# MATLAB Assignment #1:  Introduction to MATLAB       Due with HW #1

This guide is intended to help you start, set up and understand the formatting of MATLAB before beginning to code.  It is based on the first lab used in 380 and will give you a good introduction to the powerful MATLAB software. Some changes are made with an emphasis at the end of generating random numbers.
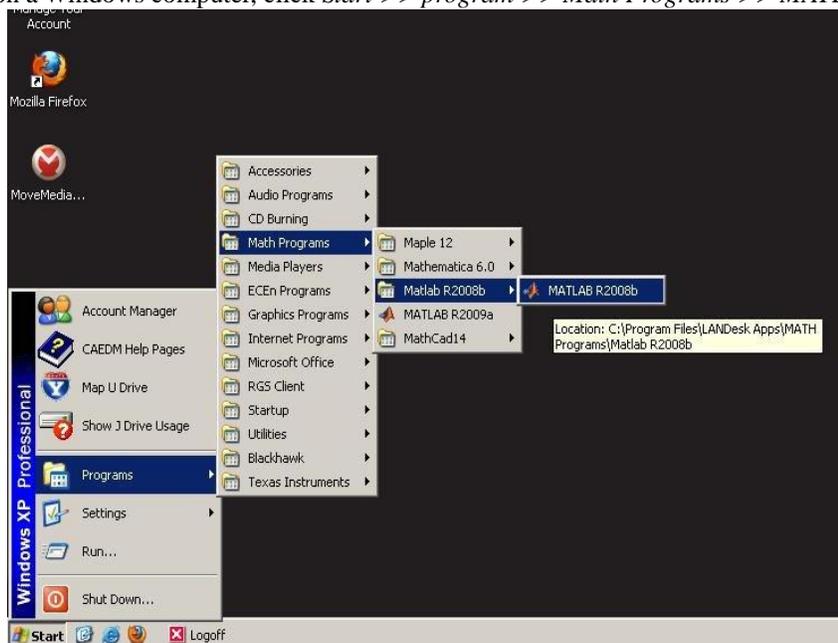
## Finding MATLAB on CAEDM Computers

### Unix Computer
To open MATLAB on a Unix computer, click on *K-Menu >> Caedm Local Apps >> MATLAB*.
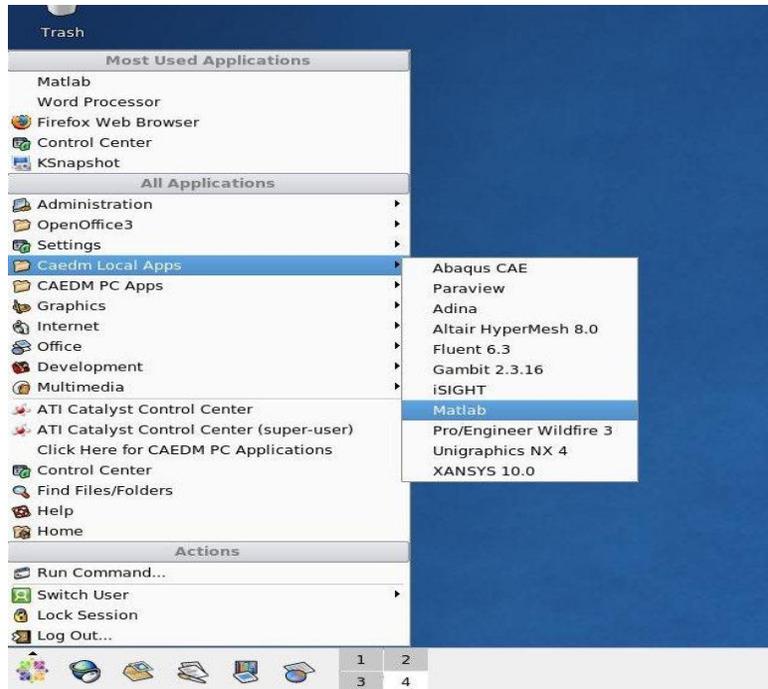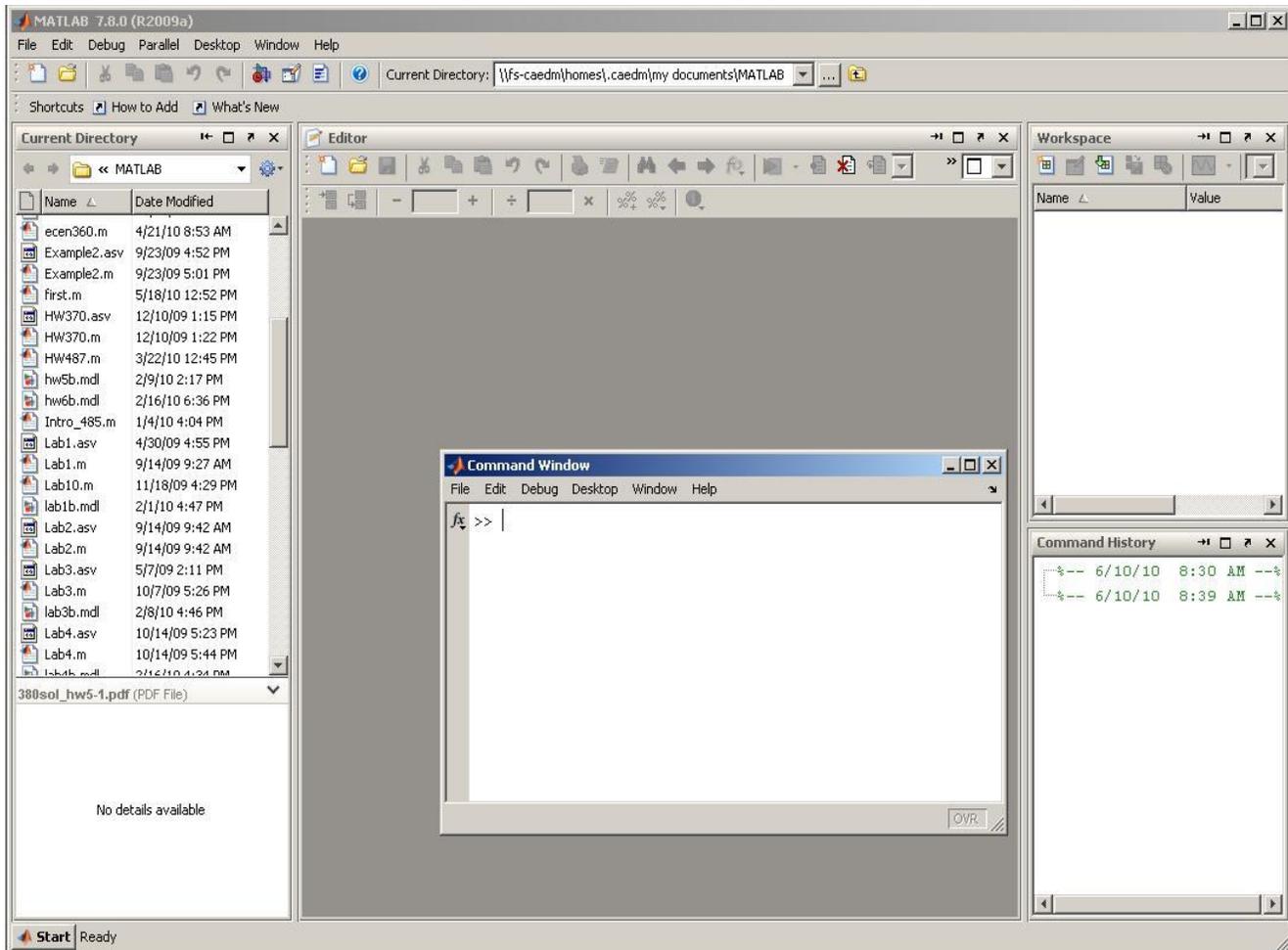


### Windows Computer
To open MATLAB on a Windows computer, click *Start >> program >> Math Programs >> MATLAB R2009a*.

# Starting MATLAB

When you first open MATLAB, it should look something like this:



Notice that there is a main program and also a smaller, separate command window. You will be using both to program.

It is probably a good idea to dock the smaller console onto the main program so you don't have to keep switching back and forth. This can be done by clicking on the *Desktop* menu option of the smaller command window and then clicking the only option: *Dock Command Window or clicking the arrow on the command window shown below*.

Your screen should now look something like this:
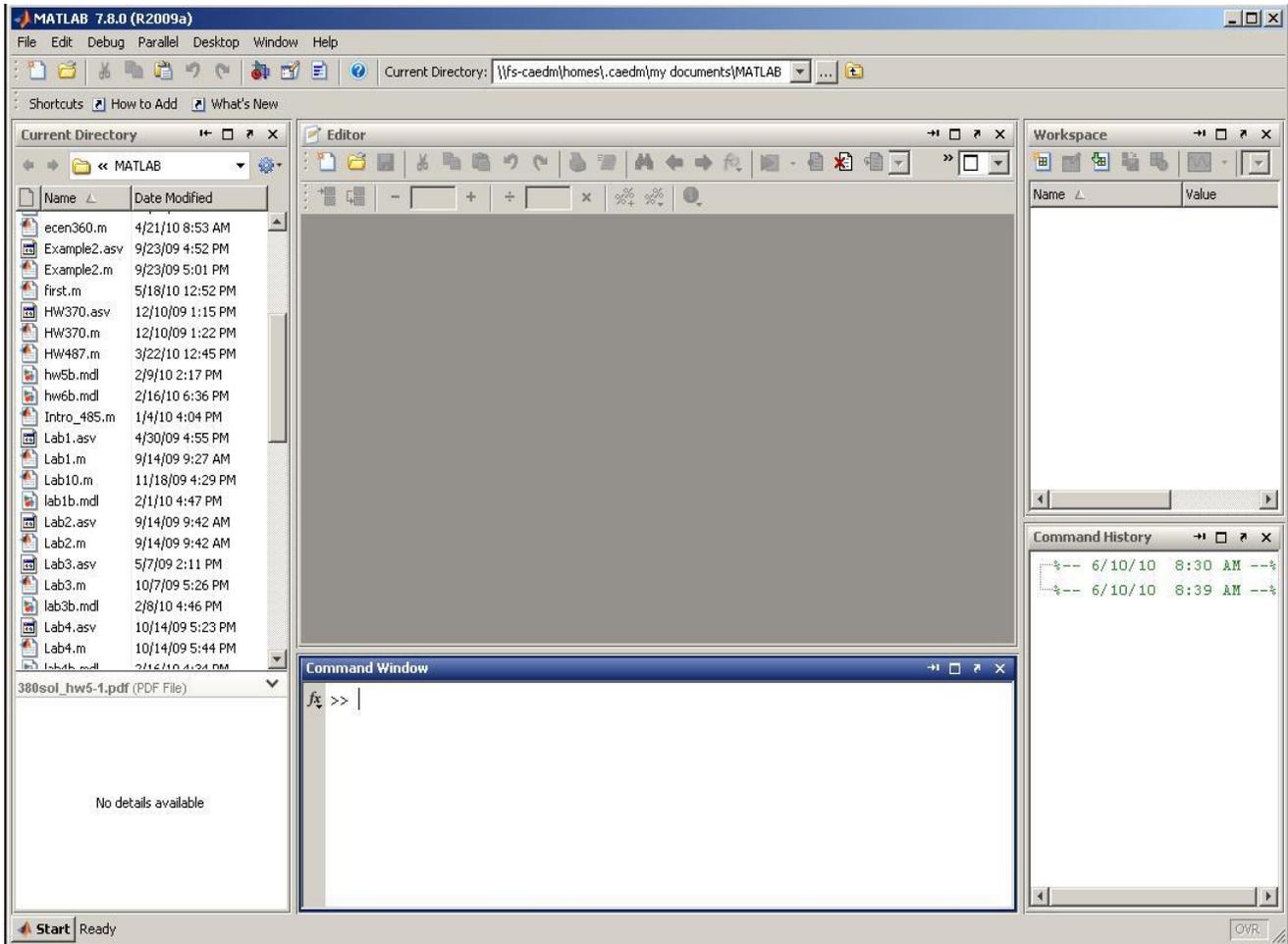


Notice on the left side of the screen, MATLAB shows the directory you are in and all the files and folders that are saved there. This is where you will keep all the programs you write or import. You can organize or create information there directly.

On the right side of the screen in the upper quadrant is where all your variables will display. The right, lower quadrant will display your recent commands from the command window.

# Creating an M-file

You can calculate problems create and use variables, etc. from the command window, but it is generally better practice to make your own file from which to run your program. M-files are macros of MATLAB commands that are stored in a text file. M-files allow the user to edit code without reentering it into the command line. To create an M-file, select *File >> New >> Blank M file* or select the *New M-File* button, circled in red in the picture below.
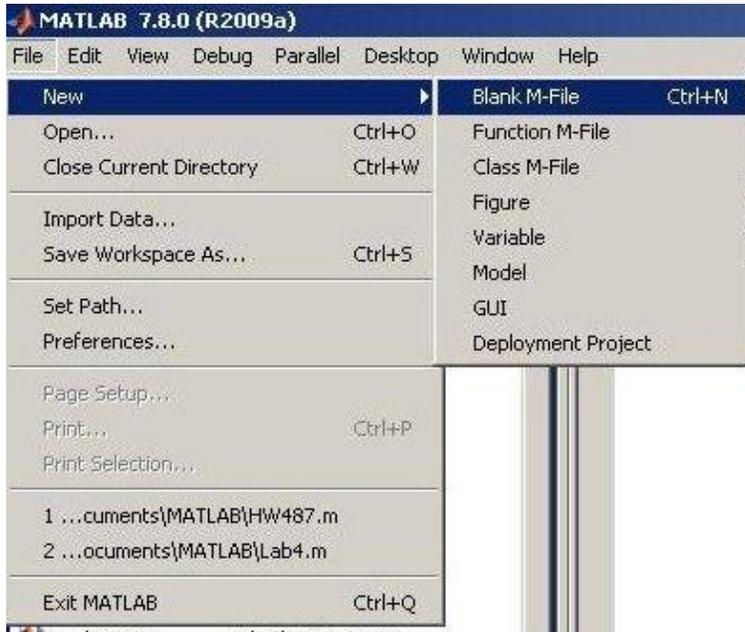
Your screen should now look like:
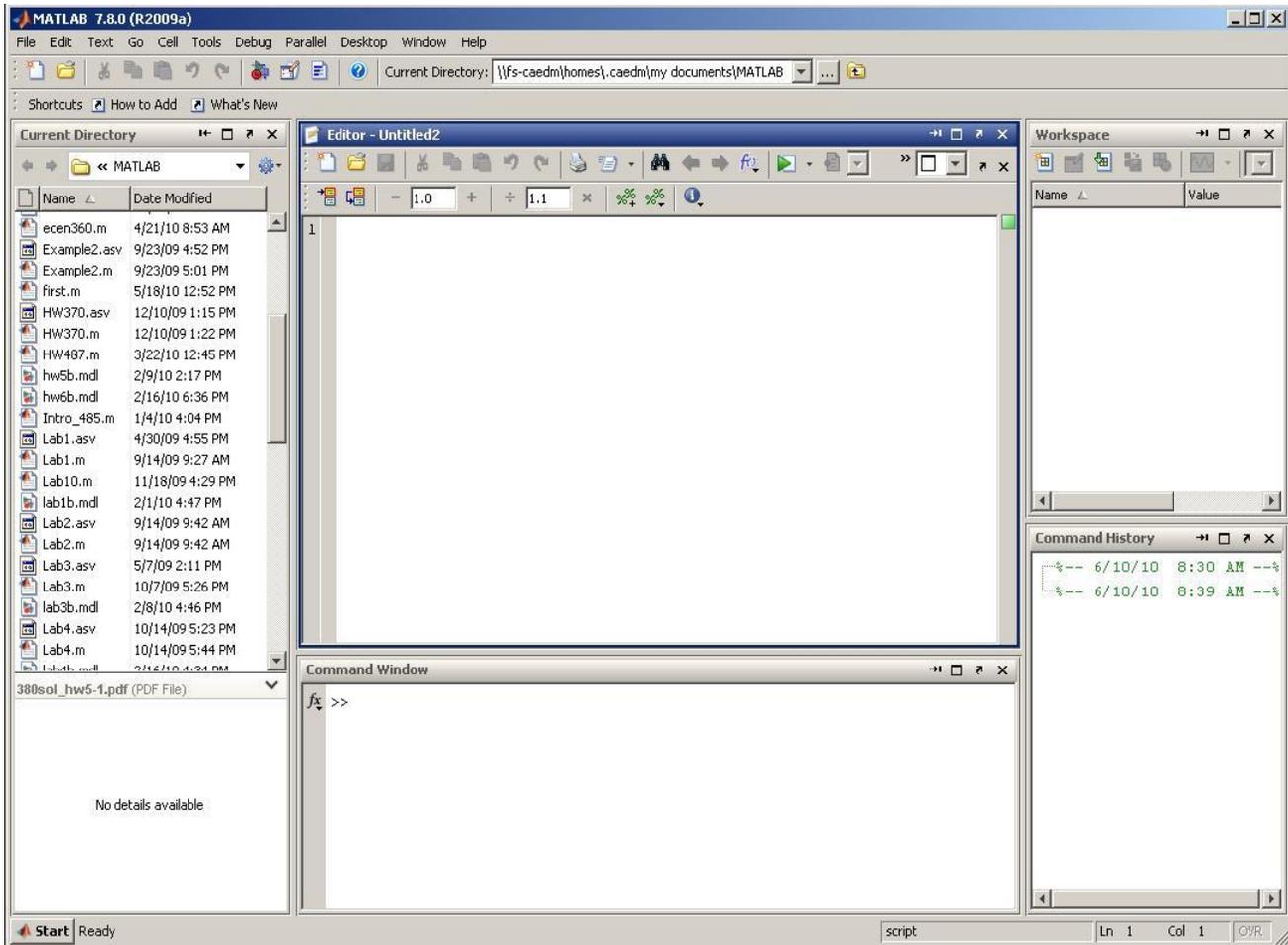


When you start writing your program, you will need to save it as *".m"* file in the MATLAB directory before you can run it.

Troubleshooting:
Where is my Editor Window?
-If your Editor Window is not on your MATLAB workstation, you can open your Editor Window by clicking the Window menu and selecting Editor.

# How to save and run an M-file

There are two ways to save and run an M-file. The first is to save using the Save As … option under the File menu and typing the filename of the macro into the command prompt and hit *Enter*.



The second is simply to click the green arrow button at the top of the file, as shown in the following picture. This green arrow save the M-file under the current directory and runs the M-file.
*Note: The hotkey to save and run your m file is F5.*
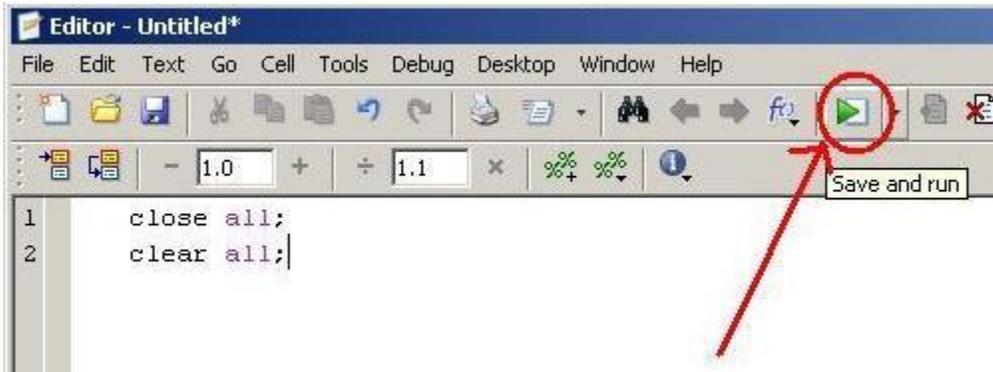


You should now be prepared to start the programming part of MATLAB. Other tutorials exist to help you with the syntax and structure of the language, as well as good programming techniques and tips. Continue to mess around with the different setting options and buttons at the top of the screen as well as the tutorials MATLAB provides, found in *Help >> Product Help >> MATLAB >> Getting Started*.

# MATLAB variables

MATLAB allows you to define variables to store numbers and calculations and to calculate values. *Whenever you see indented lines with* `this font`, *like the line below, type this code into MATLAB at the prompt.*

    2*1.5

What does the code above do?

To calculate a value and store it with a name (a variable), do the following code shown below:

Now you can just type x to see what is contained in the variable x. Ending the line with a semicolon suppresses printing of the output. To see the output, do:



In many of the statements in this lab, the semicolon is deliberately omitted so that you can see the results. However, most of the time you use MATLAB, it is probably a good idea to include the semicolon.

Now enter these same lines of code into an M-file and save it into your current working directory with the filename first.
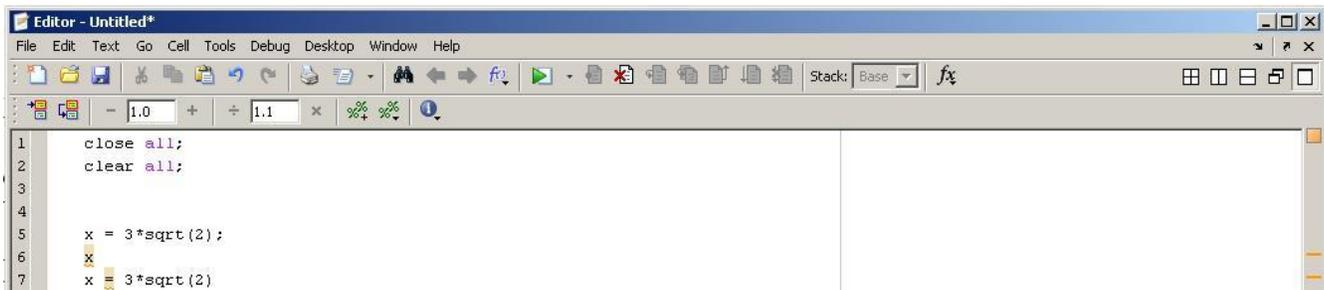


Run your M-file and observe if you get the same result.
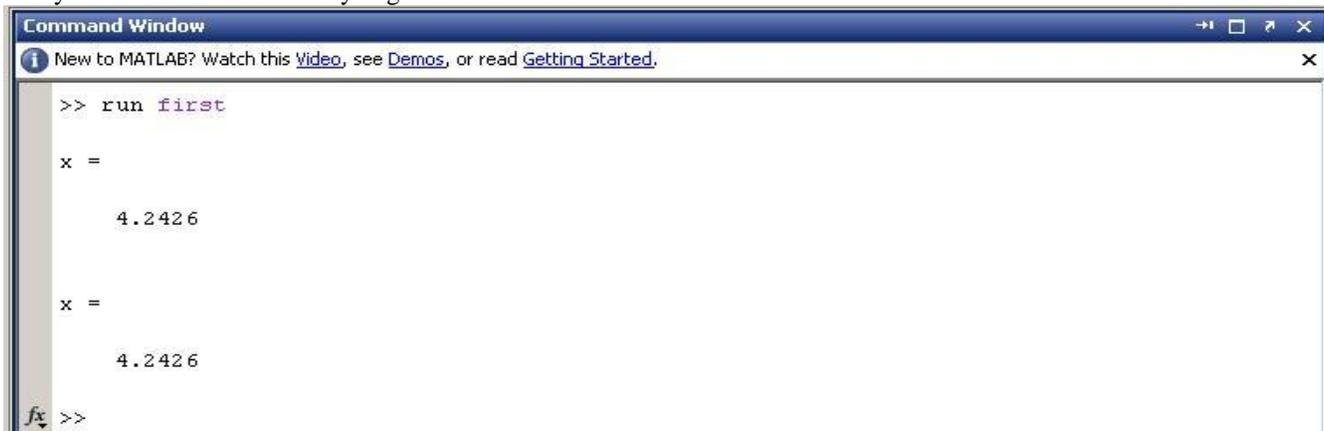


You can create a vector-valued output:

```
vec = [3 2 5.6]
```

and perform operations on it:

```
vec3 = 3*vec
```

which multiplies each element by 3.

Note: You can use the Workspace window to view the values of variables and vectors you created.



An incremental vector (that is, a vector that contains a series of numbers each separated by a fixed increment) can be created as `[start:increment:end]`. The notation `[start:end]` assumes an increment of 1. Let's first create a vector that runs from 1 to 100, incrementing by 1.

```
index1 = [1:100]
```

Remember, to suppress the output when creating the vector you can simply add a semicolon to the end of the command:

```
index1 = [1:100];
```

Try creating a vector `index2` from 0 to 5 that increments by 0.5:

```
index2 = [0:0.5:5]
```

How about an index that runs from 0 to 2*pi with 10 elements?

```
index3 = [0:2*pi/10:2*pi]
```

Does `index3` have 10 elements?  To see the number of elements in a vector, type:

```
length(index3)
```

Can you explain why `index3` has 11 elements?

To transpose our `index3` vector, type the following:

```
index3'
```

Or we could store a transposed version of `index3` in a new vector `index4`:

```
index4 = index3';
```

A 2-D array (or matrix) can be formed as follows:

```
array1 = [ 3 2 5.6 7; 1 4 5 9; 1 2 3 4]
```

To reference a single element of an array, put the index values in parentheses.  In the previous example, to reference the 3rd row and 2nd column, type:

```
q = array1(3,2)
```

You can also refer to a range of elements:

```
q = array1(1:3,4)
```

The `1:2` in this case selects the 1$^{st}$ through 3$^{rd}$ rows in the array, and the `4` selects the 4$^{th}$ column.  You can even reference ranges of arrays on multiple dimensions.  Try:

```
q = array1(2:3,2:4)
```

Spend the time to understand why you get the output you get.  Effective indexing into vectors and matrices is a powerful skill in MATLAB.

To select all elements of an array on a given dimension, a colon alone can be used for that dimension:

```
q = array1([1 3],:)
```

Note that the `[1 3]` selects just the 1$^{st}$ and 3$^{rd}$ rows of the matrix, while the `:` selects all columns.

MATLAB conveniently allows you to perform the same operation on every element of an input vector or array:

```
z1 = sqrt(index2);
```

or:

```
z2 = sqrt(array1);
```

This calculates the square root of every element of `index2` and `array1` and stores the results in `z1` and `z2` respectively.  Note that we've used the semicolon at the end of the each line to suppress the output.  To view the contents of a variable, vector, or array, just type its name at the MATLAB prompt and hit enter:

```
z1
```

## Matrix operations and element-by-element operations

MATLAB can perform operations such as multiplication on vectors and matrices in different ways.  Let's create a couple of simple 2-dimensional matrices:

```
m1 = [ 1 4 3 ; 2 3 1 ; 5 4 3 ]
m2 = [ 1 1 1 ; 0 0 1 ; 0 2 0 ]
```

Now try the following:

```
m1*m2
```

This performs a matrix multiplication.  Verify the result.  If we wish instead to multiply each element in `m1` and `m2` by each other on an element-by-element basis, we can type:

```
m1.*m2
```

Consider the difference between the following two operations:

```
m1^2
m1.^2
```

## Basic plotting

MATLAB has a number of powerful plotting capabilities. We will look at a few of them here.

Remember our vector `index3`, which contains values running from 0 to 2*pi?  Let's plot it:

```
        plot(index3)
```

Now let's plot its sin:

```
        plot(sin(index3))
```

Note that MATLAB plots each element of `index3` or `sin(index3)` as connected lines. The horizontal axis begins with 1 and counts forward by default. If you want to use your own index and add labels, type the following:

```
        plot(index3, sin(index3))
        xlabel('input')
        ylabel('output')
```

Suppose you want to compare `sin(index3)` to `cos(index3)` on the same plot with different line styles and also add a legend:

```
        y1 = sin(index3);
        y2 = cos(index3);
        plot(index3,y1,'-',index3,y2,':')
        legend('sine','cosine')
```

See `help plot` to learn about changing line colors and other line styles.

Suppose you just want to plot y1 as a discrete sequence, type the following:
```
        stem(index3,y1,'-')
        legend('sine','cosine')
```

A bar graph requires putting the input into a different format. Combine the two row vectors `y1` and `y2` into a single matrix by transforming each row into a column and combining:

```
        newmatrix = [y1' y2']
```

Then bar plot:

```
        bar(index3,newmatrix)
```

If you don't like the fact that MATLAB creates extra space on the margins of the plot, try:

```
        axis tight
```

2-D plots of different sorts are also possible:

```
        xind = [0:0.2:20];
        yind = [0:0.2:10];
        [xx,yy] = meshgrid(xind,yind); % create matrices of x values and y values
        sinxy = sin(0.1*pi*xx - 0.2*pi*yy);
        mesh(xind,yind,sinxy)
```

Try an image plot:

```
        imshow(sinxy)
```

and a contour plot:

```
        contour(sinxy)
```

Surf is a function that plots 3-D color surface. Plot the following surface.

```
        k = 5;
        n = 2^k-1
        [x,y,z] = sphere(n);
        c = hadamard(2^k);
        surf(x,y,z,c);
        colormap([.875 .875 .875; 1 1 1])
        axis equal;
```

See `help surf` to learn about the different parameters that `surf` accepts.

You can create a new figure with

```
figure
```

or activate an existing figure (for example, #1) with

```
figure(1)
```

# Control flow

MATLAB is an interactive package as well as a full-blown programming environment. You can write a series of statements that can modify variables or branch to different statements depending on the current state of certain variables. The most important of these are `if` statements and other conditional statements, `while` statements, and `for` loops. We will look at `for` loops and conditional statements here. A `for` loop allows you to step through a sequence of values of a certain variable and then redo the calculation inside the loop. It has the general form `for i=1:n, < program > , end`.

To calculate the Fibonnaci sequence:

```
f(1:2) = [0 1];
for k=3:20,
f(k) = f(k-1) + f(k-2);
end
f
```

Tests can be performed on variables to control which statements are executed and other behavior. A simple but powerful technique is the conditional statement. Consider this statement:

```
first = ([0:10]' < 3)
```

Each element in the vector `[0:10]'` is tested to see whether it is < 3. If it is, the result is a 1 (true). If not, the result is a 0 (false). Change the < to > and to <= and observe the change in output. This behavior can be used to create signals and control output variables and program behavior.

# Some useful MATLAB tips

Once you have assigned a value to a variable in MATLAB (like `n2` in the example above), that variable stays active for the remainder of your MATLAB session. In order to view all of the variables that are currently defined for your session, try typing:

```
who
```

As you see, this returns a complete list of defined variables. If you want more information about your current variables, you can type:

```
whos
```

This command gives you additional information about each variable, such as its size and type. To clear a variable and remove it from the variable space, you can use the `clear` command. Try the following:

```
z1 = zeros(10);
whos z1
z1
clear z1
z1
```

If you use the `clear` command without any arguments, it clears the entire variable space. Don't try this right now, as we'd like to keep our variable space intact for the moment.

A common mistake for scientists and engineers using MATLAB is to accidentally redefine the variables `i` and `j`. These are

both pre-defined in MATLAB as the square root of negative one:

```
i
j
```

However, MATLAB (for reasons known only to them) allows you to redefine them.

```
i = 15
j = 20
i
j
```

Since these variables are commonly used in programming as indices or counters, it is common for us to get careless and use them as indices or counters in MATLAB. I suggest getting in the habit of using other variables (`k`, `l`, `m`) for indices and counters in MATLAB. In the event that you have accidentally redefined `i` and/or `j`, you can restore them to their default values.

```
clear i j
i
j
```

While at the MATLAB prompt, you can see the current directory (or folder) that you are working in using the command:

```
pwd
```

You can change directories using the `cd` command and view files in the current directory using the `ls` command.

Your current variable space can be saved to a .mat file (a file with the .mat extension) using the `save` command, and then reload that variable space using the `load` command. Try the following:

```
save myvars
ls
clear
whos
load myvars
whos
```

As you'll see, this series of commands (1) creates a file in the current directory called "myvars.mat" which contains your current variable space, (2) clears out the variable space, and (3) reloads your variable space.

Let's clear the entire variable workspace before moving on to the next section.

```
clear
```

## Running MATLAB scripts from a file

MATLAB commands can be written to a file with a .m extension in the text editor of your choice (or the built in MATLAB text editor), and then executed by typing the filename at the MATLAB command prompt (minus the .m extension). This is useful for automating a series of MATLAB commands, such as computations that you have to perform repeatedly from the command line. It is also useful for debugging, since you can edit a series of commands to correct a mistake and rapidly re-execute.

When executed, MATLAB scripts share the variable space of your current session. You can reference or use a variable in a script without defining it in the script if it is already defined in your current variable space.

Create a text file called "petals.m" (either in the text editor of your choice or the built in MATLAB editor) that contains the following code:

```
% An M-file script to produce        % Comment lines
% "flower petal" plots
theta = -pi:0.01:pi;                  % Computations
rho(1,:) = 2 * sin(5 * theta) .^ 2;
rho(2,:) = cos(10 * theta) .^ 3;
rho(3,:) = sin(theta) .^ 2;
```

```
rho(4,:) = 5 * cos(3.5 * theta) .^ 3;
for k = 1:4
    polar(theta, rho(k,:))          % Graphics output
    pause
end
```

This file is now a MATLAB script.  At the MATLAB prompt, you can now type the command:

```
petals
```

and the script will be executed.  (Note that the file "petals.m" must either be in the current MATLAB directory or in the MATLAB path.  To view the current MATLAB path, use the command `path`.  Alternately, the graphical MATLAB interface will allow you to navigate to the script and select it for execution.)

After the script displays a plot, press **Enter** or **Return** to move to the next plot. There are no input or output arguments; `petals` creates the variables it needs in the MATLAB workspace. When execution completes, the variables (`i`, `theta`, and `rho`) remain in the workspace. To see a listing of them, enter `whos` at the command prompt.


# Creating .m file functions in MATLAB


Functions are program routines, usually implemented in M-files, that accept input arguments and return output arguments. You define MATLAB functions within a function M-file; that is, a file that begins with a line containing the `function` key word. You cannot define a function within a script M-file or at the MATLAB command line.

Functions always begin with a function definition line and end either with the first matching `end` statement, the occurrence of another function definition line, or the end of the M-file, whichever comes first. Using end to mark the end of a function definition is required only when the function being defined contains one or more nested functions.

Functions have their own variable workspace, and operate on variables within their own workspace. This workspace is separate from the base variable workspace (the workspace that you access at the MATLAB command prompt and in scripts).

**The Function Workspace:**  Each M-file function has an area of memory, separate from the MATLAB base workspace, in which it operates. This area, called the function workspace, gives each function its own workspace context.

While using MATLAB, the only variables you can access are those in the calling context (i.e., those passed in as arguments to the function call). The variables that you pass to a function must be in the calling context, and the function returns its output arguments to the calling workspace context.

Try entering the following simple commands into a text file called "average.m".  The average function is a simple M-file that calculates the average of the elements in a vector:

```
function y = average(x)
% AVERAGE Mean of vector elements.
% AVERAGE(X), where X is a vector, is the mean of vector
% elements. Nonvector input results in an error.
[m,n] = size(x);
if (~((m == 1) | (n == 1)) | (m == 1 & n == 1))
    error('Input must be a vector')
end
y = sum(x)/length(x);      % Actual computation
```

The average function accepts a single input argument (`x`) and returns a single output argument (`y`). To call the average function, enter the following at the MATLAB prompt:

```
z = 1:99;
average(z)
```

# Generating random numbers

The generation of random numbers is an important tool that will be heavily used in ECEn 370. Additionally, when random numbers are used, the results of every person's homework are also different! Keep that in mind when working with other people.

Suppose that you want to generate a lot of random numbers that are uniformly distributed from 0 to 1. Type in the following command and execute it a few times:

```
rand
```

Notice that each time you execute the `rand` command you get a different number. MATLAB is using a numerical routine to generate these numbers. If you want many random numbers you can type in:

```
rand(1,10)
```

which will give you a vector of ten uniformly distributed random numbers. Often, we will compute statistics for random variables for which we can compute what the theoretical statistics should yield. For example, the mean of the uniformly distributed random variable (we have not covered this at this point) can be shown to be 0.5. To verify this, type in the following command and execute it a few times:

```
mean(rand(1,10000))
```

Notice that the answer is different each time but it tends towards 0.5. Thus, you can see how relevant statistics can be computed from random numbers generated using numerical methods.


# Plotting histograms of random outcomes

Often, what will be reported in homework is a histogram of the shape of a random variable or function of a random variable. The command

```
random('Binomial', 10, 0.4, 20, 1)
```

will generate twenty outcomes of a Binomial random variable with n=10 and p=0.4. Note that the single quote is used to denote a string. This is the " ' key next to the semicolon on a standard keyboard. The estimated shape of the Binomial distribution with these parameters can be seen by executing the following code:

```
trials=10000;
n=10;
p=0.3;
[counts, x_divisions] = hist(random('Binomial', n, p, trials, 1), [0:n]);
stem(x_divisions, counts / trials);
```

Note that each time you execute this code, the estimated shape will be different because it is based on the outcomes of a random variable.

**TURN IN THE FOLLOWING:**

**1) Print the plot of your estimated histogram of the binomial random variable found in the last section.**

**2) Write the twentieth element of the Fibonacci sequence you generated starting from 0, 1, 1, 2,... in the lower right portion of your plot.**