns true if the left-hand side and right-hand sides are not equal – think of this as how to write $\neq$ in Matlab. So, `mod(713, n)` returns the remainder of dividing 713 by n. If this remainder is equal to zero, then n divides 713 and our condition is false. When this happens, Matlab just jumps down to the `end` below our `while` loop. If the condition is true, meaning n *does not* divide 713, then we instead bump n up by 1. We then repeat this process, checking if the new value of n divides 713 or not. We finally stop once `mod(713, n)` equals zero. Notice that our condition will not be true forever: if it turns out nothing less than 713 divides 713 (i.e., 713 is prime), then we will add one to n over and over and over until eventually $n = 713$. Once that happens, certainly `mod(713, n)` will be zero (every number divides itself), so the condition is finally false.

The last thing we do is check to see if n made it up to 713 or not. If it did, 713 must have been prime. If not, the last value of n divides 713.

Just to be as clear as possible, let's imagine what happens each time the loop iterates, starting at 2:

1. $n = 2$. The remainder of 713 divided by 2 (i.e., the value of `mod(713, 2)`) is 1, so the condition is equivalent to $1 \neq 0$. This is true, so we execute the code in the body of our while loop, which replaces n by $2 + 1 = 3$.

2. $n = 3$. The remainder of 713 divided by is 2, so the condition $2 \neq 0$ is true, and we update n to be 4.

3. $n = 4$. The remainder of 713 divided by 4 is 1, and $1 \neq 0$, so we update n to be 5.

4. $n = 5$. The remainder of 713 divided by 5 is 3. Three is not zero, so we update to 6, and continue.
5. ...
6. $n = 22$. The remainder of 713 divided by 22 is 9. This is not zero, so update n to be 23.
7. $n = 23$. The remainder of 713 divided by 23 is zero. Now the condition is false: zero equals zero, so $0 \neq 0$ is false. The loop ends. n is not updated, but remains at 23.

Sure enough, if you execute this script you will get the following:

```
>> primalityOf713
713 is not prime; 23 divides it
```

As another example, let's write a simple script to print out the first 50 integers. We can simply modify the code above by having n start at 1, and updating our condition of the `while` loop to be `n <= 50`, and in the body of the loop we'll just print out n.

```
n = 1;

while n <= 50
    fprintf("n = %d\n", n);
    n = n + 1;
end
```

Saving this as, say, `print50.m`, executing the script prints out

```
n = 1
n = 2
n = 3
...
```

```
n = 49
n = 50
```

The first few examples were a little bit simplistic, just to illustrate the idea of a `while` loop, so let's end our discussion of while loops by considering two more interesting examples: the Collatz conjecture and determining if any given number is prime or not.

The Collatz conjecture

The Collatz conjecture concerns a certain family of sequences of positive integers generated in the following way. Given an integer n, we do one of three things. If n is even, we compute $n/2$. If n is odd and $n > 1$, we compute $3n+1$. If $n = 1$, we just stop. Now we generate a list of numbers by starting with any number n, computing the next number using the rules above, then computing the next number in the list by using the same procedure, but with our most recently computed number. We continue doing this until we get to 1. Starting from the number $n = 9$, for instance, we produce the following list of numbers:

$$9, 28, 14, 7, 22, 11, 34, 17, 52, 26,$$
$$13, 40, 20, 10, 5, 16, 8, 4, 2, 1$$

Now, here's a question: is it true that for any positive integer n we use as our starting value, the sequence we generate will be finite? That is, will we always eventually reach 1, or could we generate an infinite list, never reaching 1? This is actually an open problem: no one knows the answer to this question. If we were interested in studying this open problem, we might first like to have some code that generates these kinds of lists of numbers for us. In Matlab we could create function `collatz` which prints out a list of numbers as follows:

```
function collatz(n)
    while n > 1
        disp(n)
        if mod(n, 2) == 0
            n = n / 2;
        else
            n = 3*n + 1;
```

```
        end
    end

    disp(n)
end
```

When we call this function with some argument `n`, it immediately jumps into a `while` loop. As long as the number `n` is greater than 1, we print out the value of n and then update n to be the next element in our sequence by seeing if n is even or not and applying the appropriate rule. If we eventually get down to $n = 1$, then the loop ends and we print out the last value of n. (This is of course going to be 1, and we're printing it just for the sake of completeness.)

We can now easily generate these lists of numbers with our `collatz` function. The list starting from 12, for instance, is given by executing `collatz(12)`:

```
>> collatz(12)
    12

     6

     3

    10

     5

    16

     8

     4

     2
```

```
     1
```

Testing primality

Now let's write a function which determines if a given number `num` is prime or not by modifying our code for checking the primality of 713 above. Essentially we'll just replace the `713` in our code with `num`, but let's also check that the `num` we're given makes sense. In particular, we may want to be sure `num` is an integer that's at least 2 – we don't want to bother determining if something like $-\sqrt{\pi}$ is prime or not. If we're given a value that's either not an integer or is not at least 2, then we should create an error.

Check if `num` is at least 2 is easy enough: we just use an `if` statement where the condition is `num >= 2`. But how do we check to see if `num` is an integer or not?

We said earlier that `mod(m, n)` returned the remainder of `m` divided by `n`. This is true even if `m` is not an integer. For example, consider trying to divide 72.4 by 7: 7 goes into 72.4 evenly ten times, and then there's 2.4 left,

$$72.4 = 10 \cdot 7 + 2.4.$$

In Matlab we have

```
>> mod(72.4, 7)

ans =

    2.4000
```

We can use this to see if a number `m` is an integer or not by taking the remainder of m divided by 1. The remainder will be zero if and only if 1 goes into m an even number of times – i.e., if m is an integer. Another way to say this is that `mod(m, 1)` gives the fractional part of m which is zero if m is an integer. For instance, π is not an integer whereas 19 is:

```
>> mod(pi, 1)

ans =

    0.1416

>> mod(19, 1)

ans =

     0
```

Our function `isprime` below first determines if the given `num` is an integer and if `num` is at least two. If not, then it creates an error. If `num` is an integer at least two, then we loop from `n = 2` until we find a divisor of `num`. We then check to see if this divisor is equal to `num` or not: if it is, `num` must be prime; if not, `num` is not prime.

```
function prime = isprime(num)
    if mod(num, 1) ~= 0
        error("num must be an integer.");
    end

    if num < 2
        error("num must be at least 2.");
    end

    n = 2;

    while mod(num, n) ~= 0
        n = n + 1;
    end

    if n == num
        prime = true;
    else
```

```
        prime = false;
    end
end
```

Now let's create a script which uses our `isprime` function and a loop to determine each of the prime numbers less than, say, 100. Calling this script `printPrimes.m` we might do the following:

```
n = 2;

while n <= 100
    if isprime(n)
        fprintf("%d is prime\n", n);
    end

    n = n + 1;
end
```

Now we have can generate a list of primes less than 100:

```
>> printPrimes
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
```

```
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

4.2 `for` loops

We saw in the previous section that we can create a `while` loop which iterates a fixed number of times, with something like

```
n = 1;
while n <= 10
    (code to execute ten times)
    n = n + 1;
end
```

This type of construction where we loop some fixed number of times is extremely common, and so there's a "shortcut" in Matlab using a `for` loop. The general format of a `for` loop is

```
for n=(start):(stop)
    (code to execute stop - start number of times)
    (n starts at 'start', increments by one each time)
    (then ends at at 'stop')
end
```

For example, a `for` loop that prints out the values one through ten is given by

```
for n=1:10
    fprintf("n = %d\n", n);
end
```

Executing this code prints out

```
1
2
3
4
5
6
7
8
9
10
```

Notice that we *did not* need to increment `n` ourselves in the code above: there is no line with `n = n + 1`. When you create a `for` loop, Matlab does this automatically for you.

Anything we can do with a `while` loop we can also do with a `for` loop and vice versa. However, for certain types of problems, one choice may be more natural than another. Usually it's common to use a `for` loop when you know for sure you want the loop to execute some fixed number of times, and a `while` loop when you want to loop until some condition takes place, but don't necessarily know how long that will take.

As an example of where a `for` loop might be convenient, let's create a function which computes triangular numbers. Recall that the n-th triangular number, often denoted T_n, is the number obtained by

$$T_n = 1 + 2 + 3 + \cdots + n.$$

For example,

$$\begin{aligned} T_1 &= 1 \\ T_2 &= 3 \\ T_3 &= 6 \\ T_4 &= 10 \\ T_5 &= 15 \\ &\vdots \end{aligned}$$

You should know from calculus that there's a nice formula to compute the n-th triangular number, $T_n = \frac{n(n+1)}{2}$, but let's pretend we didn't know that formula and wanted to compute the triangular numbers in the naive way by actually performing the summation above.

Computing the n-th triangular number is a situation where a for loop might be desirable, because we know how many times the loop needs to execute: n times for tne n-th triangular number. The following `triangularNumber` function computes the n-th triangular number:

```
function T = triangularNumber(n)
    T = 0;

    for i=1:n
        T = T + i;
    end
end
```

Note the in the example above we used `i` for the variable in the `for` loop since we were already using `n` to tell us which triangular number we wanted. Having the above function defined, we can compute T_n with the code `triangularNumber(n)`. For example, the fourth triangular number, T_4, is computed with

```
>> triangularNumber(4)

ans =
```

```
    10
```

Another very similar example is computing factorials: recall that $n!$ is defined as

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1.$$

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. We can easily modify the code above to create a function `fact` which computes factorials,

```
function F = fact(n)
    F = 1;

    for i=1:n
        F = F * i;
    end
end
```

Now we can compute 5! with the command `fact(5)`:

```
>> fact(5)

ans =

   120
```

Another common use of `for` loops is to iterate through a vector of values. For example, suppose we wanted to compute the average (aka arithmetic mean) of a collection of numbers. Say the numbers were v_1, v_2, v_3, ..., v_n and we wanted to compute their average,

$$\frac{v_1 + v_2 + v_3 + \cdots + v_n}{n}$$

If these values were contained in a vector `v`, so `v(1)` is the first value v_1, `v(2)` is the second value v_2, and so on, then what we'd like to do is have a loop which walks through the vector, pulling off one entry at a time,

adding it onto the current sum of values, then divides by the number of values. The number of values in a vector, recall, is given by the `length` function. E.g., if `v = [0, 5, -2]`, then `length(v)` is `3`.

Putting all of this together, we can create a function `average` which takes one argument, `v`, which is a vector of values, and returns the average of those values.

```
function avg = average(v)
    avg = 0;

    for i=1:length(v)
        avg = avg + v(i);
    end

    avg = avg / length(v);
end
```

Now we can compute the average of a list of values:

```
>> average([-3, 4, -17, 2, 0, 15])

ans =

    0.1667
```

This idea of walking through a vector one element at a time is so common, there's a built-in way to do it in Matlab using a `for` loop. If `v` is a vector, then the code

```
for i=v
    (code here will use 'i' as an element in v)
    (each element will appear exactly once in the)
```

```
    (order it appears in the vector)
end
```

will set `i` to be each element in the vector. For example, the function `displayEntries` below simply displays each entry that appears in the given vector.

```
function displayEntries(v)
    for i=v
        disp(i);
    end
end
```

Passing this function a vector simply results in each element of the vector being displayed on the screen.

```
>> displayEntries([7, 2, -1, 5, 3])
     7

     2

    -1

     5

     3
```

Using this version of the `for` loop, our `average` function above could be rewritten as

```
function avg = average(v)
    avg = 0;
```

```
    for i=v
        avg = avg + i;
    end

    avg = avg / length(v);
end
```

5 Recursion

To understand recursion, you must first understand recursion.

(Anonymous)

We now turn our attention to the last general programming topic we will consider before moving on to discuss numerical analysis. The last technique we mention is a powerful tool for solving many types of problems. In fact, the basic idea does not belong exclusively to computer science, but rather has been used by mathematicians for centuries.

5.1 The idea of recursion and induction

A common problem-solving strategy in many disciplines is to try to take a big, difficult problem and break it up into smaller, easier problems. To illustrate the idea, let's consider a mathematical problem: Show that for every positive integer n, the number $8^n - 3^n$ is divisible by 5.

At first glance this seems like a tall order: we want to show that for all infinitely-many values of n, the number computed by $8^n - 3^n$ is a multiple of 5. If we try to "spot check" the first few values of n, this certainly seems to be true: $8 - 3 = 5$, $64 - 9 = 55$, $512 - 27 = 485$. We can't actually check *all* values of n in this way, however, since there are infinitely-many. What we *can* do, however, is make one very clever observation. For each value of $n \geq 2$, we can rewrite $8^n - 3^n$ as follows:

$$\begin{aligned} 8^n - 3^n &= 8^n - 3 \cdot 8^{n-1} + 3 \cdot 8^{n-1} - 3^n \\ &= 8 \cdot 8^{n-1} - 3 \cdot 8^{n-1} + 3 \cdot 3^{n-1} - 3 \cdot 3^{n-1} \\ &= 8^{n-1}(8 - 3) + 3 \cdot (8^{n-1} - 3^{n-1}) \\ &= 5 \cdot 8^{n-1} + 3 \cdot (8^{n-1} - 3^{n-1}) \end{aligned}$$

At first glance it looks like we took a simple expression and replaced it with a complicated one, but let's notice two important things about this more complicated expression: the first term is obviously divisible by 5, and the second term will be divisible by 5 if $8^{n-1} - 3^{n-1}$ is divisible by 5. That is, we have reduced the problem of seeing if $8^n - 3^n$ is divisible by 5 to seeing if $8^{n-1} - 3^{n-1}$ is divisible by 5. Repeating the same sort of process, we then see the problem reduces to seeing if $8^{n-2} - 3^{n-2}$ is divisible by 5. But

then we can repeat the argument to see this is equivalent to $8^{n-3} - 3^{n-3}$ being divisible by 5. If we keep doing this over and over we'll eventually determine that $8^n - 3^n$ is divisible by 5 only if $8^1 - 3^1$ is divisible by 5 – but this we instantly know!

The key idea above is that we have a hard problem which we can try to reduce to a slightly easier problem, and then we can reduce that problem a little bit more, and then reduce that new problem, ..., we continue reducing the problem until it becomes trivial. Computer scientists call this kind of technique ***recursion***, whereas mathematicians call it ***induction***.

5.2 The base case

One key feature of our $8^n - 3^n$ example above is that we eventually get down to a point where we can stop reducing. Performing the arithmetic outlined above, we eventually get down to simply $8 - 3$ which of course is 5, and then we can stop doing the reduction. In general, we always need to have some kind of condition like this: a stopping point that we will always reach after some finite number of reductions. This stopping condition is usually called the ***base case***, and you should think of it as the simplest possible version of our problem. If we didn't have a base case, then our reduction procedure would never stop. This is kind of like having an infinite loop: you just keep performing the same procedure over and over again without anything to ever make you stop.

As another mathematical example before we write some code, let's consider the factorial. Recall that one way to define factorials is as

$$n! = n \cdot (n-1)!.$$

This has the same flavor as our reduction above: we reduce the problem of computing $n!$ to the problem of computing $(n-1)!$. Of course, we repeat the procedure to express $(n-1)!$ as $(n-1) \cdot (n-2)!$, and then we want to express $(n-2)!$ as $(n-2) \cdot (n-3)!$, and so forth. Putting all of this together, we get a string of multiplications for our $n!$:

$$\begin{aligned} n! &= n \cdot (n-1)! \\ &= n \cdot (n-1) \cdot (n-2)! \\ &= n \cdot (n-1) \cdot (n-2) \cdot (n-3)! \\ &\vdots \end{aligned}$$

As written, this process never ends because we can always just subtract 1 more and repeat. To prevent this procedure from going on forever we need

a base case, a smallest possible value of n where we know the value of $n!$. Of course, this should simply be $0! = 1$. Taking this as our base-case, we see that the process eventually ends: for any $n \geq 1$ you give me, I can only subtract 1 off finitely-many times before I get down to 0. E.g.,

$$\begin{aligned} 5! &= 5 \cdot 4! \\ &= 5 \cdot 4 \cdot 3! \\ &= 5 \cdot 4 \cdot 3 \cdot 2! \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 \\ &= 120 \end{aligned}$$

5.3 Recursive functions in Matlab

The basic format of a recursive function in Matlab is the following:

```
function result = myfunc(n)
    if (base case condition)
        result = (base case)
    else
        result = (some computation involving myfunc(n-1))
    end
end
```

For example, we can write a recursive factorial function in Matlab as

```
function result = factorial(n)
    if n == 0
        result = 1
    else
        result = n * factorial(n-1)
```

```
    end
end
```

5.4 Palindromic vectors

As another, perhaps more interesting, example let's write a function which takes a vector as an argument, and determines if the entries in that vector are palindromic: do they appear in the same order forwards and backwards? For example, the vector

```
[1, 2, 0, 9, 0, 2, 1]
```

would be palindromic, but the vector

```
[1, 2, 0, 9, 0, 4, 1]
```

would not be.

So, how can we determine if a given vector `v` is palindromic or not? There are a few ways we could try, but perhaps the easiest way is to "think recursively" and try to reduce the problem of determining if `v` is palindromic to seeing if some simpler vector is palindromic or not. In particular, if a vector `v` is a palindrome, then there are two things that have to happen:

1. Its first entry and its last entry must be the same, and
2. the vector consisting of everything inbetween the first and last entries must be a palindrome.

For instance, let's consider the vector `[7, 0, 5, 5, 0, 7]`. We first look at its very first and last entries,and see if they're the same. They are, they're both seven, so now we ask if the vector `[0, 5, 5, 0]` is a palindrome or not. To do this, we repeat the procedure: we see if this vector's first and last entries are the same, and if they are we see if the vector of the middle entries is a palindrome. When should this procedure stop? We should stop when we either determine there is no hope for the vector to be a palindrome (e.g., the first and last entries do not match), or the vector is trivially a plaindrome which happens if the vector has length 1 or length 0.

To actually write this in Matlab, let's notice we need to be able to determine a few things about a given vector. We have seen that the length

of a vector `v` is given in Matlab by `length(v)`, and we can access i-th element of `v` with `v(i)`. For instance, `v(1)` is the fist element of the vector, `v(2)` is the second element, and so on. We'll need to compare the first and last entries of the vector, so we'll want to compare `v(1)` and `v(length(v))`. Access the last element in a vector is very common, so there's a shortcut for it in Matlab: we can access the element at the end of a vector `v` with `v(end)`.

We'll also need to pull out the "middle" of the vector: the elements from the second index to the next-to-last index. This construction of accessing a "subvector" is also very common and there's a simple way to do this in Matlab. If `v` is a vector, then `v(i:j)` refers to the vector of all entries between the i-th entry and the j-th entry, including those two endpoints. For example, if `v` is the vector

```
v = [7, 0, -6, 1, 19, -2]
```

then `v(3:5)` is the vector containing the entries from the third element of `v` up to the fifth element of `v`.

```
>> v = [7, 0, -6, 1, 19, -2];
>> v(3:5)

ans =

    -6     1    19
```

We can thus access the "middle" of a vector `v`, the stuff between the second entry and the next-to-last entry, with `v(2:end-1)`. This gets all the elements from the second element to one shy of the end of the vector. With the vector `v` above, for example, this is

```
>> v = [7, 0, -6, 1, 19, -2];
>> v(2:end-1)

ans =
```

```
     0    -6     1    19
```

With all of this at our disposal, we can now determine if a vector is plaindromic or not with the following `isPalindrome` function.

```
function palindrome = isPalindrome(v)
    if length(v) <= 1
        % If a vector is short enough, it's
        % automatically a palindrome.
        palindrome = true;
    elseif v(1) == v(end)
        % If the ends agree, check the middle.
        middle = v(2:end-1);
        palindrome = isPalindrome(middle);
    else
        % If the ends disagree, we do not
        % have a palindrome.
        palindrome = false;
    end
end
```

Now we have a function which tells us, for example, the vector `[1, 7, 19, 3, 3, 19, 7, 1]` is a palindrome, but `[1, 7, 19, 3, 4, 19, 7, 1]` is not.

```
>> isPalindrome([1, 7, 19, 3, 3, 19, 7, 1])

ans =

  logical

   1

>> isPalindrome([1, 7, 19, 3, 4, 19, 7, 1])
```

```
ans =

  logical

   0
```

5.5 Quicksort

Let's end our discussion of recursion by implementing the quicksort algorithm. We will create a function called `quicksort` which takes one vector of numbers as an argument, and returns a vector whose elements are the same as those of the original vector, but sorted from least to greatest.

There are several different algorithms for sorting, but quicksort is one that is especially well-suited for recursion. The idea is that we will take our vector `v` and pull the first element out of the vector, and we will call this value the *pivot*. We will then divide the remaining elements of `v` into two halves: everything less-than-or-equal-to the pivot, and everything greater than the pivot. These two halves are not automatically sorted, so we'll sort them (notice these are shorter vectors) by calling `quicksort` on those vectors. We will then create a new vector whose first few entries are the elements less than the pivot, after being sorted, followed by the pivot, followed by the elements greater than the pivot, after sorting. This will result in a sorted vector. When should this procedure end? We should stop our recursive procedure once we have a vector which is short enough that we know for certain it's already sorted: if the vector has zero elements or one element, it's already sorted.

In order to implement the procedure above we need to be able to easily divide our vector into the two halves that are less-than-or-equal-to the pivot and greater-than the pivot. In principle we could do this ourselves by writing a loop that walks through the vector element-by-element, and picks off the ones less than the pivot or greater than the pivot. This sort of procedure is so common, though, that Matlab provides us with a shortcut. Given a boolean (true/false) condition that the elements of a vector `v` may satisfy, we can construct a vector containing exactly those elements for which the condition is true with

```
v( condition goes here )
```

where in the condition we use the same variable name, `v` for instance, to represent the individual elements of the vector we are testing the condition against.

For example, if `v = [1, 3, 2, -9, -4, 7, -2, 10]`, we could pull off all the even elements with `v(mod(v, 2) == 0)`, or all of the positive entries with `v(v > 0)`:

```
>> v = [1, 3, 2, -9, -4, 7, -2, 10];
>> evens = v(mod(v, 2) == 0);
>> positives = v(v > 0);
>> disp(evens)
     2    -4    -2    10

>> disp(positives)
     1     3     2     7    10
```

So, to get everything less-than-or-equal-to our `pivot` element, we might use `v(v <= pivot)`; everything greater-than the pivot element is given by `v(v > pivot)`. The one slightly subtle thing we do need to be careful about is that we actually want to remove the pivot from the vector. If we don't, then we will eventually try to sort a list which is no shorter than our current list, and when this happens the recursion repeats forever, and this is a bad thing.

Remark.
No recursion can literally repeat forever when we execute a recursive function on a computer. Each time you call a function, the computer has to record the current "state" of the function that was executing (e.g., the values of any variables created in the function), so that it can get back to that state after the function you've called has finished. This requires the computer to use up a little bit of memory. If you try to call a function infinitely-many times, however, each time you make a function call you use up a little more memory. Since there's only a finite amount of memory you eventually run out and the computer can't call any more functions.

In principle this kind of thing can happen when you call any re-

cursive function, even if the process isn't infinite it may require more memory than is available. For this reason it's sometimes desirable to use iterative methods (loops) instead of recursion, since the loop doesn't require any additional function call and doesn't use up any more memory.

So, in order to remove the pivot element from our vector, let's create a new vector that contains everything from the second element of the vector to the end, and then split *that* vector into two halves.

To put our sorted vectors back together, we can use the following type of command:

[vec1 vec2 vec3]

takes the contents of three vectors, `vec1`, `vec2`, and `vec3`, and concatenates them together into one long vector.

```
>> vec1 = [2, 7, 9];
>> vec2 = [0, -1, 0];
>> vec3 = [3, 1, 9];
>> v = [vec1 vec2 vec3]

v =

     2     7     9     0    -1     0     3     1     9
```

We can also use individual values instead of vectors to insert those values into our long vector.

```
>> vec1 = [1, 2, 3];
>> vec2 = [98, 99, 100];
>> v = [vec1 10 20 30 vec2]

v =
```

```
     1     2     3    10    20    30    98    99   100
```

```
function sorted = quicksort(v)
    if length(v) <= 1
        sorted = v;
    else
        % Remove the first element from the vector,
        % but record its value as 'pivot'.
        pivot = v(1);
        remainder = v(2:end);

        % Separate the remainder of the vector
        % into two halves.
        lessThan = remainder(remainder <= pivot);
        greaterThan = remainder(remainder > pivot);

        % Sort the halves.
        lessThan = quicksort(lessThan);
        greaterThan = quicksort(greaterThan);

        % Put everything back together.
        sorted = [lessThan pivot greaterThan];
    end
end
```

Now we can sort vectors of numbers using our `quicksort` function:

```
>> quicksort([7, 3, -2, 5, -2, 10, 19, 3, 1])

ans =

    -2    -2     1     3     3     5     7    10    19
```

Part II

Basics of Numerical Analysis

6 Computer Arithmetic

The purpose of computation is insight, not numbers.

RICHARD HAMMING

We now turn our attention away from learning the basics of Matlab, and towards more fundamental issues about how numbers are represented in a computer. The issues we discuss here are universal to any programming language or piece of software. Whether you're writing your own code in Matlab, Java, C, Python, or some other language, or if you're simply using software someone else has written, it's important to be aware of the limitations involved in doing any kind of numerical calculation on a computer.

Our goal in this chapter we will explain how numbers are represented in the IEEE floating-point format. We will begin by describing the basic idea using decimal (base ten) numbers, simply because that number system is the most familiar to us, and then we will switch to describing numbers in binary, since this is how they are actually stored on the computer.

6.1 The idea in base ten

In base 10 (the number system we are most used to), we can write integers as a string of digits, 0, 1, 2, ..., 9, and the position of a digit in this string tells us a certain amount of information about the number we are representing. Really, when we write down a string of digits like 4132 or 98, what we are representing is the number which is obtained by adding up multiples of powers of 10:

$$98 = 9 \times 10^1 + 8 \times 10^0$$
$$4027 = 4 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 7 \times 10^0.$$

For numbers that have a fractional part, we break the number into two halves with a dot: the portion to the left of the dot is multiplied by non-negative powers of 10, and the part to the right is multiplied by negative powers of 10. E.g., when we write 8.23 or 13.204 we really mean

$$8.23 = 8 \times 10^0 + 2 \times 10^{-1} + 3 \times 10^{-2}$$
$$13.204 = 1 \times 10^1 + 3 \times 10^0 + 2 \times 10^{-1} + 0 \times 10^{-2} + 4 \times 10^{-3}.$$

In general, we write

$$d_n \cdots d_2\, d_1\, d_0.d_{-1}\, d_{-2} \cdots d_{-m}$$

where each d_i is a digit 0, 1, 2, ..., 9 to mean the number

$$\sum_{i=0}^{n} d_i \times 10^i + \sum_{j=1}^{m} d_{-j} \times 10^{-j}.$$

If we extend this to allow the portion of the string to the right of the dot to be infinitely long (so we have a series instead of just a finite sum), then we can represent every real number in this format.

$$d_n \cdots d_2\, d_1\, d_0.d_{-1}\, d_{-2} \cdots = \sum_{i=0}^{n} d_i \times 10^i + \sum_{j=1}^{\infty} d_{-j} \times 10^{-j}$$

Before we discussing representing numbers in binary, let's go ahead and note that there's another way we could write such a number. We could always choose to write a (non-zero) real number x as a product

$$x = \sigma \cdot \overline{x} \cdot 10^d$$

where $\sigma = \pm 1$, $\overline{x}$ is a real number in the range $[1, 10)$, and d is an integer. E.g.,

$$\begin{aligned} 98 &= 1 \times 9.8 \times 10^1 \\ 4027 &= 1 \times 4.027 \times 10^3 \\ 8.23 &= 1 \times 8.23 \times 10^0 \\ 13.204 &= 1 \times 1.3204 \times 10^1 \\ 0.00278 &= 1 \times 2.78 \times 10^{-3} \\ -0.05283 &= -1 \times 5.283 \times 10^{-2} \end{aligned}$$

In this setup where we write $x = \sigma \cdot \overline{x} \cdot 10^d$ we call σ the *sign* of the number, $\overline{x}$ is called the *mantissa*, and the d is called the *exponent*.

Notice that *every* real number can be represented in this way – this is basically just "scientific notation." This might require, however, that we can write the $\overline{x}$ part of the number be infinitely-long. Our goal today is to explain how numbers are represented on a computer, and this infinitely-long string of numbers is going to be problematic since any computer has only a finite amount of memory. So we are going to have to deal with the restriction that $\overline{x}$ can not have infinitely-many digits. Notice that the exponent d can not be arbitrarily long either.

So let's go ahead and make a restriction here. Keeping the same notation as above, $x = \sigma \cdot \overline{x} \cdot 10^d$, let's suppose that there are restrictions on how many digits of $\overline{x}$ we can write down, and restrictions on the magnitude of d. For example, let's imagine that we want to represent numbers in this format, but we only write down a total of 8 digits. E.g., we have eight "boxes" into which we can write down the σ, the *overlinex*, and the d. How should we allocate these boxes? It's clear that the σ needs to take up one, but only one, of these boxes since we need to write down either $+1$ or -1 in that box. Now there are seven boxes left. Maybe we decide that the last two boxes should store the exponent part of the number. So this number can be between -99 and 99, and the remaining five boxes can store the mantissa. The number 123.456 we would then try to store in these eight boxes as follows: first notice that in our notation above we would write this as

$$123.45 = 1 \times 1.2345 \times 10^2.$$

So the contents of our eight boxes are as follows:

$$\underbrace{\boxed{1}}_{\sigma}\underbrace{\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5}}_{\overline{x}}\underbrace{\boxed{0}\boxed{2}}_{d}$$

Or, the number

$$-0.000048023 = -1 \times 4.8023 \times 10^{-5}$$

would be represented as

$$\underbrace{\boxed{\text{-1}}}_{\sigma}\underbrace{\boxed{4}\boxed{8}\boxed{0}\boxed{2}\boxed{3}}_{\overline{x}}\underbrace{\boxed{0}\boxed{2}}_{d}$$

Aside from the fact that we are using base 10 in these examples instead of base 2, and that we're not really making a deal about the fact that we can store positve and negative numbers in the same box (i.e., we have not literally restricted ourselves to only storing the digits 0 through 9 in our boxes), this is how fractional numbers are basically represented on a computer.

Let's notice that we represent very big numbers in this format:

$$528,340,000,000,000 = 1 \times 5.2834 \times 10^{14}$$

corresponds to

$$\boxed{1}\boxed{5}\boxed{2}\boxed{8}\boxed{3}\boxed{4}\boxed{1}\boxed{4}.$$

And we can represent very small numbers:

$$0.000000000000000000000000000010043 = 1 \times 1.0043 \times 10^{-28}$$

corresponds to

1	1	0	0	4	3	-2	8

.

Notice that the roles of the digits in the mantissa change depending on the exponent we use: i.e., the first digit could be in the 10^{14}-ths place, or the 10^{-28}-ths place. I.e., their roles are not fixed. For this reason this way of formatting numbers is called *floating point* representation.

Let's also notice there are some very obvious problems with representing numbers in this way. For one thing, the numbers represent can not be arbitrarily large: there are numbers that are too big for us to represent in this floating point format with eight boxes set up the way we currently have them set up. The largest number we can represent is

$$9.9999 \times 10^{99}$$

which corresponds to maxing out all of our boxes

1	9	9	9	9	9	9	9

So we can't store super big numbers, but let's go ahead and notice this is already a pretty huge number:

999990000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000.

So this might not actually be such a big problem in most "real world" problems where we don't need to deal with numbers that are anywhere near this big.

Likewise, we can't store arbitrarily small numbers. The smallest positive number we can represent here is

$$1.0000 \times 10^{-99}$$

which would be

1	1	0	0	0	0	-9	9

and this is admittedly a pretty small number:

0.0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000001.

So we do have a pretty big range between our largest possible and smallest possible numbers. We could maybe argue that for most people, most of the time, numbers in this range will suffice.

However, let's notice there's a much more serious problem. We only have at most five (non-zero) digits in any number we represent. I.e., if we wanted to represent a number like 278.1322, we don't have enough "boxes" to represent this number. In fact, we will have to sacrifice some of those digits to attempt to represent this number. There are two obvious choices: we could truncate the this seven-digit number (just chop off the last two digits), or we could round the number to the nearest thing we can represent. We'll have more to say about rounding versus truncating later, but let's not worry too much about it right now: in this example the end result would be the same either way. The very reasonable number 278.1322 has to get converted to 278.13 for us to represent it in this floating point format:

$$278.13 = 1 \times 2.7813 \times 10^2 \rightsquigarrow \boxed{1}\boxed{2}\boxed{7}\boxed{8}\boxed{1}\boxed{3}\boxed{0}\boxed{2}$$

Now let's suppose we were performing some sort of experiment where we were recording data from our experiment, and accuracy was important. Maybe we're studying a new type of Alzheimer's drug and we need to understand dosing very precisely. There may be a serious, real world change between 278.1322, 278.1329, and 278.1335. However, note that the setup we have here doesn't give us a way to distinguish these values. In our floating point setup with eight boxes, these all get turned into the same value. If we are ignorant that such a thing might happen and expect however we store these numbers to perfectly represent any number we want, we're going to run into some trouble. Of course, this is in some way the whole point of this class since we are trying to understand how mathematics is done on a computer.

6.2 IEEE double-precision floating-point numbers

Above we basically described floating point numbers for decimal numbers to illustrate the main ideas. Now we want to dive into how this is actually done on a computer. In particular, we will describe what is known as the *IEEE double-precision floating-point format* and is officially described in IEEE 754-2008. This is the technical reference document created by the IEEE (*Institute for Electrical and Electronics Engineers*) which describes the standard way that numbers are represented in a computer. In particular,

when you enter a number in Matlab, this is by default how the number is represented internally. (This also corresponds to the data type `double` in languages like C and Java.)

Before we can describe this floating point format, we first need to understand how numbers are represented in binary in general (i.e., in a perfect world without any restrictions on memory or formats for the mantissa and exponent, etc.)

Numbers in binary

Integers

Just as in base-10 where we represent numbers as a string of digits 0, 1, 2, ..., 9 and the position of a number tells us what power of 10 to multiply by, in binary (base-2) we represent numbers as strings of *binary digits* (aka *bits*), 0 and 1, and the position of the number tells us what power of 2 to multiply by.

In particular, the string of bits

$$b_n\, b_{n-1}\, \cdots\, b_2\, b_1\, b_0$$

corresponds to the number

$$\sum_{i=0}^{n} b_i 2^i$$

So the string 1101 corresponds to

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 0 + 1 = 13,$$

and the string 10011010 corresponds to

$$\begin{aligned} &1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ =&128 + 0 + 0 + 16 + 8 + 0 + 2 \\ =&154. \end{aligned}$$

Notice that in base-10, numbers have a 1's place, a 10's place, a 100's place, and so on, whereas in binary we have a 1's place (2^0), an 2's place (2^1), a 4's place (2^2), an 8's place (2^3), etc.

If we just write down a string of 1's and 0's there is potentially some ambiguity about whether we are representing a number in binary or decimal. For example, if we just see the string 110, this might represent the number six (if this is supposed to be binary), or one-hundered and ten (if this is supposed to be in decimal). Usually whether we're using binary or base-10

will be clear from context, but if there's a possibility for confusion we will put a subscript 2 to indicate the number is in binary, or a subscript 10 to indicate the number is in written in base-10. E.g., 110_2 for six and 110_{10} for one-hundered and ten.

To convert a number into binary, we need to find the largest power of 2 that is less than that number and then put a 1 in the corresponding spot. We subtract that power of 2 and then repeat the process, filling in zeros for any power of 2 that is too big. For example, let's convert the base-10 number 375 into binary. First we need to find the highest power of 2 that is less than 375. A moment's thought reveals this is 256 which is 2^8. So 375 in binary will look like 1------- and we need to determine whether these seven remaining dashes are supposed to be 1 or 0. As $375 - 256 = 119$, notice that the next highest power of 2, $2^7 = 128$ is too big, so the next digit is zero: there is a 0 in the 2^7's place. So now we know the number is 10-------. Now the next biggest power of 2 is $2^6 = 64$ which is less than 119, so there's a 1 in the $2^6 = 64$'s place. The number is thus 101------. We would then continue by seeing if the next highest power, 32 is less than what remains, $119 - 64 = 55$. Continuing in this way we have that 375 in binary is 101110111.

Numbers with fractional parts

In base-10, digits to the right of a decimal place are multiplied by negative powers of 10. Likewise, base-2 digits to the right of the decimal place are multiplied by negative powers of 2. That is, a number with a fractional number is in binary represented as

$$b_n \cdots b_2 b_1 b_0 . b_{-1} b_{-2} \cdots b_{-m}$$

where each bit b_i is either 0 or 1, and this corresponds to the number

$$\sum_{i=0}^{n} b_i 2^i + \sum_{j=1}^{m} b_{-j} 2^{-j}.$$

For example, let's consider the number in binary 1011.101. This corresponds to

$$\begin{aligned} & 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ =& 8 + 0 + 2 + 1 + 1/2 + 0 + 1/8 \\ =& 11 + 0.5 + 0.125 \\ =& 11.625 \end{aligned}$$

We can also convert decimal numbers into binary. Let's start with 7.375, and let's first break this up into its integer and fractional parts. I.e., we'll find the binary representation for 8 and the binary representation of 0.375 and put them together. For 7, using the technique described above, we see that $7 = 4 + 2 + 1$, so in binary we have $7 = 111_2$. For the fractional part, 0.375 we do something similar to how we find the binary representation of the integer part. We look at consecutive negative powers of 2, see if they are bigger or smaller than the number (or what remains of the number), put a 1 in the corresponding place if the number is smaller, subtract and repeat the process; put a 0 and move to the next place if the value is bigger.

First we consider $2^{-1} = 0.5$. This is bigger than our 0.375, so we put a 0 in the corresponding place. The next power is $2^{-2} = 0.25$ which is smaller than what we have, so we put a 1 in the $1/4$-th's place in our binary representation, subtract $0.375 - 0.25 = 0.125$, and move to the next digit. The next bit corresponds to $2^{-3} = 0.125$ and this completes the process (since the remainder is $0.125 - 0.125 = 0$). To summarize, we have found that $0.375 = 0 + 0.25 + 0.125 = 1/4 + 1/8$ which tells us that in binary we have $0.375 = 0.011_2$. Combining this with the above we have $7.375 = 110.011_2$.

Just as there are numbers which in base 10 require infinitely-many places after the decimal, there are numbers whose binary representations require infinitely-many places. One simple example of this is the number $1/5$. You can verify that in binary we have

$$1/5 = 0.00110011001100110011001100110011...$$

where the 0011 repeats indefinitely. Compare this to how a number like $1/3$ written as a decimal in base 10 is 0.3333.... Let's also notice that in base 10, $1/5$ is simply 0.2: there are numbers that we can represent with a finite number of decimal places in base-10 which require infinitely-many places in binary!

Remark.
(This remark is a little bit off-topic and can be safely ignored by the uninterested.)

At this point a reasonable question would be to ask why do we need to convert into binary? Why can't we just think of everything in base 10? You've probably heard before that everything in a computer is 0's and 1's, but why is that? Really, saying 0 and 1 is an abstraction of what's going on. A computer is more-or-less a collection of (lots and lots) of transistors, which are essentially tiny electric (as opposed

to mechanical) switches that consist of three terminals: an emitter, a base, and a collector. These terminals are connected to small pieces of silicon that are are "doped" with other elements, such as boron or phosphorous. That is, impurities are intentionally introduced into the silicon and this is done to change the number of electrons in the material: silicon has four valence electrons, whereas phosphorous has five and boron has three.

By replacing some of the silicon atoms in a wafer of silicon with phosphorous, we obtain a wafer with an abundance of electrons (i.e., has a negative charge). If we instead replace some of those silicon atoms with boron, then there is an absence of electrons (a positive charge). Attaching this negative wafer to a positive wafer gives a diode. If we attach terminals to the positive part and the negative part and then attach these terminals to a power source, we force the current to flow across the diode in a particular direction. The negatively charged part of the diode is called an *N-type semiconductor* and emits electrons, and the positively charged part is a *P-type semiconductor* and absorbs electrons.

If we take two such diodes and sandwich them together so they share the positively charged part (i.e., we have negative-positive-negative, or NPN) and then attach terminals to each part (one terminal on each negative end, and one terminal for the positive part in the middle), we have a transistor. At the *PN-interface* where the P-type semiconductor attaches to the N-type semiconductor, electrons from the N-side cross over to the P-side and this gives a stable arrangement of electrons. At this PN-interface there is a voltage "barrier:" current does not flow across the PN-interface unless attached to a power source with high enough voltage to overcome this barrier. The clever thing about a transistor is that we have two PN-interfaces that we can play off of each other. By connecting terminals to the three parts of our transistor, (the two N's on the outside and the P in the middle), we basically have a switch where current flows across the transistor if the power source is high enough to overcome the voltage barrier (the switch is on), but does not flow if the voltage is low (the switch is off).

This whole thing with ones and zeros is really an abstraction for setting a transistor to high voltage (1) or low voltage (0) so that our switch is "on" or "off." By combining transistors in various ways we can construct logic gates which take two inputs and produce an output if the inputs are both turned on (as in an AND gate), or if at least one of the switches is on (an OR gate), or if exactly one of the

switches is one (an XOR gate), or which can switch an off to an on and an on to an off (a NOT gate), and so on. We can also chain these gates together in various ways to build "adders" that do arithmetic with binary numbers: treating the inputs as 1's and 0's in binary representation of number, we can combine logic gates in such a way that we can add binary numbers.

So, the whole point of this binary stuff is that at a fundamental, "close to the metal" point of view, computers really are working with 1's and 0's (although this is really just saying whether there is enough voltage to cross the barriers at the PN interfaces in a transistor).

6.3 The IEEE format

In the IEEE standard which virtually all computers and programming languages use to represent numbers we have 64 bits: sixty-four "boxes" in which we can store either a 1 or a 0, and we want to use these 64 boxes to represent the binary numbers above, similarly to how we represented some fractional base-10 numbers with eight boxes above.

First some conventions. As before we will think about writing a number x (in binary) as a product:

$$x = \sigma \times \overline{x} \times 2^d$$

where $\sigma = \pm 1$ (the sign), $\overline{x}$ is a number in the range $[1, 2)$ (the mantissa), and d is an integer (the exponent).

For example, $99.59374 = 1100011.10011_2$. Written as above we have

$$1100011.10011_2 = 1 \times 1.10001110011_2 \times 2^8.$$

(In case it's easier to see in base-10, this is the same as $1 \times 1.55615234375 \times 64$.)

As another example, consider the number -0.1494140625. In binary this is $-0.0010011001_2 = -(1/8 + 1/64 + 1/128 + 1/1024)$. In our format above this is

$$-0.0010011001_2 = -1 \times 1.0011001_2 \times 2^{-3}$$

We want to use sixty-four bits (boxes) to represent this number. Notice that the exponent can be negative, so we need a way of representing this negative number with just zeros and ones (no symbol '-'). In the IEEE standard, the way to do this is to actually cheat ane make it so that we

don't have to store any negative exponents. I.e., we are going to modify our definition above so that a number x is written as

$$x = \sigma \times \overline{x} \times 2^{E-1023}$$

where E will be a non-negative integer between 0 and 2047 (which is the binary number consisting of consecutive 10 ones). So to represent the numbers 99.59374 above we would take $E = 1031$ (which of course we will be convering into binary, $E = 10000000111_2$), and for -0.1494140625 we have $E = 1020 = 1111111100_2$. So when we record the exponent in our "boxes" of sixty-four numbers, we won't record the actual exponent we want, but that exponent plus 1023 (this way we can still represent negative numbers, but recording things less than 1023), but without having to worry about how to represent a negative number without using a special symbol for $-$.

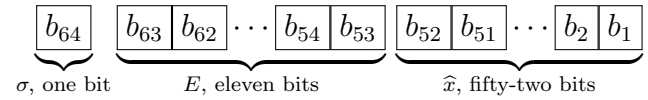
(This is actually a way that computers represent negative integers, so no mantissa or exponent stuff, without using a special symbol – or equivalently a bit for the sign, called *two's complement*, but we won't discuss that now.)

Let's now make an observation about the mantissa, the $\overline{x}$, above. In the case of base-10 numbers, that first digit in the mantissa could be any number between 1 and 9. However, for binary this number will *always* be a 1. Since this is always a 1, we don't really need to record it and can instead just record the stuff after the decimal. I.e., we will further modify our current format for floating point numbers to represent x as

$$x = \sigma \times (1 + \widehat{x}) \times 2^{E-1023}$$

where $\widehat{x}$ represents the fractional part of $\overline{x}$.

Now we're ready to describe the IEEE format: we represent a fractional number $x = \sigma \times (1 + \widehat{x}) \times 2^{E-1023}$ as a string of sixty-four bits where we use the left-most bit for the sign of the number (0 corresponds to $\sigma = 1$, and 1 corresponds to $\sigma = -1$), the next eleven bits for the E-portion of the exponent, and the remaining fifty-two bits for the $\widehat{x}$-portion of the mantissa:



So, for example, the number 99.59374 above we saw was the same as

$$\begin{aligned} 99.59374 &= 1100011.10011_2 \\ &= 1 \times 1.10001110011_2 \times 2^8 \\ &= 1 \times (1 + 0.10001110011_2) \times 2^{1031-1023} \end{aligned}$$

so, $\widehat{x} = 10001110011_2$ and $E = 1031 = 10000000111$. When this is actually stored in the computer using all 64 bits, notice we use all eleven bits of the E portion, but so far we've only written down the eleven of the 52 bits for the $\widehat{x}$-portion. To use all fifty-two bits, we pad with zeros on the left. So, putting all of this together, 99.59374 is stored in the computer as

0 10000000111 0000000000000000000000000000000000000000000010001110011

The number -0.1494140625 we have seen above is

$$\begin{aligned} -0.1494140625 &= -1 \times 1.0011001_2 \times 2^{-3} \\ &= -1 \times (1 + 0.0011001_2) \times 2^{1020-1023} \end{aligned}$$

and so we have $E = 1020 = 1111111100_2$ and $\widehat{x} = 0.0011001_2$. Since this number is negative, we have set the first bit of our 64-bit representation to 1 and we have

1 01111111100 0000000000000000000000000000000000000000000000011001

By convention, a string of a all zeros is considered to be the number 0 in the IEEE standard. Note that this is simply a convention, but without it we would have no way of representing the number zero; using the format above, a string of all zeros would correspond to the number

$$1 \times (1 + 0) \times 2^{0-1023} = 2^{-1023}.$$

There are three other special strings of bits which, by convention, do not represent the decimal numbers you might guess from the format.

The string with $\sigma = 0$, E a string of all 1's and $\overline{x}$ a string of all 0's

$$0 \underbrace{111\cdots1}_{\text{11 bits}} \underbrace{000\cdots0}_{\text{52 bits}}$$

corresponds to infinity; the same string but with the left-most bit equalling 1 corresponds to negative infinity. Setting the exponent E to be all 1's and the mantissa $\overline{x}$ to be anything other than all 0's corresponds to ***not a number***, often denoted ***NaN***. This is a value that is reserved for erroneous calculations which don't have a well defined value, such as division by zero.

Let's notice now that storing numbers in the format we've described is not perfect. There are inherent imperfections in storing real numbers in this format. In particular, "most" real numbers require an infinite amount of "information" to be represented perfectly, and this is always going to be impossible when we only have a finite amount of information to store.

Notice too that there are limits on the largest and smallest values we can store. Keeping in mind the conventions about 0, $\pm\infty$ and NaN described above, the largest number we can store in the IEEE double-precision format is

$$0\,1111111110\,111\cdots1$$

this corresponds to the value

$$1 \times (1 + (1 - 2^{-52})) \times 2^{1023} \approx 1.7977 \times 10^{308}.$$

This is a *huge* nmber. For comparision, cosmologists estimate the number of elementary particles in the visible universe is somewhere around 10^{86}.

Similarly, the smallest positive number we can store is a tiny, tiny number – about 2.225×10^{-308}. For comparision, the mass of an electron is about 9.109×10^{-31} kg.

6.4 Accuracy of floating-point representations

The IEEE format gives us a very wide range of numbers which we can represent. The range of representable numbers is not the problem for most applications; the more serious issue is the accuracy of these representations.

In particular, notice that not all numbers can be represented perfectly, so we might like to have some way of measuring how accurately we can store numbers.

The ***machine epsilon*** of a floating-point format is defined to be the difference between 1 and the next largest number that can be stored. For the IEEE double format this corresponds to

$$0\,01111111111\,000\cdots001$$

which is

$$1 \times (1 + 2^{-52}) \times 2^{1023-1023} = 1 + 2^{-52}.$$

The machine epsilon is thus $1 + 2^{-52} - 1 = 2^{-52}$, which in decimal is about

$$0.0000000000000002 = 2 \times 10^{-16}.$$

This means that two "adjacent" numbers in this format are *at least* 2×10^{-16} apart from one another. I.e., we shouldn't trust double-precision floating-poing numbers to more than about sixteen decimal places in base ten.

A more precise way to state this is that the machine epsilon tells us the maximum amount of error that occurs when we have to round a binary number up or down to store it in the floating-point format we're using.

For example, suppose we have a real number x which we have determined how to write as

$$x = \sigma \times (1 + \overline{x}) \times 2^{E-1023},$$

but suppose the mantissa, $\overline{x}$, requires more than 52 bits. Here we have to make a choice about what to do when we store the number: should we ***truncate*** the $\overline{x}$ to make it fit, or should we round it up or down?

As a simpler example, suppose we had a floating-point format where we only had six bits to use for $\overline{x}$, and we want to store something that would require seven bits – say $\overline{x} = 0.1001111$ We could truncate and throw away the seventh bit, storing `100111`, or we could round to the next nearest number up which would be $0.100111 + 0.000001 = 0.101000$ and store `101000`. (This is completely analogous to rounding 0.1234997 up to 0.1235.)

Regardless of which method we use, truncating or rounding, let's denote by $\text{fl}(x)$ the actual number we store in our floating point format. Notice that in general $\text{fl}(x) \neq x$, but it is some small perturbation of x. I.e., for some small ε,

$$\text{fl}(x) = x \cdot (1 + \varepsilon).$$

The exactly value of this ε depends on x. If we can actually store x perfectly, then $x = \text{fl}(x)$ and $\varepsilon = 0$. The bigger ε is, the more "round-off" error we have.

If we were to simply truncate the bits of $\overline{x}$ down to n bits, then $\text{fl}(x) \leq x$ and so notice $\varepsilon \leq 0$. Furthermore, the "worst case scenario" would be truncating off infinitely-many `1`'s. I.e., replacing

$$\overline{x} = 0.x_1\, x_2\, x_3\, \cdots\, x_n\, 1\, 1\, 1\, \cdots$$

with

$$0.x_1\, x_2\, x_3\, \cdots\, x_n.$$

This is equivalent to subtracting the number

$$0.\underbrace{000\cdots00}_{n \text{ bits}}111\cdots = 0.\underbrace{000\cdots0}_{n-1 \text{ bits}}1 = 2^{-n}.$$

Keeping in mind we are only doing this to the mantissa $\overline{x}$, we have

$$x = \sigma \times (1 + \overline{x}) \times 2^d$$
$$\text{fl}(x) \geq \sigma \times (1 + \overline{x} - 2^{-n}) \times 2^d.$$

Now notice that as $\text{fl}(x) = x(1+\varepsilon)$, we have

$$\varepsilon = \frac{\text{fl}(x)}{x} - 1,$$

and thus

$$\begin{aligned}
\varepsilon &= \frac{\text{fl}(x)}{x} - 1 \\
&\geq \frac{\sigma \times (1 + \overline{x} - 2^{-n}) \times 2^d}{\sigma \times (1+\overline{x}) \times 2^d} - 1 \\
&= \frac{1 + \overline{x} - 2^{-n}}{1+\overline{x}} - 1 \\
&= \frac{1+\overline{x}}{1+\overline{x}} - \frac{2^{-n}}{1+\overline{x}} - 1 \\
&= \frac{-2^{-n}}{1+\overline{x}} \\
&\geq -2^{-n}
\end{aligned}$$

That is, by using n bits for the mantissa $\overline{x}$ when truncating, the possible error in our floating-point representation is bounded by

$$-2^{-n} < \varepsilon \leq 0.$$

Remark.
In some references, such as the Atkinson & Han textbook, the bound given will have an $n+1$ where we have n. The difference between the two bounds is that we are counting the number of bits in the mantissa, $\overline{x}$, whereas they are counting the number of bits in $1+\overline{x}$ which requires one extra bit to store the 1. For example, if $\overline{x} = 1101$ we are only counting these four bits, but some sources count five bits for $1 + \overline{x} = 1.1101$.

What are the possible values of ε when we instead round up or down to the nearest value? First, when should we round up and when should we round down? For a number of the form

$$0.x_1\, x_2 \cdots x_n 0 y_1 y_2 y_3 \cdots$$

when we round, we'll round down because we have a number that's "less than halfway" to the next number.

When we round down, $\text{fl}(x) < x$. In particular, because we assumed there was a 0 in the $(n+1)$-st bit, that means we have subtracted off

$$0.\underbrace{000\cdots0}_{n+1 \text{ zeros}} y_1 y_2 y_3 \cdots$$

Regardless of what the bits $y_1 y_2 y_3 \cdots$ are, this number can be no larger than

$$0.\underbrace{000\cdots0}_{n+1 \text{ zeros}} 111\cdots = 0.\underbrace{000\cdots0}_{n \text{ zeros}} 1 = 2^{-(n+1)}$$

Thus

$$\begin{aligned}
\varepsilon &= \frac{\text{fl}(x)}{x} - 1 \\
&\geq \frac{\sigma \times (1 + \overline{x} - 2^{-(n+1)}) \times 2^d}{\sigma \times (1+\overline{x}) \times 2^d} - 1 \\
&= \frac{1 + \overline{x} - 2^{-(n+1)}}{1+\overline{x}} - 1 \\
&= \frac{1+\overline{x}}{1+\overline{x}} - \frac{2^{-(n+1)}}{1+\overline{x}} - 1 \\
&= \frac{-2^{-(n+1)}}{1+\overline{x}} \\
&\geq -2^{-(n+1)}
\end{aligned}$$

Thus, when rounding down, $\varepsilon \geq -2^{-(n+1)}$. When rounding up we have a similar type of computation. In particular, a number of the form

$$0.x_1\, x_2\, \cdots\, x_n 1 y_1 y_2 y_3 \cdots$$

will be rounded up. This means we will add to our number

$$0.\underbrace{000\cdots0}_{n-1 \text{ zeros}} 1 - 0.\underbrace{000\cdots0}_{n \text{ zeros}} 1 y_1 y_2 y_3 \cdots$$

(This is what we have to add to the number to guarantee that all of the y_i bits together with the 1 in the $(n+1)$-st bit roll over to zeros.) This that we add number is at most

$$0.\underbrace{000\cdots0}_{n \text{ zeros}} 1 = 2^{-(n+1)}.$$

Performing the same calculation as before, but now writing $\text{fl}(x) \leq \sigma \times (1 + \overline{x} + 2^{-(n+1)}) \times 2^d$, we see

$$\varepsilon \leq 2^{-(n++)}.$$

Hence, when rounding is used and we have n bits to store the manitssa, the bounds on the error ε are

$$-2^{-(n+1)} \leq \varepsilon \leq 2^{-(n+1)}.$$

Just to summarize, what we see is that there's a difference in the magnitude of the possible error when we truncate a number versus rounding the number. In particular, the worst possible error that occurs when truncating is double the worst possible error that occurs when rounding.

Let's also notice that not all integers in our range of possible values can be stored exactly. In particular, if the integer's binary representation requires more bits than we can store in the mantissa, we can't store the integer exactly, even though it falls within the range of values we can represent.

For example, in the IEEE double-precision floating-point format, we have 52 bits for the mantissa, and so we can store the numbers 1, 2, 3, ..., 2^{53} exactly:

$$\begin{aligned}
1 &= 1 \times 2^0 = 1 \times (1 + 0.0) \times 2^0 \\
2 &= 1 \times 2^1 = 1 \times (1 + 0.0) \times 2^1 \\
3 &= 1.1 \times 2^1 = 1 \times (1 + 0.1) \times 2^1 \\
4 &= 1 \times 2^2 = 1 \times (1 + 0.0) \times 2^2 \\
5 &= 1.01 \times 2^2 = 1 \times (1 + 0.01) \times 2^2 \\
6 &= 1.1 \times 2^2 = 1 \times (1 + 0.1) \times 2^2 \\
7 &= 1.11 \times 2^2 = 1 \times (1 + 0.11) \times 2^2 \\
8 &= 1 \times 2^3 = 1 \times (1 + 0.0) \times 2^3 \\
&\vdots \\
2^{53} &= 1.0 \times 2^{53} = 1 \times (1 + 0.0) \times 2^{53}
\end{aligned}$$

Notice 2^{53} corresponds to

$$\texttt{0}\ \underbrace{\texttt{10000110100}}_{E=1076}\ \underbrace{\texttt{000}\cdots\texttt{0}}_{\overline{x}}$$

What about the number $2^{53}+1$? If chopping/truncating, we can't represent this number, since we'd just drop off the last 1 in the mantissa! If rounding, this would get rounded up to

$$1 \times 1.\underbrace{000\cdots01}_{51 \text{ zeros}} \times 2^{53}$$

But this is $(1 + 2^{-52}) \times 2^{53} = 2^{53} + 2$. Thus regardless of whether we round or chop, $2^{53} + 1$ is impossible to represent in this format.

7

Quantifying error

Every careful measurement in science is always given with the probable error ... every observer admits that he is likely wrong, and knows about how much wrong he is likely to be.

BERTRAND RUSSELL

7.1 Absolute and relative error

When storing a number in the computer, there is almost always some ***round-off error***. I.e., the number we store is usually only an approximation to the number we really want. We let $\text{fl}(x)$ denote the number actually stored in the computer to represent a real number x. To understand how "good" of an approximation $\text{fl}(x)$ is, we want to consider the relative error.

Recall that if we approximate some value by some other quantity, we can consider both the absolute and the relative error int he approximation. To be more precise, suppose x_T is the true value and x_A is an approximation. The ***absolute error*** in the approximation is defined to be

$$x_T - x_A.$$

For instance, if we approximate $x_T = e$ by $x_A = 2.7183$, then the absolute error is

$$x_T - x_A = e - 2.7183 \approx -0.00001875.$$

The ***relative error*** is the absolute error divided by the true value,

$$\frac{x_T - x_A}{x_T}.$$

The relative error in approximating e by 2.7183, for example, is

$$\frac{e - 2.7183}{e} \approx -0.000006849.$$

We prefer to use relative error because it scales the absolute error to give us an idea of how bad the error is relative to the size of the true value.

Example 7.1.
Suppose the true value of a project which was estimated to be \$1,125,000 was \$1,152,000. The absolute error is \$27,000, and the relative error is

$$\frac{\$27,000}{\$1,152,000} = 0.0234.$$

Now consider a project that was estimated to cost \$10,000 but actually cost \$37,000. The absolute error is again \$27,000 but the relative error is

$$\frac{\$27,000}{\$37,000} = 0.729.$$

The absolute error in approximating a real number x by the floating point number $\text{fl}(x)$ is thus $x - \text{fl}(x)$, and the relative error is

$$\frac{x - \text{fl}(x)}{x}.$$

Let's call this value $-\varepsilon$. A little algebra reveals that this is the negative of the "perturbation" ε we had introduced before.

$$\begin{aligned} & \frac{x - \text{fl}(x)}{x} = -\varepsilon \\ \implies & \frac{\text{fl}(x) - x}{x} = \varepsilon \\ \implies & \text{fl}(x) - x = x\varepsilon \\ \implies & \text{fl}(x) = x + x\varepsilon \\ & \qquad = x(1+\varepsilon). \end{aligned}$$

How bad can this ε be? Let's notice that when we store x we first convert it to the form $x = \sigma \times (1+\overline{x}) \times 2^{E-1023}$. We lose accuracy because we might not be able to store all of the bits of $\overline{x}$. That is,

$$\text{fl}(x) = \sigma \times (1 + \text{fl}(\overline{x})) \times 2^{E-1023}$$

Now notice the relative error is

$$\begin{aligned} -\varepsilon &= \frac{x - \text{fl}(x)}{x} \\ &= \frac{\sigma \times (1+\overline{x}) \times 2^{E-1023} - \sigma \times (1 + \text{fl}(\overline{x})) \times 2^{E-1023}}{\sigma \times (1+\overline{x}) \times 2^{E-1023}} \\ &= \frac{\overline{x} - \text{fl}(\overline{x})}{1+\overline{x}} \end{aligned}$$

We have seen that when truncating $\varepsilon \geq 2^{-52}$. Notice that 2^{-52} is also the machine epsilon of the IEEE double precision format. This is not a consequence: carefully writing down all of the algebra we have described above essentially proves the following theorem.

Theorem 7.1.
If truncation is used, the relative error in approximating a real number x by the floating point number $fl(x)$ is bounded above by the machine epsilon.

7.2 Significant digits

We can relate the relative error in an approximation to the number of significant digits. In approximating a number x_T by x_A, the ***number of significant digits*** is the number of consecutive digits, starting from the left-most non-zero digit, where x_A agrees with x_T.

Example 7.2.
If $x_T = e = 2.7182818...$ and $x_A = 2.7183$, then x_A has four significant digits.

Notice that the number of significant digits depends on the base being used!

Example 7.3.
If $x_T = 15.2718$ and $x_A = 15.2701$, then in binary $x_T = 1111.010001011...$ and $x_A = 1111.010001010....$ Notice in base ten there are four significant digits, but in binary there are twelve!

In base ten, we have k significant digits if the absolute value of the relative error is less than $5 \times 10^{-(k+1)}$:

$$\left|\frac{x_T - x_A}{x_T}\right| < 5 \times 10^{-(k+1)}.$$

Example 7.4.
If $x_T = 123.4567$ and $x_A = 123.4566$, then there are six significant digits and

$$\left|\frac{x_T - x_A}{x_A}\right| = \frac{0.0001}{123.4567} = 0.000000081... < 0.0000005 = 5{\times}10^{-7} = 5{\times}10^{-(6+1)}$$

In binary we have a similar inequality: there will be k significant digits of the relative error is less than $2^{-(k+1)}$.

Example 7.5.
If $x_T = 3.625$ and $x_A = 3.75$, then

$$\left|\frac{x_T - x_A}{x_T}\right| = \left|\frac{-0.125}{3.625}\right| = 0.03448 < 0.0625 = 2^{-4} = 2^{-(3+1)}$$

Now, notice that in binary $x_T = 11.101$ and $x_A = 11.110$, and these two numbers have three significant digits.

Keep in mind that significant digits start from the left-most *non-zero* number. If $x_T = 0.0001234$ and $x_A = 0.0001257$, then there are two significant digits.

If we "normalize" the numbers by writing them in scientific notation, such as

$$x_T = 1.234 \times 10^{-4}$$
$$x_A = 1.257 \times 10^{-4}$$

then we can just count the number of agreeing digits from the left, assuming both numbers have the same exponents (the -4 in the $\times 10^{-4}$ of the example

above). If the exponents disagree, then there are no significant digits. For instance,

$$x_T = 1.234 \times 10^{-4} = 0.0001234$$
$$x_A = 1.234 \times 10^{-5} = 0.00001234$$

have no significant digits.

It's important to realize that we lose significant digits during calculations. For example, if $\text{fl}(x)$ and $\text{fl}(y)$ both have n significant digits in approximating x and y, it could be that an arithmetic operation, such as subtraction, produces fewer significant digits. That is, $\text{fl}(\text{fl}(x) - \text{fl}(y))$ may have fewer significant digits than $\text{fl}(x)$ or $\text{fl}(y)$. Intuitively this means that when we do calculations usings approximations, we can wind up with results which are *worse* approximations to the result of the calculation than the original numbers we used were approximations. Said another way, error can accumulate as we perform calculations with approximations. This is very common, for instance, when subtracting two very close numbers.

Example 7.6.
Let's consider an example in base ten, just to make the computations easier to follow; the idea in base two is exactly the same.

Suppose we represent real numbers as decimals in base ten, recording a total of four digits for the mantissa. Say the true values of x and y are

$$x = \frac{301}{2000} \approx 0.150500000\ldots$$
$$y = \frac{301}{2001} \approx 0.150424787\ldots$$

Then our approximations, which we'll continue to denote as $\text{fl}(x)$ and $\text{fl}(y)$, are

$$\text{fl}(x) = 1.505 \times 10^{-1}$$
$$\text{fl}(y) = 1.504 \times 10^{-1}.$$

Now notice that the true value of the difference $x - y$ is

$$x - y = \frac{301}{2000} - \frac{301}{2001} = \frac{301 \cdot 2001 - 301 \cdot 2000}{2001 \cdot 2000} = 0.0007521\ldots.$$

However, the difference in our approximations is

$$\begin{aligned} \text{fl}(x) - \text{fl}(y) &= 1.505 \times 10^{-1} - 1.504 \times 10^{-1} \\ &= (1.505 - 1.504) \times 10^{-1} \\ &= 0.001 \times 10^{-1} \\ &= 0.0001 \\ &= 1 \times 10^{-4} \end{aligned}$$

Now, when we approximate the true difference we have

$$\text{fl}(x - y) = 7.521 \times 10^{-5}$$

whereas our approximation to the difference using the approximations is

$$\text{fl}(\text{fl}(x) - \text{fl}(y)) = 1 \times 10^{-4}.$$

So, even though $\text{fl}(x)$ and $\text{fl}(y)$ had four significant digits, the difference has zero significant digits in approximating $x - y$!

7.3 Accumulation of error

In general, floating point approximations inherently contain error, and as we do calculations with these approximations, that error can propagate. It's good to understand how "quickly" this can occur, so we will compute how quickly the relative error grows for each of the four arithmetic operations.

Multiplication

Let's consider multiplication first. Suppose we have two real numbers x and y which we approximate with floating point numbers $\text{fl}(x)$ and $\text{fl}(y)$. Letting $-\varepsilon_x$ and $-\varepsilon_y$ denote the relative error in the approximations, we may write $\text{fl}(x) = x(1+\varepsilon_x)$ and $\text{fl}(y) = y(1+\varepsilon_y)$. We want to multiply these numbers together – we want to approximate the product xy. We'll perform multiplication with the floating-point approximations to obtain the product $\text{fl}(x) \cdot \text{fl}(y)$. Notice, however, that even though $\text{fl}(x)$ and $\text{fl}(y)$ are numbers we have represented in the computer, their product is not necessarily a number we can represent, and so the result of our multiplication is $\text{fl}(\text{fl}(x) \cdot \text{fl}(y))$. Letting $-\varepsilon_{xy}$ denote the relative error in approximating $\text{fl}(x) \cdot \text{fl}(y)$ by $\text{fl}(\text{fl}(x) \cdot \text{fl}(y))$, so we may werite $\text{fl}(\text{fl}(x) \cdot \text{fl}(y)) = \text{fl}(x) \cdot \text{fl}(y) \cdot (1 + \varepsilon_{xy})$. How

does this number compare to the true product xy? A little bit of arithmetic gives the following:

$$\begin{aligned}\text{fl}(\text{fl}(x)\cdot\text{fl}(y)) &= \text{fl}(x)\cdot\text{fl}(y)\cdot(1+\varepsilon_{xy})\\ &= x(1+\varepsilon_x)\cdot y(1+\varepsilon_y)\cdot(1+\varepsilon_{xy})\\ &= xy(1+\varepsilon_x)(1+\varepsilon_y)(1+\varepsilon_{xy})\\ &= xy(1+\varepsilon_x+\varepsilon_y+\varepsilon_x\varepsilon_y+\varepsilon_x\varepsilon_{xy}+\varepsilon_y\varepsilon_{xy}+\varepsilon_x\varepsilon_y\varepsilon_{xy}).\end{aligned}$$

Keep in mind each relative error, each ε above, is less than the machine epsilon which is a tiny number. When we multiply two tiny numbers we get a *really* tiny number, and so the sum of ε terms above has the form

$$\varepsilon_x+\varepsilon_y+(\text{product terms})\approx\varepsilon_x+\varepsilon_y.$$

That is, the relative error in approximating the true value of the product xy with the floating-point representation of product the floating-point representations of x and y is approximately the sum of the relative errors in approximating x and y.

Division

We can perform a similar sort of analysis for division, although the computation is a little bit more involved. In particular, we will replace one of our factors that appears with an infinite series. Let's first recall that if r is a real number with $|r|<1$, then the geometric series

$$\sum_{k=0}^{\infty} r^k$$

converges to

$$\frac{1}{1-r}.$$

Below we will want to replace a factor of the form $\frac{1}{1+r}$ with a series, and so we simply note

$$\frac{1}{1+r}=\frac{1}{1-(-r)}=\sum_{k=0}^{\infty}(-r)^k=\sum_{k=0}^{\infty}(-1)^k r^k.$$

Using the same notation as above, in calculating the propogation of error with multiplication, but letting $-\varepsilon_{x/y}$ denote the relative error in approxi-

mating the number $\mathrm{fl}(x)/\mathrm{fl}(y)$, we have the following:

$$\begin{aligned}\mathrm{fl}\left(\frac{\mathrm{fl}(x)}{\mathrm{fl}(y)}\right) &= \frac{\mathrm{fl}(x)}{\mathrm{fl}(y)}(1+\varepsilon_{x/y}) \\ &= \frac{x(1+\varepsilon_x)}{y(1+\varepsilon_y)}(1+\varepsilon_{x/y}) \\ &= \frac{x}{y}(1+\varepsilon_x)(1+\varepsilon_{x/y})\cdot\frac{1}{1+\varepsilon_y}\end{aligned}$$

Now we replace the right-most factor with our series as described above.

$$\begin{aligned}&\frac{x}{y}(1+\varepsilon_x)(1+\varepsilon_{x/y})\cdot\frac{1}{1+\varepsilon_y} \\ =&\frac{x}{y}(1+\varepsilon_x)(1+\varepsilon_{x/y})\cdot\sum_{k=0}^{\infty}(-1)^k\varepsilon_y^k \\ =&\frac{xy}{y}(1+\varepsilon_{x/y}+\varepsilon_x+\varepsilon_x\varepsilon_{x/y})(1-\varepsilon_y+\varepsilon_y^2-\varepsilon_y^3+\cdots)\end{aligned}$$

As mentioned when we computed the propogation of error for multiplication, each ε term is less than the machine epsilon which is a tiny number and so products of such terms are extremely tiny. We unceremoniously decide to ignore these extremely small values to write the following.

$$\begin{aligned}&\frac{xy}{y}(1+\varepsilon_{x/y}+\varepsilon_x+\varepsilon_x\varepsilon_{x/y})(1-\varepsilon_y+\varepsilon_y^2-\varepsilon_y^3+\cdots) \\ \approx&\frac{x}{y}(1+\varepsilon_{x/y}+\varepsilon_x)(1-\varepsilon_y) \\ =&\frac{x}{y}(1+\varepsilon_{x/y}+\varepsilon_x-\varepsilon_y-\varepsilon_y\varepsilon_{x/y}-\varepsilon_x\varepsilon_y) \\ \approx&\frac{x}{y}(1+\varepsilon_{x/y}+\varepsilon_x-\varepsilon_y).\end{aligned}$$

That is, the relative error in approximating the quotient $\frac{x}{y}$ is approximately

$$\varepsilon_{x/y}+\varepsilon_x-\varepsilon_y$$

Addition and Subtraction

Since subtraction is equivalent to addition with a negative number we can treat addition and subtraction as the same thing for the purposes of computing how error propagates.

Remark.
An obvious question presents itself now: why can we treat addition and subtraction as the same, but not treat multiplication and division as the same? Isn't division by y the same as multiplication by $1/y$? Mathematically, multiplication and division are basically the same, but notice that to multiply by $1/y$ we would have to actually compute what $1/y$ is. Given the decimal expansion of a number, computing $1/y$ is non-trivial: there is some actual work we'd have to do to find the decimal expansion of the inverse of y. For addition, however, life is easier. In particular, subtracting y is the same as adding $-y$ and it's completely trivial for us to find the decimal expansion of $-y$ given that of y. In terms of floating-point formats all we have to do is flip our sign bit.

Using the same sort of notation as before, letting ε_{x+y} represent the relative error in approximating $\text{fl}(x) + \text{fl}(y)$ by $\text{fl}(\text{fl}(x) + \text{fl}(y))$ we compute

$$\begin{aligned}
&\text{fl}(\text{fl}(x) + \text{fl}(y)) \\
=&\left[\text{fl}(x) + \text{fl}(y)\right](1 + \varepsilon_{x+y}) \\
=&\left[x(1+\varepsilon_x) + y(1+\varepsilon_y)\right] \cdot (1 + \varepsilon_{x+y}) \\
=&(x+y) \cdot \left[1 + \frac{x}{x+y}\varepsilon_x + \frac{y}{x+y}\varepsilon_y + \frac{x}{x+y}\varepsilon_x\varepsilon_{x+y} + \frac{y}{x+y}\varepsilon_y\varepsilon_{x+y} + \varepsilon_{x+y}\right] \\
\approx&(x+y) \cdot \left(1 + \frac{x}{x+y}\varepsilon_x + \frac{y}{x+y}\varepsilon_y + \varepsilon_{x+y}\right)
\end{aligned}$$

The relative error in addition is thus approximately

$$\frac{x}{x+y}\varepsilon_x + \frac{y}{x+y}\varepsilon_y + \varepsilon_{x+y}.$$

Notice that unlike the relative errors we had for multiplication and division, these relative errors depends on both x and y. In particular, the fractions $\frac{x}{x+y}$ and $\frac{y}{x+y}$ could actually be very large numbers. This happens when the denominators in these fractions are very small; when $x + y \approx 0$. (I.e., when we subtract two numbers which are very close to one another.)

7.4 Considerations for programming

We have seen above that the error in floating-point representations propagates differently for different arithmetic operations. In particular, multi-

plication and division are considered ***stable*** operations since the relative error grows slowly: all of the terms in our approximation of relative error are ε terms and are generally very small numbers. However, addition and subtraction are considered ***unstable*** since the relative error can grow quickly: the terms in our approximation of relative error have factors that are fractions of x and y and these fractions could be quite large.

These observations about the stability or unstability of arithmetic operations have practical consequences for computer programs, two of which we'll now demonstrate. As we will see, knowledge of how error propagates can help us to write computer programs which perform more accurate calculations. If we had a program which did some computation requiring additions and subtractions, but found some way to perform the same calculation using multiplications and divisions, there will be less concern about the error in our floating-point computations growing too quickly.

Roots of a quadratic

Consider for instance a quadratic equation which has been as

$$x^2 + 2bx - 1 = 0$$

where b is some fixed positive number. In various applications we may need to compute the roots of such a quadratic, and the most obvious way to do this is with the quadratic formula. The quadratic formula applied to the above tells us the roots are

$$\begin{aligned} x &= \frac{-2b \pm \sqrt{4b^2 + 4}}{2} \\ &= -b \pm \sqrt{b^2 + 2} \end{aligned}$$

Let's consider the solution with the positive root for a moment,

$$x = -b + \sqrt{b^2 + 2}.$$

Notice that since $b = \sqrt{b^2}$, we must have $b \approx \sqrt{b^2 + 2}$. Hence if we try to compute $-b + \sqrt{b^2 + 2}$ the error that occurs could be significant. However, we may be able to write our root in a way that avoids this subtraction of two very close numbers as follows:

$$\begin{aligned} & x^2 + 2bx - 1 = 0 \\ \implies & x^2 + 2bx = 1 \\ \implies & x(x + 2b) = 1 \\ \implies & x = \frac{1}{x + 2b}. \end{aligned}$$

Now keep in mind we know from the quadratic formula that x really does equal $-b + \sqrt{b^2 + 2}$, the issue is we don't want to compute this on a computer using floating-point numbers because of the error. However, when we replace the x in the denominator above with $-b + \sqrt{b^2 + 2}$ we have

$$\begin{aligned} x &= \frac{1}{x + 2b} \\ &= \frac{1}{-b + \sqrt{b^2 + 2} + 2b} \\ &= \frac{1}{b + \sqrt{b^2 + 2}} \end{aligned}$$

We should trust the computation

$$x = \frac{1}{b + \sqrt{b^2 + 2}}$$

more than the computation

$$x = -b + \sqrt{b^2 + 2}$$

because, even though mathematically they are the same, when performed on a computer $-b + \sqrt{b^2 + 2}$ is more likely to have greater relative error.

To make all of this concrete, let's actually do both computations and compare the results. Let's consider the quadratic of the above form where $b = 478$,

$$x^2 + 2 \cdot 478x - 1 = 0.$$

The true positive solution to this quadratic is

$$x = \sqrt{228485} - 478$$

which as a decimal is approximately 0.00104602396....

If we try to compute this using numbers stored in the IEEE double-precision floating-point stardard, then $x = \frac{1}{478 + \sqrt{478^2 + 2}}$ will give us

$$0.0010460228$$

whereas $x = -478 + \sqrt{478^2 + 2}$ will result in 0.002092045. The same number computed using divisions has *much* less error. In terms of significant digits, the first computation has six significant digits with the true value, whereas the second has zero!

In terms of relative error, the division calculation has a relative error of about

$$1.08685 \times 10^{-10}$$

whereas the relative error in the second computation (subtracting two nearby numbers) is about

$$-0.999972$$

Sums of several small numbers

As another example, let's consider computing the partial sum of the first one million terms of the harmonic series:

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{999999} + \frac{1}{1000000}$$

Of course, mathematically the order in which we perform the calculation shouldn't matter; it shouldn't matter if we compute the sum as above or as

$$\cdots \frac{1}{1000000} + \frac{1}{999999} + \cdots + \frac{1}{3} + \frac{1}{2} + 1.$$

However, if we evaluate these sums on a computer using the IEEE single-precision floating-point standard (this is similar to the double-precision standard we have discussed in class, but only 32 bits are used; this is the `float` data type in a language like C or Java), we will calculate different results.

In Matlab we can use single precision floating-point numbers by using `single` to store our numbers this way. (By default Matlab uses double precisions, so we have to explicitly tell it when we want to use single precision.) We'll compute these sums with `for`-loops that execute one million times, calling the sum of $1 + 1/2 + \cdots + 1/100000$ `forwards` and the sum of $1/1000000 + \cdots + 1/2 + 1$ `backwards`.

```
forwards = single(0);
for i = 1:10^6
    forwards = forwards + 1 / i;
end

backwards = single(0);
for i = 10^6:-1:1
    backwards = backwards + 1 / i;
end

fprintf("The forwards sum is %f\n", forwards);
fprintf("The backwards sum is %f\n", backwards);
```

Remark.
Notice that above to compute the backwards sum we used a `for` loop with `i = 10^6:-1:1`. This for loop starts at one million, 10^6, and decrements by 1 each time until 1 is reached. In general, a for loop with `i = a:b:c` will start at `a`, then add `b` after each iteration until `c` is reached.

Executing this code gives

```
The forwards sum is 14.357358
The backwards sum is 14.392652
```

Of course, mathematically these two sums are supposed to be equal, so what's going on? To understand why these two sums are different, let's consider a general summation whose terms are given by a sequence of real numbers a_1, a_2, ..., a_N,

$$S = \sum_{i=1}^{N} a_i.$$

In the following, we will let $\text{fl}(a_i) = a_i \cdot (1 + \varepsilon_i)$ for each a_i, and we will denote the sum of the first k floating point representations of the first k values of the a_i by S_k. Notice that we may write $S_k = a_k + S_{k-1}$ for $k \geq 2$.

The floating point representation of S_1 is really just the floating point representaiton of a_1:

$$S_1 = \text{fl}(a_1) = a_1 \cdot (1 + \varepsilon_1)$$

The floating point representation of S_2 requires we add a_2 onto S_1. This S_1 already has some error in it, and working through the arithmetic we see S_2 can be written as

$$\begin{aligned} S_2 &= \text{fl}(a_2 + S_1) \\ &= \text{fl}(a_2 + S_1) \cdot (1 + \varepsilon_2) \\ &= a_2(1 + \varepsilon_2) + S_1(1 + \varepsilon_2) \\ &= a_2(1 + \varepsilon_2) + a_1(1 + \varepsilon_1)(1 + \varepsilon_2) \end{aligned}$$

We can repeat this sort of procedure for S_3 as well to obtain

$$\begin{aligned} S_3 &= \text{fl}(a_3 + S_2) \\ &= (a_3 + S_2)(1 + \varepsilon_3) \\ &= a_3(1 + \varepsilon_3) + S_2(1 + \varepsilon_3) \\ &= a_3(1 + \varepsilon_3) + a_2(1 + \varepsilon_2)(1 + \varepsilon_3) + a_1(1 + \varepsilon_1)(1 + \varepsilon_2)(1 + \varepsilon_3) \end{aligned}$$

This pattern continues so that in general the k-th term of our summation is written as

$$\begin{aligned} S_k =& a_k(1 + \varepsilon_k) + a_{k-1}(1 + \varepsilon_{k-1})(1 + \varepsilon_k) + a_{k-2}(1 + \varepsilon_{k-2})(1 + \varepsilon_{k-1})(1 + \varepsilon_k) + \\ & \cdots + a_1(1 + \varepsilon_1)(1 + \varepsilon_2)\cdots(1 + \varepsilon_k) \end{aligned}$$

Notice that the earlier terms of the summation get multiplied by more error terms. Now, letting S denote the true value of the sum, the absolute error in approximating the sum by the N-th sum of floating point numbers S_N is

$$\begin{aligned} & S - S_N \\ =& \sum_{i=1}^{N} a_i - \sum_{i=1}^{N} a_i(1 + \varepsilon_i)(1 + \varepsilon_{i+1})\cdots(1 + \varepsilon_N) \\ =& \sum_{i=1}^{N} a_i \left(1 - (1 + \varepsilon_i)(1 + \varepsilon_{i+1})\cdots(1 + \varepsilon_N)\right) \\ =& \sum_{i=1}^{N} a_i \left(1 - [1 + \varepsilon_i + \varepsilon_{i+1} + \cdots + \varepsilon_N + \varepsilon_i\varepsilon_{i+1} + \varepsilon_i\varepsilon_{i+2} + \cdots \varepsilon_i\varepsilon_N + \cdots + \varepsilon_1\varepsilon_2\cdots\varepsilon_N]\right) \\ \approx& \sum_{i=1}^{N} a_i \left(1 - 1 - \varepsilon_i - \varepsilon_{i+1} - \cdots - \varepsilon_N\right) \\ =& \sum_{i=1}^{N} -a_i(\varepsilon_i + \varepsilon_{i+1} + \varepsilon_{i+2} + \cdots + \varepsilon_N) \end{aligned}$$

Again, this simply means the earlier a term appears in the summation, the more error it is going to be multiplied by in the final sum. Thus, to minimize the error we should rearrange the sum so that smaller values appear earlier in the sum. In terms of our `forwards` and `backwards` above, this means `backwards` is the more accurate estimate.

8 *Rootfinding*

The world looks like a multiplication-table, or a mathematical equation, which, turn it how you will, balances itself.

Ralph Waldo Emmerson

We now turn our attention away from technical issues of storing numbers in a computer and towards algorithms for solving mathematical problems. Many of these algorithms will require that we solve a system of linear equations, and before discussing those algorithms we should have a review of some basic linear algebra. Before doing that, however, we will go ahead and mention some common algorithms for finding the solutions of an equation which will not require any linear algebra.

Suppose we wish to find the solutions to a particular eqation of the form $f(x) = y_0$, or perhaps $f(x) = g(x)$. Notice first the we may always assume the right-hand side of the equation is zero. By subtracting anything on the right-hand side to the left, such as $f(x) - y_0 = 0$ or $f(x) - g(x) = 0$. For this reason we may always assume our equation is written in the form $f(x) = 0$, and we want to find a value of x that makes this true.

In very simple cases this may simply be an issue of doing the necessary algebra, but in most cases we won't be able to do do the algebra to explicitly solve for x. For example, it's not so clear how to find an x such that

$$\sqrt{x} + x^3 - 13 = 0,$$

or

$$\cos(x) = \sin(x^2).$$

In situation such as these, we may resort to approximating a solution of the equation numerically with a computer. There are many different ways of doing this, but we will discuss the three most important algorithms: bisection, Newton's method, and the secant method. None of these algorithms is perfect; each has its own pros and cons. As we will see, Newton's method is typically very fast, but may not converge to a root. The bisection algorithm will always converge to a root (under certain conditions), but is rather slow.

8.1 The bisection algorithm

Suppose we wish to approximate a solution to the equation $f(x) = 0$. First we might want to know that such a solution actually exists before we spend a lot of time trying to find the solution. One simple condition guaranteeing a solution is the following: if f is continuous on an interval $[a, b]$ and if $f(a) < 0$ and $f(b) > 0$ (or vice versa), then by the intermediate value theorem there must exist an $x \in [a, b]$ such that $f(x) = 0$. Throughout this section we will assume $f(x)$ is continuous and that we have found an interval $[a, b]$ where $f(a)$ and $f(b)$ have opposite signs. A concise way to say this is that $f(a) \cdot f(b) < 0$.

To estimate the solution we will try to "zero in" on a smaller interval which must contain the true solution. We do this by chopping the interval $[a, b]$ into two halves. Letting $c = \frac{a+b}{2}$ denote the midpoint between a and b, we consider the intervals $[a, c]$ and $[c, b]$. Notice that one of three things must happen:

1. $f(c) = 0$, in which case $x = c$ is the solution;
2. $f(a)f(c) < 0$, in which case the solution is contained in the interval $[a, c]$; or
3. $f(c)f(b) < 0$, in which case the solution is contained in the interval $[c, b]$.

That is, we can cut our range of possible values for the solution in half very easily. Repeating this procedure we can in principle create arbitrarily small intervals which are guaranteed to contain the solution. In particular, if we want to approximate the solution to with some ε distance of the true value, we can just iterate this procedure, continually chopping our intervals in half, until we arrive at an interval of length less than ε, and then use either endpoint of the interval as our approximation to the solution.

At each step of the iteration we check to see if our interval is less than the desired size ε. If so, we stop and return one of the endpoints. If not, we determine the midpoint c of the interval. We then simply update the endpoints of our interval by replacing the left-hand endpoint a with c if we determine the root is in the right-hand interval; or replacing the right-hand endpoint b with c if we determine the root is in the left-hand interval.

Since you will need to implement the bisection algorithm yourself in a homework assignment, we won't give the Matlab code here, but we instead give *pseudo-code*. That is, we give the basic idea of how the code should work without writing down the actual Matlab.

```
Given a function f, two endpoints a, b,
and a desired accuracy epsilon.

Check to see if f(a) * f(b) < 0.  If not, create
an error.

While b - a > epsilon:
    Set c to be (a + b) / 2

    If f(c) * f(a) < 0:
        Update b to be c
    Else:
        Update a to be c

Return the last value of b
```

Notice in the pseudo-code above we did not actually check to see if the c we computed was a root of f or not. The vast majority of the time we will never find the true root, so we don't bother to check in the above. If you wanted to, you could update the code to explicitly check if c was a root and if so then prematurely end the loop. (In Matlab this is done with the keyword `break`.)

To illustrate the idea, let's work through an explicit example. Suppose wanted to estimate a value of x solving $\cos(x) = \sin(x^2)$ in the interval $[0, \pi/2]$, and we wanted our estimate to be within $1/100$ of the true solution. Then our function f would be $f(x) = \cos(x) - \sin(x^2)$, our endpoints would be $a = 0$ and $b = \pi/2 \approx 1.57079633$, and $\varepsilon = 0.01$.

We first compute that $f(a) = 1$ and $f(b) = -0.6242$. Since this produce is negative, we begin the loop.

On the first iteration, $b-a$ is $1.57079633-0$ which is larger than 0.01. We compute $c = \frac{a+b}{2} = \frac{1.57079633}{2} = 0.78539816$. Now we multiply $f(c) \cdot f(a) = 0.78539816 \cdot 1$. This is positive, so the condition in the `if` statement is false, and we execute the code in the `else` clause. That is, we will update a to be 0.78539816, and leave b alone.

On the second iteration, $b - a = 1.57079633 - 0.78539816 = 0.78539816$. This is greater than 0.01, so the loop continues. We now compute $c = \frac{a+b}{2} = 1.1780725$, then multiply $f(c) \cdot f(a) = -0.606399 \cdot 0.12863799$. This is less than zero, so the condition in the `if` statement is true, and we update b to be the current value of c, 1.1780725, and we leave the value of a alone.

At this point we know the solution is between $a = 0.78539816$ and $b = 1.1780725$.

On the third iteration we check $b - a = 0.39269908$. This is greater than 0.01, so the loop continues. We compute $c = 0.88357293$, then check $f(c) \cdot f(a)$ is negative. It is, so we update $b = 0.88357293$ and leave a alone.

On the fourth iteration we compute $b - a = 0.19634959$, this is greater than $\varepsilon = 0.01$, so the loop continues.

In this way we continue to update a and b until their difference is less than 0.01. Noticing that the distance between a and b halves at each step. For this reason the process can not continue forever: if we take any positive number and cut it in half enough times, it will eventually get smaller than any other positive number. That is, $\lim_{n\to 0} \frac{b-a}{2^n} = 0$ and so it will eventually be less than ε. We can actually be more precise about exactly how long it will take the bisection algorithm to get down to an interval of length less than any given ε.

Suppose that we let a_n and b_n denote the endpoints of the interval we have after the n-th iteration of the bisection algorithm. Since the intervals are halved at each step we have

$$b_{n+1} - a_{n+1} = \frac{1}{2}(b_n - a_n).$$

We then easily see that $b_n - a_n = \frac{1}{2^n}(b-a)$ by substitution:

$$\begin{aligned}
b_1 - a_1 &= \frac{1}{2}(b-a) \\
\implies b_2 - a_2 &= \frac{1}{2}(b_1 - a_1) \\
&= \frac{1}{2}\cdot\frac{1}{2}(b-a) \\
&= \frac{1}{2^2}(b-a) \\
\implies b_3 - a_3 &= \frac{1}{2}(b_2 - a_2) \\
&= \frac{1}{2}\cdot\frac{1}{2^2}(b-a) \\
&= \frac{1}{2^3}(b-a) \\
\implies b_4 - a_4 &= \frac{1}{2}(b_3 - a_3) \\
&= \frac{1}{2}\cdot\frac{1}{2^3}(b-a) \\
&= \frac{1}{2^4}(b-a) \\
&\vdots \\
\implies b_n - a_n &= \frac{1}{2^n}(b_n - a_n).
\end{aligned}$$

Letting α temporarily denote the true root of the function, we notice that the distance between a_n and α goes to zero. Since $a_n \leq \alpha \leq b_n$ we have

$$|\alpha - a_n| \leq b_n - a_n = \frac{1}{2^n}(b-a).$$

Similarly, $|\alpha - b_n| \leq \frac{1}{2^n}(b-a)$. Thus both sequences of endpoints approach the true solution α as n goes to infinity.

We want to iterate the bisection algorithm until $b_n - a_n$ is less than ε. We see easily that this must happen, but how many iterations does it require? Suppose the number of required steps was N, then from our computations above we see that

$$\begin{aligned} b_N - a_N &< \varepsilon \\ \implies \frac{1}{2^N}(b-a) &< \varepsilon \\ \implies \frac{b-a}{\varepsilon} &< 2^N \\ \implies \log_2\left(\frac{b-a}{\varepsilon}\right) &< N \end{aligned}$$

That is, we want the number of iterations N to be the smallest integer such that $N > \log_2(b-a/\varepsilon)$ and this is simply the ceiling of this number. This proves the following proposition.

Proposition 8.1.
Starting with endpoints a and b, the bisection algorithm will approximate a solution to $f(x) = 0$ with ε distance of the true solution after

$$\left\lceil \log_2\left(\frac{b-a}{\varepsilon}\right)\right\rceil$$

iterations.

To estimate the solutions to $\cos(x) = \sin(x^2)$ starting in the interval $[0, \pi/2]$ within $\varepsilon = 0.01$ of the true solution, for example, will require eight iterations as

$$\left\lceil \log_2\left(\frac{\pi/2 - 0}{0.01}\right)\right\rceil = \lceil 7.2953... \rceil = 8.$$

8.2 Newton's method

Another common root finding algorithm, called *Newton's method*, has the advantage of typically being faster than the bisection algorithm (that is, it requires fewer iterations to achieve an approximation which is in within a some desired distance of the true solution), and is often taught in first-semester calculus classes. As we will see, analyzing Newton's method will require some more sophisticated ideas than were used to analyze the bisection algorithm above, and so we will have to interrupt our discussion of Newton's method to recall some facts about linear algebra and calculus.

The basic idea behind Newton's method is that if we have a differentiable function f, then we can approximate f near a point x_0 by the line tangent to the graph $y = f(x_0)$ at $(x, f(x_0))$. Since the slope of this line is the derivative $f'(x_0)$, the equation of the tangent line is

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

Instead of determining where $y = f(x)$ crosses the x-axis, we will determine where this tangent line crosses the x-axis. This is extremely easy because we simply plug 0 in for the y above and then solve for x:

$$\begin{aligned} 0 - f(x_0) &= f'(x_0) \cdot (x - x_0) \\ \implies \frac{-f(x_0)}{f'(x_0)} &= x - x_0 \\ \implies x &= x_0 - \frac{f(x_0)}{f'(x_0)}. \end{aligned}$$

We will treat this x-coordinate we've calculated, call it x_1, as a new approximation to the root of $f(x)$. Repeating the procedure from that point we can find the tangent line to $y = f(x)$ at x_1, and determine where that line crosses the x-axis, call that point x_2:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}.$$

We can then repeat this process to find a third approximation,

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)},$$

and so on. In general, given the n-th approximation x_n, we compute the next approximation x_{n+1} by

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

Example 8.1.
Use Newton's method to approximate the solution to

$$x^3 + 3x^2 + x - 40 = 0.$$

Here we have

$$f(x) = x^3 + 3x^2 + x - 40$$
$$f'(x) = 3x^2 + 6x + 1$$

Thus we compute succives approximations using the formula

$$x_{n+1} = x_n - \frac{x_n^3 + 3x_n^2 + x_n - 40}{3x_n^2 + 6x_n + 1}$$

Notice that we need a starting point x_0 from which we will begin our approximations. We will semi-arbitrarily chose $x_0 = 4$ as our stating point. The sequence of numbers generated by the formula above is then

$$x_0 = 4.000000000000000$$
$$x_1 = 2.958904109589041$$
$$x_2 = 2.622826786744639$$
$$x_3 = 2.587950413448695$$
$$x_4 = 2.587590568591517$$
$$x_5 = 2.587590530523929$$

Notice that in the example above the values Newton's method is giving us seem to have stabilized to the ten-millionths place! (The first seven digits after the decimal place have stabilized; 10^7 is ten million, 10^{-7} is one ten millionth.) For the sake of comparison, supposed we used the bisection algorithm to approximate a root of the polynomial in the example above to within one ten millionth of the true value starting with the interval $[a, b] = [2, 4]$. It would take the bisection algorithm

$$\left\lceil \log_2 \left(\frac{4-2}{10^{-7}} \right) \right\rceil = 25$$

It will take the bisection algorithm five times as long to compute the same root!

We would like to analyze Newton's method more precisely, and to do this we will need to use Taylor's remainder theorem. We will need Taylor's remainder theorem at other points in the semester as well, so this is a good time to spend a little while reviewing some facts about Taylor polynomials

and Taylor's theorem. Since Taylor polynomials come from solving a certain system of linear equations, and we will need to solve lots of various systems of linear equations in discussing other algorithms, however, we will first spend some time review the basics of linear algebra.

After the necessary background in linear algebra and calculus has been established, we will come back to this issue of analyzing Newton's method.

Part III

Review of linear algebra and calculus

9 Linear algebra

Algebra is the metaphysics of arithmetic.

John Ray

Now that we know the basics of Matlab and how numbers are stored in a computer, we're almost ready to start applying what we've learned to see how to use a computer to solve mathematical problems. In particular, we are going to be interested in developing and implementing algorithms for solving some common types of mathematical problems. Most of these algorithms will boil down to solving a system of linear equations, and some will also require some basic facts from calculus. In this chapter we recall some of the basic theory about linear algebra, and in the next we will recall some basic properties of Taylor polynomials. For the sake of time our treatment will not be completely comprehensive, but we will cover the bits of linear algebra and calculus that are most relevant to us.

9.1 Systems of linear equations

A ***linear equation*** in n variables x_1, x_2, ..., x_n is an equation which may be written as

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b$$

where a_1, a_2, ..., a_n, and b are real or complex numbers, and the x_1, ..., x_n are ***unknowns***. The constants a_1, a_2, ..., a_n are called the ***coefficients*** of the equation.

Example 9.1.
Each of the following is a linear equation:

(a) $6x + 2y = 3$

(b) $x_1 - x_2 + 3x_3 + \frac{22}{7}x_4 = 2$

(c) $x - y + z = 0$

Example 9.2.
The following *are not* linear equations:

(a) $xy = 1$

(b) $\frac{x_1+x_2}{x_3} = x_4 - x_5$

(c) $x^2 = x + y$

Let's go ahead and notice at this point that the set of all solutions to a linear equation in two variables gives us a line. For example, the collection of all (x, y) pairs that satisfy the linear equation

$$6x + 2y = 3$$

is a line. This might be easiest to see if we take our linear equation and rewrite in the more familiar slope-intercept form of a line by solving for y:

$$\begin{aligned} & 6x + 2y = 3 \\ \implies & 2y = -6x + 3 \\ \implies & y = -3x + \frac{3}{2} \end{aligned}$$

This is a line of slope -3 which passes through the point $(0, 3/2)$ as in Figure 9.1 on the following page.

It is because of this graphical interpretation that equations of the form $ax + by = c$ in two dimensions give lines that we call the functions above *linear* functions and equations with linear functions are *linear equations*. Notice that in three dimensions the set of solutions to a linear equation give a plane and not a line, as in Figure 9.2, but we still use the term "linear."

A ***system of linear equations*** is a collection of linear equations, all in the same number of variables.

Example 9.3.
Each of the following are systems of linear equations:

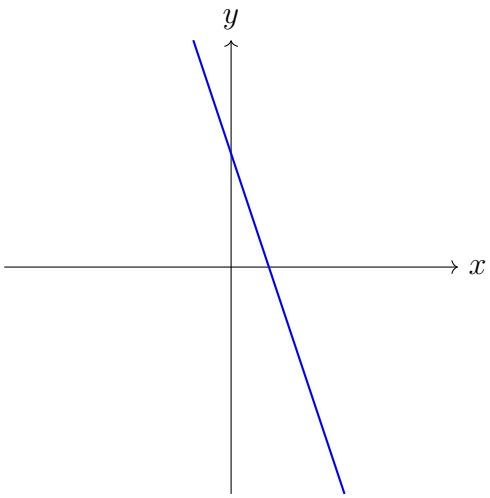


Figure 9.1: The set of points satisfying the linear equation $6x + 2y = 3$ is a line in the plane.

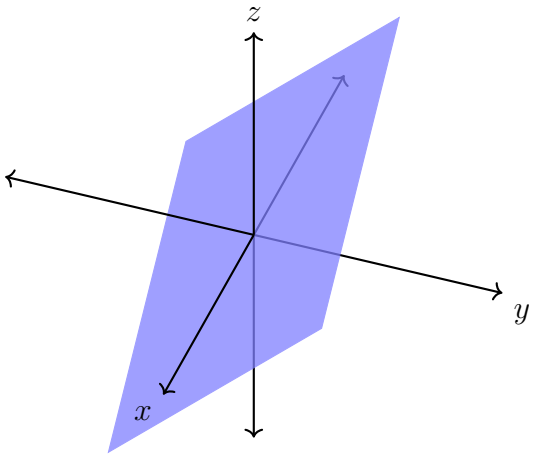


Figure 9.2: The set of points satisfying the linear equation $x - y + z = 0$ is a plane in 3-space.

(a)

$$\begin{aligned} 3x + 2y &= 4 \\ 6x - y &= 9 \end{aligned}$$

(b)

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

(c)

$$\begin{aligned} x + y + z &= 3 \\ 2x - y + 3z &= 0 \end{aligned}$$

A ***solution*** to a system of a linear equations is a collection of numbers, one for each variable, which makes *all* of the equations true simultaneously. In the case of Example 9.3(a) it's easy to check that $(x, y) = (22/15, -1/5)$ is a solution to the system by simply plugging $x = 22/15$ and $y = -1/5$ into each equation in the system and verifying that both equations are true:

$$\begin{aligned} 3 \cdot \frac{22}{15} + 2 \cdot \frac{-1}{5} &= \frac{22}{5} - \frac{2}{5} \\ &= \frac{20}{5} \\ &= 4 \end{aligned}$$

$$\begin{aligned} 6 \cdot \frac{22}{15} - \frac{-1}{5} &= \frac{44}{5} + \frac{1}{5} \\ &= \frac{45}{5} \\ &= 9 \end{aligned}$$

In Example 9.3(b) it's also easy to see that there are *no solutions* to the system: there is no choice of x and y that can make $4x + 3y = 3$ and $4x + 3y = 2$ at the same time, since $3 \neq 2$.

It's a little bit harder to see, but there are actually infinitely-many different solutions to Example 9.3(c). Let's try to explain why this is. If we solve the first equation for y we have

$$y = 3 - x - z.$$

So triple (x, y, z) solving the system has to also satisfy this equation (since this is just the first equation rewritten). Now if (x, y, z) is a solution to the system, then it must also solve the second equation as well as $y = 3 - x - z$. This means we can rewrite the second equation as

$$2x - (3 - x - z) + 3z = 0.$$

If we now solve this equation for z we have

$$\begin{aligned} & 2x - (3 - x - z) + 3z = 0 \\ \implies & 3x + 4z - 3 = 0 \\ \implies & z = \frac{3 - 3x}{4}. \end{aligned}$$

If we now plug this back into $y = 3 - x - z$ we have

$$\begin{aligned} y =& 3 - x - \frac{3 - 3x}{4} \\ =& \frac{12 - 4x}{4} - \frac{3 - 3x}{4} \\ =& \frac{9 - x}{4}. \end{aligned}$$

So, what does this mean? It means if (x, y, z) is a solution to the system, then y and z are both determined by x:

$$y = \frac{9 - x}{4} \quad \text{and} \quad z = \frac{3 - 3x}{4}.$$

Here x can be whatever value you'd like (in a situation like this we sometimes call x a ***free variable***) and once you've chosen x, you know what y and z must be. Since there are infinitely-many different choices for x (x can be any real number you'd like), there are infinitely-many solutions.

Right now the above algebra probably seems tedious – easy, but a little bit of boring work to figure out. We will quickly see that there are some algorithms that make finding solutions to systems like this much easier.

Solution Sets

Given a system of linear equations our goal will typically be to find all possible solutions to the system. The collection of all possible solutions is called the ***solution set*** of the system. In principle, the solution set of an arbitrary system of equations could be very complicated, but for systems of *linear* equations, the solution sets are actually very nice.

In fact, the solution set of a system of linear equation comes in one of three flavors: it could be empty (no solutions), it could contain exactly one point (a unique solution), or it could contain infinitely-many points. Let's think about why this is in two variables.

Let's suppose that you had a system of linear equations in two variables: say there are n equations, and the i-th equation has the form $a_i x + b_i y = c_i$, so the system looks something like the following:

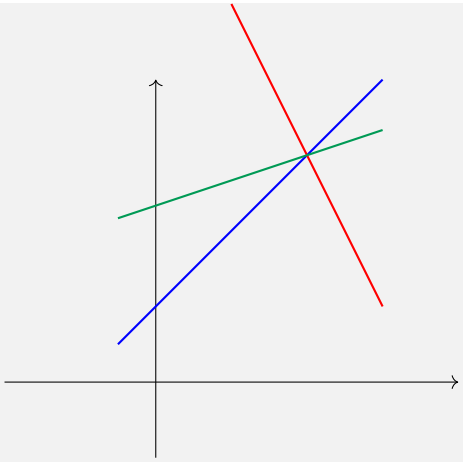
$$\begin{aligned} a_1 x + b_1 y &= c_1 \\ a_2 x + b_2 y &= c_2 \\ &\vdots \\ a_n x + b_n y &= c_n. \end{aligned}$$

Each one of the equations determines a line in the plane. The solution set of the system is the collection of points that are simultaneously on all of the lines. It could be that all of the lines intersect at a single point giving a unique solution; it could be that no point is on all of the lines at the same time (no solution); or it could be that all the lines are actually the same and there are infinitely-many solutions (every point on the line is a solution).

Let's consider one example of each situation just by considering the graphs of the lines.

Example 9.4.
The following system has one unique solution:

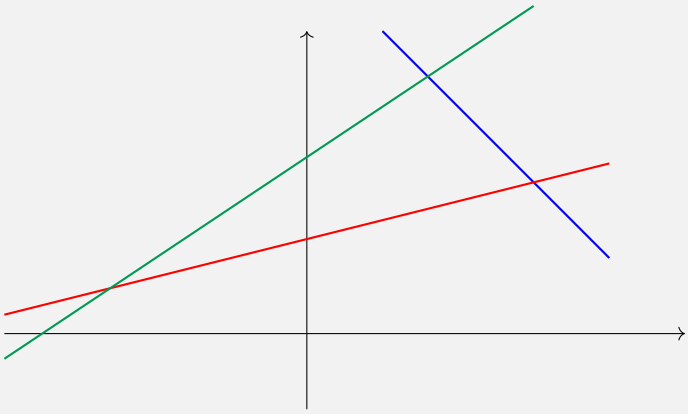
$$\begin{aligned} x - y &= -1 \\ 2x + y &= 7 \\ -3x + 9y &= 21 \end{aligned}$$



Here there is a unique solution because there is exactly one point that is on all three lines.

Example 9.5.
The following system has no solutions:

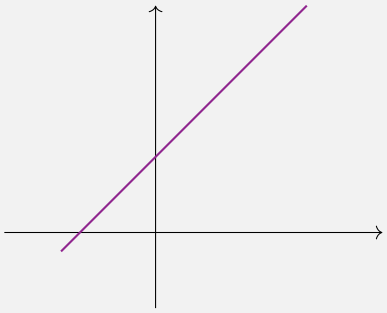
$$\begin{aligned} x + y &= 5 \\ -2x + 8y &= 10 \\ 2x - 3y &= 7 \end{aligned}$$



There are no solutions because there is not a point that is on all three lines simultaneously.

Example 9.6.
The following system has infinitely-many solutions:

$$\begin{aligned} -x + y &= 1 \\ 2x - 2y &= -2 \\ -3x + 3y &= 3 \end{aligned}$$



In this case all three lines are actually the same line on top of one another, so any solution to one equation is instantly a solution to both of the other equations.

The same situation can happen in any number of variables: regardless of whether your system of equation is in 2 variables, 3 variables, or 457 variables, a system of linear equations has either no solutions, one unique solution, or infinitely-many solutions.[1]

Whenever a system of equations has a solution (regardless of whether it has one solution or infinitely-many) we say the system is ***consistent***. If the system does not have any solutions, however, we say the system is ***inconsistent***. So the systems in Example 9.4 and Example 9.6 are consistent, while the system in Example 9.5 is inconsistent.

Solving a System of Linear Equations

Now that we what a linear system is and the different "flavors" the solution can come in, how do we go about determining the solutions?

[1]This is true if we're talking about real or complex linear systems. We will see later in the semester that there are some applications where we'd like to use another type of number system, and in that case it could be possible to have only finitely-many distinct solutions to a system.

Let's consider "manually" solving a couple of different systems before we try to come up with an algorithm.

Example 9.7.
Solve the following system:

$$\begin{aligned} x - y &= 7 \\ y &= 3 \end{aligned}$$

This is a system everyone can solve without any knowledge of linear algebra: the second equation tells us explicitly what y has to be, so all of the (x, y) solutions to the system have to take y to be three. Plugging this back into the first equation we then have $x - 3 = 7$ and we can easily solve $x = 10$. Thus $(x, y) = (10, 3)$ is a solution, the unique solution, to this system.

Example 9.8.
Solve the following system:

$$\begin{aligned} x - y &= 7 \\ 2x - y &= 17 \end{aligned}$$

To solve this system we might first try to isolate a variable in one of the equations. For example, if we subtract twice the first equation from the second equaton we would have

$$\begin{aligned} & 2x - y - 2(x - y) = 17 - 2 \cdot 7 \\ \implies & 2x - y - 2x + 2y = 17 - 4 \\ \implies & y = 3 \end{aligned}$$

We are then back to exactly the same situation as the previous example and so the solution is again $(10, 3)$.

When two different systems of equations have the same set of solutions, such as the examples above, we say the systems are ***equivalent***.

Notice that certain types of systems of equations are very easy to solve, and others can seem more complicated. Here is another easy example:

Example 9.9.
Solve the following system

$$\begin{aligned} 2x + 3y - z &= 4 \\ y + 2z &= 2 \\ z &= -1 \end{aligned}$$

This system is very easy to solve because one of the equations instantly tells us what one of the variables has to be: we know that z must be -1. If we plug this into the second equation we have $y - 2 = 2$, so $y = 4$. Now that we know y and z, we can plug back into the first equation to determine $2x + 12 + 1 = 4$, and thus $x = \frac{-9}{2}$.

We really like the types of systems as in the last example because they are almost trivial to solve: we just plug back into our equations and get one variable at a time. We would like it, then, if when given a more complicated system we were somehow able to determine an equivalent, but easy-to-solve, system. Since the systems are equivalent, solving the easy system tells us the solution to the easier system.

The main question, then, is how do we determine if two systems are equivalent?

To do this, let's come up with a list of some simple operations that we can perform on a system of equations to come up with an equivalent system. What we are about to describe will work for any number of systems in any number of variables, so we'll state things very generally but then do some simpler examples.

Theorem 9.1.
If two rows in a linear system are exchanged, the newly obtained system is equivalent to the original one.

Proof.
The equations defining the system haven't been changed, just reordered. □

Example 9.10.
The following two systems are equivalent:

$$\begin{aligned} 2x + 3y - z &= 4 \\ x - y + 3z &= 9 \\ -x + 6y + 4z &= -2 \end{aligned} \qquad \begin{aligned} -x + 6y + 4z &= -2 \\ x - y + 3z &= 9 \\ 2x + 3y - z &= 4 \end{aligned}$$

Theorem 9.2.
If one equation in a linear system is modified by adding a multiple of another equation to it, the newly obtained system is equivalent to the original one.

Before proving this theorem in general, let's consider a very simple case: two variables and two equations. Say our system looks like

$$\begin{aligned} a_1 x + a_2 y &= b \\ \alpha_1 x + \alpha_2 y &= \beta \end{aligned}$$

Suppose the system is consistent and so there's some point (s_1, s_2) that satisfies the system: if we plug in $x = s_1$ and $y = s_2$, then both equations are solved simultaneously.

Now say that we modify the system by adding a multiple of the second equation to the first. That is, we will replace the first equation by adding c times the second equation to it, for some constant c. We then have the following system:

$$\begin{aligned} (a_1 + c\alpha_1)x + (a_2 + c\alpha_2)y &= b + c\beta \\ \alpha_1 x + \alpha_2 y &= \beta \end{aligned}$$

We claim that (s_1, s_2) is still a solution to this system. Since (s_1, s_2) satisfied the second equation before, and that second equation hasn't changed, all we need to do is verify that (s_1, s_2) satisfies the modified first equation, but this is easy:

$$\begin{aligned}&(a_1 + c\alpha_1)s_1 + (a_2 + c\alpha_2)s_2\\ &= a_1s_1 + c\alpha_1s_1 + a_2s_2 + c\alpha_2s_2\\ &= (a_1s_1 + a_2s_2) + c(\alpha_1s_1 + \alpha_2s_2)\\ &= b + c\beta\end{aligned}$$

A solution to the original system is thus a solution to this modified system as well. This shows that the solution set of the first system is a subset of the solution set of the second system. We still need to show that a solution to the modified system is also a solution to the original system, but the idea is basically the same as the above, so we will leave that as an exercise.

Exercise 9.1.
Show that if (t_1, t_2) is a solution to the system

$$\begin{aligned}(a_1 + c\alpha_1)x + (a_2 + c\alpha_2)y &= b + c\beta\\ \alpha_1x + \alpha_2y &= \beta,\end{aligned}$$

then it is also a solution to the system

$$\begin{aligned}a_1x + a_2y &= b\\ \alpha_1x + \alpha_2y &= \beta.\end{aligned}$$

Suppose (t_1, t_2) is a solution to the system. I.e.,

$$\begin{aligned}(a_1 + c\alpha_1)t_1 + (a_2 + c\alpha_2)t_2 &= b + c\beta\\ \alpha_1t_2 + \alpha_2t_2 &= \beta.\end{aligned}$$

Now subtract c times the second equation from the first equation. On the left-hand side this gives

$$\begin{aligned}&(a_1 + c\alpha_1)t_1 + (a_2 + c\alpha_2)t_2 - c\,(\alpha_1t_1 + \alpha_2t_2)\\ =&a_1t_1 + a_2t_2\end{aligned}$$

On the right-hand side we have $b + c\beta - c\beta = b$. Thus (t_1, t_2) satisfies

$$a_1 t_1 + a_2 t_2 = b.$$

Hence (t_1, t_2) is a solution to the system

$$\begin{aligned} a_1 x + a_2 y &= b \\ \alpha_1 x + \alpha_2 y &= \beta. \end{aligned}$$

Proving the general theorem is basically repeating the same argument above, just with more equations and variables.

Proof of Theorem 9.2.
Consider a linear system of m equations in n variables. Say two of the equations in this system are

$$a_1 x_1 + \cdots + a_n x_n = b \quad \text{and} \quad \alpha_1 x_1 + \cdots + \alpha_n x_n = \beta.$$

We want to leave the second equation alone, but replace the first equation with

$$(a_1 + c\alpha_1)x_1 + \cdots + (a_n + c\alpha_n)x_n = b + c\beta$$

for some constant c. The claim is that doing so doesn't change the set of solutions.

Let's call the set of solutions to the original system S, and the set of solutions to the modified system T. We want to show these two sets are the same: we want to show that $S = T$, which means we need to show that $S \subseteq T$ and $T \subseteq S$.

Let $(s_1, ..., s_n) \in S$ be a solution to the original system. We need to show this is also a solution to the modified system. Of the equations defining the systems, however, $m - 1$ of the equations are the same. So the only thing we need to check is that $(s_1, ..., s_n)$ is also a solution to

$$(a_1 + c\alpha_1)x_1 + \cdots + (a_n + c\alpha_n)x_n = b + c\beta$$

We simply plug in $(x_1, ..., x_n) = (s_1, ..., s_n)$ and verify that the equation holds:

$$\begin{aligned}&(a_1 + c\alpha_1)s_1 + \cdots + (a_n + c\alpha_n)s_n\\ =&a_1s_1 + c\alpha_1s_1 + \cdots + a_ns_n + c\alpha_ns_n\\ =&(a_1s_1 + \cdots + a_ns_n) + c(\alpha_1s_1 + \cdots + \alpha_ns_n)\\ =&b + c\beta\end{aligned}$$

This shows that $S \subseteq T$.

We leave the second part of the proof, that $T \subseteq S$, as an exercise. □

Exercise 9.2.
Finish the proof of Theorem 9.2. Keeping the same notation as in the proof of the first part of Theorem 9.2, we must show that $T \subseteq S$. Let $(t_1, ..., t_n) \in T$ be a solution to the modified system in which the equation

$$a_1x_1 + \cdots + a_nx_n = b$$

has been replaced by

$$(a_1 + c\alpha_1)x_1 + \cdots + (a_n + c\alpha_n)x_n = b + c\beta.$$

We simply need to show that $(t_1, ..., t_n)$ also satisfies our original first equation,

$$a_1x_1 + \cdots + a_nx_n = b.$$

Note that $(t_1, ..., t_n)$ satisfies the following:

$$(a_1 + c\alpha_1)t_1 + \cdots + (a_n + c\alpha_n)t_n = b + c\beta.\alpha_1t_1 + \cdots + \alpha_nt_n = \beta$$

If we subtract c times the second equation from the the first, we have on the left-hand side

$$\begin{aligned}&(a_1 + c\alpha_1)t_1 + \cdots + (a_n + c\alpha_n)t_n - c\left(\alpha_1t_1 + \cdots + \alpha_nt_n\right)\\ =&a_1t_1 + \cdots + a_nt_n\end{aligned}$$

and on the right-hand side, $b + c\beta - c\beta = b$. Hence $(t_1, ..., t_n)$ is a solution to the original system of equations; $(t_1, ..., t_n) \in S$ and so $S \subseteq T$.

Finally, there's one last operation that we will introduce that can be used to replace one system of equations with an equivalent one.

Theorem 9.3.
If each term (both the left- and right-hand sides) of one equation is multipled by a nonzero constant c, then the newly obtained system is equivalent to the original system.

Again, let's consider what's happening with two variables and two equations. If our original system was

$$a_1x + a_2y = b$$
$$\alpha_1x + \alpha_2y = \beta$$

then we claim that the following system is equivalent

$$ca_1x + ca_2y = cb$$
$$\alpha_1x + \alpha_2y = \beta$$

when c is any nonzero constant.

To prove this, again suppose that (s_1, s_2) is a system to the original system. We can easily verify that (s_1, s_2) solves the modified system. Of course, the second equation has remained the same, so (s_1, s_2) still satisfies it. For the first equation we have

$$\begin{aligned} & ca_1s_1 + ca_2s_2 \\ =& c(a_1s_1 + a_2s_2) \\ =& cb \end{aligned}$$

To prove that a solution (t_1, t_2) to the modified system is also a solution to the original system we can perform the exact same procedure: just multiply through by $\frac{1}{c}$ to get the c's to cancel! In more variables and/or equations, the argument is exactly the same, so we will leave that as an exercise.

Exercise 9.3.
Prove Theorem 9.3. Suppose that we are given a system of equations

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n &= b_n \end{aligned}$$

and we modify the system by replacing one of the equations with some non-zero multiple c of itself. Without loss of generality, we may assume this is the first equation. (If it were a different equation, we could swap two equations to make it the first equation.)

Let S be the set of solutions to the original system, and T the set of solutions to the modified system. Obviously if $(s_1, ..., s_n) \in S$ satisfies

$$a_{11}s_1 + \cdots + a_{1n}s_n = b_1,$$

then in the modified system we simply have

$$ca_{11}s_1 + \cdots + ca_{1n}s_n = c(a_{11}s_1 + \cdots + a_{1n}s_n) = cb$$

and so $S \subseteq T$ as we have a solution to the modified system.

Likewise, if $(t_1, ..., t_n)$ satisfies the first equation in the modified system,

$$ca_{11}t_1 + \cdots + ca_{1n}t_n = cb_1,$$

then

$$a_{11}t_1 + \cdots + a_{1n}t_n == \frac{1}{c}\left(ca_{11}t_1 + \cdots + ca_{1n}t_n\right) = \frac{1}{c} \cdot cb_1 = b_1$$

and thus $T \subseteq S$.

A Procedure for Solving Linear Systems

We now want to use the three theorems above to develop a scheme for solving systems of linear equations, or determining that there is no solution. As we saw in an earlier example, it would be nice if the system we wanted

to solve had the following sort of form:

$$\begin{aligned} ax + by &= \alpha \\ cy &= \beta \\ fz & \end{aligned} \qquad\qquad \begin{aligned} ax + by + cz &= \alpha \\ dy + ez &= \beta \\ &= \gamma \end{aligned}$$

In this situation it is super-easy to "work backwards," solving for one variable at a time, and then determining the others. Let's give systems of this form a special name so it's easier to refer to them: we will call a system like this is in ***echelon form***.

To solve a system that is not in echelon, let's try to replace the system with an equivalent system that *is* in echelon form. We'll do this by repeatedly applying our three theorems above, modifying the system a little bit at a time until it is in the form we'd like.

Example 9.11.
Solve the following system of equations.

$$\begin{aligned} x + 4y &= 3 \\ 2x - y &= 1 \end{aligned}$$

All we need to do to put the system in echelon form is get rid of the $2x$ in the second equation. We can do this by subtracting twice the first equation from the second. We will then replace the second equation with

$$\begin{aligned} 2x - y - 2(x + 4y) &= 1 - 2 \cdot 3 \\ \implies -9y &= -5 \end{aligned}$$

thus $y = \frac{5}{9}$, and pluggin this back into the first equation,

$$\begin{aligned} x + 4 \cdot \frac{5}{9} &= 3 \\ \implies x = 3 - \frac{20}{9} &= \frac{7}{9} \end{aligned}$$

and so the system has a unique solution, $(x, y) = (5/9, 7/9)$.

Example 9.12.
Solve the following system of equations.

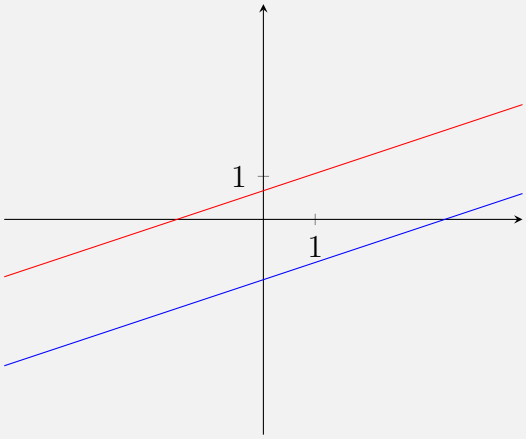
$$\begin{aligned} 2x - 5y &= 7 \\ -6x + 15y &= 10 \end{aligned}$$

We again try to put the system into echelon form by getting rid of the $-6x$ in the second equation. To do this, we add three times the first equation to the second:

$$\begin{aligned} &-6x + 15 + 3(2x - 5y) = 7 + 3 \cdot 10 \\ \implies &0 = 37 \end{aligned}$$

Now we have a problem: zero is not equal to thirty-seven! What this means is that there is no solution to the system.

Geometrically, the lines determined by each of the equations above are parallel. A solution to the system would be where the two lines intersect, but two parallel lines never intersect, hence there is no solution.



Example 9.13.

Solve the following system:

$$\begin{aligned} 3x + 2y - z &= 3 \\ 12x - 4y + 2z &= 1 \\ 15x - 2y + z &= 4 \end{aligned}$$

To put this system into echelon form we need to first get rid of the x terms in the second and third equations. We will replace the second equation by subtracting four times the first equation, and we'll replace the third equation by subtracting five times the first equation.

The second equation then becomes,

$$\begin{aligned} & 12x - 4y + 2z - 4(3x + 2y - z) = 1 - 4 \cdot 3 \\ \implies & -12y + 6z = -11 \end{aligned}$$

While the thid equation becomes

$$\begin{aligned} & 15x - 2y + z - 5(3x + 2y - z) = 4 - 5 \cdot 3 \\ \implies & -12y + 6z = -11 \end{aligned}$$

Our system thus far is

$$\begin{aligned} 3x + 2y - z &= 3 \\ -12y + 6z &= -11 \\ -12y + 6z &= -11 \end{aligned}$$

To put the system in echelon form we need to get rid of the y term in the third equation, but doing this will of course get rid of all of the terms in the third equation. Thus the echelon form of the system is

$$\begin{aligned} 3x + 2y - z &= 3 \\ -12y + 6z &= -11 \end{aligned}$$

(If you want, there's an equation $0x + 0y + 0z = 0$ at the bottom of this.)

Let's notice that if we try to kill of y in the second equation by adding six times the first equation, we would also kill of z and our

system would become

$$\begin{aligned} 3x + 2y - z &= 3 \\ 18x &= 7 \end{aligned}$$

So $x = 7/18$: no matter what y and z happen to be, x must be $7/18$. Geometrically, this means all (x, y, z) solutions to our system must live in the plane $x = 7/18$.

We could rewrite the first equation as

$$\frac{7}{6} + 2y - z = 3$$

and solving for y we would have

$$y = \frac{z}{2} + \frac{11}{12}.$$

This means that each solution to our system has the form

$$\left(\frac{7}{18}, \frac{z}{2} + \frac{11}{12}, z\right)$$

and z can take on any value: our set of solutions is a line in the plane $x = \frac{7}{18}$:

$$\left\{ \left(\frac{7}{18}, \frac{z}{2} + \frac{11}{12}, z\right) \middle| z \in \mathbb{R} \right\}.$$

In the previous example, z is called a ***free variable*** because it can take on any value we wish. When we express the solution set in terms of free variables, such as above, we have a ***parametrization*** of the solution set.

Example 9.14.
Solve the following system of equations.

$$\begin{aligned} x + y - 3z &= 4 \\ x - 2y + z &= 3 \\ -3x + y + 4z &= 0 \end{aligned}$$

Let's first try to kill off the x in the second equation by subtracting the first equation from it (i.e., we we are applying Theorem 9.2 by adding -1 times the first equation to the second equation). The second equation is then replaced with

$$\begin{aligned} x - 2y + z - (x + y - 3z) &= 3 - 4 \\ \implies -3y + 4z &= -1 \end{aligned}$$

Now our system looks like

$$\begin{aligned} x + y - 3z &= 4 \\ -3y + 4z &= -1 \\ -3x + y + 4z &= 0 \end{aligned}$$

We still need to kill of the $-3x$ in the third equation, so let's add three times the first equation to it. The third equation then becomes

$$\begin{aligned} -3x + y + 4z + 3(x + y - 3z) &= 0 + 3 \cdot 4 \\ \implies 4y - 5z &= 12 \end{aligned}$$

So far we have replaced our original system with the following equivalent one:

$$\begin{aligned} x + y - 3z &= 4 \\ -3y + 4z &= -1 \\ 4y - 5z &= 12 \end{aligned}$$

We need to perform one last step to put the system in echelon form. Let's get rid of the $4y$ in the third equation by adding $\frac{4}{3}$ the second equation:

$$\begin{aligned} 4y - 5z + \frac{4}{3}(-3y + 4z) &= 12 + \frac{4}{3}(-1) \\ \implies -5z + 16/3z &= 12 - \frac{4}{3} \\ \implies \frac{1}{3}z &= \frac{32}{3} \end{aligned}$$

We now have a system in echelon form that's equivalent to our original system:

$$\begin{aligned} x + y - 3z &= 4 \\ -3y + 4z &= -1 \\ \frac{1}{3}z &= \frac{32}{3} \end{aligned}$$

The system is already in echelon form, but let's kill off that $\frac{1}{3}$ in the third equation by using Theorem 9.3 to multiply the third equation by three:

$$\begin{aligned} x + y - 3z &= 4 \\ -3y + 4z &= -1 \\ z &= 32 \end{aligned}$$

This is a system we can easily solve by back-substitution. Plugging in $z = 32$ into the second equation gives us

$$-3y + 128 = -1$$

which tells us $y = 43$. Plugging $z = 32$ and $y = 43$ into the first equation gives us

$$x + 43 - 96 = 4$$

and so $x = 57$.

Since this system is equivalent to our original system, the solution to our original system is

$$\begin{aligned} x &= 57 \\ y &= 43 \\ z &= 32 \end{aligned}$$

9.2 Matrices

A ***matrix*** is a rectangular table of numbers, usually written inbetween parentheses or square brackets. The ***size*** of a matrix is a pair of numbers telling us how many rows and columns the matrix has: if the matrix has m

rows and n columns, we say the size of the matrix is $m \times n$, pronounced m *by* n.

Example 9.15.
The following matrices have respective sizes 2×4 and 5×3.

$$\begin{pmatrix} 4 & -7 & 0 & \pi \\ 1.5 & 2 & 1 & 1 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 0 \\ -1 & 2 & 14 \\ 3 & 7 & 7 \\ -2 & -2 & -2 \\ 5 & 4 & 2 \end{pmatrix}$$

Matrices associated to a linear system

One use of matrices is in encoding a system of linear equations. If we have a system of linear equations, all we really need to know about the system is what the coefficients are, and what the values on the right-hand side of the equations are: what we call the variables (x and y versus x_1 and x_2, for instance) doesn't really matter. If we record all of the coefficients of a system with m equations and n variables as an $m \times n$ matrix, we have the ***coefficient matrix*** of the system.

Example 9.16.
Consider the following system of three equations in four unknowns:

$$\begin{aligned} 6v + 3x - 2y + z &= 4 \\ 4v - x + y - 2z &= 3 \\ 2x + 4y + 4z &= 9 \\ v + z &= -1 \end{aligned}$$

The corresponding coefficient matrix of this system is

$$\begin{pmatrix} 6 & 3 & -2 & 1 \\ 4 & -1 & 1 & -2 \\ 0 & 2 & 4 & 4 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

Notice that we pick up zeroes in the coefficient matrix when a variable is missing. The reason for this is that if a variable is missing, such as the missing v in the third equation of the system in Example 9.16, we can write it as 0 times that variable. The system in Example 9.16, for instance, may be written as

$$\begin{aligned} 6v + 3x - 2y + z &= 4 \\ 4v - x + y - 2z &= 3 \\ 0v + 2x + 4y + 4z &= 9 \\ v + 0x + 0y + z &= -1 \end{aligned}$$

Our goal will be to solve systems of linear equations by manipulating matrices. In doing so we of course also want to keep track of the values on the right-hand side of the equations. We will do this by just adding an extra column onto our coefficient matrix containing the right-hand sides. This gives us the ***augmented coefficient matrix*** of the system.

Example 9.17.
The augmented coefficient matrix of Example 9.16 is

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

Remark.
Some people like to write a vertical bar in the augmented coefficient matrix to separate the coefficients of the left-hand sides of the equa-

tions from the values on the right-hand sides of the equations, such as

$$\left(\begin{array}{cccc|c} 6 & 3 & -2 & 1 & 4 \\ 4 & -1 & 1 & -2 & 3 \\ 0 & 2 & 4 & 4 & 9 \\ 1 & 0 & 0 & 1 & -1 \end{array}\right)$$

The addition of this vertical line is purely cosmetic. Our textbook does not use the bar, but it is fairly common. You are free to use the vertical bar if you'd like.

Elementary row operations

Recall from before that we had three different procedures that we could perform to a system of linear equations to obtain an equivalent system:

1. Swap two rows.
2. Replace one row with the sum of the original row and a multiple of another row.
3. Multiply every term in a row by the same non-zero constant.

We can perform these same three operations on the augmented coefficient matrix of a linear system and we obtain the augmented coefficient matrix of the equivalent linear system. Usually when we're performing these three operations to a matrix, we refer to them as the ***elementary row operations***. The process of using elementary row operations to turn one matrix into another is called ***row reduction***.

By repeatedly applying the elementary row operations to the augmented coefficient of a matrix, we can replace the turn our system of linear equations into an equivalent one which we can easily solve. In particular, we have a system that is easy to solve when we have put our matrix into *echelon form*. Before define the echelon form of a matrix, let's introduce one preliminary definition that will make the language a little easier.

The ***leading entry*** of a row in a matrix is the left-most non-zero element in that row.

We say that a matrix is in ***echelon form*** if the following three conditions are satisfied:

1. If the matrix has any rows consisting of only zeros, they occur at the bottom of the matrix.

2. The leading entry on each row in the matrix is to the right of the leading entry of the above rows.

3. All entries in the same column and below the leading entry in a row are zero.

Let's first see some examples of some things that are, and some things that are not, in echelon form.

Example 9.18.
The following matrices are all in echelon form:

$$\begin{pmatrix} 3 & -1 & 2 & 0 \\ 0 & 7 & 9 & -1 \\ 0 & 0 & 2 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 6 & 2 & 1 & 4 \\ 0 & 1 & 1 & 3 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 4 & 2 & -1 & 3 & 2 \\ 0 & 0 & 0 & 3 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 6 & 7 & -4 & 3 & 2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Example 9.19.

None of the following matrices are in echelon form:

$$\begin{pmatrix} 0 & 2 & 4 & -1 \\ 1 & 1 & 3 & 2 \\ 0 & 0 & 2 & 4 \end{pmatrix}$$

$$\begin{pmatrix} 7 & 2 & -1 & 2 \\ 5 & 6 & 3 & 3 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 3 & 7 & 1 & 2 \\ 0 & 0 & 0 & 4 & 5 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

We can always put a matrix into echelon form by applying elementary row operations. One way to solve a system of linear equations, then, is to write out the augmented coefficient matrix, put it into echelon form, and then use ***back substitution*** (solving for the variables one at a time).

Example 9.20.
Put the following matrix into echelon form:

$$\begin{pmatrix} 1 & 2 & 4 & 5 \\ 2 & 4 & 5 & 4 \\ 4 & 5 & 4 & 2 \end{pmatrix}$$

The first thing we need to do is zero-out the entries below 1 in the first column. Let's subtract twice the first row from the second to obtain:

$$\begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & 0 & -3 & -6 \\ 4 & 5 & 4 & 2 \end{pmatrix}$$

Now subtract four times the first row from the third:

$$\begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & 0 & -3 & -6 \\ 0 & -3 & -12 & -18 \end{pmatrix}$$

Now let's swap the second and third rows,

$$\begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & -3 & -12 & -18 \\ 0 & 0 & -3 & -6 \end{pmatrix}$$

Writing out what we're doing in words is always okay to do, but it can get tedious sometimes, so let's introduce some notation to save ourselves some writing. When we perform a row operation on a matrix A to obtain a matrix A', let's draw an arrow from A to A' and label the arrow to describe which operation we are performing.

If we obtain A' from A by swapping row i and row j, we will write

$$A \xrightarrow{R_i \leftrightarrow R_j} A'$$

For example,

$$\begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & 0 & -3 & -2 \\ 0 & -3 & -4 & -18 \end{pmatrix} \xrightarrow{R_2 \leftrightarrow R_3} \begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & -3 & -4 & -18 \\ 0 & 0 & -3 & -2 \end{pmatrix}$$

If we add c times row j to row i, we will write

$$A \xrightarrow{R_i + cR_j \to R_i} A'$$

E.g.,

$$\begin{pmatrix} 1 & 2 & 4 & 5 \\ 2 & 4 & 5 & 4 \\ 4 & 5 & 4 & 2 \end{pmatrix} \xrightarrow{R_2 - 2R_1} \begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & 0 & -3 & -2 \\ 4 & 5 & 4 & 2 \end{pmatrix}$$

If we multiply each element in row i by c, we will write

$$A \xrightarrow{cR_i \to R_i} A'.$$

For example,

$$\begin{pmatrix} 4 & 7 & 2 \\ 0 & 4 & 3 \\ -1 & 2 & 2 \\ 5 & -2 & 1 \end{pmatrix} \xrightarrow{-2R_3 \to R_3} \begin{pmatrix} 4 & 7 & 2 \\ 0 & 4 & 3 \\ 2 & -4 & -4 \\ 5 & -2 & 1 \end{pmatrix}$$

Notice that the echelon form of a matrix is not unique: if you give the same matrix to two people and ask them to put the matrix in echelon form, each person may give you back a different (but correct!) matrix in echelon form. The matrix in Example 9.20, for instance, could be put into echelon form in the following way:

Example 9.21.
Put the following matrix into echelon form:

$$\begin{pmatrix} 1 & 2 & 4 & 5 \\ 2 & 4 & 5 & 4 \\ 4 & 5 & 4 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 4 & 5 \\ 2 & 4 & 5 & 4 \\ 4 & 5 & 4 & 2 \end{pmatrix} \xrightarrow{R_3 - 2R_2 \to R_3} \begin{pmatrix} 1 & 2 & 4 & 5 \\ 2 & 4 & 5 & 4 \\ 0 & -3 & -6 & -6 \end{pmatrix}$$

$$\xrightarrow{R_2 - 2R_1 \to R_2} \begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & 0 & -3 & -6 \\ 0 & -3 & -6 & -6 \end{pmatrix}$$

$$\xrightarrow{R_2 \leftrightarrow R_3} \begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & -3 & -6 & -6 \\ 0 & 0 & -3 & -6 \end{pmatrix}$$

In Example 9.20 and Example 9.21 we started with the same matrix, but produced two different matrices in echelon form because we performed two different sequences of elementary row operations.

Remark.
The above is something to consider if you compare answers to homework problems with another student. If you were both trying to put a matrix into echelon form, you may both come up with different, correct answers!

It would be nice if there was a way to modify echelon form so that we would always calculate the same matrix. This can be done if we modify the conditions of echelon form slightly to get *row-reduced echelon form*.

We say a matrix A is in ***row-reduced echelon form*** (abbreviated ***RREF***) if the following four conditions are satsified:

1. If the matrix has any rows consisting of only zeros, they occur at the bottom of the matrix.
2. The leading entry on each row in the matrix is to the right of the leading entry of the above rows.
3. All entries in the same column above and below the leading entry in a row are zero.
4. Every leading entry is a one.

So RREF is very similar to echelon form, but we'll make sure that leading entries are always equal to one, and that everything directly *above and below* a leading entry is zero.

Remark.
Some people simply say *reduced echelon form* where we have said *row-reduced echelon form*, but this is the same thing.

Let's take our two matrices in echelon form from Example 9.20 and Example 9.21 and convert them to RREF.

Example 9.22.
Convert the following matrix to RREF:

$$\begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & -3 & -12 & -18 \\ 0 & 0 & -3 & -6 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & -3 & -12 & -18 \\ 0 & 0 & -3 & -6 \end{pmatrix} \xrightarrow{\frac{-1}{3}R_2 \to R_2} \begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & 1 & 4 & 6 \\ 0 & 0 & -3 & -6 \end{pmatrix}$$
$$\xrightarrow{R_1 - 2R_2 \to R_1} \begin{pmatrix} 1 & 0 & -4 & -7 \\ 0 & 1 & 4 & 6 \\ 0 & 0 & -3 & -6 \end{pmatrix}$$
$$\xrightarrow{-\frac{1}{3}R_3 \to R_3} \begin{pmatrix} 1 & 0 & -4 & -7 \\ 0 & 1 & 4 & 6 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$
$$\xrightarrow{R_2 - 4R_3 \to R_2} \begin{pmatrix} 1 & 0 & -4 & -7 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$
$$\xrightarrow{R_1 + 4R_3 \to R_1} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

Example 9.23.
Convert the following matrix to RREF:

$$\begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & -3 & -6 & -6 \\ 0 & 0 & -3 & -6 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & -3 & -6 & -6 \\ 0 & 0 & -3 & -6 \end{pmatrix} \xrightarrow{-\frac{1}{3}R_2 \to R_2} \begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & 1 & 2 & 2 \\ 0 & 0 & -3 & -6 \end{pmatrix}$$

$$\xrightarrow{-\frac{1}{3}R_3 \to R_3} \begin{pmatrix} 1 & 2 & 4 & 5 \\ 0 & 1 & 2 & 2 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

$$\xrightarrow{R_1 - 2R_2 \to R_1} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 2 & 2 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

$$\xrightarrow{R_2 - 2R_3 \to R_2} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & 2 \end{pmatrix}$$

The main thing about main reason we prefer RREF over echelon form is that every matrix is equivalent to exactly one matrix in RREF.

Theorem 9.4.
Performing elementary row operations to put a matrix in row-reduced echelon form produces exactly one matrix.

The proof of this fact will be easier to explain after we talk about linear independence, so we will postpone the proof of this theorem for now.

Example 9.24.
Solve the following system of linear equations by putting the augmented coefficient matrix in RREF.

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

We have seen that the augmented coefficient matrix of this system can

be put into the following matrix in RREF:

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

which is the augmented coefficient matrix of the equivalent system

$$\begin{aligned} x &= 1 \\ y &= -2 \\ z &= 2 \end{aligned}$$

and so the only solution to our system of equations is $(x, y, z) = (1, -2, 2)$.

We are now in a position to describe an algorithm for putting a matrix in RREF, but before presenting the algorithm we introduce one piece of terminology.

If a matrix is in RREF, then the location of the leading entries are called the ***pivot positions***; the columns containing pivot positions are called ***pivot columns***. More generally, if A can be reduced to a matrix A' in RREF, then the pivot positions and columns of A are defined to be the pivot positions and columns of A'.

By performing elementary row operations, we can always put a non-zero value in a pivot position. When we do this, the non-zero value we place in the pivot position is called a ***pivot***.

The algorithm for putting a matrix into RREF is as follows:

1. Starting from the top row of the matrix, the left-most position which *is not* in a column of all zeros will be a pivot position.

2. Swap rows if necessary so that the entry in the pivot position is non-zero.

3. Divide the row containing this pivot position by the new non-zero pivot value.

4. Add multiples of the row to the other rows so that we have only zeros above and below the pivot in the pivot column.

5. Repeat the process, but use the submatrix obtained by deleting everything to the left of and above the pivot position (including the row

and column containing the pivot position) to determine the next pivot position.

Example 9.25.
Put the following matrix in RREF:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 3 & 7 \\ 0 & 0 & 2 & 0 & 1 & 4 \\ 0 & 0 & 4 & 0 & 2 & 1 \\ 0 & 5 & 3 & 0 & 4 & 2 \end{pmatrix}$$

We need to work with our pivots one at a time, from the top left-most pivot down to the bottom right-most pivot. We will color code which pivot we are considering as follows: The pivot we are currently considering will be yellow, and the pivots we have finished working with will be pink.

We start with the top row. First finding the left-most entry which is not in an all-zero column.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 3 & 7 \\ 0 & 0 & 2 & 0 & 1 & 4 \\ 0 & 0 & 4 & 0 & 2 & 1 \\ 0 & 5 & 3 & 0 & 4 & 2 \end{pmatrix}$$

Now swap the top and bottom columns to put a 5 in the pivot position:

$$\begin{pmatrix} 0 & 5 & 3 & 0 & 4 & 2 \\ 0 & 0 & 2 & 0 & 1 & 4 \\ 0 & 0 & 4 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 3 & 7 \end{pmatrix}$$

Divide the first row by 5 to put a 1 into the pivot position:

$$\begin{pmatrix} 0 & 1 & \frac{3}{5} & 0 & \frac{4}{5} & \frac{2}{5} \\ 0 & 0 & 2 & 0 & 1 & 4 \\ 0 & 0 & 4 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 3 & 7 \end{pmatrix}$$

The matrix already has zeros below the pivot, so we move onto the pivot.

First find the left-most entry in the second row which is to the right of the previous pivot and *not* in a column of all zeros.

$$\begin{pmatrix} 0 & 1 & \frac{3}{5} & 0 & \frac{4}{5} & \frac{2}{5} \\ 0 & 0 & 2 & 0 & 1 & 4 \\ 0 & 0 & 4 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 3 & 7 \end{pmatrix}$$

Now divide the second row by 2 to put a 1 into the next pivot position:

$$\begin{pmatrix} 0 & 1 & \frac{3}{5} & 0 & \frac{4}{5} & \frac{2}{5} \\ 0 & 0 & 1 & 0 & \frac{1}{2} & 2 \\ 0 & 0 & 4 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 3 & 7 \end{pmatrix}$$

We need to zero out the non-zero entries in the pivot column above and below our pivot position. First we subtract $3/5$ the second row from the first:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \frac{1}{2} & -\frac{4}{5} \\ 0 & 0 & 1 & 0 & \frac{1}{2} & 2 \\ 0 & 0 & 4 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 3 & 7 \end{pmatrix}$$

Now subtract four times the second row from the third row:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \frac{1}{2} & -\frac{4}{5} \\ 0 & 0 & 1 & 0 & \frac{1}{2} & 2 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & 3 & 7 \end{pmatrix}$$

We move on to the third pivot. Since we have a zero in the pivot position, we need to swap the third and fourth rows to put the 3 into the pivot position:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \frac{1}{2} & -\frac{4}{5} \\ 0 & 0 & 1 & 0 & \frac{1}{2} & 2 \\ 0 & 0 & 0 & 0 & 3 & 7 \\ 0 & 0 & 0 & 0 & 0 & -7 \end{pmatrix}$$

Divide the third row by 3 to put a 1 into the pivot position:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \frac{1}{2} & -\frac{4}{5} \\ 0 & 0 & 1 & 0 & \frac{1}{2} & 2 \\ 0 & 0 & 0 & 0 & 1 & \frac{7}{3} \\ 0 & 0 & 0 & 0 & 0 & -7 \end{pmatrix}$$

Now zero out the entries above the third pivot. First subtract one-half the third row from the first:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & -\frac{59}{30} \\ 0 & 0 & 1 & 0 & \frac{1}{2} & 2 \\ 0 & 0 & 0 & 0 & 1 & \frac{7}{3} \\ 0 & 0 & 0 & 0 & 0 & -7 \end{pmatrix}$$

Finally subtract one-half the third row from the second:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & -\frac{59}{30} \\ 0 & 0 & 1 & 0 & 0 & \frac{5}{6} \\ 0 & 0 & 0 & 0 & 1 & \frac{7}{3} \\ 0 & 0 & 0 & 0 & 0 & -7 \end{pmatrix}$$

For the very last pivot we will simply divide by -7 to make the pivot a 1, and then zero out the entries above the pivot. Finally subtract one-half the third row from the second:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & -\frac{59}{30} \\ 0 & 0 & 1 & 0 & 0 & \frac{5}{6} \\ 0 & 0 & 0 & 0 & 1 & \frac{7}{3} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

We now have the RREF of our original matrix:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Once we put an augmented coefficient matrix into RREF, it is then *very* easy to solve the corresponding system of linear equations.

Example 9.26.
Determine all solutions to the following system of equations:

$$\begin{aligned} 3x_5 &= 7 \\ 2x_3 + x_5 &= 4 \\ 4x_3 + 2x_5 &= 1 \\ 5x_2 + 3x_3 + 4x_5 &= 2 \end{aligned}$$

The augmented coefficient matrix of this system is precisely our matrix from before, so the RREF of that matrix tells us the following system is equivalent:

$$\begin{aligned} x_2 &= 0 \\ x_3 &= 0 \\ x_5 &= 0 \\ 0 &= 1 \end{aligned}$$

Because of this last equation, the system has no solutions!

Example 9.27.
Solve the following system of linear equations.

$$\begin{aligned} 3x + 7y + 17z &= 21 \\ 2y + 4z &= 6 \\ 4x + 10y + 24z &= 30 \end{aligned}$$

To solve the system, let's put the augmented coefficient matrix, which is

$$\left(\begin{array}{ccc|c} 3 & 7 & 17 & 21 \\ 0 & 2 & 4 & 6 \\ 4 & 10 & 24 & 30 \end{array}\right),$$

into RREF:

$$\begin{pmatrix} 3 & 7 & 17 & 21 \\ 0 & 2 & 4 & 6 \\ 4 & 10 & 24 & 30 \end{pmatrix} \xrightarrow{\frac{1}{3}R_1 \to R_1} \begin{pmatrix} 1 & \frac{7}{3} & \frac{17}{3} & 7 \\ 0 & 2 & 4 & 6 \\ 4 & 10 & 24 & 30 \end{pmatrix}$$
$$\xrightarrow{R_3 - 4R_1 \to R_3} \begin{pmatrix} 1 & \frac{7}{3} & \frac{17}{3} & 7 \\ 0 & 2 & 4 & 6 \\ 0 & \frac{2}{3} & \frac{4}{3} & 2 \end{pmatrix}$$
$$\xrightarrow{\frac{1}{2}R_2 \to R_2} \begin{pmatrix} 1 & \frac{7}{3} & \frac{17}{3} & 7 \\ 0 & 1 & 2 & 3 \\ 0 & \frac{2}{3} & \frac{4}{3} & 2 \end{pmatrix}$$
$$\xrightarrow{R_1 - \frac{7}{3}R_2 \to R_1} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 2 & 3 \\ 0 & \frac{2}{3} & \frac{4}{3} & 2 \end{pmatrix}$$
$$\xrightarrow{R_3 - \frac{2}{3}R_2 \to R_3} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The last matrix above is in RREF. This tells us that our original system of equations is equivalent to

$$x + z = 0$$
$$y + 2z = 3$$

Notice that we could solve both of these equations to express x and y in terms of z:

$$x = -z$$
$$y = 3 - 2z$$

That is, if we know what z is, then we instantly know what x and y must be. However, z could be anything! There are infinitely-many different choices for z, and each one gives us a different solution to our system. Thus there are infinitely-many solutions to the system, all of which have the form

$$(-z, 3 - 2z, z)$$

where z can be any real number.

Notice that in Example 9.27, not only did we determine that there were infinitely-many solutions to the system, but we said explicitly what the solutions had to look like. In such a situation, where we describe all of the solutions in terms of some variable, we say that we have given a ***parametrization*** of the solution set and call the variable that is allowed to change a ***free variable***.

Example 9.28.
Solve the following system of linear equations:

$$\begin{aligned} 5x_1 + 3x_2 - 8x_3 - 2x_4 &= 5 \\ 2x_1 + 4x_2 - 6x_3 + 2x_4 &= 2 \\ 2x_1 + 1x_2 - 3x_3 - x_4 &= 2 \\ 4x_1 + 3x_2 - 7x_3 - x_4 &= 4 \end{aligned}$$

We again put the augmented coefficient matrix into RREF to get an

equivalent, easier-to-solve system:

$$\begin{pmatrix} 5 & 3 & -8 & -2 & 5 \\ 2 & 4 & -6 & 2 & 2 \\ 2 & 1 & -3 & -1 & 2 \\ 4 & 3 & -7 & -1 & 4 \end{pmatrix} \xrightarrow{\frac{1}{5}R_1 \to R_1} \begin{pmatrix} 1 & 3/5 & -8/5 & -2/5 & 1 \\ 2 & 4 & -6 & 2 & 2 \\ 2 & 1 & -3 & -1 & 2 \\ 4 & 3 & -7 & -1 & 4 \end{pmatrix}$$

$$\xrightarrow{R_2 - 2R_1 \to R_2} \begin{pmatrix} 1 & 3/5 & -8/5 & -2/5 & 1 \\ 0 & 14/5 & -14/5 & 14/5 & 0 \\ 2 & 1 & -3 & -1 & 2 \\ 4 & 3 & -7 & -1 & 4 \end{pmatrix}$$

$$\xrightarrow{R_3 - 2R_1 \to R_3} \begin{pmatrix} 1 & 3/5 & -8/5 & -2/5 & 1 \\ 0 & 14/5 & -14/5 & 14/5 & 0 \\ 0 & -1/5 & 1/5 & -1/5 & 0 \\ 4 & 3 & -7 & -1 & 4 \end{pmatrix}$$

$$\xrightarrow{R_4 - 4R_1 \to R_4} \begin{pmatrix} 1 & 3/5 & -8/5 & -2/5 & 1 \\ 0 & 14/5 & -14/5 & 14/5 & 0 \\ 0 & -1/5 & 1/5 & -1/5 & 0 \\ 0 & 3/5 & -3/5 & 3/5 & 0 \end{pmatrix}$$

$$\xrightarrow{\frac{5}{14}R_2 \to R_2} \begin{pmatrix} 1 & 3/5 & -8/5 & -2/5 & 1 \\ 0 & 1 & -1 & 1 & 0 \\ 0 & -1/5 & 1/5 & -1/5 & 0 \\ 0 & 3/5 & -3/5 & 3/5 & 0 \end{pmatrix}$$

$$\xrightarrow{R_1 - \frac{3}{5}R_2 \to R_1} \begin{pmatrix} 1 & 0 & -1 & -1 & 1 \\ 0 & 1 & -1 & 1 & 0 \\ 0 & -1/5 & 1/5 & -1/5 & 0 \\ 0 & 3/5 & -3/5 & 3/5 & 0 \end{pmatrix}$$

$$\xrightarrow{R_3 + \frac{1}{5}R_2 \to R_3} \begin{pmatrix} 1 & 0 & -1 & -1 & 1 \\ 0 & 1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 3/5 & -3/5 & 3/5 & 0 \end{pmatrix}$$

$$\xrightarrow{R_4 - \frac{3}{5}R_2 \to R_4} \begin{pmatrix} 1 & 0 & -1 & -1 & 1 \\ 0 & 1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

We now know that our original system of equations is equivalent to

the following system:

$$x_1 - x_3 - x_4 = 1$$
$$x_2 - x_3 + x_4 = 0$$

Solving the first two equations for x_1 and x_2, respectively, tells us that

$$x_1 = 1 + x_3 + x_4$$
$$x_2 = x_3 - x_4$$

Here there are no restrictions on either x_3 or x_4, each of these can be any real number, and so we have two free variables. A parametrization of the solution set is

$$(1 + x_3 + x_4, x_3 - x_4, x_3, x_4).$$

Consistency and Inconsistency in Terms of RREF

After we put the augmented coefficient matrix of a system into RREF, we can quickly determine whether the system is consistent or not, and if it is consistent whether it has a unique solution or infinitely-many solutions. If the matrix in RREF has a row of the form

$$\left(\begin{array}{cccccc|c} 0 & 0 & 0 & \cdots & 0 & 0 & b \end{array}\right)$$

where $b \neq 0$, then the system is inconsistent. The existence of such a row tells us that the system is equivalent to a system that has an equation of the form

$$0x_1 + 0x_2 + 0x_3 + \cdots + 0x_{n-1} + 0x_n = b$$

It doesn't matter what the other equations (or rows in the matrix) look like: there is no way to solve this one, so the system has no solutions.

If the matrix in RREF has the form

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & b_1 \\ 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & b_2 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & b_3 \\ & & & & \ddots & & & & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & b_{n-1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & b_n \end{pmatrix}$$

then the system has a single unique solution, $(b_1, b_2, ..., b_n)$.

Notice that if we took the previous matrix in RREF and added rows of zeros to the bottom, then this doesn't change the solutions. When this happens some of the equations in the original system were "redundant."

Example 9.29.
Consider the following system:

$$\begin{aligned} x + y &= 1 \\ x - y &= 2 \\ 3x + y &= 4 \end{aligned}$$

The augmented coefficient matrix of the system is

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -1 & 2 \\ 3 & 1 & 4 \end{pmatrix}$$

The RREF of this matrix is

$$\begin{pmatrix} 1 & 0 & 3/2 \\ 0 & 1 & -1/2 \\ 0 & 0 & 0 \end{pmatrix}$$

The system thus has a unique solution of $(3/2, -1/2)$. The third equation of the original system doesn't give us any additional information about the system because it is twice the first equation plus the second: once you know what the solution to the two equations are, you instantly know what the solution to the third equation is as well, so there's no real need to have the third equation.

If the matrix in RREF has rows of zeros, but there is not a unique solution to the system, then the system has infinitely-many solutions. The number of free variables is determined by the number of rows of zeros. Each non-zero row gives us an equation relating the variables, and each zero row (up to the number of variables) gives us a free variable. Notice that the number of variables is one less than the number of columns in the augmented coefficient matrix: each variable gives us one column, plus we have one more column for the right-hand sides. This means that the number

of free variables in our solution to a system is determined by the number of non-pivot columns (ignoring the right-most column corresponding to the right-hand sides of equations in our system).

In Example 9.27 we had one non-pivot column (ignoring the column corresponding to the RHS) and so one free variable; in Example 9.28 we had two non-pivot columns, so two free variables.

Example 9.30.
Determine the set of solutions to the following system by putting the augmented coefficient matrix into RREF. How many free variables are there?

$$\begin{aligned} 5x_1 + 3x_2 - 8x_3 - 2x_4 &= 5 \\ 2x_1 + 4x_2 - 6x_3 + 2x_4 &= 2 \\ 2x_1 + 1x_2 - 3x_3 - x_4 &= 2 \\ 4x_1 + 3x_2 - 7x_3 - x_4 &= 4 \\ -6x_1 - 3x_2 + 9x_3 + 3x_4 &= -6 \end{aligned}$$

In RREF this matrix becomes

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

Since the system is consistent and there are two non-pivot columns, any parametrization of the set of solutions to the system must contain two free variables.

9.3 Vectors

Vectors appear in many different areas of mathematics and sciences, and you have seen vectors before if you've taken a course in physics of multivariable calculus. In these classes vectors are usually described as quantities that have both a magnitude and a direction. We will see later in the course that many different types of quantities can be thought of as vectors, even

things that don't obviously have a magnitude or direction (for exmaple, polynomials can be thought of as vectors). In this section we start with the basics though, first defining vectors as "arrows" in $\mathbb{R}^2$ and $\mathbb{R}^3$, and then generalizing vectors to $\mathbb{R}^n$ for any dimension n. We also see that some of the questions we are naturally lead to about vectors are really questions about systems of linear equations in disguise.

Vectors in $\mathbb{R}^2$ and $\mathbb{R}^3$

We will see that vectors can be defined in any number of dimensions, but to get started we will consider vectors in two and three dimensions which will be familiar to anyone that has taken an introductory course in physics or multivariable calculus.

A vector is often described as a quantity which has both a direction and a magnitude. One physical example is force: every force has a direction (where the force is pushing from or pulling towards) and a magnitude (how strong the force is). Consider the gravitational force between the Earth and an object near the surface of the Earth. The object is being pulled "down" towards the Earth (this is the direction) and that object has some weight (this is the magnitude). If the object has more mass, then it will have a greater weight and gravity is pulling more strongly on the object. (Consider trying to hold a one kilogram object over your head versus a fifteen kilogram object. Gravity is pulling harder on the 15 kg object which is why that object feels heavier and harder to hold up.)

In two or three dimensions we represent these "direction together with a magnitude" quantities as arrows where the direction of the arrow is the direction of the vector, and the length of the vector represents the magnitude. We will focus on the two-dimensional case at first simply because it's slightly easier to draw pictures representing these quantities.

Two-dimensional vectors

A ***vector*** in $\mathbb{R}^2$ is simply an arrow in the plane: a line segment from some point (x_0, y_0) to another point (x_1, y_1) with an arrowhead at (x_1, y_1), as in Figure 9.3

One thing about vectors that may seem a little strange is that we *only* care about the direction the vector points in and its magnitude, and we *do not* care about where that vector is drawn in space. That is, given any vector we can move it around the plane or 3-space as much as we'd like and provided we don't stretch the vector (which would change its magnitude) or

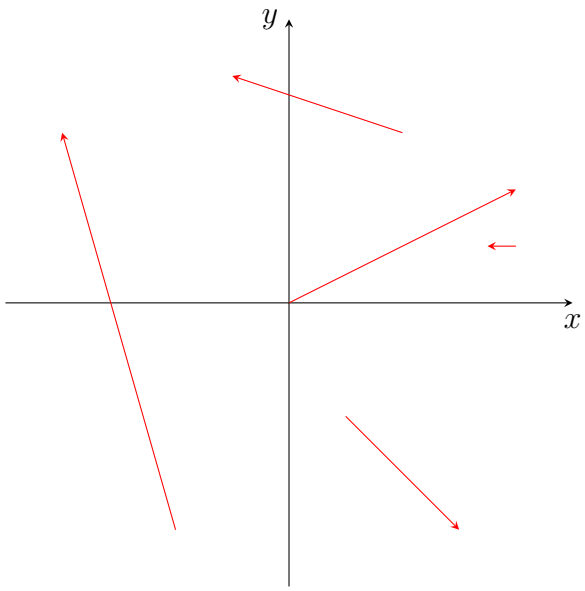


Figure 9.3: Vectors in $\mathbb{R}^2$.

rotate it (which would change its direction) we still have the same vector. See Figure 9.4.

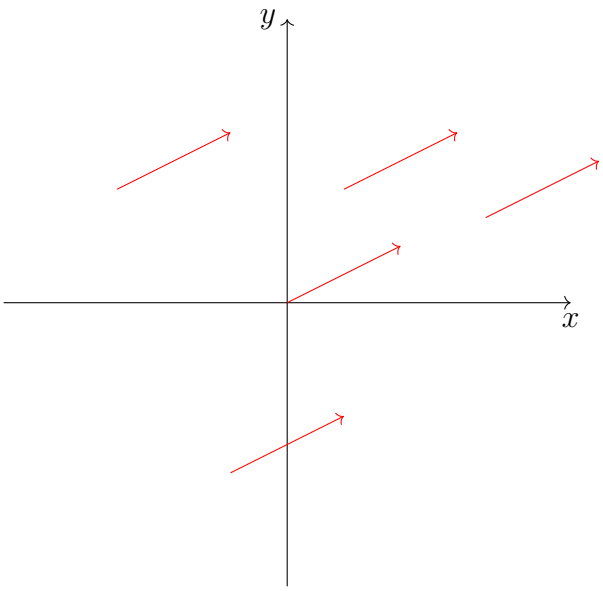


Figure 9.4: Each arrow in this picture represents the same vector since the direction and magnitude (length) is the same for each arrow.

We usually give vectors a name, just like any other mathematical quan-

tity, to make it easier to describe. Instead of just calling a vector v, however, it is common to write the letter in bold, $\mathbf{v}$, or to write an arrow over the letter, $\vec{v}$, to denote that this quantity is a vector. Almost everyone writes the arrow when they are writing vectors by hand on paper or a blackboard, and only some books (including our textbook) use the bold letters to denote vectors.

We will sometimes call the beginning point of the arrow represent a vector (the part without an arrowhead) the ***tail*** of the vector, and the other point (the part with an arrowhead) the ***tip*** of the vector.

Given two vectors, $\vec{v}_1$ and $\vec{v}_2$, we can add the two vectors together to get a new vector, $\vec{v}_1 + \vec{v}_2$. There are a few ways we can describe this new vector. One way to describe $\vec{v}_1 + \vec{v}_2$ is to slide $\vec{v}_1$ and $\vec{v}_2$ around so that the tip of $\vec{v}_1$ is at the tail of $\vec{v}_2$. We then draw in an arrow from the tail of $\vec{v}_1$ to the tip of $\vec{v}_2$, and this new vector we've drawn is $\vec{v}_1 + \vec{v}_2$. This is called the ***triangle law*** for vector addition and is illustrated in Figure 9.5.

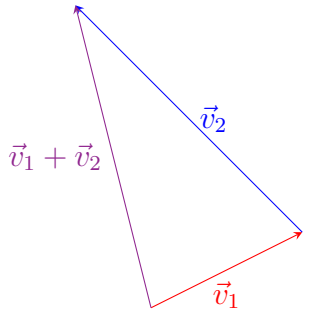


Figure 9.5: We can add two vectors by sliding them *tip-to-tail* and then completing the triangle.

Another, equivalent way to describe vector addition is with the ***parallelogram law***. Here we make copies of $\vec{v}_1$ and $\vec{v}_2$ and slide them around to make a parallelogram, and then draw in the diagonal of this parallelogram connecting the corner that has two tails to the corner that has two tips. This diagonal vector is the sum $\vec{v}_1 + \vec{v}_2$. See Figure 9.6

Remark.
It is important that the diagonal vector drawn in the parallelogram law starts at the corner where two tails meet and goes to the corner where two tips meet. If you connect the corners in the wrong way, you

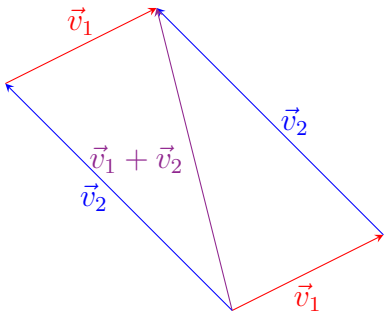


Figure 9.6: Vector addition can also be described by drawing in the diagonal of a parallelogram.

will have the wrong vector.

If you've never seen vector addition before, it may seem a little bit like an odd thing to do, but let's consider one physical application. Suppose that two different forces act on an object: e.g., you and a friend are rearranging furniture in your dorm with one of you pushing a bed in one direction, and the other pushing the bed in another direction. Say you push the bed to the East and your friend pushes the bed to the North. Though you're applying two different forces, the net effect is the same as if you were to push the bed to the North-East. This is exactly what's happening when you add the two forces: the sum of two individual forces acting on an object is the ***net force*** that acts on that object.

There's another operation we can perform on vectors. Given a real number λ and a vector $\vec{v}$, we can define a new vector $\lambda\vec{v}$ by stretching the vector out by a factor of λ. For example, $2\vec{v}$ points in the same direction as $\vec{v}$ but is twice as long; $\frac{1}{3}\vec{v}$ points in the same direction as $\vec{v}$ but is one third as long.

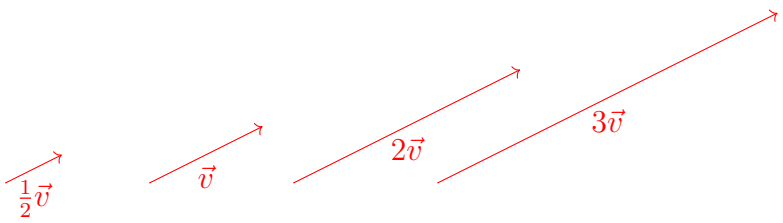


Figure 9.7: Multiplying a vector by a positive number stretches the vector.

If we multiply a vector $\vec{v}$ by a negative λ, then we stretch the vector out by a factor of $|\lambda|$, but make the vector point in the opposite direction.

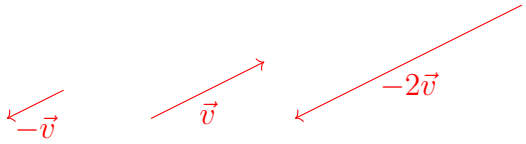


Figure 9.8: Multiplying a vector by a negative number stretches the vector and flips the vector's direction.

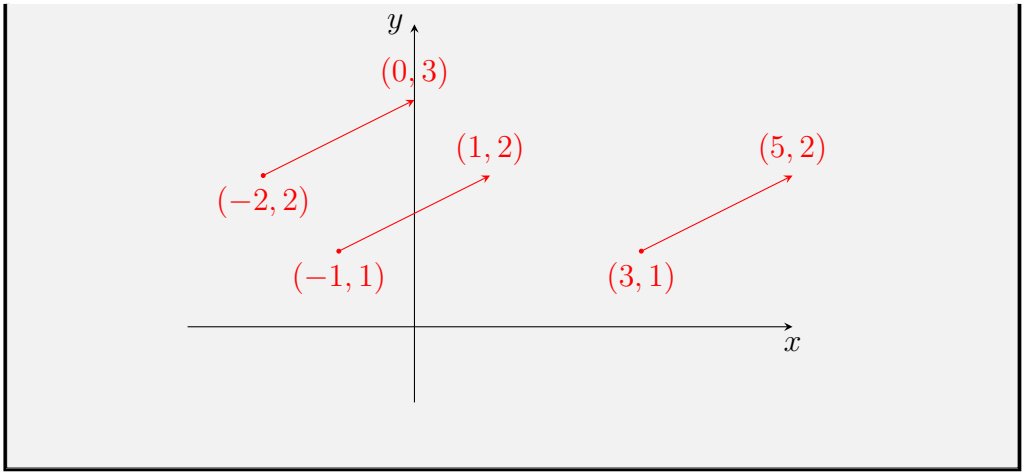
To distinguish "regular" numbers from vectors we sometimes call the numbers ***scalars*** because they scale vectors. The operation of multiplying a scalar and a vector is called ***scalar multiplication***.

Notice that we could describe a vector in the plane by saying how much the x and y coordinates change when you walk from the tail of the vector to the tip. That is, if we've positioned the vector so that its tail is at the point (x_0, y_0) and its tip is at the point (x_1, y_1), then all we really need to know is the change in the x-coordinates, $x_1 - x_0$, and the change in the y-coordinates, $y_1 - y_0$. Regardless of where we've drawn the vector, if we don't stretch it or rotate it, we have the same change in x- and y-coordinates.

Example 9.31.
Suppose $\vec{v}$ is a vector which we've drawn in the plane so that its tail is at the point $(3, 1)$ and its tip is at the point $(5, 2)$. The change in the x-coordinates is $5 - 3 = 2$ and the change in the y-coordinates is $2 - 1 = 1$.

If we were to have moved the vector around so that its tail was instead at $(-2, 2)$, then its tip would be at $(0, 3)$. Again we have the change in the x-coordinates is $0 - (-2) = 2$ and the change in y-coordinates is $3 - 2 = 1$.

If we placed the tail of the vector at $(-1, 1)$, then the tip would be at $(1, 2)$, and once again the change in x- and y-coordinates is 2 and 1, respectively.



So we could describe the vector simply by recording this change in x- and y-coordinates. There are several different ways we could record this, but two common ways would be to make a 2×1 matrix listing the change in x- and y-, or a 1×2 matrix:

$$\begin{pmatrix} x \\ y \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} x & y \end{pmatrix}$$

In the first situation, with the 2×1 matrix we say we have a ***column vector***, and the 1×2 matrix is called a ***row vector***. We will usually, but not always, use column vectors in this class. There's nothing magical about why we choose column vectors instead of row vectors, it's just a choice.

Notice that vector addition and scalar multiplication are very easy to express once we have coordinates like this:

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ y_1 + y_2 \end{pmatrix}$$

$$\lambda \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \lambda x \\ \lambda y \end{pmatrix}$$

Three-dimensional vectors

Vectors in $\mathbb{R}^3$ are completely analogous to vectors in $\mathbb{R}^2$: we they are arrows connecting a point at the tail of the vector to the tip, the direction of the vector is the direction of this arrow, and the length of the arrows is the magnitude. Adding three-dimensional vectors or doing scalar multiplication is exactly the same as adding two-dimensional vectors or doing scalar

multiplication: we can use the parallelogram or triangle laws and stretch a vector out by a given amount. It's slightly harder to draw the pictures on a two-dimensional screen or piece of paper, but everything is defined exactly the same.

Just as we can represent a two-dimensional vector using two pieces of information, telling us the displacement in the x- and y-coordinates between the vector's tail and tip, we can do precisely the same thing in three dimensions and we simply have one more piece of information to deal with: the change in the z-coordinates.

Just as in the two-dimensional vectors can be represented as column or row vectors, so can three-dimensional vectors: A 3×1 matrix is a column vector in three dimensions, and a 1×3 matrix is a row vector in three dimensions.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} x & y & z \end{pmatrix}$$

and again, vector addition and scalar multiplication are easily expressed:

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ y_1 + y_2 \\ z_1 + z_2 \end{pmatrix}$$

$$\lambda \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \lambda x \\ \lambda y \\ \lambda z \end{pmatrix}$$

Vectors in $\mathbb{R}^n$

We can define vectors in $\mathbb{R}^n$ completely analogously to how we defined row vectors (or column) vectors in $\mathbb{R}^2$ and $\mathbb{R}^3$. If we're in $\mathbb{R}^n$ for $n \geq 4$ we can't really visualize the vectors as arrows anymore, but we can still define them algebraically.

A ***column vector*** in $\mathbb{R}^n$ is a $n \times 1$ matrix, and a ***row vector*** is a $1 \times n$ matrix. We will typically use column vectors and just say ***vector*** to mean column vectors – but this is just a convenient choice.

Given two vectors in $\mathbb{R}^n$, say

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad \text{and} \quad \vec{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

then we define their sum, $\vec{x} + \vec{y}$ by adding the components of the vectors together,

$$\vec{x} + \vec{y} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{pmatrix}.$$

We define scalar multiplication by multiplying each component of the vector by the same scalar,

$$\lambda\vec{x} = \lambda \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \lambda x_1 \\ \lambda x_2 \\ \vdots \\ \lambda x_n \end{pmatrix}$$

We will usually denote the vector $-1 \cdot \vec{v}$ as just $-\vec{v}$, and define "vector subtraction" to be vector addition, but with -1 times one of the vectors:

$$\vec{v} - \vec{u} = \vec{v} + (-\vec{u}).$$

By $\vec{0}$ we always mean the vector of all zeros:

$$\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Two vectors, $\vec{v}$ and $\vec{u}$, are equal precisely when their components are equal:

$$\vec{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \vec{u}$$

really means

$$\begin{aligned} v_1 &= u_1 \\ v_2 &= u_2 \\ &\vdots \\ v_n &= u_n \end{aligned}$$

Properties of Vectors

Vector addition and scalar multiplication satisfy some of the basic algebraic properties that you would expect:

Proposition 9.5.
Let $\vec{u}$, $\vec{v}$, and $\vec{w}$ all be vectors in $\mathbb{R}^n$ (resp., $\mathbb{C}^n$) and let λ, μ be scalars in $\mathbb{R}$ (resp. $\mathbb{C}$). Then vector addition and scalar multiplication satisfy the following properties:

1. $(\vec{u} + \vec{v}) + \vec{w} = \vec{u} + (\vec{v} + \vec{w})$
2. $\vec{u} + \vec{v} = \vec{v} + \vec{u}$
3. $\lambda\,(\mu\vec{v}) = (\lambda\mu)\vec{v}$
4. $\vec{v} + \vec{0} = \vec{v}$
5. $\vec{v} + (-\vec{v}) = \vec{0}$
6. $(\lambda + \mu)\vec{v} = \lambda\vec{v} + \mu\vec{v}$
7. $\lambda(\vec{u} + \vec{v}) = \lambda\vec{u} + \lambda\vec{v}$
8. $\lambda(\mu\vec{v}) = (\lambda\mu)\vec{v}$
9. $1 \cdot \vec{v} = \vec{v}$

Vector Equations

Just we have equations involving numbers, we can have equations involving scalars. For example, consider the vectors

$$\vec{v}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \vec{v}_2 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Now consider the equation

$$x\vec{v}_1 + y\vec{v}_2 = \begin{pmatrix} 4 \\ 3 \end{pmatrix}.$$

A solution to this vector equation would be a choice of scalars x and y making the equation hold.

Let's think about what would happen if we write out all of the details in the above vector equation

$$\begin{aligned} x\vec{v}_1 + y\vec{v}_2 &= \begin{pmatrix} 4 \\ 3 \end{pmatrix} \\ \implies x\begin{pmatrix} 1 \\ 1 \end{pmatrix} + y\begin{pmatrix} 1 \\ 2 \end{pmatrix} &= \begin{pmatrix} 4 \\ 3 \end{pmatrix} \\ \implies \begin{pmatrix} x \\ x \end{pmatrix} + \begin{pmatrix} y \\ 2y \end{pmatrix} &= \begin{pmatrix} 4 \\ 3 \end{pmatrix} \\ \implies \begin{pmatrix} x+y \\ x+2y \end{pmatrix} &= \begin{pmatrix} 4 \\ 3 \end{pmatrix} \end{aligned}$$

Since two vectors are equal only when their components are equal, this means we really want to find x and y solving the system

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

So vector equations are really systems of equations in disguise! Furthermore, notice that the columns of the coefficient matrix for this system are exactly the original vectors in our vector equation!

Linear Combinations

We have two algebraic operations we can perform to vectors: scalar multiplication and vector addition. If we're given some vectors $\vec{v}_1$, $\vec{v}_2$, ..., $\vec{v}_m$ and we multiply each one by a some scalar – say we multiply $\vec{v}_i$ by λ_i – and then sum these vectors, we say that the resulting vector is a ***linear combination*** of the vectors $\vec{v}_1$, $\vec{v}_2$, ..., $\vec{v}_m$.

That is, a linear combination of $\vec{v}_1$, $\vec{v}_2$, ..., $\vec{v}_m$ is a vector that may be written as

$$\lambda_1\vec{v}_1 + \lambda_2\vec{v}_2 + \cdots + \lambda_m\vec{v}_m$$

for any choice of scalars λ_1, λ_2, ..., λ_m.

Example 9.32.

Consider the following two vectors in three-space:

$$\vec{v}_1 = \begin{pmatrix} 1 \\ -1 \\ 2 \end{pmatrix} \quad \vec{v}_2 = \begin{pmatrix} 3 \\ 3 \\ 0 \end{pmatrix}$$

One possible linear combination of these vectors is

$$7\vec{v}_1 - 4\vec{v}_2 = \begin{pmatrix} 7 \\ -7 \\ 14 \end{pmatrix} + \begin{pmatrix} -12 \\ -12 \\ 0 \end{pmatrix} = \begin{pmatrix} -5 \\ -19 \\ 14 \end{pmatrix}$$

Given $\vec{v}_1, ..., \vec{v}_m$ – all vectors of the same dimension, say n – the collection of all possible linear combinations of the vectors is called the ***span*** of the vectors and is denoted

$$\text{span}\{\vec{v}_1, \vec{v}_2, ..., \vec{v}_m\}.$$

In set-builder notation,

$$\text{span}\{\vec{v}_1, \vec{v}_2, ..., \vec{v}_m\} = \left\{ \lambda_1\vec{v}_1 + \cdots + \lambda_m\vec{v}_m \,\middle|\, \lambda_1, \lambda_2, ..., \lambda_m \in \mathbb{R} \right\}$$

Notice that in both $\mathbb{R}^2$ and $\mathbb{R}^3$, the span of a single, non-zero vector is a line through the origin. The span of two vectors could be plane (in $\mathbb{R}^2$ this would give us all possible vectors – the only plane sitting inside of $\mathbb{R}^2$ is the entire xy-plane), or it could be line. This second situation occurs when one vector is a multiple of another. That is, if our two vectors $\vec{v}_1$ and $\vec{v}_2$ have the property that $\vec{v}_2 = \mu\vec{v}_1$, then any linear combination of $\vec{v}_1$ and $\vec{v}_2$ is really just a multiple of $\vec{v}_1$:

$$\lambda_1\vec{v}_1 + \lambda_2\vec{v}_2 = \lambda_1\vec{v}_1 + \lambda_2\mu\vec{v}_1 = (\lambda_1 + \lambda_2\mu)\vec{v}_1.$$

In a situation like we say the vectors $\vec{v}_1$ and $\vec{v}_2$ are *linearly dependent*: meaning that one vector is a linear combination of another.

More generally, we say that a set of vectors $\{\vec{v}_1, ..., \vec{v}_m\}$ is ***linearly dependent*** if it's possible to write one vector as a linear combination of the others. If this can't be done – no vector is a linear combination of the others – then we say the set is ***linearly independent***.

Proposition 9.6.
A set of vectors $\{\vec{v}_1, \vec{v}_2, ..., \vec{v}_m\}$ is linearly independent if and only if

the only scalars λ_1, λ_2, ..., λ_m making the following equation hold,

$$\lambda_1\vec{v}_1 + \lambda_2\vec{v}_2 + \cdots + \lambda_m\vec{v}_m = 0$$

are $\lambda_1 = \lambda_2 = \cdots = \lambda_m = 0$.

Proof of Proposition 9.6.
If we could write

$$\lambda_1\vec{v}_1 + \lambda_2\vec{v}_2 + \cdots + \lambda_m\vec{v}_m = 0$$

for some non-zero choice of the λ_i, then we could pick one of the vectors $\vec{v}_i$ where $\lambda_i \neq 0$ and move it to the other side of the equation and divide by $-\lambda_i$ to write

$$\frac{\lambda_1}{-\lambda_i}\vec{v}_1 + \cdots + \frac{\lambda_{i-1}}{-\lambda_{i-1}}\vec{v}_{i-1} + \frac{\lambda_{i+1}}{-\lambda_{i+1}}\vec{v}_{i+1} + \cdots + \frac{\lambda_m}{-\lambda_i}\vec{v}_m = \vec{v}_i.$$

Thus if it is impossible to write one of the $\vec{v}_i$ as a linear combination of the other vectors (i.e., the vectors are linearly independent), then the only way to write $\lambda_1\vec{v}_1 + \lambda_2\vec{v}_2 + \cdots + \lambda_m\vec{v}_m = 0$ is if every λ_j was zero.

Conversely, if one of the vectors was a linear combination of the others,

$$\mu_i\vec{v}_1 + \cdots + \mu_{i-1}\vec{v}_{i-1} + \mu_{i+1}\vec{v}_{i+1} + \cdots + \mu_m\vec{v}_m = \vec{v}_i.$$

Then we can write

$$\mu_i\vec{v}_1 + \cdots + \mu_{i-1}\vec{v}_{i-1} - \vec{v}_i\mu_{i+1}\vec{v}_{i+1} + \cdots + \mu_m\vec{v}_m = 0.$$

Thus if we can not write $\lambda_1\vec{v}_1 + \cdots + \lambda_m\vec{v}_m = 0$, then the vectors must be linearly independent. □

The product of a matrix and a vector

We introduced vectors above and saw that there were two algebraic operations that could be performed on vectors: vector addition and scalar

multiplication. In general we can not multiply two vectors, but we can actually define the product of a matrix and a vector – at least if the sizes of the matrix and vector agree in a particular way. We will also see that this gives us a very concise way of expressing a system of a linear equations which will pave the way to later showing that properties of a linear system's coefficient matrix are directly related to the solutions of the system.

Suppose that $\vec{a}_1$, $\vec{a}_2$, ..., $\vec{a}_n$ are vectors in $\mathbb{R}^m$. In the last section we considered linear combinations of vectors which were scalar multiples of the vectors added together:

$$x_1\vec{v}_1 + x_2\vec{v}_2 + \cdots + x_n\vec{v}_n.$$

Notice that these scalars, x_1, x_2, ..., x_n, that we multiply each vector by could be regarded as the components of some n-dimensional vector which we might call $\vec{x}$:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}.$$

We could also think of each vector $\vec{a}_1$, ..., $\vec{a}_n$, as forming the columns of some matrix A:

$$A = \begin{pmatrix} \vec{a}_1 & \vec{a}_2 & \cdots & \vec{a}_n \\ & & & \end{pmatrix}$$

Example 9.33.
Suppose we have four three-dimensional vectors,

$$\vec{a}_1 = \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix} \quad \vec{a}_2 = \begin{pmatrix} -1 \\ 0 \\ 3 \end{pmatrix} \quad \vec{a}_3 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad \vec{a}_4 = \begin{pmatrix} 4 \\ 2 \\ -7 \end{pmatrix}$$

and we considered the linear combination

$$5\vec{a}_1 - 3\vec{a}_2 + 2\vec{a}_3 + 2\vec{a}_4.$$

Then our vector $\vec{x}$ would be

$$\vec{x} = \begin{pmatrix} 5 \\ -3 \\ 2 \\ 2 \end{pmatrix}$$

and our matrix A would be

$$A = \begin{pmatrix} 2 & -1 & 1 & 4 \\ 1 & 0 & 1 & 2 \\ 0 & 3 & 1 & -7 \end{pmatrix}.$$

In general, we will define the ***product*** of an $m \times n$ matrix A with an n-dimensional vectors $\vec{x}$ as the linear combination of the columns of A with scalars given by the components of $\vec{x}$.

Example 9.34.
If A is the matrix

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

and $\vec{x}$ is the vector

$$\vec{x} = \begin{pmatrix} -1 \\ 2 \\ 3 \end{pmatrix}$$

then the product $A\vec{x}$ is the the linear combination

$$-\begin{pmatrix} 3 \\ 2 \\ -5 \end{pmatrix} + 2\begin{pmatrix} 4 \\ 1 \\ 7 \end{pmatrix} + 3\begin{pmatrix} 0 \\ -1 \\ 2 \end{pmatrix} = \begin{pmatrix} -3+8+0 \\ -2+2-3 \\ 5+14+6 \end{pmatrix} = \begin{pmatrix} 5 \\ -3 \\ 25 \end{pmatrix}$$

Example 9.35.

If A is the matrix

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

and $\vec{x}$ is the matrix

$$\vec{x} = \begin{pmatrix} 8 \\ -3 \\ 5 \\ 2 \end{pmatrix}$$

then the product $A\vec{x}$ is the linear combination

$$\begin{aligned} & 8\begin{pmatrix} 4 \\ 0 \end{pmatrix} - 3\begin{pmatrix} 6 \\ 2 \end{pmatrix} + 5\begin{pmatrix} -5 \\ 1 \end{pmatrix} + 2\begin{pmatrix} 3 \\ 4 \end{pmatrix} \\ = & \begin{pmatrix} 32 - 18 - 25 + 6 \\ 0 - 6 + 5 + 8 \end{pmatrix} \\ = & \begin{pmatrix} -5 \\ 7 \end{pmatrix} \end{aligned}$$

Remark.
In order for this definition of the product of a matrix and a vector to make sense, it is absolutely essential that the number of columns of the matrix equals the number of rows in the vector (the dimension of the vector).

Example 9.36.

$$\begin{pmatrix} 3 & 2 & 4 \\ 1 & -2 & 3 \\ 0 & 4 & 4 \\ 1 & 1 & -1 \end{pmatrix} \begin{pmatrix} 2 \\ 0 \\ 3 \end{pmatrix} = \begin{pmatrix} 6+0+12 \\ 2+0+9 \\ 0+0+12 \\ 2+0-3 \end{pmatrix} = \begin{pmatrix} 18 \\ 11 \\ 12 \\ -1 \end{pmatrix}$$

9.4 The Matrix Equation $A\vec{x} = \vec{b}$

If $\vec{b}$ is some particular vector n-dimensional vector, we may want to know if there is a solution to the vector equation

$$x_1\vec{a}_1 + x_2\vec{a}_2 + \cdots + x_n\vec{a}_n = \vec{b}$$

which can be more easily concisely written as

$$A\vec{x} = \vec{b}$$

where A is the matrix whose columns are given by $\vec{a}_1$, ..., $\vec{a}_n$, and $\vec{x}$ is the vector containing the variables x_1, ..., x_n.

Example 9.37.
Asking for x_1, x_2, and x_3 solving the vector equation

$$x_1 \begin{pmatrix} 1 \\ -1 \\ 2 \end{pmatrix} + x_2 \begin{pmatrix} 2 \\ 3 \\ 0 \end{pmatrix} + x_3 \begin{pmatrix} 4 \\ 7 \\ 7 \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \\ 2 \end{pmatrix}$$

is the same as asking if there is a vector

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

such that

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

Notice that this is really just a system of linear equations with variables x_1, x_2, x_3 with coefficient matrix A and augmented coefficient matrix

$$\left(\begin{array}{c|c} & \\ A & \vec{b} \\ & \end{array} \right)$$

Thus solving systems of linear equations and solving the matrix equation $A\vec{x} = \vec{b}$ are two sides of the same coin.

Example 9.38.
If A and $\vec{b}$ are

$$A = \begin{pmatrix} 1 & 2 & 4 \\ -1 & 3 & 7 \\ 2 & 0 & 7 \end{pmatrix} \quad \vec{b} = \begin{pmatrix} 3 \\ 1 \\ 2 \end{pmatrix}$$

Then finding a vector $\vec{x}$ solving $A\vec{x} = b$ is the same as finding a solution (x_1, x_2, x_3) to the system

$$\begin{aligned} x_1 + 2x_2 + 4x_3 &= 3 \\ -x_1 + 3x_2 + 7x_3 &= 1 \\ 2x_1 + x_3 &= 2 \end{aligned}$$

That solving systems of linear equations and solving matrix equations $A\vec{x} = b$ are really the same thing leads to the following theorem.

Proposition 9.7.
Let A be an $m \times n$ matrix. Then the system of linear equations with augmented coefficient matrix $\left(A \mid \vec{b} \right)$ has a solution if and only if $\vec{b}$ is in the span of the columns of A: $\vec{a}_1$, $\vec{a}_2$, ..., $\vec{a}_n$.

Proof.
Suppose first that the system $A\vec{x} = \vec{b}$ has a solution. Then, by the definition of the product $A\vec{x}$, $\vec{b}$ is a linear combination of the columns of A. In particular, if the components of $\vec{x}$ are x_1, x_2, ..., x_n, then we have

$$\vec{b} = x_1\vec{a}_1 + x_2\vec{a}_2 + \cdots + x_n\vec{a}_n.$$

Now suppose that $\vec{b}$ is in the span of the columns of A, again, by the definition of the product $A\vec{x}$, this precisely means the system $A\vec{x} = \vec{b}$ has a solution. In particular, if

$$\vec{b} = \lambda_1\vec{a}_1 + \lambda_2\vec{a}_2 + \cdots + \lambda_n\vec{a}_n,$$

then a solution to $A\vec{x} = \vec{b}$ is given by

$$\vec{x} = \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{pmatrix}$$

□

Notice that nothing deep is going on in this proposition: we're just translating the language of systems of linear equations to the language of matrix equations.

Remark.
Sometimes the hardest part of solving a mathematical problem is determining the right way to express it: some problems seem easier or more difficult depending on the language you use to describe them. We are in the process of taking the ideas we described at the start of the semester (systems of linear equations) and converting them into another language (matrices and vectors) because, as we will see, it is actually a lot easier to think about many problems in terms of matrices and vectors. This may sound strange at first, especially if you're learning about matrices and vectors for the first time, but using the language matrices will actually make many problems much easier to

think about and ultimately solve.

Example 9.39.
Is there a solution to the following matrix equation?

$$\begin{pmatrix} 2 & 3 & 4 \\ 6 & 18 & 24 \\ 2 & 3 & 9 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -12 \\ -13 \end{pmatrix}$$

By the definition of the product of a matrix and a vector, this really means we want to find x_1, x_2, and x_3 such that

$$x_1 \begin{pmatrix} 2 \\ 6 \\ 2 \end{pmatrix} + x_2 \begin{pmatrix} 3 \\ 18 \\ 3 \end{pmatrix} + x_3 \begin{pmatrix} 4 \\ 24 \\ 9 \end{pmatrix} = \begin{pmatrix} 2 \\ -12 \\ -13 \end{pmatrix}$$

But if we do the scalar multiplication and vector addition we can rewrite the left-hand side of this equation to obtain

$$\begin{pmatrix} 2x_1 + 3x_2 + 4x_3 \\ 6x_1 + 18x_2 + 24x_3 \\ 2x_1 + 3x_2 + 9x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -12 \\ -13 \end{pmatrix}$$

Equating components of the vectors, this is really a system of equations,

$$\begin{aligned} 2x_1 + 3x_2 + 4x_3 &= 2 \\ 6x_1 + 18x_2 + 24x_3 &= -12 \\ 2x_1 + 3x_2 + 9x_3 &= -13 \end{aligned}$$

We know how to solve a system like this, though: we write down the augmented coefficient matrix (which we could have easily read off from the original matrix equation),

$$\left(\begin{array}{ccc|c} 2 & 3 & 4 & 2 \\ 6 & 18 & 24 & -12 \\ 2 & 3 & 9 & -13 \end{array}\right)$$

then proceed to put the matrix in RREF, which gives us

$$\begin{pmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & -3 \end{pmatrix}$$

This tells us the system of equations is equivalent to

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

and so we have a unique solution to the system.

In terms of the vector equation, we have

$$4\begin{pmatrix} 2 \\ 6 \\ 2 \end{pmatrix} + 2\begin{pmatrix} 3 \\ 18 \\ 3 \end{pmatrix} - 3\begin{pmatrix} 4 \\ 24 \\ 9 \end{pmatrix} = \begin{pmatrix} 2 \\ -12 \\ -13 \end{pmatrix}$$

And so the vector solving our original matrix equation is

$$\vec{x} = \begin{pmatrix} 4 \\ 2 \\ -3 \end{pmatrix}.$$

That is,

$$\begin{pmatrix} 2 & 3 & 4 \\ 6 & 18 & 24 \\ 2 & 3 & 9 \end{pmatrix}\begin{pmatrix} 4 \\ 2 \\ -3 \end{pmatrix} = \begin{pmatrix} 2 \\ -12 \\ -13 \end{pmatrix}$$

9.5 Existence of Solutions

We have seen that systems of linear equations sometimes have a unique solution, sometimes have no solution, and sometimes have infinitely-many solutions. Whether a solution exists or not depends less on the right-hand side of the equations of the system, and more about the coefficients of the system. In the language of matrices and vectors, solving $A\vec{x} = \vec{b}$ depends more on what A is than on what $\vec{b}$ is. In particular, we have the following theorem:

Theorem 9.8.
Suppose that A is an $m \times n$ matrix and $\vec{x}$ and $\vec{b}$ are m-dimensional vectors. Then the following are equivalent:

(a) The equation $A\vec{x} = \vec{b}$ has a solution for every *choice of $\vec{b}$.*

(b) Every m-dimensional vector $\vec{b}$ is a linear combination of the columns of A.

(c) The columns of A span $\mathbb{R}^m$.

(d) A has a pivot position in every row.

Remark.
The theorem above uses the phrase *the following are equivalent*. This means that if one of the statements is true, then all of the statements are true; if one of the statements is false, then all of the statements are false. This is really a shorthand for several *if and only if* statements. When we say "the following are equivalent: (a) ... (b) ... (c) ... (d) ..." what we really means is that statement (a) happens if and only if statement (b) happens if and only if statement (c) happens if and only if statement (d) happens.

We've seen before that "if and only if" statements are really two statements: there's actually two things to prove. If you want to prove "(a) if and only if (b)" then you need to show that statement (a) implies statement (b) and also that statement (b) implies statement (a). Thus it might seem like for the above we need to show twelve different things: (a) implies (b), (b) implies (a), (a) implies (c), (c) implies (a), (a) implies (d), (d) implies (a), (b) implies (c), (c) implies (b), and so on.

It would be completely correct to show all of these implications, but luckily there's an easier way. We can instead show that (a) implies (b), (b) implies (c), (c) implies (d), and finally (d) implies (a). If we show this then everything else can be deduced. For example, if we show the four implications above, then the fact that (c) implies (a), for instance comes for free: we know (c) implies (d) and also that (d)

implies (a), hence (c) implies (a) as well.

In hand-written notes, *the following are equivalent* is often abbreviated *TFAE*.

Proof of Theorem 9.8.

$(a) \implies (b)$

Because of the way we have defined the product of a matrix and a vector, saying $A\vec{x} = \vec{b}$ means exactly that $\vec{b}$ is a linear combination of the columns of A. Hence if $A\vec{x} = \vec{b}$ has a solution for every $\vec{b}$, then it must be the case that every $\vec{b}$ can be written as a linear combination of the columns of A.

$(b) \implies (c)$

The span of a set of vectors is exactly the set of all possible linear combinations of those vectors. So if every vector in $\mathbb{R}^m$ can be written as a linear combination of the columns of A, then the span of the columns of A is all of $\mathbb{R}^m$.

$(c) \implies (d)$

We will prove the contrapositive: if (d) does not occur, then (c) can't occur either.

So suppose that there was some row that *did not* have a pivot. This means precisely that the row-reduced echelon form of A has a row of all zeros (otherwise we would have a leading entry of 1 which would be our pivot). We could then find choices of $\vec{b}$ so that the row-reduced echelon form of $\left(A \mid \vec{b}\right)$ has a row of all zeros followed by a 1. Thus the system has no solution which means $\vec{b}$ can't be written as a linear combination of the columns of A.

We've proven the contrapositive "if not (d), then not (c)" which is logically equivalent to the original statement "if (c), then (d)."

$(d) \implies (a)$

Finally, suppose that A has a pivot position in every row. Then

the row-reduced echelon form of A has no rows of all zeros, and we can solve any system $A\vec{x} = \vec{b}$.

□

9.6 Properties of Ax

Algebraic properties

We have defined a new algebraic operation: multiplying an $m \times n$ matrix A with an m-dimensional vector $\vec{x}$. Anytime we introduce a new operation, it's natural to ask what kind of algebraic properties that operation satisfies. The following two properties are absolutely fundamental and will form the basis for what's to come when we define linear transformations.

Theorem 9.9.
If A is an $m \times n$ matrix, then for every pair of n-dimensional vectors $\vec{x}$ and $\vec{y}$, and every scalar $\lambda \in \mathbb{R}$, we have the following properties:

(a) $A(\vec{x} + \vec{y}) = A\vec{x} + A\vec{y}$

(b) $A(\lambda \cdot \vec{x}) = \lambda \cdot A\vec{x}$

Proof.
We do a direct proof, and simply verify these properties hold for any arbitrary $m \times n$ matrix A, arbitrary m-dimensional vectors $\vec{x}$ and $\vec{y}$, and arbitrary scalar λ.

We may suppose that A has the form

$$A = \begin{pmatrix} \vec{a}_1 & \vec{a}_2 & \cdots & \vec{a}_n \end{pmatrix}$$

and that

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} \quad \vec{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}.$$

(a)

$$\begin{aligned} A(\vec{x}+\vec{y}) &= A\left(\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}\right) \\ &= A\begin{pmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_m+y_m \end{pmatrix} \\ &= (x_1+y_1)\vec{a}_1 + (x_2+y_2)\vec{a}_2 + \cdots + (x_m+y_m)\vec{a}_m \\ &= x_1\vec{a}_1 + y_1\vec{a}_1 + x_2\vec{a}_2 + y_2\vec{a}_2 + \cdots + x_m\vec{a}_m + y_m\vec{a}_m \\ &= x_1\vec{a}_1 + x_2\vec{a}_2 + \cdots + x_m\vec{a}_m + y_1\vec{a}_1 + y_2\vec{a}_2 + \cdots + y_m\vec{a}_m \\ &= A\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} + A\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} \\ &= A\vec{x} + A\vec{y} \end{aligned}$$

(b)

$$\begin{aligned} A(\lambda\cdot\vec{x}) &= A\begin{pmatrix} \lambda x_1 \\ \lambda x_2 \\ \vdots \\ \lambda x_m \end{pmatrix} \\ &= \lambda x_1\vec{a}_1 + \lambda x_2\vec{a}_2 + \cdots + \lambda x_m\vec{a}_m \\ &= \lambda\cdot(x_1\vec{a}_1 + x_2\vec{a}_2 + \cdots + x_m\vec{a}_m) \\ &= \lambda\cdot A\vec{x} \end{aligned}$$

□

Computational properties

There is an alternative way to think about the product $A\vec{x}$ that is sometimes handy. Notice that if A is an $m \times n$ matrix and $\vec{x}$ is an m-dimensional vector, then the product $A\vec{x}$ is also an n-dimensional vector: it's a linear combination of n-dimensional vectors (the columns of A). This vector can be generated one element at a time by walking across each row of the matrix A, while simultaneously going down the column vector $\vec{x}$ element by element, multiplying the elements and adding them up.

Example 9.40.
Consider the product

$$\begin{pmatrix} 2 & 0 & -1 & 1 \\ 3 & 1 & 0 & 2 \\ 2 & -2 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 4 \\ -2 \\ 1 \end{pmatrix}$$

To get the first element in the product we look at the first row of the matrix, multiplying the entry in the j-th column by the j-th entry in the vector.

For the first entry we have

$$1 \cdot 2 + 4 \cdot 0 + (-2) \cdot (-2) + 1 \cdot 1 = 7.$$

We get the second entry,

$$1 \cdot 3 + 4 \cdot 1 + (-2) \cdot 0 + 1 \cdot 2 = 9.$$

For the third entry,

$$1 \cdot 2 + 4 \cdot (-2) + (-2) \cdot 1 + 1 \cdot 1 = -7.$$

And thus the product is

$$\begin{pmatrix} 7 \\ 9 \\ -7 \end{pmatrix}.$$

9.7 Matrix algebra

In this section we discuss the various types of operations that can be performed on matrices, and the algebra of these operations.

Linear Transformations

Many of the operations we perform on matrices have an interpretation in terms of functions, and understanding that interpretation helps to motivate why some of the constructions below are things we should care about.

We will say that a map $T : \mathbb{R}^n \to \mathbb{R}^m$ (that is, a map that takes n-dimensional vectors and converts them into m-dimensional vectors) is a ***linear transformation*** if it satisfies the following two axioms:

(i) For every pair of n-dimensional vectors $\vec{u}, \vec{v} \in \mathbb{R}^n$, T satisfies the following equation:

$$T(\vec{u} + \vec{v}) = T(\vec{u}) + T(\vec{v}).$$

(ii) For every n-dimensional vector $\vec{v}$ and every scalar $\lambda \in \mathbb{R}$, we have

$$T(\lambda\vec{v}) = \lambda T(\vec{v}).$$

Recall that there are two basic operations we can perform on vectors: vector addition and scalar multiplication. Linear transformations are precisely the maps that "respect" these two operations.

Example 9.41.
Consider the following which takes two-dimensional vectors and transforms them into three-dimensional vectors:

$$T : \mathbb{R}^2 \to \mathbb{R}^3$$
$$T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ -x \\ x + y \end{pmatrix}$$

This map takes $\begin{pmatrix}1\\3\end{pmatrix}$ and turns it into the vector

$$T\begin{pmatrix}1\\3\end{pmatrix} = \begin{pmatrix}3\\-1\\4\end{pmatrix}$$

and it takes the vector $\begin{pmatrix}-2\\-6\end{pmatrix}$ and turns it into

$$T\begin{pmatrix}-2\\-6\end{pmatrix} = \begin{pmatrix}-6\\2\\-8\end{pmatrix}$$

Example 9.42.
The following map takes four-dimensional vectors and turns them into two-dimensional vectors:

$$T : \mathbb{R}^4 \to \mathbb{R}^2$$
$$T\begin{pmatrix}x_1\\x_2\\x_3\\x_4\end{pmatrix} = \begin{pmatrix}x_1 - 2x_4\\4x_3 + x_2\end{pmatrix}$$

Here are some examples of what this function does:

$$T\begin{pmatrix}1\\2\\3\\4\end{pmatrix} = \begin{pmatrix}-7\\14\end{pmatrix}$$

$$T\begin{pmatrix}7\\2\\0\\-1\end{pmatrix} = \begin{pmatrix}9\\2\end{pmatrix}$$

$$T\begin{pmatrix}8\\4\\3\\3\end{pmatrix} = \begin{pmatrix}2\\16\end{pmatrix}$$

Notice that these two different maps are actually given by matrices:

Example 9.43.
The map $T : \mathbb{R}^2 \to \mathbb{R}^3$ from Example 9.41 is given by multiplying a vector $\begin{pmatrix}x\\y\end{pmatrix}$ with the matrix

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

For example,

$$\begin{pmatrix}0 & 1\\-1 & 0\\1 & 1\end{pmatrix}\begin{pmatrix}1\\3\end{pmatrix} = 1\begin{pmatrix}0\\-1\\1\end{pmatrix} + 3\begin{pmatrix}1\\0\\1\end{pmatrix} = \begin{pmatrix}3\\-1\\4\end{pmatrix}.$$

Example 9.44.
The map $T : \mathbb{R}^4 \to \mathbb{R}^2$ from Example 9.42 is given by multiplying a

four-dimensional vector with the matrix

$$A = \begin{pmatrix} 1 & 0 & 0 & -2 \\ 0 & 1 & 4 & 0 \end{pmatrix}$$

For example,

$$\begin{pmatrix} 1 & 0 & 0 & -2 \\ 0 & 1 & 4 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} -7 \\ 14 \end{pmatrix}$$

In fact, *every matrix* determines such a map: Every $m \times n$ matrix defines a map from $\mathbb{R}^n$ to $\mathbb{R}^m$ by matrix multiplication:

$$\vec{x} \mapsto A\vec{x}.$$

By the properties of multiplication between matrices and vectors, we see that such a map is always a linear transformation. In fact, it turns out that every linear transformation is determined by a matrix in this way.

The Matrix of a Linear Transformation

We said above that every matrix determines a linear transformation. It turns out, however, that *every* linear transformation is determined by a matrix. That is, for every linear transformation $T : \mathbb{R}^m \to \mathbb{R}^n$, there is some $m \times n$ matrix A such that $T(\vec{v}) = A\vec{v}$.

To see this, let's notice that every n-dimensional vector can be written as a linear combination of the vectors

$$\vec{e}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \quad \vec{e}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \quad \vec{e}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \quad \cdots \quad \vec{e}_n = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}.$$

That is, $\vec{e}_i$ is the vector that has all zeros except for a 1 in the i-th row.

Now suppose $T : \mathbb{R}^n \to \mathbb{R}^m$ is a linear transformation, and so $T(\vec{e}_i)$ is some m-dimensional vector – let's call it $\vec{a}_i$. Now consider the matrix A

whose columns are given by these vectors,

$$A = \begin{pmatrix} \vec{a}_1 & \vec{a}_2 & \cdots & \vec{a}_n \end{pmatrix}.$$

This matrix represents our linear transformation: if

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

is any vector, then we have

$$\begin{aligned} & T(\vec{x}) \\ =& T\left(x_1\vec{e}_1 + x_2\vec{e}_2 + \cdots + x_n\vec{e}_n\right) \\ =& x_1T(\vec{e}_1) + x_2T(\vec{e}_2) + \cdots + x_nT(\vec{e}_n) \\ =& x_1\vec{a}_1 + x_2\vec{a}_2 + \cdots + x_n\vec{a}_n \end{aligned}$$

But notice that

$$\begin{aligned} & x_1\vec{a}_1 + x_2\vec{a}_2 + \cdots + x_n\vec{a}_n \\ =& \begin{pmatrix} \vec{a}_1 & \vec{a}_2 & \cdots & \vec{a}_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \\ =& A\vec{x} \end{aligned}$$

Thus every linear transformation is determined by some matrix!

Matrix Addition and Scalar Multiplication

Let $T : \mathbb{R}^n \to \mathbb{R}^m$ and $S : \mathbb{R}^n \to \mathbb{R}^m$ be the corresponding linear transformations. We can produce a new linear transformation $T + S : \mathbb{R}^n \to \mathbb{R}^m$ by adding vectors. That is, given a vector $\vec{v} \in \mathbb{R}^n$ we consider the map

$$\vec{v} \mapsto T(\vec{v}) + S(\vec{v}).$$

This gives us a new map which we denote $T + S$. It's easy to see that if T and S are linear transformations, then so is $T + S$:

$$\begin{aligned}(T+S)(\vec{v}+\vec{w}) =& T(\vec{v}+\vec{w}) + S(\vec{v}+\vec{w}) \\ =& T(\vec{v}) + T(\vec{w}) + S(\vec{v}) + S(\vec{w}) \\ =& T(\vec{v}) + S(\vec{v}) + T(\vec{w}) + S(\vec{w}) \\ =& (T+S)(\vec{v}) + (T+S)(\vec{w})\end{aligned}$$

$$\begin{aligned}(T+S)(\lambda\vec{v}) =& T(\lambda\vec{v}) + S(\lambda\vec{v}) \\ =& \lambda T(\vec{v}) + \lambda S(\vec{v}) \\ =& \lambda(T+S)(\vec{v})\end{aligned}$$

Since $T + S$ is a linear transformation, there must be some matrix representing it. Before we determine what this matrix must be, let's suppose that T is given by the matrix A with columns $\vec{a}_1$, $\vec{a}_2$, ..., $\vec{a}_n$, and S is determined by the matrix B with columns $\vec{b}_1$, $\vec{b}_2$, ..., $\vec{b}_n$.

We know that the columns of the matrix representing $T+S$ are given by the vectors $(T + S)(\vec{e}_i)$, where $\vec{e}_i$ is the n-dimensional vector that consists entirely of zero except for a 1 in the i-th component.

Notice

$$\begin{aligned}(T+S)(\vec{e}_i) =& T(\vec{e}_i) + S(\vec{e}_i) \\ =& a_i + b_i\end{aligned}$$

That is, the columns of the matrix representing $T + S$ are determined by adding the columns of the matrices representing T and S.

Example 9.45.
Suppose T and S are the linear transformations corresponding to the matrices

$$A = \begin{pmatrix} 1 & 0 & 2 \\ 3 & -1 & 4 \\ 5 & 2 & 2 \\ 1 & -1 & -1 \end{pmatrix} \quad B = \begin{pmatrix} -1 & 3 & 3 \\ 2 & 2 & -2 \\ 1 & 1 & -3 \\ 4 & 1 & 2 \end{pmatrix}$$

Then the matrix corresponding to $T + S$ is

$$\begin{pmatrix} 0 & 3 & 5 \\ 5 & 1 & 2 \\ 6 & 3 & -1 \\ 5 & 0 & 1 \end{pmatrix}$$

The matrix we get by adding the columns of a matrix A with the columns of a matrix B like this is denoted $A + B$. Notice that since we add vectors (the columns of the matrices) component-by-component, we add matrices component-by-component as well – this also means that addition of matrices only makes sense if the matrices are the same size.

Example 9.46.

$$\begin{pmatrix} 2 & 3 & 4 & 2 \\ 1 & 0 & 2 & -1 \\ 3 & 4 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 & -1 & 1 & 0 \\ 2 & 2 & 7 & 13 \\ 4 & 2 & 9 & 1 \end{pmatrix} = \begin{pmatrix} 3 & 2 & 5 & 2 \\ 3 & 2 & 9 & 12 \\ 7 & 6 & 9 & 2 \end{pmatrix}$$

Given a linear transformation $T : \mathbb{R}^n \to \mathbb{R}^m$ and a scalar μ we can define a new map by multiplying the outputs of T with μ: $\vec{v} \mapsto \mu T(\vec{v})$. This map is denoted μT and is also a linear transformation:

$$\begin{aligned} \mu T(\vec{v} + \vec{w}) =& \mu(T(\vec{v}) + T(\vec{w})) \\ =& \mu T(\vec{v}) + \mu T(\vec{w}) \end{aligned}$$

$$\mu T(\lambda \vec{v}) = \mu \lambda T(\vec{v})$$

Since μT is a linear transformation it is represented by some matrix whose columns are $\mu T(\vec{e}_i)$. If T is represented by matrix A with columns $\vec{a}_i = T(\vec{e}_i)$, then μT is represented by the matrix with columns $\mu \vec{a}_i$. That is, the matrix representing μT is simply the matrix representing T, but with every entry multiplied by μ.

Example 9.47.
If $T : \mathbb{R}^2 \to \mathbb{R}^4$ is given by the matrix

$$\begin{pmatrix} 4 & 2 \\ 6 & -3 \\ 7 & 8 \\ 2 & 0 \end{pmatrix},$$

then the matrix representing $-5T$ is given by

$$\begin{pmatrix} -20 & -10 \\ -30 & 15 \\ -35 & -40 \\ -10 & 0 \end{pmatrix}.$$

The matrix obtained by multiplying each entry of a matrix A by μ is denoted μA.

Example 9.48.

$$3\begin{pmatrix} 2 & 7 & 1 & 0 \\ 4 & 2 & -2 & 3 \\ 1 & 1 & 4 & 2 \end{pmatrix} = \begin{pmatrix} 6 & 21 & 3 & 0 \\ 12 & 6 & -6 & 9 \\ 3 & 3 & 12 & 6 \end{pmatrix}$$

So we have two operations we can perform on matrices: matrix addition and scalar multiplication, corresponding to performing vector addition and scalar multiplication with the corresponding linear transformations.

Anytime we introduce an algebraic operation such as this, we'd like to know what properties the operation may satisfy; and if we have multiple operations, we want to know how the different operations interact with one another.

In what's to follow we will use 0 to mean the zero matrix: the matrix of all zeros. It will usually be clear from context when we write 0 whether we're referring to the scalar number zero, or the zero matrix.

Theorem 9.10.
Let A, B, and C be $m \times n$ matrices and let λ and μ be scalars. Then matrix addition and scalar multiplication satisfy the following properties:

(i) $A + B = B + A$

(ii) $(A + B) + C = A + (B + C)$

(iii) $A + 0 = A$

(iv) $A - A = A + (-A) = 0$

(v) $\lambda(A + B) = \lambda A + \lambda B$

(vi) $(\lambda + \mu)A = \lambda A + \mu A$

(vii) $(\lambda\mu)A = \lambda(\mu A)$

Proof.
The proofs of each of the above properties are basically identical: simply write out the left-hand and right-hand sides of each property and verify that everything is equal. This is easy, but quite tedious, so we will just give the proof of the first property.

Suppose that A and B are $m \times n$ matrices where the element in the i-th row and j-th column of each matrix is a_{ij} and b_{ij}, respectively. Then the element in the i-th row and j-th column of the sum $A + B$ is $a_{ij} + b_{ij}$. Since a_{ij} and b_{ij} are just numbers (real or complex, it doesn't matter), we know $a_{ij} + b_{ij} = b_{ij} + a_{ij}$, but this is the entry in the i-th row and j-th column of $B + A$. Thus $A + B$ and $B + A$ have the same entries are so are the same matrix: $A + B = B + A$. □

Matrix Multiplication

If $f : A \to B$ and $g : B \to C$ is a map, then their ***composition*** is a map from A to C given by

$$a \mapsto g(f(a))$$

and denoted $g \circ f : A \to C$.

Consider linear transformations $T : \mathbb{R}^n \to \mathbb{R}^m$ and $S : \mathbb{R}^m \to \mathbb{R}^p$. Their composition, $S \circ T$, is a map from $\mathbb{R}^n$ to $\mathbb{R}^p$ which takes vectors in $\mathbb{R}^n$ and maps them into $\mathbb{R}^p$ according to the rule

$$\vec{v} \mapsto S(T(\vec{v})).$$

Notice that this is a linear transformation:

$$\begin{aligned}(S \circ T)(\vec{v} + \vec{w}) =& S(T(\vec{v} + \vec{w})) \\ =& S(T(\vec{v}) + T(\vec{w})) \\ =& S(T(\vec{v})) + S(T(\vec{w})) \\ =& (S \circ T)(\vec{v}) + (S \circ T)(\vec{w})\end{aligned}$$

$$\begin{aligned}(S \circ T)(\lambda\vec{v}) =& S(T(\lambda\vec{v})) \\ =& S(\lambda T(\vec{v})) \\ =& \lambda S(T(\vec{v})) \\ =& \lambda(S \circ T)(\vec{v}).\end{aligned}$$

Since $S \circ T : \mathbb{R}^n \to \mathbb{R}^p$ is a linear transformation it must be represented by some $p \times n$ matrix, the columns of which are given by $S(T(\vec{e}_i))$. To determine what these columns look like, suppose that T is represented by the $m \times n$ matrix A and S is represented by the $p \times m$ matrix B. Suppose the columns of A are $\vec{a}_1$, $\vec{a}_2$, ..., $\vec{a}_n$ and the columns of B are $\vec{b}_1$, $\vec{b}_2$, ..., $\vec{b}_m$. Then $S(T(\vec{e}_i)) = S(\vec{a}_i)$. Applying S corresponds to multiplying by the matrix B, however, so $S(\vec{a}_i) = B\vec{a}_i$. That is, the matrix representing $S \circ T$ has the form

$$\begin{pmatrix} B\vec{a}_1 & B\vec{a}_2 & \cdots & B\vec{a}_n \\ & & & \end{pmatrix}$$

If we suppose that $\vec{a}_i$ has the form

$$\vec{a}_i = \begin{pmatrix} a_{1,i} \\ a_{2,i} \\ \vdots \\ a_{m_i} \end{pmatrix}$$

then the i-th column of the matrix above is

$$B\vec{a}_i = a_{1,i}\vec{b}_1 + a_{2,i}\vec{b}_2 + \cdots + a_{m,i}\vec{b}_i$$

Supposing that the column $\vec{b}_j$ has the form

$$\vec{b}_j = \begin{pmatrix} b_{1,j} \\ b_{2,j} \\ \vdots \\ b_{p,j} \end{pmatrix}$$

We have

$$\begin{aligned} B\vec{a_i} =& a_{1,i} \begin{pmatrix} b_{1,1} \\ b_{2,1} \\ \vdots \\ b_{p,1} \end{pmatrix} + a_{2,i} \begin{pmatrix} b_{1,2} \\ b_{2,2} \\ \vdots \\ b_{p,2} \end{pmatrix} + \cdots + a_{m,i} \begin{pmatrix} b_{1,m} \\ b_{2,m} \\ \vdots \\ b_{p,m} \end{pmatrix} \\ =& \begin{pmatrix} a_{1,i}b_{1,1} + a_{2,i}b_{1,2} + \cdots + a_{m,i}b_{1,m} \\ a_{1,i}b_{2,1} + a_{2,i}b_{2,2} + \cdots + a_{m,i}b_{2,m} \\ \vdots a_{1,i}b_{p,1} + a_{2,i}b_{p,2} + \cdots + a_{m,i}b_{p,m} \end{pmatrix} \\ =& \begin{pmatrix} b_{1,1}a_{1,i} + b_{1,2}a_{2,i} + \cdots + b_{1,m}a_{m,i} \\ b_{2,1}a_{1,i} + b_{2,2}a_{2,i} + \cdots + b_{2,m}a_{m,i} \\ \vdots \\ b_{p,1}a_{1,i} + b_{p,2}a_{2,i} + \cdots + b_{p,m}a_{m,i} \end{pmatrix} \end{aligned}$$

Putting this all together, the entry in the i-th row and j-th column of our $p \times n$ matrix is

$$\sum_{k=1}^{m} b_{i,k}a_{k,j}.$$

This matrix is called the ***product*** of the matrices B and A and is denoted BA.

Just to reiterate: given two matrices A and B where A has size $m \times n$ and B has size $n \times p$, we can define the product AB which is a $m \times p$ matrix whose entry in the i-th row and j-th column is

$$\sum_{k=1}^{n} a_{i,k}b_{k,j}$$

This matrix corresponds to the composition of the linear transformations determined by A and B; applying B first and then A.

Notice that if A is $1 \times n$ and B is $n \times 1$, then this multiplication is easy to do:

$$\begin{pmatrix} a_1 & a_2 & a_3 & \cdots & a_n \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_1b_1 + a_2b_2 + a_3b_3 + \cdots + a_nb_n \end{pmatrix}$$

Example 9.49.

$$\begin{pmatrix} 3 & 7 & 2 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 4 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \cdot 2 + 7 \cdot (-1) + 2 \cdot 4 + 1 \cdot 0 \end{pmatrix} = \begin{pmatrix} 7 \end{pmatrix}$$

Since the product of a row vector and a column vector like this is always a 1×1 matrix it's just a single number, and so we usually think of this as being a scalar instead of a vector.

Remark.
If you've taken multivariable calculus, you might notice that multiplying a row vector and a column vector like this is the same as taking the dot product of two vectors.

The entry in the i-th row, j-th column of the product AB is obtained by multiplying the i-th row of A with the j-th column of B. This observation greatly simplifies the calculation of the product of two matrices.

Example 9.50.

Let A and B be the matrices below, and compute the product AB.

$$A = \begin{pmatrix} 4 & 6 & 3 \\ 0 & 1 & -1 \\ 3 & 2 & 2 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 1 \\ 1 & 2 \\ 0 & 2 \end{pmatrix}$$

The first row, first column of AB will be the product of the first row of A with the first column of B:

$$\begin{pmatrix} 4 & 6 & 3 \end{pmatrix} \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} = 4 \cdot 3 + 6 \cdot 1 + 3 \cdot 0 = 18$$

The first row, second column of AB will be the product of the first row of A and the second column of B:

$$\begin{pmatrix} 4 & 6 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix} = 4 \cdot 1 + 6 \cdot 2 + 3 \cdot 2 = 22$$

The second row, first column of AB is the product of the second row of A and the first column of B:

$$\begin{pmatrix} 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} = 0 \cdot 3 + 1 \cdot 1 + (-1) \cdot 0 = 1$$

Continuing like this we can compute each entry of AB:

$$AB = \begin{pmatrix} 18 & 22 \\ 1 & 0 \\ 11 & 11 \end{pmatrix}$$

Example 9.51.

Suppose $T : \mathbb{R}^3 \to \mathbb{R}^2$ and $S : \mathbb{R}^2 \to \mathbb{R}^4$ are given by

$$T\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x+z \\ y-z \end{pmatrix}$$

$$S\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2y \\ x+y \\ x-y \\ 3x \end{pmatrix}.$$

What does the composition $S \circ T$ do? What is the corresponding matrix?

Our map $S \circ T$ will take convert three-dimensional vectors into four-dimensional vectors in the following way:

$$\begin{aligned} S \circ T \begin{pmatrix} x \\ y \\ z \end{pmatrix} =& S\left(T\begin{pmatrix} x \\ y \\ z \end{pmatrix}\right) \\ =& S\begin{pmatrix} x+z \\ y-z \end{pmatrix} \\ =& \begin{pmatrix} 2(y-z) \\ x+z+y-z \\ x+z-(y-z) \\ 3(x+z) \end{pmatrix} \\ =& \begin{pmatrix} 2y-2z \\ x+y \\ x-y+2z \\ 3x+3z \end{pmatrix} \end{aligned}$$

We could compute the matrix of $S \circ T$ in two different ways: by multiplying the matrices of S and T, or by computing $S \circ T(\vec{e}_i)$. We'll compute the matrix both ways.

First notice that the matrix of T is

$$B = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix}.$$

The matrix of S is

$$A = \begin{pmatrix} 0 & 2 \\ 1 & 1 \\ 1 & -1 \\ 3 & 0 \end{pmatrix}.$$

The matrix of $S \circ T$ is thus

$$\begin{aligned} AB &= \begin{pmatrix} 0 & 2 \\ 1 & 1 \\ 1 & -1 \\ 3 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix} \\ &= \begin{pmatrix} 0 \cdot 1 + 2 \cdot 0 & 0 \cdot 0 + 2 \cdot 1 & 0 \cdot 1 + 2 \cdot (-1) \\ 1 \cdot 1 + 1 \cdot 0 & 1 \cdot 0 + 1 \cdot 1 & 1 \cdot 1 + 1 \cdot (-1) \\ 1 \cdot 1 + (-1) \cdot 0 & 1 \cdot 0 + (-1) \cdot 1 & 1 \cdot 1 + (-1) \cdot -1 \\ 3 \cdot 1 + 0 \cdot 0 & 3 \cdot 0 + 0 \cdot 1 & 3 \cdot 1 + 0 \cdot (-1) \end{pmatrix} \\ &= \begin{pmatrix} 0 & 2 & -2 \\ 1 & 1 & 0 \\ 1 & -1 & 2 \\ 3 & 0 & 3 \end{pmatrix} \end{aligned}$$

Just to confirm this is correct, notice

$$S \circ T(\vec{e}_1) = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 3 \end{pmatrix}$$

$$S \circ T(\vec{e}_2) = \begin{pmatrix} 2 \\ 1 \\ -1 \\ 0 \end{pmatrix}$$

$$S \circ T(\vec{e}_3) = \begin{pmatrix} -2 \\ 0 \\ 2 \\ 3 \end{pmatrix}$$

Properties of Matrix Multiplication

Before mentioning some of the algebraic properties that matrix multiplication satisfies, we mention some things about matrix multiplication which are very different when compared to the usual multiplication of real numbers that you're used to.

Notice that unlike multiplication of real numbers, multiplication of matrices is not commutative. That is, $AB \neq BA$ in general;

Example 9.52.
Suppose A and B are the matrices below:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & -1 \\ 4 & 2 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 2 & -1 & 1 \\ 1 & 1 & 2 \\ 3 & -1 & -2 \end{pmatrix}$$

Then

$$AB = \begin{pmatrix} 13 & -2 & -1 \\ -3 & 1 & 2 \\ 13 & -3 & 6 \end{pmatrix}$$

$$BA = \begin{pmatrix} 6 & 6 & 8 \\ 9 & 6 & 4 \\ -5 & 2 & 8 \end{pmatrix}$$

Also unlike normal multiplication of numbers, we can have two non-zero matrices that multiply to the zero matrix.

Example 9.53.

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & -1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 1 & 2 \\ -1 & -2 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 0 & -2 & 4 \\ 0 & 1 & 0 & -1 & 1 \\ 2 & 1 & 1 & 1 & 7 \\ 0 & 2 & 0 & -2 & 2 \\ 3 & 2 & 3 & 4 & 14 \end{pmatrix} \begin{pmatrix} 2 & 2 \\ 1 & -1 \\ 2 & 6 \\ 0 & -2 \\ -1 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

In general we can't "divide" matrices either. For example, if x, y and z are real numbers and $xy = xz$, then as long as $x \neq 0$ we can divide out the x's to conclude $y = z$. This is not the case for matrices.

Example 9.54.
Let A, B, and C be the matrices below.

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 0 & 2 & 2 \\ 2 & 1 & 5 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 2 & 2 \\ 3 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \quad C = \begin{pmatrix} -1 & 0 & -2 \\ 2 & 0 & -3 \\ 2 & 1 & 3 \end{pmatrix}$$

Notice that

$$AB = AC = \begin{pmatrix} 15 & 5 & 4 \\ 8 & 2 & 0 \\ 10 & 5 & 8 \end{pmatrix}$$

even though $B \neq C$.

Now that we've seen some of the properties that matrix multiplication *doesn't* satisfy, let's mention some of the properties that *are* satisfied.

Theorem 9.11.
Let A, B, and C be matrices of the appropriate sizes so that products and sums below are defined, and let λ be a scalar.

(i) $A(BC) = (AB)C$

(ii) $A(B + C) = AB + AC$

(iii) $(A + B)C = AC + BC$

(iv) $\lambda(AB) = (\lambda A)B = A(\lambda B)$

Proof.
This is another theorem that is easy, but tedious, to verify just by writing out what the matrices look like in components. We will give the details for the second property, however.

Suppose that A is $m \times n$, B is $n \times p$, and C is $n \times p$, so that the sums $B+C$ and $AB+AC$ and the products $A(B+C)$, AB and AC are all defined. Let a_{ij} denote the entry in the i-th row and j-th column of A, and likewise the entries of B and C are b_{ij} and c_{ij}.

Notice that the entry in the i-th row and j-th column of AB is

$$\sum_{k=1}^{n} a_{ik}b_{kj}$$

and similarly, the entry in the i-th row and j-th column of AC is

$$\sum_{k=1}^{n} a_{ik}c_{kj}$$

Hence the corresponding entry in $AB + AC$ is

$$\sum_{k=1}^{n} a_{ik}b_{kj} + \sum_{k=1}^{n} a_{ik}c_{kj} = \sum_{k=1}^{n} \left(a_{ik}b_{kj} + a_{ik}c_{kj}\right) = \sum_{k=1}^{n} a_{ik}\left(b_{kj} + c_{kj}\right)$$

As the entry in the i-th row and j-th column of $B + C$ is $b_{ij} + c_{ij}$, the entry in the i-th row and j-th column of $A(B + C)$ is thus

$$\sum_{k=1}^{n} a_{ik}\left(b_{kj} + c_{kj}\right).$$

Thus $A(B+C) = AB+AC$ since these matrices have the same entries. □

The Transpose

Given any $m \times n$ matrix A, we can define an $n \times m$ matrix called the *transpose* of A and denoted A^T by swapping the rows and columns of A.

Example 9.55.

$$A = \begin{pmatrix} 1 & 2 & 7 & -3 & 2 & 4 \\ 4 & -2 & 1 & 1 & 0 & 2 \\ 3 & 1 & 2 & 2 & 2 & 2 \end{pmatrix}$$

$$A^T = \begin{pmatrix} 1 & 4 & 3 \\ 2 & -2 & 1 \\ 7 & 1 & 2 \\ -3 & 1 & 2 \\ 2 & 0 & 2 \\ 4 & 2 & 2 \end{pmatrix}$$

Notice that this operation turns row vectors into column vectors and vice versa:

Example 9.56.

$$\begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}^T = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

$$\begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \end{pmatrix}^T = \begin{pmatrix} 5 & 6 & 7 & 8 \end{pmatrix}.$$

Remark.
If $\vec{v}$ is a column vector (an $n \times 1$ matrix), then the transpose $\vec{v}^T$ is a row vector (a $1 \times n$ matrix). The dot product of two vectors $\vec{u}$ and $\vec{v}$ can then be written as $\vec{u}^T\vec{v}$ where we perform matrix multiplication on the $1 \times n$ and $n \times 1$ vector to get a single number.

As the transpose is defined by exchanging the roles of columns and rows, if $\vec{a}_i$ is the i-th column of A, then $\vec{a}_i^T$ is the i-th row of A^T. Similarly, if $\vec{\alpha}_j$ is the j-th row of A (here $\vec{\alpha}_j$ is a row vector), then $\vec{\alpha}_j^T$ (now a column vector) is the j-th column of A.

Theorem 9.12.
If A is any matrix, then $(A^T)^T = A$.

Proof.
Suppose the columns of A are $\vec{a}_1$, $\vec{a}_2$, ..., $\vec{a}_n$:

$$A = \begin{pmatrix} \vec{a}_1 & \vec{a}_2 & \cdots & \vec{a}_n \end{pmatrix}$$

Then the transposes of those columns give the rows of A^T:

$$A^T = \begin{pmatrix} \vec{a}_1^T \\ \vec{a}_2^T \\ \vdots \\ \vec{a}_n^T \end{pmatrix}.$$

If we take the transpose yet again, then we turn these rows back into the original columns of A:

$$(A^T)^T = \begin{pmatrix} \vec{a}_1 & \vec{a}_2 & \cdots & \vec{a}_n \end{pmatrix} = A$$

□

Remark.
Notice that if $T : \mathbb{R}^n \to \mathbb{R}^m$ is any linear transformation, say with corresponding matrix A, then the transpose of A determines a linear transformation $\mathbb{R}^m \to \mathbb{R}^n$ (in the reverse order of the original transformation T).

We now have four different operations we can perform on matrices: matrix addition, scalar multiplication, matrix multiplication, and now the transpose. It's reasonable to ask how our new operation, transpose, gets along with the previous operations.

Theorem 9.13.
Let A and B be matrices of the appropriate sizes so that the operations below are defined, and let λ be a scalar. We then have the following:

(i) $(A+B)^T = A^T + B^T$

(ii) $(\lambda A)^T = \lambda(A^T)$

(iii) $(AB)^T = B^T A^T$.

Proof of Theorem 9.13 (i).
We will only prove part (i) and leave the proofs of the other properties as exercises.

Suppose that A and B are both $m \times n$ matrices so that their sum is defined. Suppose that a_{ij} is the entry in the i-th row, j-th column of A and b_{ij} is the entry in the i-th row, j-th column of B. For simplicity let's refer to $A+B$ as C and say $c_{ij} = a_{ij} + b_{ij}$ is the entry in the i-th row, j-th column of $C = A + B$.

Notice that since the transpose reverses the roles of rows and columns, the entry in the i-th row, j-th column of A^T is a_{ji} (note i and j are in the reverse order), and similarly for B^T and C^T. Thus the i-th row, j-th column of C has entry c_{ji} which by definition is $a_{ji} + b_{ji}$, but this is the sum of what's in the i-th row and j-th column of A^T and B^T. Thus $C^T = A^T + B^T$. □

Exercise 9.4.
Prove parts (ii) and (iii) of Theorem 9.13. Notice that the i-th row, j-th column of λA is λa_{ij}, and so the entry in the i-th row, j-th column of $(\lambda A)^T$ is λa_{ji}, but this is exactly the entry in the i-th row, j-th column of $\lambda(A^T)$, and so $(\lambda A)^T = \lambda(A^T)$.

Suppose A is $m \times n$ and B is $n \times p$. The i-th row, j-th column of $(AB)^T$ is the same as the j-th row, i-th column of AB which is

$$\sum_{k=1}^{n} a_{jk} b_{ki}.$$

The i-th row, j-th column of $B^T A^T$ is

$$\sum_{k=1}^{n} b_{ki} a_{jk}.$$

Since $a_{jk} b_{ki} = b_{ki} a_{jk}$, the matrices are the same.

9.8 Inverses

If $f : A \to B$ is a map, any map $g : B \to A$ which satisfies

$$g(f(a)) = a \text{ for every } a \in A \tag{9.1}$$
$$f(g(b)) = b \text{ for every } b \in B \tag{9.2}$$

is called an ***inverse*** of f. Two important properties of inverses are given by the following theorems:

Theorem 9.14.
A map $f : A \to B$ has an inverse if and only if f is a bijection.

Proof.
Suppose that f has an inverse: we suppose there is some map $g : B \to A$ satisfying the two equations above, and we need to show that f must be both surjective and injective. For surjectivity, let $b \in B$ and notice that there exists some element $a \in A$ that f sends to b: namely, take $a = g(b)$. By the second equation in the definition of an inverse we then have

$$f(a) = f(g(b)) = b$$

and so f is surjective.

For injectivity, suppose that there are elements $a, a' \in A$ such that $f(a) = f(a')$. If we then apply g to $f(a)$ and $f(a')$, however, we have

$$a = g(f(a)) = g(f(a')) = a',$$

and so f is injective.

Now we show the converse: suppose that f is bijective, and we want to show that f has an inverse. We define a map $g : B \to A$ by declaring that for each $b \in B$, $g(b) = a$ where $a \in A$ is the element that f sends to b. Such an a must exist since f is surjective, and a is unique because f is injective. So g is a well-defined map. Now we just need to verify that $f \circ g$ and $g \circ f$ satisfy the defining properties of an inverse, but this is almost obvious because of the way we defined g. By definition, $g(f(a))$ is the element in A which f sends to $f(a)$, but that is simply a and so $g(f(a)) = a$. For the second equation, notice that $g(b)$ is the element of A that f sends to b, so $f(g(b)) = b$. □

Theorem 9.15.
If f has an inverse (i.e., if f is bijective), then its inverse is unique.

That is, there is only one map $g : B \to A$ satisfying Equations (9.1) and (9.2).

Proof.
To see this, suppose there were two different maps, say g_1 and g_2, satisfying the equations. We will show that g_1 and g_2 must in fact be the same map. Notice that

$$f(g_1(b)) = b = f(g_2(b)),$$

but f is injective so $g_1(b) = g_2(b)$. □

Since inverses are unique, we are justified in saying *the* inverse of a map instead of *an* inverse of a map. We adopt the notation f^{-1} to denote the inverse of f. Notice that this *is not* f raised to the negative first power; this is not one over f. (In fact, since we're just talking about sets that don't necessarily have a notation of any sort of "arithmetic" with their elements, this is a non-issue.)

Notice that Equations (9.1) and (9.2) imply the following:

$$\begin{aligned} f(a) = b &\implies f^{-1}(b) = a \\ f^{-1}(b) = a &\implies f(a) = b. \end{aligned}$$

To see this, simply apply f^{-1} to both sides of $f(a) = b$ and then use Equation (9.1); and similarly apply f to both sides of $f^{-1}(b) = a$ and use Equation (9.2). This can be stated more simply as $f(a) = b$ if and only if $f^{-1}(b) = a$, which we can write symbolically as $f(a) = b \iff f^{-1}(b) = a$.

We can simplify Equations (9.1) and (9.2) by introducing the ***identity map***. For every set A, the identity map is a function from A to itself which fixes every element: that is, $a \mapsto a$ for every $a \in A$. The identity map is denoted id_A or id if the set A is clear from context.

Lemma 9.16.
Given any map $f : A \to B$, composing f with the identity map does

not change f:

$$f \circ id_A = f = id_B \circ f.$$

Proof.
For every $a \in A$,

$$f(\mathrm{id}_A(a)) = f(a), \text{ and}$$
$$\mathrm{id}_B(f(a)) = f(a)$$

□

Equations (9.1) and (9.2) can then be expressed more tersely as

$$f \circ f^{-1} = f^{-1} \circ f = \mathrm{id}.$$

Inverse of a Matrix

Now suppose that $T : \mathbb{R}^n \to \mathbb{R}^n$ is a bijective linear map (i.e., the columns of the corresponding $n \times n$ matrix are linearly independent and span $\mathbb{R}^n$), and so has some inverse T^{-1}.

Lemma 9.17.
If $T : \mathbb{R}^n \to \mathbb{R}^n$ is a bijective linear transformation, then its inverse T^{-1} is also linear.

Proof.
By definition, $T^{-1}(\vec{v_1}) = \vec{u_1}$ if $T(\vec{u_1}) = \vec{v_1}$. Also consider vectors $\vec{v_2}$ and $\vec{u_2}$ with $T^{-1}(\vec{v_2}) = \vec{u_2} \iff T(\vec{u_2}) = \vec{v_2}$. We need to show that $T^{-1}(\vec{v_1} + \vec{v_2}) = \vec{u_1} + \vec{u_2}$, but notice

$$T(\vec{u_1} + \vec{u_2}) = T(\vec{u_1}) + T(\vec{u_2}) = \vec{v_1} + \vec{v_2}.$$

Since T^{-1} is the inverse of T, $T^{-1}(\vec{v_1}+\vec{v_2})$ is the element of $\mathbb{R}^n$ which T takes to $\vec{v_1}+\vec{v_2}$, but we have just shown that $\vec{u_1}+\vec{u_2}$ is that element, and so

$$T^{-1}(\vec{v_1}+\vec{v_2}) = T^{-1}(\vec{v_1}) + T^{-1}(\vec{v_2}).$$

Similarly, suppose $T^{-1}(\vec{v}) = \vec{u}$. We need to show that $T^{-1}(\lambda\vec{v}) = \lambda\vec{u}$, but this must be the case as

$$T(\lambda\vec{u}) = \lambda T(\vec{u}) = \lambda\vec{v}.$$

Thus T^{-1} is linear. □

So if T is a linear bijection, then so is its inverse T^{-1}. Thus there is some matrix that corresponds to T^{-1}. To figure out what this matrix is, let's consider some properties of this matrix. First we need to know about the identity transformation and identity transformation.

The ***identity transformation*** is a map $\mathrm{id} : \mathbb{R}^n \to \mathbb{R}^n$ which leaves vectors alone. That is, $\mathrm{id}(\vec{v}) = \vec{v}$. Notice that if $T : \mathbb{R}^n \to \mathbb{R}^n$ is a linear bijection, then Equations (9.1) and (9.2) can be rewritten as

$$T \circ T^{-1} = \mathrm{id} = T^{-1} \circ T.$$

Since id is a composition of linear maps, id is linear. (It's also very easy to check that id is a linear transformation directly.)

The matrix of the identity transformation is called the ***identity matrix*** and is denoted by I:

$$I = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 & 0 \\ & & & \ddots & & & \\ 0 & 0 & 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix}$$

That is, the identity matrix for $\mathbb{R}^n$ is a square, $n \times n$ matrix that has 1's on the diagonal, and zeros everywhere else.

Sometimes we will write I_n to mean the $n \times n$ identity matrix, and

sometimes we will just write I if the dimension is clear from context.

$$I_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Just as composing a map with the identity map doesn't change the map, multiplying a matrix with the identity matrix doesn't change the matrix:

$$AI = A = IA.$$

Say $T : \mathbb{R}^n \to \mathbb{R}^n$ is a bijective linear transformation with corresponding matrix A, and let A^{-1} denote the matrix of T. Since $T \circ T^{-1} = T^{-1} \circ T =$ id and since matrix multiplication corresponds to composition of linear transformations, we know

$$AA^{-1} = A^{-1}A = I.$$

Our goal is to determine what A^{-1} is given A. To do this we introduce elementary matrices.

Elementary Matrices

An ***elementary matrix*** is a matrix produced by performing an elementary row operation to the identity matrix. That is, an elementary matrix is a square matrix which is given by taking the identity matrix I and performing one of the following operations to it:

(i) Swap two rows.

(ii) Add a multiple of one row to another.

(iii) Multiply everything in one row by a constant.

Example 9.57.

The following are some 3×3 elementary matrices.

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} -4 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Theorem 9.18.
If E is an elementary matrix, then the product EA is the same as performing the corresponding elementary row operation on A.

Before proving Theorem 9.18, let's make an observation. Suppose that A is any $m \times n$ matrix and B is any $n \times p$ matrix. Then the rows of AB are linear combinations of the rows of B where the scalars each row is multiplied by are determined by the entries in each row of A. For example, if the rows of B are the row vectors $\vec{r}_1$, $\vec{r}_2$, ..., $\vec{r}_n$ and the k-th row of A has the form

$$\begin{pmatrix} \lambda_1 & \lambda_2 & \cdots & \lambda_n \end{pmatrix},$$

then the k-th row of AB will be

$$\lambda_1 \vec{r}_1 + \lambda_2 \vec{r}_2 + \cdots + \lambda_n \vec{r}_n.$$

With this fact in hand, we can easily prove Theorem 9.18.

Proof of Theorem 9.18.
Let A be any $m \times n$ matrix whose rows we will suppose are the row vectors $\vec{r}_1$, $\vec{r}_2$, ..., $\vec{r}_m$. Let E be an elementary $m \times m$ matrix.

There are three cases to consider corresponding to the three elementary row operations.

- Suppose E is obtained from I_m by swapping two rows, say rows i and j:

  $$I \xrightarrow{R_i \leftrightarrow R_j} E.$$

  This means that the i-th row of E is $\vec{e}_j^{\,T}$, the j-th row of E is $\vec{e}_i^{\,T}$, and for any k that is not i or j, the k-th row of E is $\vec{e}_k^{\,T}$.

If k is not i or j, then the k-th row of EA is

$$0 \cdot \vec{r}_1 + 0 \cdot \vec{r}_2 + \cdots + 0 \cdot \vec{r}_{k-1} + 1 \cdot \vec{r}_k + 0 \cdot \vec{r}_{k+1} + \cdots + 0 \cdot \vec{r}_m = \vec{r}_k$$

So the k-th row of A remains the same. The i-th row of EA, however is

$$0 \cdot \vec{r}_1 + 0 \cdot \vec{r}_2 + \cdots + 0 \cdot \vec{r}_{j-1} + 1 \cdot \vec{r}_j + 0 \cdot \vec{r}_{j+1} + \cdots + 0 \cdot \vec{r}_m = \vec{r}_j$$

and the k-th row of EA is

$$0 \cdot \vec{r}_1 + 0 \cdot \vec{r}_2 + \cdots + 0 \cdot \vec{r}_{i-1} + 1 \cdot \vec{r}_i + 0 \cdot \vec{r}_{i+1} + \cdots + 0 \cdot \vec{r}_m = \vec{r}_i.$$

That is, every row of A is unchanged except for the i-th and j-th rows which are swapped.

- Suppose E is obtained from I_m by adding c times the i-th row to the j-th row. Then all rows of E, except for the j-th row, are all zeros except for a 1 in the k-th position of the k-th row. Thus every row of EA, except the j-th row, is the same as the corresponding row of A. The j-th row of E is all zeros except for a 1 in the j-th position and a c in the i-th position. Hence the j-th row of EA is the j-th row of A plus c times the i-th row of A.
- Left as an exercise.

□

Exercise 9.5.
Suppose that E is obtained from I by multiplying the i-th row of I by c. Show that EA is obtained from A by multiplying the i-th row of A by c. For $\ell \neq i$, notice that row ℓ of E is all zeros except for a 1 in the ℓ-th position, thus in row ℓ column j of the matrix EA we have

$$\sum_{k=1}^{n} e_{\ell k} a_{kj} = a_{\ell j}$$

since $e_{\ell k} = 0$ for all k except $e_{\ell\ell} = 1$.

Similarly, in row i the i-th row, j-th column of EA is

$$\sum_{k=1}^{n} e_{ik} a_{kj} = c a_{ij}$$

since $e_{ik} = 0$ for all k except $e_{ii} = c$.

The above theorem about elementary row operations will be combined with the following observation to obtain a method for determining inverse matrices.

Lemma 9.19.
If $T : \mathbb{R}^n \to \mathbb{R}^n$ is a linear bijection, then the corresponding matrix A becomes the identity matrix when put into RREF.

Proof.
Since T is surjective, its matrix has a pivot in every row. However, since T is injective, it also has a pivot in every column. The only matrix in RREF with pivots in every row and column is the identity matrix. □

This lemma tells us that there is some sequence of elementary row operations that we can perform to A to get the identity matrix I. Each of these elementary row operations corresponds to multiplication by some elementary matrix. So there is some collection of elementary matrices, E_1, E_2, E_3, ..., E_q such that

$$E_q E_{q-1} \cdots E_2 E_1 A = I$$

The product $E_q E_{q-1} \cdots E_2 E_1$ is thus A^{-1}:

$$A^{-1} = E_q E_{q-1} \cdots E_2 E_1.$$

Example 9.58.
Let A be the following 2×2 matrix

$$\begin{pmatrix} 2 & 4 \\ 1 & 6 \end{pmatrix}$$

We can put this matrix into RREF with the following sequence of elementary row operations:

$$\begin{pmatrix} 2 & 4 \\ 1 & 6 \end{pmatrix} \xrightarrow{\frac{1}{2}R_1 \to R_1} \begin{pmatrix} 1 & 2 \\ 1 & 6 \end{pmatrix}$$
$$\xrightarrow{R_2 - R_1 \to R_2} \begin{pmatrix} 1 & 2 \\ 0 & 4 \end{pmatrix}$$
$$\xrightarrow{\frac{1}{4}R_2 \to R_2} \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$$
$$\xrightarrow{R_1 - 2R_2 \to R_1} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

This corresponds to multiplying A by the following elementary matrices:

$$E_1 = \begin{pmatrix} 1/2 & 0 \\ 0 & 1 \end{pmatrix}$$
$$E_2 = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$$
$$E_3 = \begin{pmatrix} 1 & 0 \\ 0 & 1/4 \end{pmatrix}$$
$$E_4 = \begin{pmatrix} 1 & -2 \\ 0 & 1 \end{pmatrix}$$

Now we multiply $E_4E_3E_2E_1$ to get A^{-1}:

$$A^{-1} = E_4E_3E_2E_1 = \begin{pmatrix} 3/4 & -1/2 \\ -1/8 & 1/4 \end{pmatrix}.$$

We can easily check that this really is the inverse of A: i.e., that

$A^{-1}A = I$:

$$\begin{aligned} A^{-1}A &= \begin{pmatrix} 3/4 & -1/2 \\ -1/8 & 1/4 \end{pmatrix} \begin{pmatrix} 2 & 4 \\ 1 & 6 \end{pmatrix} \\ &= \begin{pmatrix} 3/4 \cdot 2 + (-1/2) \cdot 1 & 3/4 \cdot 4 + (-1/2) \cdot 6 \\ -1/8 \cdot 2 + 1/4 \cdot 1 & -1/8 \cdot 4 + 1/4 \cdot 6 \end{pmatrix} \\ &= \begin{pmatrix} 3/2 - 1/2 & 3 - 3 \\ -1/4 + 1/4 & -1/2 + 3/2 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{aligned}$$

There is a little trick we can use to make obtaining A^{-1} slightly easier. Suppose that A is $n \times n$ and consider the $n \times 2n$ matrix obtained by augmenting A with the $n \times n$ identity matrix:

$$\left(A \mid I\right).$$

We then start performing the elementary row operations that put A into RREF (this is the same as multiplying by E_1, then E_2, then E_3 and so on). Eventually, through some sequence of elementary row operations the above matrix becomes

$$\left(I \mid A^{-1}\right)$$

Example 9.59.
We compute the inverse of the matrix

$$A = \begin{pmatrix} 2 & 4 \\ 1 & 6 \end{pmatrix}$$

from the previous example using this trick.

$$
\begin{aligned}
&\left(\begin{array}{cc|cc} A & & I & \end{array}\right) \\
&= \left(\begin{array}{cc|cc} 2 & 4 & 1 & 0 \\ 1 & 6 & 0 & 1 \end{array}\right) \\
&\xrightarrow{\frac{1}{2}R_1 \to R_1} \left(\begin{array}{cc|cc} 1 & 2 & 1/2 & 0 \\ 1 & 6 & 0 & 1 \end{array}\right) \\
&\xrightarrow{R_2 - R_1 \to R_2} \left(\begin{array}{cc|cc} 1 & 2 & 1/2 & 0 \\ 0 & 4 & -1/2 & 1 \end{array}\right) \\
&\xrightarrow{\frac{1}{4}R_2 \to R_2} \left(\begin{array}{cc|cc} 1 & 2 & 1/2 & 0 \\ 0 & 1 & -1/8 & 1/4 \end{array}\right) \\
&\xrightarrow{R_1 - 2R_2 \to R_1} \left(\begin{array}{cc|cc} 1 & 0 & 3/4 & -1/2 \\ 0 & 1 & -1/8 & 1/4 \end{array}\right)
\end{aligned}
$$

The inverse matrix A^{-1} is now the right-hand side of this augmented matrix,

$$A^{-1} = \begin{pmatrix} 3/4 & -1/2 \\ -1/8 & 1/4 \end{pmatrix},$$

which agrees with our previous calculation.

This trick for computing the inverse works for any square matrix which is invertible, regardless of the size, though the work certainly gets more tedious as we consider larger and larger matrices.

Example 9.60.
Compute the inverse of the following matrix:

$$A = \begin{pmatrix} -2/3 & 1 & 0 \\ 1 & -1 & 0 \\ 4 & -6 & 1 \end{pmatrix}$$

$$
\begin{aligned}
\left(A \mid I\right) &\xrightarrow{R_1 \leftrightarrow R_2} \left(\begin{array}{ccc|ccc} 1 & -1 & 0 & 0 & 1 & 0 \\ -2/3 & 1 & 0 & 1 & 0 & 0 \\ 4 & -6 & 1 & 0 & 0 & 1 \end{array}\right) \\
&\xrightarrow{R_2 + \frac{2}{3}R_1 \to R_2} \left(\begin{array}{ccc|ccc} 1 & -1 & 0 & 0 & 1 & 0 \\ 0 & 1/3 & 0 & 1 & 2/3 & 0 \\ 4 & -6 & 1 & 0 & 0 & 1 \end{array}\right) \\
&\xrightarrow{R_3 - 4R_1 \to R_3} \left(\begin{array}{ccc|ccc} 1 & -1 & 0 & 0 & 1 & 0 \\ 0 & 1/3 & 0 & 1 & 2/3 & 0 \\ 0 & -2 & 1 & 0 & -4 & 1 \end{array}\right) \\
&\xrightarrow{3R_2 \to R_2} \left(\begin{array}{ccc|ccc} 1 & -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 3 & 2 & 0 \\ 0 & -2 & 1 & 0 & -4 & 1 \end{array}\right) \\
&\xrightarrow{R_1 + R_2 \to R_1} \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 3 & 3 & 0 \\ 0 & 1 & 0 & 3 & 2 & 0 \\ 0 & -2 & 1 & 0 & -4 & 1 \end{array}\right) \\
&\xrightarrow{R_3 + 2R_2 \to R_3} \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 3 & 3 & 0 \\ 0 & 1 & 0 & 3 & 2 & 0 \\ 0 & 0 & 1 & 6 & 0 & 1 \end{array}\right)
\end{aligned}
$$

Hence

$$A^{-1} = \begin{pmatrix} 3 & 3 & 0 \\ 3 & 2 & 0 \\ 6 & 0 & 1 \end{pmatrix}$$

Notice that this only tells us A^{-1} if the RREF of A is the identity matrix. If the RREF of A *is not* the identity matrix, then A does not have an inverse.

In the special case of 2×2 matrices there is a quick and easy formula for the derivative.

Theorem 9.20.
If A is a 2×2 matrix of the form

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix},$$

then A is invertible if and only if $ad - bc \neq 0$, and the inverse of A is

$$A^{-1} = \frac{1}{ad-bc}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}.$$

Proof.
First suppose that A is invertible, so the RREF of A is the identity matrix. We then proceed to calculate A^{-1} as above:

$$\begin{pmatrix} a & b & 1 & 0 \\ c & d & 0 & 1 \end{pmatrix}$$

$$\xrightarrow{\frac{1}{a}R_1 \to R_1} \begin{pmatrix} 1 & b/a & 1/a & 0 \\ c & d & 0 & 1 \end{pmatrix}$$

$$\xrightarrow{R_2 - cR_1 \to R_2} \begin{pmatrix} 1 & b/a & 1/a & 0 \\ 0 & (ad-bc)/a & -c/a & 1 \end{pmatrix}$$

$$\xrightarrow{\frac{a}{ad-bc}R_2 \to R_2} \begin{pmatrix} 1 & b/a & 1/a & 0 \\ 0 & 1 & -c/(ad-bc) & a/(ad-bc) \end{pmatrix}$$

$$\xrightarrow{R_1 - \frac{b}{a}R_2 \to R_1} \begin{pmatrix} 1 & 0 & 1/a + bc/a(ad-bc) & -b/(ad-bc) \\ 0 & 1 & -c/(ad-bc) & a/(ad-bc) \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & (ad-bc+bc)/a(ad-bc) & -b/a \\ 0 & 1 & -c/(ad-c) & a/(ad-bc) \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & d/(ad-bc) & -b/a \\ 0 & 1 & -c/(ad-bc) & a/(ad-bc) \end{pmatrix}$$

Thus

$$A^{-1} = \begin{pmatrix} d/(ad-bc) & -b/a \\ -c/(ad-bc) & a/(ad-bc) \end{pmatrix} = \frac{1}{ad-bc}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

This of course implies that $ad - bc \neq 0$ since if it were the above expression for A^{-1} would be undefined.

Now suppose that $ad - bc \neq 0$. The calculation above shows us that the RREF of A is the identity, and further tells us what A^{-1} is, so A must be invertible. □

Properties of Inverses

Now that we know how to compute the inverse of a matrix, we turn our attention to some properties that inverses satisfy.

Lemma 9.21.
If A is an invertible matrix and if B and C are matrices satisfying that $AB = AC$, then $B = C$.

Proof.

$$\begin{aligned} & AB = AC \\ \implies & A^{-1}AB = A^{-1}AC \\ \implies & IB = IC \\ \implies & B = C \end{aligned}$$

□

As we saw in the last section, the above lemma is false if A is not invertible.

Lemma 9.22.
If A and B are invertible $n \times n$ matrices and if $AB = BA = I$, then $A^{-1} = B$ and $B^{-1} = A$.

Proof.
We are assuming $AB = I$. Multiplying both sides on the right by B^{-1} gives $ABB^{-1} = IB^{-1}$ but since $BB^{-1} = I$ this simplifies to $A = B^{-1}$.

Similarly if we were to multiply both sides on the left by A^{-1} we have $A^{-1}AB = A^{-1}I$ which simplifies to $B = A^{-1}$. □

Theorem 9.23.

(i) If A is an invertible matrix, then so is A^{-1} and $(A^{-1})^{-1} = A$.

(ii) If A and B are both invertible matrices of the same size, then their product AB is invertible and $(AB)^{-1} = B^{-1}A^{-1}$.

(iii) If A is an invertible matrix, then so is A^T and $\left(A^T\right)^{-1} = (A^{-1})^T$.

Proof.

(i) For the moment let C denote the matrix $(A^{-1})^{-1}$. Then $A^{-1}C = CA^{-1} = I$. But we know that $A^{-1}A = AA^{-1} = I$ and so by Lemma 9.22, $C = A$.

(ii) Simply notice that

$$\begin{aligned} B^{-1}A^{-1}AB &= B^{-1}(A^{-1}A)B \\ &= B^{-1}IB \\ &= B^{-1}B \\ &= I \end{aligned}$$

(iii) Recall that $(AB)^T = B^TA^T$ and notice that $I^T = I$

$$\begin{aligned} & AA^{-1} = I \\ \implies & \left(AA^{-1}\right)^T = I^T = I \\ \implies & \left(A^{-1}\right)^T A^T = I \end{aligned}$$

but this implies that the inverse of A^T is $(A^{-1})^T$.

□

The following theorem tells us there are several different ways to think about invertible matrices. Some of the items in the theorem we have already seen, but we list them in this theorem so that we will have a single theorem to refer to that characterizes invertible matrices.

Theorem 9.24.
Let A be an $n \times n$ square matrix. The following are equivalent:

(a) A is an invertible matrix.

(b) The RREF of A is the identity matrix.

(c) A has n pivots.

(d) A has a pivot in every row.

(e) A has a pivot in every column.

(f) The only solution to the homogeneous equation $A\vec{x} = \vec{0}$ is the trivial solution.

(g) The columns of A are linearly independent.

(h) The columns of A span $\mathbb{R}^n$.

(i) The equation $A\vec{x} = \vec{b}$ has one solution for every $b \in \mathbb{R}^n$.

(j) The linear transformation $\vec{x} \mapsto A\vec{x}$ is injective.

(k) The linear transformation $\vec{x} \mapsto A\vec{x}$ is surjective.

(l) There is an $n \times n$ matrix B so that $AB = I$.

(m) There is an $n \times n$ matrix B so that $BA = I$.

(n) A^T is invertible.

10 Linear Algebra in Matlab

Mathematics is not about numbers, equations, computations, or algorithms: it is about understanding.

William Thurston

Matlab has built-in commands for all of the basic operations of linear algebra making it very easy to perform linear algebra in Matlab.

10.1 Vectors and matrices

We have already seen how to create vectors in Matlab: we simply have a pair of square brackets with a comma-delimited list of numbers inbetween the brackets, corresponding to the components of the vector. This actually creates a ***row vector***, but sometimes it is necessary or helpful to have a ***column vector***. In Matlab this is created in the same way as the row vectors we have seen before, except our components are separated by semi-colons instead of commas.

```
>> rowVector = [1, 2, 3, 4]

rowVector =

     1     2     3     4

>> columnVector = [1; 2; 3; 4]

columnVector =

     1
     2
     3
     4
```

Recall that elements of a vector in Matlab are accessed by giving the index of the element in parentheses:

```
>> v = [-2, 4, 7, -15];
>> v(3)

ans =

     7
```

The same convention applies for column vectors:

```
>> u = [9; 3; 14; 2];
>> u(3)

ans =

    14
```

Matrices are created in Matlab by combining these two conventions. That is, we create a matrix in Matlab by using a pair of square brackets, and inbetween the square brackets we list the entries of the matrix along the rows, with rows separated by semicolons. For example, the 3×4 matrix M below,

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

is created in Matlab with the following command:

```
>> M = [1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12]

M =
```

```
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

We access individual elements of a matrix by giving the row and column of the element in parenthesis after the matrix. For example, in the matrix `M` above, `M(2, 1)` refers to the element in the second row and first column.

```
>> M(2, 1)

ans =

     5
```

We can also retrieve an entire row or column of a matrix as a row or column vector. We do this by specifying the row (or column) we are interested in, and placing a `:` in the column (or row) index. For instance, the third row of matrix `M` is accessed by `M(3,:)` whereas the second column is accessed by `M(:,2)`.

```
>> M(3,:)

ans =

     9    10    11    12

>> M(:,2)

ans =

     2
     6
    10
```

There are times when we may wish to create a matrix of all zeros or all ones. In some problems, for example, we may know we will ultimately need a matrix of some given size $m \times n$ but may not know what the entries of the matrix will be when we create the matrix. In such a situation we can easily create a matrix of all zeros, or all ones, by using the `zeros` or `ones` functions. These both take the number of rows and columns as arguments and return a matrix of all zeros or all ones of the specified size. For example, the two commands below create a 3×5 matrix of all zeros and a 7×3 matrix of all ones.

```
>> zeros(3, 5)

ans =

     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0

>> ones(7, 3)

ans =

     1     1     1
     1     1     1
     1     1     1
     1     1     1
     1     1     1
     1     1     1
     1     1     1
```

As an example of where this might be useful, suppose we needed to create a matrix where each entry was the product of the row and column of that entry. For example, the entry in row 4 column 3 would be 12 and the matrix in row 2 column 2 would be 4. Writing out the entries of this matrix by hand when we create it will be very tedious for matrices with several rows and several columns, so what we could do is have Matlab create a matrix of all zeros (or all ones) of the right size, then have a loop that walks along the rows and columns of the matrix and updates each entry to

the appropriate value. The code below, for instance, does this for a 7×5 matrix.

```
>> numrows = 7;
>> numcols = 5;
>> M = zeros(numrows, numcols);
>> for i = 1:numrows
       for j=1:numcols
           M(i, j) = i*j;
       end
   end
>> disp(M)
     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
     4     8    12    16    20
     5    10    15    20    25
     6    12    18    24    30
     7    14    21    28    35
```

The $n \times n$ ***identity matrix*** is by definition the $n \times n$ where every entry is zero, except for the entries on the diagonal of the matrix which are ones. This matrix, often denoted I_n or simpl I, is generated in Matlab by calling `eye(n)`:

```
>> eye(5)

ans =

     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

For some operations it is necessary to know the size of a matrix: the number of rows or columns. These can be computed by passing the matrix to the `size` function which returns two values. The first value is the number of rows, and the second is the number of columns of the given matrix. To save both of these values we can assign them to a vector of variables:

```
>> M = [4, 7, 2, 3; 1, 2, 9, 9];
>> [numrows, numcols] = size(M);
>> disp(numrows)
     2

>> disp(numcols)
     4
```

10.2 Arithmetic of matrices and vectors

Matrices and vectors can be added, subtracted, multiplied, or multiplied by scalars, in the ways you would probably guess. Both matrices and vectors can be added together with `+`, provided both operands have the same dimensions.

```
>> M = [1, 2, 3, 4; 2, 4, 8, 6; 3, 6, 9, 12]

M =

     1     2     3     4
     2     4     8     6
     3     6     9    12

>> N = [1, -2, 3, -5; 7, -11, 13, -17; 19, -23, 29, -31]

N =

     1    -2     3    -5
```

```
     7   -11    13   -17
    19   -23    29   -31

>> M + N

ans =

     2     0     6    -1
     9    -7    21   -11
    22   -17    38   -19

>> u = [1; 2; 3]

u =

     1
     2
     3

>> v = [4; 5; 6]

v =

     4
     5
     6

>> u + v

ans =

     5
     7
     9
```

Scalar multiplication is accomplished by using the operator * where operand is a scalar (a number) and the other is any matrix or vector.

```
>> 3 * M

ans =

     3     6     9    12
     6    12    24    18
     9    18    27    36

>> 2 * v

ans =

     8
    10
    12
```

Using * computes the product of two matrices, or of a matrix and a vector:

```
>> M = [4, 3, 7, 2; 0, 1, -1, 3; 1, 2, 3, 4]

M =

     4     3     7     2
     0     1    -1     3
     1     2     3     4

>> N = [-2, 9; 1, 1; 5, -4; 2, 6]

N =

    -2     9
     1     1
     5    -4
     2     6

>> M * N
```

```
ans =

    34    23
     2    23
    23    23

>> v = [1; 2; 3; 4]

v =

     1
     2
     3
     4

>> M * v

ans =

    39
    11
    30
```

Note that Matlab is picky about the dimensions of the matrices/vectors you are trying to multiply. Above we multiplied the 3×4 matrix with the 4-dimensional column vector v, but Matlab will complain if we try to multiply M by a 4-dimensional row vector:

```
>> u = [1, 2, 3, 4]

u =

     1     2     3     4

>> M*u
```

```
Error using  *
Inner matrix dimensions must agree.
```

Notice that we can interpret an m-dimensional row vector as a $1 \times m$ matrix, and similarly an m-dimensional column vector as an $m \times 1$ matrix. The product of two such things should be a 1×1 matrix, which is simply a number.

```
>> u = [4, 7, 1];
>> v = [1; 2; 3];
>> u * v

ans =

    21
```

Remark.
If you've taken multivariable calculus, you may recognize that this is the same as the dot product of u and v.

Notice too that the product of an m-dimensional column vector (aka, $m \times 1$ matrix) and n-dimensional row vector (aka, $1 \times n$ matrix) is an $m \times n$ matrix:

```
>> u = [1; 3; 2];
>> v = [4, 3, 2, 1];
>> u * v

ans =

     4     3     2     1
    12     9     6     3
```

```
     8     6     4     2
```

Notice, however, there is no sense in which squaring a vector (i.e., multiplying a vector with itself) makes sense. If you were to try to multiply a row vector with itself, then you'd be trying to multiply a $1 \times m$ matrix with a $1 \times m$ matrix, but this isn't defined; likewise for multiplying a column vector with itself. We can make the dimensions agree by taking the ***transpose*** of one of the factors to convert it from a row vector to a column vector, or vice versa. This is accomplished in Matlab with the `transpose` function.

```
>> v = [1; 2; 3]

v =

     1
     2
     3

>> transpose(v)

ans =

     1     2     3

>> transpose(v) * v

ans =

    14
```

Remark.
This is an alternative way of thinking about the dot product of two vectors: if u and v are column vectors, then their dot product can be

computed as $u \cdot v = u^T v$.

The transpose of a matrix is also accomplished with the `transpose` function. In this case an $m \times n$ matrix is turned into an $n \times m$ matrix where the i-th row of the original matrix becomes the i-th column of the transpose.

```
>> M = [1, 2, 3, 4; 2, 4, 8, 6; 3, 6, 9, 12]

M =

     1     2     3     4
     2     4     8     6
     3     6     9    12

>> transpose(M)

ans =

     1     2     3
     2     4     6
     3     8     9
     4     6    12
```

The `transpose` function can also be used by putting a single apostrophe after the matrix or vector you wish to tranpose:

```
>> v = [1, 2, 3]

v =

     1     2     3

>> v'
```

```
ans =

     1
     2
     3

>> M = [1, 2, 3; 4, 5, 6]

M =

     1     2     3
     4     5     6

>> M'

ans =

     1     4
     2     5
     3     6
```

10.3 Submatrices

In some types of problems it is helpful to consider a submatrix of a given matrix. That is, we take a matrix and remove some of its rows and/or columns to obtain a new matrix. For example, consider the matrix

$$\begin{pmatrix} 1 & 1 & 0 \\ 2 & 3 & 1 \\ 3 & 4 & 2 \end{pmatrix}$$

Now imagine that we deleted the third row and the second column from this matrix, leaving only the entries in the first and second rows from the first and third columns. This would give us the matrix

$$\begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}$$

Creating this submatrix is extremely easy in Matlab. Recall that we can access elements of a matrix `M` with `M(row, col)`. We can access a submatrix

consisting of the entries in rows `row1`, `row2`, ..., `rowm` and columns `col1`, `col2`, ..., `coln` by giving a vector of these rows and a vector of these columns:

```
M([row1, row2, ..., rowm], [col1, col2, ..., coln])
```

For example, the submatrix consisting of rows one and two and columns one and three is given by `M([1, 2], [1, 3])`:

```
>> M = [1, 1, 0; 2, 3, 1; 3, 4, 2]

M =

     1     1     0
     2     3     1
     3     4     2

>> M([1, 2], [1, 3])

ans =

     1     0
     2     1
```

If we want to include all rows or all columns we can use the colon in place of the vector of rows or columns. The submatrices consisting of rows one and two (and all columns), or of columns one and three (and all rows), for instance, are given by `M([1, 2], :)` and `M(:, [1, 3])`, respectively:

```
>> M([1, 2], :)

ans =

     1     1     0
     2     3     1

>> M(:, [1, 3])
```

```
ans =

     1     0
     2     1
     3     2
```

Though this syntax might seem a little strange at first, it can be extremely useful. In particular, it makes the elementary row operations very easy to write in Matlab, as we will see soon.

10.4 Systems, elementary row operations, and inverses

Recall that we introduced matrices as a tool of helping us systematically solve systems of linear equations. We did this by associating to any linear system its augmented coefficient matrix, which contained the coefficient matrix of the system augmented by a vector of "right-hand side" values.

In Matlab we can augment a matrix by a vector by placing the matrix and the vector inside a pair of square brackets separated by a space. For example, if M and v are matrix and vector

$$M = \begin{pmatrix} 1 & 1 & 0 \\ 2 & 3 & 1 \\ 3 & 4 & 2 \end{pmatrix} \qquad v = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

and we want to create the augmented matrix

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 2 & 3 & 1 & 2 \\ 3 & 4 & 2 & 3 \end{pmatrix}$$

in Matlab we would use `[M v]`:

```
>> M = [1, 1, 0; 2, 3, 1; 3, 4, 2]

M =

     1     1     0
```

```
     2     3     1
     3     4     2

>> v = [1; 2; 3]

v =

     1
     2
     3

>> [M v]

ans =

     1     1     0     1
     2     3     1     2
     3     4     2     3
```

Recall that there are three elementary row operations we described in the last chapter:

1. Swapping two rows.
2. Replacing one row by a constant multiple of that row.
3. Replacing one row by that row plus a constant multiple of another row.

In Matlab we can easily accomplish each operation by accessing a row or updating a row with `M(row, :)`.

For example, in the matrix `M` created above, we could swap the first and second rows with

```
M([1, 2], :)  = M([2, 1], :)
```

This might seem like a strange command, so let's slowly decipher what's happening. On the right-hand side we access the elements in the second and first rows of the matrix. Because we place `2` before `1` in `[2, 1]`, the second row is listed before the first row. Recall the colon, `:`, simply means we should access all columns.

```
>> M = [1, 1, 0; 2, 3, 1; 3, 4, 2]

M =

     1     1     0
     2     3     1
     3     4     2

>> M([2, 1], :)

ans =

     2     3     1
     1     1     0
```

Similarly, `M([1, 2], :)` refers to the first two rows of the matrix. We are changing the values of rows 1 and 2 to the values of rows 2 and 1 when we execute `M([1, 2], :) = M([2, 1], :)`.

```
>> M = [1, 1, 0; 2, 3, 1; 3, 4, 2]

M =

     1     1     0
     2     3     1
     3     4     2

>> M([1, 2], :) = M([2, 1], :);
>> disp(M)
     2     3     1
     1     1     0
     3     4     2
```

In general, rows `i` and `j` of matrix `M` are swapped by `M([i, j], :) = M([j, i], :)`.

Similarly, we can multiply all elements in row `i` by some constant `c` with

```
M(i, :)  = c * M(i, :)
```

The lines of code below, for instance, replace the second row of a matrix `M` by -3 times that row.

```
>> M = [1, 1, 0; 2, 3, 1; 3, 4, 2]

M =

     1     1     0
     2     3     1
     3     4     2

>> M(2, :) = -3 * M(2, :);
>> disp(M)
     1     1     0
    -6    -9    -3
     3     4     2
```

Finally, we can add `c` times row `i` to row `j` with

```
M(j, :)  = M(j, :)  + c * M(i, :)
```

The example below adds -2 times the first row to the second row of M:

```
>> M = [1, 1, 0; 2, 3, 1; 3, 4, 2]

M =

     1     1     0
     2     3     1
     3     4     2

>> M(2, :) = M(2, :) + (-2) * M(1, :);
>> M

M =
```

```
     1     1     0
     0     1     1
     3     4     2
```

We can now combine all of the operations above to solve systems of equations in Matlab by entering the agumented coefficient system of the matrix, and performing the necessary elementary row operations to put the matrix into an echelon form.

For example, let's solve the system

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

First we enter the augmented coefficient matrix:

```
>> M = [1, 2, 4, 5;
        2, 4, 5, 4;
        4, 5, 4, 2]

M =

     1     2     4     5
     2     4     5     4
     4     5     4     2
```

Now we subtract two times the second row from the third row:

```
>> M(3, :) = M(3, :) - 2 * M(2, :)

M =

     1     2     4     5
```

```
     2     4     5     4
     0    -3    -6    -6
```

Next, we subtract twice the first row from the second:

```
>> M(2, :) = M(2, :) - 2 * M(1, :)

M =

     1     2     4     5
     0     0    -3    -6
     0    -3    -6    -6
```

Finally, we can swap the second and third rows:

```
>> M([2, 3], :) = M([3, 2], :)

M =

     1     2     4     5
     0    -3    -6    -6
     0     0    -3    -6
```

This new matrix is in an echelon form, and so the system we started with is equivalent to the system

$$\begin{aligned} x + 2y + 4z &= 5 \\ -3y - 6z &= -6 \\ -3z &= -6 \end{aligned}$$

Of course, puting a matrix into an echelon form is such a common procedure that Matlab has a built-in function for putting a matrix into echelon form.

In general, an echelon form of a matrix is not unique: two different people could start with the same matrix, perform different sequences of

elementary row operations and arrive at two different matrices in echelon form. To fix this we can ask fo the row-reduced echelon form of a matrix, which is an echelon form with two additional properties:

- All entries in the matrix above and below the left-most non-zero entry in each row are zero.
- The left-most non-zero entry in each row is one.

The ***row reduced echelon form*** (or ***rref***) of a matrix *is* unique. We can compute the rref of a matrx in Matlab with the `rref` function.

```
>> M = [1, 1, 0; 2, 3, 1; 3, 4, 1]

M =

     1     1     0
     2     3     1
     3     4     1

>> rref(M)

ans =

     1     0    -1
     0     1     1
     0     0     0
```

This makes it extremely easy to solve a system of linear equations in Matlab. For example, our system from before,

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

is now extremely easy to solve:

```
>> M = [1, 2, 4, 5;
        2, 4, 5, 4;
        4, 5, 4, 2];
>> rref(M)

ans =

     1     0     0     1
     0     1     0    -2
     0     0     1     2
```

This tells us the original system is equivalent to the trivial system

$$\begin{aligned} x &= 1 \\ y &= -2 \\ z &= 2 \end{aligned}$$

Of course, we can determine this from the (non-row-reduced) echelon form we had before, but using the rref makes life a little bit easier since we don't have to do any "back-substitution" to determine the variables; the rref has essentially already performed the back-substitution for us.

When a system has a unique solution the coefficient matrix of the system is invertible. Writing the system as the vector equation $A\vec{x} = \vec{b}$, we can compute the solution $\vec{v}$ as $A^{-1}\vec{b}$, assuming A^{-1} exists. In Matlab we compute the inverse of a matrix using the `inv` function. Letting A be the coefficient matrix of our system from before, the inverse is computed in Matlab as

```
>> A = [1, 2, 4; 2, 4, 5; 4, 5, 4];
>> inv(A)

ans =

    1.0000   -1.3333    0.6667
   -1.3333    1.3333   -0.3333
    0.6667   -0.3333         0
```

We can then compute the solution to the system as follows:

```
>> A = [1, 2, 4; 2, 4, 5; 4, 5, 4];
>> b = [5; 4; 2];
>> inv(A) * b

ans =

    1.0000
   -2.0000
    2.0000
```

This multiplication of a vector by the inverse of a matrix is also an extremely common operation, and so there's a minor shortcut in Matlab: `inv(A) * b` is equivalent to `A \b`. (The idea behind this syntax is that multiplying `b` by the inverse of `A` is kind of like dividing `b` by `A`. This isn't the usual division operation though, so the backslash \ is used instead of the forward slash, `/`.)

```
>> A = [1, 2, 4; 2, 4, 5; 4, 5, 4];
>> b = [5; 4; 2];
>> A \ b

ans =

    1.0000
   -2.0000
    2.0000
```

Recalling that matrix multiplication is very much not commutative, you have to be extremely careful with this notation. If you reverse the order of `A` and `b` in the command above, you will not calculate the solution to the system:

```
>> b \ A

ans =

    0.4667    0.8000    1.0667
```

11 Taylor Polynomials

Wahrlich es ist nicht das Wissen, sondern das Lernen, nicht das Besitzen sondern das Erwerben, nicht das Da-Seyn, sondern das Hinkommen, was den grössten Genuss gewährt.
It is not knowledge, but the act of learning, not possession but the act of getting there, which grants the greatest enjoyment.

Carl Friedrich Gauss

11.1 Deriving the formula for a Taylor polynomial

"Most" mathematical functions can not be evaluted exactly. For example, functions like $\ln(x)$, $\sqrt[3]{x}$, or $\sin(x)$ can not, for a generic choice of x, be evaluated directly. If we want to have a numeric value after evaluating such a function, we must instead find some way to approximate the function with another function which we can actually evaluate.

There are different types of functions we may wish to use in different types of contexts, but one simple family of functions which we can in principle actually evaluate are polynomials. Thus we may want to find a way of approximating a general function $f(x)$ with a polynomial. In this chapter we will discuss ***Taylor polynomials***, which are one particularly easy-to-define type of polynomial which we can use to approximate a function. Furthermore, we will see how to place bounds on the accuracy of our approximation with Taylor's theorem.

We will see how to define the Taylor polynomial of degree n approximating a function $f(x)$ near a chosen value $x = a$, but we start off with the simplest possible case: when $n = 1$.

The ***Taylor polynomial of*** $f(x)$ ***of degree*** 1 ***centered at*** $x = a$ is the unique degree one polynomial (i.e., a polynomial of the form $b_1x + b_0$ for some constants b_1 and b_0), which we will denote $p_1(x)$, such that $p_1(a)$

and $p_1'(a)$ equal $f(a)$ and $f'(a)$:

$$\begin{aligned} p_1(a) &= f(a) \\ p_1'(a) &= f'(a). \end{aligned}$$

Writing $p_1(x) = b_1 x + b_0$, this becomes

$$\begin{aligned} b_1 a + b_0 &= f(a) \\ b_1 &= f'(a). \end{aligned}$$

This is a system of linear equations where the unknowns are the coefficients b_0 and b_1. Writing this system in matrix notation gives

$$\begin{pmatrix} a & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} b_1 \\ b_0 \end{pmatrix} = \begin{pmatrix} f(a) \\ f'(a) \end{pmatrix}$$

Of course, we instantly see $b_1 = f'(a)$, and plugging this into the first equation we determine $b_0 = f(a) - f'(a) \cdot a$. Thus our polynomial is

$$p_1(x) = f'(a) \cdot x + f(a) - f'(a) \cdot a.$$

Rewriting this as

$$p_1(x) = f'(a) \cdot (x - a) + f(a)$$

we see this is the function whose graph is the tangent line of $y = f(x)$ at $x = a$. Note also this is precisely the linearization of $f(x)$ at $x = a$ that you learned in a first-semester calculus course.

We can use $p_1(x)$ as a reasonable approxmation to $f(x)$, at least for values of x that are close to a.

Example 11.1.
Find the Taylor polynomial of degree 1 which approximates $f(x) = \sqrt{x}$ centered at $x = 4$, and use this to approximate $\sqrt{4.03}$.

Notice in our formula for $p_1(x)$ above we will need to evalute $f(4)$ and $f'(4)$, but this is easy to do:

$$\begin{aligned} f(x) = \sqrt{x} &\implies f(4) = 2 \\ f'(x) = \frac{1}{2\sqrt{x}} &\implies f'(4) = \frac{1}{4} \end{aligned}$$

Thus our polynomial is

$$\begin{aligned} p_1(x) &= f'(4) \cdot (x-4) + f(4) \\ &= \frac{1}{4}(x-4) + 2 \end{aligned}$$

We can now approximate $\sqrt{4.03}$ using the Taylor polynomial:

$$\begin{aligned} \sqrt{4.03} \approx & p_1(4.03) \\ = & \frac{1}{4}(4.03-4) + 2 \\ = & \frac{1}{4} \cdot 0.03 + 2 \\ = & 0.0075 + 2 \\ = & 2.0075 \end{aligned}$$

In the example above we used the Taylor polynomial of degree one to approximate $\sqrt{4.03}$ and computed this to be approximately 2.0075. If you were to enter $\sqrt{4.03}$ into a calculator or computer, however, it would instead given you something like

$$\sqrt{4.03} \approx 2.00748598999$$

which is close to, but not quite the same as, our approximation.

To improve our approximation we may ask for our polynomial to agree not only with $f(a)$ and $f'(a)$, but also $f''(a)$. In order for the second derivative to be non-zero, however, notice we will require a degree two polynomial. We thus define the second degree Taylor polynomial of $f(x)$ centered at $x = a$ to be the polynomial

$$p_2(x) = b_2x^2 + b_1x + b_0$$

satisfying the following three conditions:

$$\begin{aligned} p_2(a) &= f(a) \\ p_2'(a) &= f'(a) \\ p_2''(a) &= f''(a) \end{aligned}$$

Computing our derivatives, $p_2'(x) = 2b_2x + b_1$ and $p_2''(x) = 2b_2$, and plugging into the above we again have a system of linear equations where the

coefficients of the polynomial are the unknowns:

$$\begin{aligned} b_2a^2 + b_1a + b_0 &= f(a) \\ 2b_2a + b_1 &= f'(a) \\ 2b_2 &= f''(a) \end{aligned}$$

Or, in matrix notation,

$$\begin{pmatrix} a^2 & a & 1 \\ 2a & 1 & 0 \\ 2 & 0 & 0 \end{pmatrix} \begin{pmatrix} b_2 \\ b_1 \\ b_0 \end{pmatrix} = \begin{pmatrix} f(a) \\ f'(a) \\ f''(a) \end{pmatrix}$$

We can of course easily see that $b_2 = {}^{f''(a)}/_2$ from the third equation. The second equation then becomes

$$\begin{aligned} f''(a) \cdot a + b_1 &= f'(a) \\ \implies b_1 &= f'(a) - f''(a) \cdot a \end{aligned}$$

and finally the first equation becomes

$$\begin{aligned} ({}^{f''(a)}/_2)\, a^2 + (f'(a) - f''(a) \cdot a) \cdot a + b_0 &= f(a) \\ \implies b_0 &= f(a) - \frac{f''(a)}{2}a^2 - f'(a) \cdot a + f''(a) \cdot a^2 \end{aligned}$$

We can simplify this last expression to

$$b_0 = f(a) - f'(a) \cdot a + \frac{f''(a)}{2} \cdot a^2$$

Plugging all of this back into the b_0, b_1, and b_2 of $p_2(x)$ above gives us

$$p_2(x) = \frac{f''(a)}{2}x^2 + (f'(a) - f''(a) \cdot a)x + \frac{f''(a)}{2} \cdot a^2 - f'(a) \cdot a + f(a)$$

We can rearrange this a little bit to get

$$p_2(x) = \frac{f''(a)}{2}x^2 - f''(a)ax + f''(a)a^2 + f'(x)x - f'(a)a + f(a)$$

Which we may further simplify to

$$p_2(x) = \frac{f''(a)}{2}(x-a)^2 + f'(a)(x-a) + f(a)$$

Example 11.2.
Approximate $\sqrt{4.03}$ using the second degree Taylor polynomial of $f(x) = \sqrt{x}$ centered at $x = 4$.

We simply need to evaluate function and its first and second derivatives at 4 and plug them into the expression we obtained above:

$$\begin{aligned} f(x) = \sqrt{x} &\implies f(4) = 2 \\ f'(x) = \frac{1}{2\sqrt{x}} &\implies f'(4) = \frac{1}{4} \\ f''(x) = \frac{-1}{4\sqrt{x^3}} &\implies f''(4) = \frac{-1}{32} \end{aligned}$$

Hence our Taylor polynomial is

$$p_2(x) = \frac{-1}{64}(x-4)^2 + \frac{1}{4}(x-4) + 2$$

and so we approximate

$$\begin{aligned} \sqrt{4.03} \approx & p_2(4.03) \\ = & \frac{-1}{64}(0.03)^2 + \frac{1}{4} \cdot 0.03 + 2 \\ = & -0.000140625 + 0.0075 + 2 \\ = & 2.007485937 \end{aligned}$$

We could of course repeat this process asking for polynomials to agree with $f(a)$, $f'(a)$, $f''(a)$, $f'''(a)$, $f''''(a)$, and so on. In general, to agree with the first n derivatives of the function we will require a polynomial of degree n which has $n+1$ unknown coefficients ($n+1$ because we have b_0 through b_n). We equate the values of the polynomial's derviatves with those of f to obtain a system of $n+1$ linear equations in $n+1$ unkowns, which we can then solve to determine the coefficients of the polynomial. This gives us the ***degree n Taylor polynomial of $f(x)$ centered at $x = a$***, denoted

$$p_n(x) = b_n x^n + b_{n-1}x^{n-1} + \cdots + b_2 x^2 + b_1 x + b_0$$

where the coefficients are chosen to satisfy

$$\begin{aligned} p_n(a) &= f(a) \\ p_n'(a) &= f'(a) \\ p_n''(a) &= f''(a) \\ &\vdots \\ p_n^{(n-1)}(a) &= f^{(n-1)}(a) \\ p_n^{(n)}(a) &= f^{(n)}(a) \end{aligned}$$

Writing out the left-hand sides of the equations above gives us

$$\begin{aligned} b_n a^n + b_{n-1}a^{n-1} + \cdots + b_2 a^2 + b_1 a + b_0 &= f(a) \\ n b_n a^{n-1} + (n-1)b_{n-1}a^{n-2} + \cdots + 2b_2 a + b_1 &= f'(a) \\ n(n-1)b_n a^{n-2} + (n-1)(n-2)b_{n-1}a^{n-3} + \cdots + 2b_2 &= f''(a) \\ n(n-1)(n-2)b_n a^{n-2} + (n-1)(n-2)(n-3)b_{n-1}a^{n-4} + \cdots + 3\cdot 2 b_3 &= f''(a) \\ &\vdots \\ n(n-1)(n-2)\cdots 3 b_n a^2 + (n-1)(n-2)\cdots 2 b_{n-1} a + (n-2)(n-3)\cdots 2\cdot 1 b_{n-2} &= f^{(n-2)}(a) \\ n(n-1)(n-2)\cdots 2 b_n a + (n-1)(n-2)\cdots 2\cdot 1 b_{n-1} &= f^{(n-1)}(a) \\ n(n-1)(n-2)\cdots 2\cdot 1 b_n &= f^{(n)}(a) \end{aligned}$$

We can simplify our notation a little bit by using factorials,

$$n! = n(n-1)(n-2)\cdots 2\cdot 1.$$

The last three equations above, for example, become

$$\begin{aligned} \frac{n!}{2!}b_n a^2 + \frac{(n-1)!}{1!}b_{n-1}a + (n-2)!b_{n-2} &= f^{(n-2)}(a) \\ \frac{n!}{1!}b_n a + (n-1)!b_{n-1} &= f^{(n-1)}(a) \\ n! b_n &= f^{(n)}(a) \end{aligned}$$

Extending the pattern, the entire system of may be written as

$$\begin{aligned}
\frac{n!}{n!}b_n a^n + \frac{(n-1)!}{(n-1)!}b_{n-1}a^{n-1} + \cdots \frac{2!}{2!}b_2 a^2 + \frac{1!}{1!}b_1 a + 0!b_0 &= f(a)\\
\frac{n!}{(n-1)!}b_n a^{n-1} + \frac{(n-1)!}{(n-2)!}b_{n-1}a^{n-2} + \cdots + \frac{2!}{1!}b_2 a + 1!b_1 &= f'(a)\\
\frac{n!}{(n-2)!}b_n a^{n-2} + \frac{(n-1)!}{(n-3)!}b_{n-1}a^{n-3} + \cdots + 2!b_2 &= f''(a)\\
\frac{n!}{(n-3)!}b_n a^{n-2} + \frac{(n-1)!}{(n-4)!}b_{n-1}a^{n-4} + \cdots + 3!b_3 &= f'''(a)\\
&\vdots\\
\frac{n!}{2!}b_n a^2 + \frac{(n-1)!}{1!}b_{n-1}a + (n-2)!b_{n-2} &= f^{(n-2)}(a)\\
\frac{n!}{1!}b_n a + (n-1)!b_{n-1} &= f^{(n-1)}(a)\\
n!b_n &= f^{(n)}(a)
\end{aligned}$$

Though slightly tedious to do by hand, we now see how to easily compute the solution to our system of equations. The last equation instantly tells us $b_n = \frac{f^{(n)}(a)}{n!}$, plugging this into the next-to-last equation gives us

$$\begin{aligned}
&\frac{n!}{1!}b_n a + (n-1)!b_{n-1} = f^{(n-1)}(a)\\
\implies &\frac{n!}{1!}\frac{f^{(n)}(a)}{n!}a + (n-1)!b_{n-1} = f^{(n-1)}(a)\\
\implies &f^{(n)}(a)a + (n-1)!b_{n-1} = f^{(n-1)}(a)\\
\implies &b_{n-1} = \frac{1}{(n-1)!}\left(f^{(n-1)}(a) - f^{(n)}(a)a\right)
\end{aligned}$$

While we can solve this system by hand, or on a computer, it turns out if we do a little bit of algebra we can rewrite our resulting polynomial so that each term has a very simple format. The idea here is very simple. If $a = 0$ in the above, "most" terms in our expression for the b_k coefficients cancel out to give us

$$b_k = \frac{f^{k)}(0)}{k!}.$$

and so, if $a = 0$, the polynomial would become

$$p_n(x) = \frac{f^{(n)}(0)}{n!}x^n + \frac{f^{(n-1)}(0)}{(n-1)!}x^{n-1} + \cdots + \frac{f''(0)}{2!}x^2 + f'(0)x + f(0)$$

So, what should we do if a is not equal to zero? We can just perform a change of variables: if we let $u = x - a$ and find the Taylor polynomial for $f(u)$ at $u = 0$, we would have

$$p_n(u) = \frac{f^{(n)}(u=0)}{n!}u^n + \frac{f^{(n-1)}(u=0)}{(n-1)!}u^{n-1} + \cdots + \frac{f''(u=0)}{2!}u^2 + f'(u=0)u + f(0)$$

where the "$u = 0$" above is to remind us that u is zero above, not x. Now to rewrite this in terms of x, notice that since $u = x - a$, $x = u + a$. Hence when $u = 0$, $x = a$. Thus our 0's above become a's, and the u's are replaced by $x - a$ to give us

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

Remark.
If you don't like this change of variable stuff, an alternative way to think about this is that we'll find the Taylor polynomial of $g(x) = f(x - a)$ centered at 0, then rewrite the terms in that polynomial in terms of f instead of g.

We have thus derived the following formula for the n-th degree Taylor polynomial of $f(x)$ centered at $x = a$:

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

One nice thing about the formula for the Taylor polynomials above is that they have a recursive definition:

$$p_n(x) = \frac{f^{(n)}}{n!}(x-a)^n + p_{n-1}(x).$$

Thus if you already know one Taylor polynomial for your function, you can easily modify it to get the next degree.

Example 11.3.
Approximate $\sqrt{4.03}$ using the third degree Taylor polynomial of $f(x) = \sqrt{x}$ centered at $x = 4$.

By our recursive formula, we may write

$$p_3(x) = \frac{f'''(4)}{3!}(x-4)^3 + p_2(x)$$

We easily compute

$$f'''(x) = \frac{3}{8\sqrt{x^5}}$$
$$f'''(4) = \frac{3}{256}$$

and so

$$\begin{aligned}\sqrt{4.03} &\approx p_3(4.03)\\ &= \frac{3}{256}(4-4.03)^3 + p_2(4.03)\\ &= 0.00000031640625 + 2.007485937\\ &= 2.00748625390625\end{aligned}$$

11.2 The error in Taylor polynomial approximation

We have claimed that the Taylor polynomial is a "good" approximation for a function near the center, $x = a$, of the approximation. But what does this mean? Our next goal is to quantify the error in a Taylor polynomial approximation to understand "how good" the approximation is.

First let's recall some notation. We say that a function is C^n on an interval $[\alpha, \beta]$ if the function is continuous on $[\alpha, \beta]$, n-times differentiable on (α, β), and the n-th derivative is also continuous on this interval. The set of all such functions is denoted $C^n([\alpha, \beta])$. The set of all continuous functions (differentiable or not) on $[\alpha, \beta]$ is denoted $C^0([\alpha, \beta])$.

Example 11.4.
The following function,

$$f(x) = \begin{cases} x^2 \sin\left(\frac{1}{x}\right) & \text{if } x \neq 0 \\ 0 & \text{if } x = 0 \end{cases}$$

is in $C^0([-1,1])$, but not $C^1([-1,1])$. It is easy to verify this function is continuous at each point in $[-1,1]$ using the limit definition of continuity, and in fact the function is differentiable at each point as well. However the derivative $f'(x)$ *is not* continuous at $x = 0$. Applying the fundamental theorem of calculus to construct an antiderivative of $f(x)$, call it $F(x)$, then gives a function that is C^1 but not C^2; the antiderivative of that is a function which is C^2 but not C^3, and so on.

Taylor's theorem will tell us what the error in our approximation of $f(x)$ by the n-th degree Taylor polynomial, $p_n(x)$, looks like, and once we have that information we can try to determine what the degree n should bemake $p_n(x)$ as close to $f(x)$ as we would like.

Theorem 11.1 (Taylor's theorem)**.**
Suppose $f \in C^{n+1}([\alpha,\beta])$. Let $p_n(x)$ denote the n-th degree Taylor polynomial of $f(x)$ centered at $x = a$, and let $R_n(x)$ denote the error in this approximation. That is, $R_n(x) = f(x) - p_n(x)$. For each $x \in (\alpha,\beta)$, there exists a value $c_x \in (\alpha,\beta)$ such that

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

The prove Taylor's theorem we will use a version of the mean value theorem for higher-order derivatives, but first let's recall the basic situation of the mean value theorem and Rolle's theorem for first derivatives.

Theorem 11.2 (Rolle's theorem)**.**
Suppose $f \in C^1([\alpha,\beta])$ and $f(\alpha) = f(\beta)$. Then there exists a $c \in$

(α, β) *such that* $f'(c) = 0$.

Proof.
Suppose instead that no such c existed: $f'(x) \neq 0$ for each $x \in (\alpha, \beta)$. Then by the intermediate value theorem, $f'(x) > 0$ for every x, or $f'(x) < 0$ for every x. If $f'(x) > 0$ for each x, then f is a strictly increasing function on (α, β) and this contradicts the assumption $f(\alpha) = f(\beta)$. Similarly, if $f'(x) < 0$ for every x, then f is a strictly decreasing function and this also contradicts the assumption $f(\alpha) = f(\beta)$. Hence the only way to avoid a contradiction is to have $f'(c) = 0$ for some c. □

Once we know Rolle's theorem, we can easily prove the mean value theorem.

Theorem 11.3 (The mean value theorem)**.**
Suppose $f \in C^1([\alpha, \beta])$. *Then there exists a* $c \in (\alpha, \beta)$ *such that*

$$f'(c) = \frac{f(\beta) - f(\alpha)}{\beta - \alpha}$$

Proof.
We reduce this to Rolle's theorem by considering the functin $g : [\alpha, \beta] \to \mathbb{R}$ defined by

$$g(x) = f(x) - f(\alpha) - (x - \alpha) \cdot \frac{f(\beta) - f(\alpha)}{\beta - \alpha}.$$

This function g is obviously in $C^1([\alpha, \beta])$ since f is. Furthermore, we

can easily compute that g satisfies the hypotheses of Rolle's theorem:

$$\begin{aligned} g(\alpha) &= f(\alpha) - f(\alpha) - (\alpha - \alpha) \cdot \frac{f(\beta) - f(\alpha)}{\beta - \alpha} \\ &= 0 \end{aligned}$$

$$\begin{aligned} g(\beta) &= f(\beta) - f(\alpha) - (\beta - \alpha) \cdot \frac{f(\beta) - f(\alpha)}{\beta - \alpha} \\ &= f(\beta) - f(\alpha) - (f(\beta) - f(\alpha)) \\ &= 0 \end{aligned}$$

Thus by Rolle's theorem there exists some value of c in (α, β) such that $g'(c) = 0$. Notice, however,

$$g'(x) = f'(x) - \frac{f(\beta) - f(\alpha)}{\beta - \alpha}$$

and so

$$\begin{aligned} & g'(c) = 0 \\ \implies & f'(c) - \frac{f(\beta) - f(\alpha)}{\beta - \alpha} = 0 \\ \implies & f'(c) = \frac{f(\beta) - f(\alpha)}{\beta - \alpha} \end{aligned}$$

□

Though we haven't proven Taylor's theorem yet, notice that if you were to apply Taylor's theorem in the space case of a zero degree polynomial, $p_0(x)$ is the constant function $p_0(x) = f(a)$, then we would have the mean value theorem. That is, approximating $f(x)$ by the constant function $p_0(x) = f(a)$, Taylor's theorem claims the existence of a c_x such that

$$R_0(x) = f(x) - f(a) = f'(c_x) \cdot (x - a)$$

and solving for $f'(c_x)$ gives

$$f'(c_x) = \frac{f(x) - f(a)}{x - a},$$

and this is precisely what the mean value theorem tells us when applied to the interval $[a, x]$.

To prove Taylor's theorem ing eneral, we take our cues from the proof of the mean value theorem: Taylor's theorem is just a version of the mean value theorem for higher-order derivatives, and since the mean value threom is just Rolle's theorem, it makes sense that we should look for a higher-order version of Rolle's theorem.

Theorem 11.4 (Higher-order Rolle's theorem)**.**
Suppose $f \in C^{n+1}([\alpha, \beta])$ *for some integer* $n \geq 0$*, and further assume*

$$(f\alpha) = f'(\alpha) = f''(\alpha) = ... = f^{(n)}(\alpha) = f(\beta) = 0.$$

Then there exists a $c \in [\alpha, \beta]$ *such that* $f^{(n+1)}(c) = 0$*.*

Proof.
By the "normal" Rolle's theorem, there exists a $c_1 \in [\alpha, \beta]$ such that $f'(c_1) = 0$. Now apply the usual Rolle's theorem again to $f'(x)$ on the interval $[\alpha, c_1]$ to see there exists a $c_2 \in [\alpha, c_1]$ such that $f''(c_2) = 0$. Apply Rolle's theorem to $f''(x)$ on $[\alpha, c_2]$ to determine the existence of a $c_3 \in [\alpha, c_2]$ such that $f'''(c_3) = 0$. Continuing this process we generate a sequence of numbers $c_k \in [\alpha, c_{k-1}]$ such that $f^{(k+1)}(c_k) = 0$. Taking c to be the element c_n in this sequence proves the theorem. □

We can now prove a higher-order version of the mean value theorem, aka Taylor's theorem.

Proof of Taylor's theorem.
Suppose $f \in C^{n+1}([\alpha, \beta])$ and let $p_n(x)$ be the n-th order Taylor poly-

nomial of f centered at some $a \in (\alpha, \beta)$. Recall that this means

$$\begin{aligned} p_n(a) &= f(a) \\ p_n'(a) &= f'(a) \\ p_n''(a) &= f''(a) \\ &\vdots \\ p_n^{(n)}(a) &= f^{(n)}(a) \end{aligned}$$

Fix any number$b \in [\alpha, \beta]$ and consider the function

$$g(x) = f(x) - p_n(x) - \frac{(x-a)^{n+1}}{(b-a)^{n+1}}(f(b) - p_n(b)).$$

Notice

$$\begin{aligned} g'(x) &= f'(x) - p_n'(x) - \frac{(n+1)(x-a)^n}{(b-a)^{n+1}}(f(b) - p_n(b)) \\ g''(x) &= f''(x) - p_n''(x) - \frac{(n+1)n(x-a)^{n-1}}{(b-a)^{n+1}}(f(b) - p_n(b)) \\ g'''(x) &= f'''(x) - p_n'''(x) - \frac{(n+1)n(n-1)(x-a)^{n-2}}{(b-a)^{n+1}}(f(b) - p_n(b)) \\ &\vdots \\ g^{(n)}(x) &= f^{(n)}(x) - p_n^{(n)}(x) - \frac{(n+1)!(x-a)}{(b-a)^{n+1}}(f(b) - p_n(b)) \end{aligned}$$

It's now easy to see that g satisfies the hypotheses of the higher-order version of Rolle's theorem, and so there exists a $c \in [a, b]$ such that $g^{(n+1)}(c) = 0$. But since

$$g^{(n+1)}(x) = f^{(n+1)}(x) - p_n^{(n+1)}(x) - \frac{(n+1)!}{(b-a)^{n+1}}(f(b) - p_n(b))$$

We must have

$$\begin{aligned} f^{(n+1)}(c) &= \frac{(n+1)!}{(b-a)^{n+1}}(f(b) - p_n(b)) \\ \implies f(b) - p_n(b) &= \frac{(b-a)^{n+1}}{(n+1)!} f^{(n+1)}(c). \end{aligned}$$

Replacing b with the variable x, we have the conclusion of Taylor's theorem:

$$R_n(x) = f(x) - p_n(x) = \frac{(x-a)^{n+1}}{(n+1)!} f^{(n+1)}(c_x)$$

where the choice of c above depends on the x. □

Notice that as

$$R_n(x) = f(x) - p_n(x) = \frac{(x-a)^{n+1}}{(n+1)!} f^{(n+1)}(c_x)$$

we can write

$$\begin{aligned} f(x) &= p_n(x) + R_n(x) \\ &= \sum_{k=0}^{n} \frac{(x-a)^k}{k!} f^{(k)}(a) + \frac{(x-a)^{n+1}}{(n+1)!} f^{(n+1)}(c_x) \end{aligned}$$

and this is an equality, not simply an approximation. The "hard part" of the abvoe expression – the thing that might cause you some concern if you were to try to use the above to compute the exact value of $f(x)$ for a given x – is that it doesn't tell us how to find c_x. Taylor's theorem promises us such a c_x must exist, but it doesn't give us any clue as to how ot go about computing c_x.

Despite this, we can still sometimes use Taylor's theorem to put bounds on the remainder $R_n 9x)$. I.e., even if $p_n(x)$ is only an approximation to $f(x)$, we may be able to give a quantitative answer to the question "how good of an approximation is $p_n(x)$?".

Example 11.5.
For x in $(-\pi/2, \pi/2)$, what is the maximum amount of error in approximating $\sin(x)$ with the Taylor polynomial $p_n(x)$ centered at $a = 0$ for $n = 1$, $n = 2$, and $n = 3$?

In the case of $n = 1$ the Taylor polynomial is

$$\begin{aligned} p_1(x) &= \frac{(x-0)^0}{0!} \frac{d^0}{dx^0}\bigg|_{x=0} \sin(x) + \frac{(x-1)^1}{1!} \frac{d^1}{dx^1}\bigg|_{x=0} \sin(x) \\ &= \sin(0) + x\cos(0) \\ &= x \end{aligned}$$

The error in this approximation is

$$\begin{aligned} R_1(x) &= \frac{(x-0)^2}{2!}\frac{d^2}{dx^2}\bigg|_{x=c_x} \sin(x) \\ &= \frac{-x^2}{2}\cos(c_x). \end{aligned}$$

Now, we don't know what c_x is, so we don't know exactly what $R_1(x)$ is. However, we do know $|x| < \pi/2$ (because we are explicitly assuming $x \in (-\pi/2, \pi/2)$ in this problem), and $|\cos(x)| \leq 1$ for all x. Thus for $|x| < \pi/2$ we know

$$\begin{aligned} |R_1(x)| &= \left|\frac{-x^2}{2}\cos(c_x)\right| \\ &= \left|\frac{-x^2}{2}\right| |\cos(c_x)| \\ &= \frac{1}{2}|x|^2|cos(c_x)| \\ &< \frac{1}{2}\cdot\left(\frac{\pi}{2}\right)^2 \cdot 1 \\ &= \frac{\pi^2}{4} \\ &\approx 2.4674 \end{aligned}$$

and so for x's between $-\pi/2$ and $\pi/2$, the approximation $p_1(x) = x$ is always within $\pi^2/4 \approx 2.4764$ of the true value of $\sin(x)$.

For $n = 2$ we have

$$p_2(x) = x + \frac{x^2}{2}\sin(0) = x$$

and

$$R_2(x) = \frac{(x-0)^3}{3!}\sin(c_x).$$

Thus

$$\begin{aligned} |R_2(x)| &= \left| \frac{x^3}{6} \sin(c_x) \right| \\ &< \frac{1}{6} \left| \frac{\pi}{2} \right|^3 \\ &= \frac{\pi^3}{48} \\ &\approx 0.645 \end{aligned}$$

Finally, for $n = 3$ we have

$$p_3(x) = x - \frac{x^3}{6}$$

and

$$\begin{aligned} |R_3(x)| &= \left| \frac{x^4}{4!} \cos(c_x) \right| \\ &< \frac{1}{24} \left| \frac{\pi}{2} \right|^4 \\ &= \frac{\pi^4}{384} \\ &\approx 0.25367 \end{aligned}$$

Notice that in the above example, we can compute the upper bound for $|R_n(x)|$. So suppose we wanted to find the n that made this error small. If we express an upper bound for $|R_n(x)|$ as a function of n, we can determine how big n must be for $p_n(x)$ to be as good of an approximation as we like.

Example 11.6.
For what values of n is the n-th order Taylor polynomial of $\sin(x)$ centered at $a = 0$ within one one-millionth of the true value of $\sin(x)$ for all $x \in (-\pi/2, \pi/2)$?

We're trying to find the n that guarantees

$$|R_n(x)| < 10^{-6} = 0.000001$$

for all $x \in (-\pi/2, \pi/2)$.

Notice that

$$|R_n(x)| = \left| \frac{(x-0)^{n+1}}{(n+1)!} \frac{d^{n+1}}{dx^{n+1}} \Big|_{x=c_x} \sin(x) \right|$$

But as each derivative of $\sin(x)$ is $\pm\sin(x)$ or $\pm cos(x)$, we know these derivatives are always less than or equal to 1 in absolute value. That is,

$$|R_n(x)| \leq \frac{1}{(n+1)!}|x|^{n+1}$$

But since $-\pi/2 < x < \pi/2$, $|x| < \pi/2$ and so

$$|R_n(x)| < \frac{1}{(n+1)!}\frac{\pi^{n+1}}{2^{n+1}}.$$

Let's notice that $\pi \approx 3.14159...$, so $\pi < 4$ meaning $\pi^{n+1} < 4^{n+1} = (2^2)^{n+1} = 2^{2n+2}$ and so we have

$$|R_n(x)| < \frac{1}{(n+1)!}\frac{2^{2n+2}}{2^{n+1}} = \frac{1}{(n+1)!}2^{n+1}.$$

So, to find the n making $|R_n(x)| < 10^{-6}$, it suffices to find an n making

$$\frac{2^{n+1}}{(n+1)!} < 10^{-6}$$

A very quick calculation on a computer, for instance a loop in Matlab which starts at `n = 1` and computes $\frac{2^{n+1}}{(n+1)!}$ until this is less than 10^{-6}, shows this inequality is satisfied for $n \geq 13$.

So if we are performing some numerical calculation where we need to approximate sine to within one one-millionth of the true value on a computer for all x in $(-\pi/2, \pi/2)$, it suffices for us to use the 13-th degree Taylor polynomial.

Part IV

Numerical Algorithms

12

Root finding revisited, and fixed point iteration

I keep the subject constantly before me, and wait 'till the first dawnings open slowly, by little and little, into a full and clear light.

Isaac Newton
Comment about how he made scientific discoveries, in Biographia Britannica.

12.1 Newton's method, part 2

We described Newton's method previously and then made an excursion to review some linear algebra and calculus. We are now ready to apply that knowledge to study Newton's method more thoroughly. Recall that we have a continuously differentiable function f and we are trying to find a root of f, i.e. a value of x so that $f(x) = 0$. In Newton's method we begin with some initial approximation to the root of the function, call it x_0, and then we iteratively build a sequence of better approximations using

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

We had seen that another root finding method, the bisection algorithm, will always produce approximations which approach a root of the function (assuming a root exists within the initial region given to the bisection algorithm), however we had seen (just via one hand-wavy example) that Newton's method converges to the root much faster. However, Newton's method has a few problems which we will illustrate with examples.

Example 12.1.
Suppose $f(x) = \frac{-11}{3}x^3 - 3x^2 + 5x + 5$ and we use $x_0 = 0$ as our initial approximation. First notice this function does in fact have a root:

Using $f'(x) = -11x^2 - 6x + 5$, it's easy for us to compute the sequence of approximations x_n generated by Newton's method:

$$\begin{aligned} x_1 &= x_0 - \frac{f(x_0)}{f'(x_0)} \\ &= 0 - \frac{5}{5} = -1 \end{aligned}$$

$$\begin{aligned} x_2 &= -1 - \frac{f(-1)}{f'(-1)} \\ &= -1 - \frac{2/3}{0} \end{aligned}$$

Of course, now we have a problem since we can't divide by zero. At this point we're forced to stop Newton's method since we can't continue the calcuation.

Example 12.2.
Consider $f(x) = x^3 - 2x + 2$. Apply Newton's method with $x_0 = 0$.

Since $f'(x) = 3x^2 - 2$ we easily compute

$$\begin{aligned} x_1 &= 0 - \frac{f(0)}{f'(0)} \\ &= 0 - \frac{2}{-2} \\ &= 0 - (-1) \\ &= 1 \end{aligned}$$

$$\begin{aligned} x_2 &= 1 - \frac{f(1)}{f'(1)} \\ &= 1 - \frac{1}{1} \\ &= 0 \end{aligned}$$

Of course, at this point we see the iterates will just bounce back and forth between 0 and 1:

$$\begin{aligned} x_3 &= 1 \\ x_4 &= 0 \\ x_5 &= 1 \\ x_6 &= 0 \\ &\vdots \end{aligned}$$

That is, we keep oscillating between 0 and 1, never converging to the root of our polynomial.

Example 12.3.
Consider $f(x) = \sqrt[3]{x} = x^{1/3}$. Newton's method will give us sequences

generated by

$$\begin{aligned} x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} \\ &= x_n - \frac{x^{1/3}}{\frac{1}{3}x^{-2/3}} \\ &= x_n - 3x^{1/3}x^{2/3} \\ &= x_n - 3x_n \\ &= -2x_n \end{aligned}$$

Thus, starting from $x_0 = 1$, the sequence of our approximations given by Newton's method is

$$\begin{aligned} x_0 &= 1 \\ x_1 &= -2 \\ x_2 &= 4 \\ x_3 &= -8 \\ x_4 &= 16 \\ &\vdots \end{aligned}$$

Now, you might argue the above function is cheating a little bit since $x^{1/3}$ is not continuously differentiable everywhere (since $f'(0)$ is undefined), but we can easily modify this to get a smooth function with the same sort of behavior by replacing our function with an appropriately chosen polynomial in a neighborhood of $x = 0$, such as

$$f(x) = \begin{cases} \sqrt[3]{x} & \text{if } x \leq -1 \text{ or } x \geq 1 \\ -\frac{1}{3}x^3 + \frac{4}{3}x & \text{if } -1 < x < 1 \end{cases}$$

In all three examples above the function we were trying to find the root of did in fact have a root, but Newton's method did not give us approximations converging to the root. So, is there any way we can guarantee Newton's method will produce approximations converging to the root of a function?

To answer this, recall that we found the formula for Newton's method by computing the equation of the line tangent to $y = f(x)$ at x_n and then determining where that line intersected the x-axis. The equation of this

tangent line is

$$y = f(x_n) + f'(x_n) \cdot (x - x_n).$$

Notice that this is simply the first-order Taylor polynomial approximating $f(x)$ centered at x_n,

$$p_1(x) = f(x_n) - f'(x_n) \cdot (x - x_n).$$

Now let $R_1(x)$ be the error in this approximation,

$$R_1(x) = f(x) - p_1(x).$$

By Taylor's theorem there exists some c, depending on x, such that

$$R_1(x) = \frac{f''(c)}{2}(x - x_n)^2.$$

In particular, if we let z denote the true root of the function, then

$$R_1(z) = \frac{f''(c)}{2} \cdot (z - x_n)^2.$$

I.e.,

$$\begin{aligned} f(z) - p_1(z) &= \frac{f''(c)}{2}(z - x_n)^2 \\ \implies f(z) &= p_1(z) + \frac{f''(c)}{2}(z - x_n)^2 \\ &= f(x_n) + f'(x_n)(z - x_n) + \frac{f''(c)}{2}(z - x_n)^2. \end{aligned}$$

However, z is a root of f and so

$$\begin{aligned} 0 &= f(x_n) + f'(x_n)(z - x_n) + \frac{f''(c)}{2}(z - x_n)^2 \\ \implies 0 &= \frac{f(x_n)}{f'(x_n)} + z - x_n + \frac{f''(c)}{2f'(x_n)}(z - x_n)^2 \\ &= -\left[x_n - \frac{f(x_n)}{f'(x_n)}\right] + z + \frac{f''(c)}{2f'(x_n)}(z - x_n)^2 \\ \implies 0 &= z - x_{n+1} + \frac{f''(c)}{2f'(x_n)} \cdot (z - x_n)^2 \\ \implies z - x_{n+1} &= (z - x_n)^2 \cdot \left(\frac{-f''(c)}{2f'(x_n)}\right) \end{aligned}$$

That is, at least for x_n "sufficiently close" to z, the error in approximating z by x_{n+1} is some constant times the square of the error in approximating z by x_n,

$$z - x_{n+1} = K(z - x_n)^2$$

Notice that when $z - x_n < 1$, $(z - x_n)^2$ is even smaller: the error essentially squares at each step.

The K above technically depends on x_n, so the K for $z - x_{n+1}$ and the K for $z - x_{n+2}$ are different. To be more precise, we should write

$$z - x_{n+1} = K_n(z - x_n)^2.$$

If f' and f'' are continuous, however, then for x_n's "close" to the true root z, $f'(x_n) \approx f'(z)$ and similarly $f''(c) \approx f''(z)$, since $c \in [z, x_n]$. Thus, if we define

$$M = \frac{-f''(z)}{2f'(z)}$$

then each $K_n \approx M$ (for x_n sufficiently close to z).

Thus we can write

$$z - x_{n+1} \approx M(z - x_n)^2,$$

but the same sort of analysis shows that

$$z - x_n \approx M(z - x_{n-1})^2$$

and

$$z - x_{n-1} \approx M(z - x_{n-2})^2.$$

Continuing this process we can say

$$\begin{aligned} z - x_{n+1} &\approx M(z - x_n)^2 \\ &\approx M(M(z - x_{n-1})^2)^2 = M^3(z - x_{n-1})^4 \\ &\approx M^3(M(z - x_{n-2})^2)^4 = M^7(z - x_{n-2})^2 \\ &\approx \cdots \\ &\approx M^{2^n-1}(z - x_0)^{2^n} \end{aligned}$$

Now if we multiply both sides of our approximation

$$z - x_{n+1} \approx M^{2^n-1}(z - x_0)^{2^n}$$

by M we obtain

$$\begin{aligned} M(z - x_{n+1}) &\approx M \cdot M^{2^n-1}(z - x_0)^{2^n} \\ &= M^{2^n}(z - x_0)^{2^n} \\ &= [M(z - x_0)]^{2^n} \end{aligned}$$

If $x_{n+1} \to z$, then $|z - x_{n+1}| \to 0$ which means

$$\begin{aligned} & M|z - x_{n+1}| \to 0 \\ \implies & |M(z - x_{n+1})|^{2^n} \to 0 \end{aligned}$$

But if r is a positive number and $r^n \to 0$, then $r < 1$. That is, if $x_n \to z$, we must have

$$\begin{aligned} & |M(z - x_0)| < 1 \\ \implies & |z - x_0| < {}^1\!/_{|M|} \\ \implies & |z - x_0| < \left| \frac{-f'(z)}{2f''(z)} \right| \end{aligned}$$

This tells us that Newton's method converges provided we choose an initial x_0 satisfifying the about inequality. I.e., if x_0 is "sufficiently close" to the true root z, then Newton's method started from x_0 will converge to z. As we have seen, however, with a poor choice of x_0, Newton's method may very well not converge.

In principle Newton's method generates an infinite sequence of approximations which approach the true root of the function, provided our initial approximation x_0 is "sufficiently close" to this true root. Of course, when we actually implement Newton's method on a computer we can't generate an infinite sequence of approximations; our process must stop at some point. So, when should we stop iterating Newton's method? In the case of the bisection method we iterated until our approximation was within some ε distance of the true solution, and we determined this requires

$$\left\lceil \log_2 \left(\frac{b-a}{\varepsilon} \right) \right\rceil$$

if our initial interval is $[a, b]$. How long should we iterate Newton's method if we likewise want our approximation to be within ε of the true root?

A simple mean value theorem calculation shows that if x_n is our sequence of approximations from Newton's method, and if z is the true root of the function $f(x)$, then for each n there exists some $c \in [x_n, z]$ (c depends on x_n, so really this is a sequence of c values) such that

$$\begin{aligned} f'(c) &= \frac{f(x_n) - f(z)}{x_n - z} \\ \implies f'(c)(x_n - z) &= f(x_n) - f(z) \end{aligned}$$

Keeping in mind $f(z) = 0$, though, this becomes

$$\begin{aligned} f'(c)(x_n - z) &= f(x_n) \\ \implies x_n - z &= \frac{f(x_n)}{f'(c)} \\ &\approx \frac{f(x_n)}{f'(x_n)} \\ \implies z - x_n &\approx -\frac{f(x_n)}{f'(x_n)} \\ &= x_n - x_n - \frac{f(x_n)}{f'(x_n)} \\ &= x_n - \frac{f(x_n)}{f'(x_n)} - x_n \\ &= x_{n+1} - x_n \end{aligned}$$

That is, for "sufficiently close" x_n, the difference between x_n and the true root z is roughly equal to the difference between successive approximations. So, if we want to estimate z to within ε of the true value, we are justified in iterating Newton's method until successive approximations are within ε of one another.

12.2 Fixed point iteration

Notice that in Newton's method we have a sequence of points x_{n+1} given by the recurrence relation

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

If we write the right-hand side as $g(x) = x - {}^{f(x)}/_{f'(x)}$, then Newton's method can be written as

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

Notice that if z is a root of the original function, so $f(z) = 0$, then

$$g(z) = z - \frac{f(z)}{f'(z)} = z.$$

I.e., finding a solution to $f(x) = 0$ is equivalent to finding a solution to $g(x) = x$.

In general, if g is a function and x is a point such that $g(x) = x$, then we call x a ***fixed point*** of g. There are many theorems in mathematics which guarantee the existence of fixed points of functions under very general conditions, as well as many applications that rely on computing the fixed points of a given function.

For example, one general theorem is the ***Brouwer fixed point theorem*** which states that every continuous function from a closed ball to itself has a fixed point. This has some fun and surprising applications, and also some more useful applications. One fun application is the following: if you were to take a cup of coffee, take a snapshot of every single particle in that coffee, stir the coffee, and then take another snapshot, there would be at least one particle that was in the same place in both snapshots. An important application is the fundamental theorem of algebra, which states all polynomials with complex coefficients have a root.

Another important application is Google's Page Rank algorithm, which is how Google decides to rank the search results you see when you use Google. This algorithm essentially revolves around creating a special function which encodes how web pages containing the phrase you search for link to one another, and the fixed points of this function tell you which pages are the most "popular." Google then sorts the pages you see by their popularity as determined by this fixed point. (If you've taken a course where you learned about Markov chains and stationary distributions, this is essentially the same thing as finding the stationary distribution of some giant Markov chain where the vertices represent web pages and the edges represent links between web pages.)

Just for fun, we'll mention one more interesting application of fixed points in economics. In game theory, a ***Nash equilibrium*** is where no player of the game can benefit from changing their strategy if no other player changes their strategy. This is essentially a fixed point problem, figuring this out won John Nash the Nobel prize in economics in 1994. (Nash's original proof used the Brouwer fixed point theorem, by the way, to show that such equilibria exist.)

Anyway, suppose we have a function $g : \mathbb{R} \to \mathbb{R}$ which we want to find a fixed point of. If we know some basic information about g, we can actually find fixed points by simply iterating the function. That is, pick any starting point x_0, and then construct a sequence of points x_n by defining $x_{n+1} = g(x_n)$. If we make a few assumptions about g, it can be shown such a sequence always converges to the true fixed point of the function. (This, by the way, is what's going on when you try to find the stationary distribution of a Markov chain by simply raising the transition matrix to higher and higher powers.) This process is called ***fixed point iteration***.

Let's first start building up some theory about fixed points that will justify when a a sequence produced in this way converges to a fixed point.

First let's make a really simple geometric observation: if $g(x)$ has a fixed point, then that fixed point occurs when the graph $y = g(x)$ crosses the line $y = x$. That is, if $y = g(x)$ and if $y = x$, then we must have $x = g(x)$, which is exactly what it means to have a fixed point. Though this is a very simple observation, it can helpful to keep it in mind while reading Lemma 12.1 and Theorem 12.2 below.

From our geometric observation above it's clear that not every function has a fixed point. For example, the function $g(x) = x^2+1$ can't have a fixed point since the line $y = x$ never intersects the parabola $y = x^2 + 1$. This is clear if you graph the line and parabola, but is also easy to determine algebraically. If $g(x) = x^2 + 1$ had a fixed point, then there would be a (real) solution to $x^2 + 1 = x$, or equivalently $x^2 - x + 1 = 0$. But plugging into the quadratic formula shows there are no real solutions:

$$x = \frac{1 \pm \sqrt{1-4}}{2} = \frac{1 \pm \sqrt{-3}}{2}.$$

In the simplest situations finding a fixed point is just solving such an algebraic equation, but "most" of the time we won't be able to do this algebra and so we must estimate the fixed point numerically. Before doing that, it might be good to know for sure that a fixed point does in fact exist. The following lemma gives us a sufficient condition guaranteeing the existence of a fixed point.

Lemma 12.1.
If $g(x)$ is a continuous function on an interval $[a, b]$ where $g(a) \geq a$ and $g(b) \leq b$, then g has a fixed point in $[a, b]$. That is, there exists some $x \in [a, b]$ so that $g(x) = x$.

Proof.
Consider the function $G(x) = g(x) - x$. Notice that G has a root if and only if g has a fixed point: $G(x) = 0$ if and only if $g(x) = x$. However, $G(a) = g(a) - a \geq 0$ and $G(b) = g(b) - b \leq 0$. Since g is

continuous, G is as well, and by the intermediate value theorem there must exist a point in $[a, b]$ where $G(x) = 0$, meaning $g(x) = x$. □

Lemma 12.1 tells us a fixed point must exist, but it doesn't tell us how to find it. Of course, our goal right now is to approximate the fixed point using fixed point iteration (constructing the sequence $x_{n+1} = g(x_n)$). The next theorem goes a step further and tells us that under some some stronger hypothesis not only will a fixed point exist, but fixed point iteration will give us a sequence converging to the fixed point from *any* starting point.

Theorem 12.2.
Suppose g is a continuously differentiable function defined on $[a, b]$ whose range is also contained in $[a, b]$ and which has the property that if λ is the largest absolute value of $g'(x)$ on $[a, b]$,

$$\lambda = \max_{a \le x \le b} |g'(x)|$$

and $\lambda < 1$, then

1. *g has a unique fixed point in $[a, b]$.*
2. *Given any $x_0 \in [a, b]$ the sequence of x_n values generated by $x_n = g(x_{n-1})$ converges to the unique fixed point.*
3. *Letting z denote the unique fixed point,*

$$|z - x_n| \le \frac{\lambda^n}{1 - \lambda} |x_1 - x_0|$$

4. *The derivative $g'(z)$ equals*

$$g'(z) = \lim_{n \to \infty} \frac{z - x_{n+1}}{z - x_n}$$

and as a consequence if x_n is "sufficiently close" to z,

$$z - x_{n+1} \approx g'(z) \cdot (z - x_n).$$

Proof.

1. By assumption $g(a) \geq a$ and $g(b) \leq b$, so we know there exists at least one fixed point by Lemma 12.1.

   Now suppose there were two distinct fixed points $\alpha, \beta \in [a, b]$, and suppose without loss of generality that $\alpha < \beta$. Note that the mean value theorem promises there exists a $c \in [\alpha, \beta]$ such that

   $$\begin{aligned} g'(c) &= \frac{g(\beta) - g(\alpha)}{\beta - \alpha} \\ \implies g(\beta) - g(\alpha) &= g'(c) \cdot (\beta - \alpha) \\ \implies |g(\beta) - g(\alpha)| &= |g'(c)| \cdot |\beta - \alpha| \end{aligned}$$

   But we know $|g'(x) \leq 1$, and so

   $$|g(\beta) - g(\alpha)| < |\beta - \alpha|.$$

   But α and β were assumed to be fixed points, meaning $g(\beta) = \beta$ and $g(\alpha) = \alpha$, and the above inequality thus becomes

   $$|\beta - \alpha| < |\beta - \alpha|$$

   and this is impossible. Hence it is impossible that g has two distinct fixed points in $[a, b]$.

2. Let $x_n = g(x_{n-1})$. For the fixed point z, notice

   $$|z - x_n| = |g(z) - g(x_{n-1})| = g'(c)|z - x_{n-1}|$$

   by the mean value theorem, for some c between x_{n-1} and z. However, we assumed $g'(c) < 1$ for all c and so

   $$|z - x_n| < |z - x_{n-1}|.$$

In particular, because the maximum absolute value of the derivative is λ, the $g'(c)$ above is at most λ and so

$$\begin{aligned}|z - x_n| &\leq \lambda|z - x_{n-1}| \\ &\leq \lambda \cdot \lambda|z - x_{n-2}| = \lambda^2|z - x_{n-2}| \\ &\leq \lambda \cdot \lambda^2|z - x_{n-3}| = \lambda^3|z - x_{n-3}| \\ &\vdots \\ &\leq \lambda^n|z - x_0|.\end{aligned}$$

But $\lambda < 1$, so $\lambda^n \to 0$ and hence $|z - x_n| \to 0$, and so $z \to x_n$.

3. For any n, let c_n be the x-value guaranteed by the mean value theorem to satisfy

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

Notice, though, that c_n is in the interval between x_n and z_n and x_n converges to z. Hence c_n must also converge to z and since $g'(x)$ is continuous we must have

$$g'(z) = \lim_{n\to\infty} g'(c_n) = \lim_{n\to\infty} \frac{z - x_{n+1}}{z - x_n}.$$

4. This is an immediate consequence of part (3) of the theorem. Since $\frac{z-x_{n+1}}{z-x_n} \to g'(z)$ and $x_n \to z$, for large values of n we have

$$\begin{aligned}\frac{z - x_{n+1}}{z - x_n} &\approx g'(z) \\ \implies z - x_{n+1} &\approx g'(z)(z - x_n).\end{aligned}$$

□

The take-away of this theorem is that it gives us a sufficient condition for guaranteeing the existence of a fixed point, as well as telling us how to approximate the fixed point. Moreover, it tells us (approximately) how quickly the approximations x_n converge to the fixed point z.

In general, if a sequence x_n converges to z and if

$$|z - x_{n+1}| \leq c|z - x_n|$$

for some constant c, we say the sequence converges ***linearly***. If instead

$$|z - x_{n+1}| \leq c|z - x_n|^2,$$

then we say the sequence converges ***quadratically***. In general, we say the convergence is of ***order*** p if

$$|z - x_{n+1}| \leq c|z - x_n|^p.$$

So, for instance, Newton's method converges quadratically, and fixed point iteration converges linearly.

Using our preious theorem requires that we have an interval $[a, b]$ where the function is defined, and which also contains the image of $[a, b]$ after applying the function. This can be hard to check in practice, but we do have the following.

Theorem 12.3.
If g is continuously differentiable on $[c, d]$ and if g has a fixed point z in $[c, d]$ with $|g'(z)| < 1$, then there exists an interval $[a, b]$ contained in $[c, d]$ around z which satisfies the hypotheses of Theorem 12.2.

By our earlier theorem, we thus have that if g is continuously differentialbe, and assuming $|g'(z)| < 1$ at the fixed point, there is some interval around z such that iterates of g converge to z. In particular,

$$z - x_{n+1} \approx g'(z) \cdot (z - x_n).$$

Let's write $\mu = g'(z)$. Then, supposing x_{n-1} is "sufficiently close" to z, we also have

$$z - x_n \approx \mu(z - x_{n-1})$$

or

$$\begin{aligned} & z \approx x_n + \mu(z - x_{n-1}) \\ \implies & z - \mu z \approx x_n - \mu x_{n-1} \\ \implies & (1 - \mu)z \approx x_n - \mu x_{n-1} \\ \implies & z \approx \frac{x_n - \mu x_{n-1}}{1 - \mu}, \end{aligned}$$

and this last approximation can be rewritten as

$$\begin{aligned} z &\approx \frac{x_n - \mu x_{n-1}}{1-\mu} \\ &= \frac{x_n - \mu x_n + \mu x_n - \mu x_{n-1}}{1-\mu} \\ &= \frac{x_n(1-\mu) + \mu(x_n - x_{n-1})}{1-\mu} \\ &= x_n + \frac{\mu}{1-\mu}(x_n - x_{n-1}). \end{aligned}$$

Note, however, we don't know what μ is since $\mu = g'(z)$ and we don't know what z is. We can estimate μ, however: by the mean value theorem there eists a $c_n \in [x_{n-2}, x_{n-1}]$ such that

$$g'(c_n) = \frac{g(x_{n-1}) - g(x_{n-2})}{x_{n-1} - x_{n-2}} = \frac{x_n - x_{n-1}}{x_{n-1} - x_{n-2}}$$

and since $x_n \to z$, $g'(c_n) \to \mu$. Thus we have a sequence of approximations to μ,

$$\mu_n = \frac{x_n - x_{n-1}}{x_{n-1} - x_{n-2}}.$$

Hence

$$z \approx x_n + \frac{\mu_n}{1-\mu_n}(x_n - x_{n-1}).$$

This formula for approximationg z is known as ***Aitken's extrapolation formula***.

Let's notice that by plugging in the definition of μ_n, we can rewrite the above formula as

$$\begin{aligned} z &\approx x_n + \frac{\mu_n}{1-\mu_n}(x_n - x_{n-1}) \\ &= x_n + \frac{\left(\frac{x_n - x_{n-1}}{x_{n-1} - x_{n-2}}\right)}{1 - \frac{x_n - x_{n-1}}{x_{n-1} - x_{n-2}}}(x_n - x_{n-1}) \\ &= x_n + \frac{\left(\frac{x_n - x_{n-1}}{x_{n-1} - x_{n-2}}\right)}{\left(\frac{x_{n-1} - x_{n-2} - x_n + x_{n-1}}{x_{n-1} - x_{n-2}}\right)}(x_n - x_{n-1}) \\ &= x_n + \frac{x_n - x_{n-1}}{x_{n-1} - x_{n-2} - x_n + x_{n-1}}(x_n - x_{n-1}) \\ &= x_n + \frac{(x_n - x_{n-1})^2}{-x_n + 2x_{n-1} - x_{n-2}} \\ &= x_n - \frac{(x_n - x_{n-1})^2}{x_n - 2x_{n-1} - x_{n-2}} \end{aligned}$$

If we introduce the notation Δ to mean the difference between two successive elements in a sequence, e.g.,

$$\begin{aligned}\Delta x_n &= x_n - x_{n-1}\\ \Delta x_{n-1} &= x_{n-1} - x_{n-2}\end{aligned}$$

and let Δ^2 mean the idfference in the differences,

$$\begin{aligned}\Delta^2 x_n &= \Delta(\Delta x_n)\\ &= \Delta x_n - \Delta x_{n-1}\\ &= (x_n - x_{n-1}) - (x_{n-1} - x_{n-2})\\ &= x_n - 2x_{n-1} + x_{n-2}\end{aligned}$$

then the above approximation to z becomes

$$z \approx x_n - \frac{(\Delta x_n)^2}{\Delta^2 x_n}.$$

Notice that we can use this to construct a new method for computing a sequence which converges to a fixed point, known as ***Steffensen's method***. Given x_0, compute x_1 and x_2 as before:

$$\begin{aligned}x_1 &= g(x_0)\\ x_2 &= g(x_1).\end{aligned}$$

But now that we have three previous approximations, let's use Aitken's formula above to compute x_3:

$$x_3 = x_2 - \frac{(\Delta x_2)^2}{\Delta^2 x_2} = x_2 - \frac{(x_2 - x_1)^2}{x_2 - 2x_1 + x_0}.$$

In general, we compute two iterates of g and then compute the next iterate using Aitken's formula:

$$\begin{aligned}x_4 &= g(x_3)\\ x_5 &= g(x_4)\\ x_6 &= x_5 - \frac{(\Delta x_5)^2}{\Delta^2 x_5} = x_5 - \frac{x_5 - x_4}{x_5 - 2x_4 + x_3}\\ x_7 &= g(x_6)\\ x_8 &= g(x_7)\\ x_9 &= x_8 - \frac{(\Delta x_8)^2}{\Delta^2 x_8} = x_8 - \frac{x_8 - x_7}{x_8 - 2x_7 + x_6}\\ &\vdots\end{aligned}$$

This procedure has the effect of accelerating the rate of convergence of the sequence towards the fixed point.

Consider for example approximating the fixed point of e^{-x} – i.e., finding the x solving $x = e^{-x}$ – using fixed point iteration and Steffensen's method.

	FPI	Steffensen
x_0	1	1
x_1	0.36787	0.36787
x_2	0.69220	0.69220
x_3	0.50047	0.58222
x_4	0.60624	0.55865
x_5	0.54539	0.57197
x_6	0.57961	0.567166
x_7	0.56011	0.567130
⋮	⋮	⋮
x_{10}	0.56843	0.5671432904
x_{11}	0.566414	0.5671432904
⋮	⋮	⋮
x_{40}	0.567143905	0.5671432904
x_{41}	0.5671432904	0.5671432904

The approximations calculated using Steffensen's method stabilized after about ten iterations, but using fixed point iterations required forty!

The above values can easily be computed both ways in Matlab using a loop as follows:

```
g = @(x) exp(-x);

% Compute using fixed point iteration
x0 = 1;
for j = 1:45
    fprintf("x%d = %1.10f\n", j, x0);
    x0 = g(x0);
end

% Compute using Steffensen's method
x0 = 1;
for j = 1:15
```

```
    x1 = g(x0);
    x2 = g(x1);
    x3 = x2 - (x2 - x1)^2 / (x2 - 2*x1 + x0);
    fprintf("x%d = %1.10f\n", 3*(j-1)+1, x1);
    fprintf("x%d = %1.10f\n", 3*(j-1)+2, x2);
    fprintf("x%d = %1.10f\n", 3*(j-1)+3, x3);
    x0 = x3;
end
```

This almost seems like magic – we're doing almost exactly the same thing as before, except modifying every third element of the sequence in some odd way that came from playing around with the derivative of our $g(x)$ function, yet the sequence seems to converge much faster. Why does this happen?

To explain this, let's think geometrically. Consider the secant line of $y = g(x)$ through the points on the graph with $x = x_{n-2}$ and $x = x_{n-1}$. The points on our line are $(x_{n-2}, g(x_{n-2}))$ and $(x_{n-1}, g(x_{n-1}))$. But $g(x_{n-2}) = x_{n-1}$ and $g(x_{n-1}) = x_n$. So the points on our line are really just (x_{n-2}, x_{n-1}) and (x_{n-1}, x_n). Notice the slope of this line is

$$\frac{x_n - x_{n-1}}{x_{n-1} - x_{n-2}}$$

which is exactly our μ_n from before. Now, we're trying to approximate $g(x) = x$, which occurs when $y = g(x)$ intersects the line $y = x$. So, let's take the secant line above and solve for its point of intersection with $y = x$. To do this we just write down the equation of our secant line in point-slope form using (x_{n-1}, x_n) are the point and μ_n as the slope, and then set $y = x$:

$$\begin{aligned}
x - x_n &= \mu_n(x - x_{n-1}) \\
\implies x - x_n &= \mu_n x - \mu_n x_{n-1} \\
\implies x - \mu_n x &= x_n - \mu_n x_{n-1} \\
\implies (1 - \mu_n)x &= x_n - \mu_n x_{n-1} \\
\implies x &= \frac{x_n - \mu_n x_{n-1}}{1 - \mu_n}
\end{aligned}$$

We now rewrite the right-hand side as follows:

$$\begin{aligned} x &= \frac{x_n - \mu_n x_{n-1}}{1-\mu_n} \\ &= \frac{x_n - \mu_n x_n + \mu_n x_n + \mu_n x_{n-1}}{1-\mu_n} \\ &= \frac{(1-\mu_n)x_n + \mu_n(x_n - x_{n-1})}{1-\mu_n} \\ &= x_n + \frac{\mu_n}{1-\mu_n}(x_n - x_{n-1}) \end{aligned}$$

Of course, now we can perform the same manipulations as appeared before to obtain

$$x = x_n - \frac{(\Delta x_n)^2}{\Delta^2 x_n}.$$

That is, the third iterate we construct is simply where the secant line through the last two iterates intersects the line $y = x$.

13 Interpolation

I think it is a relatively good approximation to the truth – which is much too complicated to allow anything but approximations – that mathematical ideas originate in empirics.

John von Neumann
The Mathematician

Often we are interested in finding a function which passes through a given set of points. For instance, the function we care about may model some sort of behavior we're interested in, but isn't known exactly. However, there may be some "partial" information which is known – i.e., we may know the exact value of the function at a few special points and want to approximate the remaining unknown values of the function. This process is called ***interpolation*** and there are several different interpolation techniques, but we will start off with the simplest: polynomial interpolation, building our way up from the basics. We will ultimately show, via construction, that given any n-points in the xy-plane,

$$(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$$

where the x_i are distinct, there exists a unique polynomial of degree at most $n-1$ which passes through all n points.

13.1 Polynomial interpolation

The simplest case is when there are two points, and in that situation we are just looking for the line which passes through the two points. If our points are (x_1, y_1) and (x_2, y_2), then the slope of the line is

$$\frac{y_2 - y_1}{x_2 - x_1}$$

and so the equation of the line in point-slope form is

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1),$$

and so our polynomial is

$$\frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1) + y_1.$$

Now suppose there were three points,

$$(x_1, y_1), (x_2, y_2), (x_3, y_3).$$

The claim is that there is some polynomial of degree at most two whose graph passes through all of these points. If so, what would this polynomial be?

Suppose the polynomial were

$$Ax^2 + Bx + C.$$

Then our three points above give us three equations:

$$\begin{aligned} Ax_1^2 + Bx_1 + C &= y_1 \\ Ax_2^2 + Bx_2 + C &= y_2 \\ Ax_3^2 + Bx_3 + C &= y_3 \end{aligned}$$

Thus we have a system of linear equations whose solution will tell us what the coefficients A, B, and C are.

Example 13.1.
Find the polynomial of degree two which passes through the points

$$(1, 7),\ (2, 5),\ \textit{and}\ (3, 29).$$

If our polynomial is $Ax^2 + Bx + C$, then we must have

$$\begin{aligned} A \cdot 1^2 + B \cdot 1 + C &= 7 \\ A \cdot 2^2 + B \cdot 2 + C &= 15 \\ A \cdot 3^2 + B \cdot 3 + C &= 29 \end{aligned}$$

This is a system of linear equations, and so we can easily solve the system with some simple linear algebra. Notice that written in terms of matrices the system becomes

$$\begin{pmatrix} 1 & 1 & 1 \\ 4 & 2 & 1 \\ 9 & 3 & 1 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} 7 \\ 15 \\ 29 \end{pmatrix}$$

We take the augmented coefficient matrix of this system and can easily

put it into an echelon form

$$\left(\begin{array}{ccc|c} 1 & 1 & 1 & 7 \\ 4 & 2 & 1 & 15 \\ 9 & 3 & 1 & 29 \end{array}\right) \xrightarrow{R_2-4R_1\to R_2} \left(\begin{array}{ccc|c} 1 & 1 & 1 & 7 \\ 0 & -2 & -3 & -13 \\ 9 & 3 & 1 & 29 \end{array}\right)$$
$$\xrightarrow{R_3-9R_1\to R_3} \left(\begin{array}{ccc|c} 1 & 1 & 1 & 7 \\ 0 & -2 & -3 & -13 \\ 0 & -6 & -8 & -34 \end{array}\right)$$
$$\xrightarrow{R_3-3R_2\to R_3} \left(\begin{array}{ccc|c} 1 & 1 & 1 & 7 \\ 0 & -2 & -3 & -13 \\ 0 & 0 & 1 & 5 \end{array}\right)$$

That is, our original system is equivalent to the system

$$\begin{aligned} A + B + C &= 7 \\ -2B - 3C &= -13 \\ C &= 5 \end{aligned}$$

As $C = 5$, we can substitute into $-2B-3C = -13$ to see that $B = -1$, and then plug both of these values into $A + B + C = 7$ to determine $A = 3$.

Thus our desired polynomial is

$$3x^2 - x + 5$$

and we can easily check the graph of this polynomial does in fact go through the three points listed above.

13.2 Lagrange basis polynomials

In principle we can always find our interpolating polynomial by writing down a system of equations whose solution gives the coefficients of the polynomial and then solve the system, however this can be very tedious to do. Luckily, there's a clever trick due to the 18th century French mathematician Joseph Louis Lagrange.

Let's suppose, as in the example above, we want to find a quadratic polynomial which passes through the points $(1, 7)$, $(2, 15)$, and $(3, 29)$. Suppose, though, that we were able to find three simple quadratic polynomials, call

them L_1, L_2, and L_3 which had the following properties:

$$\begin{array}{lll} L_1(1) = 1 & L_2(1) = 0 & L_3(1) = 0 \\ L_1(2) = 0 & L_2(2) = 1 & L_3(2) = 0 \\ L_1(3) = 0 & L_2(3) = 0 & L_3(3) = 1 \end{array}$$

Notice that if we had three such quadratic polynomials, then the expression

$$7 \cdot L_1(x) + 15 \cdot L_2(x) + 29 \cdot L_3(x)$$

would also be a quadratic polynomial, and because of the properties of the L_i's listed above, the graph of this polynomial would necessarily go through the points $(1, 7)$, $(2, 15)$, and $(3, 29)$. For example, plugging 1 in for x the expression above gives

$$7 \cdot L_1(1) + 15 \cdot L_2(1) + 29 \cdot L_3(1) = 7 \cdot 1 + 15 \cdot 0 + 29 \cdot 0 = 7$$

and similarly for $x = 2$ and $x = 3$.

Now, how could we go about finding such L_i polynomials? Notice that in the case of L_1, we explicitly want $L_1(2) = 0$ and $L_1(3) = 0$. That is, we want 2 and 3 to be roots of the polynomial. This means $x - 2$ and $x - 3$ must be factors of L_1, and since L_1 has degree 2, L_1 must of the form

$$L_1(x) = k(x - 2)(x - 3)$$

for some constant k. We can easily determine what k needs to be in order for $L_1(1)$ to equal 1 as desired, however:

$$\begin{aligned} & L_1(1) = 1 \\ \implies & k(1 - 2)(1 - 3) = 1 \\ \implies & k = \frac{1}{(1 - 2)(1 - 3)} \\ \implies & k = \frac{1}{(-1)(-2)} \\ \implies & k = \frac{1}{2} \end{aligned}$$

Thus we claim $L_1(x) = \frac{1}{2}(x-2)(x-3)$ has the desired properties: $L_1(1) = 1$ while $L_1(2) = L_1(3) = 0$.

Similar computations show us that $L_2(x)$ and $L_3(x)$ must be

$$\begin{aligned} L_2(x) &= -1(x - 1)(x - 3) \\ L_3(x) &= \frac{1}{2}(x - 1)(x - 2) \end{aligned}$$

We thus arrive at the polynomial

$$\frac{7}{2}(x-2)(x-3) - 15(x-1)(x-3) + \frac{29}{2}(x-1)(x-2)$$

without having to solve any systems of equations: we instantly know what each L_i looks like because of its degree and roots, and then we do some very simple arithmetic to find the required constant.

Expanding the expression above and combining like terms, by the way, does show us that our polynomial is precisely $3x^2 - x + 5$, just as we had computed earlier.

Generalizing the process above proves the following theorem:

Theorem 13.1.
Let (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) be n points with distinct x-values. Then there exists a polynomial $p(x)$ of degree at most $n-1$ which satisfies $p(x_i) = y_i$ and the polynomial is given by

$$p(x) = y_1 L_1(x) + y_2 L_2(x) + \cdots + y_n L_n(x)$$

where

$$L_i(x) = \frac{(x-x_1)(x-x_2)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_n)}{(x_i-x_2)(x_i-x_3)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_n)}$$

The L_i polynomials which appear above are called the ***Lagrange basis polynomials***.

Written more concisely, the polynomial $p(x)$ whose graph passes through the points (x_1, y_1), ..., (x_n, y_n), where no two x_i are equal, is given by

$$p(x) = \sum_{i=1}^{n} y_i L_i(x)$$

where the Lagrange polynomials may be written as

$$L_i(x) = \frac{\prod_{k=1}^{i-1}(x-x_k) \cdot \prod_{k=i+1}^{n}(x-x_k)}{\prod_{k=1}^{i-1}(x_i-x_k) \cdot \prod_{k=i+1}^{n}(x_i-x_k)}$$

Example 13.2.
Find a polynomial of degree at most 4 whose graph passes through the points

$$(-2,3),\ (-1,7),\ (3,-2),\ (4,0)\ (9,2)$$

Plugging into the formula above we have

$$\begin{aligned}
&3\frac{(x+1)(x-3)(x-4)(x-9)}{(-2+1)(-2-3)(-2-4)(-2-9)}\\
+&7\frac{(x+2)(x-3)(x-4)(x-9)}{(-1+2)(-1-3)(-1-4)(-1-9)}\\
-&2\frac{(x+2)(x+1)(x-4)(x-9)}{(3+2)(3+1)(3-4)(3-9)}\\
+&0\frac{(x+2)(x+1)(x-3)(x-9)}{(4+2)(4+1)(4-3)(4-9)}\\
+&2\frac{(x+2)(x+1)(x-3)(x-4)}{(9+2)(9+1)(9-3)(9-4)}
\end{aligned}$$

Exercise 13.1.
What happens if we try to construct a polynomial given only its roots using the Lagrange basis polynomials above? For example, what will the above formula give us for a polynomial which passes through the points $(1,0)$, $(2,0)$, and $(3,0)$? Why does this not contradict the theorem above?

13.3 Divided differences

Consider the following problem: suppose you had n data points (x_1, y_1), ..., (x_n, y_n) and wanted a function which passed through each point, so you computed the interpolating polynomial wit the Lagrange basis polynomials as above. Later, however, you discovered there was another data point, (x_{n+1}, y_{n+1}), which your function also needed to pass through. Is there an easy way to modify our polynomial so that it passes through this new point

in addition to the previous points? Of course, we could add an extra factor to each term of the polynomial written down in terms of the Lagrange basis polynomials, though this is very tedious. What we'd like instead is a way to write down the "old" interpolating polynomial with one more termto get the "new" interpolating polynomial. This is easy to do using the *divided differences* of the function.

We will define the ***divided differences*** of a function f of numbers x_1, x_2, ..., x_n as a number denoted

$$f[x_1, x_2, ..., x_n]$$

and we will see that the interpolating polynomial for (x_1, y_1), ..., (x_n, y_n), (x_{n+1}, y_{n+1}) can be written in terms of the interpolating polynomial of (x_1, y_1), ..., (x_n, y_n), plus one more term which incorporates the divided difference $f[x_1, x_2, ..., x_{n+1}]$.

Divided differences are defined recursively: first we define the divided difference of x_1 and x_2 as

$$f[x_1, x_2] = \frac{f(x_2) - f(x_1)}{x_2 - x_1}.$$

Notice this is simply the slope of the secant line through two points on the graph $y = f(x)$.

Now, supposing $f[x_1, ..., x_{n-1}]$ has been defined, we define

$$f[x_1, ..., x_n] = \frac{f[x_2, ..., x_n] - f[x_1, ..., x_{n-1}]}{x_n - x_1}.$$

For example, consider $f(x) = x^2$. Then the divided difference of f at 1 and 2 is

$$f[1, 2] = \frac{2^2 - 1^2}{2 - 1} = \frac{3}{1} = 3.$$

The divided difference at 1, 2, and 3 is

$$\begin{aligned} f[1, 2, 3] &= \frac{f[2, 3] - f[1, 2]}{3 - 1} \\ &= \frac{\left(\frac{3^2 - 2^2}{3 - 2} - \frac{2^2 - 1^2}{2 - 1}\right)}{3 - 1} \\ &= \frac{9 - 4 - 4 + 1}{2} \\ &= 1 \end{aligned}$$

The divided difference of 1, 2, 3, and 4 is

$$\begin{aligned} f[1,2,3,4] &= \frac{f[2,3,4] - f[1,2,3]}{4-1} \\ &= \frac{\left(\frac{f[3,4]-f[2,3]}{4-2} - \frac{f[2,3]-f[1,2]}{3-1}\right)}{3} \\ &= \frac{\frac{1}{2}\left[(4^2-3^2)-(3^2-2^2)\right]-1}{3} \\ &= \frac{\frac{1}{2}\left[16-9-9+4\right]-1}{3} \\ &= \frac{5}{3} \end{aligned}$$

The next theorem, due to Newton, relates divided differences to interpolating polynomials.

Theorem 13.2 (Newton's divided difference formula)**.**
Given a list of n x-values, x_1 through x_n, where $f(x_1)$ through $f(x_n)$ are known, let $p_i(x)$ be the interpolating polynomial for $(x_1, f(x_1))$, $(x_2, f(x_2))$, ..., $(x_{i+1}, f(x_{i+1}))$. Then the interpolating polynomials are related by the following recurrence relation: $p_1(x)$ is equal to

$$p_1(x) = f(x_1) + (x - x_1)f[x_1, x_2]$$

and for each $1 < i \leq n$, we have

$$p_i(x) = p_{i-1}(x) + (x - x_1)(x - x_2) \cdot \ldots \cdot (x - x_i)f[x_1, x_2, ..., x_{i+1}].$$

Example 13.3.
Find the interpolating polynomials for the following lists of points:

(a) $(0, 2)$, $(3, 1)$

(b) $(0, 2)$, $(3, 1)$, $(-1, 4)$.

(a) We compute $p_1(x)$ as described in the theorem:

$$\begin{aligned} p_1(x) &= f(x_1) + (x - x_1)f[x_1, x_2] \\ &= f(0) + (x - 0)f[0, 3] && = 2 + x \cdot f[0, 3] \\ &= 2 + x \cdot \frac{f(3) - f(0)}{3 - 0} \\ &= 2 + x \cdot \frac{1 - 2}{3} \\ &= 2 - \frac{1}{3}x \end{aligned}$$

(b) Now that $p_1(x)$ is known, we can compute $p_2(x)$ by adding on one more term:

$$\begin{aligned} p_2(x) &= 2 - \frac{1}{3}x + (x - 0)(x - 3) \cdot f[0, 3, -1] \\ &= 2 - \frac{1}{3}x + x(x - 3) \cdot \left(\frac{f[3, -1] - f[0, 3]}{-1 - 0} \right) \\ &= 2 - \frac{1}{3}x + x(x - 3) \cdot \left(\frac{\frac{4-1}{3-(-1)} - \left(\frac{-1}{3}\right)}{-1} \right) \\ &= 2 - \frac{1}{3}x - x(x - 3) \cdot \left(\frac{3}{4} + \frac{1}{3} \right) \\ &= 2 - \frac{1}{3}x - \frac{13}{12}x(x - 3) \end{aligned}$$

13.4 Error in polynomial interpolation

Once we have our interpolating polynomial $p(x)$ which approximates $f(x)$ and agrees with $f(x)$ at x_1, x_2, ..., x_n,

$$\begin{aligned} p(x_1) &= f(x_1) \\ p(x_2) &= f(x_2) \\ &\vdots \\ p(x_3) &= f(x_3) \end{aligned}$$

a reasonable question to ask is how good of an approximation is $p(x)$? This is answered by the following theorem which gives a formula for the error in our approximation, $f(x) - p(x)$.

Theorem 13.3.
Suppose $p(x)$ is an interpolating polynomial for $f(x)$ which agrees with $f(x)$ at the points x_1, x_2, ..., x_n. Suppose also that all points x_1 through x_n are contained in some interval $[a, b]$ with $f \in C^n([a, b])$. Then for each $x \in [a, b]$ there exists a value of c in $[a, b]$ such that

$$f(x) - p(x) = \frac{\prod_{k=1}^{n}(x - x_k)}{n!} f^{(n)}(c).$$

(Notice this c depends on x.)

Proof.
For notational simplicity, write

$$\Psi(x) = \prod_{k=1}^{n}(x - x_k).$$

The claim is then that there exists some $c \in [a, b]$ such that

$$f(x) - p(x) = \frac{\Psi(x)}{n!} f^{(n)}(c).$$

Note this is obviously true if x is one of x_1, x_2, ..., x_n as both sides of the equation simply become zero. For any other $x \in [a, b]$, consider the function

$$\varphi_x(t) = f(t) - p(t) - \frac{f(x) - p(x)}{\Psi(x)} \Psi(t)$$

Note φ_x is an n-times continuously differentiable function since $f(t)$, $p(t)$, and $\Psi(t)$ are n-times continuously differentiable.

By the above, $\varphi_x(t)$ is zero if t is one of x_1, x_2, ..., x_n. Additionally, $\varphi_x(x)$ is zero. So, φ_x has $n + 1$ distinct zeros. By Rolle's theorem (or the mean value theorem), φ_x' has at least n zeros, φ_x'' has at least $n - 1$

zeros, φ_x''' has at least $n-2$ distinct zeros, and so on. In particular, $\varphi_x^{(n)}$ has at least one zero. Now consider

$$\begin{aligned}\varphi_x^{(n)}(t) &= \frac{d^n}{dt^n}\left(f(t) - p(t) - \frac{f(x)-p(x)}{\Psi(x)}\Psi(t)\right) \\ &= f^{(n)}(t) - p^{(n)}(t) - \frac{f(x)-p(x)}{\Psi(x)}\Psi^{(n)}(t).\end{aligned}$$

As p is a polynomial of degree at most $n-1$, $p^{(n)}(t) = 0$. Notice that $\Psi(t)$ is a monic polynomial of degree n:

$$\Psi(t) = \prod_{k=1}^{n}(t - x_k) = t^n + \text{ some other terms of degree less than } n$$

and so $\Psi_x^{(n)}(t) = n!$. That is,

$$\varphi_x^{(n)}(t) = f^{(n)}(t) - \frac{f(x)-p(x)}{\Psi(x)}n!.$$

But this function has some root c, and so

$$\begin{aligned}&\varphi_x^{(n)}(c) = f^{(n)}(c) - \frac{f(x)-p(x)}{\Phi(x)}n! \\ \implies &0 = f^{(n)}(c) - \frac{f(x)-p(x)}{\Phi(x)}n! \\ \implies &\frac{f(x)-p(x)}{\Psi(x)}n! = f^{(n)}(c) \\ \implies &f(x) - p(x) = \frac{\Psi(x)}{n!}f^{(n)}(c) = \frac{\prod_{k=1}^{n}(x-x_k)}{n!}f^{(n)}(c).\end{aligned}$$

□

Example 13.4.
What is the maximum error that occurs when approximating $\sin(\pi x)$ over the interval $[0, 2]$ with the interpolating polynomial through the points $(0, 0)$, $(1/2, 1)$, $(1, 0)$, $(3/2, -1)$?

First note our approximating polynomial, using the Lagrange basis

polynomials, is

$$\begin{aligned} p(x) &= 0 \cdot L_1(x) + 1 \cdot L_2(x) + 0 \cdot L_3(x) - 1 \cdot L_4(x) \\ &= \frac{(x-0)(x-1)(x-3/2)}{(1/2-0)(1/2-1)(1/2-3/2)} - \frac{(x-0)(x-1/2)(x-1)}{(3/2-0)(3/2-1/2)(3/2-1)} \end{aligned}$$

By Theorem 13.3,

$$|\sin(x) - p(x)| = \left| \frac{(x-0)(x-1/2)(x-1)(x-3/2)}{4!} \frac{d^4}{dx^4}\bigg|_{x=c} \sin(\pi x) \right|$$

for some $c \in [0, 2]$. Notice, however,

$$\left| \frac{d^4}{dx^4}\bigg|_{x=c} \sin(\pi x) \right| \leq \pi^4$$

for each c. Also notice that for each factor appearing in the numerator,

$$(x-0)(x-1/2)(x-1)(x-3/2),$$

we have that each factor is at most 2, and so

$$|(x-0)(x-1/2)(x-1)(x-3/2)| \leq 2^4 = 16$$

Hence our error is bounded above by

$$\frac{16}{24}\pi^4 = \frac{2\pi^4}{3}$$

The above thus gives us an upper bound on the error – it tells us the worst-case scenario for the error in approximating $\sin(\pi x)$ by an interpolating polynomial. It may very well be that the actual error is less than this, but in many problems simply having *an* upper bound on the error is desirable. In fact, since the x-coordinates of our points above are evenly spaced, we can do much better. Because of this even distribution, it's easy to see that for each $x \in [0, 2]$ one of the factors in

$$(x-0)(x-1/2)(x-1)(x-3/2),$$

will be at most $1/2$, one factor will be at most 1, one factor will be at most $3/2$, and one factor will be at most 2, and so we can actually determine

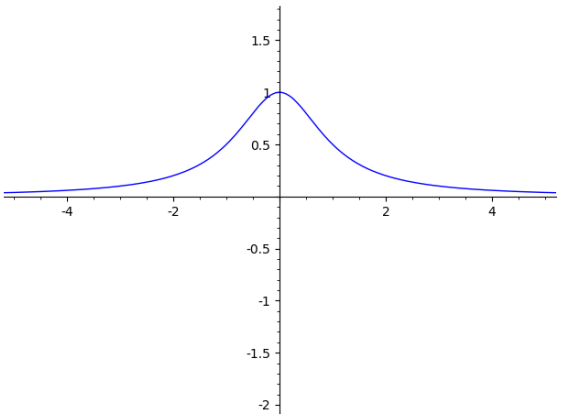
$$|(x-0)(x-1/2)(x-1)(x-3/2)| \leq \frac{1}{2} \cdot 1 \cdot \frac{3}{2} \cdot 2 = \frac{3}{2}$$

and so we can replace the upper bound in the calculation above with

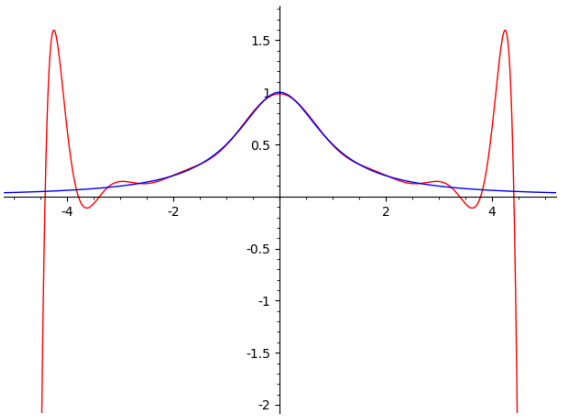
$$|\sin(x) - p(x)| \leq \frac{3/2\pi^4}{24}$$

13.5 Splines

In general, we expect that more sample points (x_1, y_1), ..., (x_n, y_n) will give us a more accurate inteprolating polynomial. However, the more points we use the higher the degree of the polynomial, and the higher the degree of the polynomial, the more it can "wiggle," especially at the ends of our list of sample points. For example, consider the function $f(x) = \frac{1}{1+x^2}$ on $[-5, 5]$.



Interpolating using sixteen equally-spaced points on $[-5, 5]$ gives a polynomial of degree fifteen which, as we see from the graph below, is a very poor approximation of the original $f(x)$ near the endpoints of the interval.



We'd like an interpolation method where something like this doesn't happen, and the trick is to use piecewise polynomial functions, each piece interpolating the function on a small interval. It is easy to construct such piecewise interpolating polynomials which have undersirable properties, such as discontinuities or places where the function is not differentiable. Usually we want our functions to be smooth, and so we will impose some extra conditions on our piecewise interpolating polynomial to make sure the result is continuous and the derivatives agree where two pieces fit together. The result of this is a function called a ***spline***

While we can define splines of any degree, we will only consider cubic splines, i.e., functions which are piecewise cubic polynomials. To be precise, a ***cubic spline*** on an interval $[a, b]$ is a function

$$S : [a, b] \to \mathbb{R}$$

together with a partition of $[a, b]$,

$$a = t_0 < t_1 < t_2 < \cdots < t_{n-1} < t_n = b,$$

such that S is a piecewise function,

$$S(x) = \begin{cases} S_1(x) & \text{if } t_0 \le x \le t_1 \\ S_2(x) & \text{if } t_1 \le x \le t_2 \\ \vdots & \\ S_n(x) & \text{if } t_{n-1} \le x \le t_n \end{cases}$$

where

- each $S_i(x)$ is a cubic polynomial,
- $S_i(t_i) = S_{i+1}(t_i)$ for $1 \le i \le n-1$,
- $S_i'(t_i) = S_{i+1}'(t_i)$ for $1 \le i \le n-1$, and
- $S_i''(t_i) = S_{i+1}''(t_i)$ for $1 \le i \le n-1$,

In particular, if we wish to interpolate (x_0, y_0), (x_1, y_1), ..., (x_n, y_n), then we want a spline defined on the interval $[x_0, x_n]$ where our partition is

$$x_0 < x_1 < \cdots < x_{n-1} < x_n$$

and we make the additional requirement that $S(x_i) = y_i$ for $0 \le i \le n$. (Notice this means $S_i(x_i) = y_i = S_{i+1}(x_i)$ for $1 \le i \le n-1$). Writing all

of this out explicitly will give us a system of linear equations with $4n - 2$ equations and $4n$ variables. Since we have two more variables than we have equations, there are two degrees of freedom in choosing a solution to the system. In order to have a unique solution we will impose two more conditions on $S(x)$. Making the assumption $S''(x_0) = 0$ and $S''(x_n) = 0$ gives us additional equations, and a spline satisfying all of these additional conditions is called a ***natural cubic spline***.

Example 13.5.
Find the natural cubic spline interpolating the following five points:

$$(1, 0),\ (2, 2),\ (3, 1),\ (4, 10),\ (5, 0).$$

Our partition here is $1 < 2 < 3 < 4 < 5$, and so we want a piecewise cubic polynomial defined on the intervals $[1, 2]$, $[2, 3]$, $[3, 4]$, and $[4, 5]$. Such a function has the form

$$S(x) = \begin{cases} Ax^3 + Bx^2 + Cx + D & \text{if } 1 \le x \le 2 \\ Ex^3 + Fx^2 + Gx + H & \text{if } 2 \le x \le 3 \\ Ix^3 + Jx^2 + Kx + L & \text{if } 3 \le x \le 4 \\ Mx^3 + Nx^2 + Px + Q & \text{if } 4 \le x \le 5 \end{cases}$$

But these polynomials must agree with the points provided, and this gives us eight equations:

$$\begin{aligned} A \cdot 1^3 + B \cdot 1^2 + C \cdot 1 + D &= 0 \\ A \cdot 2^3 + B \cdot 2^2 + C \cdot 2 + D &= 2 \\ E \cdot 2^3 + F \cdot 2^2 + G \cdot 2 + H &= 2 \\ E \cdot 3^3 + F \cdot 3^2 + G \cdot 3 + H &= 1 \\ I \cdot 3^3 + J \cdot 3^2 + K \cdot 3 + L &= 1 \\ I \cdot 4^3 + J \cdot 4^2 + K \cdot 4 + L &= 10 \\ M \cdot 4^3 + N \cdot 4^2 + P \cdot 4 + Q &= 10 \\ M \cdot 5^3 + N \cdot 5^2 + P \cdot 5 + Q &= 0 \end{aligned}$$

We also require that the derivatives of the different pieces agree at the

endpoints. Notice the derivative of $S(x)$ is

$$S'(x) = \begin{cases} 3Ax^2 + 2Bx + C & \text{if } 1 \leq x \leq 2 \\ 3Ex^2 + 2Fx + G & \text{if } 2 \leq x \leq 3 \\ 3Ix^2 + 2Jx + K & \text{if } 3 \leq x \leq 4 \\ 3Mx^2 + 2Nx + P & \text{if } 4 \leq x \leq 5 \end{cases}$$

Asking that the derivatives agree at the endpoints gives us three more equations,

$$\begin{aligned} 3A \cdot 2^2 + 2B \cdot 2 + C &= 3E \cdot 2^2 + 2F \cdot 2 + G \\ 3E \cdot 3^2 + 2F \cdot 3 + G &= 3I \cdot 3^2 + 2J \cdot 3 + K \\ 3I \cdot 4^2 + 2J \cdot 4 + K &= 3M \cdot 4^2 + 2n \cdot 4 + P \end{aligned}$$

Of course, we can rewrite these equations as

$$\begin{aligned} 3A \cdot 2^2 + 2B \cdot 2 + C - 3E \cdot 2^2 - 2F \cdot 2 - G &= 0 \\ 3E \cdot 3^2 + 2F \cdot 3 + G - 3I \cdot 3^2 - 2J \cdot 3 - K &= 0 \\ 3I \cdot 4^2 + 2J \cdot 4 + K - 3M \cdot 4^2 - 2n \cdot 4 - P &= 0 \end{aligned}$$

We also require that the second derivatives of the pieces agree at the endpoints. Notice the second derivative of $S(x)$ is

$$S''(x) = \begin{cases} 6Ax + 2B & \text{if } 1 \leq x \leq 2 \\ 6Ex + 2F & \text{if } 2 \leq x \leq 3 \\ 6Ix + 2J & \text{if } 3 \leq x \leq 4 \\ 6Mx + 2N & \text{if } 4 \leq x \leq 5 \end{cases}$$

Requiring these functions to agree when two subintervals meet gives us three more equations,

$$\begin{aligned} 6A \cdot 2 + 2B &= 6E \cdot 2 + 2F \\ 6E \cdot 3 + 2F &= 6I \cdot 3 + 2J \\ 6I \cdot 4 + 2J &= 6M \cdot 4 + 2N \end{aligned}$$

These may be rewritten as

$$\begin{aligned} 6A \cdot 2 + 2B - 6E \cdot 2 - 2F &= 0 \\ 6E \cdot 3 + 2F - 6I \cdot 3 - 2J &= 0 \\ 6I \cdot 4 + 2J - 6M \cdot 4 - 2N &= 0 \end{aligned}$$

Finally, to be a natural cubic spline require $S''(1) = 0$ and $S''(5) = 0$, and this gives us two more equations,

$$\begin{aligned} 6A \cdot 1 + 2B &= 0 \\ 6M \cdot 5 + 2N &= 0 \end{aligned}$$

All together we have a system of sixteen equations in sixteen unknowns. While tedious to do by hand, in principle this is a system we can solve by writing out the corresponding augmented coefficient matrix and putting that matrix into RREF. Doing this will tell us that the system is solved by

$$\begin{array}{llll} A = \dfrac{-1}{2} & B = \dfrac{3}{2} & C = 1 & D = -2 \\ E = \dfrac{-1}{2} & F = \dfrac{3}{2} & G = 1 & H = -2 \\ I = \dfrac{-1}{2} & J = \dfrac{3}{2} & K = 17 & L = -50 \\ M = \dfrac{-769}{2} & N = \dfrac{11535}{2} & P = -28463 & Q = 46190 \end{array}$$

We can now explicitly write down our spline:

$$S(x) = \begin{cases} \frac{-1}{2}x^3 + \frac{3}{2}x^2 + x - 2 & \text{if } 1 \le x \le 2 \\ \frac{-1}{2}x^3 + \frac{3}{2}x^2 + x - 2 & \text{if } 2 \le x \le 3 \\ \frac{-1}{2}x^3 + \frac{3}{2}x^2 + 17x - 50 & \text{if } 3 \le x \le 4 \\ \frac{-769}{2}x^3 + \frac{11535}{2}x^2 - 28463x + 46190 & \text{if } 4 \le x \le 5 \end{cases}$$

Evaluating this spline at $x = 2.5$, for example, gives us $^{33}/_{16} = 2.0625$.

14 Least Squares Approximation

All exact science is dominated by the idea of approximation.

BERTRAND RUSSELL

14.1 Motivation

We have discussed two different interpolation techniques: polynomial interpolation (which we saw three different ways to compute) and interpolation with cubic splines. Both techniques try to approximate a given function $f(x)$ by building polynomials (or piecewise polynomials) which pass through some given set of points representing known values of f. Notice this does not necessarily mean the resulting approximation is "good" away from these special interpolation points. We now turn our attention towards a similar problem: what is the "best" approximation to a given function?

To make this more precise, given a function $f(x)$ defined on an interval $[a, b]$ and a fixed integer n, which polynomial $p(x)$ of degree at most n is the "best" approximation to the function $f(x)$ in the interval $[a, b]$? To answer this question we first need some way of measuring how good an approximation is. Of course, we can consider the error in our approximation, $f(x) - p(x)$; notice this a function of x.

What should it mean for $p(x)$ to be a good approximation to $f(x)$? Of course we want the error to be small, but since the error is a function of x we have to ask *where* do we want that error to be small? It could be, for instance, the error is very small – maybe even zero! – at one point x_0, but then very large at x_1. Since the error can vary from point to point, perhaps what we should try to do is minimize the average error.

Recall that given a continuous function $g(x)$ defined on an interval $[a, b]$, the ***average value*** of g on the interval is

$$\frac{1}{b-a}\int_a^b g(x)\,dx.$$

So, the average value of the error in approximating $f(x)$ by $p(x)$ is

$$\frac{1}{b-a}\int_a^b \left(f(x) - p(x)\right)\,dx.$$

Intuitively, we want to make this average error small, but there's one issue we have to contend with. Since $f(x)-p(x)$ might be positive sometimes and negative sometimes, it could be that these positives and negatives cancel out to give us an "average error of zero" even when our function is not a particularly good approximation. As a silly example, suppose $f(x)$ was the constant function 1 on the interval $[-1,1]$, and suppose $p(x)$ was $2x+1$. This is a pretty terrible approximation to the function 1, but the average value on $[-1,1]$ is easily seen to be zero:

$$\frac{1}{1-(-1)}\int_{-1}^{1}(1-(2x+1))\,dx = \frac{1}{2}\int_{-1}^{1}-2x\,dx = \frac{-1}{2}x^2\Big|_{-1}^{1} = \frac{-1}{2}\left(1^2-(-1)^2\right) = 0.$$

To fix this issue of having positives and negatives cancel out, we'll just square the error to be certain everything is positive.

That is, given a continuous function $f(x)$, we want to find the polynomial $p(x)$ of degree at most n which minimizes the average square of the error over an interval $[a,b]$,

$$\frac{1}{b-a}\int_a^b (f(x)-p(x))^2\,dx.$$

A function minimizing this average square of the error is often called a ***least squares approximation***.

14.2 Minimizing the average square of the error with calculus

We now have a minimization problem, and so the most obvious route to solving that problem would be to use calculus. That is, given a function $f(x)$ we could explicitly compute the quantity

$$\frac{1}{b-a}\int_a^b (f(x)-p(x))^2\,dx.$$

as a function of the coefficients of the $p(x)$, differentiate that quantity and set it equal to zero to find critical points (candidates for our minima), then determine which of these candidates actually gives the smallest value. We'll see there's actually another way to solve this minimization problem, but we'll work through one example using calculus to illustrate the idea.

Example 14.1.
What polynomial of degree at most one minimizes the average square of the error in approximating $f(x) = \sqrt{x}$ on the interval $[0, 1]$?

We are looking for the polynomial $p(x) = Ax + B$ which makes $\int_0^1 \left(f(x) - p(x)\right)^2 \, dx$ as small as possible. First we compute this integral in terms of the unknown A and B:

$$\begin{aligned}
\int_0^1 \left(f(x) - p(x)\right)^2 \, dx &= \int_0^1 \left(\sqrt{x} - (Ax + B)\right)^2 \, dx \\
&= \int_0^1 \left(x - 2\sqrt{x}(Ax + B) + (Ax + B)^2\right) \, dx \\
&= \int_0^1 \left(x - 2Ax^{3/2} - 2Bx^{1/2} + A^2x^2 + 2ABx + B^2\right) \, dx \\
&= \left(\frac{x^2}{2} - \frac{4a}{5}x^{3/2} - \frac{4b}{3}x^{3/2} + \frac{A^2}{3}x^3 + ABx^2 + B^2x\right)\Bigg|_0^1 \\
&= \frac{1}{2} - \frac{4A}{5} - \frac{4B}{3} + \frac{A^2}{3} + AB + B^2 \\
&= \frac{15 - 24A - 40B + 10A^2 + 30AB + 30B^2}{30}
\end{aligned}$$

Our goal now is to find the choice of A and B which make this quantity as small as possible. Since the denominator is a constant, obviously it suffices to minimize the numerator. Let's write $G(A, B)$ for the numerator in the expression above. Notice this is a function of two variables which we need to minimize. We can do this by first finding candidates for the minima by seeing where the partial derivatives of G with respect to both A and B are equal to zero:

$$\begin{aligned}
\frac{\partial G}{\partial A} &= 0 \\
\frac{\partial G}{\partial B} &= 0
\end{aligned}$$

This gives us a system of equations which we might be able to solve. Computing $\frac{\partial G}{\partial A} = -24 + 20A + 30B$ and $\frac{\partial G}{\partial B} = -40 + 30A + 60B$. Our system of equations, after moving constants to the right-hand sides, is

thus

$$20A + 30B = 24$$
$$30A + 60B = 40$$

and solving this system will tell us $A = 4/5$ and $B = 4/15$. Thus we claim the polynomial of degree one which minimizes the square of the error in approximating $\sqrt{x}$ on $[0, 1]$ is

$$p(x) = \frac{4}{5}x + \frac{4}{15}.$$

Remark.
Technically in the example we've just found a critical point, but haven't justified that it's a global minimum. We can do this however, by verifying the surface $z = G(A, B)$ has positive curvature at every point. If the surface has positive curvature everywhere and the function has one critical point, that critical point must be a global maximum or a global minimum. The curvature of this surface is given by the determinant of the Hessian matrix of G,

$$\det \begin{pmatrix} G_{AA} & G_{AB} \\ G_{BA} & G_{BB} \end{pmatrix} = \det \begin{pmatrix} 20 & 30 \\ 30 & 60 \end{pmatrix} = 300.$$

The surface thus has constant positive curvature at every point, so our critical point is either a global maximum or a global minimum. To see which case we are in we simply compute $G_{AA} = 20$. Since this is positive the surface is "opening upwards."

14.3 Minimizing using inner products

The calculus-based procedure described above certainly works for minimizing the average square of the error, but it gets very laborious very quickly. Consider, for example, repeating Example 14.1 but for degree two or degree three polynomials.

Luckily there is another way to solve our minimization problem which is computationally much simpler, although it does require some setup.

First, some notation. Given two continuous functions f and g on an interval $[a, b]$, let $\langle f, g \rangle$ denote the real number which is computed as

$$\langle f, g \rangle = \int_a^b f(x) \cdot g(x)\, dx.$$

That is, we have a way of pairing two functions together to get a real number. Let's notice a few simple properties of this pairing we have introduced.

Lemma 14.1.
Let f, g, and h be continuous functions defined on an interval $[a, b]$ and let λ be any real number. The pairing described above then satisfies the following six conditions:

1. $\langle f, g \rangle = \langle g, f \rangle$
2. $\langle f + g, h \rangle = \langle f, h \rangle + \langle g, h \rangle$
3. $\langle \lambda f, g \rangle = \lambda \langle f, g \rangle$
4. $\langle f, f \rangle \geq 0$
5. $\langle f, f \rangle = 0$ *if and only if* f *is the constant zero function.*

Proving these six properties is really just an exercise in "unwinding" the definition of the pairing above and then using basic properties of integrals from calculus, so we won't bother to prove the lemma here.

In general, pairings satisfying the six properties above are called ***inner products*** and apply to much more general settings than just continuous functions. In multivariable calculus, for example, you learn about another inner product on the space of vectors in $\mathbb{R}^n$: the dot product. In statistics you learn about an inner product on the space of random variables defined on a sample space: the covariance[1].

While inner products may seem strange or abstract when you first learn about them, they're very handy to have. When you have an inner product

[1]Technically we need to restrict ourselves to the space of random variables with finite second moment because we want our pairings to always give us finite values.

you basically give your space (the space of continuous functions, the space of vectors in Euclidean space, the space of random variables, etc.) a geometry, and you can use that geometry to help you solve problems.

In particular, once you have an inner product you get a notion of length, which is often called a ***norm***. The norm of a continuous function f defined on $[a, b]$, is defined to be the square root of the inner product of f with itself and is denoted $\|f\|$:

$$\|f\| = \sqrt{\langle f, f \rangle} = \sqrt{\int_a^b f(x)^2 \, dx}.$$

You also get a notion of angle. The angle between two functions f and g is defined to be the value of θ satisfying

$$\cos\theta = \frac{\langle f, g \rangle}{\|f\| \, \|g\|}.$$

We say that two functions are ***orthogonal*** if the angle between them is $\pi/2$ (aka 90°). Notice when this happens $\cos\theta$ is zero, hence the numerator in the fraction above is zero and so we have proven the following simple (but very useful) lemma:

Lemma 14.2.
Two functions f and g are orthogonal if and only if

$$\langle f, g \rangle = \int_a^b f(x) \cdot g(x) \, dx = 0.$$

Maybe it seems weird to talk about the length of a function or the angle between two functions, but all that's going on is that we're generalizing the notion of length and angle from normal Euclidean geometry. The dot product between two 2-dimensional vectors $\vec{u} = (u_1, u_2)$ and $\vec{v} = (v_1, v_2)$ is defined to be

$$\vec{u} \cdot \vec{v} = u_1 v_1 + u_2 v_2,$$

and it's not hard (but it is a little tedious) to show this pairing which associates a real number to two vectors is an inner product: it satisfies the six properties described in the lemma above. Then some basic geometry shows the length of a vector $\vec{u}$ is equal to $\sqrt{u_1^2 + u_2^2}$ which is equal to $\sqrt{\vec{u} \cdot \vec{u}}$,

and $\|\vec{u}\|$ is just short-hand for that quantity. The angle between two vectors $\vec{u}$ and $\vec{v}$ can be computed using the law of cosines (placing the tails of the vectors together then filling in the triangle with $\vec{u} - \vec{v}$), and some simple manipulations will show that angle θ satisfies $\cos\theta = \frac{\vec{u}\cdot\vec{v}}{\|\vec{u}\|\,\|\vec{v}\|}$. Given these nice geometric interprations of dot products in the Euclidean plane, we *define* length and angle in more general contexts using the same formulas for whatever inner product we have. (This, by the way, is how you can do geometry in spaces of more than three dimensions: even though you can't visualize in more dimensions, you can still define geometric notions and compute them.)

Anyway, the idea is that we're going to use this geometry of the space of functions to help us solve our minimization problem. Before going any further, though, let's make a simple observation: we can always transform our problem to assume our interval is $[-1, 1]$ – we'll see why we want to do this in a minute.

In particular, suppose we perform the following change of coordinates (aka u-substitution) in our integral defining $\langle f, g\rangle$:

$$\begin{aligned} u &= \frac{2}{b-a}\cdot x - \frac{2a}{b-a} - 1 \\ du &= \frac{2}{b-a}dx \end{aligned}$$

and so

$$\begin{aligned} x &= \frac{b-a}{2}(u+1) + a \\ dx &= \frac{b-a}{2}du. \end{aligned}$$

Thus our integral

$$\int_a^b \left(f(x) - p(x)\right)^2 dx$$

above can be rewritten as

$$\frac{b-a}{2}\int_{-1}^{1}\left[f\left(\frac{b-a}{2}(u+1)+a\right) - p\left(\frac{b-a}{2}(u+1)+a\right)\right]du$$

and, even though it looks ugly,

$$p\left(\frac{b-a}{2}(u+1)+a\right)$$

is just a polynomial: it's just a polynomial in u instead of x, but it does have the same degree as our polynomial in x. So there's no loss of generality

in assuming our interval is $[-1, 1]$ since we can always transform back and forth between this and any other interval $[a, b]$.

The advantage of supposing the interval is $[-1, 1]$, though, is we can use a nice "standard" set of polynomials which can generate all other polynomials and have some nice properties.

We define the ***n-th Legendre polynomial*** $P_n(x)$ for $n = 0, 1, 2, 3, ...$ to be the following:

$$P_n(x) = \frac{1}{2^n \cdot n!} \frac{d^n}{dx^n} \left(x^2 - 1\right)^n.$$

For example,

$$\begin{aligned} P_0(x) &= \frac{1}{2^0 \cdot 0!} \frac{d^0}{dx^0} \left(x^2 - 1\right)^0 \\ &= 1 \\ P_1(x) &= \frac{1}{2^1 \cdot 1!} \frac{d}{dx} \left(x^2 - 1\right) \\ &= x \\ P_2(x) &= \frac{1}{2^2 \cdot 2!} \frac{d^2}{dx^2} \left(x^2 - 1\right)^2 \\ &= \frac{1}{2} \left(3x^2 - 1\right) \end{aligned}$$

If you don't like this definition, we can also do some algebra to show $P_n(x)$ is also given by

$$P_n(x) = \sum_{k=0}^{n} \binom{n}{k} \binom{n+k}{k} \left(\frac{x-1}{2}\right)^k.$$

There are three very important properties of these Legendre polynomials:

1. If $m \neq n$, then

$$\langle P_m, P_n \rangle = \int_{-1}^{1} P_m(x) P_n(x)\, dx = 0.$$

2. For each n,

$$\langle P_n, P_n \rangle = \frac{2}{2n+1}$$

3. Every polynomial of degree n can be written as a linear combination of $P_0(x)$, $P_1(x)$, ..., $P_n(x)$. That is, for any polynomial $p(x)$ of degree n, there exist constants λ_0, λ_1, ..., λ_n such that

$$p(x) = \lambda_0 P_0(x) + \lambda_1 P_1(x) + \cdots + \lambda_n P_n(x).$$

These properties are actually pretty hard to prove "by hand" from the definitions of the Legendre polynomials above; usually these are proven by showing the Legendre polynomials are solutions to a certain system of differential equations and using some theory about differential equations. We will take these properties for granted, and mention that these properties actually give us a very simple way to solve our minimization problem.

The idea behind what we're about to do is actually very geometric. Imagine that you have a plane in three-dimensional space and some point not on that plane. Of all the points on the plane, which one is closest to the given point? To answer this, imagine drawing a line segment from each point on the plane to the given point in space. Which line segment is shortest? It's pretty easy to convince yourself the shortest line segment will be the one that orthogonally meets the plane. This means we can find our closest point on the plane by doing orthogonal projection. This is basically what we're about to do.

Instead of three-dimensional space, we instead have the space of all continuous functions defined on the interval $[-1, 1]$. Instead of a plane, we have the the space of all polynomials of degree at most n. What we will do, then, is use our inner product $\langle f, g \rangle$ on the space of continuous functions to orthogonally project the function f onto the space of polynomials of degree at most n.

More precisely, there is some polynomial $p(x)$ which minimizes $\langle f, p \rangle$. Since every polynomial of degree at most n can be written as a linear combination of the Legendre polynomials, there is some choice λ_0, λ_1, ..., λ_n such that

$$p(x) = \sum_{i=0}^{n} \lambda_i P_i(x).$$

Our goal is to minimize

$$\int_{-1}^{1} (f(x) - p(x))^2 \, dx$$

which we can write as

$$\langle f - p, f - p \rangle .$$

However we may rewrite $p(x)$ as the linear combination above to obtain

$$\left\langle f - \sum_{j=0}^{n} \lambda_j P_j, f - \sum_{i=0}^{n} \lambda_i P_i \right\rangle.$$

We have used different indices in the sums above because we will manipulate this expression and in doing so these sums will get combined and it's helpful to keep track of which terms come from which summation.

We now use the properties of inner products from Lemma 14.1 to continue rewriting the above as

$$\begin{aligned}
&\left\langle f, f - \sum_i \lambda_i P_i \right\rangle - \left\langle \sum_j P_j, f - \sum_i \lambda_i P_i \right\rangle \\
&= \langle f, f \rangle - \left\langle f, \sum_i \lambda_i P_i \right\rangle - \left(\left\langle \sum_j \lambda_j P_j, f \right\rangle - \left\langle \sum_j \lambda_j P_j, \sum_i \lambda_i P_i \right\rangle \right) \\
&= \langle f, f \rangle - \left\langle f, \sum_i \lambda_i P_i \right\rangle - \left\langle \sum_j \lambda_j P_j, f \right\rangle + \left\langle \sum_j \lambda_j P_j, \sum_i \lambda_i P_i \right\rangle \\
&= \langle f, f \rangle \sum_i \lambda_i \langle f, P_i \rangle - \sum_j \lambda_j \langle P_j, f \rangle + \sum_j \lambda_j \left\langle P_j, \sum_i \lambda_i P_i \right\rangle \\
&= \langle f, f \rangle - \sum_i \lambda_i \langle f, P_i \rangle - \sum_j \lambda_j \langle f, P_j \rangle + \sum_j \sum_i \lambda_j \lambda_i \langle P_j, P_i \rangle \\
&= \langle f, f \rangle - 2 \sum_i \langle f, P_i \rangle + \sum_i \lambda_i^2 \langle P_i, P_i \rangle \\
&= \langle f, f \rangle + \sum_i \left(\lambda_i^2 \langle P_i, P_i \rangle - 2\lambda_i \langle f, P_i \rangle \right) \\
&= \langle f, f \rangle + \sum_i \langle P_i, P_i \rangle \left(\lambda_i^2 - \frac{2\lambda_i \langle f, P_i \rangle}{\langle P_i, P_i \rangle} \right) \\
&= \langle f, f \rangle + \sum_i \langle P_i, P_i \rangle \left(\lambda_i^2 - \frac{2\lambda_i \langle f, P_i \rangle}{\langle P_i, P_i \rangle} + \left[\frac{\langle f, P_i \rangle}{\langle P_i, P_i \rangle} \right]^2 - \left[\frac{\langle f, P_i \rangle}{\langle P_i, P_i \rangle} \right]^2 \right) \\
&= \langle f, f \rangle + \sum_i \langle P_i, P_i \rangle \left(\lambda_i^2 - \frac{2\lambda_i \langle f, P_i \rangle}{\langle P_i, P_i \rangle} + \left[\frac{\langle f, P_i \rangle}{\langle P_i, P_i \rangle} \right]^2 \right) - \sum_i \langle P_i, P_i \rangle \left[\frac{\langle f, P_i \rangle}{\langle P_i, P_i \rangle} \right]^2 \\
&= \langle f, f \rangle + \sum_i \langle P_i, P_i \rangle \left(\lambda_i - \frac{\langle f, P_i \rangle}{\langle P_i, P_i \rangle} \right)^2 - \sum_i \langle P_i, P_i \rangle \frac{\langle f, P_i \rangle^2}{\langle P_i, P_i \rangle^2} \\
&= \langle f, f \rangle - \sum_i \frac{\langle f, P_i \rangle^2}{\langle P_i, P_i \rangle} + \sum_i \langle P_i, P_i \rangle \left(\lambda_i - \frac{\langle f, P_i \rangle}{\langle P_i, P_i \rangle} \right)^2
\end{aligned}$$

Notice that $\langle f, f\rangle - \sum_i \frac{\langle f, P_i\rangle^2}{\langle P_i, P_i\rangle}$ is just some constant that doesn't depending on $p(x)$ and so doesn't change. To this we add on

$$\sum_i \langle P_i, P_i\rangle \left(\lambda_i - \frac{\langle f, P_i\rangle}{\langle P_i, P_i\rangle}\right)^2$$

which is non-negative and is minimized when each term is zero, meaning

$$\lambda_i = \frac{\langle f, P_i\rangle}{\langle P_i, P_i\rangle}.$$

So, to summarize what the above is telling us: the polynomial of degree at most n which minimizes the average square of the error in approximating $f(x)$ over $[-1, 1]$ is

$$p(x) = \lambda_o P_0(x) + \lambda_1 P_1(x) + \cdots + \lambda_n P_n(x)$$

where P_i is the i-th Legendre polynomial and

$$\lambda_i = \frac{\langle f, P_i\rangle}{\langle P_i, P_i\rangle} = \frac{\int_{-1}^{1} f(x)P_i(x)dx}{2/2i+1} = \frac{2i+1}{2}\int_{-1}^{1} f(x)P_i(x)dx.$$

Example 14.2.
Find the quadratic polynomial minimizing the average square of the error in approximating $\cos(\pi x)$ on the interval $[-1, 1]$.

By the above calculation we only need to compute λ_0, λ_1 and λ_2. Since $P_0(x) = 1$, we have

$$\begin{aligned}
\lambda_0 &= \frac{\langle \cos(\pi x), 1\rangle}{\langle 1, 1\rangle} \\
&= \frac{1}{2}\int_{-1}^{1} \cos(\pi x)dx \\
&= \frac{1}{2}\frac{\sin(\pi x)}{\pi}\Bigg|_{-1}^{1} \\
&= \frac{1}{2}\left(\frac{\sin(\pi) - \sin(-\pi)}{\pi}\right) \\
&= 0.
\end{aligned}$$

Recalling $P_1(x) = x$ we now compute

$$\lambda_1 = \frac{\langle \cos(\pi x), x \rangle}{\langle x, x \rangle} = \frac{2}{3} \int_{-1}^{1} x \cos(\pi x)\, dx.$$

Performing integrating by parts with

$$\begin{aligned} u &= x & dv &= \cos(\pi x) dx \\ du &= dx & v &= \frac{\sin(\pi x)}{x} \end{aligned}$$

we compute

$$\begin{aligned} \lambda_1 &= \frac{2}{3} \left(\frac{x \sin(\pi x)}{\pi} \Big|_{-1}^{1} - \int_{-1}^{1} \frac{\sin(\pi x)}{\pi} dx \right) \\ &= \frac{2}{3} \left(\frac{\sin(\pi) - \sin(-\pi)}{\pi} + \frac{\cos(\pi x)}{\pi^2} \Big|_{-1}^{1} \right) \\ &= \frac{2}{3} \left(0 + \frac{\cos(\pi) - \cos(-\pi)}{\pi^2} \right) \\ &= 0. \end{aligned}$$

Finally, we compute λ_2:

$$\begin{aligned} \lambda_2 &= \frac{\left\langle \cos(\pi x), \frac{1}{2}(3x^2 - 1) \right\rangle}{\left\langle \frac{1}{2}(3x^2 - 1), \frac{1}{2}(3x^2 - 1) \right\rangle} \\ &= \frac{5}{2} \int_{-1}^{1} \cos(\pi x) \cdot \frac{1}{2}(3x^2 - 1) dx \\ &= \frac{5}{2} \cdot \left(\frac{3}{2} \int_{-1}^{1} x^2 \cos(\pi x) dx - \frac{1}{2} \int_{-1}^{1} \cos(\pi x) dx \right) \\ &= \frac{15}{4} \int_{-1}^{1} x^2 \cos(\pi x) dx. \end{aligned}$$

The second integral appearing in the next-to-last step above we already know is zero from our λ_0 calculation.

To calculate this remaining integral we perform integration by parts with

$$\begin{aligned} u &= x^2 & dv &= \cos(\pi x)dx \\ du &= 2xdx & v &= \frac{\sin(\pi x)}{\pi} \end{aligned}$$

this becomes

$$\lambda_2 = \frac{15}{4}\left(\frac{x^2\sin(\pi x)}{\pi}\bigg|_{-1}^{1} - \frac{2}{\pi}\int_{-1}^{1} x\sin(\pi x)dx\right).$$

Notice

$$\frac{x^2\sin(\pi x)}{\pi}\bigg|_{-1}^{1} = \frac{\sin(\pi) - \sin(-\pi)}{\pi} = 0.$$

To compute $\int_{-1}^{1} x\sin(\pi x)dx$ we perform yet another integration by parts with

$$\begin{aligned} u &= x & dv &= \sin(\pi x)dx \\ du &= dx & v &= \frac{-\cos(\pi x)}{\pi} \end{aligned}$$

Then

$$\begin{aligned} \int_{-1}^{1} x\sin(\pi x)dx &= \frac{-x\cos(\pi x)}{\pi}\bigg|_{-1}^{1} + \frac{1}{\pi}\int_{-1}^{1}\cos(\pi x)dx \\ &= \frac{-\cos(\pi) - \cos(-\pi)}{\pi} + \frac{\sin(\pi x)}{\pi^2}\bigg|_{-1}^{1} \\ &= \frac{-(-1) - (-1)}{\pi} + \frac{\sin(\pi) - \sin(-\pi)}{\pi^2} \\ &= \frac{2}{\pi} \end{aligned}$$

Hence $\lambda_2 = \frac{-15}{\pi^2}$ and so our approximating polynomial is

$$\begin{aligned} p(x) &= 0\cdot P_0(x) + 0\cdot P_1(x) - \frac{15}{\pi^2}\cdot P_2(x) \\ &= \frac{-15}{2\pi^2}\left(3x^2 - 1\right) \end{aligned}$$

15 Numerical Integration

Michael: Homer, this is Floyd. He's an idiot savant; give him any two numbers and he can multiply them in his head, just like that!
Homer: Okay, five times nine!
Floyd: Forty-five.
Homer: Wow...

The Simpsons
Season 3, Episode 1

15.1 Motivation

In the last chapter we saw that in order to determine a least squares approximation to a function we had to compute certain integrals. The way we usually like to calculate integrals "by hand" is by finding an antiderivative and evaluating it at the endpoints of the interval we are integrating over. However, there are two main problems we have to contend with: sometimes finding an antiderivative is hard, and sometimes it's impossible.

More precisely, the fundamental theorem of calculus tells us that every continuous function has an antiderivative. However, this antiderivative is defined in terms of definite integrals. In particular, if $f(x)$ is a continuous function on an interval $[a, b]$, then the antiderivative of $f(x)$ is the function

$$F(x) = \int_a^x f(t)dt.$$

When we're lucky the function $f(x)$ is simple enough that we can essentially apply our rules for differentiation in reverse to find a simpler way of expressing this quantity. However, there are continuous functions where the antiderivative has no simpler form than the integral above. For example, the function $f(x) = e^{-x^2/2}$ is a perfectly reasonable continuous function, so the fundamental theorem of calculus says it must have an antiderivative, but we can't write that antiderivative as anything simpler than

$$F(x) = \int_{-\infty}^x e^{-t^2/2}dt.$$

This means if we want to actually evaluate the integral of $e^{-x^2/2}$ we are going to have to resort to numerically computing the integral using something like a limit of Riemann sums.

It's worth mentioning this is actually a pretty big deal: the function mentioned above is (essentially) the probability density function for the standard normal distribution, and the standard normal is extremely important in probability and statistics. The central limit theorem says that the average of a sequence of independent, identically distributed random variables (normalized in a certain way) converges to the standard normal. This is the cornerstone of all of statistics, so we actually do want to evaluate integrals like the above in many practical applications.

15.2 Riemann sums and Riemann integration

Recall from your first semester calculus class that the definite integral of a function $f(x)$ over an interval $[a, b]$ is defined as a limit of a quantity called a Riemann sum. A ***Riemann sum*** for a function $f(x)$ on an interval $[a, b]$ is a number that's computed in the following way. First we select a partition of the interval, say

$$a = x_0 < x_1 < x_2 < \cdots < x_{n-1} < x_n = b.$$

This chops the interval $[a, b]$ into n subintervals, $[x_0, x_1]$, $[x_1, x_2]$, ..., $[x_{n-1}, x_n]$. We let Δx_i denote the length of the i-th subinterval, $\Delta x_i = x_i - x_{i-1}$. We now select some point in each interval, letting x_i^* denote the point from the interval $[x_{i-1}, x_i]$. Finally, we calculate the following quantity,

$$\sum_{i=1}^{n} f\left(x_i^*\right) \Delta x_i.$$

There are several choices we made in computing this quantity: we chose a partition, and we chose a point in each subinterval determined by our partition. If I made one set of choices and you made a different set of choices, we may very well compute different numbers. However, if $f(x)$ is a continuous function, then regardless of the choices we make, in the limit our calculations we converge to the same value. To make this precise we have to be a little bit careful about exactly how we take the limit. In particular, it's not good enough to just take partitions with more elements and let the number of elements go off to infinity.

Given any partition, let's use $\mathcal{P}$ to denote the set of points:

$$\mathcal{P} = \{x_0, x_1, x_2, ..., x_n\}.$$

We then define the ***norm*** of $\mathcal{P}$, denoted $|\mathcal{P}|$, to be the maximum of the Δx_i values:

$$|\mathcal{P}| = \max\{\Delta x_1, \Delta x_2, ..., \Delta x_n\}.$$

The ***definite Riemann integral of*** f ***over*** $[a,b]$ is then the limit of Riemann sums as the norm goes to zero,

$$\int_a^b f(x)\,dx = \lim_{|\mathcal{P}|\to 0} \sum_{i=1}^{n} f(x_i^*)\Delta x_i.$$

Since this is a limit, our first question should be whether this limit exists or not. It is a theorem that we will not try to prove that provided f is continuous on the interval $[a,b]$ this limit will exist, and furthermore any sequence of partitions and points x_i^* whose norm goes to zero will give Riemann sums that converge to the Riemann integral.

Since we can use any choice of partitions and x_i^* points to compute the Riemann sums in our limit, we may as well make a convenient choice. For example, we could choose to consider partitions of n intervals of the same width, and use the right-hand endpoints of our intervals as our x_i^* points. In this case we would have each

$$\Delta x_i = \frac{b-a}{n}$$
$$x_i^* = a + i\left(\frac{b-a}{n}\right).$$

The Riemann integral is then equal to

$$\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=1}^{n} f\left(a + i\left(\frac{b-a}{n}\right)\right) \cdot \frac{b-a}{n}.$$

It is possible, but very tedious, to evaluate these types of limits "by hand," but that's not what we want to do in this class anyway. Instead, we want to use a computer to numerically approximate the integral by computing a Riemann sum (or something similar) for large values of n.

For example, we may approximate

$$\int_{-1}^{1} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx$$

using $n = 10$ subintervals above to estimate this integral is approximately

$$\sum_{i=1}^{10} \frac{1}{\sqrt{2\pi}} e^{-\left(-1+\frac{2i}{10}\right)^2/2} \frac{2}{10}.$$

This type of sum is extremely easy for us to evaluate with a loop in Matlab. We could easily create a function `integralapprox` which takes one argument, the number of subintervals `n`, and then computes the corresponding Riemann sum for $\int_{-1}^{1} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx$.

```
function rs = integralapprox(n)
    rs = 0;

    for i=1:n
        xi = -1 + 2 * i / 10;
        deltax = 2 / n;
        rs = rs + 1 / sqrt(2 * pi) * exp(-xi^2 / 2) * deltax;
    end
end
```

Evaluating `integralapprox(10)` will tell us

$$\int_{-1}^{1} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx \approx 0.6811.$$

using the Riemann sum with ten subintervals. (Interpreting the integrand above as the density of the standard normal, this means about 68.11% of random samples from a standard normal distribution fall within one standard deviation of the mean.)

15.3 Trapezoidal Riemann sums

To get better approximations to our integrals, we can either increase the number of subintervals we use, or we can find a way to getter more accurate approximations with the same number of subintervals. One way to do this would be to replace the rectangles that appear in the usual Riemann sum calculation with trapezoids.

First we recall that if a trapezoid has base B and sides with heights H_1 and H_2, then the area of the trapezoid is $B\left(\frac{H_1+H_2}{2}\right)$. Then, if we approximate the integral using trapezoids where the left- and right-hand sides of the trapezoids are x_{i-1} and x_i, so the heights are $f(x_{i-1})$ and $f(x_i)$, the area of that trapezoid is then

$$(x_i - x_{i-1}) \cdot \frac{f(x_{i-1}) + f(x_i)}{2} = \frac{f(x_{i-1}) + f(x_i)}{2} \Delta x_i$$

and so our Riemann sum is

$$\sum_{i=1}^{n} \frac{f(x_{i-1}) + f(x_i)}{2} \Delta x_i.$$

Let's notice that if the intervals all have the same width Δx_i, then we can we write out the terms of this sum to obtain

$$\frac{f(x_0) + f(x_1)}{2}\Delta x + \frac{f(x_1) + f(x_2)}{2}\Delta x + \frac{f(x_2) + f(x_3)}{2}\Delta x + \cdots \frac{f(x_{n-1}) + f(x_n)}{2}\Delta x$$
$$= \Delta x \left(\frac{f(x_0) + f(x_n)}{2} + \sum_{i=1}^{n-1} f(x_i) \right).$$

Recalling that

$$\Delta x_i = \frac{b-a}{n}$$
$$x_i = a + i\left(\frac{b-a}{n}\right)$$

we can write more specifically:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{n}\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f\left(a + i \cdot \frac{b-a}{n}\right)\right)$$

for large n. This, of course, is easy to translate into Matlab.

Just for the sake of comparing our typical "rectangular" Riemann sum with this new-and-improved trapezoidal Riemann sum, we note that the integral

$$\int_0^1 \left(x^2 + 2x + 2\right) dx$$

has a true value of $\frac{4}{3} = 3.3333....$ When approximated with ten rectangles of equal width, we compute 3.4849, but when approximated with ten trapezoids of equal width we get 3.3350. This have relative errors of -0.0435 and -0.0005, respectively. I.e., the trapezoidal sum produced a *much* better approximation than the rectangular sum.

15.4 Quadratic approximations / Simpson's rule

Notice that the trapezoidal Riemann sum we computed above is exactly the same as the integral of the piecewise linear function interpolating $f(x)$ at

the points $(x_0, f(x_0))$, $(x_1, f(x_1))$, $(x_2, f(x_2))$, ..., $(x_n, f(x_n))$. To get better approximations, then, it seems reasonable to consider higher-order piecewise polynomials interpolating these points. The next thing to consider, then, is piecewise quadratic approximations.

To be precise, once we have picked our partition

$$a = x_0 < x_1 < x_2 < \cdots < x_{n-1} < x_n = b,$$

we want to find a piecewise quadratic function, where the domain of each quadratic is one of our subintervals $[x_{i-1}, x_i]$, whose graph passes through $(x_{i-1}, f(x_{i-1}))$ and $(x_i, f(x_i))$. Of course, there are infinitely-many quadratics that pass through two given points, so we need an additional point in each of our intervals to single out a nice choice of quadratic. (I.e., three points determine a quadratic, just like two points determine a line.)

To single out our quadratic we will ask that on the interval $[x_{i-1}, x_i]$ we pass through not only $(x_{i-1}, f(x_{i-1}))$ and $(x_i, f(x_i))$, but also a third point which we will denote $(m_i, f(m_i))$. This m_i will be the midpoint between x_{i-1} and x_i,

$$m_i = \frac{f(x_{i-1}) + f(x_i)}{2}.$$

That is, on the interval $[x_{i-1}, x_i]$ we approximate our function with the following interpolating polynomial, written using the Lagrange basis polynomials:

$$\frac{(x-m_i)(x-x_i)}{(x_{i-1}-m_i)(x_{i-1}-x_i)}f(x_{i-1})+\frac{(x-x_{i-1})(x-x_i)}{(m_i-x_{i-1})(x_{i-1}-x_i)}f(m_i)+\frac{(x-x_i)(x-m_i)}{(x_i-x_{i-1})(x_i-m_i)}f(x_i)$$

We then approximate the integral of $f(x)$ over this interval by the integral of this quadratic,

$$\begin{aligned}\int_{x_{i-1}}^{x_i} \Bigg(&\frac{(x-m_i)(x-x_i)}{(x_{i-1}-m_i)(x_{i-1}-x_i)}f(x_{i-1})+ \\ &\frac{(x-x_{i-1})(x-x_i)}{(m_i-x_{i-1})(x_{i-1}-x_i)}f(m_i)+ \\ &\frac{(x-x_i)(x-m_i)}{(x_i-x_{i-1})(x_i-m_i)}f(x_i)\Bigg)dx.\end{aligned}$$

This is of course a simple integral to calculate, but we will introduce some notation to make it a little bit easier.

First we relabel our x- and y-coordinates of our interpolating points as follows:

$$\begin{aligned} a &= x_{i-1} & A &= f(a) \\ b &= m_i & B &= f(b) \\ c &= x_i & C &= f(c) \end{aligned}$$

Notice with this notation $b = \frac{a+c}{2}$.

We now have

$$\begin{aligned} &\int_a^c \left(\frac{(x-b)(x-c)}{(a-b)(a-c)} A + \frac{(x-a)(x-c)}{(b-a)(b-c)} B + \frac{(x-a)(x-b)}{(c-a)(c-b)} C \right) dx \\ =&\frac{A}{(a-b)(a-c)} \int_a^c (x-b)(x-c)dx + \\ &\frac{B}{(b-a)(b-c)} \int_a^c (x-a)(x-c)dx + \\ &\frac{C}{(c-a)(c-b)} \int_a^c (x-a)(x-b)dx. \end{aligned}$$

Each of these integrals is then very easy to compute:

$$\begin{aligned} &\frac{A}{(a-b)(a-c)} \int_a^c (x-b)(x-c)dx \\ =&\frac{A}{(a-b)(a-c)} \int_a^c \left(x^2 - (b+c)x + bc\right) dx \\ =&\frac{A}{(a-b)(a-c)} \left(\frac{x^3}{3} - \frac{(b+c)x^2}{2} + bcx \right) \Bigg|_a^c \\ =&\frac{A}{(a-b)(a-c)} \left(\frac{c^3}{3} - \frac{(b+c)c^2}{2} + bc^2 - \frac{a^3}{3} + \frac{(b+c)a^2}{2} - abc \right) \\ =&\frac{A}{(a-b)(a-c)} \left(\frac{c^3 - a^3}{3} + (b+c)\left(a^2 - c^2\right) + bc(c-a) \right). \end{aligned}$$

Using the fact $b = \frac{a+c}{2}$, this all simplifies down to

$$\frac{A(c-a)}{6}.$$

We similarly compute

$$\frac{C}{(c-a)(c-b)} \int_a^c (x-a)(x-b)dx = \frac{C(c-a)}{6}$$

The middle term is similar,

$$\frac{B}{(b-a)(b-c)}\int_a^c (x-a)(x-c)dx = \frac{2B(c-a)}{3}.$$

Putting all of this together, our integral above is

$$\frac{A(c-a)}{6} + \frac{2B(c-a)}{3} + \frac{C(b-a)}{6} = \frac{c-a}{6}\left(A+4B+C\right).$$

Or, in terms of our x_i coordinates,

$$\frac{x_i - x_{i-1}}{6}\left(f(x_{i-1}) + 4f\left(\frac{x_{i-1}+x_i}{2}\right) + f(x_i)\right)$$

Adding these integrals of quadratic pieces together to approximate the original integral gives

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} \frac{b-a}{6n}\Bigg(f\left(a + \frac{(b-a)(i-1)}{n}\right) + 4f\left(a + \frac{(b-a)(i-1)}{2n}\right) + f\left(a + \frac{(b-a)i}{n}\right)\Bigg)$$

Remark.
The approximation to the integral above has a division by $6n$ where n is one less than the number of points. This is assuming we introduce new points m_i inbetween our existing x_i points, and so the n in this expression equals the number of subintervals over which we have found a quadratic interpolating polynomial.

Using Matlab to evaluate this sum to approximate the integral

$$\int_0^1 \left(x^2 + 2x + 2\right) dx$$

with ten subintervals of equal width is accomplished with

```
>> sum = 0;
>> f = @(x) x^2 + 2*x + 2;
>> for i=1:10
     sum = sum + f((i-1)/10) + 4 * f((2*i-1)/20) + f(i/10);
   end;
>> sum = 1/60 * sum;
>> disp(sum);
   3.3333
```

We can simplify our formula for the sum slightly by adopting the convention that the endpoints of our interpolation intervals are the points with an even index. I.e., our intervals for the quadratic interpolations are

$$[x_0, x_2], [x_2, x_4], [x_4, x_6], ...$$

and then take the points with an odd index to be the midpoints of our intervals. That is, $x_1 = \frac{x_0+x_2}{2}$ is the midpoint for $[x_0, x_2]$; $x_3 = \frac{x_2+x_4}{2}$ is the midpoint for $[x_2, x_4]$; and so on. In doing this we will suppose that n is an even number. The number of subintervals is then $\frac{n}{2}$, and the sum of our integrals of the quadratic approximations then becomes

$$\begin{aligned}
&\frac{b-a}{6\frac{n}{2}}\left(f(x_0)+4f(x_1)+f(x_2)\right)\\
+&\frac{b-a}{6\frac{n}{2}}\left(f(x_2)+4f(x_3)+f(x_4)\right)\\
+&\frac{b-a}{6\frac{n}{2}}\left(f(x_4)+4f(x_5)+f(x_6)\right)\\
+&\cdots\\
+&\frac{b-a}{6\frac{n}{2}}\left(f(x_{n-2})+4f(x_{n-1})+f(x_n)\right)
\end{aligned}$$

Combining like-terms we may write this as

$$\begin{aligned}
&\frac{b-a}{3n}\left(f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+\cdots+2f(x_{n-2})+4f(x_{n-1})+f(x_n)\right)\\
=&\frac{b-a}{3n}\left(f(x_0)+f(x_n)+\sum_{i=i}^{n/2}4f(x_{2i-1})+\sum_{i=1}^{n/2-1}2f(x_{2i})\right)
\end{aligned}$$

This expression is sometimes referred to as ***Simpson's rule***.

Remark.
Notice in the expression above the n that appears still refers to the highest index which appears, x_n, which is one less than the number of points. However, because we have half as many subintervals as did compared to our earlier expression (which had denominator $6n$) the denominator is half as large.

15.5 Error in numerical approximations

As always, once we have an algorithm for producing approximations of quantities we care about, we want some way to understand how "good" our approximations are. That is, we want to know the error in the approximation.

Let's begin by analyzing the error in approximating an integral by our trapezoidal Riemann sums. For notation convenience, let's denote the trapezoidal sum in approximating $\int_a^b f(x)dx$ with n trapezoids all of which have the same width as $T_n(f;a,b)$. That is,

$$T_n(f;a,b) = \frac{b-a}{n}\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f\left(a + \frac{i(b-a)}{n}\right)\right).$$

Theorem 15.1.
If f is a twice continuously differentiable function defined on the interval $[a,b]$, then there exists some c in $[a,b]$ such that

$$\int_a^b f(x)dx - T_n(f;a,b) = \frac{-(b-a)^3}{12n^2} f''(c).$$

Even though we don't know what c is and so can't compute the above quantity directly, we can use this to get a bound on the error by bounding $f''(x)$ on the interval $[a,b]$.

Example 15.1.
What is an upper bound for the largest possible absolute value of the error in approximating $\int_0^1 e^{-x^2}dx$ using n trapezoids of equal width.

Here, our $f(x)$ is e^{-x^2}, $a = 0$, and $b = 1$. We're trying to find an upper bound for

$$\left| \int_0^1 e^{-x^2}dx - T_n\left(e^{-x^2}; 0, 1\right) \right|$$

By our theorem we know there exists some $0 \leq c \leq 1$ such that this quantity equals

$$\left| \frac{-(1-0)^3}{12n^2} f''(c) \right| = \frac{1}{12n^2}|f''(c)|,$$

so we need to try to find an upper bound for $|f''(c)|$. Notice

$$\begin{aligned} f(x) &= e^{-x^2} \\ f'(x) &= -2xe^{-x^2} \\ f''(x) &= 4x^2e^{-x^2} - 2e^{-x^2} \\ &= 2e^{-x^2}\left(2x^2 - 1\right) \end{aligned}$$

We need to find the extrema of $f''(x)$ on $[0, 1]$. First we find the critical points of this function:

$$\begin{aligned} &\frac{d}{dx} 2e^{-x^2}\left(2x^2 - 1\right) = 0 \\ \implies &2(-2x)e^{-x^2}(2x^2 - 1) + 2e^{-x^2} \cdot 4x = 0 \\ \implies &8xe^{-x^2} - 4xe^{-x^2}(2x^2 - 1) = 0 \\ \implies &4xe^{-x^2}(2 - (2x^2 - 1)) = 0 \\ \implies &4xe^{-x^2}(3 - 2x^2) = 0 \end{aligned}$$

This is solved when either factor is zero, and we either have $4xe^{-x^2} = 0$ which means $x = 0$, or when $3 - 2x^2 = 0$ which means $x = \pm\sqrt{\frac{3}{2}}$. Notice, however, $-\sqrt{\frac{3}{2}} < 0$ and $\frac{3}{2}/>1$, so our only critical points are $x = 0$ and the other endpoint, $x = 1$. Evaluating $f''(x)$ at these

endpoints shows us $f''(0) = -2$ and $f''(1) = 2e$. Hence $|f''(x)| < 2e$ for all $x \in [0, 1]$.

Thus the error in our trapezoidal approximation satisfies

$$\left| \int_0^1 e^{-x^2} dx - T_n(e^{-x^2}; 0, 1) \right| < \frac{2e}{12n^2} = \frac{e}{6n^2} < \frac{3}{6n^2} = \frac{1}{2n^2}$$

So, for example, using $n = 10$ trapezoids of equal width we know the error in our approximation is no more than

$$\frac{1}{2 \cdot 10^2} = \frac{1}{200} = 0.005.$$

Example 15.2.
How many trapezoids of equal width are required to approximate $\int_0^1 e^{-x^2} dx$ to within one one-billionth of the true value?

Using our error formula in the previous example, we know that for n trapezoids the error is at most $\frac{1}{2n^2}$. Hence we need to find the n that guarantees $\frac{1}{2n^2} < \frac{1}{10^9}$, but this is simply arithmetic:

$$\begin{aligned} & \frac{1}{2n^2} < \frac{1}{10^9} \\ \implies & 10^9 < 2n^2 \\ \implies & n^2 > \frac{10^9}{2} = 5 \times 10^8 \\ \implies & n > \sqrt{5 \times 10^8} \approx 22360.6797 \end{aligned}$$

Thus we require at least 22361 trapezoids for the error to be less than one one-billionth.

We have a similar theorem for the error in approximating an integral using Simpson's rule / quadratic interpolation. We will let $S_n(f(x); a, b)$ denote the approximation to $\int_a^b f(x)dx$ given by Simpson's rule

$$S_n(f; a, b) = \frac{b-a}{3n} \left(f(x_0) + f(x_n) + \sum_{i=i}^{n/2} 4f(x_{2i-1} + \sum_{i=1}^{n/2-1} 2f(x_{2i}) \right).$$

Theorem 15.2.
If f is a four-times continuously differentiable function, then there exists a constant $c \in [a, b]$ such that

$$\int_a^b f(x)dx - S_n(f(x); a, b) = \frac{-(b-a)^5}{180n^4} f^{(4)}(c).$$

Example 15.3.
What is an upper bound for the absolute value of the error in approximating $\int_0^1 e^{-x^2}$ by Simpson's rule with n quadratic pieces?

The theorem above tells us

$$\left| \int_0^1 e^{-x^2} dx - S_n(e^{-x^2}; 0, 1) \right| = \frac{1}{180n^4} \left| f^{(4)}(c) \right|$$

for some $0 \leq c \leq 1$. Notice that the fourth derivative of our function is

$$f^{(4)}(x) = 16x^4 e^{-x^2} - 48x^2 e^{-x^2} + 12e^{-x^2} = (16x^4 - 48x^2 + 12)e^{-x^2}.$$

The critical points of this function are given by setting the fifth derivative equal to zero. As the fifth derivative is

$$f^{(5)}(x) = -8e^{-x^2} x(4x^4 - 20x^2 + 15),$$

this is zero when either $x = 0$ or when $4x^4 - 20x^2 + 15 = 0$. This second equation can be thought of as a quadratic: setting $t = x^2$ the equation becomes $4t^2 - 20t + 15 = 0$, which can be solved with the quadratic formula to obtain

$$\frac{5}{2} \pm \sqrt{\frac{5}{2}}.$$

As $x = t^2$, x is then the positive or negative square root of these; this

gives four roots:

$$\sqrt{\frac{5}{2}+\sqrt{\frac{5}{2}}} \approx 2.0202$$
$$-\sqrt{\frac{5}{2}+\sqrt{\frac{5}{2}}} \approx -2.0202$$
$$\sqrt{\frac{5}{2}-\sqrt{\frac{5}{2}}} \approx 0.9586$$
$$-\sqrt{\frac{5}{2}-\sqrt{\frac{5}{2}}} \approx -0.9586$$

Of course, all but one of these critical points is outside our interval $[0,1]$. This means there are three critical points (candidates for maxima or minima) which we need to evaluate:

$$f^{(4)}(0) = 12$$
$$f^{(4)}\left(\sqrt{\frac{5}{2}-\sqrt{52}}\right) \approx -7.4195$$
$$f^{(4)}(1) \approx -7.3576$$

Of course, $|f^{(4)}(x)|$, among points in $[0,1]$, is then maximized at $x=0$. This means the error in our approximation is bounded above by

$$\frac{12}{180n^4} = \frac{1}{15n^4}.$$

Example 15.4.
How large must n be to ensure the absolute value of the error in approximating $\int_0^1 e^{-x^2}dx$ with Simpson's rule is less than one one-billionth.

Using the upper bound obtained in the last example, we see

$$\begin{aligned} & \frac{1}{15n^4} < \frac{1}{10^9} \\ \implies & 10^9 < 15n^4 \\ \implies & n^4 > \frac{10^9}{15} \\ \implies & n > \sqrt[4]{\frac{10^9}{15}} \approx 90.36. \end{aligned}$$

Thus we reuquire that n be at least 91 to approximate $\int_0^1 e^{-x^2} dx$ to within one one-billionth of the true value using Simpson's rule.

A

Installing and running Matlab

In M-371 we will be using Matlab for all of our examples and programming exercises, so you will need to have access to Matlab to follow along in the notes and do the required homework. In this appendix we describe how to install Matlab on your own computer, using the information from the UITS Knowledge Base article *Download, install, or update Matlab*, https://kb.iu.edu/d/ajmh. You will need at least 2GB to install Matlab on your computer.

Indiana University has a site license for Matlab, meaning that all students may download and install Matlab on their personal computer at no expense. Matlab may be downloaded and installed on any compatible Windows, Mac, or Linux computer, but first you must obtain a license code. The first step in installing Matlab on your computer is to obtain a license code for your operating system.

Depending on the operating system you are using, follow one the links below to be directed to an UITS page which will request that you login, accept the terms and conditions of the Matlab license, and then click a button obtain a product key. This is a twenty-five digit code, broken up by hyphens into five groups of five. You may eventually need to copy and paste this key into Matlab, so either leave this page open or save the key somewhere.

Windows
: If you are going to install Matlab on a Windows computer, visit https://iuware.iu.edu/Windows/Title/3528.

Mac
: If you are going to install Matlab on a Mac, visit https://iuware.iu.edu/Mac/Title/3528.

Linux
: If you are going to install Matlab on a Linux machine, visit https://iuware.iu.edu/Linux/Title/3523.

Visit the MathWorks site https://www.mathworks.com/mwaccount and create an account *using your IU email address*, then visit the MathWorks License Center, https://www.mathworks.com/licensecenter/licenses. You will be prompted to enter the product key obtained in the step above. After entering the product key, visit the MathWorks download page https://www.mathworks.com/downloads/web_downloads to download Matlab.

In these notes we will be assuming you are using Matlab version R2018b, but any older or newer version is probably fine.

After downloading, run the installer. For Windows and Mac users this should be just like installing any other application. For Linux users, you will need to move the tarball you downloaded to either `/usr/local/src` or `/opt` depending on your distribution. Once moved to one of those directories, open a terminal, change to the directory you've placed the tarball in, and execute the following commands:

```
sudo tar xf matlab_linux_<release>.tgz
cd MATHWORKS_<release>
sudo ./install
```

where `<release>` depends on the version you downloaded, e.g. `<release>` is probably something like `R2018b`.

Regardless of which operating system you're using, once the installer begins you will need to log into the MathWorks account you made earlier, then select the license for Matlab.

When you start Matlab for the first time you may be asked to Activate Matlab, but this should amount to just select a checkbox and clicking 'Next'.