Math 321:
Introduction to Mathematical Software

David Koslicki, PhD

MTH 321: Introduction to Mathematical Software

Catalog Description: An examination of select mathematical software packages to support problem solving. Topics include using computational resources to solve basic numerical and symbolic problems in mathematics, visualization and presentation of data, creation of simple programming scripts, and applications of basic programming techniques to promote mathematical understanding. The scientific typesetting language LaTeX will also be covered. All courses used to satisfy MTH prerequisites must be completed with a C- or better.

Credits: 3

Meets: Two 80 minute classes meeting weekly for 10 weeks in a computer laboratory.

Terms offered: F,S

Enforced Prerequisites: MTH 252 and either MTH 341 or 306.

Course Content: This course is designed to familiarize students with the use of software resources commonly utilized in the mathematical sciences. Students will learn how to use modern computing environments such as MATLAB and Mathematica for the purpose of symbolic and numerical problem solving and visualization. Students will become acquainted with the syntax and usage of each system through computer-aided lectures as well as through projects. The relative merits and disadvantages of each system will also be discussed. Basic programming paradigms and concepts will be introduced where appropriate. This course will also introduce LaTeX, the de facto standard for the communication and publication of mathematical and scientific documents.

Learning Resources: In lieu of a text, relevant information will be posted on blackboard: http://www.my.oregonstate.edu. (All materials will be updated to canvas when appropriate.)

Learning Outcomes for MTH 482: A successful student in MTH 399 will be able to use MATLAB and Mathematica to:

1. Solve basic numerical and symbolic mathematics problems.
2. Visualize and present data.
3. Create simple programming scripts and functions.
4. Understand and apply basic programming techniques and paradigms

In addition, a successful student will be able to typeset and communicate mathematical results using LaTeX.

Evaluation of Student Learning: The grading of the course is on a points-based system:

Homework:	300 points
Project:	100 points
Total:	400 points

Students with Disabilities: Accommodations are collaborative efforts between students, faculty and Disability Access Services (DAS). Students with accommodations approved through DAS are responsible for contacting the faculty member in charge of the course prior to or during the first week of the term to discuss accommodations. Students who believe they are eligible for accommodations but who have not yet obtained approval through DAS should contact DAS immediately at 541-737-4098. Consult http://ds.oregonstate.edu/home.

Student Conduct: All students are expected to obey OSU's student conduct regulations. In particular, students are expected to be honest and ethical in their academic work. Academic dishonesty is defined as an intentional act of deception in one of the following areas:
Cheating - use or attempted use of unauthorized materials, information or study aids;
Fabrication - falsification or invention of any information;
Assisting - helping another commit an act of academic dishonesty;
Tampering - altering or interfering with evaluation instruments and documents;
Plagiarism - representing the words or ideas of another person as one's own.
If evidence of academic dishonesty is found, University procedures will be followed, including the assignment of a grade of "F" for the guilty parties. For more information about academic integrity and the University's policies and procedures in this area, visit the Student Conduct web site at
http://studentlife.oregonstate.edu/studentconduct/offenses-0

Syllabus
Introduction to Mathematical Software
Math 399, 3 credits
Fall Quarter, 2014

Prerequisite:
MTH 252 and MTH 341 or 306.

Course Content:
This course is designed to familiarize students with the use of software resources commonly utilized in the mathematical sciences. Students will learn how to use modern computing environments such as MATLAB and Mathematica for the purpose of symbolic and numerical problem solving and visualization. Students will become acquainted with the syntax and usage of each system through computer-aided lectures as well as through projects. The relative merits and disadvantages of each system will also be discussed. Basic programming paradigms and concepts will be introduced where appropriate.
This course will also introduce LaTeX, the *de facto* standard for the communication and publication of mathematical and scientific documents.

Learning Outcomes:
A successful student in MTH 399 will be able to use MATLAB and Mathematica to:

1. Solve basic numerical and symbolic mathematics problems.
2. Visualize and present data.
3. Create simple programming scripts and functions.
4. Understand and apply basic programming techniques and paradigms

In addition, a successful student will be able to typeset and communicate mathematical results using LaTeX.

Meeting times:
Tuesday and Thursday, 10:00am-11:20am, KIDD 033.

Text:
In lieu of a text, relevant information will be posted on blackboard:
http://www.my.oregonstate.edu

Professor:
Dr. David Koslicki ("cause-lick-ee")
Office: KIDD 354
Phone: 514-737-5172
Email: david.koslicki@math.oregonstate.edu
Office Hours: By appointment and
T, R: 8-9am
No appointments are required during office hours.

Homework:
Homework assignments (along with their due dates) are contained in the course notes posted on blackboard. The due date for each homework assignment will typically be 3-7 days after it is assigned. Please attend class and keep an eye on the blackboard website for the homework assignment due dates. There will be at least 10 homework assignments, with the lowest two scores being dropped.

Exams/Projects:
In lieu of exams/quizzes, you will complete a final project. This project will consist of taking a math, science, or data related problem and writing a Matlab or Mathematica program to solve it. You will then summarize your results (including figures) in a LaTeX document and submit this (along with the source code) during finals week.
Briefly, you will be graded on correctness and completeness of the source code (including how well commented the code is), along with how well written and presented the LaTeX document is.
There are a couple of example projects posted on blackboard. You may either choose to perform one of these projects, or come up with your own. Please make an appointment with me (before dead week) to discuss what you project will consist of.
Some other project ideas include: personal analytics (via location-based apps, email frequency analysis, personal genomics, etc.), constructing an interactive Fractal plotting program, investigating cellular automata, bioinformatics applications (eg. visualization of de Bruijn graphs).
I am more than happy to look at your code/LaTeX document and give feedback about your project (but I will not be making any meetings during finals week, so start working on this early!).

Grading Policy:
The grading of the course is on a points-based system:

Homework:	300 points
Project:	100 points
Total:	400 points

I reserve the right to alter this grading scheme at the end of the term, but only to your advantage.

A	369-400 points
A-	360-368 points
B+	342-359 points
B	323-341 points
B-	320-328 points
C+	311-319 points
C	280-310 points
D	240-279 points
F	000-239 points

Help:
Make a note of my office hours. I will be glad to help you whenever I can - just ask!

Your classmates are an important resource. There are many locations on campus for study groups to meet and work. The Mathematics Learning Center is an excellent place to study and meet with your classmates. Be sure you are working together towards understanding the solution, not just "getting the answer".
Tutoring (free of charge) is available in the Mathematics Learning Center (Kidd 108) and in the Library.

Disability Access Services:

Accommodations are collaborative efforts between students, faculty and Disability Access Services (DAS). Students with accommodations approved through DAS are responsible for contacting the faculty member in charge of the course prior to or during the first week of the term to discuss accommodations. Students who believe they are eligible for accommodations but who have not yet obtained approval through DAS should contact DAS immediately at 541-737-4098. More information is available at http://ds.oregonstate.edu.

Academic Dishonesty:

Students are expected to be honest and ethical in their academic work. Academic dishonesty is defined as an intentional act of deception in one of the following areas:

Cheating - use or attempted use of unauthorized materials, information or study aids;
Fabrication - falsification or invention of any information;
Assisting - helping another commit an act of academic dishonesty;
Tampering - altering or interfering with evaluation instruments and documents;
Plagiarism - representing the words or ideas of another person as one's own.

If evidence of academic dishonesty is found, University procedures will be followed, including the assignment of a grade of “F” for the guilty parties. For more information about academic integrity and the University's policies and procedures in this area, visit the Student Conduct web site at http://www.orst.edu/admin/stucon/achon.htm.

Part 1: Matlab

Introduction to MATLAB

Table of Contents

David Koslicki

Oregon State University

8/27/2014

Preliminary

1. Starting up matlab
2. Layout (command window, workspace, command history, current folder, editor)
3. Using the editor (control flow, saving, executing cells, saving, loading, running)

Basics

MATLAB (short for Matrix Laboratory) is a numerical computing environment / Computer Algebra System and programming language originally released in 1984. MATLAB is one of the premier numerical computing environments and is widely used in academia as well as industry. One of the reasons for its widespread usage is the plethora of built-in functions and libraries/packages for common numerical routines.

While MATLAB is rather expensive, there exists a "clone" called GNU Octave. There are only a few slight syntactical differences between MATLAB and Octave. While Octave is free and open source, MATLAB does have a more robust library/package environment and is generally faster than Octave. Interestingly, Octave is named after Octave Levenspiel, an emeritus chemical engineering professor at Oregon State University.

Let's jump right in to using MATLAB. MATLAB is only on the Windows machines, and should be located under the "Math Software" portion of the start menu. When you first start up, you should notice a few different windows, namely, the **Current Folder**, **Workspace**, **Command Window**, **Command History**, and **Editor** windows. We will first concentrate on the **Command Window**.

You may enter commands at the command line prompt ">>". Let's start with the simplest command possible

```
1+1
```

You'll notice that the answer has been printed to your screen. This answer is also stored in the global variable *ans*. This variable will store whatever the previous output to your screen was. Note also that this variable now shows up in your **Workspace** window. This window contains all the variables that have been created in your MATLAB session along with some other information about them (like the value of the variable, how much memory it takes up, etc).

There are a couple different ways to delete a variable. One way is to use

```
clear('ans')
```

Another is to click on the variable *ans* in the **Workspace** window and hit delete (or right mouse click and select delete). All the variables may be cleared by using

```
clear
```

And the command window can be cleared by using

```
clc
```

To quit your matlab session, you can either just close the window, or else type the command *exit*.

There is a built-in help menu for matlab functions that can be accessed by typing *help* and then a search term. For example

```
help lsqnonneg
```

More extensive information can also be obtained by putting your cursor over the function of interest, and then hitting F1. In my opinion, the help documentation is rather poorly done in MATLAB itself, but the internet help documentation is much better. For example, more extensive help information on the *lsqnonneg* function can be found at http://www.mathworks.com/help/matlab/ref/lsqnonneg.html

MATLAB has all the standard "calculator" commands including basic arithmetic operations.

```
1+2*(3/4-5/6)^2
```

As usual, the order of operations is

1. Brackets ()
2. Powers ^
3. *,/ (left to right)
4. +,- (left to right)

Be careful, the operator \ is very different from /!

While it might appear that MATLAB only gives a few significant figures, this is just a style choice on their part. You can view more digits by using the *format* command

```
format short
1/3
format long
1/3
```

You can see that more digits are displayed using the long format. We will see later how to precisely control the number of digits to be displayed.

There are a few special numbers/constants in MATLAB. A few of the important ones are

```
pi
i % square root of negative one
j % square root of negative one
eps % The smallest number MATLAB can represent
NaN % Means Not A Number. Used to handle things like 0/0
Inf % Infinity
```

Variables

We have already seen one variable (namely, *ans*). Variables are key components of any programming language and are used to store numbers (or any other kind of data structure). Let's define our first variable

```
my_sum = 1+2
```

Notice that the value of this variable has been printed to the screen. If you want to suppress this effect, end the expression with a semi-colon

```
my_sum = 1+2;
```

There are a number of variable naming conventions that you will want to follow.

- Pick a name that indicates what is stored in that variable. It is much clearer to use the name *TotalCost* rather than *x*.
- Don't use names that are too long or too short. While the variable *a* is uninformative, the variable *Tesing_raster2DCinput2DCalg2evaluatePressure_LIF2_input_Flatten* is probably too long.
- Pick a convention for multi-word variable: *TotalCost* (Camel Case) or *total_cost* (Snake case)
- Pick a convention for capitalization.
- Don't use a built-in function as a variable name. Basically, take some time to think up your personal naming convention and stick with it.

MATLAB has some other restrictions on variable names. For example, a variable must start with a letter, is case sensitive, cannot include a space, and can be no longer than a fixed maximum length. To find this maximum length, enter the command

```
namelengthmax
```

You should also avoid using any of the special characters (as listed above). I.e. Don't use *n!*, but rather use *n_factorial*. This can lead to issues when using *i* as an index, so be careful. For better or for worse, the built in constants can be changed, so be **extra** careful.

```
pi
pi=14;
pi
```

Built in functions

MATLAB has a plethora of built in functions. A non-exhaustive list can be found at http://www.mathworks.com/help/matlab/functionlist.html. Some of the more common functions can be found using

```
help elfun
```

which will list most of the elementary mathematical functions. A handy cheat sheet can be found at https://engineering.purdue.edu/~bethel/MATLAB_Commands.pdf.

Script Files

You'll soon get tired of using the command window to enter all your commands, so it's good practice to use the *editor* instead. The editor can be brought up by typing the command

```
edit
```

into the command window. When the window opens, type the following commands into the window (It's ok if you don't know what all the commands mean)

```
x=linspace(0,2*pi,200);
y=sin(x);
plot(x,y)
axis tight
xlabel('x-axis')
ylabel('y-axis')
title('The graph of y=sin(x).')
```

Save the file in your directory as *sineplot.m* The extension *.m* tells Matlab that you can run this file as a script. There are a number of different ways to run a script. The first is to make sure that the file is in your directory (using *addpath* or *cd*) and then typing

```
sineplot
```

Or else, you can use *F5*, or the *editor+run* option on the top menu. For quick prototyping, it's convenient to use two percent signs *%%* to break your code up into individual cells. The cells can then be evaluated individually by placing your cursor in that cell, and then pressing *ctrl+enter*. You can abort the evaluation of a command by pressing *ctrl+C*.

ICE

1. Create a variable Speed_MPH to store a speed in miles per hour (MPH). Convert this to kilometers per hour (KPH) by using the conversion formula $KPH = 1.60934MPH$.
2. Create a variable Temp_F to store a temperature in Fahrenheit (F). Convert this to Celsius (C) by using the conversion formula $C = \frac{5}{9}(F - 32)$.
3. Wind chill factor (WCF) measures how cold it feels when the air temperature is *T* (in Celsius) and the wind speed is *W* (in kilometers per hour). The expression is given by $WCF = 13.12 + 0.6215T - 11.37V^{0.16} + 0.3965T * V^{0.16}$. Calculate the wind chill factor when the temperature is *50F* and the wind speed is *20mph*.
4. Find a **format** option that would result in the command *1/2+1/3* evaluating to *5/6*.
5. Find MATLAB expressions for the following $\sqrt{2}$, $3^{1.2}$, and $\tan(\pi)$.
6. There is no built-in constant for *e* (Euler's constant). How can this value be obtained in MATLAB?

7. What would happen if you accidentally assigned the value of *2* to *sin* (i.e. *sin=2*)? Could you calculate *sin(pi)*? How could you fix this problem?

8. Say you have a variable *iterator*. Let *iterator=7*. How could you increment the value of this iterator by one unit? Is there only one way to do this?

9. Random number generation is quite useful, especially when writing a program and the data is not yet available. Programmatic random number generation is not truly random, but pseudo-random: they take an input (called a seed) and generate a new number. Given the same random seed the same number will be generated. However, the numbers that are produced are "random" in the sense that each number is (almost) equally likely. Use the function *rand* to generate a random number. Now close out MATLAB, open it up again, enter *rand*, and record the number. Repeat this process. Did you notice anything? If you want a truly different number to be generated each time, you need to use a different seed. The command *rng('shuffle')* will use a new seed based on the current time. Open the help menu for *rng* and explore the different kinds of random number generators.

Vectors

As its name suggests, MATLAB is designed to primarily be a matrix-based numerical calculation platform. Hence, most of its algorithms for linear algebra (and matrix-based calculations in general) are highly optimized. Today we will explore how to define and work with vectors and matrices.

There are two kinds of vectors in MATLAB: row vectors and column vectors. Row vectors are lists of numbers separated by either commas or spaces. They have dimensions *1xn*. Here are a few examples

```
v1=[1 3 pi]
v2=[1,2,3,4]
```

Column vectors, on the other hand, have dimensions *nx1* and are separated by semi-colons

```
v3=[1;2;3]
v4=[i;2i;1+i]
```

It's very important to keep clear the differences between row vectors and column vectors as a very common MATLAB issue is to accidentally use a row vector where you meant to use a column one (and vice-versa). To determine what kind of vector you have, you can use the *size()* command.

```
size(v1)
size(v2)
size(v3)
```

Additionally, you can use the *whos* command.

```
whos v1
```

The size command lets you store the dimensions of the vector for later usage, while the *whos* command gives you a bit more information.

```
[m,n] = size(v1)
m %number of rows
n %number of columns
```

The command *length()* gives the size of the largest dimensions of a vector (or matrix). Be careful with this unless you are absolutely sure which dimension (the rows or the columns) is going to be larger.

```
v1 = [1 2 3];
v2 = [1;2;3];
length(v1)
```

```
length(v2)
```

Now say we wanted to perform some basic operations on these vectors. For example, say we wanted to add two of the vectors together.

```
v5=[sqrt(4) 2^2 1];
v1 + v5
```

Note what happens when we try to do the following

```
v1 + v2
```

or

```
v1 + v3
```

The error **Matrix dimensions must agree** in this case is an indication that you are trying to add vectors of different dimensions.

Other arithmetic operations can be performed on vectors

```
v1 - v5
2*v1
```

However, we have to be careful when using division, for example, note what happens when you enter

```
[2 4 6]/[2 2 2]
```

Why didn't we get the answer *[1 2 3]*? The operation / is a very special MATLAB operator which we will cover later. If you want to perform an element-wise operation on a pair of vectors, use the following notation

```
[2 4 6]./[2 2 2]
```

The same is true of the * and ^ operators

```
[2 4 6].*[2 2 2]
[2 4 5].^[2 2 2]
```

To be consistent, MATLAB *should* use the operations .+ and .- to designate vector addition and subtraction, but they chose not to for stylistic reasons.

To access individual elements of a vector, round braces are used.

```
v = [18 -1 1/2 37];
v(1)
v(2)
```

This can be used to modify individual elements of a vector. For example if you write

```
v = [1 1 1 1 1 1];
```

You can change individual elements using assignment

```
v(2) = 13;
v
v(5) = -1;
v
```

This permanently changes the corresponding element. Just be sure that you specify the correct number of elements. In this case *v(4)* is a single element, so while you can use

```
v(4) = 77;
```

you cannot do

```
v(4) = [1 2 3];
```

MATLAB will automatically make your vector longer if necessary when changing elements in this way

```
v = [1 2 3];
v(4) = 18;
v
```

You can also build vectors up from existing ones using concatenation.

```
u = [1 2 3];
v = [4 5 6];
w = [u v]
```

Once again, be careful with the dimensions of the vectors.

```
u = [1 2 3];
v = [4; 5; 6];
w = [u v] %Won't work
```

To convert a row vector into a column vector (and the other way around too) you can use the *transpose* command. There is a bit of subtlety here as there are two different kinds of transposes: *transpose* and *conjugate transpose*. Transpose simply interchanges the dimensions of a vector and is denoted .'

```
v = [1 2 3]
vt = v.'
size(v)
size(vt)
```

The conjugate transpose ' is used to interchange the dimensions of a vector, as well as take the conjugate of any imaginary entries

```
v = [1 2 i]
vct = v' % Note the i has been replaced with -i
size(v)
size(vct)
```

It is *very* common for programmers to use ' when they actually mean .', but thankfully this is only a concern if you are going to be using imaginary numbers. Just be aware of this.

Colon Notation

To aid in the rapid creation of vectors, MATLAB includes special notation called *colon notation*. It can create vectors, arrays of subscripts, and iterators. The notation is *start:increment:end* with the *increment* part being optional. This is basically like creating a table of numbers.

```
1:10
1:2:10 %Note that the end, 10, isn't included due to the increment size.
-10:0
```

As mentioned, this can be used to access entire portions of a vector (called: using an array of subscripts).

```
v = [1 -1 2 -2 3 -3 4 -4 5 -5];
v(1:2)
v(1:2:length(v))
v(2:2:length(v))
```

There is a shorthand for the last two commands: if you are using colon notation to access elements of a vector, you can use the *end* command to denote the end of the vector.

```
v(1:2:end)
v(2:2:end)
v(end-2:end)
```

Note that all of MATLAB's vector notation is *1*-based. So the first element of a vector is *1* (as opposed to *0* as in some other programming languages (such as Python)).

The colon notation and transpose can be combined to create column vectors as well.

```
(1:.1:2)' %Note that 1:.1:2' results in a row vector as 2' is the same as 2
```

There are two different kinds of subscripting in MATLAB: subscripts and logical indexing. We have already seen subscripts:

```
v = [pi exp(1) 1/2 -9 0];
```

To access the first and third element using subscripts we can use

```
v([1 3])
```

Another way to do this is to use *logical indexing*. That is, treating *1* as *include* and *0* as *exclude* we can access the first and third elements of v in the following way too

```
v(logical([1 0 1 0 0]))
```

This can actually be very handy in selecting certain kinds of elements from a vector. For example, say we wanted all the strictly positive elements from v

```
v>0
```

This command returns a logical vector indicating which elements are strictly positive. To retrieve these, we can use this to index our vector v

```
v(v>0)
```

The command *find* will return the subscripts of the *1*'s in a logical vector

```
find(v>0)
```

Vectorization

One of the most powerful aspects of MATLAB is its ability to **vectorize** calculations. That is, given a vector of numbers, MATLAB can perform the same operation on each one in a very efficient fashion. For example if

```
v = [1 2 3];
```

then

```
sin(v)
```

applies *sin* to each element of *v*. In many cases, MATLAB uses special optimization "tricks" to make this kind of operation incredibly quick. Say we wanted to compute *sin* of the first *10^8* integers. This could be accomplished by taking *sin* of *1*, then *sin* of *2* etc:

```
tic; %starts a "stopwatch" and will tell us how long the calculation takes
for i=1:10^7 %starting at 1, going up to 10^7
    sin(i); %calculate sin(i)
```

```
end;
loop_timing = toc %stop the "stopwatch"
```

However, when we vectorize this, you can notice it is much faster

```
tic;
sin(1:10^7);
vectorize_timing = toc
```

Note how many times faster it is

```
loop_timing / vectorize_timing
```

It takes practice and experience to recognize when you can vectorize code, but it is typically the number 1 way to speed up your MATLAB code.

Summation

Lastly, the *sum()* command will add up all the elements in a given vector:

```
x=[1 2 3 4];
sum(x)
```

This function is remarkably efficient, and you can easily sum up a few million random numbers in this fashion.

```
x=rand(1,10000000);
sum(x);
```

ICE

1. How would you use the colon operator to generate the following vector: *[9 7 5 3 1]*?
2. Use the *length()* command to find the length of the following vectors: *x=1:0.01:5*, *y=2:0.005:3*, *z=(100:0.05:200)'*
3. Use Matlab's *sum()* command and vector colon notation to construct the sum of the first 1000 natural numbers.
4. Use Matlab's *sum()* command and vector colon notation to find the sum of the fist 1,000 odd positive integers.
5. Let $u = [1, 2, 3]$ and $v = [4, 5, 6]$. Use Matlab to compute $(u + v)'$ and $u' + v'$. Compare the results. Vary u and v. What have can you observe about these two expressions?
6. Let $v = [1, 3, 5, 7], w = [5, 2, 5, 4]$. Think about the output of each of the following commands; write down by hand what you think the answer will be. Next, run the commands in Matlab and check your results. Did they match? $v > w,\ v >= w,\ v <= w,\ v < w,\ v == w,\ v \sim= w$.
7. The empty vector is denoted *[ ]*. This can be used to delete elements of a vector. Create a vector *v* of length 5 and delete the first element by using *v(1)=[ ]*. What happens when you evaluate v(1)? What about v(5)?
8. Read the documentation for the *primes()* command. Use that command to find the sum of the first 100 primes. Create a histogram of the prime gaps (differences between successive primes) for all primes less than or equal to 10,000,000.

9. Another handy function is *linspace(x,y,n)* which gives a row vector with *n* elements linearly (evenly) spaced between *x* and *y*. Using this function, make a row vector of 100 evenly spaced elements between 1 and 0.

10. Using vectorization, estimate the limit $\lim_{x \to 0} \frac{\sin(x)}{x}$.

11. Generate a list of 10,000 random numbers uniformly between 0 and 1. Approximate the average value of $\frac{1}{x+1}$ on $[0, 1]$. Does this answer make sense? (Hint: use Calculus).

Plotting

The plotting capabilities of MATLAB are designed to make their programmatic generation relatively easy. However, this means that sometimes plots take a little fiddling before they come out exactly how you want them to. The MATLAB *plot()* command can be called in a couple of different ways. In particular, *plot(x,y)* creates a 2D line plot with *x* specifying the x-coordinates of the data contained in *y*. For example

```
x = [1 2 3 4];
y = [2 4 6 8];
plot(x,y)
```

Note that MATLAB automatically connects up the data points; this can be changed by using

```
plot(x,y,'.') %The '.' tells MATLAB to treat the data as points
```

Here's an example of plotting a parabola

```
x = linspace(-10,10,100);
y = x.^2;
plot(x,y)
```

If we were to omit the second argument to *plot(x,y)*, then MATLAB would treat the x-axis as *1:length(y)*

```
y=[2 4 6 8];
plot(y)
```

You may have noticed that each time we call *plot()* the previous plot we had is overwritten. Different figures can be created by using the *figure* command

```
figure
plot(-10:.5:10,(-10:.5:10).^3)
figure
plot(0:.01:(2*pi),sin(0:.01:(2*pi)))
```

To combine two or more plots, use

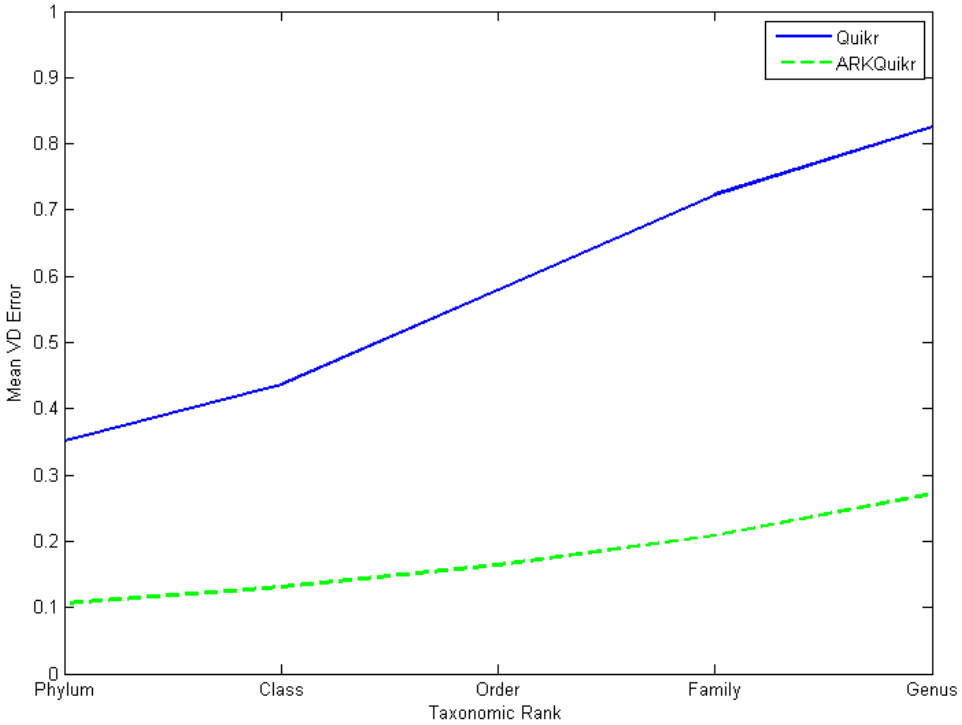
```
figure
x=0:.01:1;
plot(x,2*x+1)
hold on
plot(x,3*x+1)
```

If you want the plot to be different colors, use a single plot command. If you would like to label your axes (always a good idea), you can do this usin *xlabel* and *ylable*. The easiest way to do this is to call these functions immediately after using the *plot()* command.

```
plot(x,2*x+1,x,3*x+1);
legend('2x+1','3x+1')
xlabel('x')
ylabel('y')
title('My plot')
```

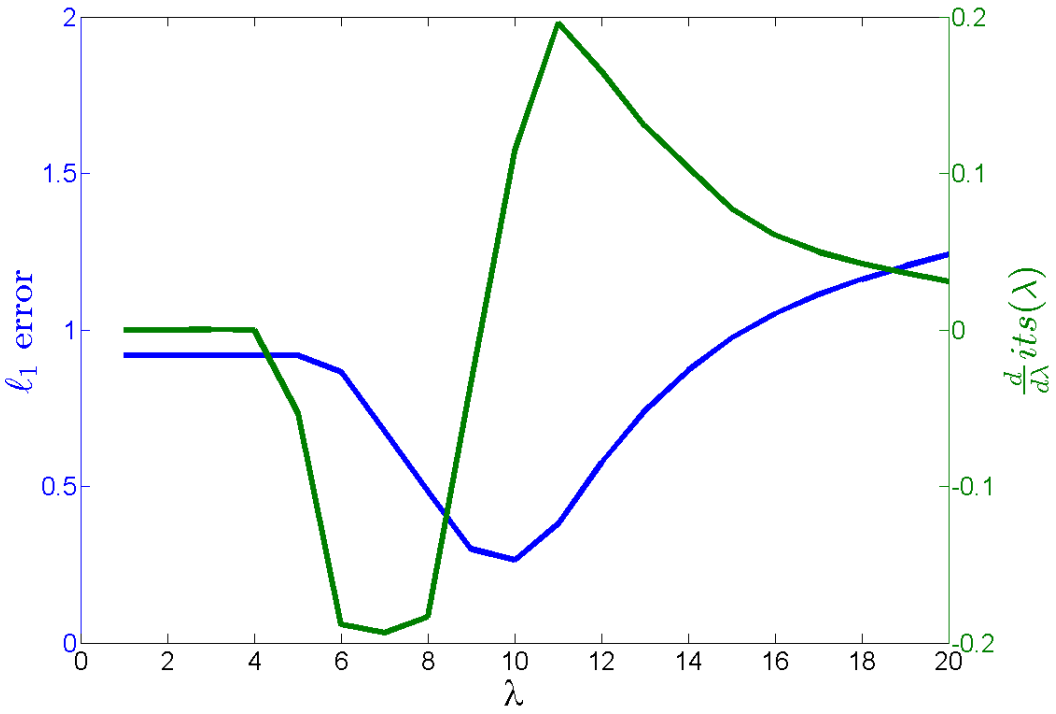
Let's look at a few more complicated examples of plots

```
meansQuikr = load('meansQuikr.mat'); %load in the data
meansQuikr = meansQuikr.meansQuikr; %get the correct variable
meansARKQuikr = load('meansARKQuikr.mat');
meansARKQuikr = meansARKQuikr.meansARKQuikr;
figure %create a new figure
plot(meansQuikr(1:5),'-b','LineWidth',2); %dashed blue thicker plot
ylim([0,1]) %set the y-axis limits
hold on
plot(meansARKQuikr(1:5),'--g','LineWidth',2);
ylim([0,1])
legend('Quikr','ARKQuikr')
%set the x-tick labels
set(gca,'XTickLabel',{'Phylum','Class','Order','Family','Genus'})
set(gca,'XTick',[1 2 3 4 5 6]) %set the xticks
xlabel('Taxonomic Rank')
ylabel('Mean VD Error')
ylim([0,1])
```




```
myerror = load('error.mat') %load the data
myerror = myerror.error; %Get the proper variable
```

```
[AX,H1,H2]=plotyy(1:20,myerror(1:20),1:20,diff(myerror(1:21)));
set(AX,'fontsize',20)
set(get(AX(1),'Ylabel'),'String','$\ell_1$ error','interpreter','latex',...
    'fontsize',30)
set(get(AX(2),'Ylabel'),'String','$\frac{d}{d\lambda} its(\lambda)$',...
    'interpreter','latex','fontsize',30)
set(H1,'LineWidth',5)
set(H2,'LineWidth',5)
xlabel('\lambda','fontsize',30)
```



ICE

1. When explicitly specifying the x-coordinates of a plot, it is important to keep in mind the trade-off between speed and accuracy. Say you want to plot the function *sin(3*pi*x)* for *0<x<1*. Use *x=linspace(0,1,N)* and *plot(x, sin(3*pi*x))*. Compare the results for various values of *N*.

2. Generate a list of the first 100 powers of 2. The following function selects the first digit of a number x: $\left\lfloor \frac{x}{10^{\lfloor \log_{10}(x) \rfloor}} \right\rfloor$. Plot a histogram of the first digits of 1,000 powers of 2. Use this to estimate the probability that the first digit of a power of 2 is *1*. What is the probability that the fist digit is *2*? and so on. This phenomena is referred to as *Benford's Law*: For *most* numbers, the probability that the first digit is *d* is given by $\log_{10}\left(1+\frac{1}{d}\right)$. Plot this function and compare this plot to the results we obtained. Interestingly, Benford's law is periodically used to check for fraud on tax returns.

For the following 6 exercises, set *x=linspace(a,b,n)* for the given values of *a*, *b*, and *n*. Calculate *y=f(x)* for the given function *f(x)*. Plot with *plot(x,y,s)* for the given formatting string *s*. Explain the result of the formatting string.

1. $f(x) = \sin(x),\ a = 0,\ b = 4\pi,\ n = 24,\ s = \texttt{'rs-'}$.
2. $f(x) = \arccos(x),\ a = -1,\ b = 1,\ n = 24,\ s = \texttt{'mo'}$.
3. $f(x) = 2^x,\ a = -2,\ b = 4,\ n = 12,\ s = \texttt{'gd:'}$.
4. $f(x) = \sinh(x),\ a = -5,\ b = 5,\ n = 20,\ s = \texttt{'kx--'}$.
5. $f(x) = \log_{10}(x),\ a = 0.1,\ b = 10,\ n = 20,\ s = \texttt{'c--'}$.
6. $f(x) = x^2,\ a = -5,\ b = 5,\ n = 20,\ s = \texttt{'b*-.'}$.

Graph the given function on a domain that shows all the important features of the function (intercepts, extrema, etc.), or on the domain specified. Use enough points so that your plot appears smooth. Label the horizontal and vertical axis with Matlab's *xlabel()* and *ylabel()* commands. Include a *title()*.

1. $y = x^2 - 2x - 3$
2. $y = x^3 - 3x^2 - 28x + 60$
3. $y = xe^{-x^2}$ on $[-5, 5]$
4. $y = \frac{1}{1+x^2}$ on $[-10, 10]$
5. $y = (x^2 - 2x - 3)e^{-x^2}$ on $[-5, 5]$

Matrices

Matrices are the bread and butter of MATLAB, and so as you might imagine, there are myriad matrix and matrix-related functions in MATLAB. Let's start with the very basics.

At its most basic, a matrix is a rectangular arrow of numbers with *m* rows and *n* columns. This is referred to as an *m x n* matrix. There are a few different ways to enter a matrix into MATLAB, but most methods are "row based":

```
A = [1 2 3; 4 5 6]
```

This is a *2x3* sized matrix. This can be verified by using the *size()* command

```
[m,n] = size(A)
```

Note that the *length()* command gives the maximum of *m* and *n* for the command *[m,n] = size(A)*.

Here, *m* and *n* will store the number of rows and number of columns respectively of *A*.

Just like with row vectors, the elements of a row of *A* can be separated with a comma instead of a whitespace

```
A = [1, 2, 3; 4, 5, 6]
```

The colon notation can be used to create matrices as well

```
B = [1:5; 6:10; 11:2:20]
```

Just be careful that you do not create a "jagged" matrix. Each row needs to have the same number of columns and conversely as well.

```
C = [1 2 3; 4 5] %This will lead to an error
```

Again, similarly to row/column vectors, the ′ symbol will transpose a matrix.

```
A' %This is the conjugate transpose, but all the entries are real
size(A')
```

Indexing of matrices

Matlab has two different kinds of indexing for matrices: *linear indexes* and *subscripts*. Subscript notation is the typical notation most mathematicians are used to. For example, to refer to the element in the second row and third column of the matrix A (that is, A_{23}), we can use

```
A(2,3)
```

Colon notation can again be used to indicate elements of A. Here's how to get the first column

```
A(:,1)
```

Here's how to get the second row

```
A(2,:)
```

Here's how to get the first square submatrix of A

```
A(1:2,1:2)
```

And the second square submatrix of A

```
A(1:2,2:3)
```

To understand linear indexing, imagine that you were counting the individual elements of the matrix *A* column-wise. In this case, you would have the following

```
A(1,1) %First element
A(1,2) %Second element
A(2,1) %Third element
A(2,2) %Fourth element
A(3,1) %Fifth element (Not Milla Jovovich :) )
A(3,2) %Sixth element
```

Using linear indexing, we can refer to these very same elements using

```
A(1)
A(2)
A(3)
A(4)
A(5)
A(6)
```

Though it doesn't seem like it, linear indexing can actually be quite handy. For example, you can easily get all the elements in a matrix (in essence "flattening it out") by using the command

```
A(:)
```

Just make sure you don't ask for an element of a matrix that isn't there

```
[m,n]=size(A);
A(m+1,n) %won't work
A(m,n+1) %won't work
A(m*n+1) %won't work
```

Matrix Assignment

Since we now know how to access individual elements of a matrix, we can easily change them just like we did for vectors.

```
A = [1 2 3; 4 5 6];
A(1,1) = -7;
A(4) = 100;
A
```

You can also change entire rows or columns via

```
A = [1 2 3; 4 5 6];
A(1,:) = [-100 -10 -1];
A
```

Just be careful that the dimensions match between what you are trying to replace in the matrix and what you are replacing it to. I.e.

```
length(A(1,:))
length([-100 -10 -1])
```

the sizes match, so it will work. However, we can't do

```
A(:,1) = [10 100 1000]; %Won't work
```

since

```
length(A(:,1))
```

does not equal

```
length([10 100 1000]);
```

Special Matrices

MATLAB has a bunch of built-in ways to generate a variety of matrices of a given size. Let's run through a few of them

```
m=3; %Number of rows
n=4; %Number of columns
A = ones(m,n) %A matrix of all ones
B = zeros(m,n) %A matrix of all zeros
 %An identity matrix (or more precisely, a matrix with ones down the main
 %diagonal)
C = eye(m,n)
```

Matlab has a few shorthands that can speed the creation of matrices. One of these includes creating a square matrix of dimensions *n x n* when using the following commands

```
D = eye(10);
size(D)
```

The command *diag* can be used to extract the diagonal of a matrix, as well as used to create a diagonal matrix. For example:

```
E = [1 0 0; 0 2 0; 0 0 3];
```

Then to get the diagonal, we can use.

```
Ediag = diag(E)
```

Now to create a diagonal matrix, say with diagonal entries 10, 100, and 1000, we could use

```
Fdiag = [10 100 1000];
F = diag(Fdiag)
```

Note that the matrix is automatically of square of size *length(Fdiag)*.

Element-wise matrix operations

Just like with element-wise vector operations, we can do all the typical element-wise matrix operations.

```
A = rand(2)
B = rand(2)
A + B %Element-wise addition
A - B %Element-wise subtraction
A .* B %Element-wise multiplication
A ./ B %Element-wise division
A .^ B %Element-wise power
```

Note that you can also do the above operations when the second argument to your binary operator is not a matrix, but a scalar

```
A = rand(2)
B = 5
A + B %Element-wise addition
A - B %Element-wise subtraction
A .* B %Element-wise multiplication
A ./ B %Element-wise division
A .^ B %Element-wise power
```

Again, watch out for dimensions mismatches

```
A = rand(2);
B = rand(3);
A + B %This won't work.
```

Recalling the *sum()* command, you will notice that it operates on the *columns* of a matrix. That is, *sum(A)* will return a vector where each entry is the sum of the corresponding column in *A*.

```
A=[1 2; 3 4];
sum(A)
```

You can change this to work on the rows by using the following command

```
sum(A,2)
```

ICE

1. Create a *5x8* matrix of all zeros and store it in a variable. Now replace the second row in the matrix with all 4's and the first column with the entries *[2 4 6 8 10]*.

2. Create a *100 x 100* matrix of random reals in the range from -10 to 10. Get the sign of every element. What percentage of the elements are positive?

In the following exercises, predict the output of the given command, then validate your response with the appropriate Matlab command. Use *A=magic(5)*.

1. $A > 14$
2. $A <= 12$
3. $(A > 3)\ \&\ (A <= 20)$
4. $(A < 5)\ \&\ (A >= 21)$
5. $\sim (A <= 6)$
6. $(A > 6)\ \&\ \sim (A > 8)$

Matrix/Vector and Matrix/Matrix Operations

(Insert short review of elementary linear algebra operations here)

Matrix/vector and matrix/matrix operations are extremely efficient in MATLAB, and a common programming technique to is reduce whatever problem you have at hand into such an operation. But first, we need to cover how these operations work in MATLAB.

First, we will cover the definition of the product of a matrix with a vector: *A*x*. This is only defined for a column vector that is of the same length as the number of columns in the matrix. I.e. the following two quantities need to be the same

```
A = rand(10,12);
[m,n]=size(A);
x = rand(12,1);
n == length(x)
```

We imagine the matrix *A* as being made up of *m* row vectors stacked vertically, and then we take the element-wise product of one of these rows with *x*, then sum up this vector. Let's look at an example

```
A = [4 6 8; 1 -1 2];
x = [2; -1; 1];
```

The first entry of *A*x* will be *4*2 + 6*(-1) + 8*1 = 10*. The second entry will be *1*2 + (-1)*(-1) + 2*1 = 5*. Indeed, we have

```
A*x
```

Once again, we need to be very careful that the dimensions match up

```
A = rand(10,12);
x = rand(10,1);
A*x %Won't work
```

```
A = rand(10,12);
x = rand(1,12);
A*x %Won't work
```

Matrix/Matrix products (*A***B*) are very similar, except that they view the matrix *B* as a bunch of column vectors sitting side by side. Here, the inner dimensions of *A* and *B* need to match

```
A = [1 2 3; 4 5 6];
B = [1 -1; 2 3; 0 1];
A*B
```

More on dimensions:

```
rand(10,11)*rand(11,12) %will work
rand(10,11)*rand(12,12) %Won't work
rand(1,5)*rand(5,1) %Will work
```

Systems of Linear Equations

Many applied mathematics problems can be reduced to (or at least approximated by) a system of linear equations. A system of linear equations can be expressed in terms of a matrix of coefficients *A*, a vector of constants *b*, and a vector of variables *x*. Then the system

$$a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n = b_1$$

$$\vdots$$

$$a_{m,1}x_1 + a_{m,2}x_2 + \ldots + a_{1,n}x_n = b_m$$

can be expressed compactly as *Ax*=*b*. Every such system either has no solution, exactly one solution, or infinitely many solutions. When *A* is nonsingular (i.e. you can form it's invers A^{-1}) and square, then the system has a unique solution $x = A^{-1}b$. While this is theoretically of much value, computationally it is inefficient at best, and at worst can result in quite significant numerical errors (and prohibitively long execution times). However, since solving systems of linear equations is such an important topic, there are myriad different numerical approaches that can be taken in a variety of situations. Thankfully, MATLAB does the thinking for you and will automatically choose with algorithm to use depending on what kind of input you choose. These algorithms are all contained in the innocuous looking \ operator.

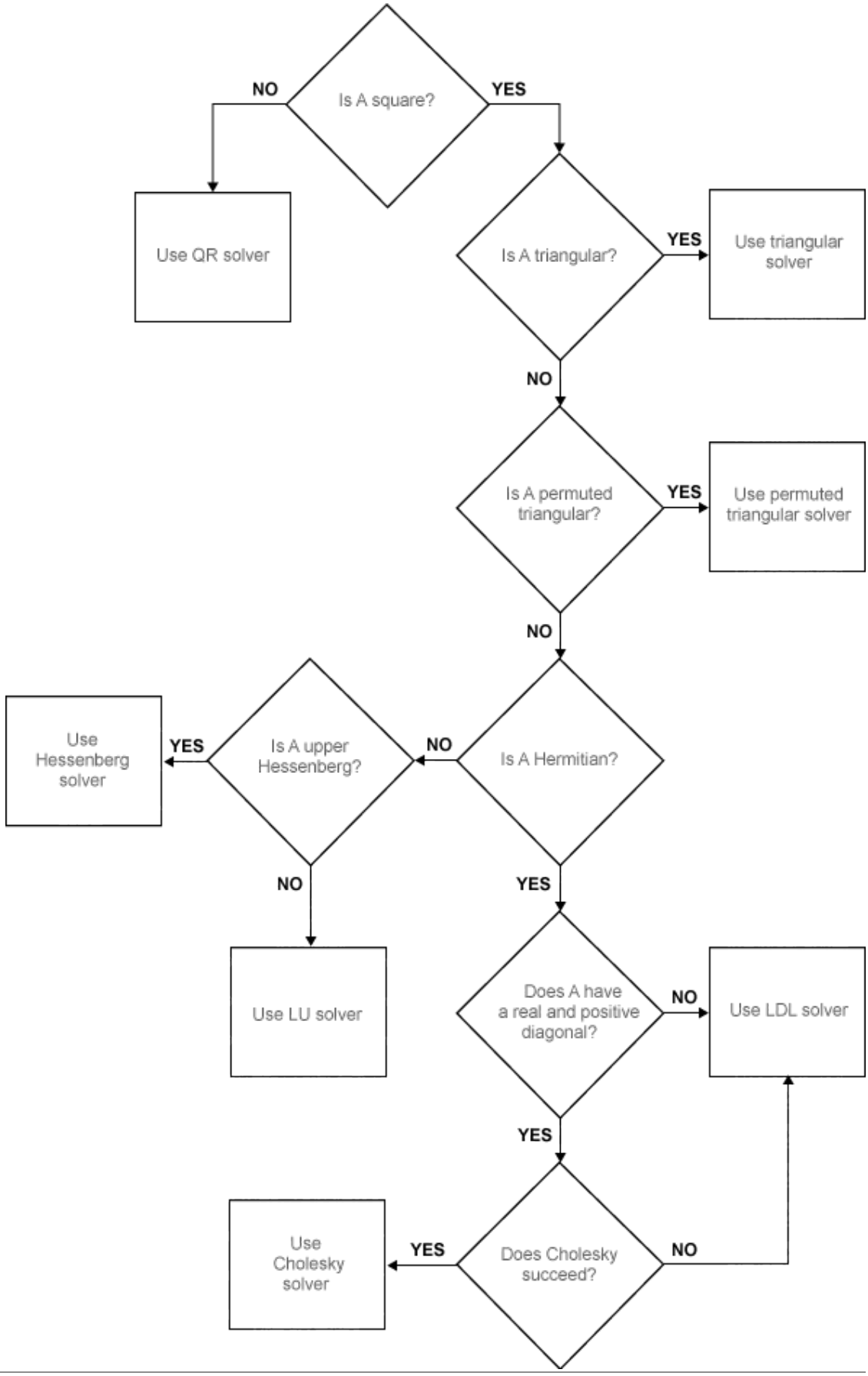
Before we discuss what \ does, let's see it in action.

```
A = [1 2 3; 4 5 6];
b = [2;8];
x=A\b % This is solving Ax=b for x
```

We can verify that this is the correct answer by testing

```
isequal(A*x,b)
```

Now the backslash operator is more properly called a *polyalgorithm* since the same notation actually encapsulates many different algorithms. We won't be going through them all, but here is a flowchart showing how MATLAB chooses the algorithm for \


Is A square?
NO
YES
Use QR solver
Is A triangular?
YES
Use triangular solver
NO
Is A permuted triangular?
YES
Use permuted triangular solver
NO
Is A Hermitian?
NO
Is A upper Hessenberg?
YES
Use Hessenberg solver
NO
Use LU solver
YES
Does A have a real and positive diagonal?
NO
Use LDL solver
YES
Does Cholesky succeed?
NO
YES
Use Cholesky solver

ICE

Part 1. Given the matrices $A = \begin{bmatrix} 3 & 3 \\ 2 & 1 \end{bmatrix}$, $B \begin{bmatrix} 1 & 1 \\ 2 & 3 \end{bmatrix}$ and $C \begin{bmatrix} 3 & 1 \\ 5 & 5 \end{bmatrix}$, use Matlab to verify each of the following properties. Note that 0 represents the zero matrix.

1. $A + B = B + A$
2. $(A + B) + C = A + (B + C)$
3. $A + 0 = A$
4. $A + (-A) = 0$

Part 2. The fact that matrix multiplication is not commutative is a **very** important point to understand. Use the matrices A and B from above to show that none of the following properties is true

1. $(ab)^2 = a^2b^2$
2. $(a + b)^2 = a^2 + 2ab + b^2$
3. $(a + b)(a - b) = a^2 - b^2$

Part 3.

1. Enter the following coefficient matrix $A = \begin{bmatrix} 2 & -1 & 0 \\ 1 & -2 & 1 \\ 0 & -1 & 2 \end{bmatrix}$ and constant vector $b = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$. Solve the system $Ax = b$ using the following three methods: First, calculate the inverse of *A* using inv(A) and calculate $x = A^{-1}b$. Next, calculate *x=A\b*. Finally, Use *lsqlin()*.
2. The *trace* of a matrix is the sum of the diagonal elements of a matrix. Come up with two different ways to compute the trace (one using a built-in MATLAB function, one using *diag* and *sum*).
3. How might we test whether or not a matrix is square?

Flow Control

In this section we will discuss the basic programming flow control constructs available in Matlab. Specifically, we will look at *if* statements, *switch* statements, *for* loops, *while* loops, *any*, *all*, *break*, etc. These flow control constructs are foundational to programming; it's amazing what can be accomplished with the proper combination of these!

If

The *if* statement evaluates a logical expression and executes a block of statements based on whether the logical expression evaluates to *true* (that is, the logical *1*) or *false* (that is, the logical *0*). The basic structure is as follows

```
if logical_expression
    statements
end
```

If the *logical_expression* is true, then the *statements* are evaluated. If it's not true, then the *statements* are simply skipped. For example

```
mynum = 4;
if mod(mynum,2) == 0
    fprintf('The number %d is even!\n',mynum)
end
```

Note what happens when you change *mynum* to *5*.

Else

We can provide an alternative if the *logical_expression* evaluates to false by using an *else* command:

```
if logical_expression
    statements_true
else
    statements_false
end
```

Here, if *logical_expressions* is true, then *statements_true* is evaluated; if *logical_expressions* is false, then *statements_false* is evaluated. The advantage of using an *if*,*else* construct is that you can be sure that one of your statement blocks is guaranteed to be evaluated. for example:

```
mynum = 5;
if mod(mynum,2) ==0
    fprintf('The number %d is even!\n',mynum)
else
    fprintf('The number %d is odd!\n',mynum)
end
```

Elseif

Sometimes you need to add more than one alernative. For this, we have the following construct:

```
if logical_expression_1
    statements_1
elseif logical_expression_2
    statements_2
else
    statements_3
end
```

Here, Matlab first evaluates *logical_expression_1*, if it's true, it evaluates *statements_1* and then skips the rest. If *logical_expression_1* is false, it moves on to *logical_expression_2*, if *logical_expression_2* is true, it

evaluates *statements_2* and skips the rest. If *logical_expression_2* is false, then it evaluates _statements_3. Let's look at an example

```
mynum = 17;
if mynum < 10
    fprintf('The number %d is less than 10!\n', mynum)
elseif mynum > 20
    fprintf('The number %d is greater than 20!\n', mynum)
else
    fprintf('The number %d is between 10 and 20\n', mynum)
end
```

Try different values of *mynum* and see what happens. You can enter as many *elseif* statements that you wish. Note that it's good programming practice to include an *else* block to catch any cases you might not have considered. For example

```
mynum = input('Enter an integer: ');
if mynum == 1
    fprintf('Your number is equal to 1\n')
elseif mynum == 2
    fprintf('Your number is equal to 2\n')
elseif mynum == 3
    fprintf('Your number is equal to 3\n')
else
    fprintf('Your number is not equal to 1, 2, or 3\n')
end
```

However, this code can be more succinctly represented with a *switch* expression, which is what we will look at next.

Switch

A *switch* block conditionally executes one set of statements from several choices. The basic usage is

```
switch switch_expression
   case case_expression
      statements
   case case_expression
      statements
   otherwise
      statements
end
```

You can include as many cases as you want. Typically, you use a *switch* statement when you need to pick one set of code based on the variable value, whereas an *if* statement is meant for a series of Boolean checks. Here's and example of the above *if* statement re-written as a *switch* statement:

```
mynum = input('Enter an integer: ');
switch mynum
    case 1
        fprintf('Your number is equal to 1\n')
    case 2
        fprintf('Your number is equal to 2\n')
    case 3
        fprintf('Your number is equal to 3\n')
    otherwise
```

```
        fprintf('Your number is not equal to 1, 2, or 3\n')
end
```

Sometimes a *switch* statement reads much more clearly than an *if* statement with a bunch of *elseif*'s. Here's a slightly more complicated *switch* statement:

```
a = input('Enter a number: ');
b = input('Enter another number: ');
fprintf('\n Enter one of the following choices\n')
fprintf('1) Add a and b.\n')
fprintf('2) Subtract a and b.\n')
n = input('Enter your choice: ');
switch n
    case 1
        fprintf('The sum of %.2f and %.2f is %.2f.\n',a,b,a+b)
    case 2
        fprintf('The difference of %.2f and %.2f is %.2f.\n',a,b,a-b)
    otherwise
        fprintf('%d is an invalid choice, please try again\n',n)
end
```

For loops

A *for* loop is a control structure that is designed to execute a block os statements a predetermined number of times. The general syntax of the *for* loop is:

```
for index=start:increment:finish
    statements
end
```

As and example, let's print out the square of the first 10 natural numbers:

```
for i=1:10
    fprintf('The square of %d is %d\n', i, i^2)
end
```

Note that we don't need to use the colon notation for the index of a *for* loop. If *values* is an iterable element, then we can use

```
for index = values
    statements
end
```

For example:

```
vals = randi(100,10,1);
for i = vals
    fprintf('Your number is %d.\n', i)
end
```

If you set the values of a *for* loop to a matrix, then Matlab will proceed along the values of the matrix column-wise:

```
A = [1 2 3; 4 5 6; 7 8 9];
for i = A
    fprintf('The value of i is %d\n',i)
end
```

for loops are extremely powerful, but be careful since Matlab doesn't excute *for* loops in the most efficient way possible. If you can ever vectorize your code (or reduce a loop to a linear algebra operation), then this will be much faster.

```
mysum = 0;
vals = 1:10^7;
looptic = tic();
for i = vals
    mysum = mysum + i;
end
fprintf('For loop timing: %f seconds.\n', toc(looptic))
vectorizetic = tic();
sum(vals);
fprintf('Vectorized timing: %f seconds.\n', toc(vectorizetic))
```

While loops

while loops are very similar to *for* loops, but continue evaluating the statements until some expression no longer holds true. The basic syntax is

```
while expression
    statements
end
```

For example

```
i = 1;
N = 10;
mynums = zeros(1,N);
while i<=N
    mynums(i) = i;
    i = i + 1;
end
```

One must be very careful with *while* loops to not get stuck in an infinite loop. The following is an example of an infinite loop

```
i = 1;
N = 10;
mynums = zeros(1,N);
while i<=N
    mynums(i) = i;
end %press Ctrl+C to exit this infinite loop.
```

for loops are good for when you know in advance the precise number of itmes the loop should iterate. *while* loops are handy for when you have no predetermined knowledge of how many times you want the loop to execute. Here's a perfect example of when you want a *while* loop instead of a *for* loop (this example comes from the **Collatz Conjecture**).

```
n = input('Input a natural number: ');
iter = 0;
myseq = [n];
while n>1
    if mod(n,2) == 0
        n = n/2;
    else
```

```
        n = 3*n+1;
    end
    iter = iter + 1;
    myseq(end+1) = n;
end
fprintf('It took %d iterations to reach n=1\n',iter)
```

break

The *break* command terminates the execution of a *for* loop or *while* loop. This happens when the programmer wishes to terminate the loop after a certain state is achieved and pass control to the code that follows the end of the loop. For example, let's say you don't want the above Collatz Conjecture code to run for too long. We can break out of the loop if the number of iterations exceeds some predetermined amount.

```
n = input('Input a natural number: ');
iter = 0;
myseq = [n];
breakflag = 0;
while n>1
    if mod(n,2) == 0
        n = n/2;
    else
        n = 3*n+1;
    end
    iter = iter + 1;
    myseq(end+1) = n;
    if iter >= 100
        breakflag = 1;
        break
    end
end
switch breakflag
    case 0
        fprintf('It took %d iterations to reach n=1\n',iter)
    case 1
        fprintf('Failed to converge after 100 iterations\n')
end
```

Here, if the number of iterations surpasses 100, the *break* command tells Matlab to leave the *while* loop and proceed to the next set of statements (the *switch* statement in this case).

continue

The *continue* keyword passes control to the next iteration of a *for* or *while* loop, skipping the rest of the body of the *for* or *while* loop on that iteration. For example, consider the following code

```
i=1;
while i<10
    if mod(i,2) == 0
        i = i + 1;
        continue
    end
    fprintf('%d\n',i)
    i = i + 1;
end
```

Here, if the index *i* is even, then *i* is incremented and the rest of the *while* loop body (that is, the part that prints out the result) is skipped. This is particularly useful if you want to skip over certain cases in your code (for example, don't process prime numbers, invalid input, null entries, NaN's etc.).

Nesting Loops

It's possible to nest one loop inside another! Let's look at an example: A Hilbert matrix is a square matrix with entries being the unit fractions $H(i,j) = \frac{1}{i+j-1}$. Let's see how we can define our own Hilbert matrix

```
N = 10;
H = zeros(N,N);
for i=1:N
    for j=1:N
        H(i,j)=1/(i+j-1);
    end
end
```

Note that pre-allocating an array saves quite a bit of time in Matlab (this is because otherwise Matlab needs to make an entirely new copy of the array in memory). Try the following: Let *N* be a large number and compare the results of the above code to that where the line *H = zeros(N,N)* is removed. Which is faster?

ICE

1. The number of ways to choose k objects from a set of n objects is defined and calculated with the formula $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Define a Pascal matrix P with the formula $P(i,j) = \binom{i+j-2}{i-1}$. Use this definition to find a Pascal matrix of dimensions *4x4*. Use Matlab's *pascal()* command to check your result.

2. Write a for loop that will output the cubes of the first 10 positive integers. Use fprintf to output the results, which should include the integer and its cube. Write a second program that uses a while loop to produce an identical result.

3. Write a single program that will count the number of divisors of each of the following integers: 20, 36, 84, and 96. Use fprintf to output each result in a form similar to “The number of divisors of 12 is 6.”

4. Write a program that uses nested for loops to produce Pythagorean Triples, positive integers a, b and c that satisfy $a^2 + b^2 = c^2$. Find all such triples such that $1 \leq a, b, c \leq 20$ and use fprintf to produce nicely formatted results.

5. Write a program to perform the following task: Use Matlab to draw a circle of radius 1 centered at the origin and inscribed in a square having vertices (1, 1), (-1, 1), (-1,-1), and (1,-1). The ratio of the area of the circle to the area of the square is pi : 4 or $\pi/4$. Hence, if we were to throw darts at the square in a random fashion, the ratio of darts inside the circle to the number of darts thrown should be approximately equal to $\pi/4$. Write a for loop that will plot 1000 randomly generated points inside the square. Use Matlab’s rand command for this task. Each time a random point lands within the unit circle, increment a counter *hits*. When the *for* loop terminates, use *fprintf* to output the ratio darts that

land inside the circle to the number of darts thrown. Calculate the relative error in approximating $\pi/4$ with this ratio.

Functions

You've already encountered function notation in other mathematics courses $f(x) = 2x^2 - 3$. Matlab handles the definition of functions in a couple different ways: Anonymous functions and Function M-files.

Anonymous functions

The general syntax of an anonymous function is

```
funhandle = @(arglist) expr
```

Here *expr* is the body of the function; it consists of any **single** valid Matlab expression. The *arglist* should be a comma separated list of all input variables to be passed to the function. The *funhandle* is the function handle that stores the anonymous function. You can then evaluate your function using

```
funhandle(arg1,arg2,...
    ,argN)
```

For example, we could use

```
funhandle = @(x) 2*x.^2-3 %be careful to make your function "array smart"
funhandle(10)
```

You can then use this function just like any other matlab command. For example, let's apply this function to every element of the list *1:10*

```
res = arrayfun(funhandle, 1:10); %equivalently funhandle(1:10)
```

We can define as many variables as we please

```
myfunc = @(x,y,z) x.^2+y.^2+z.^2;
myfunc(1,2,3)
```

You can include pre-defined variables in your function as well

```
A=1;
B=2;
C=3;
f=@(x,y,z)A*x+B*y+C*z;
f(-1,0,1)
```

However, if you change the value of *A*,*B* or *C*, then you will need to re-define the function handle

```
A=10;
f(-1,0,1)
f=@(x,y,z)A*x+B*y+C*z;
f(-1,0,1)
```

Lastly, you don't actually need an argument for the function to work

```
t = @() datestr(now);
```

```
t()
```

Function M-files

Anonymous function should be used very sparingly, and should have relatively small bodies. A function M-file, however, is the "meat and potatoes" of Matlab, allowing you to turn a script into a multi-purpose, reusable, honest-to-goodness program.

Like a script file, you write function M-files in the Matlab editor. The first line of a function M-file must contain the following syntax

```
function [out1, out2, ..] = funname(in1,in2,..)
```

The keyword function must be the first word in the function m-file. This alerts the Matlab interpreter that the file contains a function, not a script. Next comes a comma delimited list of output arguments (surrounding brackets are required if more than one output variable is used), then an equal sign, followed by the function name. The function name funname must be string composed of letters, digits, and underscores. The first character of the name must be a letter. A valid length for the function name varies from machine to machine, but can be determined by the command namelengthmax.

```
namelengthmax
```

If you name your function *funname*, then you must save the file as *funname.m.* After the function name comes a comma delimited list of input variables, surrounded by required parentheses. Input arguments are passed by value and all variables in the body of the function are “local” to the function (that is, they aren't accessible outside this particular function).

Let's start with a very simple example. Open a new editor window and type

```
function y = myfun(x)
    y = 2*x.^2-3;
end
```

Save this as a file called *myfun.m* If this file is in your path, you should be able to run it as

```
myfun(10)
```

You are allowed to have more than one output

```
function [area,circumference]=area_and_circumference(r)
    area=pi*r^2;
    circumference=2*pi*r;
end
```

Once again, save this as a file called *area_and_circumference.m* You can then evaluate it using

```
radius = 12;
[A, C] = area_and_circumference(radius)
```

Documenting your function

The first contiguous block of comment files in a function m-file are displayed to the command window when a user enters *help funname*, where *funname* is the name of the function. You terminate this opening comment block with a blank line or an executable statement. After that, any later comments do not appear when you type help funname.

```
function [area,circumference]=area_and_circumference(r)
```

```
%AREA_AND_CIRCUMFERENCE finds the area and circumference
%of a circle.
% [A,C]=AREA_AND_CIRCUMFERENCE(R) returns the area A and
% circumference C of a circle of radius R.
area=pi*r^2;
circumference=2*pi*r;
```

You should get in the habit **immediately** of documenting all of your functions. I guarantee that if you don't, after some time (maybe a few months, maybe a few years) you'll come back to your code, have no idea what it does, and then if there aren't any comments, spend the next few hours/days re-figuring it out.

You can then type

```
help area_and_circumference
```

To get information about this function.

Functions with no output

Sometimes you will want to define a function that has no output (for example, one that writes the results to a file instead of passing them back to Matlab). This is easy to implement, simply leave the *[out1, out2, ...]* part of your function off. For example

```
function write_to_file(input_vector, output_file)
    fid = fopen(output_file,'w');
    for i=1:length(input_vector)
        fprintf(fid,'%f\n', input_vector(i))
    end
    fclose(fid);
end
```

ICE

1. Write an anonymous function to emulate the function $h(u,v) = \cos(u)\cos(v)$. Evaluate the value of $h(1,2)$.
2. Consdier the anonymous function `f=@(x) x.*exp(-C.*x.^2);`. Write a *for* loop that will plot this family of functions on the interval $[-1,1]$ for C in $\{-1,-0.8,-0.6,\ldots 1\}$.
3. Write a function M-file that will emulate the function $f(x) = xe^{-Cx^2}$ an will allow you to pass C as a parameter. Save this to a file and then write a *for* loop using this funciton on nterval $[-1,1]$ for C in $\{-1,-0.8,-0.6,\ldots 1\}$.
4. Euclid proved that there are an infinite number of primes. He also believed that there was an infinite number ot twin primes, primes that differ by 2 (like 5 and 7 or 11 and 13), but no one has been able to prove this conjecture for the past ~2300 years. Write a function that will produce all twin primes between two inputs, integers a and b.
5. Write a function called *mymax* that will take a column vector of numbers as input and find the largest entry. The routine should spit out two numbers, the largest entry *val* and

its position *pos* in the vector. Test the speed of your routine with the following commands:

```
test=rand(1000000,1); tic; [pos, val]=mymax(test); toc
```

Compare the speed of your maximum routine with Matlab's

```
tic; [pos, val] = max(test); toc
```

Variable Scope

Now that we know the general structure and use of a function, we'll now look at the scope of a variable (which is loosely defined as the range of functions that have access to that variable in order to modify or access its value).

Base workspace

Matlab stores variables in parts of memory called workspaces. The base workspace is the part of memory that is used when you are entering commands at the Matlab prompt in the command window. You can clear all variables from your base workspace by entering the command

```
clear
```

at the Matlab prompt in the command window.

The command *whos* will list all the variables in the current workspace.

```
whos
```

To get more information about a specific variable, we can use

```
M = magic(10);
whos M
```

Scripts and the base workspace

When you execute a script file from the Matlab prompt in the command window, the script uses the base workspace. It can access any variables that are present in the base workspace, and any variables created by the script remain in the base workspace when the script finishes. As an example, open the Matlab editor and enter the following lines:

```
x = 1:5
s = sum(x)
p = prod(x)
y = y + 1
```

Save the script as *sumprod.m* and return to the Matlab command window. Type the command

```
y = 1;
```

Now execute the script by typing its name at the command prompt. Now examine the state of the base workspace by typing the *whos* command at the command prompt.

```
whos
```

Note that the variables *x*, *s*, and *p* were created in the script and are now present in the base workspace. Note also that the value of *y* was incremented by one. Hence a script both has access to as well as the ability to change the base workspace variables.

You must be **VERY** careful about this, since it is very easy to make a script that will behave very differently depending on what variables are present within your base workspace. It's good practice to either use *clear* before running a script, or, better yet, re-write the script as a function. We'll see why this has an advantage next.

Function Workspaces

Function do not sue the base workspace. Each function has its own workspace where it stores its varibles, and this is kept separate from the base workspace.

For example, say you make the following very simple function, and call it simple.m

```
function simple
    v=10;
    fprintf('The value is: %d',v)
```

Then running this function using

```
simple
```

will output the message "The value is: 10". However, if you try to check the value of v in your workspace using

```
whos v
```

nothing will get returned since the variable *v* doesn't exist on your base workspace.

One important feature of Matlab is since it cordons off variables for functions, to allow a function to access a variable in your workspace, it has to duplicate the variable (and its associated data) before allowing your function to access it. This is called "passing a value" (rather than "passing a reference". Keep this in mind if your variables are storing data that takes a large amount of memory.

Global variables are one way to allow a function to access variables in the base workspace (though one must be very careful about unintended consequences of this). Globabl variables are declared using

```
global v
```

Then if we assign the value of

```
v = 0
```

and then run the function

```
simple
```

we will see that the value of the variable has indeed changed!

```
v
```

ICE

1. Clear the workspace by entering the command clear all at the Matlab prompt in the command window. Type *whos* to examine the base workspace and insure that it is empty. Open the editor, enter the following lines, then save the file as VariableScope.m. Execute the cell in cell-enabled mode.

```
%Exercise #1
```

```
clc
P=1000;
r=0.06;
t=10;
I=P*r*t;
fprintf('Simple interest gained is %.2f.\n',I)
```

Examine the base workspace by entering *whos* at the command window prompt. Explain the output (of the *whos* command, not of the script itself).

2. Clear the workspace by entering the command clear all at the Matlab prompt in the command window. Type whos to examine the base workspace and insure

```
%that it is empty. At the command prompt, enter and execute the following
%command.
P=1000; r=0.06; t=10;
```

Examine the workspace using *whos*. Now add the following cell to the file VariableScope.m

```
clc
I=P*r*t;
fprintf('Simple interest gained is %.2f.\n',I)
```

State the output of the script. Use *whos* to examine the base workspace variables. Anything new? Explain.

3. Clear the workspace by entering the command clear all at the Matlab prompt in the command window. Type whos to examine the base workspace and ensure that it is empty. At the command prompt, enter and execute the following command.

```
P=1000; r=0.06; t=10;
```

Examine the workspace with *whos*. Now open the editor, enter the following lines, and save the funciton as the M-file *simpleInterest.m*

```
function I=simpleInterest
I=P*r*t;
fprintf('Simple interest gained is %.2f.\n',I)
```

Add the following cell to VariableScope.m

```
clc
I=simpleInterest1;
fprintf('Simple interest gained is %.2f.\n',I)
```

Evaluate the cell and the state what happens and why.

File I/O

Direct input of data from the keyboard becomes impractical when the data being analyzed is of any reasonable size (or is being analyzed repeatedly). In these situations, input and output is preferably accomplished via data files. We have already used the commands *load* and *save* to load *.mat files, but Matlab is able to interact with much more diverse kinds of data.

When data is written to or read from a file, it is very important that the correct data format is used. The data format is the key to interpreting the contents of a file and must be known in order to correctly interpret

the data in an input file. There are a number of different file types we will be primarily concerned with: formatted text files, and binary files.

Formatted Text Files

First, let's see how we can write text to a data file. Let's say you are recording the temperature on various days of the week. So your data might look like

```
months = {'January', 'February', 'March'};
days = [1, 8, 15, 22, 29];
temps = 10*rand(1,length(days)*length(months)) + 32;
```

Our goal is to output this in a flat text file using the following format: In the first column, list the month, in the second column, list the day, in the third column, list the temperature. So it might look like:

January 1 37.1231

January 8 31.51

etc.

To write a file, we first need to open that file up. This is done using the *fopen* command:

```
fid = fopen('/path/to/file/temperatures.txt','w');
```

This creates a file temperatures.txt and prepares it for writing (erasing any preexising contents). The variable *fid* is a file identifier telling Matlab which file has been opened.

Let's write our temperature data to this file:

```
for month_index = 1:length(months)
    for day_index = 1:length(days)
        fprintf(fid,'%s\t%d\t%f\n',months{month_index},...
            days(day_index),temps((month_index-1)+day_index))
    end
end
```

If you try to open this file in your OS, it will show as a blank file since Matlab has placed a lock on the file (as it can still write contents to it). You can indicate that you are finished writing to the file by using

```
fclose(fid);
```

We can now reverse the whole procedure by indicating that we want to read the file, and using fscanf

```
clear
fid = fopen('/path/to/file/temperatures.txt','r');
Results = textscan(fid,'%s\t%d\t%f\n');
fclose(fid);
months = Results{1};
days = Results{2};
temps = Results{3};
```

One annoying fact is that Matlab will by default split strings on whitespace characters (so read in the string 'Test String' as {'Test', 'String'}). This can wreck havoc when reading in your data, but can be avoided by using:

```
textscan(fid,FormatString,'Delimiter','\n')
```

If you have a matrix written in a text file (so too many columns to make a long format string), you can use *dlmread* and *dlmwrite*. Let's write a large matrix to a text file, and then read it back in:

```
rand_nums = rand(10^3,10^3);
dlmwrite('rand_nums.csv',rand_nums) %Default delimiter is ','
```

Now read it back in

```
clear
rand_nums = dlmread('rand_nums.csv');
%
```

Binary Files

There are a number of binary file types, and while these are typically not human readable, they have the advantage of being very efficient to read from/write to, as well as typically take less hard drive space. Here, we will look at one such binary format, called HDF5.

HDF5 is a hierarchical data format designed to store **massive** amounts of numeric data. It was originally designed for use in supercomputing centers, and currently has a very efficient and robust C library implementing its functionalities. We will look at how to use Matlab to call these functions.

You can think of each HDF5 file as its own "hard drive". In fact, you initialize an HDF5 file by giving it a "drive name". Here's an example:

```
[num_rows,num_cols] = size(rand_nums);
if exist('test_file.h5')
    delete('test_file.h5')
end
h5create('test_file.h5','/drive_name', [num_rows num_cols]);
```

This creates an empty HDF5 file of size 10^3 x 10^3. Let's now write a bunch of random numbers to it

```
h5write('test_file.h5','/drive_name', rand_nums)
```

We can get some information about this file by using

```
info = h5info('test_file.h5');
info.Filename
info.Datasets.Dataspace
```

Reading the data back in consists of:

```
rand_nums = h5read('test_file.h5','/drive_name');
```

Try timing *h5read* versus *dlmread* to see the performance advantage of HDF5. Furthermore, HDF5 allows for compression. Let's create another data set inside of test_file.h5 and compress it.

```
h5create('test_file.h5','/compressed_drive', [10^3 10^3], 'Datatype',...
    'int32', 'ChunkSize', [10 10^3], 'Deflate', 7)
```

The 'Datatype' is an optional command telling what kind of data will go in there, while 'ChunkSize' tells the HDF5 library how much of the matrix to read/write in any given read/write operation. The 'Deflate' command specifies the level of compression. Let's now put some data in this dataset.

```
my_ints = int32(reshape(1:10^6,10^3,10^3));
```

```
h5write('test_file.h5','/compressed_drive', my_ints)
```

Note this took longer to write, but if we were to write this to a CSV file, note the differences in file size:

```
dlmwrite('rand_ints.csv',my_ints);
```

There are many, many more features of HDF5 files that we won't go into here. One last advantage of using HDF5 files is that most programming languages include ways to easily read/write these kinds of files.

ICE

1. Study the available information on Mathworks.com regarding the fscanf/sscanf and sprintf/fprintf. What's the meaning of the format string '%3d\n'?

2. Create a 10^4 x 3 matrix where the rows correspond to 1:10^4 and the columns correspond to the functions *sin(x)*, *log(x)*, and *x^2*. Let the *(i,j)* entry be the value of the jth function evaluated on i. Save this matrix as a *.mat file (use *save*), a comma seperated file (use *dlmwrite*), text file (use *fprintf*) as an Excel file (use *xlswrite*), and as a compressed HDF5 file (use *h5write*). Include these files in the dropbox folder when you complete this homework assignment.

```
a) Which format took the longest/shortest amount of time to
write?

b) Which format used the least/most amount of disk space?

c) Which format took the longest/shortest amount of time to
read in?
```

Parallel Computing

As more CPU cores are squeezed into a single processor (at the time of writing, you could buy a single CPU that has 16 cores on it from Newegg.com), and as the amount of data available to analyze continues to grow (see: Big Data), parallel processing has become an integral part of scientific data analysis. Large computational clusters consisting of thousands of cores are commonly used at research institutions around the world. Here at OSU for example, we have the CGRB which currently has 2,432 CPUs spread over 76 different servers. How can we take advantage of all this processing power? Matlab has a very user-friendly implementation of parallel programming that we will review here (introduced in version 2008a).

Matlabpool

To start a parallel session of matlab, open Matlab as usual. Next, type the following into the command window:

```
matlabpool open
```

This will automatically spawn independant instances of matlab, one on each physical processor.

Parallel computing is most efficient when the computational job to be accomplished can be divided into as many tasks as processors, the pieces of data sent to the individual processors, the computation performed, and then afterwards recombined. It is less efficient (sometimes horribly so) if the processors need to talk to each other in order to complete the task.

Discuss parallel programming concepts, independence, etc.

Let's look at the most commonly used parallel programming tool in Matlab: parfor

Parfor

parfor has exactly the same syntax as the usual *for* loop, but matlab automatically splits the job up to each individual processor. Let's see how this works in action and compare it to the serial version. Let's factor all the integers less than 10^5.

```
N=10^5;
tic;
for i=1:N
    factor(i);
end
fprintf('Serial version timing: %f seconds\n',toc)

tic;
parfor i=1:N
    factor(i);
end
fprintf('Parallel version timing: %f seconds\n',toc)
```

To close the worker instances of matlab, use

```
matlabpool close
```

Parfor variable

To minimize the amount of time spent in the very slow act of communicating between processors or accessing memory, certain types of variables are classified in different ways in Matlab. Let's look at an example

```
a = 0;
z = 0;
c = pi;
r = rand(1,10);
parfor i = 1:10
    a = i; %a is a temporary variable, i is the loop variable
    z = z + i; %z is a reduction variable
    b(i) = r(i); %b(i) is a sliced output variable, r(i)
    % is a sliced input variable
    if i <= c %c is a broadcast variable
        d = 2*a;
    end
end
```

A loop variable serves as a loop index for arrays. Modifying the loop variable is not allowed. A temporary variable is a "local" variable that might be different on each worker. It is any variable that is the target of a direct, non-indexed assignment. A sliced variable is one whose value can be broken up into segments, and this small segments sent to each individual worker. A reduction varible accumulates a value that depends on all the iterations together, but is independent of the iteration order. Matlab CAN parallelize a loop containing a reduction variable. However, Matlab cannot parallelize a loop variable whos value depends on the iteration order.

Here's an example of a loop that cannot be parallelized

```
vals = rand(1,10);
for i=1:10
    vals(i) = vals(max([1,i-1]));
```

```
    %Note that vals(i) depends on i-1, so matlab can't split up the task
    %into independant processes.
end
```

Here's an example of a loop with an reduction variable, a variable that depends on all the iterations, but not the order

```
total = 0;
vals = rand(1,10);
parfor i=1:10
    total = total + vals(i);
end
```

Slicing

Slicing is an important concept as it allows Matlab to break an array into smaller pieces and let each worker operate on just these pieces. Unfortunately, it's a bit complicated to define a sliced variable. You must make sure that:

1. The first level of indexing is either praentheses or curly braces
2. Within the first level indexing, the list of indices is the same for all occurrences of a given variable.
3. Within the list of indicies for the variable, exactly one index involves the loop variable.
4. In assigning to a sliced variable, the right hand side of the assignment is not deletion (i.e. not assigning to []).

Here's an example of a sliced variable

```
N = 1000;
A = rand(N);
total = 0;
tic
parfor i=1:N-1
    total = total + sum(A(i,:));
end
toc
```

If you don't slice the variable *A*, then Matlab needs to make an entirely new copy of it in memory, one for each worker.

Let's look now at an example of what happens when you don't slice a variable.

```
tic
N = 10000;
A = rand(N);
total = 0;
parfor i=1:N-1
    total = total + A(i,i+1);
end
toc
```

Note that we are doing less work than before (only summing on a diagonal of A, not the whole thing), yet this takes significantly longer. Open up Windows Task Manager and increase the size of N to see why this is happening. Note in particular the memory usage in both cases.

SPMD

SPMD stands for "Single program multiple data" and is useful when you want to do the same program on multiple data sets. Here is a very simple example

```
spmd
  % build random matrices in parallel
  mat = rand(labindex + 2);
end
for ii=1:length(mat)
  % plot each matrix
  figure
  imagesc(mat{ii});
end
```

The resulting composite object gives the value of *mat* on each processor.

This is particularly handy if you have multiple data files (or one large data set split into multiple files) and want to perform the same analysis on all of them. Here's an example

```
num_procs = matlabpool('size'); %Get the number of processors
% Create the test data
for i=1:num_procs
    data = rand(10^3,1);
    file = sprintf('test_data_%d.csv',i);
    dlmwrite(file,data);
end
%In parallel, compute the mean of each data set. The variable _mean_ will
%have a different value on each processor
spmd
    file = sprintf('test_data_%d.csv',labindex);
    data = dlmread(file);
    means = mean(data);
end
%Go through and remove the files we just created
for i=1:num_procs
    file = sprintf('test_data_%d.csv',i);
    delete(file);
end
```

ICE

1. We will test Girko's circle law, which states that the eigenvalues of a set of random N x N real matrices with entries independent and taken from the standard normal distribution (i.e. using *randn*) are asymptotically constrained inside a unit circle of radius $\sqrt{N}$ in the complex plane. Create a program that calculates (in parallel using parfor!) the eigenvalues of many (~1000) normally distributed random matrices of size 100x100. Plot the real and imaginary parts of all these eigenvalues on the plane (i.e. use the real parts as x-values and the imaginary parts as y-values), and draw a circle of radius $\sqrt{100}$ around it to verify Girko's circle law. How long would it have taken if you performed this task in serial instead of parallel?

2. Download the 4 FASTA files located in the ZIP file at http://www.math.oregonstate.edu/~koslickd/TestSequences.zip. Use an spmd program to read the files in (in parallel) using *fastaread*. Count the

number of bases in each sequence (using *basecount*, note that you can access the results of *basecount* by using the following syntax: *output = basecount(sequence); output.G*, this will count the number of Gs). Tally up the total number of Gs that appear in each file. Which file has the most Gs? How much quicker did this task take performing it in parallel rather than in serial?

Acknowledgement

Parts of this text are based on **An Introduction to Matlab** by David F. Griffiths, as well as on David Arnold and Bruce Wagner's **Matlab Programming**.

Published with MATLAB® R2013b

Matlab Homework

Homework Assignments

Table of Contents

Assignment #1: Due 10/2/2014

Basics ICE, questions 1-9:

1. Create a variable Speed_MPH to store a speed in miles per hour (MPH). Convert this to kilometers per hour (KPH) by using the conversion formula $KPH = 1.60934MPH$.
2. Create a variable Temp_F to store a temperature in Fahrenheit (F). Convert this to Celsius (C) by using the conversion formula $C = \frac{5}{9}(F - 32)$.
3. Wind chill factor (WCF) measures how cold it feels when the air temperature is *T* (in Celsius) and the wind speed is *W* (in kilometers per hour). The expression is given by $WCF = 13.12 + 0.6215T - 11.37V^{0.16} + 0.3965T * V^{0.16}$. Calculate the wind chill factor when the temperature is *50F* and the wind speed is *20mph*.
4. Find a **format** option that would result in the command *1/2+1/3* evaluating to *5/6*.
5. Find MATLAB expressions for the following $\sqrt{2}$, $3^{1.2}$, and $\tan(\pi)$.
6. There is no built-in constant for *e* (Euler's constant). How can this value be obtained in MATLAB?
7. What would happen if you accidentally assigned the value of *2* to *sin* (i.e. *sin=2*)? Could you calculate *sin(pi)*? How could you fix this problem?
8. Say you have a variable *iterator*. Let *iterator=7*. How could you increment the value of this iterator by one unit? Is there only one way to do this?
9. Random number generation is quite useful, especially when writing a program and the data is not yet available. Programmatic random number generation is not truly random, but pseudo-random: they take an input (called a seed) and generate a new number. Given the same random seed the same number will be generated. However, the numbers that are produced are "random" in the sense that each number is (almost) equally likely. Use the function *rand* to generate a random number. Now close out MATLAB, open it up again, enter *rand*, and record the number. Repeat this process. Did you notice anything? If you want a truly different number to be generated each time, you need to use a different seed. The command *rng('shuffle')* will use a new seed based on the current time. Open the help menu for *rng* and explore the different kinds of random number generators.

Assignment #2: Due 10/7/14

Vectors ICE, questions 1-11

1. How would you use the colon operator to generate the following vector: *[9 7 5 3 1]*?
2. Use the *length()* command to find the length of the following vectors: *x=1:0.01:5, y=2:0.005:3, z=(100:0.05:200)'*
3. Use Matlab's *sum()* command and vector colon notation to construct the sum of the first 1000 natural numbers.
4. Use Matlab's *sum()* command and vector colon notation to find the sum of the fist 1,000 odd positive integers.
5. Let $u = [1, 2, 3]$ and $v = [4, 5, 6]$. Use Matlab to compute $(u + v)'$ and $u' + v'$. Compare the results. Vary u and v. What have can you observe about these two expressions?
6. Let $v = [1, 3, 5, 7], w = [5, 2, 5, 4]$. Think about the output of each of the following commands; write down by hand what you think the answer will be. Next, run the commands in Matlab and check your results. Did they match? $v > w,\ v >= w,\ v <= w,\ v < w,\ v == w,\ v \sim= w$.
7. The empty vector is denoted *[ ]*. This can be used to delete elements of a vector. Create a vector *v* of length 5 and delete the first element by using *v(1)=[ ]*. What happens when you evaluate v(1)? What about v(5)?
8. Read the documentation for the *primes()* command. Use that command to find the sum of the first 100 primes. Create a histogram of the prime gaps (differences between successive primes) for all primes less than or equal to 10,000,000.
9. Another handy function is *linspace(x,y,n)* which gives a row vector with *n* elements linearly (evenly) spaced between *x* and *y*. Using this function, make a row vector of 100 evenly spaced elements between 1 and 0.
10. Using vectorization, estimate the limit $\lim_{x \to 0} \frac{\sin(x)}{x}$.
11. Generate a list of 10,000 random numbers uniformly between 0 and 1. Approximate the average value of $\frac{1}{x+1}$ on $[0, 1]$. Does this answer make sense? (Hint: use Calculus).

Assignment #3, Due 10/14/14

Plotting ICE

1. When explicitly specifying the x-coordinates of a plot, it is important to keep in mind the trade-off between speed and accuracy. Say you want to plot the function *sin(3*pi*x)* for *0<x<1*. Use *x=linspace(0,1,N)* and *plot(x, sin(3*pi*x))*. Compare the results for various values of *N*.
2. Generate a list of the first 100 powers of 2. The following function selects the first digit of a number x: $\left\lfloor \frac{x}{10^{\lfloor \log_{10}(x) \rfloor}} \right\rfloor$. Plot a histogram of the first digits of 1,000 powers of 2. Use this to estimate the probability that the first digit of a power of 2 is *1*. What is the probability that the fist digit is *2*? and

so on. This phenomena is referred to as *Benford's Law*: For *most* numbers, the probability that the first digit is *d* is given by $\log_{10}\left(1+\frac{1}{d}\right)$. Plot this function and compare this plot to the results we obtained. Interestingly, Benford's law is periodically used to check for fraud on tax returns.

For the following 6 exercises, set *x=linspace(a,b,n)* for the given values of *a*, *b*, and *n*. Calculate *y=f(x)* for the given function *f(x)*. Plot with *plot(x,y,s)* for the given formatting string *s*. Explain the result of the formatting string.

1. $f(x)=\sin(x),\ a=0,\ b=4\pi,\ n=24,\ s=$ `'rs-'`.
2. $f(x)=\arccos(x),\ a=-1,\ b=1,\ n=24,\ s=$ `'mo'`.
3. $f(x)=2^x,\ a=-2,\ b=4,\ n=12,\ s=$ `'gd:'`.
4. $f(x)=\sinh(x),\ a=-5,\ b=5,\ n=20,\ s=$ `'kx--'`.
5. $f(x)=\log_{10}(x),\ a=0.1,\ b=10,\ n=20,\ s=$ `'c--'`.
6. $f(x)=x^2,\ a=-5,\ b=5,\ n=20,\ s=$ `'b*-.'`.

Graph the given function on a domain that shows all the important features of the function (intercepts, extrema, etc.), or on the domain specified. Use enough points so that your plot appears smooth. Label the horizontal and vertical axis with Matlab's *xlabel()* and *ylabel()* commands. Include a *title()*.

1. $y=x^2-2x-3$
2. $y=x^3-3x^2-28x+60$
3. $y=xe^{-x^2}$ on $[-5,5]$
4. $y=\frac{1}{1+x^2}$ on $[-10,10]$
5. $y=(x^2-2x-3)e^{-x^2}$ on $[-5,5]$

Assignment #4, Due 10/16/14

Matrices ICE

1. Create a *5x8* matrix of all zeros and store it in a variable. Now replace the second row in the matrix with all 4's and the first column with the entries *[2 4 6 8 10]*.
2. Create a *100 x 100* matrix of random reals in the range from -10 to 10. Get the sign of every element. What percentage of the elements are positive?

In the following exercises, predict the output of the given command, then validate your response with the appropriate Matlab command. Use *A=magic(5)*.

1. $A>14$

2. $A <= 12$
3. $(A > 3)$ & $(A <= 20)$
4. $(A < 5)$ & $(A >= 21)$
5. $\sim (A <= 6)$
6. $(A > 6)$ & $\sim (A > 8)$

Assignment #5, Due 10/18/14

Matrix Operations ICE

Part 1. Given the matrices $A = \begin{bmatrix} 3 & 3 \\ 2 & 1 \end{bmatrix}$, $B \begin{bmatrix} 1 & 1 \\ 2 & 3 \end{bmatrix}$ and $C \begin{bmatrix} 3 & 1 \\ 5 & 5 \end{bmatrix}$, use Matlab to verify each of the following properties. Note that 0 represents the zero matrix.

1. $A + B = B + A$
2. $(A + B) + C = A + (B + C)$
3. $A + 0 = A$
4. $A + (-A) = 0$

Part 2. The fact that matrix multiplication is not commutative is a **very** important point to understand. Use the matrices A and B from above to show that none of the following properties is true

1. $(ab)^2 = a^2b^2$
2. $(a + b)^2 = a^2 + 2ab + b^2$
3. $(a + b)(a - b) = a^2 - b^2$

Part 3.

1. Enter the following coefficient matrix $A = \begin{bmatrix} 2 & -1 & 0 \\ 1 & -2 & 1 \\ 0 & -1 & 2 \end{bmatrix}$ and constant vector $b = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$. Solve the system $Ax = b$ using the following three methods: First, calculate the inverse of *A* using inv(A) and calculate $x = A^{-1}b$. Next, calculate *x=A\b*. Finally, Use *lsqlin()*.

2. The *trace* of a matrix is the sum of the diagonal elements of a matrix. Come up with two different ways to compute the trace (one using a built-in MATLAB function, one using *diag* and *sum*).

3. How might we test whether or not a matrix is square?

Assignement #6, Due 10/23/14

Flow Control

1. The number of ways to choose k objects from a set of n objects is defined and calculated with the formula $\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!}$. Define a Pascal matrix P with the formula $P(i,j) = \left(\begin{array}{c} i+j-2 \\ i-1 \end{array} \right)$. Use this definition to find a Pascal matrix of dimensions *4x4*. Use Matlab's *pascal()* command to check your result.

2. Write a for loop that will output the cubes of the first 10 positive integers. Use fprintf to output the results, which should include the integer and its cube. Write a second program that uses a while loop to produce an identical result.

3. Write a single program that will count the number of divisors of each of the following integers: 20, 36, 84, and 96. Use fprintf to output each result in a form similar to “The number of divisors of 12 is 6.”

4. Write a program that uses nested for loops to produce Pythagorean Triples, positive integers a, b and c that satisfy $a^2 + b^2 = c^2$. Find all such triples such that $1 \leq a, b, c \leq 20$ and use fprintf to produce nicely formatted results.

5. Write a program to perform the following task: Use Matlab to draw a circle of radius 1 centered at the origin and inscribed in a square having vertices (1, 1), (-1, 1), (-1,-1), and (1,-1). The ratio of the area of the circle to the area of the square is pi : 4 or $\pi/4$. Hence, if we were to throw darts at the square in a random fashion, the ratio of darts inside the circle to the number of darts thrown should be approximately equal to $\pi/4$. Write a for loop that will plot 1000 randomly generated points inside the square. Use Matlab’s rand command for this task. Each time a random point lands within the unit circle, increment a counter *hits*. When the *for* loop terminates, use *fprintf* to output the ratio darts that land inside the circle to the number of darts thrown. Calculate the relative error in approximating $\pi/4$ with this ratio.

Assignment #7, due 10/28/14

Functions

1. Write an anonymous function to emulate the function $h(u,v) = \cos(u)\cos(v)$. Evaluate the value of $h(1,2)$.

2. Consdier the anonymous function `f=@(x) x.*exp(-C.*x.^2);`. Write a *for* loop that will plot this family of functions on the interval $[-1,1]$ for C in $\{-1, -0.8, -0.6, \ldots 1\}$.

3. Write a function M-file that will emulate the function $f(x) = xe^{-Cx^2}$ an will allow you to pass C as a parameter. Save this to a file and then write a *for* loop using this funciton on nterval $[-1, 1]$ for C in $\{-1, -0.8, -0.6, \ldots 1\}$.

4. Euclid proved that there are an infinite number of primes. He also believed that there was an infinite number ot twin primes, primes that differ by 2 (like 5 and 7 or 11 and 13), but no one has been able to prove this conjecture for the past ~2300 years. Write a function that will produce all twin primes between two inputs, integers a and b.

5. Write a function called *mymax* that will take a column vector of numbers as input and find the largest entry. The routine should spit out two numbers, the largest entry *val* and its position *pos* in the vector. Test the speed of your routine with the following commands:
`test=rand(1000000,1); tic; [pos, val]=mymax(test); toc`
Compare the speed of your maximum routine with Matlab's
`tic; [pos, val] = max(test); toc`

Assignment #8, due 10/30/14

Variable Scope

1. Clear the workspace by entering the command clear all at the Matlab prompt in the command window. Type *whos* to examine the base workspace and insure that it is empty. Open the editor, enter the following lines, then save the file as VariableScope.m. Execute the cell in cell-enabled mode.

```
%Exercise #1
clc
P=1000;
r=0.06;
t=10;
I=P*r*t;
fprintf('Simple interest gained is %.2f.\n',I)
```

Examine the base workspace by entering *whos* at the command window prompt. Explain the output (of the *whos* command, not of the script itself).

2. Clear the workspace by entering the command clear all at the Matlab prompt in the command window. Type whos to examine the base workspace and insure that it is empty. At the command prompt, enter and execute the following command.

```
P=1000; r=0.06; t=10;
```

Examine the workspace using *whos*. Now add the following cell to the file VariableScope.m

```
clc
I=P*r*t;
fprintf('Simple interest gained is %.2f.\n',I)
```

State the output of the script. Use *whos* to examine the base workspace variables. Anything new? Explain.

3. Clear the workspace by entering the command clear all at the Matlab prompt in the command window. Type whos to examine the base workspace and ensure that it is empty. At the command prompt, enter and execute the following command.

```
P=1000; r=0.06; t=10;
```

Examine the workspace with *whos*. Now open the editor, enter the following lines, and save the funciton as the M-file *simpleInterest.m*

```
function I=simpleInterest
I=P*r*t;
fprintf('Simple interest gained is %.2f.\n',I)
```

Add the following cell to VariableScope.m

```
clc
I=simpleInterest1;
fprintf('Simple interest gained is %.2f.\n',I)
```

Evaluate the cell and the state what happens and why.

File I/O

1. Study the available information on Mathworks.com regarding the fscanf/sscanf and sprintf/fprintf. What's the meaning of the format string '%3d\n'?

2. Create a 10^4 x 3 matrix where the rows correspond to 1:10^4 and the columns correspond to the functions *sin(x)*, *log(x)*, and *x^2*. Let the *(i,j)* entry be the value of the jth function evaluated on i. Save this matrix as a *.mat file (use *save*), a comma seperated file (use *dlmwrite*), text file (use *fprintf*) as an Excel file (use *xlswrite*), and as a compressed HDF5 file (use *h5write*). Include these files in the dropbox folder when you complete this homework assignment.

```
a) Which format took the longest/shortest amount of time to write

b) Which format used the least/most amount of disk space?

c) Which format took the longest/shortest amount of time to read
```

Assignment #9, (not assigned due to unavailability of parallel computing toolbox)

Parallel Programming

1. We will test Girko's circle law, which states that the eigenvalues of a set of random N x N real matrices with entries independent and taken from the standard normal distribution (i.e. using *randn*) are asymptotically constrained inside a unit circle of radius $\sqrt{N}$ in the complex plane. Create a program that calculates (in parallel using parfor!) the eigenvalues of many (~1000) normally distributed random matrices of size 100x100. Plot the real and imaginary parts of all these eigenvalues on the plane (i.e. use the real parts as x-values and the imaginary parts as y-values), and draw a circle of radius $\sqrt{100}$ around it to verify Girko's circle law. How long would it have taken if you performed this task in serial instead of parallel?

2. Download the 4 FASTA files located in the ZIP file at http://www.math.oregonstate.edu/~koslickd/TestSequences.zip. Use an spmd program to read the files in (in parallel) using *fastaread*. Count the number of bases in each sequence (using *basecount*, note that you can access the results of *basecount* by using the following syntax: *output = basecount(sequence); output.G*, this will count the number of

Gs). Tally up the total number of Gs that appear in each file. Which file has the most Gs? How much quicker did this task take performing it in parallel rather than in serial?

Published with MATLAB® R2013b

Part 2: Mathematica

Introduction to Mathematica

MTH 399

David Koslicki
Oregon State University
2014

Table of Contents

Basics

Introduction to *Mathematica*

As opposed to Matlab, Mathematica is a very interactive/GUI-based computer algebra system. While Matlab excels at linear algebra and vectorized numerical computations, Mathematica is the premier symbolic computational platform. Additionally, Mathematica has numerous visualization routines and is ideal for exploratory data analysis. In Mathematica, everything is treated as a symbolic expression, and so this platform is highly optimized to handle symbolic operations (though numeric routines also exist).

Mathematica has thousands upon thousands of built in functions, and possesses libraries robust enough to create their own programming language (referred to as “Wolfram Language”). We will not attempt to give an exhaustive treatment of Mathematica, nor will we even be able to touch on all the areas of that Mathematica excels in, but we will rather try to demonstrate a number of capabilities of Mathematica and give you the tools necessary to continue learning on your own.

Notebook interface

Mathematica has an interface very different from Matlab: everything in *Mathematica* is done in an interface referred to as a “Notebook”. Press **ctrl+N** to open a new notebook. All expressions are entered into the notebook (including comments and other text). A notebook is divided into subsections referred to as “cells”. Each cell contains a *Mathematica* expression (or text) and can be evaluated by

placing your cursor into a cell and pressing **enter** on the numeric keypad, or else **shift+enter** on your keyboard. When you enter your first command, say:

```
1 + 1
```

```
2
```

You will notice that two labels come up. *Mathematica* automatically numbers every input it receives and every output it produces starting from 1, from the beginning of a session. These can be used to keep track of when input was entered, and can also be used to refer to output (if you haven't stored it to a variable). For example, we can access Out[1] via

```
Out[1]
```

```
2
```

Note that there are brackets on the right hand of the screen to denote the divisions of cells. These not only allow you to keep track of the organization of your notebook, but also allow you to edit and format cells (by right clicking on the brackets). If you see nested brackets (like in the notebook this tutorial itself is written in), you can double click an outer bracket to hide the sub-cells. This will allow you to quickly navigate between sections of your notebook.

It is always a good idea to organize your notebook to proceed from top to bottom. Not only will this be much easier to follow, but it will allow you to quickly evaluate every expression cell in your notebook quickly by clicking "Evaluation" then "Evaluate Notebook". This will proceed from top to bottom and sequentially evaluate every cell that can be evaluated (equivalent to going through and pressing shift+enter in each cell).

Mathematica has many autocomplete features, and you are encouraged to click around and experiment with any pop-ups or help menus that appear (*Mathematica* will, for example, suggestion completions for expressions that you enter. These can be selected using the tab and enter keys). For example, enter the expression and press **shift+enter**

```
Sqrt[Pi]
```

$$\sqrt{\pi}$$

Next, click on the suggestion "Integer part" (if using version 10) or some other auto suggestion. Populating the suggestions bar are a variety of functions or procedures that *Mathematica* predicts you might want to do next.

```
IntegerPart[√π]
```

```
1
```

We can include comments in a notebook by using the format and style options, or else by surrounding your text with "(*" and "*)"

```
(*This is a comment*)
```

Entering Expressions

Now that we know how a basic notebook works, we can focus on entering mathematical expressions. *Mathematica* has many, many different ways to enter expressions, but we it's hard to go wrong if you

keep the following rules in mind:

1. Functions in *Mathematica* are capitalized.
2. Use [ ] to surround function arguments.
3. Use { } to denote lists, matrices, and ranges.
4. Use [[ ]] to access elements of a list.

This will allow us to immediately start entering expressions.

```
1 + 2 - 3 / 4 * 5
```

$$-\frac{3}{4}$$

Note that we can use “space” (or juxtaposition) to imply multiplication

```
1 2 3
```

$$1 \times 2 \times 3$$

```
6
```

This was created by simply entering: 1+space+2+space+3.

Note that Mathematica conforms to all conventional operator precedence rules.

```
3 + 4 * 5
```

```
23
```

```
3 + (4 * 5)
```

```
23
```

```
(3 + 4) * 5
```

```
35
```

Since *Mathematica* treats everything as a symbol, it prefers to operate on exact expressions if possible. Hence entering “1/3” will return this exact value.

```
1 / 3
```

$$\frac{1}{3}$$

However, if you would like a numerical approximation, you can use the N[ ] function.

```
N[1 / 3]
```

```
0.333333
```

The second optional argument of N[ ] tells it how many digits to return.

```
N[1 / 3, 100]
```

```
0.33333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333 ↪
 33333333333333333333
```

Furthermore, including a “.” (decimal point) anywhere in your expression signifies to *Mathematica* that you are talking about an inexact quantity, hence the following evaluates to a numerical approximation:

```
1 / 3.
```

```
0.333333
```

Mathematica has the capability to return a very large number of digits of arbitrary inputs. For example, to return the first 1,000 digits of Pi, simply enter

```
N[Pi, 1000]
```

```
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899
 86280348253421170679821480865132823066470938446095505822317253594081284811174502
 84102701938521105559644622948954930381964428810975665933446128475648233786783165
 27120190914564856692346034861045432664821339360726024914127372458700660631558817
 48815209209628292540917153643678925903600113305305488204665213841469519415160943
 30572703657595919530921861173819326117931051185480744623799627495673518857527248
 91227938183011949129833673362440656643086021394946395224737190702179860943702770
 53921717629317675238467481846766940513200056812714526356082778577134275778960917
 36371787214684409012249534301465495853710507922796892589235420199561121290219608
 64034418159813629774771309960518707211349999998372978049951059731732816096318595
 02445945534690830264252230825334468503526193118817101000313783875288658753320838
 14206171776691473035982534904287554687311595628638823537875937519577818577805321
 71226806613001927876611195909216420199
```

Variables and Assignments

Mathematica uses assignment similar to other functional style programming languages. In particular, to store the value “1” to the variable “a”, use the following syntax:

```
a = 1
```

```
1
```

If you would ever like to suppress the output of your command, you can use the same semi-colon trick as in Matlab: “;” suppresses the output.

```
a = 1;
```

Good variable naming conventions are suggested here as well. There are a few restrictions on variable names, for example, a variable cannot start with a number nor contain a space. It is also a good idea to always use variables that start with lower cases, this way you won’t ever have to worry about conflicting with any of *Mathematica*’s built in functions. Unlike Matlab, *Mathematica* will not let you assign a value to a built-in function.

```
Sin = 1
```

```
Set::wrsym : Symbol Sin is Protected. >>
```

```
1
```

Be aware that *Mathematica* has a number of reserved variables. These start with the $ symbol and should not (cannot) be overwritten.

```
$Packages
```

```
{JSONTools`, JLink`, Macros`, CalculateUtilities`AlgorithmUtilities`,
 CalculateUtilities`UserVariableUtilities`, CalculateScan`Packages`Get3DRange`,
 CalculateScan`Packages`Get2DRange`, CalculateScan`Packages`Get1DPolarPlotRange`,
 CalculateUtilities`SuggestPlotRanges`, QuantityUnits`,
 URLUtilities`, WolframAlphaClient`, Utilities`URLTools`,
 HTTPClient`, HTTPClient`OAuth`, HTTPClient`CURLInfo`,
 HTTPClient`CURLLink`, DocumentationSearch`, GetFEKernelInit`,
 TemplatingLoader`, ResourceLocator`, PacletManager`, System`, Global`}
```

Mathematica does not show you all the variables you have given definitions to, but thankfully it does use syntax coloring to indicate when a variable has been given a name or not. Compare the following

```
a = 1
b
```

```
1
```

```
b
```

Since b has not been given a value, it is shown in blue. Any variable that does have a value is shown in black.

Note also that *Mathematica* is case sensitive

```
aa = 1;
AA = 2;
aA = 3;
Aa = 4;
```

```
{aa, AA, aA, Aa}
```

```
{1, 2, 3, 4}
```

If you need to clear the value of a variable, use

```
Clear[aa]
```

```
aa
```

```
aa
```

I would recommend using camelCase for variable names as starting out with a lower case avoids built in functions. Furthermore, underscores "_" have special meanings in *Mathematica*.

```
myArea = 10;
thisIsALongVariableName = 100;
```

It is important to realize that all *Mathematica* variables (except certain variables used inside of functions) are global variables. That means that if you define

```
a = 1;
```

Then "a" will be given the value of "1" in ANY place of your notebook from that point on. It will even retain this value if you open a new notebook.

You can clear all the values of your variables using

```
Clear["Global`*"]
```

You can use a previously defined variable to define a new one:

```
a = 3;
b = 4;
c = 6;
s = (a + b + c) / 2;
A = Sqrt[s (s - a) (s - b) (s - c)]
```

$$\frac{\sqrt{455}}{4}$$

Note that if you change the value of “a”, then the value of s does not immediately update

```
s
a = 100;
s
```

$$\frac{13}{2}$$

$$\frac{13}{2}$$

we would need to re-define s before it took on this new value

```
s = (a + b + c) / 2;
s
```

```
55
```

However, if you would like *Mathematica* to automatically update the value of “s” for you, you can accomplish this using something called “delayed assignments”. These are assignments that aren’t made until the right hand side of the assignment is evaluated afresh each time. Consider the following example:

```
x = 1;
y := x + 100;
```

Here we have defined y to be x + 100, but instead of evaluating x once, adding 100, and then storing the result to y, each time y is called, *Mathematica* will go back, check what the **current** value of x is, and then add 100 and store the result in y.

```
y
```

```
101
```

```
x = 2;
y
```

```
102
```

We will come back to delayed assignment when we introduce functional notation.

Substitution

There is yet another way to define variables and substitute values into expressions. The last way we will look at today is something called “substitution”. The special syntax

```
expression /. substitutionRule
```

tells *Mathematica* to apply the substitution rules given in substitutionRule to the expression. For example

```
Clear[a, b, c]
quadSoln = (-b + Sqrt[b^2 - 4 a c]) / (2 a);
quadSoln /. {a → 1, b → 2, c → 3}
quadSoln /. {a → 1, b → 2, c → 3.}
```

$$-2+i\sqrt{2}$$

```
-5.30958
```

This allows us to change the values of a,b, and c without having to use delayed assignment or re-updating the definition of the quadSoln. In general, the notation

```
x → y
```

is used to represent a rule specifying that x is to be replaced by y.

Predefined functions, constants, and help

As mentioned before, there are literally thousands of built in functions and constants in *Mathematica*. Thankfully *Mathematica* also has a very extensive and very thorough help menu. This help menu is much more useful than Matlab's since *Mathematica* includes many examples that can be evaluated and edited in the help menu itself. To access the help menu. Use the same "F1" option while your courser is on the function of interest. Alternately, you can press F1 anywhere, and then use the search bar. Press F1 while your cursor is over the following function

```
ArcTan
```

Just for the fun of it, here are a number of built in constants. Some of these are written by pressing the escape key, entering a key combination, and then pressing the escape key again. For example, Pi is entered by pressing **esc+pi+esc**.

```
N[{π, ⅇ, ⅈ, C, GoldenRatio, EulerGamma}]
```

```
{3.14159, 2.71828, 0. + 1. ⅈ, 0.915966, 1.61803, 0.577216}
```

And here a number of common math functions

```
{Log[x], Exp[x], Abs[x], Sqrt[x], Sin[x], ArcSin[x], Sinh[x]} /. x → 1
```

$$\left\{0,\ e,\ 1,\ 1,\ \text{Sin}[1],\ \frac{\pi}{2},\ \text{Sinh}[1]\right\}$$

Free form input

One relatively recent innovation in *Mathematica* is the inclusion of free-form input. This allows you to use natural language to enter commands into *Mathematica.* In fact, Apple's Siri uses this feature whenever you ask her a very specific question like "How many seconds in a day". This can be done directly in *Mathematica* by prefacing a command with a single "="

How many seconds in one day?

Result

86 400 s

86 400 s

This contacts Wolfram's servers (via Wolfram|Alpha) and basically does some speech processing and then a database lookup to return your results. Since this computation is not done locally, you aren't guaranteed to always get the same results.Sometimes, however, this free-form input can be converted back into standard *Mathematica* language.

Homework Problems

#1.

Order the following five numbers from smallest to largest

$$7.149\,\pi,\quad \pi^{e},\quad e^{\pi},\quad \left(\frac{1+\sqrt{5}}{2}\right)^{11\,\pi/5},\quad (2+e)^{\pi-1}$$

#2.

Assign a variable to each of the following symbolic expressions. After verifying that you have entered the expression correctly, use the substitution syntax

```
varible /. x → {}
```

to evaluate it at the indicated value. Then give a 10-digit numerical approximation of the expression.

- $\sqrt{\frac{e^{\sin(x+1)}}{\cos(x)+1}}$, at x=0
- e^{x^3}, at x=2
- $\frac{\sqrt{16-x^2}+1}{2x}$, at x=3
- $|4\cos(x)+\pi|$, at x=π

#3.

The following expression is a well-known mathematical constant. Use *Mathematica* to figure out which constant it is equal to. Can you use a built in function to prove that your guess is correct?

$$4 \tan^{-1}\left(\frac{1}{4}\right) + 4 \tan^{-1}\left(\frac{3}{5}\right)$$

#4.

Find the smallest positive integer *n* that satisfies

$$n! > n^{10} + n^6 + 10$$

#5.

Stirling' s formula provides an approximation for *n!,* in the form of

$$\frac{\sqrt{2\,\pi\,n}\;\; n^n}{e^n}$$

Using substitution, evaluate this formula for the values of n=20,40,80, and 160. Compare your results for 20!, 40!, 80! and 160!.

#6.

Evaluate the quantity

$$\frac{\left|\frac{\sqrt{2\,\pi\,n}\;\; n^n}{e^n} - n!\right|}{n!}$$

for the values of n give in problem #5. Why can we say that “Stirling’s formula approximates *n!* with an error on the order of *1/n*”?

#7.

Is the following a good approximation for Log[n]?

$$n\left(n^{1/n} - 1\right)$$

#8.

Find the integral power of 3 that is closest to each of one million, one billion, and one trillion.

#9.

Get Mathematica to tell you the following:
a) What the number of stars in our galaxy is
b) The height of the tallest person
c) The US’s per capita beer consumption (compare this to the highest beer consuming country in the

world via *Mathematica*).

#10.

Using *Mathematica*'s free form input, figure out if there are enough people in the USA to have them stand on each others' heads to reach from the earth to the moon (assuming that everyone is of average height).

Lists, Vectors, and Matrices

Collections of numbers

Lists/Vectors/Sets

We have already seen lists being used a bit, but let's formalize it. Here's a list of the first 5 natural numbers

```
myList = {1, 2, 3, 4, 5}
```

```
{1, 2, 3, 4, 5}
```

I could have used the following function to create this table

```
myList = Table[i, {i, 1, 5}]
```

```
{1, 2, 3, 4, 5}
```

This "Table" function is the primary way to create lists in *Mathematica*. Here's a list of the first 10 primes

```
tenPrimes = Table[Prime[n], {n, 1, 10}]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

To access elements of a list, you use double bracket notation. The notation is

```
tenPrimes[[start ;; end ;; increment]]
```

with increment being an optional argument. Note that the ";;" notation is referred to as a "Span"

```
tenPrimes[[1 ;; 10]]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

You can also omit "start" or "end" if you want it to be the first element or the last element respectively

```
tenPrimes[[ ;; 10]]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

Mathematica is quite flexible in that the same data structure (a list) is used to represent vectors, lists, and sets. Hence if you want to create a vector of ones, you can do this using

```
Table[1, {i, 1, 10}]
```

```
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

But lists can also contain other elements than just numbers. For example

```
{1, "Test", a, , π}
```

is a list that contains a number of different data types (integers, strings, symbols, images, etc).

To perform element-wise elementary math operations on a list of numbers, you simply use the standard operations (-,+,*, etc) and make sure that the dimensions match. However, you can combine scalars and vectors with no issue.

```
{1, 2, 3} + {4, 5, 6}
{1, 2, 3} * {4, 5, 6}
10 * {1, 2, 3}
10 + {1, 2, 3}
{1, 2, 3}^2
```

```
{5, 7, 9}
```

```
{4, 10, 18}
```

```
{10, 20, 30}
```

```
{11, 12, 13}
```

```
{1, 4, 9}
```

Other set-like operations include built-in functions like:

```
Union
Intersection
Complement
Join
Tally
DeleteDuplicates
```

Take a look at the help entry for each one of these functions.

Lists of Lists/Matrices

To represent matrices, *Mathematica* uses lists of lists. These are listed row-wise

```
mymat = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
mymat // MatrixForm
Dimensions[mymat]
```

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

```
{3, 3}
```

Matrix indexing is once again done using the double square brackets [[ ]]. Hence you would access

elements in the following way:

```
mymat[[1, 2]] (*First row, second column*)
mymat[[2, 3]] (*second row, third column*)
```

```
2
```

```
6
```

You can also access multiple indices by passing a list to the index notation.

```
mymat[[{1, 2}, {1, 3}]]
```

```
{{1, 3}, {4, 6}}
```

You can use the span notation for matrices too

```
mymat[[ ;; , 3]] (*Just the third column*)
```

```
{3, 6, 9}
```

Unlike Matlab, *Mathematica* does not have linear indexing. On the contrary, when giving [[ ]] a single argument, it interprets it as if you are asking for the first “level” of elements. From this view point, mymat is a list with three elements: {1,2,3}, {4,5,6}, and {7,8,9}. So the second element would be {4,5,6}

```
mymat[[2]]
```

```
{4, 5, 6}
```

This hierarchical viewpoint is very important to grasp (as lists are the main data structure in *Mathematica*). Let’s now look at a more complicated example

```
jaggedList = {1, {1, 2, 3}, {{π, e}}};
```

This list has length 3

```
Length[jaggedList]
```

```
3
```

The first element is the number 1

```
jaggedList[[1]]
```

```
1
```

Whereas the second element is the list {1,2,3}

```
jaggedList[[2]]
```

```
{1, 2, 3}
```

Since jaggedList[[2]] is itself a list, we can access its third element (a 3)

```
jaggedList[[2]][[3]]
```

```
3
```

The following is just shorthand for the above

```
jaggedList[[2, 3]]
```

```
3
```

In this same line of reasoning, the third element of jaggedList is a list of lists: {{π,e}}. We can access the first item of this list {π,e}, and the second item of that list: e.

```
jaggedList[[3, 1, 2]]
```

e

While there are many ways to create a matrix, here are a few of the most common

```
IdentityMatrix[5] // MatrixForm
DiagonalMatrix[Range[10]] // MatrixForm
Table[1 / (i + j - 1), {i, 1, 5}, {j, 1, 5}] // MatrixForm (*Hilbert Matrix*)
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 10 \end{pmatrix}$$

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

Apply and Map

Since *Mathematica* is very “list oriented”, many of its operations are made quicker if vectorized. Recall that vectorization is the process of performing the same function or operation to every element of a list. Most of *Mathematica*’s built in functions are already vectorized. For example, to perform the absolute value function on each one of the elements of a list, we would use the following syntax

```
Abs[{1, -1, 10, -.1, 34}]
```

```
{1, 1, 10, 0.1, 34}
```

Note that the Abs[ ] function is automatically performed on each element of the list. This is what it means to be automatically vectorized.
Another way to create vectorization, is to use the Map function. The function Map take a function and a list as input, and then performs the function on each on of the elements in the list. Here’s an example

```
Clear[f, mylist]
mylist = {a, b, c};
Map[f, mylist]
```

```
{f[a], f[b], f[c]}
```

So you can see that f has been applied to each element of {a,b,c}. This will be especially useful once we learn how to create anonymous functions (see the next section). There is some shorthand notation for Map that you may see me use:

```
f /@ {a, b, c}
```

```
{f[a], f[b], f[c]}
```

This notation is chosen to make it look like f is being “mapped” to each one of the elements of the list. Once again, after we learn anonymous functions, the power of this kind of notation will become clear.

The Apply function is similar to Map, but instead of performing the same operation on each element of a list, Apply gives the entire list to the function :

```
Apply[f, {a, b, c}]
```

```
f[a, b, c]
```

For example, this can be used to quickly multiply all the elements of a list together

```
Apply[Times, {1, 2, 3, 5, 6}]
```

```
180
```

Linear Algebra Functions

The linear algebra capabilities of Mathematica aren’t as robust as Matlab’s, but once can argue that more built-in functions and routines exist in Matlab. Let’s look at a few of the most commonly used functions

```
mymat1 = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
mymat2 = {{0, 1, 0}, {0, -1, -1}, {1, 1, 1}};
myvector1 = {1, 2, 3};
mymat1 // MatrixForm
mymat2 // MatrixForm
myvector1 // MatrixForm
```

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

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & -1 & -1 \\ 1 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

The " * " operator stands for element-wise multiplication

```
mymat1 * mymat2 // MatrixForm
```

$$\begin{pmatrix} 0 & 2 & 0 \\ 0 & -5 & -6 \\ 7 & 8 & 9 \end{pmatrix}$$

Whereas the Dot function performs matrix multiplication (and matrix vector multiplications

```
Dot[mymat1, myvector1] // MatrixForm
```

$$\begin{pmatrix} 14 \\ 32 \\ 50 \end{pmatrix}$$

```
Dot[mymat1, mymat2] // MatrixForm
```

$$\begin{pmatrix} 3 & 2 & 1 \\ 6 & 5 & 1 \\ 9 & 8 & 1 \end{pmatrix}$$

Inverses are calculated using

```
Inverse[mymat1](*Note the helpful error message*)
Inverse[mymat2] // MatrixForm
```

Inverse::sing : Matrix {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}} is singular. >>

```
Inverse[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}]
```

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ -1 & -1 & 0 \end{pmatrix}$$

And you can solve matrix equations using

```
soln = LinearSolve[mymat1, myvector1] (*Solves Ax==b for x*)
```

$$\left\{-\frac{1}{3}, \frac{2}{3}, 0\right\}$$

```
Dot[mymat1, soln] == myvector1
```

```
True
```

Here are a smattering of other useful linear algebra functions

```
MatrixForm /@ N[SingularValueDecomposition[mymat2]]
```

$$\left\{\begin{pmatrix} 0.327985 & -0.736976 & -0.591009 \\ -0.591009 & 0.327985 & -0.736976 \\ 0.736976 & 0.591009 & -0.327985 \end{pmatrix},\right.$$
$$\left.\begin{pmatrix} 2.24698 & 0. & 0. \\ 0. & 0.801938 & 0. \\ 0. & 0. & 0.554958 \end{pmatrix}, \begin{pmatrix} 0.327985 & 0.736976 & -0.591009 \\ 0.736976 & -0.591009 & -0.327985 \\ 0.591009 & 0.327985 & 0.736976 \end{pmatrix}\right\}$$

```
MatrixForm /@ N[QRDecomposition[mymat2]]
```

$$\left\{\begin{pmatrix} 0. & 0. & 1. \\ 0.707107 & -0.707107 & 0. \\ -0.707107 & -0.707107 & 0. \end{pmatrix}, \begin{pmatrix} 1. & 1. & 1. \\ 0. & 1.41421 & 0.707107 \\ 0. & 0. & 0.707107 \end{pmatrix}\right\}$$

```
Eigenvalues[mymat1]
```

$$\left\{\frac{3}{2}\left(5+\sqrt{33}\right),\ \frac{3}{2}\left(5-\sqrt{33}\right),\ 0\right\}$$

```
Eigenvectors[mymat1] // MatrixForm
```

$$\begin{pmatrix} -\frac{-7-\sqrt{33}}{2\left(11+2\sqrt{33}\right)} & -\frac{-29-5\sqrt{33}}{4\left(11+2\sqrt{33}\right)} & 1 \\ -\frac{7-\sqrt{33}}{2\left(-11+2\sqrt{33}\right)} & -\frac{29-5\sqrt{33}}{4\left(-11+2\sqrt{33}\right)} & 1 \\ 1 & -2 & 1 \end{pmatrix}$$

```
MatrixForm[Chop[N[#]]] & /@ Eigensystem[mymat2]
```

$$\left\{\begin{pmatrix} -1. \\ 0.5+0.866025\,i \\ 0.5-0.866025\,i \end{pmatrix},\ \begin{pmatrix} -1. & 1. & 0 \\ 0.+0.57735\,i & -0.5+0.288675\,i & 1. \\ 0.-0.57735\,i & -0.5-0.288675\,i & 1. \end{pmatrix}\right\}$$

Homework Problems

#1.

Define a list which gives the first 8 odd numbers. Give this list the name oddList. Try the following commands and comment on what they do.

```
Reverse[oddList]
Append[oddList, 0]
FullForm[oddList]
Length[oddList]
Delete[oddList, 4]
Insert[oddList, 100, 3]
oddList = ReplacePart[oddList, 3 → -1]
Take[oddList, 4]
Drop[oddList, 4]
RotateLeft[oddList, 5]
RotateRight[oddList]
```

#2.

Create a list of random numbers (between 0 and 1) with length given by the 123rd prime number. Assign this list to a variable, and then calculate the mean, total, and standard deviation of this list. Next, extract every 3rd item in this list and calculate the mean of this list.

#3.

type the following to define a very complex “list of lists of lists of...”

```
multiList = {{{{{1, 2, 3 {{4, 5}}}, 6}}, {7, 8}}, 9};
```

Test the effect of the following command and comment on the behavior.

```
Flatten[multiList]
Flatten[multiList, 1]
Flatten[multiList, 2]
```

Then set

```
flatList = Flatten[multiList];
```

and try

```
Partition[flatList, 3]
Partition[flatList, 4]
Partition[flatList, 3, 1]
```

comment on the effect of these commands.

#4.

Given that

```
list1 = {1, 2, 3, 4, 5, 6};
list2 = {5, 6, 7, 8, 9, 10};
```

Use *Mathematica* to find the intersection, union, and symmetric difference of these sets.

#5.

Use “Apply[ ]” to find the product of the first 100 primes.

#6.

Use “Map[ ]” to find the derivatives of the following functions, returning them as a list given in the same order

$$\{x^2, \log(x), \cos(x)\}$$

#7.

Create a list of 10^7 random integers between 0 and 9. Add up all the elements of this list in three different ways:
1. Using the built-in Total function.
2. Using Apply and Plus
3. Using the Sum function
Verify that all three approaches return the same result. Next, using the Timing function, test which method is fastest.

#8.

Count how many of the first N=10,000 natural numbers are square free (that is, they contain no prime

factor of the form p^2). Make a conjecture about what happens to the fraction of square free numbers as N goes to infinity.

#9.

The Perron-Frobenius Theorem is a very useful theorem that asserts that a real square matrix with positive entries has particular constraints on its spectrum (eigenvalues and vectors). The theorem is often used in dynamical systems and probability. One of the consequences of the theorem is that for particular matrices (called irreducible and non-negative), the following sequence has a certain behavior

$$\left\{ \frac{A^k}{r^k} \right\}_{k=1\ldots\infty}$$

Where “r” is the largest eigenvalue of “A”. Using the matrix

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

(which is irreducible and nonnegative), conjecture about what happens to the above sequence.

Functions

How to define and use functions

While *Mathematica* has many built-in functions, we will now focus on the syntax required for you to define your own functions. There are a couple different ways you can do this, and we will focus on the Module function, delayed assignment functions, and anonymous functions.

Delayed assignment functions

In *Mathematica,* you can use a variant of the familiar f(x) notation to define a function. Let’s look at a simple example

```
f[x_] := x^2
```

The "x_" argument is a pattern telling *Mathematica* to treat “x” as a value to be specified, not as a usual variable. The fact that the syntax has been highlighted green indicates that “x” is treated locally (so if “x” was assigned a value elsewhere, this won’t effect the function definition). You can call the function using the following syntax.

```
f[1]
f[2]
f[3]
```

```
1

4

9
```

Here's an example emphasizing the local nature of the variable "x"

```
x = 1;
f[x_] := x^2
f[2]
Clear[x]
```

```
4
```

So you see that declaring "x=1" doesn't effect the definition of the function "f".
Note that this function will work on symbolic arguments as well:

```
Clear[y]
f[y + 1]
```

$$(1 + y)^2$$

This leads to quite a bit of flexibility. For example, function composition is as easy as

```
f[x_] := x^2
g[x_] := 1 / x
f[g[x]]
```

$$\frac{1}{x^2}$$

If you want to define a function with more than one variable, we need only separate the arguments with commas

```
Clear[f]
f[x_, y_] := x^2 + y^2
f[3, 4]
```

```
25
```

Piecewise defined functions have a similar notation

```
Clear[f]
f[x_] := Piecewise[{ {1, x ≥ 0}, {0, x < 0} }]
(*one for nonnegative x, zero for negative x*)
g[x_] := Piecewise[{{1, x ≠ 0}, {0, x == 0}}]
(*One everywhere except at zero, where the function is zero*)
{f[-1], f[0], f[1]}
{g[-1], g[0], g[1]}
```

```
{0, 1, 1}
```

```
{1, 0, 1}
```

Let's look at an application of the delayed function assignment notation.

Suppose we want to write a program to create a random walk in one dimension. We create a vector of random real numbers between -1 and 1.

```
RandomReal[{-1, 1}, {12}]
```

```
{-0.642249, 0.37087, 0.0677216, 0.597011, -0.742312, 0.246464,
 -0.820805, 0.956329, 0.979461, -0.451849, 0.258094, 0.32887}
```

And we can use the "accumulate" function to create running sums.

```
Accumulate[{a, b, c, d, e}]
```

```
{a, a + b, a + b + c, a + b + c + d, a + b + c + d + e}
```

Putting those together, we can have a single command that yields a one-dimensional random walk.

```
Accumulate[RandomReal[{-1, 1}, {12}]]
```

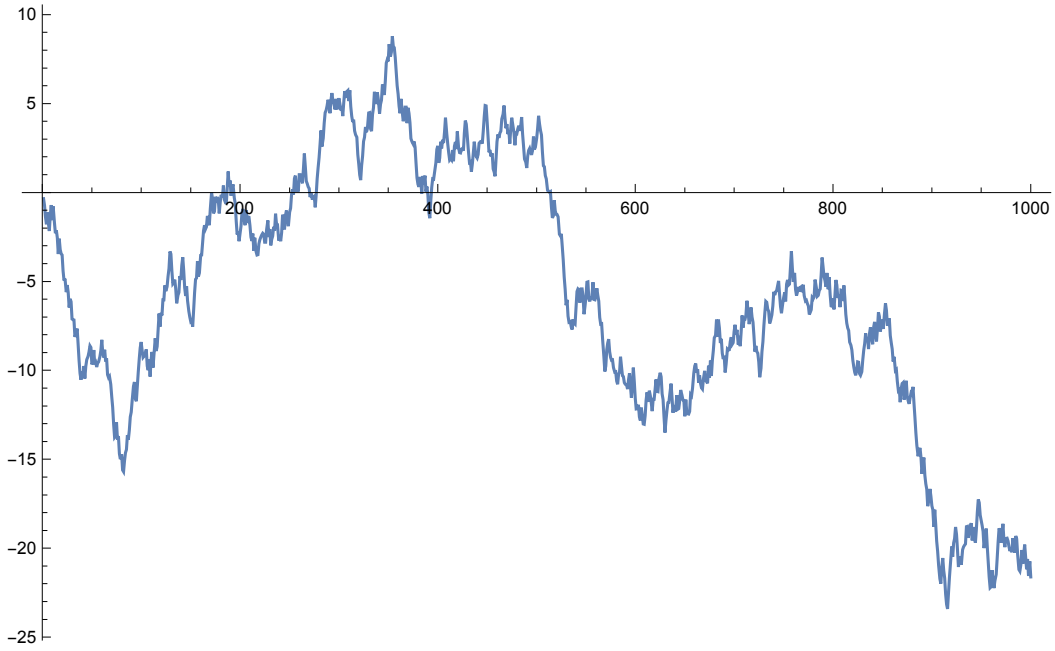
```
{0.897681, 0.846868, -0.130453, 0.586121, 0.250751, 0.281828,
 0.0477735, 0.974423, 0.403668, 0.525962, 0.393203, -0.45754}
```

We can turn this into a function by which we can specify the number of steps.

```
myRandomWalk[steps_] := Accumulate[RandomReal[{-1, 1}, {steps}]]
```

Let's plot a large one-dimensional random walk.

```
ListLinePlot[myRandomWalk[1000]]
```



On average, how far does the walk stray from the origin? We can look at the last position of a 1000-step walk.

```
Last[myRandomWalk[1000]]
```

```
-10.4917
```

Run 10 trials, and then average.

```
Table[Last[myRandomWalk[1000]], {i, 1, 10}]
Mean[%]
```

```
{10.1887, 2.36982, -20.6535, -10.9255,
 -10.0927, -6.05395, -9.57671, -5.96947, 3.34893, 12.2657}

-3.50987
```

Let's turn this into a function...

```
myRWMean[steps_, trials_] := Mean[Table[Last[myRandomWalk[steps]], {trials}]]
```

Now we can look at 10,000 trials for a 1000-step walk...

```
myRWMean[1000, 10 000]
```

```
-0.147895
```

Anonymous Functions

Anonymous functions are handy if you need to use a bit of shorthand to define a function (and don't need to actually give it a name). Anonymous functions use the following syntax

```
(#^2) &
```

This is a function representing the f(x)=x^2 function. Here, the "#" stands for the variable (and is called

a “slot”), and the & denotes the end of the function. We could call this using

```
(#^2) &[1]
(#^2) &[2]
```

```
1
```

```
4
```

If you want an anonymous function of multiple variables, you can name your individual slots

```
(Sqrt[#1] + #2^2) &[1, 2]
```

```
5
```

In general, “#n” stands for the nth function argument. If you want to refer to ALL the function arguments, you would use “##”

```
(Length[List[##]]) &[1, 2, 3, 4]
(Length[List[##]]) &[1, 2, 3, 4, 5, 6]
```

```
4
```

```
6
```

Anonymous functions are especially handy when using the Map function. For example, say we want to perform the same function to every element of a list. This could be performed using

```
Mean[Map[#^2 + # + 1 &, RandomReal[{0, 1}, 10]]]
```

```
2.0261
```

So this expression generates 10 random numbers between 0 and 1, and then evaluates the function f(x)=x^2+x+1 on each one of them, then takes the average of this result.

Remember the Map function from last time? Let’s look at a few more in depth examples now that we know how to make anonymous functions

```
Map[#^2 &, {1, 2, 3}]
```

```
{1, 4, 9}
```

Now the shorthand notation should become more clear

```
#^2 & /@ {1, 2, 3}
```

```
{1, 4, 9}
```

Even though this may look a bit cumbersome, it’s amazing how useful this will come in handy. For example

```
StringReplace[#, "e" → "E"] & /@ {"Here", "is", "a", "list", "of", "text", "words"}
```

```
{HErE, is, a, list, of, tExt, words}
```

is much shorter (and clearer, in my opinion) than:

```
listOfWords = {"Here", "is", "a", "list", "of", "text", "words"};
Table[StringReplace[listOfWords[[i]], "e" → "E"], {i, 1, Length[listOfWords]}]
```

```
{HErE, is, a, list, of, tExt, words}
```

Here is a fun little example of repeatedly nesting a function. First, we need to quickly look at the

NestList function:

```
Clear[f, x]
NestList[f, x, 5]
```

```
{x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]], f[f[f[f[f[x]]]]]}
```

What the NestList function does is repeatedly apply a function to an argument, and return the results in a list.

```
NestList[z^# &, z, 5]
```

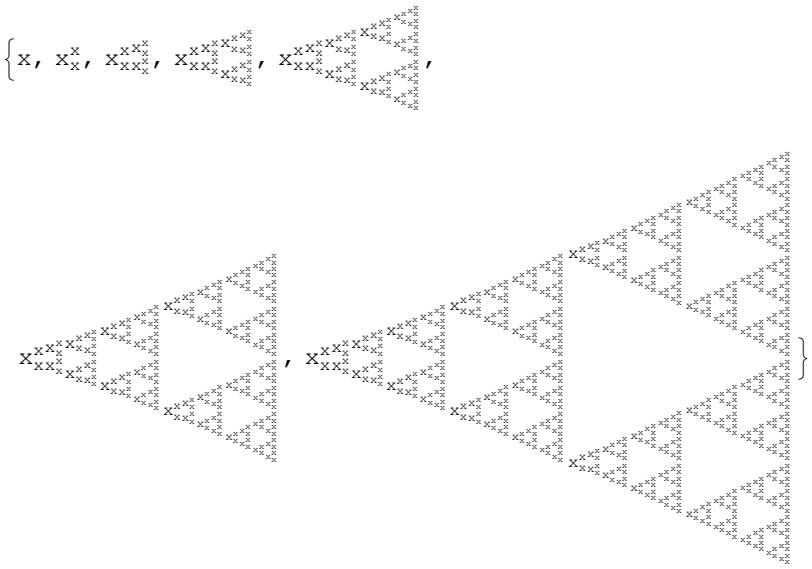
$$\left\{z,\ z^{z},\ z^{z^{z}},\ z^{z^{z^{z}}},\ z^{z^{z^{z^{z}}}},\ z^{z^{z^{z^{z^{z}}}}}\right\}$$

We can actually make fractals quite quickly using this notation. First, note what Subsuperscript does

```
Subsuperscript[a, b, c]
```

$$a_b^c$$

```
NestList[Subsuperscript[#, #, #] &, x, 6]
```



The "Module" function

When you are writing a more involved function, it is best to use the Module function. A Module allows you to specify which variables are to be treated as local variables. As a simple example, consider

```
Clear[f]
f[x_] := Module[{}, x^2]
f[1]
f[2]
```

```
1
```

```
4
```

Here, I have not specified any local variables (except "x"), and so the first argument of Module is blank. However, if we are trying to use Heron's Formula for the area of a triangle with sides of length a, b, and

c:

$$\sqrt{\left(\frac{1}{2}(a+b+c)\right)\left(\frac{1}{2}(a+b+c)-a\right)\left(\frac{1}{2}(a+b+c)-b\right)\left(\frac{1}{2}(a+b+c)-c\right)}$$

Then it would be easier to define an intermediate variable for the expression 1/2(a+b+c)

```
Clear[heron, a, b, c, s]
heron[a_, b_, c_] := Module[{s},
  s = (a + b + c) / 2;
  Sqrt[s (s - a) (s - b) (s - c)]
 ]
heron[3, 4, 5]
```

```
6
```

Here I have defined “s” to be an intermediate local variable. Note that “s” has not been given any value

```
s
```

```
s
```

Contrast this behavior to when we use the Function function

```
Clear[heron2, a, b, c, s]
heron2 = Function[{a, b, c},
   s = (a + b + c) / 2;
   Sqrt[s (s - a) (s - b) (s - c)]
  ];
heron2[3, 4, 5]
```

```
6
```

```
s
```

```
6
```

Here s has not been declared to be a local variable, and so will get a value assigned to it each time the function heron2 is called

```
heron2[10, 11, 12]
s
```

$$\frac{33\sqrt{39}}{4}$$

$$\frac{33}{2}$$

Other Ways

There a MANY ways to define functions in *Mathematica*. Here are a few examples of how to define the factorial function

```
f[n_] := n!
```

```
f[n_] := n f[n - 1]; f[1] = 1
```

```
f[n_] := Times @@ Range[n]
```

```
f[n_] := ∏_i^n i
```

```
f[n_] := Length[Permutations[Range[n]]]
```

```
f[n_] := Fold[Times, 1, Range[n]]
```

```
f[n_] := Module[{t = 1}, Do[t = t i, {i, n}]; t]
```

```
f[n_] := Module[{t = 1, i}, For[i = 1, i ≤ n, i++, t *= i]; t]
```

```
f[n_] := If[n == 1, 1, n f[n - 1]]
```

```
f = If[#1 == 1, 1, #1 #0[#1 - 1]] &
```

```
f[n_] := Fold[#2[#1] &, 1, Array[Function[t, #1 t] &, n]]
```

```
f[n_] := First[{1, n} //. {a_, b_ /; b > 0} :→ {b a, b - 1}]
```

Homework Problems

#1.

Define two functions

$$f(x) = 3x + 19$$

$$g(x) = \frac{x - 19}{3}$$

And use *Mathematica* to algebraically demonstrate that they are inverses of each other.

#2.

An ellipse with a semi-major axis of length “a” and a semi-minor axis of length “b” has a perimeter of approximately

$$f(a, b) = 2\pi \sqrt{\frac{1}{2}\left(a^2 + b^2\right)}$$

Define this function in *Mathematica* and use it to approximate the perimeter of an ellipse having a major axis of length 5 and a minor axis of length 4.

What is the average perimeter for 10,000 ellipses with randomly distributed (on [0,1]) semi-major/minor axes?

#3.

The length of a parabolic arc from (0,0) to (x,y) along the parabola y^2=x is given by the arc length function “f” for points (x,y)=(x,Sqrt[x]) on the curve in the first quadrant by writing

$$f(x,\, y) = \frac{1}{2}\left(\frac{y^2 \log\left(\frac{u+2\,x}{y}\right)}{2\,x} + u\right)$$

where

$$u = \sqrt{4\,x^2 + y^2}$$

Write a function f(x,y) that computes the length of the arc from (0,0) to (x,y), according to the formula above. Use the Module function to define f. Next, use this function to evaluate the arc length at the points (4,2) and (9,3).

#4.

The function

$$g(x) = \begin{cases} 2\,x & 0 \le x \le \frac{1}{2} \\ 2\,x - 1 & \frac{1}{2} \le x \le 1 \end{cases}$$

defined on [0,1] is called the baker’s transformation. Consider a glob of dough of length 1 that is to be kneaded in a certain fashion. If a point of the dough is a distance x from the end, with 0<x<1, let g(x) represent its position after kneading the dough once.
Define the function g, and using the built-in function NestList, determine the sequence of positions attained by the point of dough starting at x=1/10 through several repeat applications of g. Is there a pattern?

#5.

Create a function that searches for the first occurrence of string in π using the convention that “a”->1, “b”->2, etc. For example, to search for my name “David” I would look for the first occurrence of the pattern 4 1 22 9 4 (so the digits {4,1,2,2,9,4}).
This function should have two arguments: the first gives the pattern, and the second tells how far in π to look. For example, your syntax should look like

```
lookInPi[{4, 1, 2, 2, 9, 4}, 1 000 000]
(*Looks for my name in the first 1 Million digits of Pi*)
```

Given an arbitrary pattern, is it guaranteed that this function will return a result if allowed access to an arbitrary number of digits of π?

#6.

The logistic map is typically shown as the first example of how chaos can come from a very simple non-linear dynamical system. This dynamical system is described the in the following discrete manner

$$x_{n+1} = r\, x_n\, (1 - x_n)$$

Define the function

$$f(x,\, r) = r\, x\, (1 - x)$$

so that this dynamical system can be described by composing f with itself many times over. i.e. we can compose f with itself many times over

$$\{x,\, f(x,\, r),\, f(f(x,\, r),\, r),\, f(f(f(x,\, r),\, r),\, r),\ \ ...\}$$

and see what happens to the sequence of numbers.
a) If r=.1, what happens to this dynamical system in the long run?
b) if r=3.2, what happens to this dynamical system in the long run?

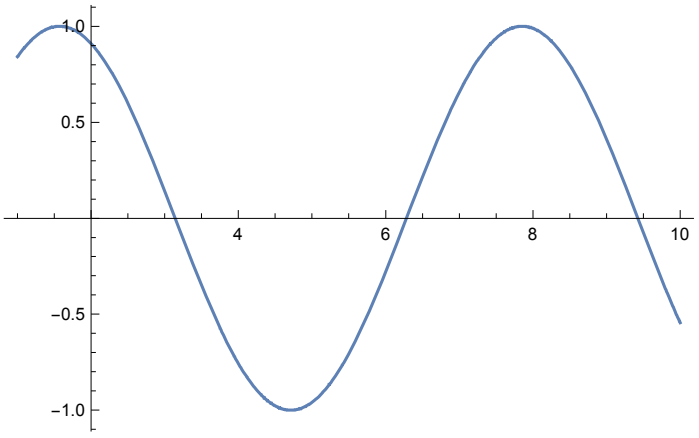
Graphics

Basic Plotting

Plotting 1D Functions

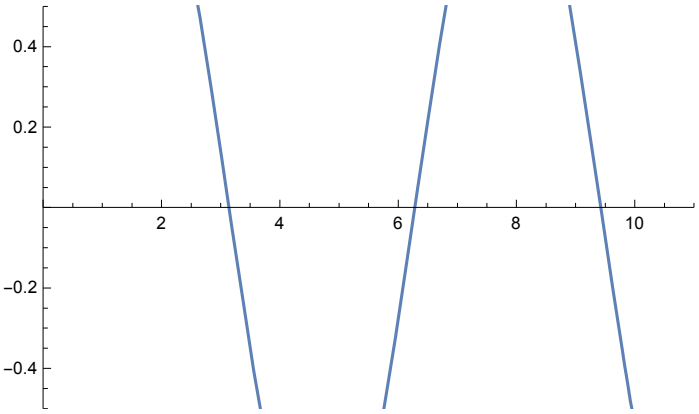
Mathematica has many ways to plot functions and data, while simultaneously automating many of the details (like specifying the y-axis, the scale, color aesthetic choices, sample rate, etc). Let's start out simple and continue from there. The basic syntax for plot is to supply it with a function of a single variable, and then specify what this variable is and what the range is.

```
Plot[Sin[x], {x, 1, 10}]
```

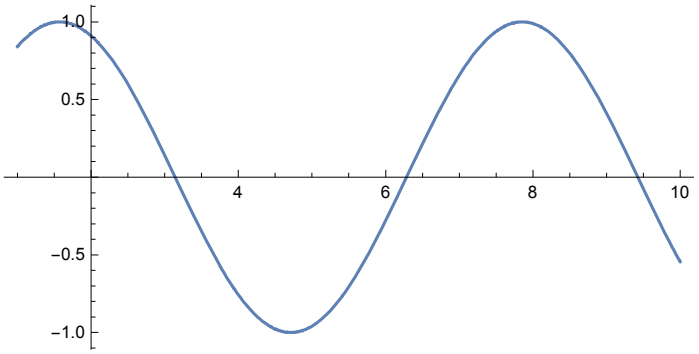


To specify the x and y range, use the PlotRange option

```
Plot[Sin[x], {x, 1, 10}, PlotRange → {{0, 11}, {-.5, .5}}]
```

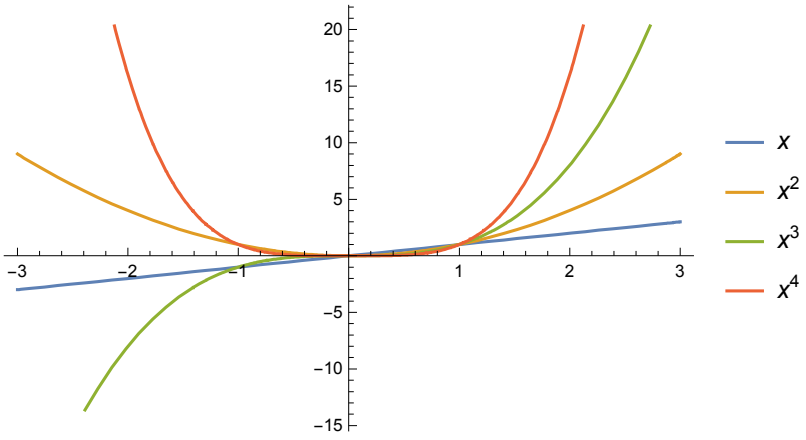



```
Plot[Sin[x], {x, 1, 10}, AspectRatio → 1 / 2]
```



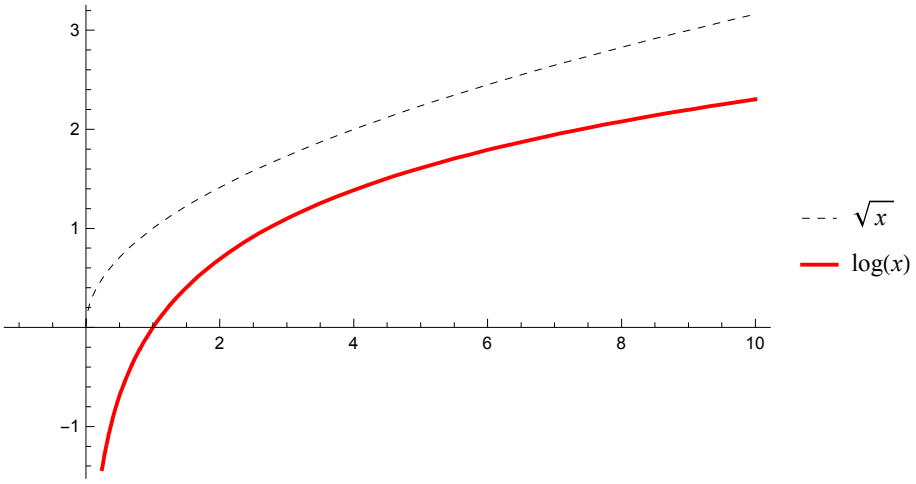
Multiple functions can be plotted together, *Mathematica* will automatically take care of the coloring, though you will have to specify if you want a legend or not.

```
Plot[{x, x^2, x^3, x^4}, {x, -3, 3}, PlotLegends → "Expressions"]
```



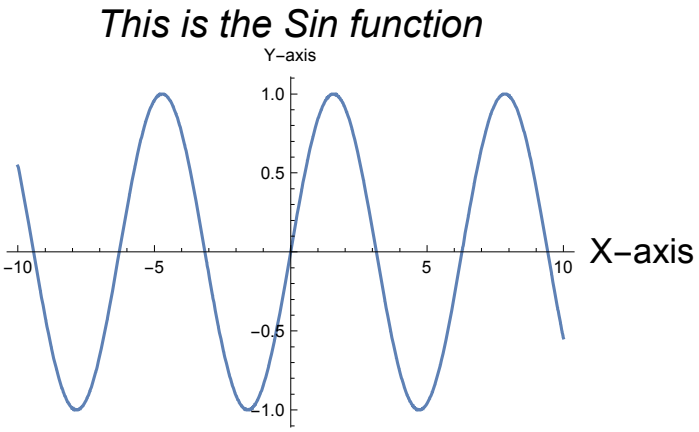
Note that we have not had to specify the exact x-values and y-value for *Mathematica*, it automatically does this for us. All we need to do is specify the function.
The “PlotStyle” option of Plot allows you to have finer control over the display details.

```
Plot[{Sqrt[x], Log[x]}, {x, -1, 10}, PlotStyle → {{Black, Dashed}, {Thick, Red}},
 PlotLegends → "Expressions", ImageSize → 400]
```



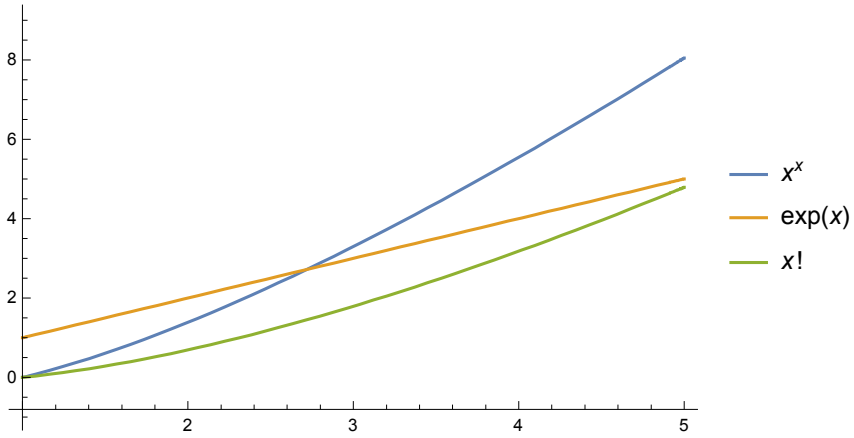
Labeling axes are straightforward as well

```
Plot[Sin[x], {x, -10, 10},
 PlotLabel → Style["This is the Sin function", Italic, FontSize → 20],
 AxesLabel → {Style["X-axis", FontSize → 18], "Y-axis"}]
```

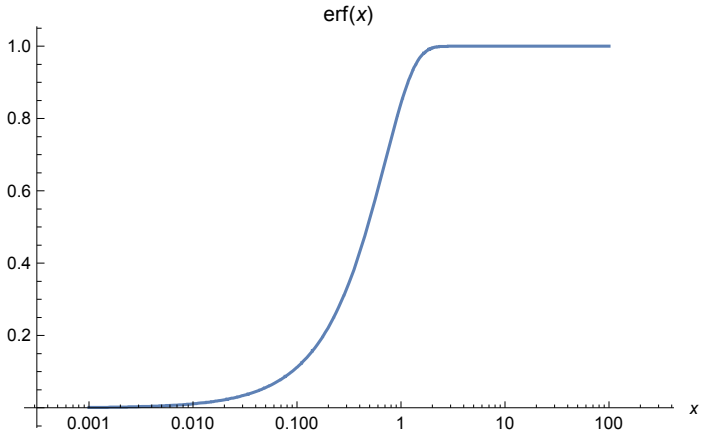


There are a number of other basic function visualization plots

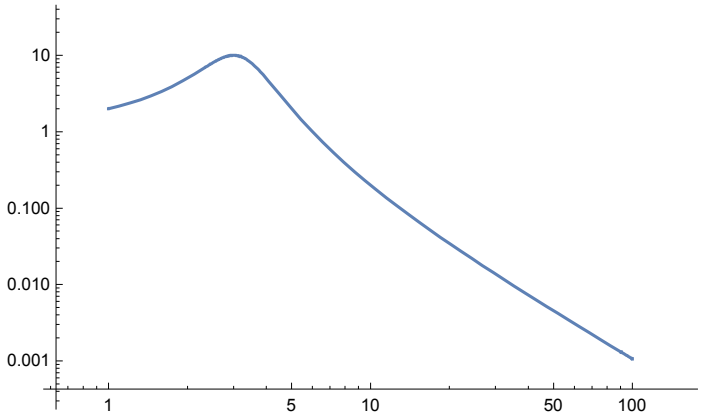
```
LogPlot[{x^x, Exp[x], x!}, {x, 1, 5}, PlotLegends → "Expressions"]
```




```
LogLinearPlot[Erf[x], {x, 10^-3, 10^2}, AxesLabel → Automatic, PlotLabel → Erf[x]]
```




```
LogLogPlot[10 / (x^2 - 6 x + 10), {x, 1, 100}]
```

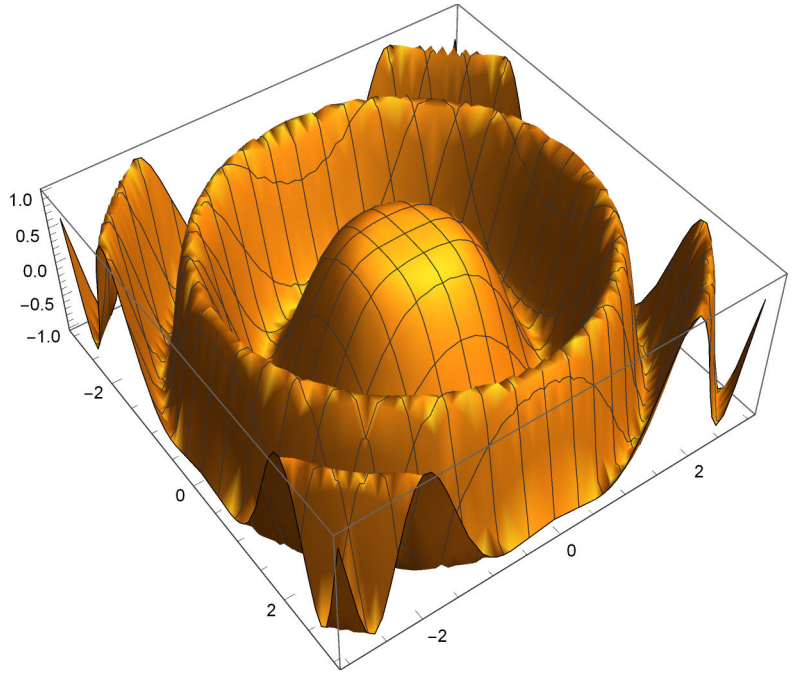


There are almost a limitless number of options you can modify for each plot, so we will not go into all the details here. Please consult the Help Menu for “Plot” to see more about modifying plots.

Plotting 2D Functions

Plotting functions of two variables is as easy as appending “3D” to the functions we just looked at.

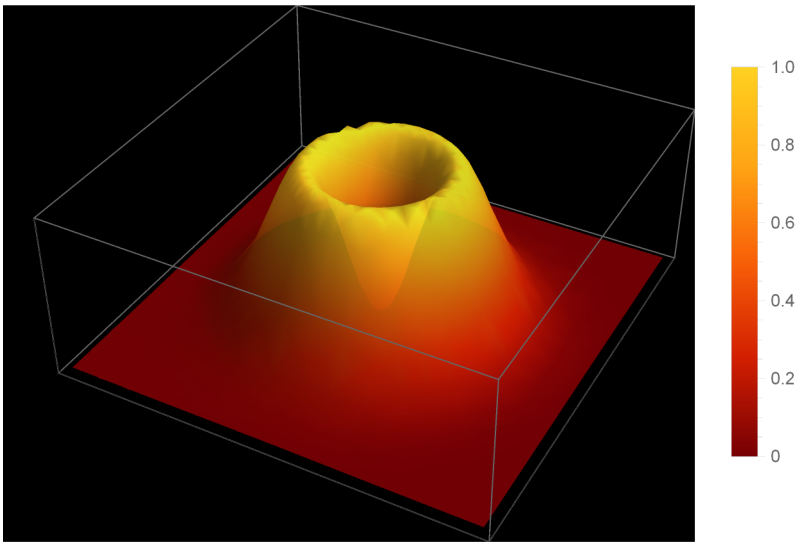
```
myfunc[x_, y_] := Cos[x^2 + y^2]
Plot3D[myfunc[x, y], {x, -3, 3}, {y, -3, 3}]
```



Note that you can immediately rotate and manipulate these plots.

Let's look at a particularly involved example

```
Plot3D[ (x^2 + y^2) Exp[1 - x^2 - y^2], {x, -3, 3}, {y, -3, 3},
 PlotStyle → Opacity[0.8], PerformanceGoal → "Quality", Mesh → None,
 ColorFunction → "SolarColors", PlotLegends → Automatic,
 Background → Black, TextureCoordinateFunction → ({#1, #3} &)]
```

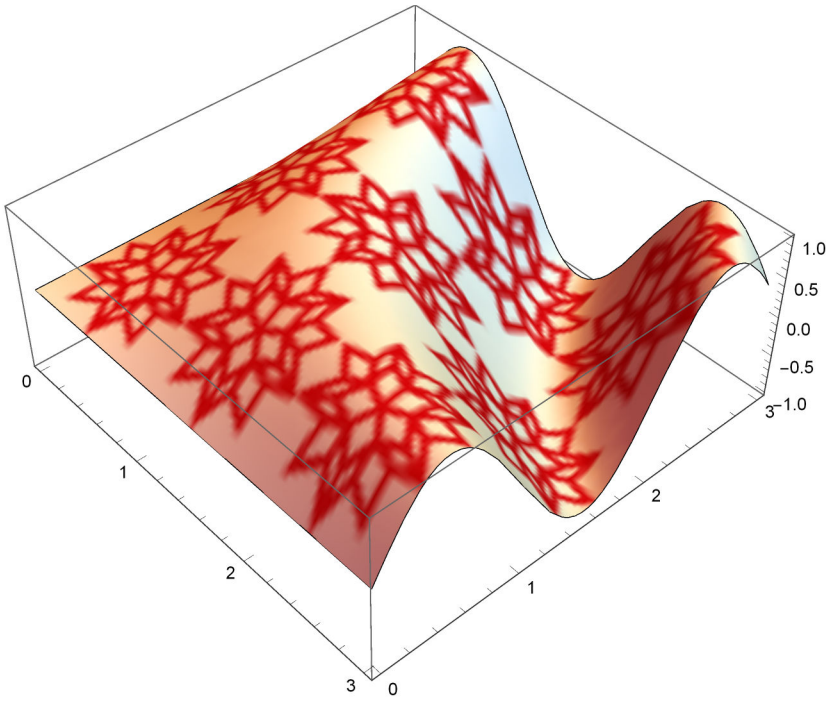


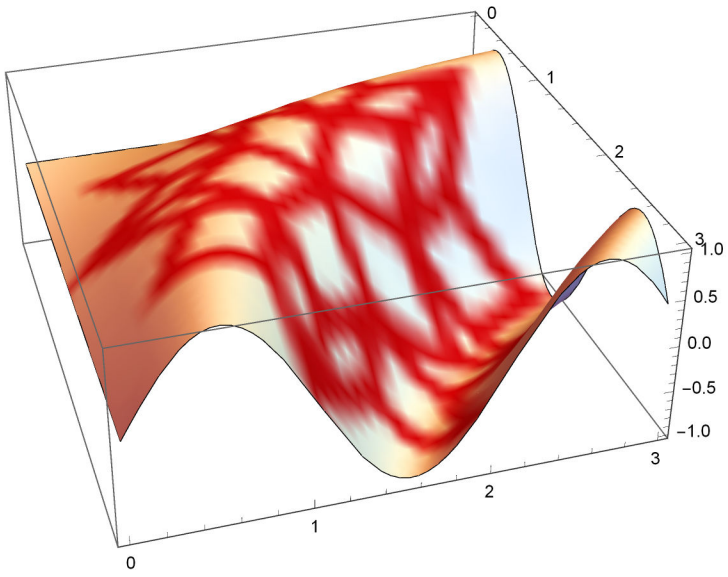
Another especially cool ability is to use custom textures to overlay your graphs.

```
texture =
```

```
;
```

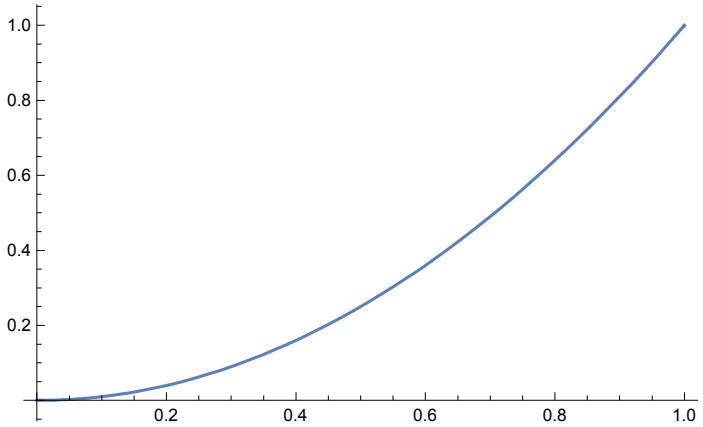
```
Plot3D[Sin[x y], {x, 0, 3}, {y, 0, 3}, Mesh → None,
 TextureCoordinateScaling → False, PlotStyle → Texture[texture]]
Plot3D[Sin[x y], {x, 0, 3}, {y, 0, 3}, Mesh → None,
 TextureCoordinateScaling → True, PlotStyle → Texture[texture]]
```



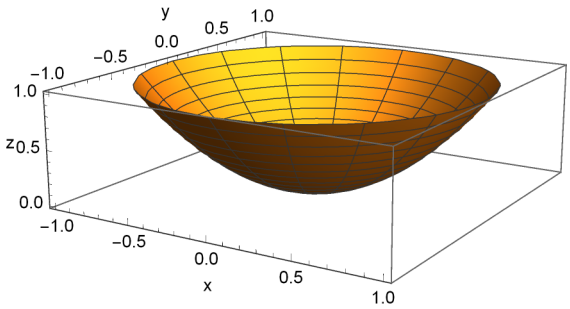


Remember surfaces of revolution from calculus? Well *Mathematica* can plot such surfaces with ease

```
Plot[x^2, {x, 0, 1}]
```



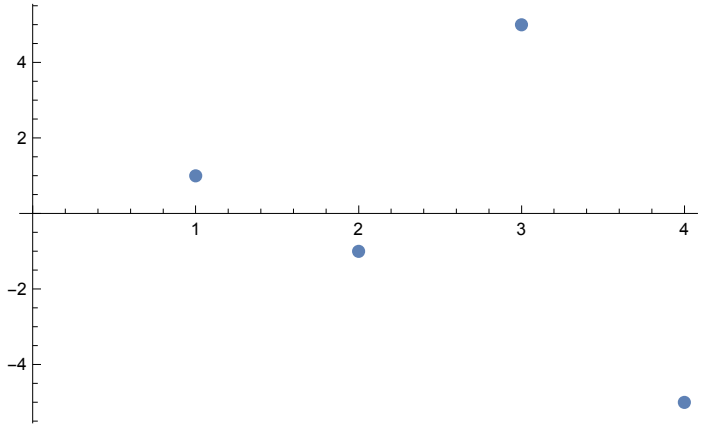

```
RevolutionPlot3D[x^2, {x, 0, 1}, AxesLabel → {"x", "y", "z"}]
```



Plotting Data

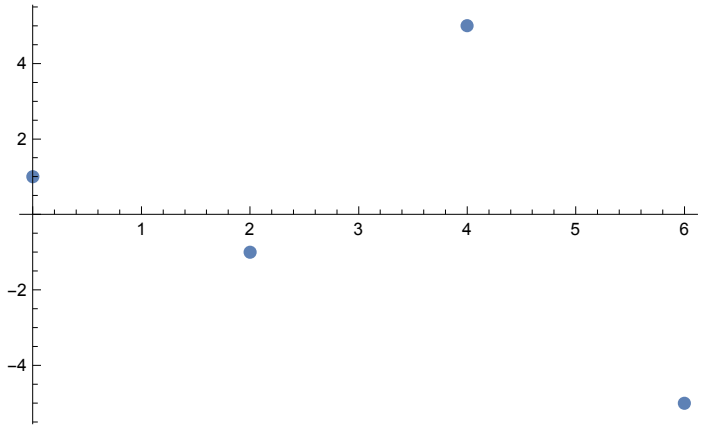
Sometimes instead of plotting a function, it's more useful to plot data points. This is where list plotting comes into play. Most of the basic plotting functions come in a "list" version as well. These work by treating the input as y-values and (if no x-values are specified), plotting them against {1,2,3,...}.

```
yVals = {1, -1, 5, -5};
ListPlot[yVals]
```



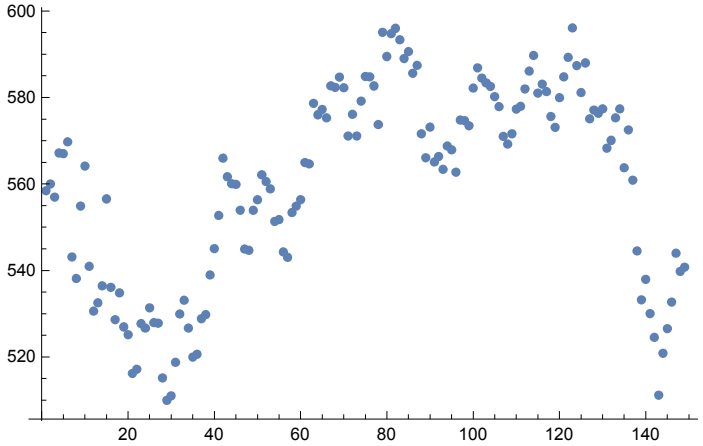
X - values are specified by passing coordinate pairs to ListPlot

```
pairs = {{0, 1}, {2, -1}, {4, 5}, {6, -5}};
ListPlot[pairs]
```



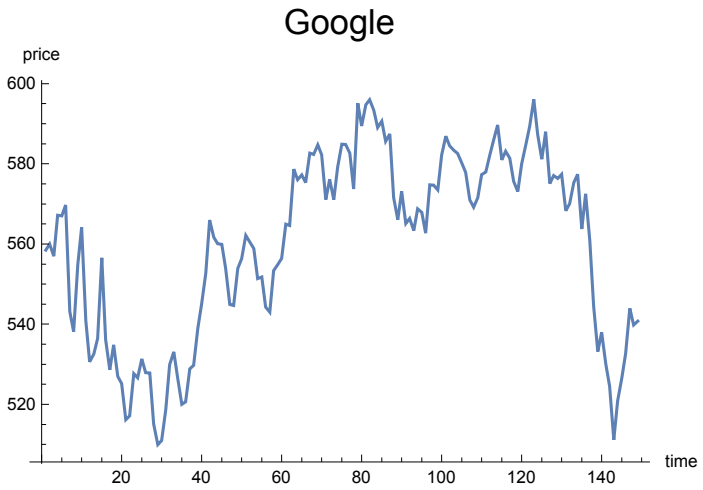
ListPlot is especially handy for visualizing real-world data.

```
googleStockDateAndPrice = FinancialData["NASDAQ:GOOG", All];
googleStockPrice = googleStockDateAndPrice[[ ;; , 2]];
ListPlot[googleStockPrice]
```



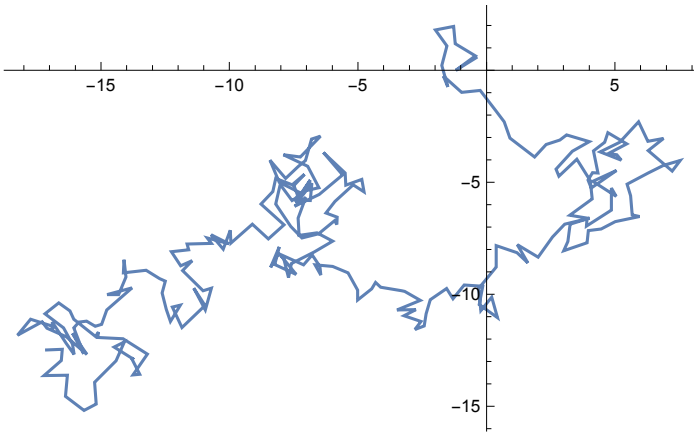
If you want the plot to show a trend, connect the dots using ListLinePlot

```
ListLinePlot[googleStockPrice,
 PlotLabel → Style["Google", 18], AxesLabel → {"time", "price"}]
```



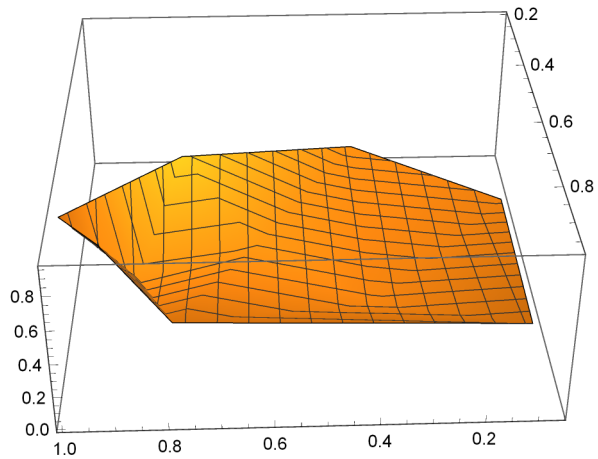
ListLinePlot can be used to create a random walk in 2 dimensions

```
ListLinePlot[Accumulate[RandomReal[{-1, 1}, {250, 2}]]]
```



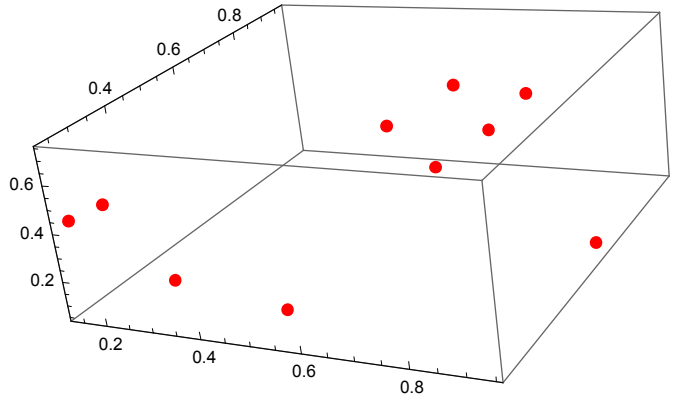
As you might assume, ListPlot has a three dimensional analog called ListPlot3D

```
ListPlot3D[RandomReal[{0, 1}, {10, 3}]]
```



Note that ListPlot3D automatically includes a mesh. If you just want the points to be plotted, use the function ListPointPlot3D

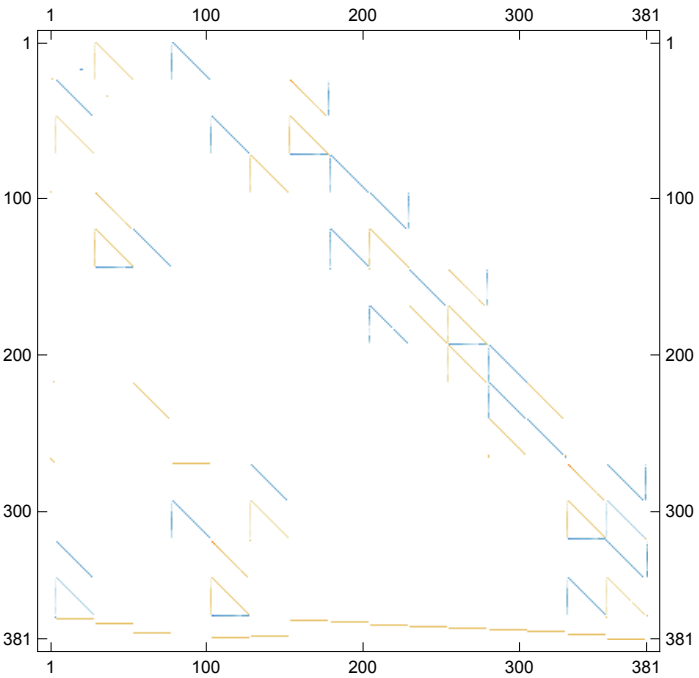
```
ListPointPlot3D[RandomReal[{0, 1}, {10, 3}], PlotStyle → {{Red, PointSize[0.02]}}]
```



Often times your data will be given to you in terms of a matrix. As you might expect, there are a few different ways to visualize matrices (do you see a pattern developing here...*Mathematica* has TONS of built in functions).

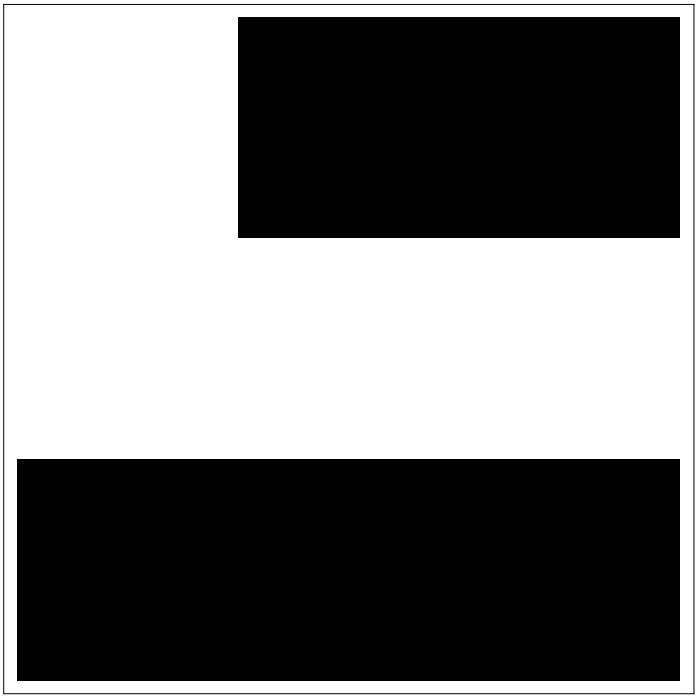
```
mymat = Normal[ExampleData[{"Matrix", "HB/west0381"}, "Matrix"]];
Dimensions[mymat]
MatrixPlot[mymat]
```

```
{381, 381}
```

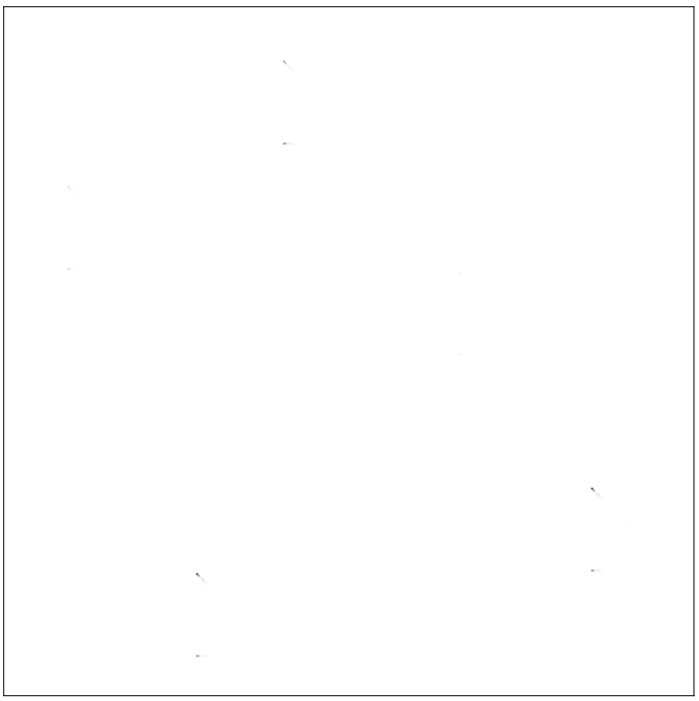


If you just want to see which elements are zero and which are nonzero, use ArrayPlot (this is similar to Matlab's spy()).
Since the matrix "mymat" is so sparse above, trying to use ArrayPlot on it will result in a plot that looks mostly zero due to the resolution of your screen.

```
ArrayPlot[{{0, 1, 1}, {0, 0, 0}, {1, 1, 1}}]
```

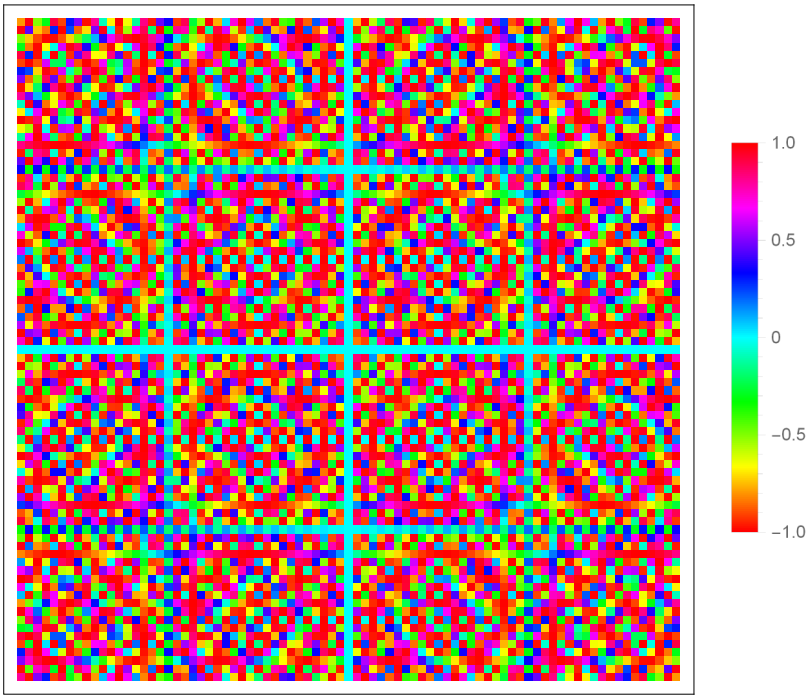


```
ArrayPlot[mymat]
```



ArrayPlot is also nice to visualize functions in a sort of “heat map” style.

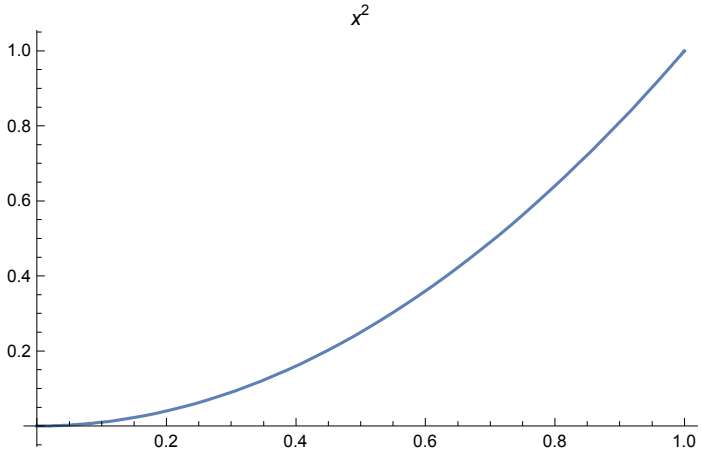
```
ArrayPlot[Table[Sin[x y], {x, -40, 40}, {y, -40, 40}],
 ColorFunction → Hue, PlotLegends → Automatic]
```

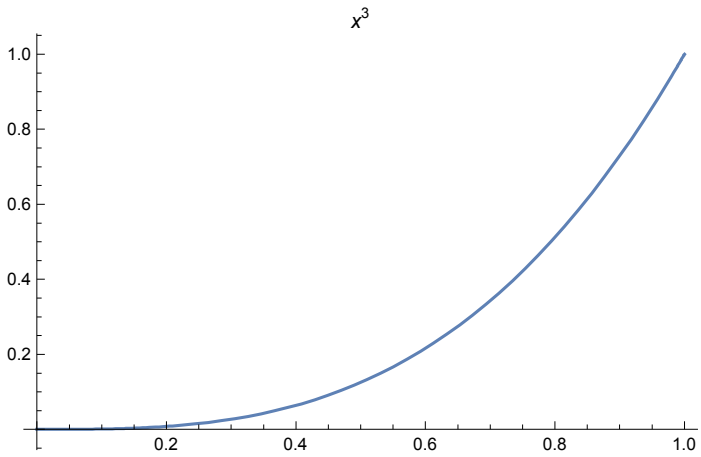


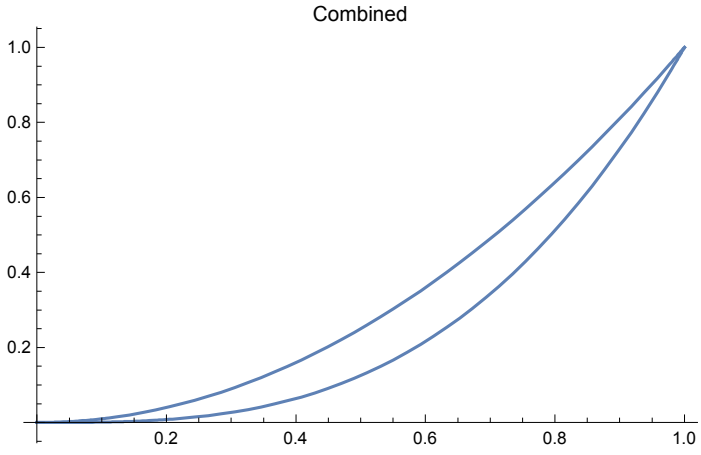
Combining Plots

When you have a number of plots that you want to combine into one figure, you have a few different options.

```
p1 = Plot[x^2, {x, 0, 1}, PlotLabel → "x^2"]
p2 = Plot[x^3, {x, 0, 1}, PlotLabel → "x^3"]
Show[p1, p2, PlotLabel → "Combined"]
```

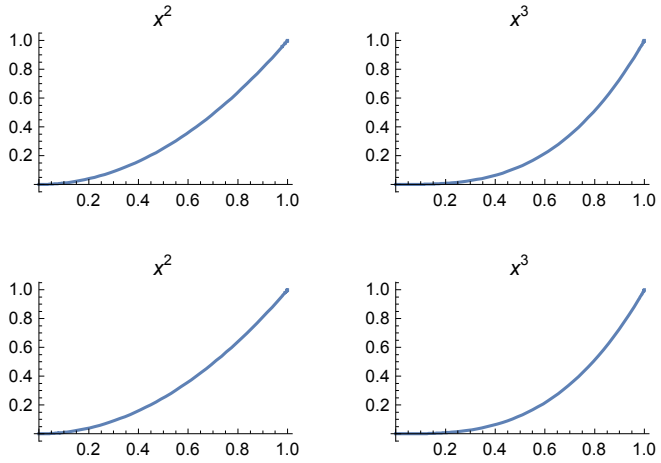






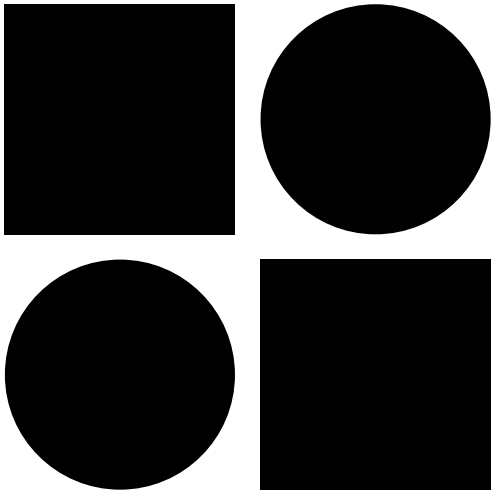
The function Show combines two figures.
If, however, you want to put the figures side by side (instead of on top of each other), then GraphicsGrid (or GraphicsRow) is the function you are looking for.

```
GraphicsGrid[{{p1, p2}}]
GraphicsRow[{p1, p2}]
```



The advantage of GraphicsGrid is that you can specify the exact dimensions (how many rows and columns) your figures will occupy.

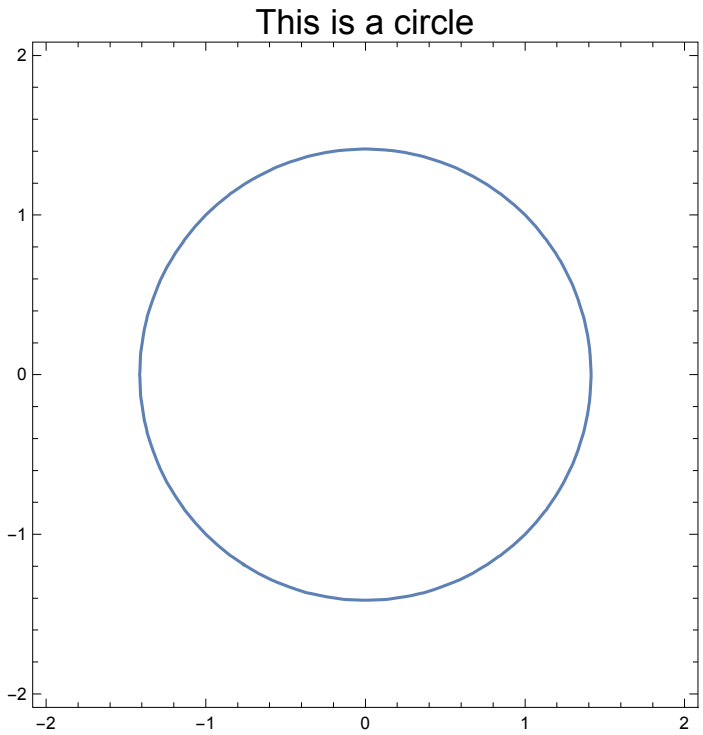
```
r = Graphics[Rectangle[]];
d = Graphics[Disk[]];
GraphicsGrid[{{r, d}, {d, r}}]
```



General Plots

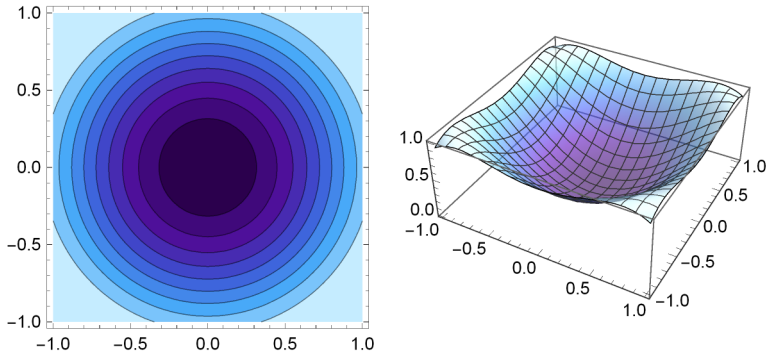
Since not every equation can be written in the form y=f(x) or $y = f(x_1, x_2, \ldots)$, *Mathematica* has a number of parametric plotting options.

```
ContourPlot[x^2 + y^2 == 2, {x, -2, 2},
 {y, -2, 2}, PlotLabel → Style["This is a circle", 18]]
```



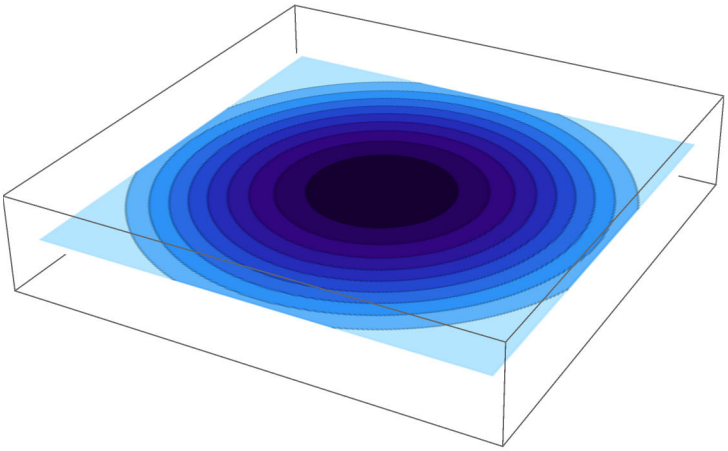
ContourPlot works by shading different regions of a graph with different colors depending on the function value. Since the above was given as an equality, only two values can be output (true or false). However, you don't need to just supply ContourPlot with equations, you can give functions as well

```
GraphicsRow[{ContourPlot[Sin[x^2 + y^2], {x, -1, 1}, {y, -1, 1},
    ColorFunction → "DeepSeaColors"], Plot3D[Sin[x^2 + y^2], {x, -1, 1},
    {y, -1, 1}, ColorFunction → "DeepSeaColors", PlotStyle → Opacity[.6]]}]
```

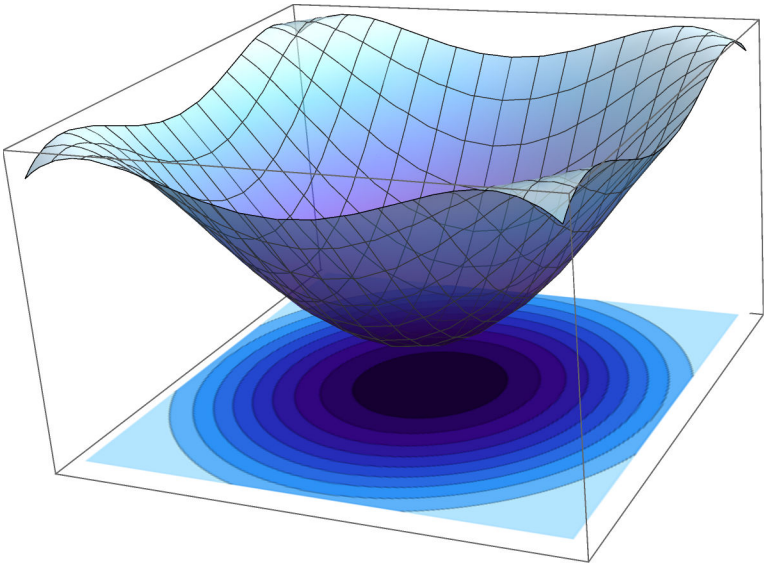



```
contourPlot = ContourPlot[Sin[x^2 + y^2], {x, -1, 1}, {y, -1, 1},
    ColorFunction → "DeepSeaColors", Frame → False, Axes → False];
plot3D = Plot3D[Sin[x^2 + y^2], {x, -1, 1}, {y, -1, 1},
    ColorFunction → "DeepSeaColors", PlotStyle → Opacity[.6]];
```

```
height = -.2;
ground = Graphics3D[
  {Texture[contourPlot], EdgeForm[], Polygon[{{-1, -1, height}, {1, -1, height},
     {1, 1, height}, {-1, 1, height}}, VertexTextureCoordinates →
     {{0, 0}, {1, 0}, {1, 1}, {0, 1}}]}, Lighting → "Neutral"]
```

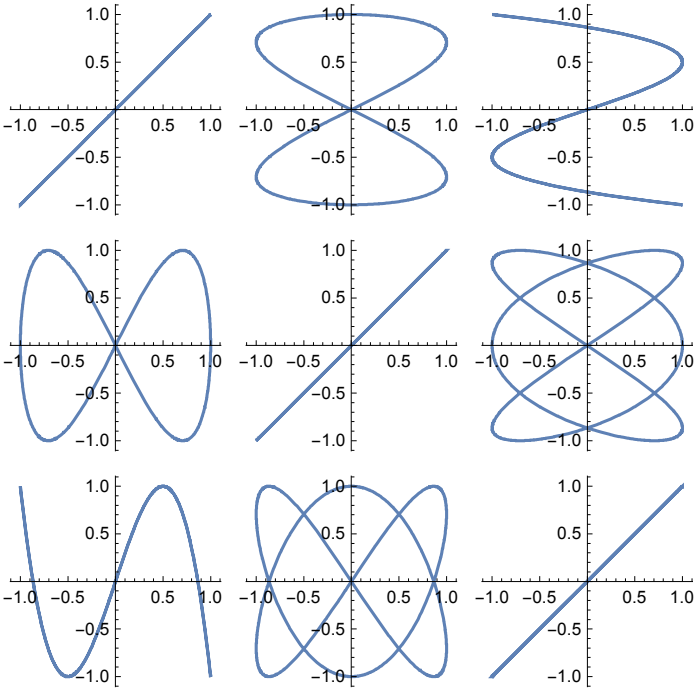


```
Show[ground, plot3D, PlotRange → All]
```

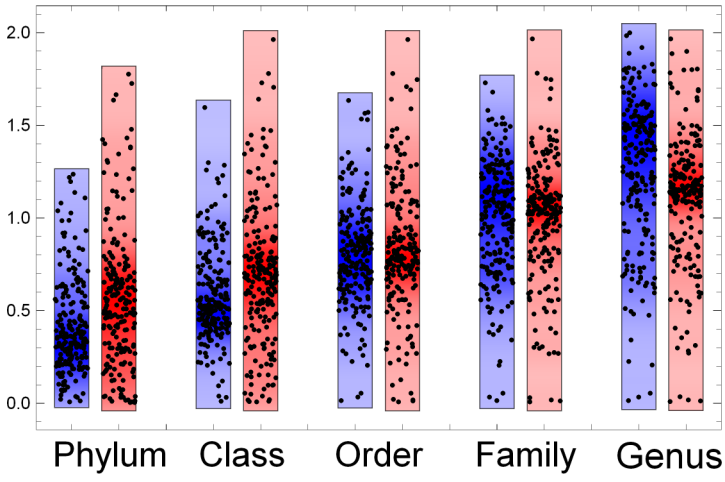


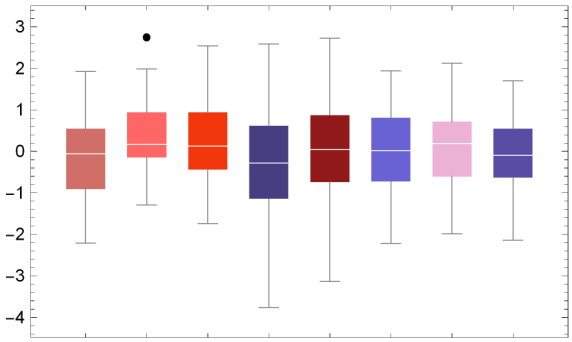
We can also plot parametric curves

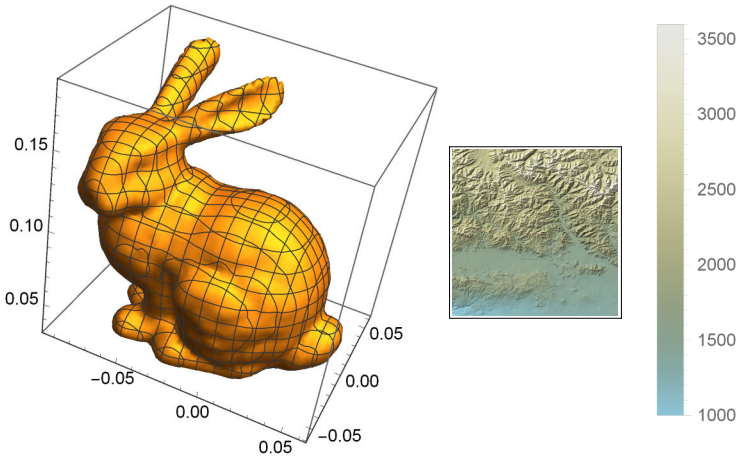
```
GraphicsGrid@Table[ParametricPlot[{Sin[m u], Sin[n u]}, {u, 0, 2 Pi}], {n, 3}, {m, 3}]
```

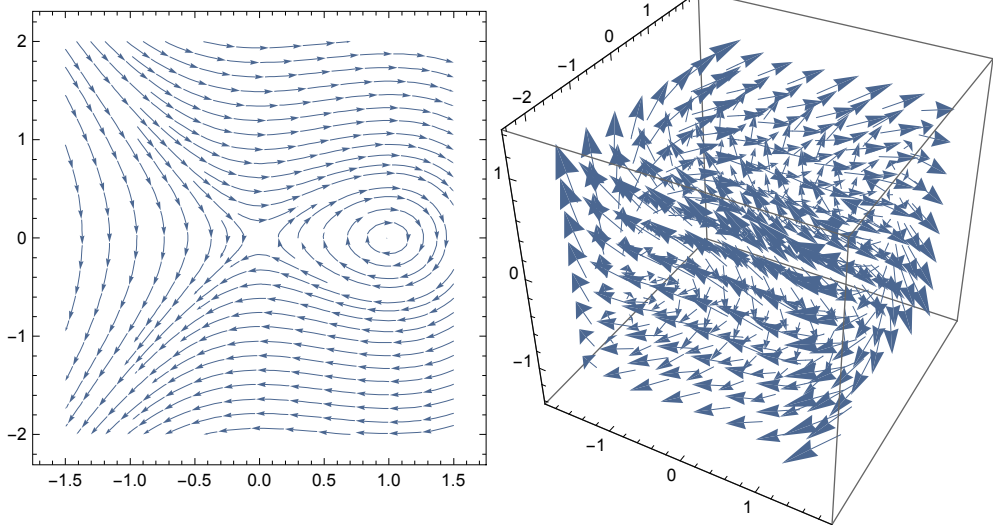


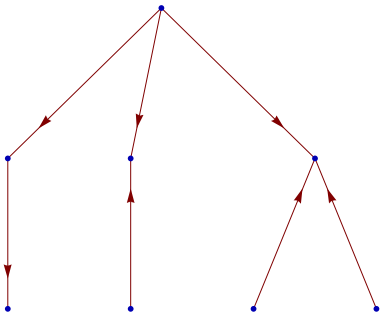
There are dozens (if not hundreds) of plots in *Mathematica;* take some time poking around to discover a few of the more interesting kinds.

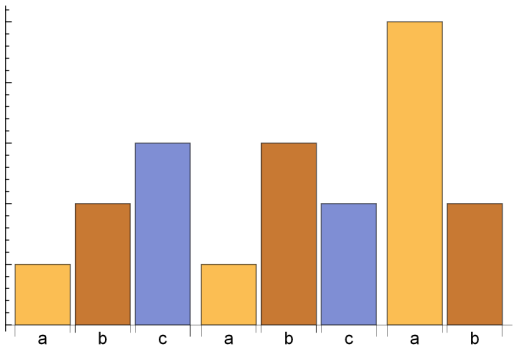


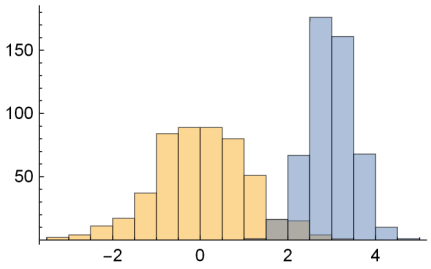

3
2
1
0
−1
−2
−3
−4

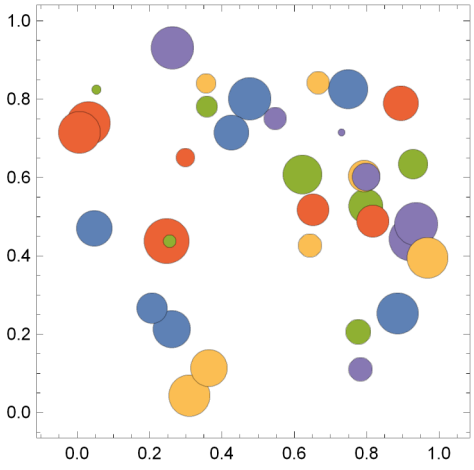

0.15
0.10
0.05
−0.05
0.00
0.05
0.05
0.00
−0.05
3500
3000
2500
2000
1500
1000

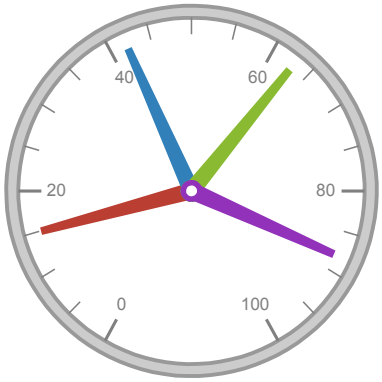

2
1
0
−1
−2
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2
1
0
−1
−2
1
0
−1
−1
0
1

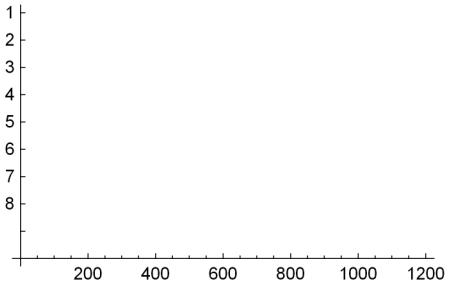



a
b
c
a
b
c
a
b


150
100
50
−2
0
2
4


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


40
60
20
80
0
100



Homework

#1.

In how many points do the following ellipses intersect? Hint: graph them.

$$\frac{1}{4}(x-1)^2 + y^2 = 1$$

$$(x-2)^2 + \frac{1}{4}(y-2)^2 = 1$$

#2.

Graph the following functions on the same axis.

$$e^x$$

$$1$$

$$x + 1$$

$$\frac{x^2}{2} + x + 1$$

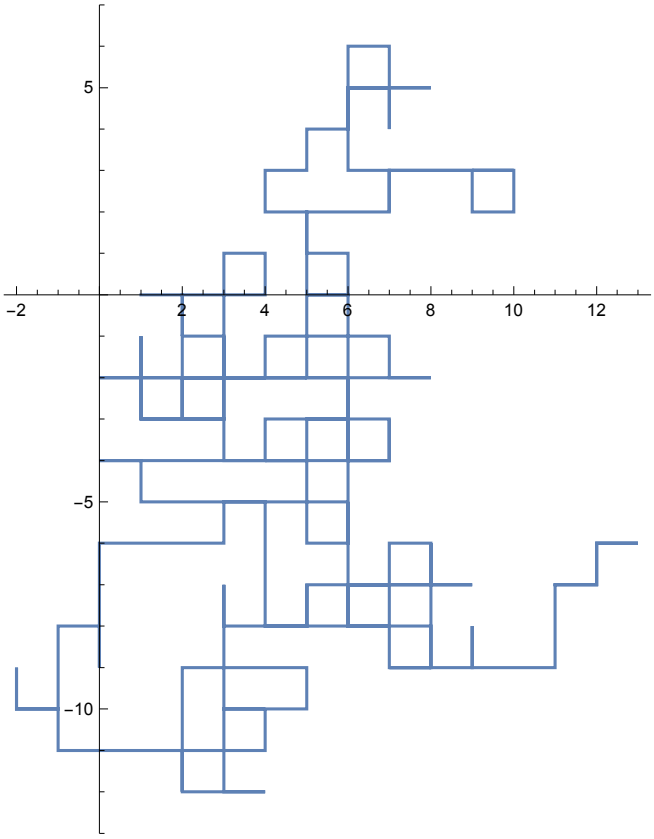
$$\frac{x^3}{6} + \frac{x^2}{2} + x + 1$$

$$\frac{x^4}{24} + \frac{x^3}{6} + \frac{x^2}{2} + x + 1$$

Include plot labels. Comment on the pattern that you see developing, and comment on why this might be the case.

#3.

Similar to the random walk in 2 dimensions given in the notes above, create a random walk constrained to the lattice. That is, the only allowable moves are to move one unit west, east, north, or south. The function RandomChoice will come in handy here. An example output is



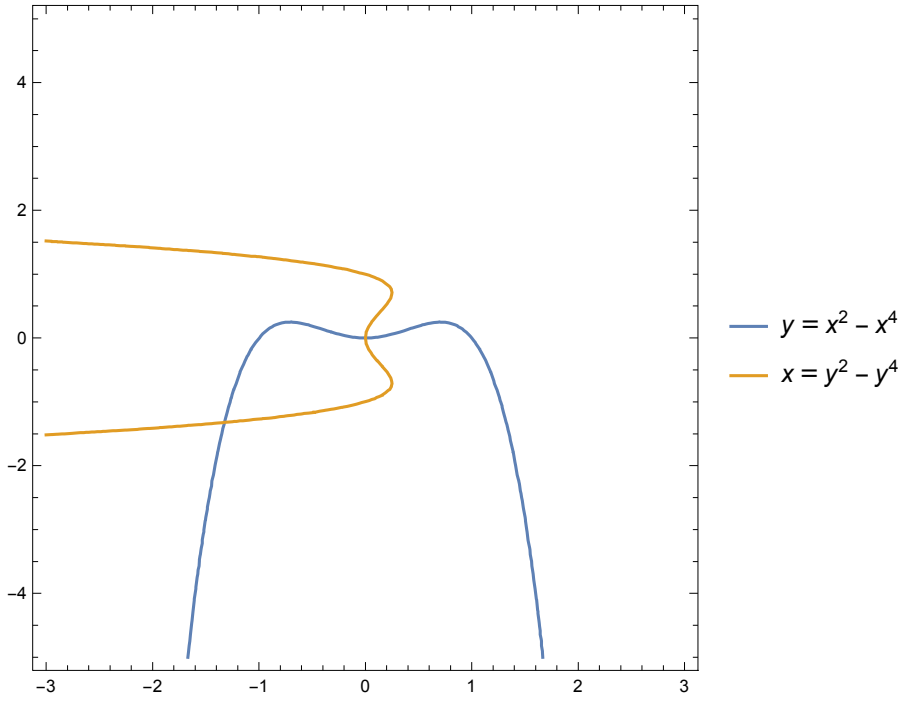
#4.

Write a function called plotSwitchXY that returns a plot of a given equation, along with the plot of the

same equation with the roles of x and y reversed. For example

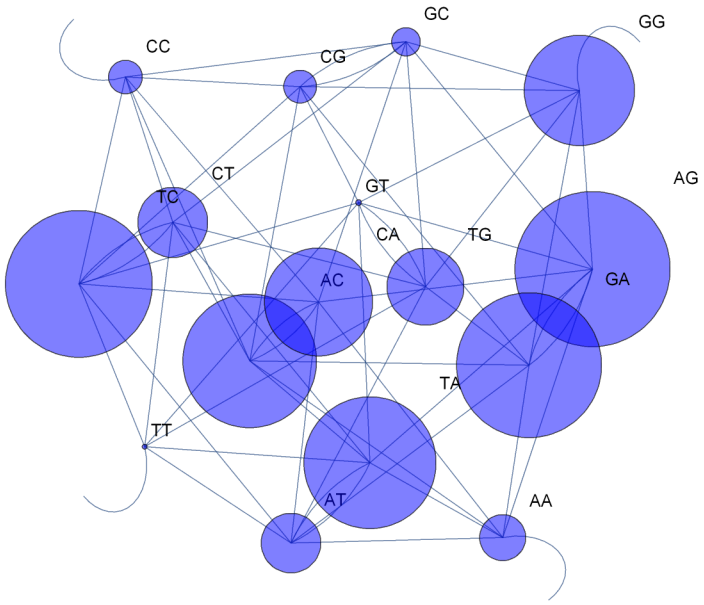
```
plotSwitchXY[x^2 - x^4, {x, -3, 3}]
```

should return a plot that looks like



#5.

A de Bruijn graph is a directed graph representing overlaps between sequences of symbols. There is a built in function called DeBruijnGraph that creates such a graph. By using the options GraphLayout, EdgeShapeFunction, VertexLabels, VertexSize, and VertexStyle, see if you can replicate a figure such as the following. The plot underlying this figure was DeBruijnGraph[4,2]. The vertex sizes were chosen at random.



Data Import/Export

Import/Export

Import

Import is the primary way to get data from a file into your *Mathematica* session.

```
Import[
 "C:\\Users\\David\\Dropbox\\Teaching\\Math399IntrotoMathSoftwareFall2014\\test.
   h5", "Elements"]
```

```
{Annotations, Attributes, Data,
 DataEncoding, DataFormat, Datasets, Dimensions, Groups}
```

By including the “Elements” option, this will return to you a list of pieces that you can extract from the given file. For example, if we want to see what sorts of datasets exist in that HDF5 file, we can use

```
Import[
 "C:\\Users\\David\\Dropbox\\Teaching\\Math399IntrotoMathSoftwareFall2014\\test.
   h5", "Datasets"]
```

```
{/data}
```

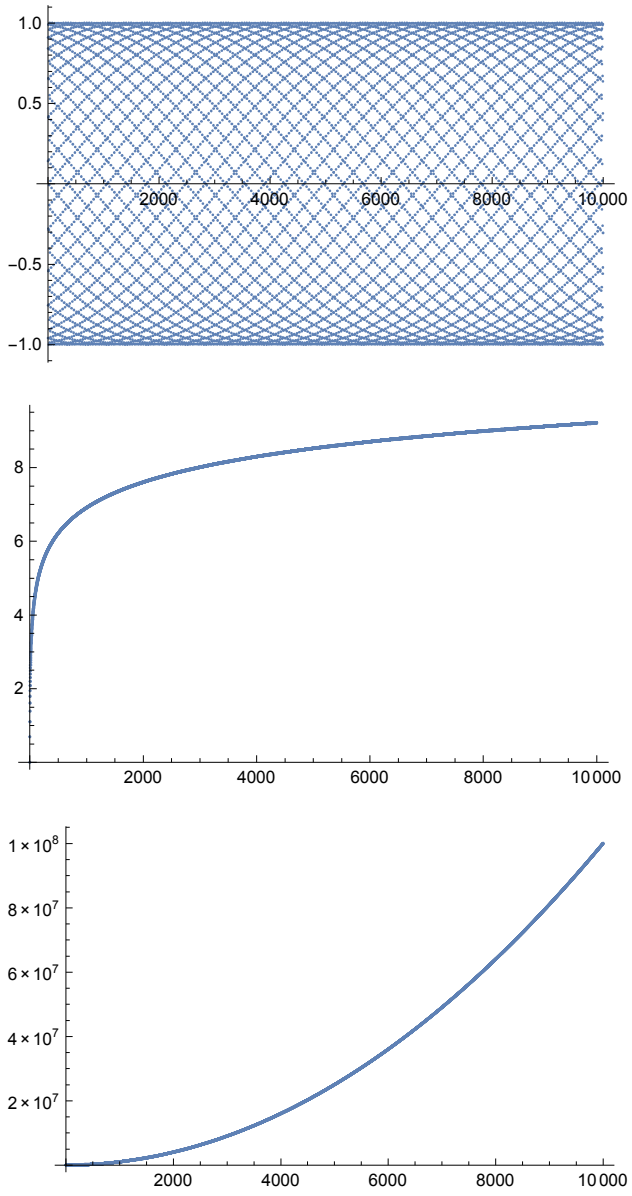
So there is a single dataset in test.h5 called /data. Let’s import this data

```
mymat = Import[
    "C:\\Users\\David\\Dropbox\\Teaching\\Math399IntrotoMathSoftwareFall2014\\test
      .h5", {"Datasets", "/data"}];

Dimensions[mymat]
```

```
{3, 10 000}
```

```
GraphicsColumn[Table[ListPlot[mymat[[i, ;;]]], {i, 1, 3}], ImageSize → 350]
```



Typically *Mathematica* recognizes the file type based on its extension. Here is a list of extensions that *Mathematica* recognizes:

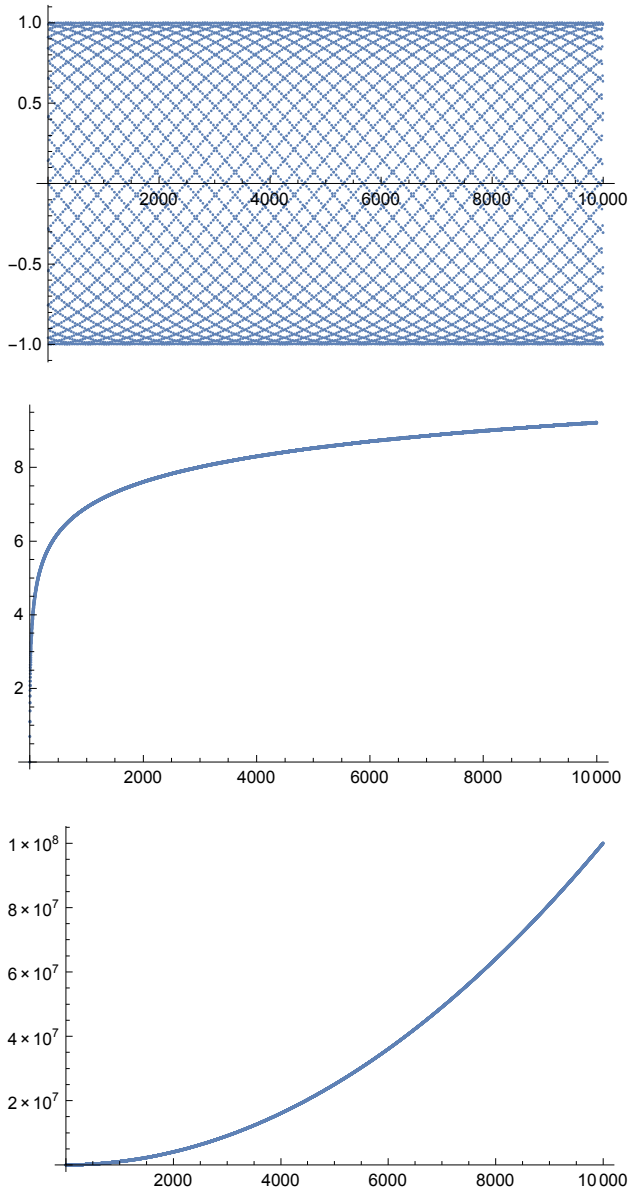
$ImportFormats

```
{3DS, ACO, Affymetrix, AgilentMicroarray, AIFF, ApacheLog, ArcGRID, AU, AVI, Base64,
 BDF, Binary, Bit, BMP, Byte, BYU, BZIP2, CDED, CDF, Character16, Character8,
 CIF, Complex128, Complex256, Complex64, CSV, CUR, DBF, DICOM, DIF, DIMACS,
 Directory, DOT, DXF, EDF, EPS, ExpressionML, FASTA, FASTQ, FCS, FITS, FLAC,
 GenBank, GeoTIFF, GIF, GPX, Graph6, Graphlet, GraphML, GRIB, GTOPO30, GXL, GZIP,
 HarwellBoeing, HDF, HDF5, HIN, HTML, ICC, ICNS, ICO, ICS, Integer128, Integer16,
 Integer24, Integer32, Integer64, Integer8, JCAMP-DX, JPEG, JPEG2000, JSON, JVX,
 KML, LaTeX, LEDA, List, LWO, MAT, MathML, MBOX, MDB, MESH, MGF, MIDI, MMCIF, MOL,
 MOL2, MP3, MPS, MTP, MTX, MX, NASACDF, NB, NDK, NetCDF, NEXUS, NOFF, OBJ, ODS,
 OFF, OGG, OpenEXR, Package, Pajek, PBM, PCX, PDB, PDF, PGM, PLY, PNG, PNM, PPM,
 PXR, QuickTime, RawBitmap, Real128, Real32, Real64, RIB, RSS, RTF, SCT, SDF,
 SDTS, SDTSDEM, SFF, SHP, SMILES, SND, SP3, Sparse6, STL, String, SurferGrid,
 SXC, Table, TAR, TerminatedString, Text, TGA, TGF, TIFF, TIGER, TLE, TSV,
 UnsignedInteger128, UnsignedInteger16, UnsignedInteger24, UnsignedInteger32,
 UnsignedInteger64, UnsignedInteger8, USGSDEM, UUE, VCF, VCS, VTK, WAV,
 Wave64, WDX, WebP, XBM, XHTML, XHTMLMathML, XLS, XLSX, XML, XPORT, XYZ, ZIP}
```

If, however, your file is in one of the above formats, but does not have the correct extension, you can still specify this in Import. For example, let's use the same HDF5 file, but change the extension to "xyz" and then use import to read it in.

```
mymat2 = Import[
   "C:\\Users\\David\\Dropbox\\Teaching\\Math399IntrotoMathSoftwareFall2014\\test
     .xyz", {"HDF5", "Datasets", "/data"}];
```

```
GraphicsColumn[Table[ListPlot[mymat2[[i, ;;]]], {i, 1, 3}], ImageSize → 350]
```



By specifying “HDF5” we are indicating to *Mathematica* that it is to read this file as if it is an HDF5 file. Note that most times *Mathematica* is intelligent enough to recognize the file format, even if the file extension is non-standard and we have not specified the format.

Export

Export is extremely analogous to Import. Let’s see how to use it by creating a random matrix and saving it as a Matlab *.mat file.

```
toExport = RandomReal[{0, 1}, {10, 100}];
```

```
Export[
 "C:\\Users\\David\\Dropbox\\Teaching\\Math399IntrotoMathSoftwareFall2014\\
   test_export.mat", toExport]
```

```
C:\Users\David\Dropbox\Teaching\Math399IntrotoMathSoftwareFall2014\test_export.
  mat
```

You can now go over to Matlab and load this file as you would any other .mat file.

Here is a list of all the available export formats

```
$ExportFormats
```

```
{3DS, ACO, AIFF, AU, AVI, Base64, Binary, Bit, BMP, Byte, BYU, BZIP2, C, CDF,
 Character16, Character8, Complex128, Complex256, Complex64, CSV, CUR,
 DICOM, DIF, DIMACS, DOT, DXF, EMF, EPS, ExpressionML, FASTA, FASTQ, FCS,
 FITS, FLAC, FLV, GIF, Graph6, Graphlet, GraphML, GXL, GZIP, HarwellBoeing,
 HDF, HDF5, HTML, HTMLFragment, ICNS, ICO, Integer128, Integer16, Integer24,
 Integer32, Integer64, Integer8, JPEG, JPEG2000, JSON, JVX, KML, LEDA,
 List, LWO, MAT, MathML, Maya, MGF, MIDI, MOL, MOL2, MP3, MTX, MX, NASACDF,
 NB, NetCDF, NEXUS, NOFF, OBJ, OFF, OGG, Package, Pajek, PBM, PCX, PDB, PDF,
 PGM, PLY, PNG, PNM, POV, PPM, PXR, QuickTime, RawBitmap, Real128, Real32,
 Real64, RIB, RTF, SCT, SDF, SND, Sparse6, STL, String, SurferGrid, SVG, SWF,
 Table, TAR, TerminatedString, TeX, TeXFragment, Text, TGA, TGF, TIFF, TSV,
 UnsignedInteger128, UnsignedInteger16, UnsignedInteger24, UnsignedInteger32,
 UnsignedInteger64, UnsignedInteger8, UUE, VideoFrames, VRML, VTK, WAV, Wave64,
 WDX, WebP, WMF, X3D, XBM, XHTML, XHTMLMathML, XLS, XLSX, XML, XYZ, ZIP, ZPR}
```

Note that the import formats and export formats aren't all the same

```
Length[$ImportFormats]
Length[$ExportFormats]
Length[Intersection[$ImportFormats, $ExportFormats]]
```

```
172
```

```
144
```

```
129
```

Other

Readlist, write

While Import and Export will fulfill most of your needs, sometimes you need finer-level control over your files. This can be achieved with functions like ReadList and Write.

ReadList is useful if your file contains non-standard formatting, and you just want to read in the entire file as a text string.

```
windowsLogFileContents = ReadList["C:\\Users\\David\\debug.log", "String"];
```

```
windowsLogFileContents // Short
```

```
{[0708/095405:ERROR:client_util.cc(293)] Could not find
   exported function RelaunchChromeBrowserWithNewCommandLineIfNeeded,
 <<5>>, [0806/133805:ERROR:client_util.cc(327)] Coul …
  launchChromeBrowserWithNewCommandLineIfNeeded}
```

The Write function is a Linux-like stream writing function

```
str = OpenWrite["C:\\Users\\David\\test.txt"]
```

OutputStream[Name: test.txt
Unique ID: 2155]

```
Write[str, "Let's write some text to this file"]
```

```
Close[str]
```

```
C:\Users\David\test.txt
```

Built in data

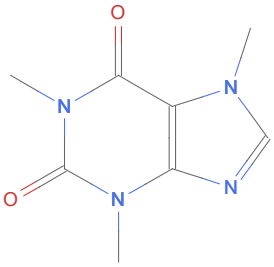
SocialMediaData, GenomeData, OceanData, ChemicalData, FinancialData, CountryData, CityData, ZipCodeData, FlightData

Mathematica builds in access to a plethora of data.This data is curated by WolframResearch and resides on Wolfram central servers. Whenever you call one of the *Data functions, it accesses the data from these servers. Let's look at an example.

```
GenomeData["ChromosomeXGenes"] // Short
CountryData["France", "Population"]
ChemicalData["Caffeine"]
FinancialData["GE", "MarketCap"]
CityData[{Large, "Ohio", "UnitedStates"}]
```

```
{ABCB7, ABCD1, ACE2, ACOT9, ACRC, ACSL4, ACTBP1, ACTGP10,
 ACTRT1, ADFN, AFF2, AGMX2, AGTR2, <<1902>>, ZNF41, ZNF449, ZNF630,
 ZNF645, ZNF673, ZNF674, ZNF711, ZNF75D, ZNF81, ZRSR2, ZXDA, ZXDB}
```

6.3783×10^7 people



2.576×10^{11}

{ Columbus , Cleveland , Cincinnati , Toledo , Akron , Dayton }

In general, the format for these data functions is to call the *Data function with a string argument. The allowable arguments can be found by using the following command

```
GenomeData["Properties"] // Short
CountryData["Properties"] // Short
```

```
{AlternateNames, AlternateStandardNames, BiologicalProcesses,
 CellularComponents, Chromosome, <<31>>, TranscriptGenBankIndices,
 TranscriptNCBIAccessions, UniProtAccessions, UnsequencedPositions, UTRSequences}
```

```
{AdultPopulation, AgriculturalProducts, AgriculturalValueAdded,
 Airports, AlternateNames, AlternateStandardNames, <<212>>,
 UnpavedAirports, UnpavedRoadLength, ValueAdded, WaterArea, WaterwayLength}
```

You can also view all of the available data tags by calling the function with no argument

```
GenomeData[] // Short
CountryData[] // Short
```

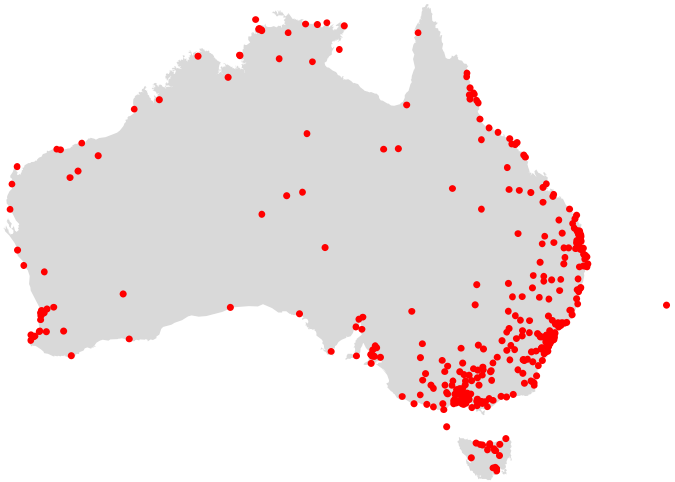
```
{381, 3812, 3813, 3814, 3815, 5HT3c2, 7A5, A1BG, A1CF, A26A1,
 A26B1, A26B2, A26B3, A26C1A, <<39 892>>, ZUFSP, ZW10, ZWILCH, ZWINT,
 ZWINTAS, ZWS1, ZXDA, ZXDB, ZXDC, ZYG11A, ZYG11B, ZYX, ZZEF1, ZZZ3}
```

{Afghanistan, Albania, Algeria, American Samoa, Andorra, Angola, Anguilla, Antigua and Barbuda, Argentina, Armenia, Aruba, Australia, Austria, Azerbaijan, Bahamas, Bahrain, Bangladesh, Barbados, Belarus, Belgium, Belize, Benin, Bermuda, Bhutan, Bolivia, Bosnia and Herzegovina, Botswana, Brazil, British Virgin Islands, Brunei, Bulgaria, Burkina Faso, Burundi, Cambodia, Cameroon, Canada, Cape Verde, Cayman Islands, Central African Republic, Chad, Chile, China, Christmas Island, Cocos Keeling Islands, Colombia, Comoros, Cook Islands, Costa Rica, Croatia, Cuba, Curacao, Cyprus, Czech Republic, Democratic Republic of the Congo, Denmark, Djibouti, Dominica, Dominican Republic, East Timor, Ecuador, Egypt, El Salvador, Equatorial Guinea, Eritrea, Estonia, Ethiopia, Falkland Islands, Faroe Islands, Fiji, Finland, France, French Guiana, French Polynesia, Gabon, Gambia, Gaza Strip, Georgia, Germany, Ghana, Gibraltar, Greece, Greenland, Grenada, Guadeloupe, Guam, Guatemala, Guernsey, Guinea, Guinea-Bissau, Guyana, Haiti, Honduras, Hong Kong, Hungary, Iceland, India, Indonesia, Iran, Iraq, Ireland, Isle of Man, Israel, Italy, Ivory Coast, Jamaica, Japan, Jersey, Jordan, <<25>>, Mauritania, Mauritius, Mayotte, Mexico, Micronesia, Moldova, Monaco, Mongolia, Montenegro, Montserrat, Morocco, Mozambique,

Myanmar , Namibia , Nauru , Nepal , Netherlands , New Caledonia , New Zealand ,
Nicaragua , Niger , Nigeria , Niue , Norfolk Island , Northern Mariana Islands ,
North Korea , Norway , Oman , Pakistan , Palau , Panama , Papua New Guinea ,
Paraguay , Peru , Philippines , Pitcairn Islands , Poland , Portugal , Puerto Rico ,
Qatar , Republic of the Congo , Réunion , Romania , Russia , Rwanda ,
Saint Helena, Ascension and Tristan da Cunha , Saint Kitts and Nevis , Saint Lucia ,
Saint Pierre and Miquelon , Saint Vincent and the Grenadines , Samoa , San Marino ,
São Tomé and Príncipe , Saudi Arabia , Senegal , Serbia , Seychelles , Sierra Leone ,
Singapore , Sint Maarten , Slovakia , Slovenia , Solomon Islands , Somalia ,
South Africa , South Korea , South Sudan , Spain , Sri Lanka , Sudan , Suriname ,
Svalbard , Swaziland , Sweden , Switzerland , Syria , Taiwan , Tajikistan ,
Tanzania , Thailand , Togo , Tokelau , Tonga , Trinidad and Tobago , Tunisia ,
Turkey , Turkmenistan , Turks and Caicos Islands , Tuvalu , Uganda , Ukraine ,
United Arab Emirates , United Kingdom , United States , United States Virgin Islands ,
Uruguay , Uzbekistan , Vanuatu , Vatican City , Venezuela , Vietnam ,
Wallis and Futuna Islands , West Bank , Western Sahara , Yemen , Zambia , Zimbabwe }

You can view what cities WolframResearch has information on

```
Graphics[{LightGray, CountryData["Australia", "Polygon"], Red,
  PointSize[.01], Select[Point[Reverse[CityData[#, "Coordinates"]]] & /@
    CityData[{All, "Australia"}], FreeQ[#, _Missing] &]}]
```



Here's the population of Corvallis along with it's latitude and longitude

```
CityData[{"Corvallis", "Oregon", "UnitedStates"}, "Population"]
{CityData[{"Corvallis", "Oregon", "UnitedStates"}, "Latitude"],
 CityData[{"Corvallis", "Oregon", "UnitedStates"}, "Longitude"]}
```

```
54 998 people
```

```
{44.5699°, -123.278°}
```

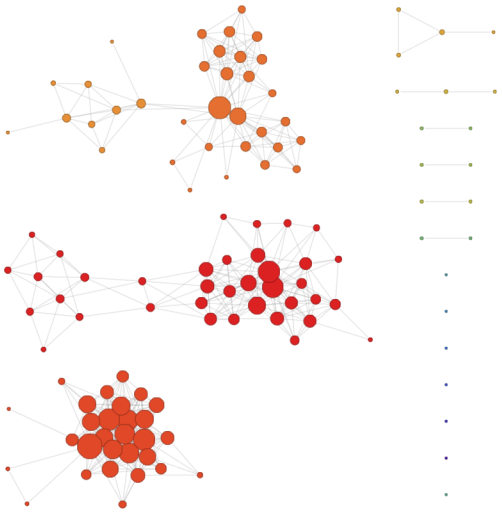
One important thing to note is that much of the data does not come with references, so the usefulness of this built-in data for publication use is somewhat limited.

API's and Social Media

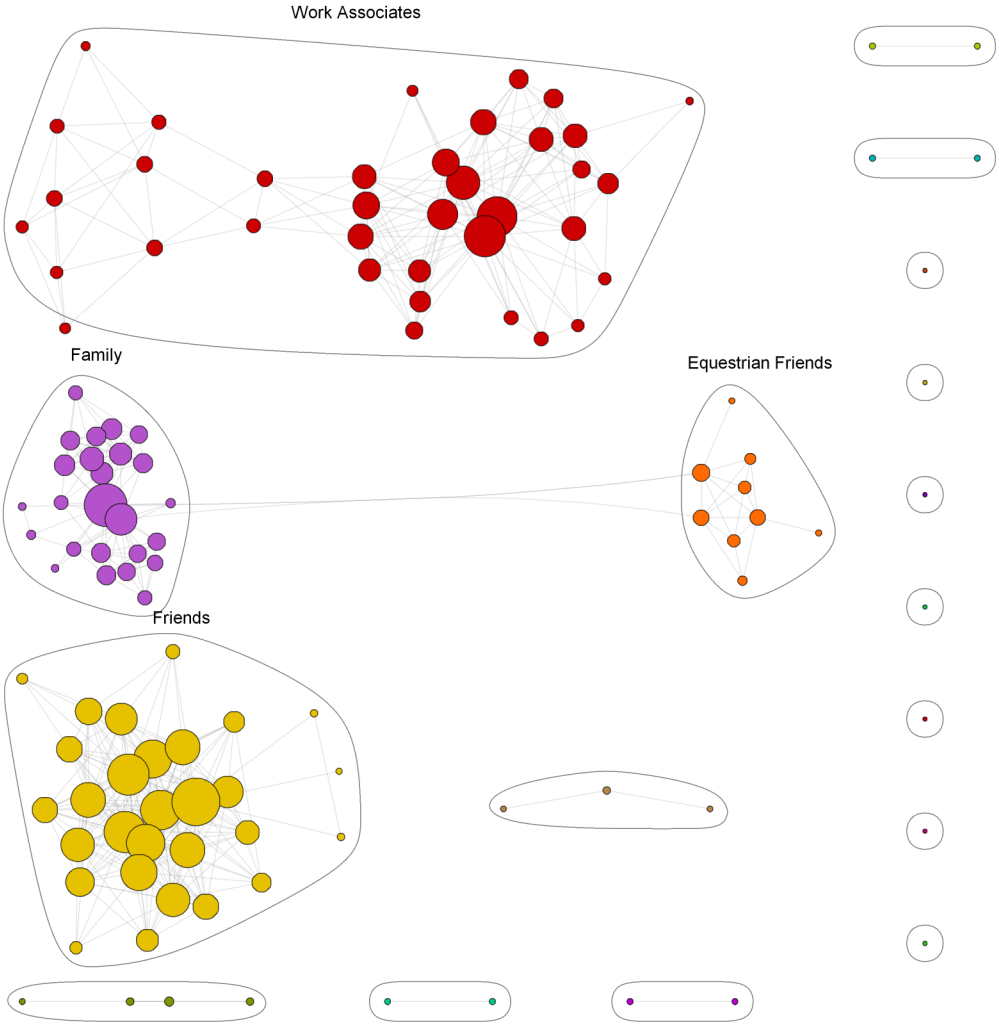
Beyond just delivering data stored on Wolfram servers, *Mathematica* also includes API's to access data from other sources. One particularly interesting example of this is the SocialMediaData function, which allows you to analyze information from a number of popular social media sites (such as Facebook, Google+, Twitter, Instagram, etc.).
Let's look at the Facebook integration. When calling the function for the first time, *Mathematica* will as you to go to Facebook, grant *Mathematica* permission to access your data, and then will give you a code that you can enter into *Mathematica* to finish the process.

Let' s start with visualizing my Facebook friend network. (FYI: I don't use Facebook a whole lot).

```
SocialMediaData["Facebook", "FriendNetwork"]
```



```
CommunityGraphPlot[SocialMediaData["Facebook", "FriendNetwork"],
 CommunityLabels → {"Work Associates", "Friends", "Family", "Equestrian Friends"}]
```



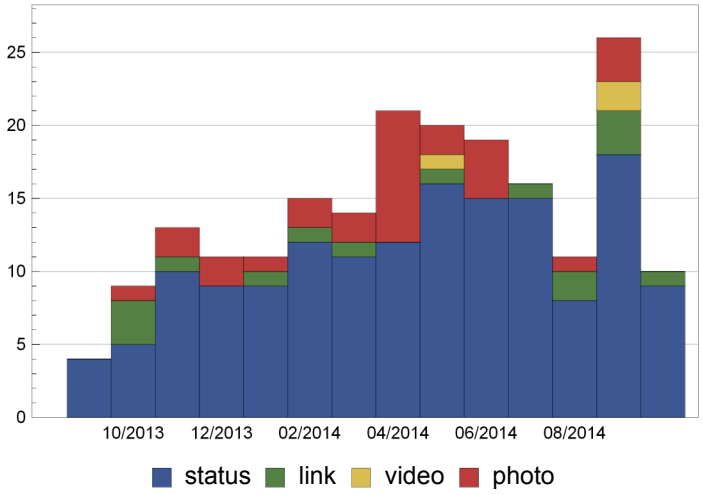
As you can see, there are a number of “central nodes” that seem to be “hubs” for each of the networks. Let’s select those using a function called DegreeCentrality

```
g = SocialMediaData["Facebook", "FriendNetwork"];
friends = Map[PropertyValue[{g, #}, "Name"] &, VertexList[g]];
ranking = Reverse[Ordering[DegreeCentrality[g]]];
Part[VertexList[g], ranking[[ ;; 5]]]
```

```
{Lisa Prinz, Hilary Rose Barker,
 Robert Pettinger, Samuel Kenneth Handelman, Kylar A Dunn}
```

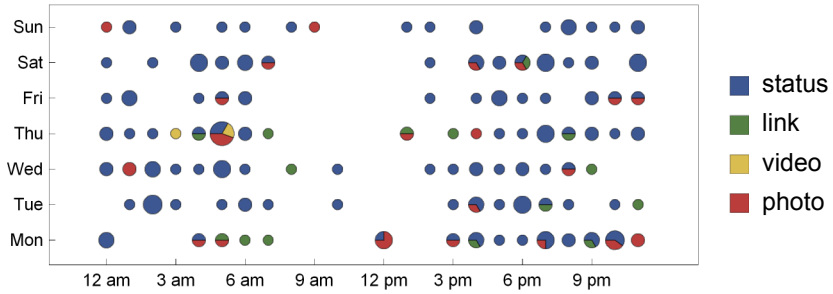
We can also take a look at my activity history

```
SocialMediaData["Facebook", "ActivityRecentHistory", "FormattedData"]
```



As you can see in the following distribution of my activity over the day, I pretty much don't use Facebook at all during working hours.

```
SocialMediaData["Facebook", "ActivityWeeklyDistribution", "FormattedData"]
```



You can also take a look at the post statistics

```
SocialMediaData["Facebook", "WallPostStatistics", "FormattedData"]
```

analyzed posts	195
total likes	537 (average: 2.754 per post)
total comments	187 (average: 0.959 per post)
average post length	10.769 words | 56.795 characters

And pull out things like the most frequently liked post

```
SocialMediaData["Facebook", "WallMostLikedPost", "FormattedData"]
```

```
Netherlands here we come!!
```

```
(≈ 2 months | 77 people like this)
```

To find out all the options and properties of SocialMediaData for Facebook, use the command

```
SocialMediaData["Facebook", "Properties"]
```

```
{ActivityRecentHistory, ActivityTypes, ActivityWeeklyDistribution,
 BimodalCommentNetwork, BimodalLikeCommentNetwork, BimodalLikeNetwork, Books,
 Category, CheckedCount, CommentNetwork, Family, Feeds, FriendIDs, FriendNetwork,
 Friends, LikeCommentNetwork, LikeCount, LikeNetwork, Link, Location, Movies,
 Music, MutualFriendIDs, MutualFriends, Name, PageID, Phone, Photos, Picture,
 Places, PostCommentNetwork, PostLikeCommentNetwork, PostLikeNetwork, Posts,
 RateLimit, TaggedPhotos, TaggedPosts, TaggedVideos, TalkingAboutCount, UserData,
 WallMostCommentedPost, WallMostLikedPost, WallPostLength, WallPostStatistics,
 WallTopCommenter, WallWeeklyAppActivity, WallWordFrequencies, Website}
```

CDF

Computable Document Format (CDF) files supply a rich deployment method leveraging the power and flexibility of the *Mathematica* language with the wide distribution provided by a public format.
As a built-in feature of *Mathematica* (version 8 and above), it's easy to save .cdf files straight from your working notebooks, custom-formatted papers and articles, or dedicated application development workflows. Anything you compute in *Mathematica* can be made into a user-interactive object offering maximum clarity in the presentation of your concepts, and there are no special considerations when creating documents just for viewing in Wolfram *CDF Player*; all notebook features can be displayed and printed.

You will need the CDF player plugin (available for free) to view CDF's and *Mathematica* notebooks in your browser. See the following website to download the CDF player
http://www.wolfram.com/cdf-player/

The easiest way to create a new CDF file is to select File->New->FreeCDF

These can be used to embedd *Mathematica* applets in a website. See the following website for more information about this:
http://www.wolfram.com/cdf/adopting-cdf/deploying-cdf/web-delivery.html

Homework

#1.

Similar to the Matlab problem, Create a 10^4 x 3 matrix where the rows correspond to 1, 2 ... 10^4 and the columns correspond to the functions sin(x), log(x), and x^2. Let the (i,j) entry be the value of the jth

function evaluated on i. Save this matrix as a *.mat file, a comma separated file, text file, and as an Excel file. Include these files in the dropbox folder when you complete this homework assignment.

a) Which format took the longest/shortest amount of time to write?
b) Which format used the least/most amount of disk space?
c) Which format took the longest/shortest amount of time to read in?

#2.

Use Import to read in the FASTA files located at http://www.math.oregonstate.edu/~koslickd/TestSequences.zip. Note that you can use Import directly on web links. You will have to first find the names of the files in this zip file (hint: Import[link.zip, "FileNames"]). Report the length of each one of the files.

#3.

Use GenomeData to find the 5 shortest and 5 longest genes in the human genome.

#4.

Optional: If you use Facebook, try replicating the commands I used above in the API's and Social Media section. You do not need to include this in the homework you turn in.

Exploratory Math and Problem Solving

Manipulate

Manipulate Basics

The single command Manipulate lets you create an astonishing range of interactive applications with just a few lines of input. Manipulate is designed to be used by anyone who is comfortable using basic commands such as Table and Plot: it does not require learning any complicated new concepts, nor any understanding of user interface programming ideas. The output you get from evaluating a Manipulate command is an interactive object containing one or more controls (sliders, etc.) that you can use to vary the value of one or more parameters. The output is very much like a small applet or widget: it is not just a static result, it is a running program you can interact with.

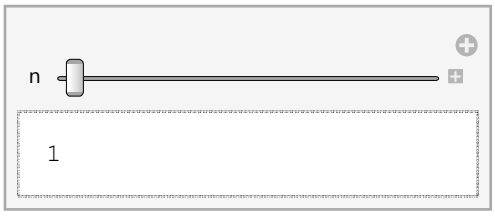
At its most basic, the syntax of Manipulate is identical to that of the humble function Table.Consider this Table command, which produces a list of numbers from one to twenty.

```
Table[n, {n, 1, 20}]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

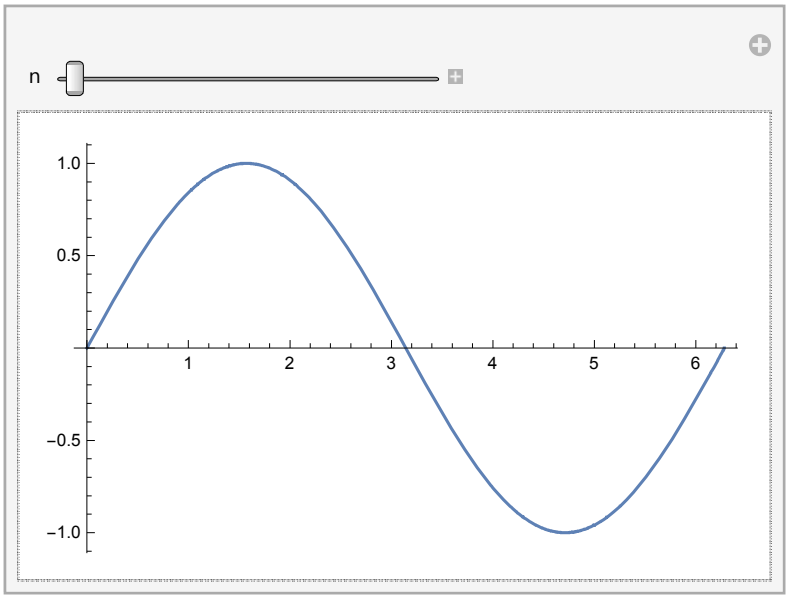
Simply replace the word Table with the word Manipulate, and you get an interactive application that lets you explore values of n with a slider.

```
Manipulate[n, {n, 1, 20}]
```



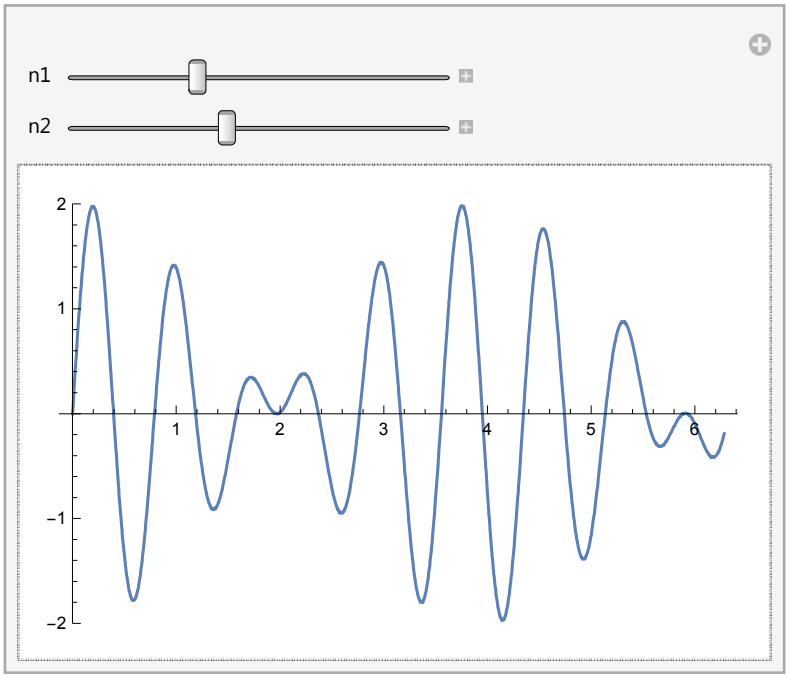
Of course the whole point of Manipulate (and Table for that matter) is that you can put any expression you like in the first argument, not just a simple variable name.Moving the slider in this very simple output already starts to give an idea of the power of Manipulate.

```
Manipulate[Plot[Sin[n x], {x, 0, 2 Pi}], {n, 1, 20}]
```



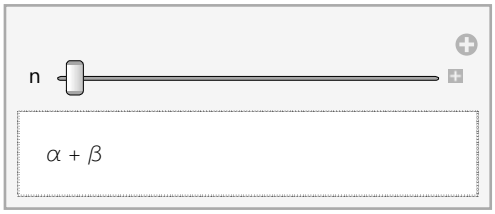
Just like Table, Manipulate allows you to give more than one variable range specification.

```
Manipulate[Plot[Sin[n1 x] + Sin[n2 x], {x, 0, 2 Pi}, PlotRange → 2],
 {n1, 1, 20}, {n2, 1, 20}]
```



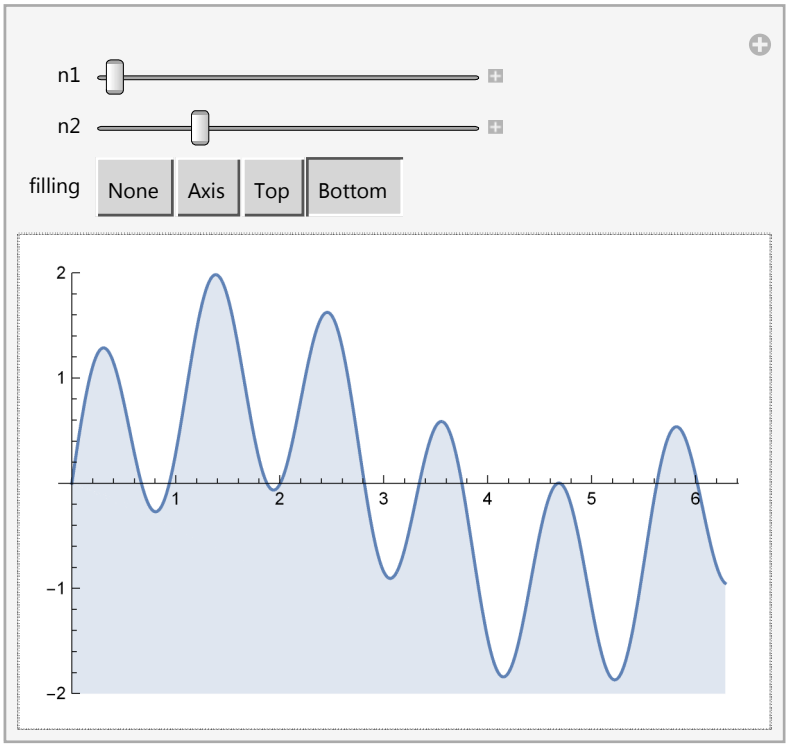
You can also specify an explicit step size. Combining this with an Expand command leads to the following

```
Manipulate[Expand[(α + β)^n], {n, 1, 20, 1}]
```



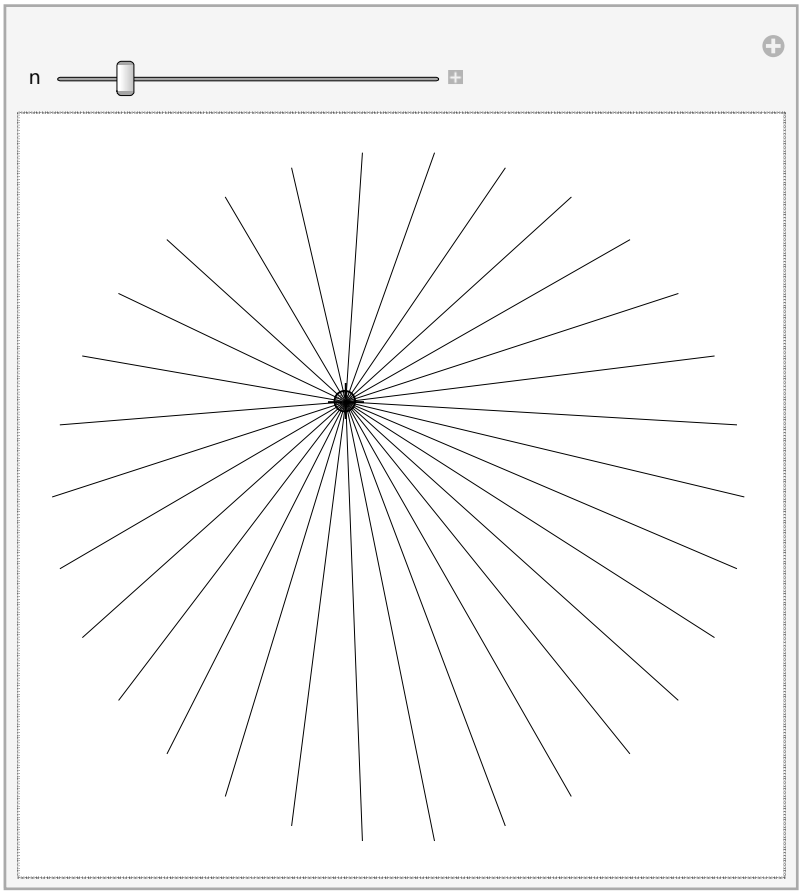
You can also specify the type of controller, instead of always using a slider

```
Manipulate[Plot[Sin[n1 x] + Sin[n2 x], {x, 0, 2 Pi}, Filling → filling, PlotRange → 2],
 {n1, 1, 20}, {n2, 1, 20}, {filling, {None, Axis, Top, Bottom}}]
```



For creating interactive graphics examples, one of the most important features of Manipulate is the ability to place a control point, called a Locator, inside graphics that appear in the output area.

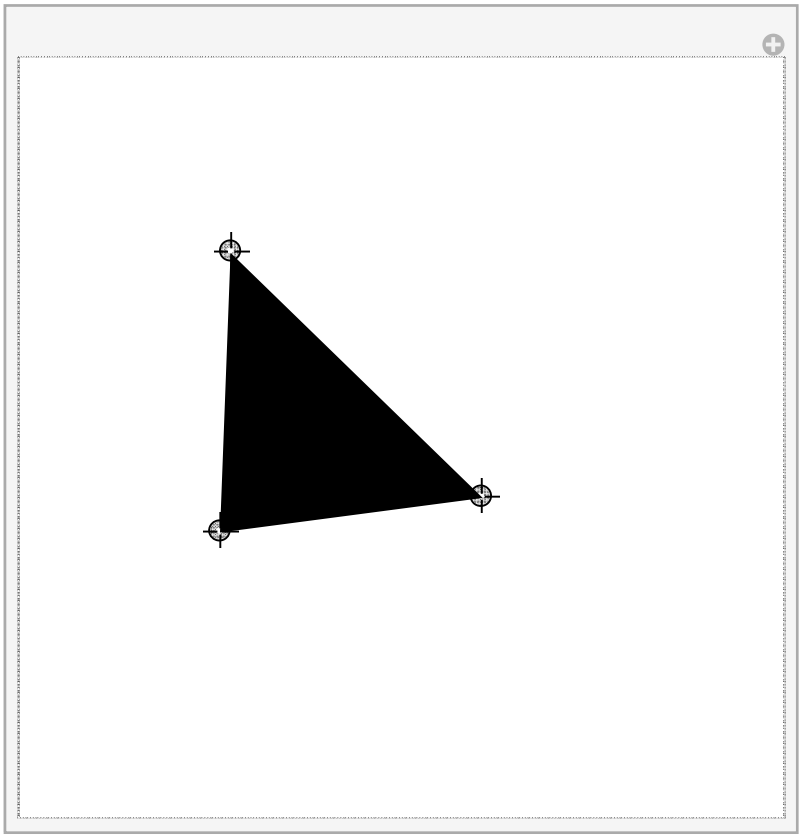
```
Manipulate[
 Graphics[{Line[Table[{{Cos[t], Sin[t]}, pt}, {t, 2. Pi / n, 2. Pi, 2. Pi / n}]]},
  PlotRange → 1], {{n, 30}, 1, 200, 1}, {{pt, {0, 0}}, Locator}]
```



Now you can click anywhere in the graphic and the center point of the lines will follow the mouse as long as you keep the mouse button down.(It is not necessary to click exactly on the center; it will jump to wherever you click, anywhere in the graphic.)

You can have multiple Locator controls by listing them individually, and it is perfectly fine to have a Manipulate with no controls outside the content area, so you can create purely graphical examples.

```
Manipulate[Graphics[Polygon[{pt1, pt2, pt3}], PlotRange → 1],
 {{pt1, {0, 0}}, Locator}, {{pt2, {0, 1}}, Locator}, {{pt3, {1, 0}}, Locator}]
```



Curve Fitting

Let's use what we've learned to create a manipulate applet that actively fits 5 points that we can drag around. First we need to familiarize ourselves with the Fit command.

```
data = RandomReal[{0, 2}, {5, 2}]
```

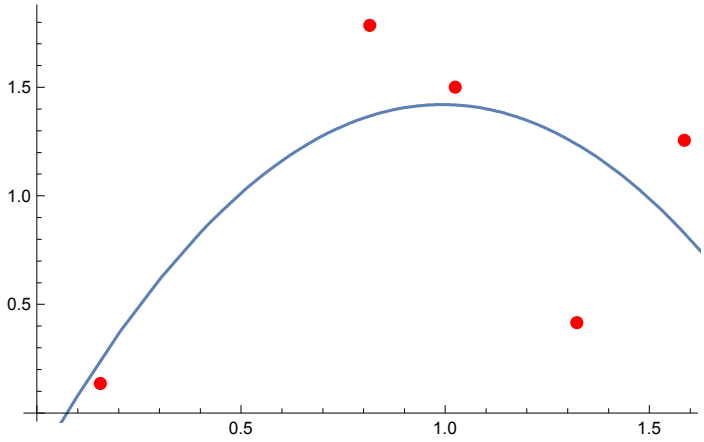
```
{{1.0244, 1.50029}, {1.3225, 0.415692},
 {0.814988, 1.78607}, {1.58592, 1.25574}, {0.155307, 0.135638}}
```

```
parabola = Fit[data, {1, x, x^2}, x]
```

$$-0.236085 + 3.33866\,x - 1.68214\,x^2$$

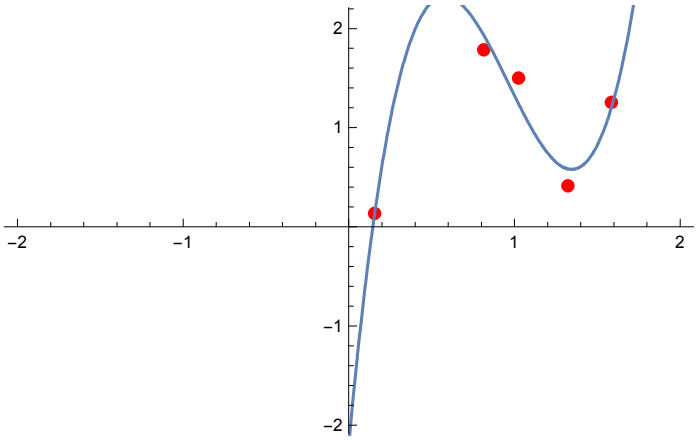
Now we can plot the data and the fitted function

```
Show[ListPlot[data, PlotStyle → Red], Plot[parabola, {x, 0, 5}]]
```



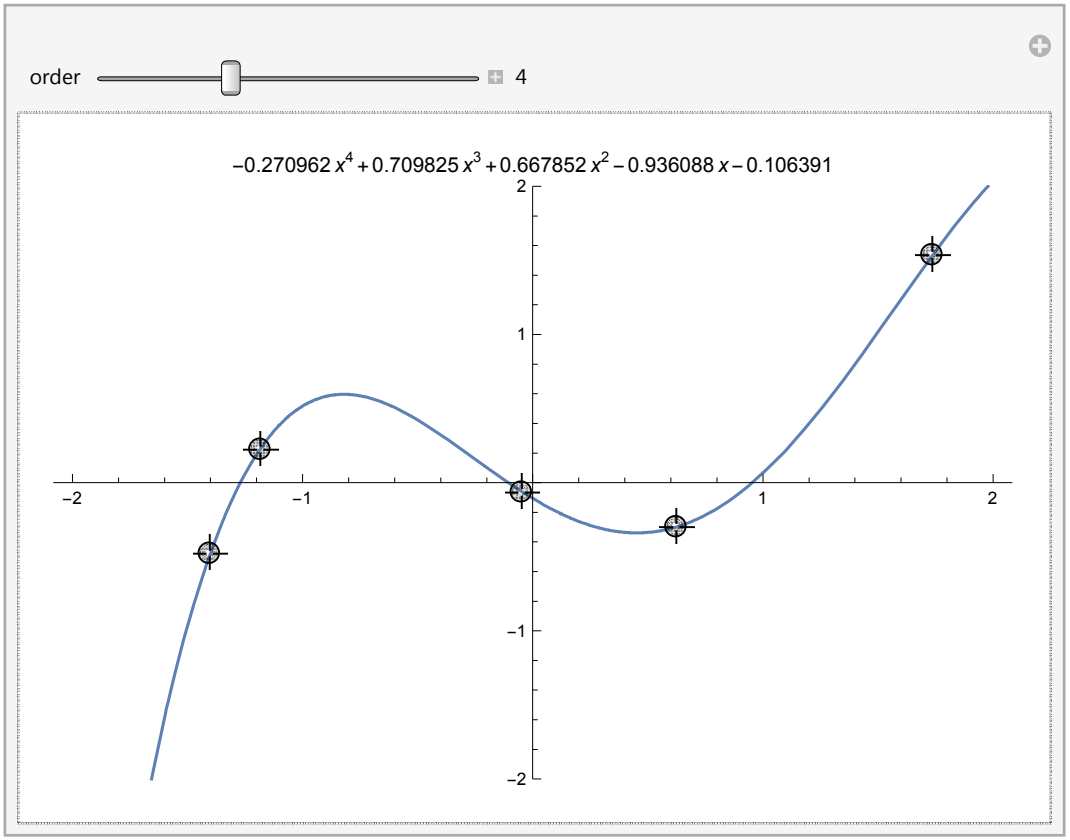
We can put these together and make the order of the fit variable as well

```
order = 3;
Show[ListPlot[data, PlotStyle → Red],
 Plot[Evaluate[Fit[data, Table[x^i, {i, 0, order}], x]], {x, 0, 5}], PlotRange → 2]
```



Now we can put it all together

```
Manipulate[Plot[Evaluate[poly = Fit[points, Table[x^i, {i, 0, order}], x]],
   {x, -2, 2}, PlotRange → 2, ImageSize → 500, PlotLabel → poly],
 {{order, 3}, 1, 10, 1, Appearance → "Labeled"},
 {{points, RandomReal[{-2, 2}, {5, 2}]}, Locator}]
```

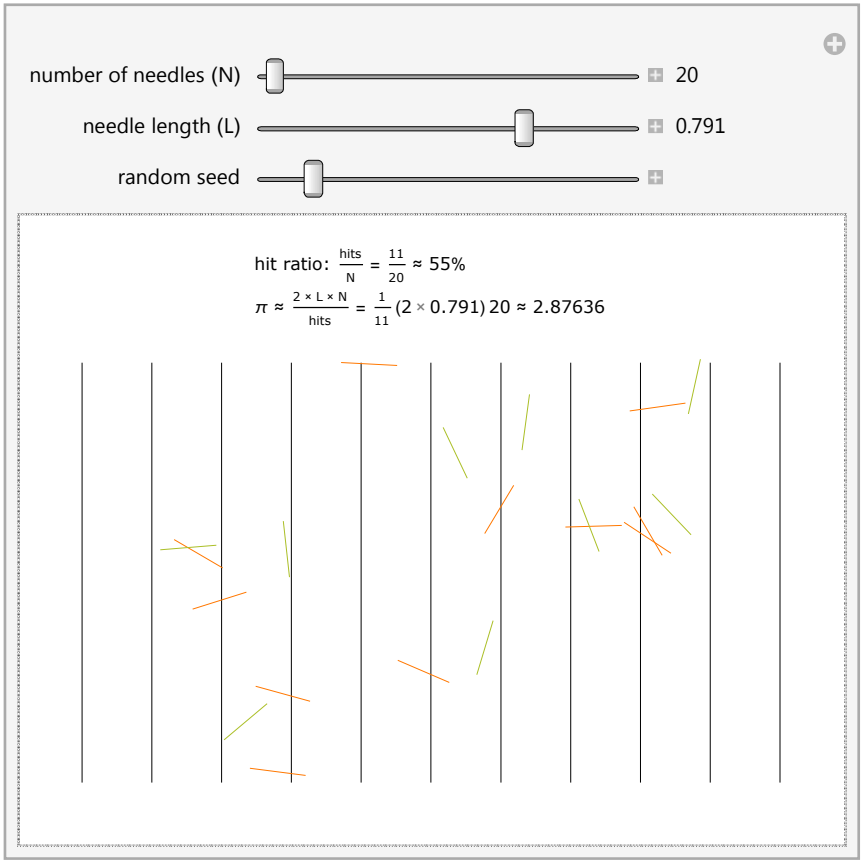


Buffon Needle Problem

Throwing a needle onto a floor with cracks one unit apart actually turns out to be a way to estimate π. Without going into details here, we note that this problem can be solved using integral geometry. Check out this nifty Manipulate applet.

```
BuffonsNeedle[r_, {x_, y_}, n_] := Module[
    {
     data = Table[{{x RandomReal[], y RandomReal[]}, π RandomReal[]}, {n}],
     lines, hits, misses
    },
    Graphics[lines =
      (Function[s, First[#] + s  r / 2 Through[{Cos, Sin}[Last[#]]]] /@ {1, -1}) & /@
       data;
     {
      Line[{{#, 0}, {#, y}}] & /@ Range[0, Ceiling[x]],
      PointSize[.01],
      {misses, hits} = Map[Last, Split[
          Sort[Transpose[{Abs[Subtract @@ (Floor /@ First /@ #) & /@ lines], lines}]],
          (First[#1] == First[#2] &)], {2}];
      Transpose[{
        {RGBColor[1, .47, 0], RGBColor[.67, .75, .15]},
        {Line /@ #} & /@ {hits, misses}
       }]
     }, PlotLabel → Column[{
        Style[
         Row[{"hit ratio: ", ToString[TraditionalForm["hits" / "N"]],
           " = ", With[{a = Length[hits], b = n}, HoldForm[a / b]], " ≈ ",
           ToString[Round[100 Length[hits] / n]], "%"}], "Label"],
        Style[
         Row[{"π", " ≈ ", ToString[TraditionalForm[("2 × L × N") / "hits"]],
           " = ", With[{a = Length[hits], b = n, c = r}, HoldForm[(2 c b) / a]],
           " ≈ ", ToString[N[2 n r / Length[hits]]]}],
         "Label"]}], PlotRange → {{-1 / 2, x + 1 / 2}, {-1 / 2, y + 1 / 2}},
     ImageSize → {400, 300}]
   ];
```

```
Manipulate[SeedRandom[sr];
 BuffonsNeedle[length, {10, 6}, needles],
 {{needles, 200, "number of needles (N)"}, 20, 10 000, 1, Appearance → "Labeled"},
 {{length, .6, "needle length (L)"}, .25, 1, Appearance → "Labeled"},
 {{sr, 12, "random seed"}, 1, 100, 1}, SaveDefinitions → True]
```



Higher Math Topics

We will include here a smattering of different mathematics topics that *Mathematica* is particularly good at. This is not intended to be a thorough overview, just a "warp speed" tour of *Mathematica*'s capabilities

Discrete Math

Amongst other things, *Mathematica* can do graph theory

```
words = DictionaryLookup["wol*"];
```

```
words // Short
```

```
{wold, wolds, wolf, wolfed, wolfhound, wolfhounds, wolfing,
 wolfish, wolfishly, wolfram, wolfs, wolverine, wolverines, wolves}
```

Nearest will select the nearest word in terms of the edit distance.

```
Nearest[words, "wolf", 3]
```

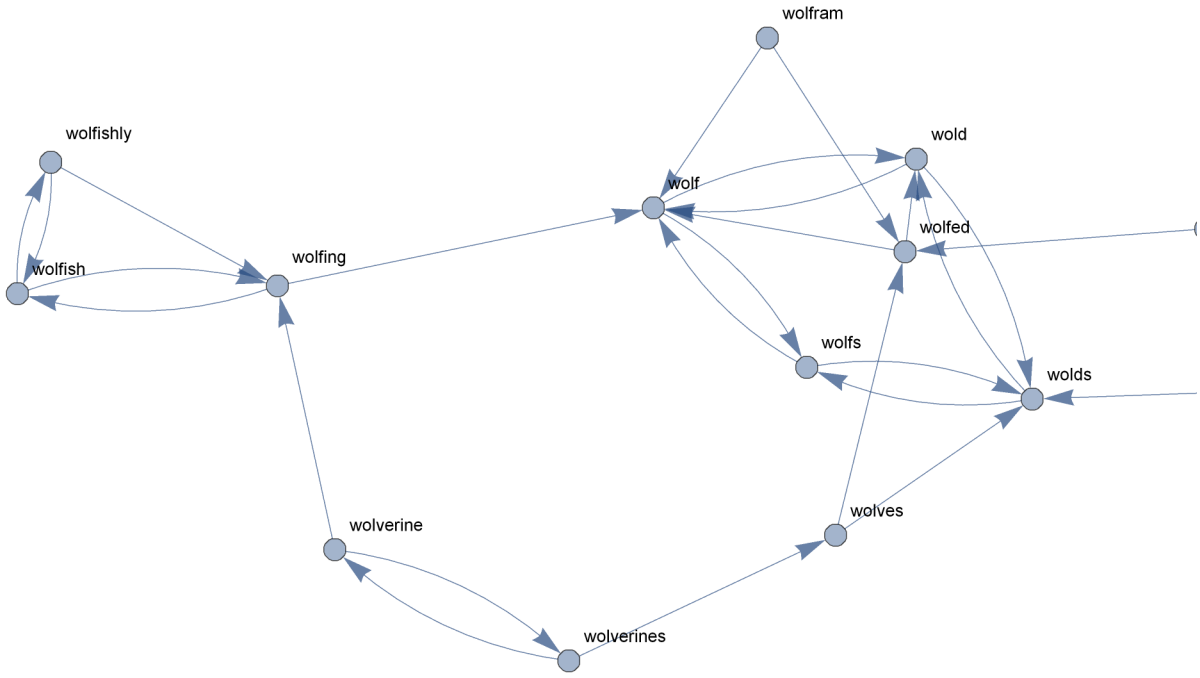
```
{wolf, wold, wolfs}
```

```
wordRules =
  Flatten[Map[(Thread[# → DeleteCases[Nearest[words, #, 3], #]]) &, words]];
```

```
wordRules // Short
```

```
{wold → wolds, wold → wolf, wolds → wold, wolds → wolfs,
 wolf → wold, wolf → wolfs, wolfed → wold, wolfed → wolf, <<13>>,
 wolfs → wolf, wolverine → wolverines, wolverine → wolfing,
 wolverines → wolverine, wolverines → wolves, wolves → wolds, wolves → wolfed}
```

```
Graph[wordRules, VertexLabels → "Name", ImageSize → 550]
```



Number Theory

We can operate in different algebraic number fields (field extensions of the rationals) as well. Let's see what Sqrt[2] looks like when adjoin $2^{1/4}$ to the rationals.

```
ToNumberField[Sqrt[2], 2^(1 / 4)]
```

```
AlgebraicNumber[Root[-2 + #1^4 &, 2], {0, 0, 1, 0}]
```

This means that Sqrt[2] is represented as the 2nd power of the second root of the polynomial $-2 + x^4$ in the rationals adjoined with $2^{1/4}$.

Complex Analysis

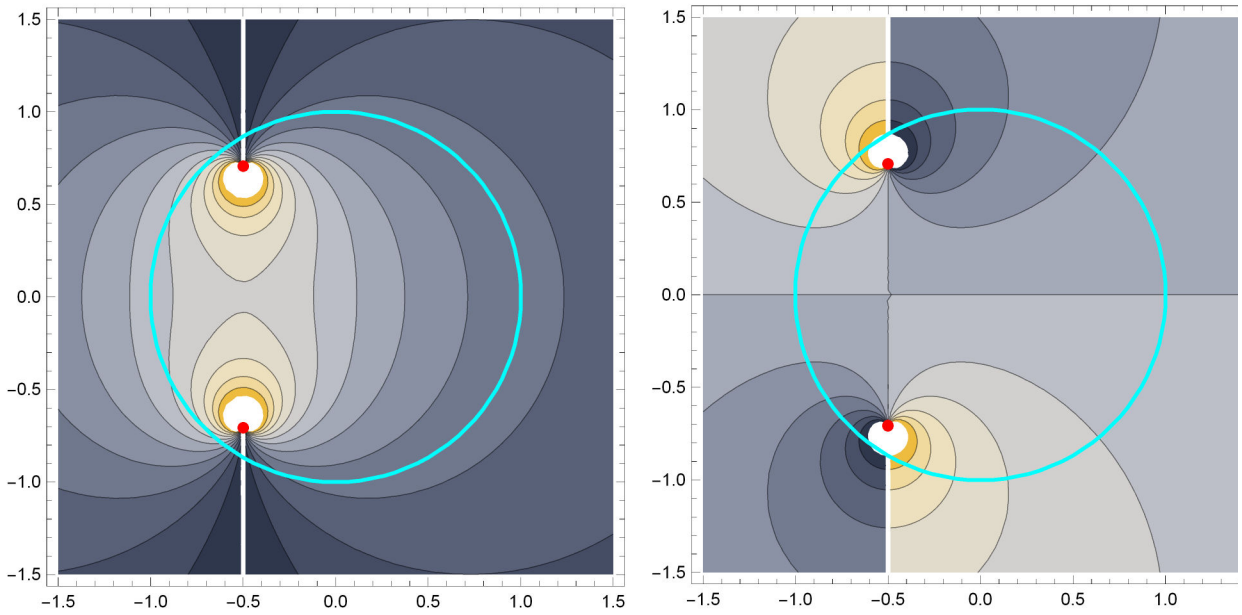
Visualize contour integrals. Here we are considering

$$\frac{1}{\sqrt{4\,z^2+4\,z+3}}$$

```
{z1, z2} = (Solve[4 z^2 + 4 z + 3 == 0, z][[ ;; , 1, 2]])
```

$$\left\{\frac{1}{2}\left(-1-i\sqrt{2}\right),\ \frac{1}{2}\left(-1+i\sqrt{2}\right)\right\}$$

```
GraphicsRow[Table[ContourPlot[g[1 / Sqrt[4 (x + I y - z1) (x + I y - z2)]],
    {x, -1.5, 1.5}, {y, -1.5, 1.5}, ColorFunction → "GrayYellowTones",
    Epilog → {Cyan, Thick, Circle[{0, 0}], Red, PointSize[0.02],
      Point[{Re@#, Im@#} & /@ {z1, z2}]}, Contours → 11], {g, {Re, Im}}]]
```

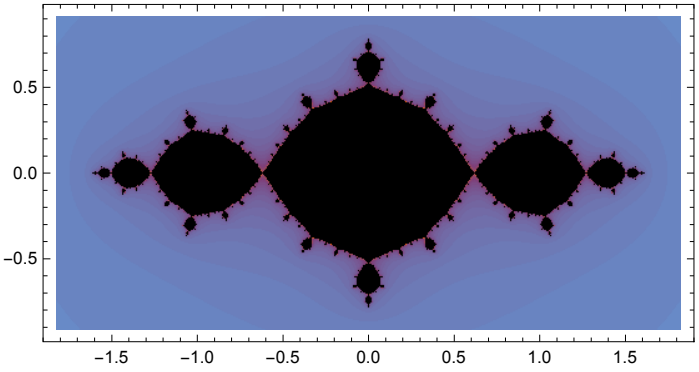


Dynamical Systems

We can easily plot Julia sets.

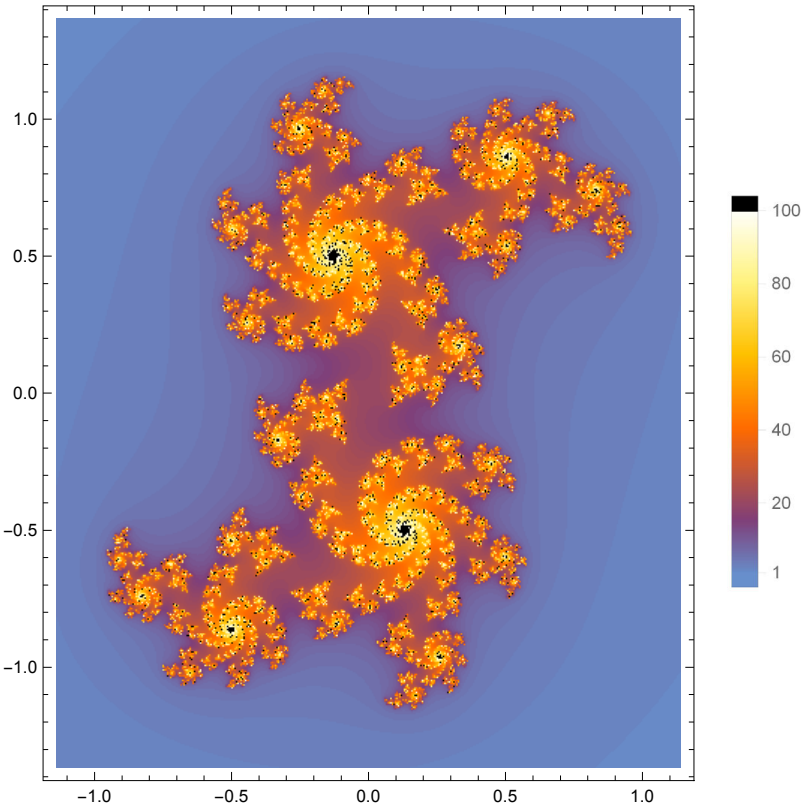
Here's the Julia set of z^2-1

```
JuliaSetPlot[-1]
```



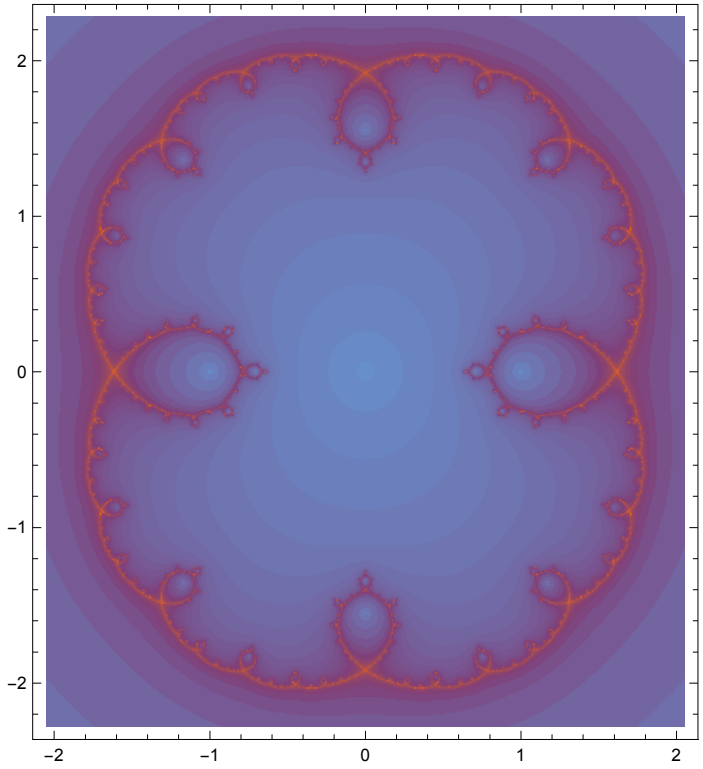
And here's the Julia set of z^2+.365-.37i

```
JuliaSetPlot[0.365 - 0.37 I, PlotLegends → Automatic]
```



We can also plot Julia sets of different rational functions

```
JuliaSetPlot[z^2 / (z^2 - 1), z]
```



Probability

Mathematica has an exceedingly robust integration of probability distributions and probability calculations. Here's a taste of what we can do

This is calculating the expectation of the random variable 2x+3 when x is distributed like a standard normal distribution

```
Expectation[2 x + 3, x ≈ NormalDistribution[]]
```

```
3
```

You can get rather involved with these computations, for example, stipulating that your random variable is given by a mixture of two normal distributions, and then calculating the expectation of a function of this random variable

```
Expectation[ℯ^x + 2, x ≈
  MixtureDistribution[{1, 2}, {NormalDistribution[2, 3], NormalDistribution[4, 5]}]]
```

$$\frac{1}{3}\left(6 + e^{13/2} + 2\, e^{33/2}\right)$$

You can also do this on data

```
data = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

```
Expectation[x^2 + 3, x ≈ data]
```

```
49
```

Which in this simple example, is the same as computing an average

```
Mean[Table[x^2 + 3, {x, data}]]
```

```
49
```

Conditional probabilities are also no problem. Here's the expectation of x^5 when x is distributed as a standard normal distribution, given that x is greater than 2.

```
N[Expectation[x^5 ⧐ x > 2, x ≈ NormalDistribution[]]]
```

```
94.9286
```

Theorem Proving

You can also start to wade into automatic theorem proving using a number of built in Boolean and logical functions.
Here are the conditions for a quadratic expression to be strictly positive over the reals

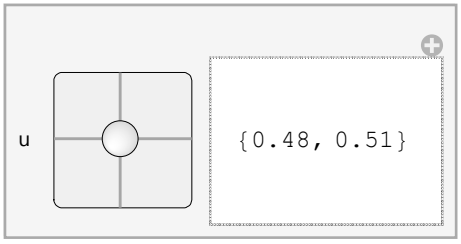
```
Resolve[ForAll[x, a x^2 + b x + c > 0], Reals]
```

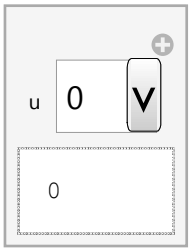
$$(a == 0 \&\& b == 0 \&\& c > 0) \,||\, \left(a \geq 0 \&\& b == 0 \&\& c > 0 \&\& -b^2 + 4\,a\,c > 0\right) \,||\, \left(a > 0 \&\& -b^2 + 4\,a\,c > 0\right)$$

Homework

#1.

There are a number of different control types we haven't covered above. Look at the help menu for the ControlType option of Manipulate to see how you can create the following 2D slider and pop-up menu





#2.

Create a Manipulate applet that demonstrates what happens when you change a, b, c, and d for the following functions. Make your applet so that you can select which function as a button, and use a slider for a, b, c, and d.Try to get as fancy as you like, using the following link as a suggestion for formatting:

http://demonstrations.wolfram.com/ALibraryOfFunctionsWithTransformations/

$$f(x) = a\cos(b\,x + c) + d$$
$$g(x) = a\,(b\,x + c)^2 + d$$
$$h(x) = a\,e^{b\,x+c} + d$$

#3.

Go to http://demonstrations.wolfram.com/ and find a demonstration that you find interesting. Figure out how to obtain the source code for the demonstration, and get it to run in your homework notebook.

#4.

Pretend you are teaching MTH 252 and wish to introduce the concept of Riemann sums. Create an applet that visualizes Riemann sums (while allowing you to specify the number of rectangles drawn under a given curve, as well as the type of rectangle (left, right, midpoint)). The function DiscretePlot (and its option ExtentSize) will be your friend here.

Programming

Basic Programming

Flow Control (procedural programming)

Do, while, for, table, nest, if, which, switch, and with, etc

While Mathematica is a very functionally oriented language (hence the nesting of functions, the usage of Map[ ], etc.), you can also program in a procedural style if you wish (similar to how we programmed in Matlab). Let's look at a few of the basic control flow functions available in *Mathematica*.

The first command we will look at is the For[ ] function. The syntax for For[ ] is
For[start, test, increment, body]
So if we wanted to print out the first 4 integers, we could do that using

```
For[i = 1, i <= 4, i++, Print[i]]
```

```
1
2
3
4
```

Note that "i" has been assigned a value
Be very careful about your commas and semicolons. If you want to include more than one command in the body of the for loop, you need to be sure to end the lines with semicolons. For example

```
For[i = 1, i <= 4, i++, toPrint = i^2; Print[toPrint]]
```

```
1

4

9

16
```

I suggest formatting your For[ ] loops in the following way

```
For[i = 1, i <= 4, i++,
  toPrint = i^2;
 Print[toPrint]
]
```

Keeping the body seperated from the start, test, and increment allows us to better debug errors.

While loops have similar syntax:
While[test, body]

```
myList = {};
i = 0;
While[i < 5,
 AppendTo[myList, i];
 i++;
]
myList
```

```
{0, 1, 2, 3, 4}
```

Note that the While[ ] command does not produce any output (since all the body commands are terminated with semicolons).

Lastly, we have If[ ] statements:
If[condition, true body, false body]

```
If[1 < 2,
 Print["True"],
 Print["False"]
]
```

```
True
```

Once again, it is very important to distinguish the roles of commas and semicolons.

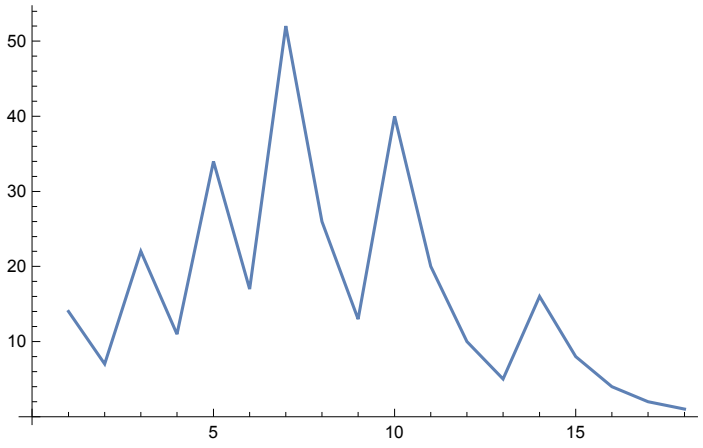
Let's recreate the Collatz Conjecture code, but this time in Mathematica

```
collatzProcedural[n0_] := Module[{iter = 0, myseq, n = n0},
  myseq = {n};
  While[n > 1,
   If[Mod[n, 2] == 0,
    n = n / 2,
    n = 3 n + 1
   ];
   iter++;
   AppendTo[myseq, n];
  ];
  myseq
 ]
```

And now try calling the function

```
collatzProcedural[14]
ListLinePlot[collatzProcedural[14]]
```

```
{14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```



Try n = 27.

Functional Programming

Map, apply, mapthread, nest, nestlist, fold, nestwhile, select, array, sortby, gatherby, select, cases, position

We've already been using functional programming quite a bit (using built in functions like Map, Apply, etc), but I want to emphasize how functionally oriented *Mathematica* is by re-doing the collatz example, but with a functional style.
First off, we need to think about functions that remember values that they have found. When we have a recursively defined function, it would be nice to have it remember values it has seen before. For example, we know that for starting at n=7, the Collatz sequence procceds as follows:
{7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1}
It would be quite the massive shortcut if we could just insert this sequence whenever we ran across the number 7.

Let's look at a simple example of doing this
f[x_]:=f[x]=RHS

Let's look at the recursion relation $S_n = S_{n-1} + S_{n-2}$, with the initial condition $S_o=S_1=1$

```
S[0] = 1;
S[1] = 1;
S[n_] := S[n] = S[n - 1] + S[n - 2]
```

```
? S
```

```
Global`S
```

```
S[0] = 1

S[1] = 1

S[n_] := S[n] = S[n - 1] + S[n - 2]
```

The second equals sign means we are storing the value of S[n-1]+S[n-2] into the variable S[n]. Watch what happens if we evaluate

```
S[5]
```

```
8
```

```
? S
```

```
Global`S
```

```
S[0] = 1

S[1] = 1

S[2] = 2

S[3] = 3

S[4] = 5

S[5] = 8

S[n_] := S[n] = S[n - 1] + S[n - 2]
```

So all the values of S[n] for n=0...5 have now been stored in memory.

Using the paradigm, let's re-do the collatz sequence

```
Clear[collatzFunctional]
collatzFunctional[1] = {1};
collatzFunctional[n_] := collatzFunctional[n] = If[EvenQ[n] && n > 0,
   Join[{n}, collatzFunctional[n / 2]], Join[{n}, collatzFunctional[3 * n + 1]]]
```

```
collatzFunctional[14]
```

```
{14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}
```

Note that quite a bit of previously discovered values have now been stored in memory

```
? collatzFunctional
```

```
Global`collatzFunctional
```

```
collatzFunctional[1] = {1}

collatzFunctional[2] = {2, 1}

collatzFunctional[4] = {4, 2, 1}

collatzFunctional[5] = {5, 16, 8, 4, 2, 1}

collatzFunctional[7] = {7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}

collatzFunctional[8] = {8, 4, 2, 1}

collatzFunctional[10] = {10, 5, 16, 8, 4, 2, 1}

collatzFunctional[11] = {11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}

collatzFunctional[13] = {13, 40, 20, 10, 5, 16, 8, 4, 2, 1}

collatzFunctional[14] = {14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}

collatzFunctional[16] = {16, 8, 4, 2, 1}

collatzFunctional[17] = {17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}

collatzFunctional[20] = {20, 10, 5, 16, 8, 4, 2, 1}

collatzFunctional[22] = {22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}

collatzFunctional[26] = {26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}

collatzFunctional[34] = {34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}

collatzFunctional[40] = {40, 20, 10, 5, 16, 8, 4, 2, 1}

collatzFunctional[52] = {52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1}

collatzFunctional[n_] := collatzFunctional[n] = If[EvenQ[n] && n > 0,
   Join[{n}, collatzFunctional[n/2]], Join[{n}, collatzFunctional[3 n + 1]]]
```

So we are getting the same output as the previous implementation, but compare the speedup

```
AbsoluteTiming[collatzProcedural /@ Range[2000];]
AbsoluteTiming[collatzFunctional /@ Range[2000];]
```

```
{0.812000, Null}

{0.004000, Null}
```

That's about a 200X speedup!

String Manipulation

stringjoin, stringlength, stringsplit, stringtake, stringreplace, stringcases, stringfreeq, stringcount, sequencealignment, characters, <>

Homework

#1.

Remember Pascal's matrix? It's defined elementwise as

$$P_{i,j} = \begin{pmatrix} i+j-1 \\ i-1 \end{pmatrix}$$

Come up with two different ways to create such matrices. The first way should be functional (i.e. involve some combination of Table, Map, Thread, etc.) and the second way should be procedural (i.e. use a For loop, While loop, Do loop, etc).

#2.

Write two programs that will output all pythagorean triples $a^2 + b^2 == c^2$ for 1<=a,b,c<=20. The first program should be procedural (i.e. use a For loop, While loop, etc.) and the second should be functional (no loops, use built in functions such as Apply, Pick, anonymous functions, etc).

#3.

Write a single program that will count the number of divisors of each of the integers from 1 to 10. Your program should use no more than 35 characters to accomplish this. As a bonus, see if you can do it in 27 characters.

Parallel Programming

Parallel Programming

Parallel computing in the Wolfram Language is based on launching and controlling multiple Wolfram Language kernel (worker) processes from within a single master Wolfram Language, providing a distributed - memory environment for parallel programming.Every copy of the Wolfram System comes with all the components and tools to run and create parallel applications.

A first step that may just demonstrate that the system is running is a ParallelEvaluate.If this is the first parallel computation, it will launch the configured parallel kernels.

```
ParallelEvaluate[1 + 1]
```

```
{2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
```

This will launch a number of workers on your system. The number of workers is controlled by the options contained in Evaluation->Parallel Kernel Configuration. Don't try to launch more workers than you have computational threads.

Parallelize

Parallelize is a function in *Mathematica* that tries to parallelize any expression it is given. Typically when combined with simple functions like Table, Map, Count, or other functions where order doesn't matter, this function is quite successful.

Now you can carry out an actual computation. One very simple type of parallel program is to do a search. Let's look for Mersenne primes.

```
Parallelize[Select[Range[5000], PrimeQ[2^# - 1] &]]
```

```
{2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107,
 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423}
```

However, Parallelize doesn't always work

```
Parallelize[Integrate[1 / (x - 1), x]]
```

Parallelize::nopar1 : $\int \frac{1}{x-1}\, dx$ cannot be parallelized; proceeding with sequential evaluation. >>

```
Log[-1 + x]
```

This is due to the fact that we can't break up the Integrate function into smaller computations to be run on independent workers.

ParallelMap/Table/Do/Sum/Product/Array

Instead of using Parallelize, I highly recommend identifying yourself what parts of your computation can be parallelized and then using a function like ParallelMap, ParallelTable, etc.

As a first example, let's sum up a bunch of primes

```
AbsoluteTiming[ParallelSum[Prime[i], {i, 1, 10^7}, Method -> "CoarsestGrained"]]
```

```
{3.068307, 870 530 414 842 019}
```

Let's see what happens if we did this in serial

```
AbsoluteTiming[Sum[Prime[i], {i, 1, 10^7}]]
```

```
{19.432943, 870 530 414 842 019}
```

That's a speedup of around 6 times!

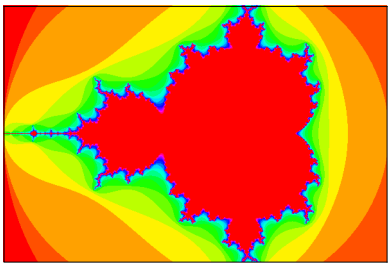
Now for a more real-world example, let's compute the Mandelbrot set in parallel

```
mlength[z_] := Length[FixedPointList[#^2 + z &, z, 20, SameTest → (Abs[#] > 2 &)]]
 (*Stop after 20 iterations,
 or until we start to blow up, that is, get larger than 2*)

AbsoluteTiming[points = ParallelTable[mlength[x + I y],
    {y, -1, 1, 0.005}, {x, -2, 1, 0.005}, Method -> "FinestGrained"];]
```

```
{1.099110, Null}
```

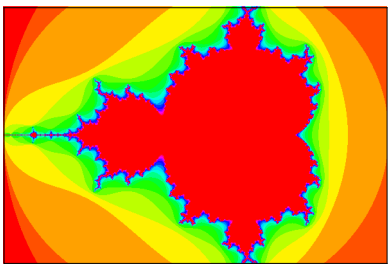
```
ArrayPlot[points, ColorFunction → Hue, PixelConstrained → True]
```



Let's see what happens if we did this in serial

```
AbsoluteTiming[
 points2 = Table[mlength[x + I y], {y, -1, 1, 0.005}, {x, -2, 1, 0.005}];]
ArrayPlot[points2, ColorFunction → Hue, PixelConstrained → True]
```

```
{5.920592, Null}
```



Quite the nice speedup!

Homework

#1.

We will test Girko's circle law, which states that the eigenvalues of a set of random N x N real matrices with entries independent and taken from the standard normal distribution (i.e. using RandomVariate[NormalDistribution[],{N,N}]) are asymptotically constrained inside a unit circle of radius Sqrt[N] in the complex plane. Create a program that calculates (in parallel! Try ParallelTable) the eigenvalues of many (~1000) normally distributed random matrices of size 100x100. Plot the real and imaginary parts of all these eigenvalues on the plane (i.e. use the real parts as x-values and the imaginary parts as y-values), and draw a circle of radius Sqrt[100] around it to verify Girko's circle law. How long would it have taken if you performed this task in serial instead of parallel?

#2.

Download the 4 FASTA files located in the ZIP file at http://www.math.oregonstate.edu/~koslickd/TestSequences.zip. Count the number of bases (i.e. A,C,T,G) in each sequence (using ParallelMap and StringCount). Tally up the total number of Gs that appear in each file. Which file has the most Gs? How much quicker did this task take performing it in parallel rather than in serial?

Disclaimer

Parts of these notes are based on G.E. Keogh's "Mathematica Notes" as well as *Mathematica*'s built in help menus.
Feel free to copy any or all of these notes for any academic purpose.

Mathematica Homework

Homework Problems #1, Basics. Due 11/6/14

#1.

Order the following five numbers from smallest to largest

$$7.149\,\pi,\quad \pi^{e},\quad e^{\pi},\quad \left(\frac{1+\sqrt{5}}{2}\right)^{11\,\pi/5},\quad (2+e)^{\pi-1}$$

#2.

Assign a variable to each of the following symbolic expressions. After verifying that you have entered the expression correctly, use the substitution syntax

```
varible /. x → {}
```

to evaluate it at the indicated value. Then give a 10-digit numerical approximation of the expression.

- $\sqrt{\frac{e^{\sin(x+1)}}{\cos(x)+1}}$, at x=0
- e^{x^3}, at x=2
- $\frac{\sqrt{16-x^2}+1}{2x}$, at x=3
- $|4\cos(x)+\pi|$, at x=π

#3.

The following expression is a well-known mathematical constant. Use *Mathematica* to figure out which constant it is equal to. Can you use a built in function to prove that your guess is correct?

$$4\tan^{-1}\left(\frac{1}{4}\right)+4\tan^{-1}\left(\frac{3}{5}\right)$$

#4.

Find the smallest positive integer *n* that satisfies

$$n!>n^{10}+n^{6}+10$$

#5.

Stirling' s formula provides an approximation for *n!,* in the form of

$$\frac{\sqrt{2\,\pi\,n}\;\;n^n}{e^n}$$

Using substitution, evaluate this formula for the values of n=20,40,80, and 160. Compare your results for 20!, 40!, 80! and 160!.

#6.

Evaluate the quantity

$$\frac{\left|\frac{\sqrt{2\,\pi\,n}\;\;n^n}{e^n}-n\,!\right|}{n\,!}$$

for the values of n give in problem #5. Why can we say that “Stirling’s formula approximates *n!* with an error on the order of *1/n*”?

#7.

Is the following a good approximation for Log[n]?

$$n\left(n^{1/n}-1\right)$$

#8.

Find the integral power of 3 that is closest to each of one million, one billion, and one trillion.

#9.

Get Mathematica to tell you the following:
a) What the number of stars in our galaxy is
b) The height of the tallest person
c) The US’s per capita beer consumption (compare this to the highest beer consuming country in the world via *Mathematica*).

#10.

Using *Mathematica*’s free form input, figure out if there are enough people in the USA to have them stand on each others’ heads to reach from the earth to the moon (assuming that everyone is of average height).

Homework Problems #2, Lists. Due 11/11/14

#1.

Define a list which gives the first 8 odd numbers. Give this list the name oddList. Try the following commands and comment on what they do.

```
Reverse[oddList]
Append[oddList, 0]
FullForm[oddList]
Length[oddList]
Delete[oddList, 4]
Insert[oddList, 100, 3]
oddList = ReplacePart[oddList, 3 → -1]
Take[oddList, 4]
Drop[oddList, 4]
RotateLeft[oddList, 5]
RotateRight[oddList]
```

#2.

Create a list of random numbers (between 0 and 1) with length given by the 123rd prime number. Assign this list to a variable, and then calculate the mean, total, and standard deviation of this list. Next, extract every 3rd item in this list and calculate the mean of this list.

#3.

type the following to define a very complex “list of lists of lists of...”

```
multiList = {{{{{1, 2, 3 {{4, 5}}}, 6}}, {7, 8}}, 9};
```

Test the effect of the following command and comment on the behavior.

```
Flatten[multiList]
Flatten[multiList, 1]
Flatten[multiList, 2]
```

Then set

```
flatList = Flatten[multiList];
```

and try

```
Partition[flatList, 3]
Partition[flatList, 4]
Partition[flatList, 3, 1]
```

comment on the effect of these commands.

#4.

Given that

```
list1 = {1, 2, 3, 4, 5, 6};
list2 = {5, 6, 7, 8, 9, 10};
```

Use *Mathematica* to find the intersection, union, and symmetric difference of these sets.

#5.

Use “Apply[ ]” to find the product of the first 100 primes.

#6.

Use “Map[ ]” to find the derivatives of the following functions, returning them as a list given in the same order

$$\{x^2, \log(x), \cos(x)\}$$

#7.

Create a list of 10^7 random integers between 0 and 9. Add up all the elements of this list in three different ways:
1. Using the built-in Total function.
2. Using Apply and Plus
3. Using the Sum function
Verify that all three approaches return the same result. Next, using the Timing function, test which method is fastest.

#8.

Count how many of the first N=10,000 natural numbers are square free (that is, they contain no prime factor of the form p^2). Make a conjecture about what happens to the fraction of square free numbers as N goes to infinity.

#9.

The Perron-Frobenius Theorem is a very useful theorem that asserts that a real square matrix with positive entries has particular constraints on its spectrum (eigenvalues and vectors). The theorem is often used in dynamical systems and probability. One of the consequences of the theorem is that for particular matrices (called irreducible and non-negative), the following sequence has a certain behavior

$$\left\{ \frac{A^k}{r^k} \right\}_{k=1\ldots\infty}$$

Where “r” is the largest eigenvalue of “A”. Using the matrix

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

(which is irreducible and nonnegative), conjecture about what happens to the above sequence.

Homework Problems #3, Functions. Due 11/13/14

#1.

Define two functions

$$f(x) = 3x + 19$$

$$g(x) = \frac{x - 19}{3}$$

And use *Mathematica* to algebraically demonstrate that they are inverses of each other.

#2.

An ellipse with a semi-major axis of length “a” and a semi-minor axis of length “b” has a perimeter of approximately

$$f(a, b) = 2\pi \sqrt{\frac{1}{2}\left(a^2 + b^2\right)}$$

Define this function in *Mathematica* and use it to approximate the perimeter of an ellipse having a major axis of length 5 and a minor axis of length 4.
What is the average perimeter for 10,000 ellipses with randomly distributed (on [0,1]) semi-major/minor axes?

#3.

The length of a parabolic arc from (0,0) to (x,y) along the parabola y^2=x is given by the arc length function “f” for points (x,y)=(x,Sqrt[x]) on the curve in the first quadrant by writing

$$f(x, y) = \frac{1}{2}\left(\frac{y^2 \log\left(\frac{u+2x}{y}\right)}{2x} + u\right)$$

where

$$u = \sqrt{4x^2 + y^2}$$

Write a function f(x,y) that computes the length of the arc from (0,0) to (x,y), according to the formula above. Use the Module function to define f. Next, use this function to evaluate the arc length at the points (4,2) and (9,3).

#4.

The function

$$g(x) = \begin{cases} 2x & 0 \le x \le \frac{1}{2} \\ 2x - 1 & \frac{1}{2} \le x \le 1 \end{cases}$$

defined on [0,1] is called the baker's transformation. Consider a glob of dough of length 1 that is to be kneaded in a certain fashion. If a point of the dough is a distance x from the end, with 0<x<1, let g(x) represent its position after kneading the dough once.
Define the function g, and using the built-in function NestList, determine the sequence of positions attained by the point of dough starting at x=1/10 through several repeat applications of g. Is there a pattern?

#5.

Create a function that searches for the first occurrence of string in π using the convention that "a"->1, "b"->2, etc. For example, to search for my name "David" I would look for the first occurrence of the pattern 4 1 22 9 4 (so the digits {4,1,2,2,9,4}).
This function should have two arguments: the first gives the pattern, and the second tells how far in π to look. For example, your syntax should look like

```
lookInPi[{4, 1, 2, 2, 9, 4}, 1 000 000]
(*Looks for my name in the first 1 Million digits of Pi*)
```

Given an arbitrary pattern, is it guaranteed that this function will return a result if allowed access to an arbitrary number of digits of π?

#6.

The logistic map is typically shown as the first example of how chaos can come from a very simple non-linear dynamical system. This dynamical system is described the in the following discrete manner

$$x_{n+1} = r\, x_n\, (1 - x_n)$$

Define the function

$$f(x, r) = r\, x\, (1 - x)$$

so that this dynamical system can be described by composing f with itself many times over. i.e. we can compose f with itself many times over

$$\{x,\ f(x, r),\ f(f(x, r), r),\ f(f(f(x, r), r), r),\ \ldots\}$$

and see what happens to the sequence of numbers.
a) If r=.1, what happens to this dynamical system in the long run?

b) if r=3.2, what happens to this dynamical system in the long run?

Homework Problems #4, Graphing. Due 11/18/14

#1.

In how many points do the following ellipses intersect? Hint: graph them.

$$\frac{1}{4}(x-1)^2 + y^2 = 1$$

$$(x-2)^2 + \frac{1}{4}(y-2)^2 = 1$$

#2.

Graph the following functions on the same axis.

$$e^x$$

$$1$$

$$x+1$$

$$\frac{x^2}{2} + x + 1$$

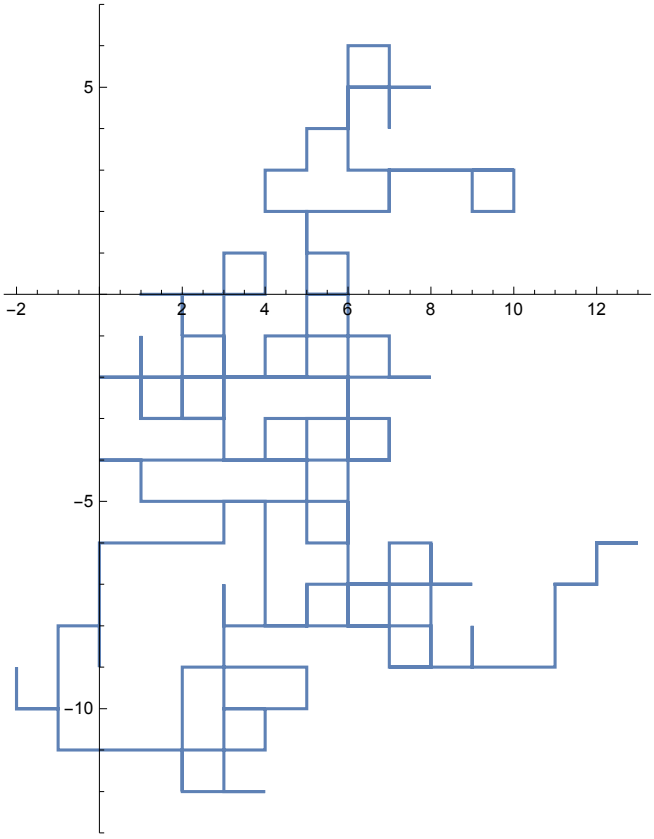
$$\frac{x^3}{6} + \frac{x^2}{2} + x + 1$$

$$\frac{x^4}{24} + \frac{x^3}{6} + \frac{x^2}{2} + x + 1$$

Include plot labels. Comment on the pattern that you see developing, and comment on why this might be the case.

#3.

Similar to the random walk in 2 dimensions given in the notes above, create a random walk constrained to the lattice. That is, the only allowable moves are to move one unit west, east, north, or south. The function RandomChoice will come in handy here. An example output is

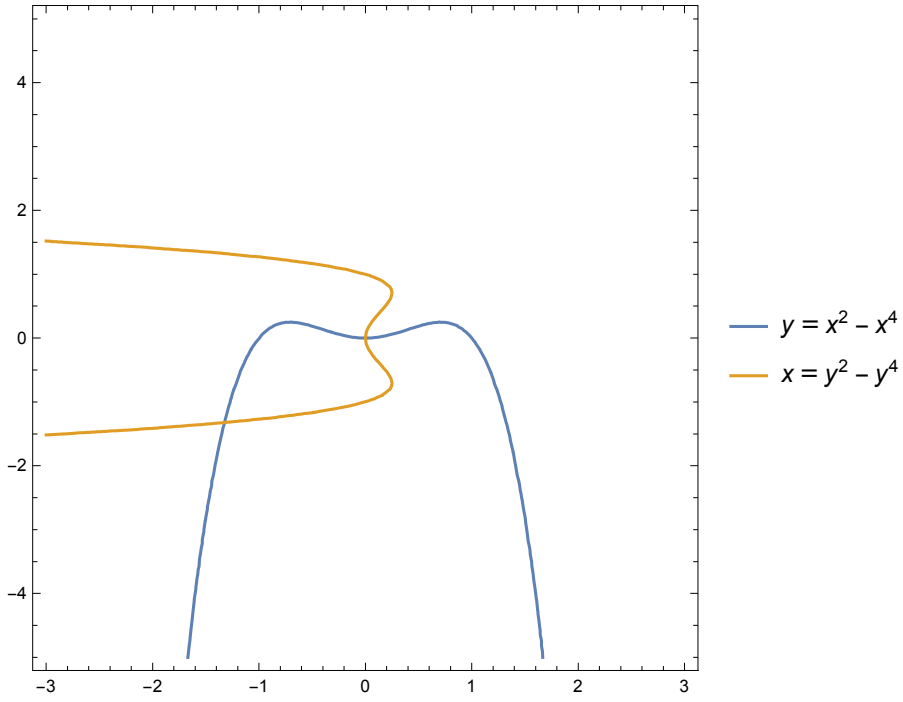


#4.

Write a function called plotSwitchXY that returns a plot of a given equation, along with the plot of the same equation with the roles of x and y reversed. For example

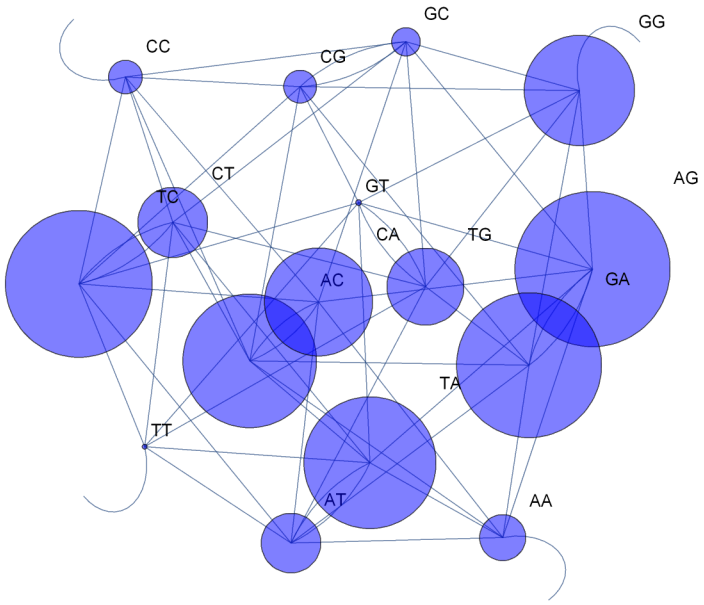
```
plotSwitchXY[x^2 - x^4, {x, -3, 3}]
```

should return a plot that looks like



#5.

A de Bruijn graph is a directed graph representing overlaps between sequences of symbols. There is a built in function called DeBruijnGraph that creates such a graph. By using the options GraphLayout, EdgeShapeFunction, VertexLabels, VertexSize, and VertexStyle, see if you can replicate a figure such as the following. The plot underlying this figure was DeBruijnGraph[4,2]. The vertex sizes were chosen at random.



Homework Problems #5, Graphing. Due 11/20/14

#1.

Similar to the Matlab problem, Create a 10^4 x 3 matrix where the rows correspond to 1, 2 ... 10^4 and the columns correspond to the functions sin(x), log(x), and x^2. Let the (i,j) entry be the value of the jth function evaluated on i. Save this matrix as a *.mat file, a comma separated file, text file, and as an Excel file. Include these files in the dropbox folder when you complete this homework assignment.

a) Which format took the longest/shortest amount of time to write?
b) Which format used the least/most amount of disk space?
c) Which format took the longest/shortest amount of time to read in?

#2.

Use Import to read in the FASTA files located at http://www.math.oregonstate.edu/~koslickd/TestSequences.zip. Note that you can use Import directly on web links. You will have to first find the names of the files in this zip file (hint: Import[link.zip, “FileNames”]). Report the length of each one of the files.

#3.

Use GenomeData to find the 5 shortest and 5 longest genes in the human genome.

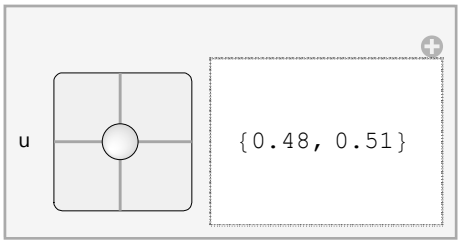
#4.

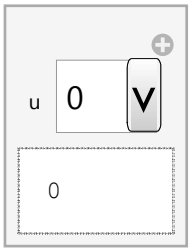
Optional: If you use Facebook, try replicating the commands I used above in the API's and Social Media section. You do not need to include this in the homework you turn in.

Homework Problems #6, Manipulate. Due 11/25/14

#1.

There are a number of different control types we haven't covered above. Look at the help menu for the ControlType option of Manipulate to see how you can create the following 2D slider and pop-up menu





#2.

Create a Manipulate applet that demonstrates what happens when you change a, b, c, and d for the following functions. Make your applet so that you can select which function as a button, and use a slider for a, b, c, and d.Try to get as fancy as you like, using the following link as a suggestion for formatting: http://demonstrations.wolfram.com/ALibraryOfFunctionsWithTransformations/

$$f(x) = a\cos(b\,x + c) + d$$
$$g(x) = a\,(b\,x + c)^2 + d$$
$$h(x) = a\,e^{b\,x+c} + d$$

#3.

Go to http://demonstrations.wolfram.com/ and find a demonstration that you find interesting. Figure out how to obtain the source code for the demonstration, and get it to run in your homework notebook.

#4.

Pretend you are teaching MTH 252 and wish to introduce the concept of Riemann sums. Create an applet that visualizes Riemann sums (while allowing you to specify the number of rectangles drawn under a given curve, as well as the type of rectangle (left, right, midpoint)). The function DiscretePlot

(and its option ExtentSize) will be your friend here.

Homework Problems #7, Programming. Due 11/26/14

#1.

Remember Pascal's matrix? It's defined elementwise as

$$P_{i,j} = \binom{i+j-1}{i-1}$$

Come up with two different ways to create such matrices. The first way should be functional (i.e. involve some combination of Table, Map, Thread, etc.) and the second way should be procedural (i.e. use a For loop, While loop, Do loop, etc).

#2.

Write two programs that will output all pythagorean triples $a^2 + b^2 == c^2$ for 1<=a,b,c<=20. The first program should be procedural (i.e. use a For loop, While loop, etc.) and the second should be functional (no loops, use built in functions such as Apply, Pick, anonymous functions, etc).

#3.

Write a single program that will count the number of divisors of each of the integers from 1 to 10. Your program should use no more than 35 characters to accomplish this. As a bonus, see if you can do it in 27 characters.

Homework Problems #8, Programming. Due 12/2/14

#1.

We will test Girko's circle law, which states that the eigenvalues of a set of random N x N real matrices with entries independent and taken from the standard normal distribution (i.e. using RandomVariate[NormalDistribution[],{N,N}]) are asymptotically constrained inside a unit circle of radius Sqrt[N] in the complex plane. Create a program that calculates (in parallel! Try ParallelTable) the eigenvalues of many (~1000) normally distributed random matrices of size 100x100. Plot the real and imaginary parts of all these eigenvalues on the plane (i.e. use the real parts as x-values and the imaginary parts as y-values), and draw a circle of radius Sqrt[100] around it to verify Girko's circle law. How long would it have taken if you performed this task in serial instead of parallel?

#2.

Download the 4 FASTA files located in the ZIP file at http://www.math.oregonstate.edu/~koslickd/TestSequences.zip. Count the number of bases (i.e. A,C,T,G) in each sequence (using ParallelMap and StringCount). Tally up the total number of Gs that appear in each file. Which file has the most Gs? How much quicker did this task take performing it in parallel rather than in serial?

Part 3: LaTeX

The following is an excerpt from the textbook written by Kevin Cooper for Math 300 at Washington State University. It is reproduced here with permission.

Chapter 3

LaTeX

LaTeX is a markup language. It is essentially an upgrade of Donald Knuth's landmark typesetting language called TeX [3]. In the late 1970s Knuth realized that printing technology had neared a point at which very sophisticated typesetting could be done by computers. He realized further that mathematical and technical typesetting had been entirely ignored by the primitive programs of the day. He undertook to create a typesetting language that would be able to produce book-quality output for all subjects, including mathematics and science.

The original "plain TeX" was very powerful, but in many ways difficult to use. Leslie Lamport [4] created a package of additional commands to simplify and extend the abilities of TeX. This package was called LaTeX, and has become the defacto standard for typesetting highly technical documents, particularly those containing mathematical notation. It is so prevalent that many publishers of mathematical papers provide their own class and style files [5].

The "tags" of the markup language are identified by starting with a backslash (\ ), i.e. every LaTeX command or variable starts with a backslash. As with other markup languages, every LaTeX document has a preamble section and a body section. Unlike most other markup languages with which we are familiar, there are many commands in LaTeX that do not require, or even possess, closing tags.

In any computer language, it is common to make one's first application a so-called "Hello, world!" program. Such a document in LaTeX follows.

```
\documentclass{article}
\% Any text following a percentage sign is ignored - a comment
\begin{document}
Hello, world!

This is a second paragraph.
\end{document}
```

The structure of a basic LaTeX document is evident. The file must begin by declaring the class of the document – the "article" class here. Other possibilities

include “book”, “report”, and “letter”. Following that are preamble commands, which pertain only to formatting and new command structures. The body of the document begins with the \begin{document} statement. Any text that is not preceded by a backslash actually appears on the formatted page.

Note that the blank line in the body of the document makes a new paragraph. It is very important to watch blank lines - starting a new paragraph can generate errors if it is done e.g. in the middle of an equation.

Note also that there are several special characters that cannot be used in an ordinary way. They include \, _, ^ , {, }, &, #, ~ , %, and $. If you actually want one of these characters to appear in your document, you must enter them as \textbackslash, _, \^{} , \{, \}, \&, \#, \~{}, \% and \$, respectively. Some of these characters are only valid in math mode, or require some extra tricks to manage.

3.1 The \documentclass Statement

The line that sets many properties that apply to the entire document is the first. In the \documentclass statement, we choose the basic format settings for the document. The choices include: article, book, letter, report, and beamer. The article class is designed for academic papers and class reports. It uses the same margins for both even- and odd-number pages, sets no extra page headers, and does minimal titles. By contrast, the book class makes an entire page for the title and author information, uses different margins for even- and odd-numbered pages, and puts the section heading at the top of odd-numbered pages, while the chapter heading goes to the top of even-numbered pages. We will discuss the beamer class in some detail later, but it is used to make a “powerpoint” style presentation.

There are two other classes that are useful for mathematicians and scientists. The amsart and amsbook classes replace the article and book classes, respectively, and are mostly interchangeable with those. The chief difference is that the AMS classes load special AMS formatting packages automatically. When we want to use those packages in e.g. the ordinary article class, we must load them explicitly:

```
\usepackage{amsmath,amsfonts,amsthm,amssymb}
```

The documentclass command takes many optional arguments. Probably the most important is a specification for the font magnification. Specifying

```
\documentclass[10pt]{article}
```

typesets an article using a 10 point default font. All other fonts are scaled accordingly. In other words, when we change that option to 12pt, then the entire page is scaled up in size by a factor of 1.2. All of the headings, footnotes, and other fonts that are different from the default are scaled. The point is that that specification is not so much a designation of the font size for the paper as a magnification. The only choices are 10pt, 11pt, and 12pt.

Another useful option for the documentclass command specifies whether to place equation numbers to the left or right. These options are `leqno` and `reqno`, respectively. Note that the ordinary `article` and `book` classes place equation numbers to the right by default. The AMS classes place them to the left.

TEX installations typically set default paper sizes according to the standards of the continent where they exist. European installations should set the default paper size to A4, which is somewhat longer and narrower than North American letter size. If you need to change the size of the paper that LATEX is to work with, you can set the `letterpaper` or `a4paper` optional arguments.

There are many other options, some of which are only valid for particular choices of document class. The `beamer` class in particular has a number of extraordinary options available. We will discuss these when we describe the beamer package.

3.2 Headings

Creating headings in LATEX is easy, and the system tries to help by keeping track of many things for you. For example, you could create headings as in simpleminded word processing programs by typing them in directly and using a larger font, but LATEX prefers to memorize those headings, count the sections for you, and then offer those titles and numbers for the sake of reference whenever you want them.

In most document classes, the best way to create a title area for our documents is to set the values for the text to appear there in the header. For example we could typeset the title for this document using the commands

```
\title{My Document Title}
\author{Your Name Here}
```

specify the title and author for the document we are working on. This does not typeset those values anywhere. In order for the title area actually to appear in the document, we must invoke `\maketitle` inside the document, i.e. between the `\begin{document}` and the `\end{document}` statements. Needless to say, most people would put this statement as the first thing in the body of the document.

Inside the document, we can create headers using commands such as `\section`. These commands accomplish several things: they typeset the appropriate heading in some uniform way, according to the specifications of whatever class we are using; they number the heading; and they allow us to reference that number if we need to. The headings for which this works include `\part`, `\chapter`, `\section`, `\subsection`, `\subsubsection`, `\paragraph`, and `\subparagraph`. The `article` class does not include the `\chapter` heading.

Sometimes we want the heading, but we do not want it to be numbered, or for the associated counter to be incremented. When that is the case, we can change the appropriate command slightly by appending an asterisk (*). Thus

```
\section*{My Unnumbered Section}
```

creates the header without all the numbering.

3.3 Mathematical Typesetting

LaTeX was created for typesetting mathematics. There are many fine points to mathematical typesetting that are easy to overlook when we simply read math books. We discussed these in Section 2.4, but recall that in order to typeset mathematics, we require a special "math italic" font set, with different spacings than ordinary text, we need to be able to do complicated alignments both horizontally and vertically, and we use a huge collection of special symbols, diacritical marks, and notations.

3.3.1 Math Mode

In TeX, as in MathML, we need to change to a special "math mode" whenever we need to type any mathematical symbol. In a line this is done by surrounding our mathematical text with dollar signs, as in x, or alternatively using a begin and end symbol: `\(x\)`. If we want to *display* the expression (make it centered, on its own line, with extra space around it, and a more expansive format), we must use a display math environment:

```
\begin{displaymath}
f(x) = x+2.
\end{displaymath}
```

or

```
$$f(x) = x+2.$$
```

or

```
\[f(x) = x+2.\].
```

or

```
\begin{equation*}
f(x) = x+2.
\end{equation*}
```

All of these produce the same thing:

$$f(x) = x + 2.$$

We will henceforth use the double-dollar-sign formulation to start display math mode without equation numbers. This is probably the oldest form, inasmuch as it dates from the original (plain) TeX.

To have your equations numbered automatically as well as set in a display mode, use the equation environment.

```
\begin{equation}
f(x) = x+2.
\end{equation}
```

This gives the same displayed equation as above, but with an equation number:

$$f(x) = x + 2. \tag{3.1}$$

Note that this is almost the same as one of the formats we described earlier – it differs by an asterisk from `\begin{equation*}...\end{equation*}`. This is something found often in LaTeX. There are many environments that have asterisks in their names for which the only difference is that there is no number associated with them.

3.3.2 Math Notation

One of the most basic differences between mathematical typesetting and ordinary text is the prominent use of subscripts and superscripts in math. This is very easy to do in TeX math mode. A subscript is created by placing an underscore (_) character followed by the subscript object. Thus `$X_1$` produces X_1; `$a_i$` produces a_i. If there is more than one object in the subscript, we must use braces to group the objects so that they appear as one thing to the subscript operator; e.g. `$a_{i+1}$` produces a_{i+1}, whereas `$a_i+1$` produces $a_i + 1$.

Making superscripts is very similar, but uses a caret (^) instead of the underscore. Thus, `$x^2$` produces x^2; `$e^{-x}$` produces e^{-x}.

We see here the use of grouping in LaTeX to make several objects appear as one to certain commands. Most commands in TeX require a specific number of inputs. When those input arguments are single characters or even single symbols then there is no ambiguity. However, when those arguments are more complex constructions, we use braces { and } to enclose them as a group.

Another way mathematical notation differs from ordinary typesetting is the common use of Greek letters and other special symbols. These are usually easy to guess: their LaTeX names typically are just their names spelled out, preceded by a backslash. Upper-case Greek letters spell the names using a capital letter. For example, `\lambda` produces λ, while `\Lambda` gives Λ. Note in particular that `\int` produces an integral sign, `\sum` produces a summation symbol (do not use `\Sigma`!), and `\infty` makes an infinity symbol. Most of these symbols can only be produced in math mode – do not try them in ordinary text.

Mathematics involves other special constructions not found in ordinary text, such as fractions and square roots. LaTeX provides macros to make all of these. The `\frac` command takes two arguments: the numerator of a fraction first, the denominator second. Thus `$\frac{1}{2}$` produces $\frac{1}{2}$. The `\sqrt` command takes only one argument, which it puts under a square root symbol, viz. `$\sqrt{b^2-4ac}$` produces $\sqrt{b^2 - 4ac}$.

When function or operation names are spelled out, as with sines, cosines and other trigonometric functions, or as with the limit symbol, then it is traditional to typeset these in a normal font, so that they look like words, rather than e.g. $c \cdot o \cdot s$. Thus, when we set the expression $\cos x$, we type `$\cos x$`. Just for comparison, note that `$cos x$` produces $cosx$, which probably does not express the mathematics very well. Likewise, the LaTeX macros `\sin`, `\log`, `\exp`, and `\lim` are all defined, as well as many others.

There are many mathematical symbols that stack symbols. We can usually typeset these by using only the subscript and superscript tags. In some cases these are displayed differently depending on whether they use in-line or display mode. For example, consider a simple summation. The sum

$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

looks different in a line of text, viz. $\sum_{n=1}^{\infty} \frac{1}{n^2}$. We typeset this the same way regardless of whether it is displayed or in-line. It looks like

```
$\sum_{n=1}^\infty \frac{1}{n^2}$
```

and if we want it to be displayed, we just put another dollar sign at the beginning and end. Another construction that has this behavior is the limit. An in-line limit looks like $\lim_{x\to0}\frac{\sin x}{x}=1$; in display mode this looks like the following

$$\lim_{x\to0}\frac{\sin x}{x}=1.$$

In both cases, we typeset it as

```
$\lim_{x\to0}\frac{\sin x}{x}=1$
```

where only the number of dollar signs changes.

Sometimes we need English words in lines of mathematics. These are again typed using an ordinary font rather than math italic, so that they are not confused with variables. We use the `\text` command to accomplish that. Be aware that we have to put the spaces for text typesetting in ourselves. For example, to typeset the definition

$$p_n(x) = \frac{x^n}{n!} \text{ for } n = 1, 2, 3, \ldots$$

we can use

```
$$p_n(x) = \frac{x^n}{n!}\ \text{for}\ n=1,2,3,\ldots$$
```

Frequently we want to enclose large expressions or matrices in parentheses or braces. For this we use the `\left` and `\right` commands. These commands both take one argument: the parenthesis, brace, or other object that is to enclose some body of text. They are aware of their partners. In other words, when you use `\left`, you *must* use `\right`. For example, here is a set.

```
$$S = \left\{ 1, 2, \left( \frac{2}{3} \right) \right\}$$,
```

which produces

$$S = \left\{ 1, 2, \left(\frac{2}{3} \right) \right\}.$$

Note that piecewise function definitions and certain other notations require the use of a brace or other delimiter on one side of a collection of mathematical expressions, with no matching brace on the other side. When that happens, use e.g. `\right.` to close the `\left`-`\right` construction. The period does not appear in your document - this closes the construction without making the delimiter.

Sometimes we want to emphasize that two letters are not symbols being multiplied, or we want to have one symbol overlap another. LaTeX provides negative space for this. The negative thin space command is `\!`. Likewise we want just a little extra space between symbols, and LaTeX provides a positive thin space for this: the macro is `\,`. Thin spaces are useful for subtle typesetting effects required for example in integrals:

```
$$\int_0^x \!f(t)\,dt.$$
```

yields

$$\int_0^x f(t)\,dt.$$

3.3.3 Alignment

One of the more problematic issues in typesetting mathematics involves alignment. Often we need to align the equals signs in a set of equations, or we must break a single equation into more that one line. There are many mathematical constructions, such as piecewise defined functions and matrices, that have alignment built into them. Ultimately, all alignments are based on tables, with columns of text that are forced left or right. The basic LaTeX environment for aligning equations is “eqnarray”. This is not the best way to align equations, but we’ll look at it as an example, and for completeness.

```
\begin{eqnarray}
f(x) &=& (x-1)(x+1)\\
\nonumber &=& x^2-1.
\end{eqnarray}
```

This produces the following display:

$$\begin{aligned} f(x) &= (x-1)(x+1) \\ &= x^2-1. \end{aligned} \tag{3.2}$$

We should note several things here

- Think of the eqnarray as a table with three columns. The ampersand characters (&) separate the columns.
- We make a new line with the double-backslash. Thus, we should not put a double-backslash on the last line. The expression above has two lines - if we ended the second with a double-backslash, it would have three lines.
- Every line gets an equation number. If we do not want a separate equation number for a line, we should put the command `\nonumber` on that line.

The amsmath package provides a number of environments to do alignments. These are more versatile than "eqnarray", and they are superior in appearance. In particular the "align" environment provides a better way to align operators for a collection of equations.

```
\begin{align}
f(x) &= (x-1)(x+1)\\
\nonumber &= x^2-1.
\end{align}
```

This renders the same equations as above, but the output looks like this:

$$\begin{aligned} f(x) &= (x-1)(x+1) \\ &= x^2-1. \end{aligned} \tag{3.3}$$

- There is a little bit less space around the equals sign.
- This time the alignment only has two columns. The ampersand character (&) separates the columns.
- The align environment allows an arbitrary number of columns, so you can align more than one set of equations. Odd-numbered columns are right justified, while even-numbered columns are left justified. In another way of looking at it, odd-numbered columns represent the left side of equations, while even-numbered columns represent the right side of equations.
- Every line gets an equation number. If we do not want any equation numbers for a line, we should use the "align*" environment. That suppresses all equation numbers. The `\nonumber` macro works here as well.

The "align" environment starts math mode. Sometimes we need to do a complex alignment in a context that does not involve a matrix, and in which math mode must be started before the alignment occurs. In this case, the "aligned" environment allows us to do the formatting of the "align" environment, but in a math mode that is already in progress. For example, a book concerning linear algebra has an equation that looks like the following:

```
$$
\left.
\begin{aligned}
&(A-\lambda_1I)^3x_1=0,&&(A-\lambda_1I)^2x_4=0,
  &&(A-\lambda_2I)^2x_6=0,&&(A-\lambda_3I)x_8=0\\
&(A-\lambda_1I)^2x_2=0,&&(A-\lambda_1I)x_5=0,
  &&(A-\lambda_2I)x_7=0\\
&(A-\lambda_1I)x_3=0
\end{aligned}
\right\}.
$$
```

This produces

$$\left.\begin{aligned}&(A-\lambda_1I)^3x_1=0,\ &&(A-\lambda_1I)^2x_4=0,\ &&(A-\lambda_2I)^2x_6=0,\ &&(A-\lambda_3I)x_8=0\\&(A-\lambda_1I)^2x_2=0,\ &&(A-\lambda_1I)x_5=0,\ &&(A-\lambda_2I)x_7=0\\&(A-\lambda_1I)x_3=0\end{aligned}\right\}.$$

Note that in this case we are aligning the left edges of equations. Since we want every column of equations to be left justified, we require text to appear only in even-numbered columns. The double-ampersands effectively skip odd-numbered columns.

A matrix is another kind of alignment. There are several ways to do this. The LaTeX way is to use the "array" environment. This does not put in parentheses or brackets - they must be inserted using `\left` and `\right`. Alternatively, if using the amsmath package, there are "pmatrix" and "bmatrix" environments, the first for a matrix encompassed by parentheses, the second for one encompassed by brackets. The columns of the matrix are again separated by ampersands; rows of the matrix are ended by double backslashes. The code

```
$$
A=
\begin{pmatrix}
1 & 2\\
0 & 4
\end{pmatrix}
$$
```

produces the matrix

$$A=\begin{pmatrix}1&2\\0&4\end{pmatrix}.$$

Another kind of alignment that arises often is for a piecewise defined function. This could be handled relatively easily using the "aligned" environment, but the amsmath package provides an even easier way: the "cases" environment. For example, we could define the unit step function using the syntax

```
$$
u(x)=
\begin{cases}
0 & x<0\\
1 & x\ge0.
\end{cases}
$$
```

This produces the result

$$u(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0. \end{cases}$$

3.4 Font modifications

We have already noted that the documentclass statement allows us to specify the size of the default font. This constitutes so-called normalsize. To change the size of fonts inside the document, you must group the text you want to change, and then use a new font size specifier. For example, we can make tiny text. When we type `{\tiny text}` we get text. The size specifiers are: `\tiny`, `\scriptsize`, `\footnotesize`, `\small`, `\normalsize`, `\large`, `\Large`, `\LARGE`, `\huge`, `\Huge`.

Changing font styles and weights is similar. For example typing `{\it italic}` produces *italic* text. The styles include `\it`, `\bf` for bold face, `\tt` for typewriter monospace, and `\sc` for a small-caps style. In more modern times, one typically uses macros for this: `\textit`, `\textbf`, `\texttt`, `\textsc`. Thus, in LaTeX typing `{\sc Small Caps}` produces SMALL CAPS.

If you want to change the default fonts throughout a document, there are a couple of ways to do that. To do this, we need to know a very little bit about the way LaTeX classifies fonts.

LaTeX has three basic font classifications: roman, by which TeX means serif; sans-serif, and teletype, by which TeX means monospace. We can set the default fonts for each of these classifications respectively using e.g. the commands:

```
\renewcommand{\sfdefault}{phv}
\renewcommand{\rmdefault}{ptm}
\renewcommand{\ttdefault}{pcr}
```

These commands tell LaTeX to use a `phv` (a Helvetica) font family for all sans-serif fonts; to use `ptm` (a Times)family for all serif fonts; and to use `pcr` (a Courier) for all monospace fonts. Unfortunately, to use these commands directly requires us to know exactly which font substitutions we want. Even finding this information can sometimes be difficult.

Probably the easiest way to change fonts throughout a document is simply to load a different font package. Somewhere in the preamble you can type the command e.g.

```
\usepackage{times}
```

that includes the name of a font family that is supported. The collection of these packages will vary from one TEX installation to the next. In the very common "texlive" distribution, the families available include avant, charter, pifont, bookman, courier, newcent, times, chancery, helvet, palatino, and utopia. A few of these might not show up properly when you first request their packages. This is often because they do not specify fonts in all three classifications. That might mean that you have to change your choice of default font family. For example, simply loading the helvet package does not automatically change your document default font to Helvetica – you must also tell LATEX that you want to use a sans-serif as your default font family. You can do this using the commands

```
\usepackage{helvet}
\renewcommand{\familydefault}{\sfdefault}
```

3.5 Counting

As we saw in Section 3.3, LATEX counts many things, such as including chapters, sections and other document parts, equations, citations, theorems, and so on. We can manipulate these counters using the `\setcounter` and `\addtocounter` macros. This is important for several reasons. First, it seems obvious that we do not want to have to assign the equation numbers ourselves. If we were forced to number section, theorems, and equations ourselves, we would constantly have to renumber everything as we edited the file. Insert an equation, and then change all the numbers. Change your mind and delete that equation – change all the numbers back. It is very helpful for TEX to do that for us. Second, it is difficult or impossible to remember the numbers of all the equations. It is easier to remember names. Finally, when we refer to equations and theorems by name, then when we insert that new equation, TEX not only renumbers for us, but changes the references. This is a huge time saver, and can prevent embarrassing errors.

TEX always registers the name of whatever counter applies to the current element. We can assign a label to the value of that counter using the `\label` macro. We can then get that value by using the `\ref` macro. To label a counter, all we need to do is to put the `\label` macro in the area where that counter applies. It is safest to do this right after the counter is incremented. Thus we want to label a section right after it starts, we label equations right after the `\begin{equation}`, and so on.

For example, in this section the counter that applies is the section counter, which has the value 3.5.

We place the macro `\label{sec:counting}` in a line after the `\section`, and then refer to it using `\ref{sec:counting}` Note that it is not necessary to put the `\label` command immediately after the `\section` command. There are many places throughout the section where you could put it. However, it is

safe to put it right at the beginning. The name of the label is just something we made up. We could have called it "hubert" if we wanted to, however using the name "sec:counting" gives us a hint that this is a section number, and it is the section about counting. The point is that in a long paper or a book like this, there are hundreds of labels, so it is important to name them well.

In order to refer to values that have been assigned to labels, you should run LaTeX twice: once to get the value into the .aux file, and then again to get it back out for the reference. Many TeX editors do this for you – for example, Texstudio and Texmaker just compile until all the equations are numbered properly.

This works just as well for equations. If we set a `\label` inside a numbered equation, that means the equation counter is the most recently updated, so that the equation number will be assigned to the label. For example, there is a label on equation (3.3). Be sure to put the label on the line that is numbered.

We can always create counters of our own using the `\newcounter` command. This simply defines a counter – to initialize it, use `\setcounter`. From then on, we may increment it using `\stepcounter`. For example, we could define a counter called counttopic, and set its value to 24 using the commands

```
\newcounter{counttopic}
\setcounter{counttopic}{23}
\stepcounter{counttopic}
```

Since the counter is probably not associated with a text element, to get its value we probably need to use the `\arabic` macro. This evaluates the counter and typesets the resulting number using arabic numerals. There are corresponding macros called e.g. `\roman` and `\Roman` to typeset the counter using lower or upper case roman numerals. The current arabic value of the counttopic counter is 24, while in upper case roman numerals it is XXIV. If you prefer to represent the counter using letters, you could use the `\alph` or `\Alph` macros, which currently have values x and X respectively.

There are several environments that produce other counters. In particular, the theorem environment is essential for writing rigorous mathematical papers. In general we can create a structure for numbering theorems and such using the `\newtheorem` macro. In this section we have typed

```
\newtheorem{Thm}{Consequence}[section]
\newtheorem{Def}[Thm]{Fact}
```

This creates two new theorem environments. The first is associated with a name `Thm`, and every instance of it will start with the word "Consequence" typeset in a bold face. The optional `section` argument here indicates that the numbering for the `Thm` counter will appear after chapter and section numbers, with period separators. Likewise, the next line defines another theorem environment whose name is `Def`. Every instance of this will start with the bold face word "Fact". The optional `Thm` argument in this position indicates that this new environment will share the `Thm` counter.

Here is an example.

Fact 3.5.1 *Any offset text in italic style, with a number by which to refer to it, is treated as a theorem by LaTeX.*

Consequence 3.5.2 *Definitions, propositions, theorems, corollaries, lemmas, and many other textual elements can all be treated using theorem environments.*

These are typeset as follows.

```
\begin{Def}
Any offset text in italic style, with a number by
which to refer to it, is treated as a theorem by \LaTeX.
\label{thm:description}
\end{Def}
\begin{Thm}
Definitions, propositions, theorems, corollaries,
lemmas, and many other textual elements can all
be treated using theorem environments.
\label{thm:justification}
\end{Thm}
```

With that we can refer to Fact 3.5.1 using `\ref{thm:description}`, and to Consequence 3.5.2 using `\ref{thm:justification}`.

3.6 Lists

In most markup languages, there are essentially three kinds of lists: ordered, unordered, and description. In LaTeX, these are created using environments called `enumerate`, `itemize`, and `description`, respectively.

An ordered (enumerate) list looks like the following:

1. Enumerate
2. Itemize
3. Description;

while an unordered (itemize) list looks more like:

- Enumerate
- Itemize
- Description.

In both cases, individual items in the list are preceded by the `\item` macro. The numbering happens automatically. Thus, to typeset the ordered list, we use

```
\begin{enumerate}
\item Enumerate
\item Itemize
\item Description;
\end{enumerate}
```

We can nest these lists at will. LaTeX keeps track of all the numbering for us, and even changes the way counters are displayed according to the level of the list. For example, the nested list

```
\begin{enumerate}
\item Enumerate
  \begin{enumerate}
  \item Numbered items
  \item Lettered items
\end{enumerate}
\item Itemize
\begin{itemize}
  \item Bulleted items
  \item Boxed items
\end{itemize}
\item Description
\end{enumerate}
```

displays as

1. Enumerate
    (a) Numbered items
    (b) Lettered items
2. Itemize
    - Bulleted items
    - Boxed items
3. Description

We can gain a greater measure of control over ordered lists by loading the enumerate package:

```
\usepackage{enumerate}
```

This allows us to insert an optional argument for the enumerate environment to act as a template for the numbering format we want. For example

```
\begin{enumerate}[a)]
\item Numbered items
\item Roman numeral items
\item Letter items
\end{enumerate}
```

produces

a) Numbered items

b) Roman numeral items

c) Letter items

while changing the first line of the list code to

```
\begin{enumerate}[I:]
```

produces

I: Numbered items

II: Roman numeral items

III: Letter items

We can see the simplicity of the package. The optional argument will order the list by letters of the alphabet if it contains the letter “a”. If the “A” is capitalized, then the ordering is by upper case letters; otherwise lower case is used. If instead the optional argument contains “i” or “I”, then the list is ordered by lower case or upper case Roman numerals, respectively. Naturally, a “1” in the optional argument produces an ordinary arabic enumeration. Any other characters found in the optional argument appear verbatim in the list.

We see that the last feature provides an easy way to change the “bullets” on an unordered list as well. For example, if we wanted to use a math `\clubsuit` symbol instead of the usual bullets, we could change the first line of the environment to

```
\begin{enumerate}[$\clubsuit$]
```

Note that we must still use the enumerate environment, but the numbers do not show up now. Instead, we see

♣ Numbered items

♣ Roman numeral items

♣ Letter items

There are more powerful and sophisticated packages available for changing the enumeration or bullet style of a list, but these are also somewhat more complicated to use. Look at e.g. the enumitem package if you want something fancier.

The description list uses a slightly different syntax. Each individual item still is preceded by the `\item` macro, but this time it accepts an optional argument giving the term to be described. Thus, the first line below is `\item[First type:] Enumerate`.

First type: Enumerate

Second type: Itemize

Third type: Description

3.7 Boxes, Glue, and Space

It is probably time to discuss the way that LaTeX assembles text into lines and pages, and learn something about how we can change that. This is a somewhat obscure topic, but it is necessary to understand this if we are to make sense of how TeX places things on pages.

TeX fills a page using elementary building blocks, patiently assembled to make larger blocks called words, lines, and then fitting those together those into pages. The smallest building blocks are obviously individual characters and letters. Each character is associated with a box that has certain dimensions. It rests on the *baseline*, and has some height above that line, and a depth below it. It has a width, which is fixed and uniform for monospace fonts such as Courier, but can vary from character to character otherwise. This is illustrated in Figure 3.1. These small letter boxes are pasted together by TeX into word boxes using stretchable space, called *glue*. The words, in turn, are assembled using more glue into lines. The point of allowing the space to stretch is to allow TeX to bump the text up against both the right and left sides of the page – to *justify* the text. The amount of glue TeX uses to assemble a line varies according to the total width of the boxes containing the letters and words. Figure 3.2 illustrates this process.

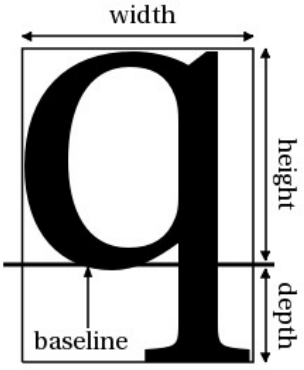


Figure 3.1: A box encompassing a character

Throughout the assembly of a line of text, LaTeX is working only horizontally. Once it has a couple of lines manufactured, it then must fit them together vertically. This is straightforward, but it can be important for us to understand the switch between horizontal and vertical mode. LaTeX always finishes its horizontal mode work before it is willing to think about anything having to do with vertical spacing.

Figure 3.2: Assembly of letter boxes into lines

The simplest way for us to change spacing is to insert one stretchable space. For this we use the `\ ` (backslash-space) macro. The space involved does not stretch much, but it can change size slightly to fill an alloted area. If we need to make a fixed-width, unbreakable space, we use instead the `~` character. Remember that LaTeX ordinarily pays no attention to spaces except insofar as they delimit words and other text objects. TeX always thinks that it is the authority on how best to construct a page of text.

If we need to put a longer horizontal space into a line, we can use a `\quad` or `\qquad`. These names are old printers' terms. A quad is the width of the

current font size - i.e. if the font size is 12pt, then a quad is 12 points. The name is probably a reflection that this is the length of a side of a square in the current font. This is not coincidentally the same as one em – the width of the a capital M. Thus, typing `\quad` produces one quad (one em) of space. A qquad is two quads. If we need different horizontal spaces than this, we can make these manually using the `\hspace{dimen}` command. The `dimen` parameter represents some distance. Thus we can put a one centimeter space between the words “centimeter” and “space” by typing `\hspace{1cm}` between them.

Vertical space is almost as easy to come by. There are several commands to create small vertical spaces: `\smallskip`, `\medskip`, and `\bigskip`. The amounts of space these provide varies with the document class. If you need a different amount, you can use `\vspace`. Thus, to put an extra 1 centimeter of space after this paragraph, we can type `\vspace{1cm}` somewhere near the end of it. Note that vertical space is never inserted until TEX is in vertical mode, i.e. when it gets to the end of a line. Thus, even if we type our `\vspace` command a few words before the end of the paragraph, the space will appear after the line the command is in finishes.

Note that LATEX is quite happy to work with negative space. This can be used to put text in a margin, or to overlap symbols. For example, one way to typeset the word “two” is to type

```
\hspace{2em}o\hspace{-1.3em}w\hspace{-1.1em}t\hspace{1em}.
```

We do not recommend this method, but it illustrates the possibilities.

Since TEX works on boxes from the level of individual letters all the way up to vertical mode lines and paragraphs, it would surprise us if it did not provide a way to create boxes of arbitrary size. Naturally it does. the `\makebox` command creates a horizontal mode box of any width we specify. The `\makebox` command accepts two optional parameters: a width, and a position. The obligatory argument is the text to appear in the box. The position can be “c” for centering the text in the box, “left” to left-justify the text, “r” to right-justify it, or “s” to spread the text throughout the box. Here are some examples.

```
\makebox{a fox in a box}
\makebox[2in][r]{a fox in a box}
\makebox[2in][l]{a fox in a box}
\makebox[2in][s]{a fox in a box}
\makebox[0.4in][s]{a fox in a box}
```

This produces the lines illustrated in Figure 3.3. Notice that when the width of the box is set to be less than the width of the text, then the text is simply allowed to overlap the box.

There is another macro called \framebox that does much the same thing as \makebox. The only difference is that it puts a border around the box. If you want to display text or other single-line content in a frame, use the \framebox macro.

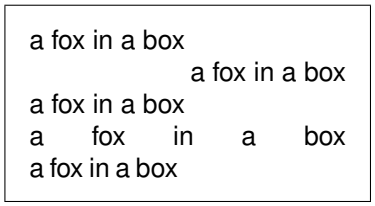


Figure 3.3: The results of various \makebox commands

It is important to note that both \makebox and \framebox operate only in horizontal mode – they can only typeset one line. Occasionally we want to put an entire page area in a box of a specified size, complete with line breaks and justification. LaTeX provides two principal ways of doing that. The \parbox macro takes two arguments: the width of the "paragraph box" to be created, and the content of that box. The minipage environment is another way to create such areas.

The idea for the minipage is simple enough. It is an environment that takes one obligatory argument, which contains the width of the area. The environment itself contains the content to be typeset. In Figure 3.4 the minipage on the left shows the LaTeX code to display the minipage on the right. Since each minipage has a width of less than half the width of the page, they appear side by side. The point is that a minipage is treated as just another horizontal mode element to be lined up with others.

```
\begin{minipage}
{.45\textwidth}
*\hfill**\\
*\hfill*\hfill*\\
\vfill
*\hfill*\hfil*
\end{minipage}
```

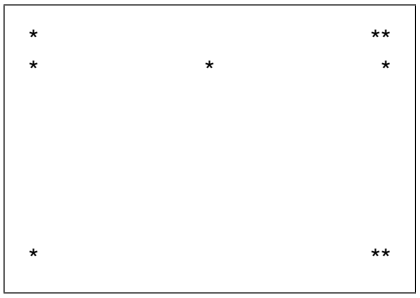


Figure 3.4: Two minipages used to illustrate \hfill and \vfill

This last point is worth emphasizing. The minipage (or parbox) grouped a lot of content into a single box. That box might be fairly tall – indeed the point is to make a box that contains more than one line. On the other hand, that box is treated as a single element for LaTeX to assemble into a line in horizontal mode. Thus, we could put a frame around the minipage using a \framebox. Again, the \framebox command can only work in horizontal mode, but since the minipage has grouped vertical material into a single element that is typeset in horizontal mode, the \framebox accepts it. There are many other horizontal mode tricks that can be applied once you have concentrated your vertical problem into a minipage.

Often in arranging minipages and other content on a page, we need to

separate them by "as much space as is available." We could just guess the amount of space for an \hspace command, look at the result in LaTeX, then change the amount until the result looks right. That would be crazy. It would take too much time and trouble, and the first time we changed anything we would have to start all over with the process. Fortunately, TeX provides several commands that make "as much space as is available." These are \hfill, \hfil, \vfill, and \vfil. The commands whose names contain two "l"s are in some sense stronger than those with only one.

The use of these commands is illustrated in Figure 3.4. We see that the \hfill command simply separates the characters to its left and right by as much space as it can fit on a line. Note that if there is no character to its right, then it does nothing – putting an \hfill at the end of a line would be pointless. The \hfil command does the same thing, with a couple of special rules.

- \hfill can make space up to the entire width of the text; \hfil can only go to half the width of the text.
- If n \hfill commands are found on the same line, they each take $1/n$ of the available space.
- If \hfill and \hfil are found on the same line, \hfill takes all the space, and \hfil gets none.

The \vfill and \vfil commands work on the same principles. Remember that these are vertical mode commands, so they only take effect after horizontal mode is finished, i.e. after a line has been completed.

There are other commands that work in a similar way. The \hrulefill command makes a horizontal rule that expands to fill the space between two characters. The \dotfill command does the same with dots. Note finally that the \rule command also makes horizontal rules, but this command takes width and height arguments. Thus,

```
\rule{.33\textwidth}{0.5pt}
\makebox[.33\textwidth]{\hrulefill}
```

makes two identical horizontal rules, each 1/3 of the page in width.

3.8 Page Layout

In Section 3.7 we used a TeX object called \textwidth in several places without explanation. At the time, we took it to be largely self-explanatory, but it is time to discuss it now. It turns out that this is not a TeX command; instead it is a variable that has a certain value. Initially that value is set by the document class, but we can change it if we like. It is just one of the variables that control the page layout in LaTeX.

Page layout in TeX is not quite as transparent as in wysiwyg word processors. The first thing to note is that TeX does not know how big the page is. It

only knows where the text area starts, and how big it is. For this purpose it maintains several variables whose values can be manipulated. They are

- oddsidemargin – the left margin on odd-numbered pages, measured from a normal of 1 inch. In the `article` document class this is the only margin variable used.
- evensidemargin – gives the left margin on even numbered pages, from a normal of 1 inch. This applies only in certain document classes, e.g. the book class.
- topmargin – the top margin, from a normal of 1 inch. Note that there is also a header area below the top margin.
- headheight – the height of the header.
- headsep – the distance by which the header is separated from the text area.
- textwidth – the width of the text area.
- textheight – the height of the text area.
- footskip – the distance skipped to the footer.
- baselineskip – the distance from one baseline to another. Note that this variable is typically set in the `\begin{document}` statement, so if we want to change it, we usually cannot do that in the preamble.
- baselinestretch – stretches the baselineskip. 1.0 would be the default, 2.0 would give double spacing. This must be set using `\renewcommand`, since it is not a dimension. This is considered to be the preferred way to change the line spacing in LaTeX.
- parskip – the distance between paragraphs.
- parindent – the amount of indentation in the first line of a paragraph.

We can set values in these variables in several ways. Even in plain TeX it was possible to assign values directly: e.g. `\textwidth=6.5in`. This works in LaTeX as well. In LaTeX we have other options. We could replace the equals sign with a space: `\textwidth 6.5in`. Or again, we could use a `\setlength` command: `\setlength{\textwidth}{6.5in}`.

3.9 Figures

LaTeX does not naturally understand anything about graphics. Before we can draw figures or display images, we must load a package that can do that. There are a few packages that handle aspects of the task, but the one almost universally used now is called graphicx. Thus, in the preamble of our document, we type

```
\usepackage{graphicx}
```

Once the graphicx package is loaded, this provides a macro called `\includegraphics` that loads any image file into the document. There are, of course, a few provisos.

- The `pdflatex` program only admits image files. In particular, it cannot use encapsulated postscript.
- The ordinary `latex` program cannot use any file that does not print directly; i.e. it cannot use image files – only encapsulated postscript.
- The filename parameter contains a path to the file we want relative to the directory this document resides in. The easiest way to do this is to have the image file in the same directory as the document, and then just type the file name.
- The file name should not be enclosed in quotes.

Thus, typing

```
\includegraphics{filename}
```

at any point in the document inserts the image contained in `filename` at that point. The image is inserted in horizontal mode, so you might want to put it in its own paragraph, or at least on its own line.

The `\includegraphics` command accepts an optional argument, where we can specify scaling, width, height, or other attributes. It might be that the most practical way to handle the size of the image is to decide how much horizontal space you have for its display, and specify the height accordingly. For example, we might decide that we are willing to allow our image to take up to 50% of the width of the page. In that case we would type

```
\includegraphics[width=.5\textwidth]{filename}
```

This tells TEX to make the image 1/2 of the value of `\textwidth` wide.

Unfortunately, just sticking an image into the document at any old spot is probably not a good plan. It might not fit well, resulting in a large blank space at the bottom of a page, or having the image encroach on the margins. Perhaps we would play around with it until we could make it fit well, but that would take significant time. We might want a caption for the image, which would then need to be aligned in some way. Again, it is traditional in academic papers to number the image and refer to it by number – we should not refer to "the image below." LATEX handles all of these issues by providing floating areas in which we can put images, and building captions with counters into those. The most-used such utility is the figure environment.

The figure environment does not have much to do with figures. It simply creates a floating region that is as wide as the current `\textwidth`. It does permit a `\caption` command that increments the `figure` counter and inserts a caption. Once we have the floating region created, we can insert anything we want into it. Figure 3.5 was inserted using the following code.

Figure 3.5: A scene from the Wind River Range

```
\begin{figure}[t]
\begin{center}
\includegraphics[width=.6\textwidth]{windrivers.jpg}
\caption{\label{fig:winds} A scene from the Wind River Range}
\end{center}
\end{figure}
```

The optional parameter `t` following the the figure environment declaration tells TeX that we want the figure to appear at the top of a page. Note that it might not be the same page where the text above or below falls, but remember, we can refer to the figure using the label in the caption command. It is somewhat important to put the label *inside* the caption – remember that the figure counter is not incremented until we invoke the caption command.

This is not the only way to put a figure into our document. Often we want to place a figure to one side, and have the text flow around it, as we did with HTML. This too is possible if we first load the wrapfig package:

```
\usepackage{wrapfig}
```

Figure 3.6: A view of the Cascade Range

This package defines the wrapfigure environment. Observe the difference in names: the package name is wrapfig, but the environment is wrapfigure. The use of this environment differs slightly from the plain figure. Here we must tell the environment which side of the page to use to display the image, and how wide the area should be. Thus there are two

obligatory arguments to the wrapfigure environment: a letter indicating the position first, then the width of the area. There are several choices for the positioning letter: r, l, i, or o. Obviously "r" stands for right; "l" for left. However, sometimes we want our picture to be on the inside (near the book binding) of a page. That is what "i" specifies. Then "o" tells LaTeX to put the area on the outside of the page. These lower case letters all indicate that the image upper left corner should start exactly where the code appears in the text; using upper case letters would allow the wrapfigure region to float.

The wrapfigure environment often leaves too much vertical space above and below its content. This can be dealt with clumsily but easily by inserting a few `\vspace` commands with negative arguments. The code for inserting Figure 3.6 was as follows.

```
\begin{wrapfigure}{I}{.45\textwidth}
\begin{center}
\includegraphics[width=.40\textwidth]{easypass.jpg}
\caption{\label{fig:easypass} A view of the Cascade Range}
\vspace{-1cm}
\end{center}
\end{wrapfigure}
```

Recall that the `\includegraphics` command inserts images in horizontal mode. Thus, aligning images side by side is quite easy. However, assigning and referring to captions for those individual images can be more problematic. There are several packages that help with this, but most of them are clumsy and fragile. The subcaption package is probably the best of these, though it is not included in some popular TeX distributions. You might have to install it separately.

(a) Wallowa Range

(b) Cascade Range

Naturally, the subcaption package defines an environment with a different name: viz. subfigure. Thus, we produce the individual side-by-side figures by invoking `\begin{subfigure}`. The LaTeX code to produce Figures 3.7a and 3.7b follows.

```
\begin{figure}[h]
\hfill
```

```
\begin{subfigure}{.45\textwidth}
\includegraphics[width=.9\textwidth]{wallowas.jpg}
\caption{Wallowa Range}\label{fig:wallowa}
\end{subfigure}
\hfill
\begin{subfigure}{.45\textwidth}
\includegraphics[width=.9\textwidth]{cascades.jpg}
\caption{Cascade Range}\label{fig:cascade}
\end{subfigure}
\hfill
\end{figure}
```

The subfigure environment requires one argument: the width of the subregion it describes. The subfigure environment makes those subregions using the minipage environment, so that the widths specified in the `\includegraphics` statement are relative to the subregions defined by the subfigure environment.

3.10 Tables

When TEX first appeared, one of the most difficult things to do with it was to make a table. This became one of the most important things driving people to adopt the LATEX package when it appeared – it made table creation much easier. Nonetheless, tables remain something of a sore spot for TEX. There are many packages that strive to improve their appearance and ease of creation, but it is still more troublesome to make good tables in LATEX than in word-processing programs or even HTML.

It is important to understand at the start that the table environment does not produce a table. Instead it is like the figure environment: it creates a floating region in which one can put a table. Indeed, the only differences between the table and figure environments is that they use different counters, and one counter is labeled “Table” while the other is not. The table environment accepts the same optional arguments as the figure environment.

The environment that we use to create actual tables is called “tabular”. This environment takes an obligatory argument that describes the columns of the table and the separators around them. This argument understands only a few specifiers, which are listed in Table 3.1. The idea is simply to place vertical bars wherever the table will have a line separating columns or acting as a border, and then to place l, r, or c characters according to whether columns of text should be left or right justified, or centered.

By default, LATEX does not allow table cells to occupy more than one line. However, if we need cells that contain multiple lines then LATEX allows one format specification that does that. The `p{w}` format specifier makes a paragraph in a single cell. The text is vertically aligned at the top of the cell. The argument `w` is the width of that column. If we need the text to be aligned differently, we

Table 3.1: Table column specifiers

Symbol	Significance
|	a vertical line, used at the edge of the table or as a column separator.
l	left-aligned column
c	centered column
r	right-aligned column
p{w}	a column containing paragraph text with width w, aligned at the top
m{w}	a column containing paragraph text with width w, aligned in the middle. This requires the array package.
b{w}	a column containing paragraph text with width w, aligned at the bottom. This requires the array package.

can load the array package to enable two other macros: `m{w}` aligns the text vertically in the middle of the cell, while `b{w}` aligns it at the bottom.

Table 3.1 lists these format specifiers. The code to generate that table follows.

```
\begin{table}
\begin{center}
\begin{tabular}{||l|p{.5\textwidth}||}
  \hline
  Symbol & Significance\\
  \hline
  $\vert$ & a vertical line, used at the edge of the table or
  as a column separator.\\
  l & left-aligned column\\
  c & centered column\\
  r & right-aligned column\\
  p\{w\} & a column containing paragraph text
  with width w, aligned at the top\\
  m\{w\} & a column containing paragraph text
  with width w, aligned in the middle.  This
  requires the array package.\\
  b\{w\} & a column containing paragraph text
  with width w, aligned at the bottom.  This
  requires the array package.\\
  \hline
\end{tabular}
\caption{\label{tab:tablespec} Table column specifiers}
\end{center}
```

Table 3.2: Study of cold-stressed salmon

	Weights (gm)	
Day of Study	Control	Low Temp.
36	4.2	4.3
0	8.5	4.7
4	8.6	4.9
16	9.4	5.6
26	9.8	6.0
49	10.3	7.1
80	11.2	7.5
208	15.5	12.5

```
\end{table}
```

Note that we must put the horizontal lines in ourselves using the `\hline` command. As with all of our alignments, the character that indicates column separation is the ampersand (&), and we end lines with the double-backslash macro.

Sometimes we need to have one table cell that stretches across more than one column of the table. LaTeX provides the `\multicolumn` command for that. This takes three arguments: namely the number of columns spanned by this cell; the format for this collection of cells; and the content for the cells. Likewise, sometimes we need to have a horizontal line that does not cross all the columns. For this we can use the `\cline` command. The argument here comprises the starting and ending columns, separated by a hyphen. This is illustrated in Table 3.2. The most important code to generate that table follows.

```
\begin{tabular}{|l|c|c|}
\hline
&\multicolumn{2}{c|}{Weights (gm)}\\
\cline{2-3}
Day of Study & Control & Low Temp.\\
\hline
36 & 4.2 & 4.3\\
0 & 8.5 & 4.7\\
      ...
208 & 15.5 & 12.5\\
\hline
\end{tabular}
```

The ability of LaTeX to create tables is tremendously more powerful than we have described here, however, most of that power comes from a large collection of extra packages that have been developed over a long period of time. In particular, the array package adds a number of useful capabilities, some of which we have discussed in Table 3.1. The multirow package adds the ability to make a cell in one column than spans many rows of the table. The tabularx

package redefines and inserts commands to give a wider variety of formatting specifications, and to predefine certain aspects of the format. The xcolor package allows us to specify the color of individual rows or cells of tables. All of these packages are rather specialized, and we will not discuss them further.

3.11 Programming

One of the most powerful aspects of LaTeX lies in our ability to create new macros for specialized tasks. We can save ourselves typing and make complicated environments. While, as usual, there are many packages to enhance this capability, the basic tools are simple: they are the `\newcommand` and `\newenvironment` macros.

One of the simplest uses of `\newcommand` is to save typing. In the same way as we write "LaTeX" using a macro so as to avoid all the extra typing of the different commands required to make that, we can define new commands with short names to avoid typing long, commonly used expressions. For example, in a paper about ponderosa pines, one might would presumably need to type the scientific name of the species, *P. ponderosa*, very often. In LaTeX this is typed as `\textit{p. ponderosa}`. We could instead make a new macro called `\pp` that would typeset that for us.

```
\newcommand{\pp}{\textit{P. ponderosa}}
```

Thereafter, any time we need to refer to the scientific name, we would just type `\pp`. Note that if there is a space after the scientific name, we must put it in using the backslash-space command. This is so that when *P. ponderosa* is followed by a period or comma, those punctuation marks will be right next to the text.

Often we need to use some complicated typesetting commands that apply to different bits of text. If we want e.g. to automate the typesetting commands and apply them to the text we put in, then we must design a macro that accepts arguments. For example, in this text we have needed to typeset HTML commands. This requires us to use a typewriter font, and to use less-than and greater-than signs that TeX cannot typeset as we type them. We define a macro

```
\newcommand{\ht}[1]{\texttt{\textless #1\textgreater}}
```

This tells `\newcommand` that our new macro requires one argument, the value of which is inserted at the point where the #1 appears in the definition. In that case, `\ht{ul}` produces the text `<ul>`. We can use as many arguments as we require, and the definition can extend over multiple lines. As another example, in this text we constantly need to type LaTeX commands. We do so by using a specialized macro called `\tc`. Its definitiion is

```
\newcommand{\tc}[1]{{\texttt{\textbackslash#1}}}
```

Sometimes we need to change the definition of existing commands. This is accomplished using the command \renewcommand. Its syntax is the same as that for \newcommand, except in that the command name is no longer one we make up, but instead is the name of an existing command. We already saw this in action in Section 3.4:

```
\renewcommand{\sfdefault}{phv}
```

This redefined the existing sans-serif default font to one associated with the name "phv". Another place where we saw this in action was in the macro that specifies the space between lines:

```
\renewcommand{\baselinestretch}{2.0}
```

would double the spacing from one line to the next.

In order to change formatting on long bodies of text, we typically need to impose the formatting changes at the beginning of the text, and then undo those changes at the end. In other words, we need to make an environment. Obviously, LaTeX provides a command to let us do that. The following command defines an environment that contains indented text.

```
\newenvironment{inden}
{\hfill\begin{minipage}{.9\textwidth}}
{\end{minipage}}
```

This illustrates most of the critical features of the \newenvironment command. The command takes three arguments: the name of the environment we want to define; the commands that will be used at the start of the environment, and those used at the end. Thus, to invoke the environment defined above, we would type \begin{inden}. When that happens then the new environment starts a minipage, that is only 90% of the page width, with horizontal space to indent it. This is how most new environments we define work: We build them off existing environments, and add few commands to modify those somewhat.

Once again, we can change an existing environment using the \renewenvironment command. It works in the obvious way.

LaTeXHomework

LaTeX Homework Assignment
Due 12/5/14

Your Name Here

November 24, 2014

Typeset the following lines.

1 First group of exercises

1. $v = (v_1, v_2, v_3)^t$
2. $f_n(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$
3. $g(x) = \int_0^\infty G(t, x)dt$
4. $\sum_{i=0}^{\infty} x^i$
5. $$\sum_{i=0}^{\infty} x^i$$
6. We say that λ is an eigenvalue of a matrix A corresponding to eigenvector u if $Au = \lambda u$. For example,

   $$\begin{bmatrix} -1 & \frac{1}{3} \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = (-1) \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

   so -1 is an eigenvalue of the matrix on the left corresponding to eigenvector $(1, 0)^T$.

Fruit Prices	
apples	$.50
peaches	$1.25

2 Second group of exercises

Here are some more ice exercises for TEX. Try a piecewise defined function:

$$f(x) = \begin{cases} x^2 & x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Here is an example using Greek letters.

$$\Lambda_N = \sum_{n=0}^{N} \frac{1}{\lambda^n}$$

For the following,

$$\sum_{i=1}^{2^N-1} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{2^N - 1} \tag{1}$$

$$> \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \overbrace{\frac{1}{8} + \cdots + \frac{1}{8}}^{4 \text{ times}} + \cdots + \frac{1}{2^N}, \quad \text{for } N > 3 \tag{2}$$

note lines (1) and (2). You must use `\label{} and \eqref{}` for that last line.

Part 4: Projects

Project: Mandelbrot Set

Dr. David Koslicki
Math 399
Fall 2014

September 29, 2014

1 Introduction

This project consists of developing code to draw and explore a certain kind of fractal called a *Mandelbrot set*.

2 Background

2.1 Mandelbrot Set

Benoit Mandelbrot was a Polish born, French/American mathematician who spent most of his career at the IBM research center in New York. He is credited for coining the term "fractal" and developed a theory of "roughness"/"self-similarity"/"fractals"/"chaos". His book *The Fractal Geometry of Nature* **??** was published in 1982 and was quite influential at the time (this was when computer graphics were just becoming widely available). One of his inventions/discoveries, the *Mandelbrot set* has stimulated significant mathematical research and is popularly used to generate interesting visual images (and also test the limits of hardware!).

To understand the Mandelbrot set, let's first consider what happens when we take a region in the complex plane, and then repeatedly square each point

$$z_{k+1} = z_k^2, \quad k = 0, 1, \ldots$$

For which initial values z_0 does the sequence $\{z_k\}_{k\geq 0}$ remain bounded as $k \to \infty$? After a little thought, one can see that this set is simply the unit disc, $|z_0| \leq 1$. If $|z_0| > 1$, then the sequence is unbounded.

The definition of the Mandelbrot set is only slightly more complicated: The Mandelbrot set is the region in the complex plane consisting of those values z_0 such that the sequence $\{z_k\}_{k\geq 0}$ remains bounded for

$$z_{k+1} = z_k^2 + z_0, \quad k = 0, 1, \ldots.$$

Even with such a simple definition like this, a fantastic amount of complexity can be generated.

3 Project Components

1. Define a function `Mandelbrot(xlim,ylim, maxIterations, gridSize)` that plots the Mandelbrot set (as defined above). As we can't let $k \to \infty$ in finite time, you will need to calculate

$$z_{k+1} = z_k^2 + z_0, \quad k = 0, 1, \ldots, \texttt{maxIterations}$$

   The `gridSize` will determine how many points to plot (for example, if `xlim=[-1,1]`, then you might use `linspace(xlim(1), xlim(2), gridSize)` to define the real parts of z_0. To color your plot, count the number of times the iteration $z_{k+1} = z_k^2 + z_0$ must be applied until $|z_k| > 1$.

2. Explore a few different regions of the Mandelbrot set by using different values for `xlim` and `ylim`. For example, try

$$xlim = [-0.748766713922161, -0.748766707771757]$$
$$ylim = [0.123640844894862, 0.123640851045266]$$

   or

$$xlim = [-1, 1]$$
$$ylim = [-1, 1]$$

   Plot a few of these and include them in your report.

3. Generalize the definition by implementing analogous functions for the following Mandelbrot-like sets:

$$\left\{ z_0 \in \mathbb{C} : \forall k \geq 0,\ |z_k| < \infty, z_{k+1} = z_k^3 + z_0 \right\}$$

   and

$$\left\{ z_0 \in \mathbb{C} : \forall k \geq 0,\ |z_k| < \infty, z_{k+1} = \sin\left(\frac{z_k}{z_0}\right) \right\}$$

References

[1] Mandelbrot, B. (1982) The Fractal Geometry of Nature. W.H. Freeman and Company, 468p.

Project: Substitutions and Fractals

Dr. David Koslicki
Math 399
Fall 2014

September 17, 2014

1 Introduction

This project consists of developing code that will generate fractal-like images from a substitution on a four letter alphabet. We shall use the method of Dekking [1] to obtain planar curves from an arbitrary substitution. First, I will describe the general process of obtaining these curves, and then I will include directions on how to complete this project.

2 Background

2.1 Substitutions

Let $\mathcal{A} = \{a, b, c, d\}$ be a set of 4 different arbitrary letters. Define a word, w, as a finite concatenation of the elements of $\mathcal{A}$. Let W^n denote the set of all n-length words formed from the elements of $\mathcal{A}$.

For example:

$$W^2 = \{aa, ab, ac, ad, ba, bb, bc, bd, ca, cb, cc, cd, da, db, dc, dd\}$$

Now let $W^* = \bigcup_{n\geq 0} w^n$ be the set of all finite n-length words over $\mathcal{A}$. We can now define a substitution σ, as a mapping $\sigma : W^* \to W^*$, such that for $w = w_1 w_2 \dots w_i$,

$$\sigma(w_1 w_2 \dots w_i) = \sigma(w_1)\sigma(w_2)\dots\sigma(w_i).$$

For example, let

$$\sigma_{\text{eg}} : \begin{array}{c} a \to ab \\ b \to bc \\ c \to cd \\ d \to da \end{array}$$

We will use σ^n to denote the n^{th} application of σ. Then we have the following

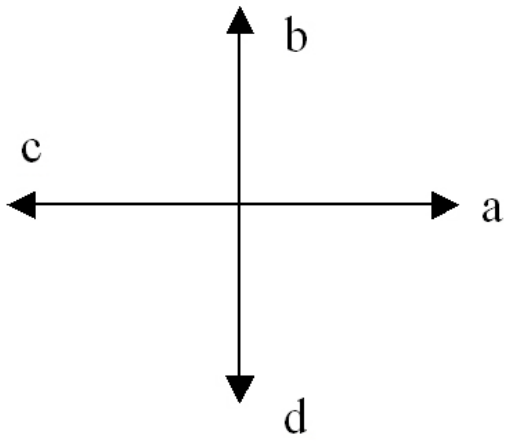
$$\sigma_{\text{eg}}(a) = ab$$
$$\sigma_{\text{eg}}^2(a) = \sigma_{\text{eg}}(ab) = abbc$$
$$\sigma_{\text{eg}}^3(a) = \sigma_{\text{eg}}(abbc) = abbcbccd$$
$$\sigma_{\text{eg}}^4(a) = \sigma_{\text{eg}}(abbcbccd) = abbcbccdbccdcdda$$

If for a particular letter $a \in \mathcal{A}$, if $\sigma(a)$ begins with a, then for each n, $\sigma^n(a)$ will be a subword of $\sigma^{n+1}(a)$. Hence repeat application of σ to a will result in a limiting word (or fixed point) $\sigma^\infty(a) = \lim_{n\to\infty} \sigma^n(a)$. Throughout the following, we assume that $\sigma(a) = a\ldots$.

Note that the given example has the property such that for each $l \in \mathcal{A}$, $|\sigma(l)| = 2$. This is called a *constant length* substitution. You need not restrict yourself to this case.

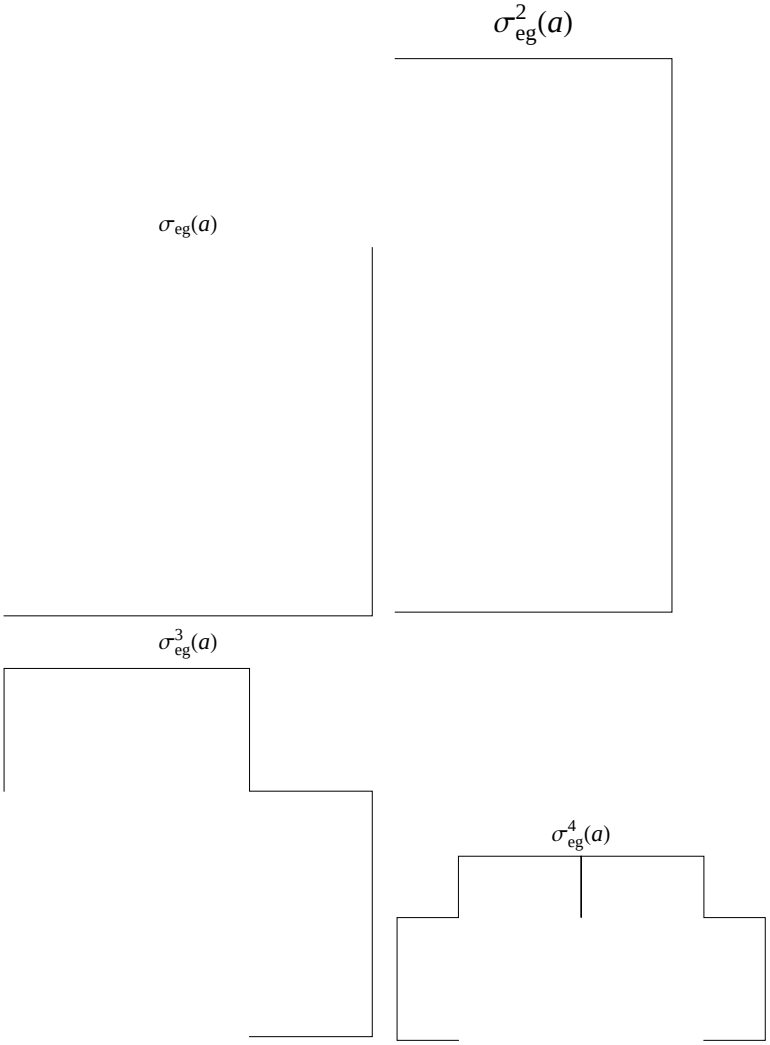
2.2 Curves

To obtain a curve, we associate a vector of unit length with each element of $\mathcal{A}$. Here, a represents the vector $(1, 0)$, b represents the vector $(0, 1)$, c represents the vector $(-1, 0)$, and d represents the vector $(0, -1)$. Pictorially, we can think of the elements of $\mathcal{A}$ as being parts of the the coordinate unit axis:



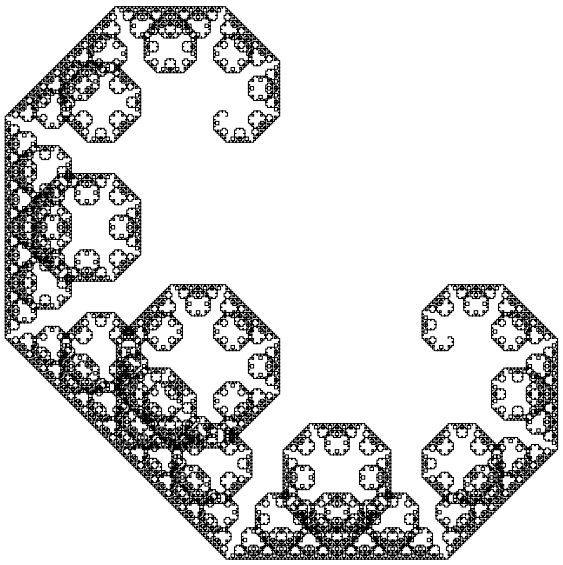
Given a word $w = \sigma^n(a)$ (formed by repeat application of the substitution σ), we obtain a curve by interpreting the word w as a set of "directions": reading left to right, placing the vectors represented by the letters of w head to tail.

In our example, the first through fourth iterations of σ_{eg} would appear as follows (rescaling as necessary):



You can begin to see fractal-like images result at higher successive applications of σ. Here we begin to see the Levy Dragon begin to form:

$\sigma_{\text{eg}}^{15}(a)$



3 Project Components

To complete this project, use either Matlab or Mathematica to perform the following tasks. The deliverable will be either a Matlab script or Mathematica notebook containing the solutions to each one of the following steps.

1. Create a function
   `Substitution(sigma, letter, iterations)`
   that takes in a substitution $\sigma =$`sigma`, a starting letter $a =$`letter`, and number of iterations $n =$`iterations` and returns the word $w = \sigma^n(a)$. An example of calling this function in Mathematica would be
   `Substitution[{"a"->"ab","b"->"bc","c"->"cd","d"->"da"},"a",12]`.
   Instead of using strings in Matlab, you might want to use the convention that $a = 1, b = 2, c = 3$, and $d = 4$ and use the basis $\mathcal{A} = \{a, b, c, d\}$ to represent the rule. Hence, calling this function in Matlab might look like `Substitution({[1 2], [2 3], [3 4], [4 1]}, [1], 12)`.

2. Create a function
   `PlotWord(word)`
   that will generate a curve from the word $w =$`word` just like in section 2.2.

3. Create a function
   `PlotSubstitution(sigma, letter, iterations)`

that will generate a curve (as in section 2.2) obtained from the word $w = \sigma^n(a)$.

4. Find 3 "interesting" and 3 "non-interesting" substitution curves by varying the rules defining σ. Use your own definition of "interesting" (though please be reasonable). Try to find if the "interesting" curves have been seen before (hint: Google image search), if not, congrats! You found a new fractal!

5. **Bonus:** The concept of a random substitution is defined in [2]. Briefly, a random substitution is just like a substitution as defined above in section 2.1, except at each letter, one of a finite set of replacement rules are chosen at random. An example of such a random substitution is given by

$$\Sigma_{\text{eg}} : \begin{cases} a \to \begin{cases} ab \text{ with probability } 1/4 \\ ba \text{ with probability } 3/4 \end{cases} \\ \\ b \to b \text{ with probability } 1 \end{cases}$$

So now $\Sigma_{\text{eg}}(a)$ no longer represents a single word, but rather is a distribution over words. Here, $\Sigma_{\text{eg}}(a)$ is equal to ab with probability $1/4$ and is equal to ba with probability $3/4$. Furthermore, $\Sigma^2_{\text{eg}}(a)$ is equal to abb with probability $1/16$, bab with probability $3/16$, bab with probability $3/16$, and bba with probability $9/16$.
Write a function
`RandSub[rules, probabilities, letter, iterations]`
that implements this. For example,
`RandSub[{"a"->{"ab","ba"},"b"->{"b"}},{{1/4,3/4},{1}},"a",2]`
should return aab $1/16^{\text{th}}$ of the time, bab $3/16^{\text{th}}$ of the time, etc.
Use the function
`PlotSubstitution()` that you defined above and plot a few curves generated by random substitutions. How do these curves differ from the deterministic substitutions defined before?

References

[1] Dekking, F.M. Substitutions, branching processes and fractal sets. Proceedings of the NATO ASI on Fractal Geometry and Analysis, Montréal, July, 1989, (Eds.: J. Bélair, S. Dubuc) Kluwer Acad. Publ. Dordrecht, Boston, London, 1991, *Fractal Geometry and Analysis* 99-119

[2] Koslicki, D. Substitution Markov chains with applications to molecular evolution. PhD thesis, Pennsylvania State University, 2012.

Project: Entropy

Dr. David Koslicki
Math 399
Fall 2014

September 17, 2014

1 Introduction

This project consists of developing code that computes metric entropy as well as a finite implementation of topological entropy. This will then be used to calculate entropy of a set of DNA sequences.

2 Background

2.1 Entropy

Entropy is a measure of information content and complexity first introduced by Claude Shannon in 1948 [3]. Since then, the mathematical concept of entropy has been very useful in a variety of disciplines, assisting fields as diverse as cryptography, bioinformatics, engineering, etc. We will focus here on entropy of strings of letters.

Given an alphabet $\mathcal{A}$ Define a word/string, w, as a finite concatenation of the elements of $\mathcal{A}$. Let $\mathcal{A}^n$ denote the set of all n-length words formed from the elements of $\mathcal{A}$. Let $\mathcal{A}^* = \bigcup_{n\geq 0} \mathcal{A}^n$ and $\mathcal{A}^\infty$ be the set of all infinite length words formed from $\mathcal{A}$. Let $|w|$ denote the length of the string w. For concreteness we will let $\mathcal{A} = \{A, C, G, T\}$.

The original definition of entropy given by Claude Shannon is referred to as *metric* entropy as it depends on a metric (that is, the probability of occurrence of letters in the alphabet $\mathcal{A}$). More precisely, given a word w over $\mathcal{A} = \{A, C, G, T\}$, if the probability of occurrence of A in w is P_A, the probability of occurrence of C in w is P_C, etc. then the metric entropy of w is given by

$$H(w) = -\left(P_A \log\left(P_A\right) + P_C \log\left(P_C\right) + P_T \log\left(P_T\right) + P_G \log\left(P_G\right)\right).$$

Here log is the base e logarithm (i.e. the natural logarithm).

In general,

Definition 2.1 *For $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ and P_{a_i} the probability of occurrence in $w \in \mathcal{A}^*$ of a_i, then metric entropy of w is given by*

$$H(w) = -\sum_{i=1}^{n} P_{a_i} \log\left(P_{a_i}\right)$$

For example, if $w = AAAACCCGGT$, then $|w| = 10$, $P_A = 4/10$, $P_C = 3/10$, $P_G = 2/10$, and $P_T = 1/10$. So then

$$H(w) = -\left(\frac{4}{10}\log\left(\frac{4}{10}\right) + \frac{3}{10}\log\left(\frac{3}{10}\right) + \frac{2}{10}\log\left(\frac{2}{10}\right) + \frac{1}{10}\log\left(\frac{1}{10}\right)\right) = 1.27985$$

3 Project Component 1

1. Define a function `MetricEntropy(w, alphabetSize)` that computes the metric entropy of w if it is assumed that the alphabet has size `alphabetSize` (that is $|\mathcal{A}|$ =`alphabetSize`). You must code this function for yourself and cannot use a built-in "Entropy" function.

2. Download the following file
   `http://www.math.oregonstate.edu/~koslickd/DNASequences.fasta`
   Read this file into Matlab using `fastaread()` or Mathematica using `Import[]`.

3. Create a histogram of the lengths of the DNA sequences.

4. Calculate the metric entropy for each sequence in the file. Create a histogram of the metric entropy values. Select the sequences that have metric entropy less than 1.2 (that is, the bottom 1.43% of the entropies) and save them to a file.

4 Topological Entropy

One of the issues with metric entropy is that it is difficult to compare entropy values for sequences of different lengths (since for shorter sequences, the probabilities of occurrence of subletters is prone to larger deviations). One alternate way to characterize complexity is to use *topological entropy*. Topological entropy was first defined in [1] to characterize dynamical systems resulting from continuous mappings. This definition of topological entropy is appropriate for *infinite* length sequences, so a modified definition adapted for finite sequences was given in [2]. Before giving that definition, we need to define the complexity function

Definition 4.1 *For a given sequence w, the complexity function $p_w : \mathbb{N} \to \mathbb{N}$ is defined as the number of different n-length subwords (with overlaps) that appear in w. That is,*

$$p_w(n) = |\{u : |u| = n \text{ and } u \text{ appears as a subword of } w\}|.$$

Definition 4.2 *Let w be a finite sequence of length $|w|$ on the alphabet $\mathcal{A}$, let n be the unique integer such that*

$$4^n + n - 1 \leq |w| < 4^{n+1} + (n-1) - 1$$

Then for $w_1^{4^n+n-1}$ the first $4^n + n - 1$ letters of w, the topological entropy of w is defined as

$$H_{top}(w) = \frac{\log_4\left(p_{w_1^{4^n+n1}}(n)\right)}{n}.$$

Here, the base 4 logarithm is being used.

As an example, if $w = AAAAACCCCCTTTTTGGGGG$, then $|w| = 20$, and $17 = 4^2 + 2 - 1 \leq |w| < 4^3 + 3 - 1 = 66$ so $n = 2$. Now all the length 2 subwords of w are $\{AA, AC, CC, CT, TT, TG, GG\}$ so $p_w(2) = 7$. Hence $H_{top}(w) = \frac{\log_4(7)}{2} = 0.2432$.

As defined, $H_{top}(w)$ is always between 0 and 1, with low complexity sequences having entropy values closer to 0 (as well as conversely).

One advantage of topological entropy over metric entropy is that with topological entropy, entropy values of different length sequences become comparable.

5 Project component 2

1. Definie a function `ComplexityFunction(w,n)` that computes the complexity function given in definition 4.1 on the sequence `w` for the subword length `n`.

2. Define a function `TopologicalEntropy(w)` that computes the topological entropy of w.

3. Download the following file
   `http://www.math.oregonstate.edu/~koslickd/DNASequences.fasta`
   Read this file into Matlab using `fastaread()` or Mathematica using `Import[]`.

4. Calculate the topological entropy for each sequence in the file. Create a histogram of the topological entropy values. Select the sequences that have topological entropy less than .75 (that is, the bottom 1.44% of the entropies) and save them to a file.

5. Compare and contrast the histograms of metric entropies and topological entropies. Consider the low metric entropy and low topological entropy sequences that you saved; what observations can you make about what it means for a sequence to have low metric/topological entropy?

References

[1] Adler, R.L., Allan, G., Mcandrew, M.H. (1965) Topological entropy. Trans. of the Amer. Math. Soc. **114** (2), 209–319.

[2] Koslicki, D. (2011) Topological entropy of DNA sequences. Bioinformatics **27** (8), 1061–1067.

[3] Shannon, C.E. (1948) A mathematical theory of communication. Bell Sys. Tech. J. **27** (3), 379–423.

Project: Mandelbrot Set

Dr. David Koslicki
Math 399
Fall 2014

September 29, 2014

1 Introduction

For this project, I used Matlab [1] to create code that would draw Mandelbrot sets.

1.1 Part 1: Code

The Mandelbrot set is defined as the set of complex numbers $\{z_0\}$ such that the sequence $\{z_k\}_{k\geq 0}$ stays bounded for $k = 0, 1, \ldots$ for $z_{k+1} = z_k^2 + z_0$. To implement this, I defined a Matlab function

`Mandelbrot(xlim,ylim, maxIterations, gridSize)`

that will return a square matrix A with dimensions `gridSize` where the entry $A_{i,j}$ gives the number of iterations of the recursion $z_{k+1} = z_k^2 + z_0$ required before $|z_k| > 1$. The code for this function is copied below in its entirety:

1.2 Part 2: Plots

In figure 1 is a plot of the of the function

`Mandelbrot([-1.5,.5], [-1,1], 750, 1500).`

In figure 2 is a plot of the of the function:

```
Mandelbrot([-0.748766713922161, -0.748766707771757],...
  [0.123640844894862, 0.123640851045266], 750, 1500).
```

1.3 Part 3: Other Mandelbrot sets

Here, we develop code to implement the following two Mandelbrot-like sets:

$$\left\{z_0 \in \mathbb{C} : \forall k \geq 0,\ |z_k| < \infty, z_{k+1} = z_k^3 + z_0\right\}$$

and

$$\left\{z_0 \in \mathbb{C} : \forall k \geq 0,\ |z_k| < \infty, z_{k+1} = \sin\left(\frac{z_k}{z_0}\right)\right\}$$

To implement the first, we used Matlab to develop the function

`MandelbrotCubed(xlim,ylim, maxIterations, gridSize).`

The complete source code is included below:

To implement the second, we used Matlab to develop the function

`MandelbrotSin(xlim,ylim, maxIterations, gridSize).`

The complete source code is included below:

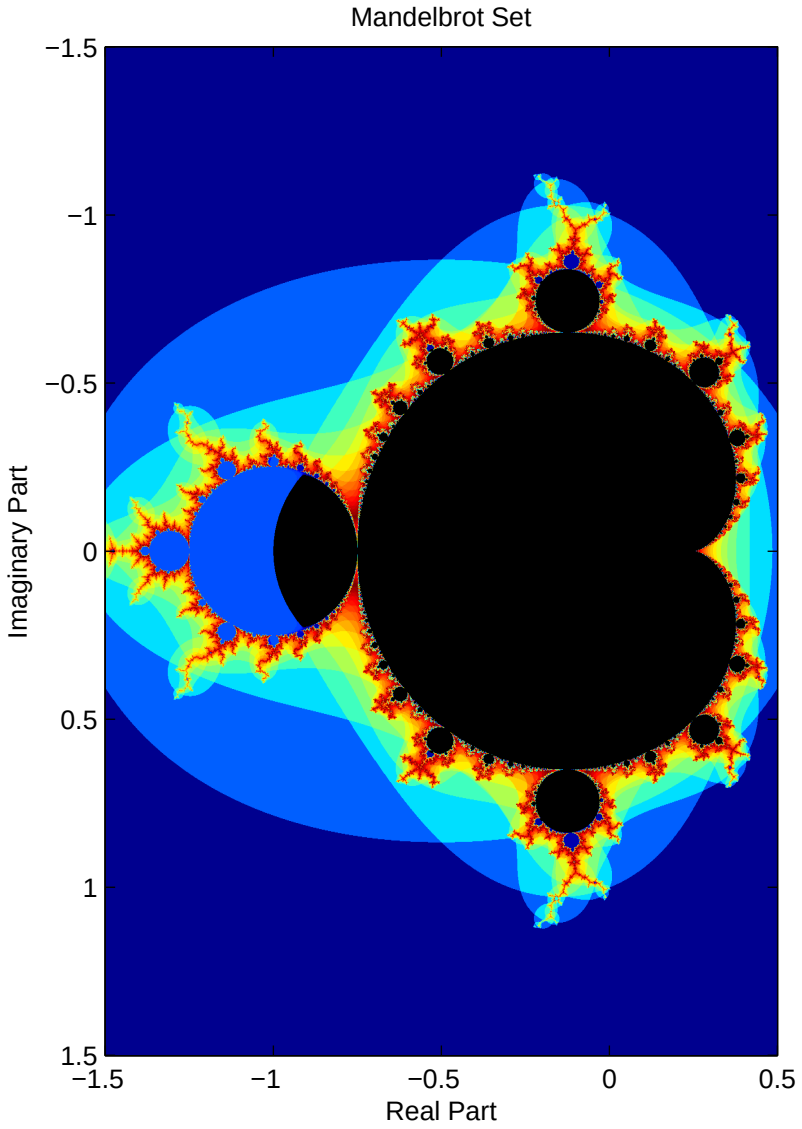


Figure 1: Mandelbrot plot 1.

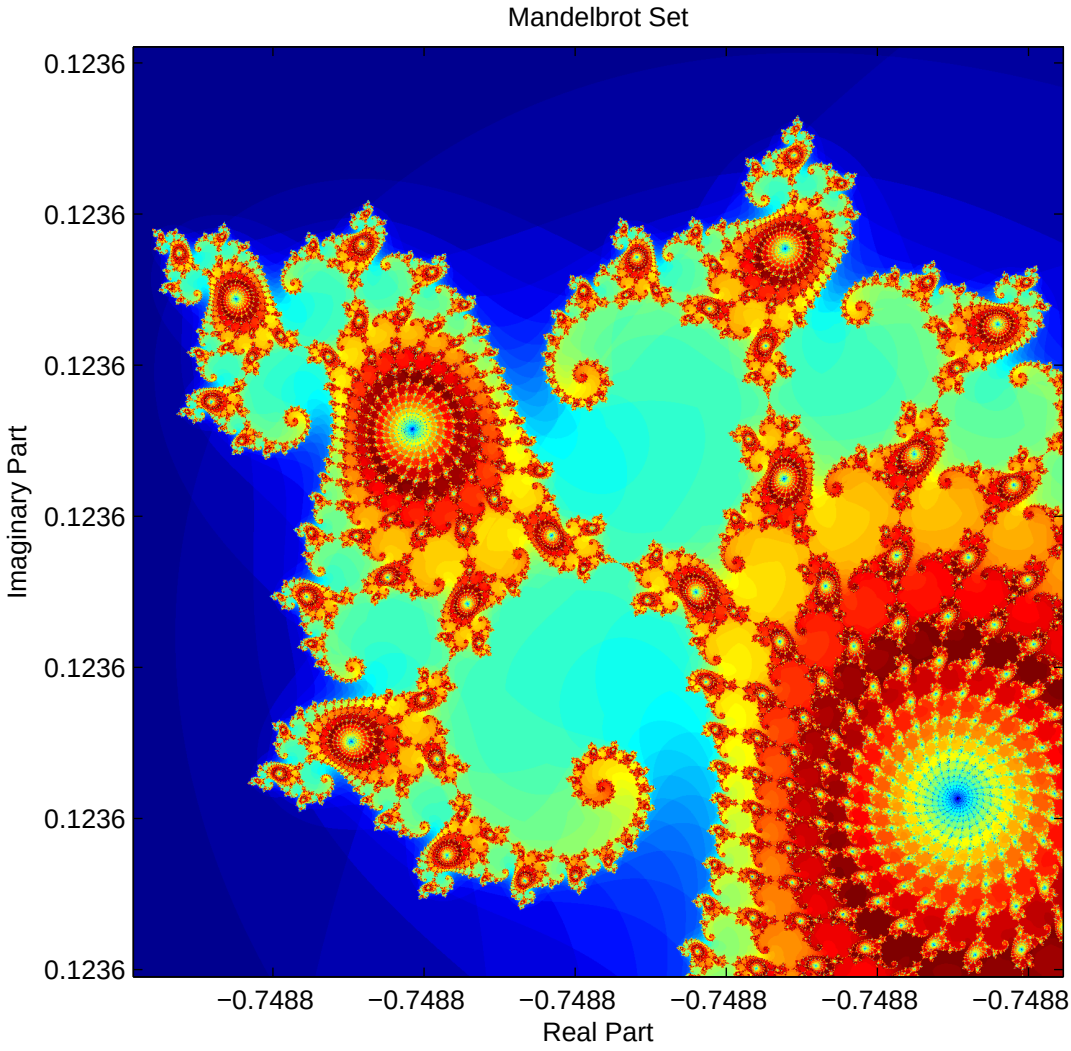


Figure 2: Mandelbrot plot 2.

```
function [x,y,count] = MandelbrotCubed(x1x2, y1y2, maxIterations, gridRes)
% This function will return  the x values, the y values, and a square
% matrix of size gridRes where the i,jth entry is the number of iterations
% required for the recurrence z_{k+1} = z_k^3 + c0 to leave the unit circle.
x = linspace( x1x2(1), x1x2(2), gridRes );% Makes the x values
y = linspace( y1y2(1), y1y2(2), gridRes );% Makes the y values
[xGrid,yGrid] = meshgrid( x, y );% Where is the mesh of complex values
c0 = xGrid + 1i*yGrid;% Whose will be the input points
count = ones( size(c0) );% Where keeps track of how many iterations until it
%leaves the unit circle
% Calculate
z = c0;
for n = 0:maxIterations
    z = z.^3 + c0;
    inside = abs( z )<2;% What if outside the unit circle
    count = count + inside;% Increment the count
end
count = log( count );% Makes the log just to make everything look pretty.
```

In figure 3 we include a plot of the of the function:

`MandelbrotCubed([-1, 1],[1, 1], 750, 1500).`

In figure 4 we include a plot of the of the function:

`MandelbrotSin([-1, 1],[1, 1], 750, 1500).`

References

[1] Matlab 2013a, The MathWorks, Inc., Natick, Massachusetts, United States.

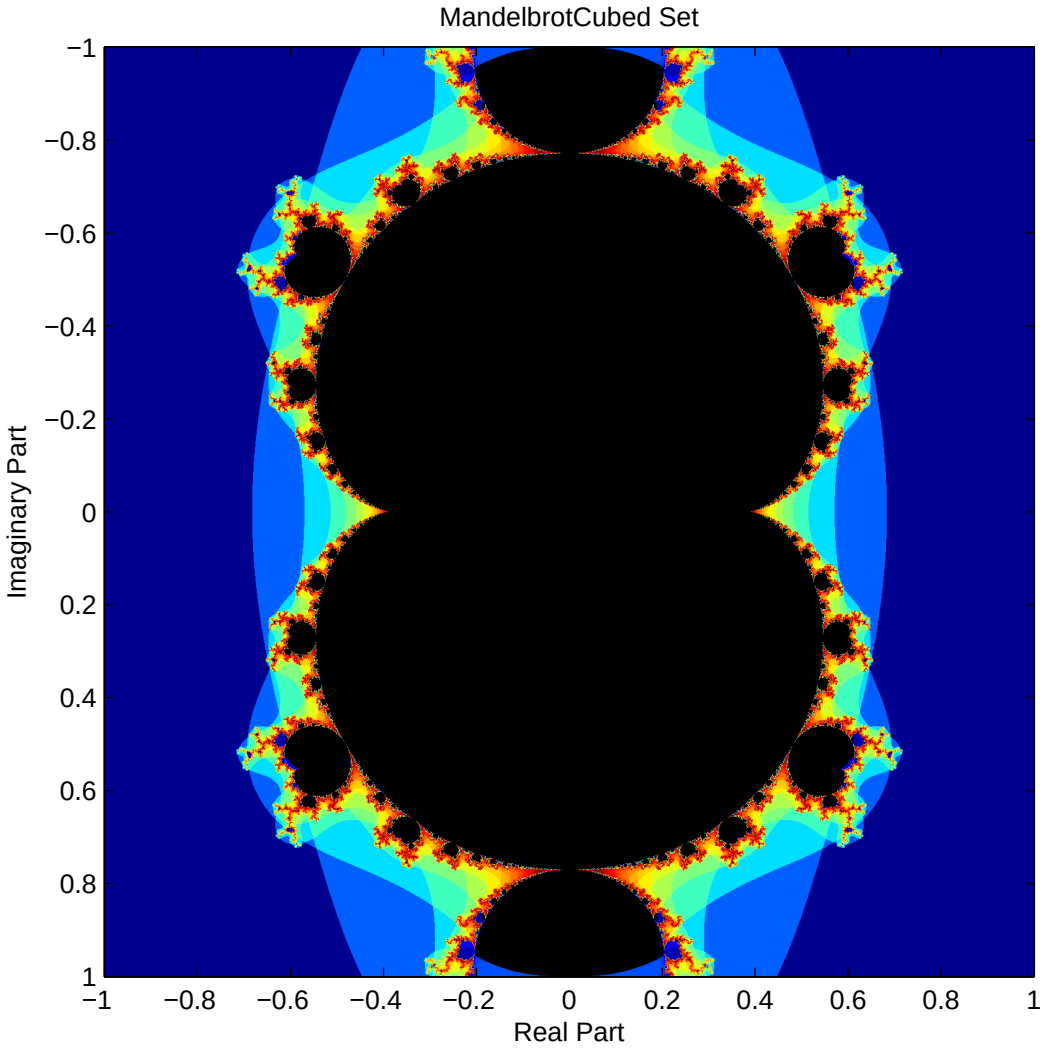


Figure 3: MandelbrotCubed plot.

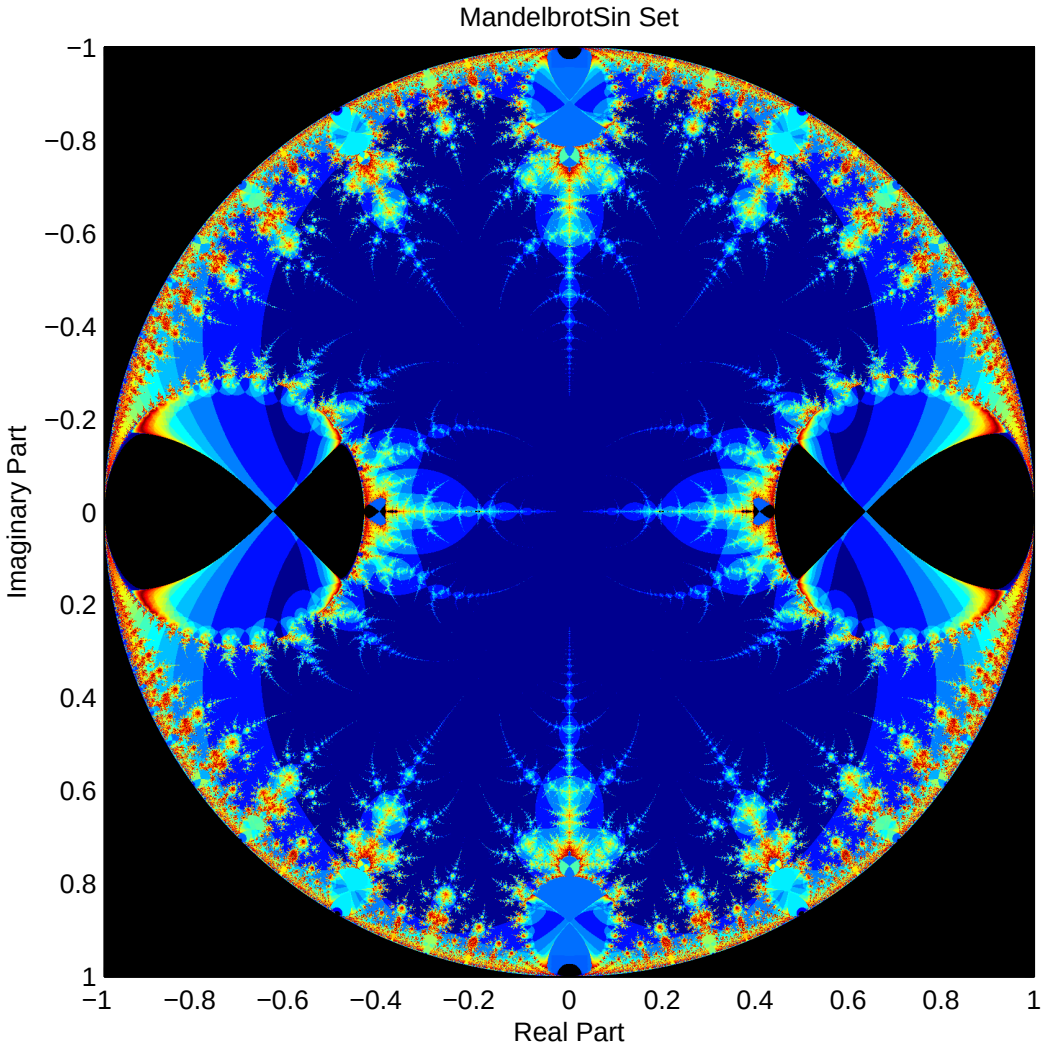


Figure 4: MandelbrotCubed plot.