Introduction

MATLAB is both a programming language and an integrated development environment (IDE). This introduction discusses the programming language.

MATLAB is an abbreviation for Matrix Laboratory and its creators original focus was to create a tool for matrix calculations. A matrix is a two-dimensional array of numbers. Similarly, a vector is a one-dimension array of numbers. Examples of a matrix 3x3 matrix *A* and a 3x5 matrix *B* is

$$[A] = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix} \qquad [B] = \begin{bmatrix} 1 & 2 & 3 & 8 & 10 \\ 2 & 4 & 5 & 14 & 15 \\ 3 & 5 & 6 & -1 & 0 \end{bmatrix}$$

Two examples of vectors *a* and *b* are

$$[a] = \begin{bmatrix} 70 & 80 & 1 & 2 & 4 & 7 \end{bmatrix}$$
$$[b] = \begin{bmatrix} 4 \\ 5 \end{bmatrix}$$

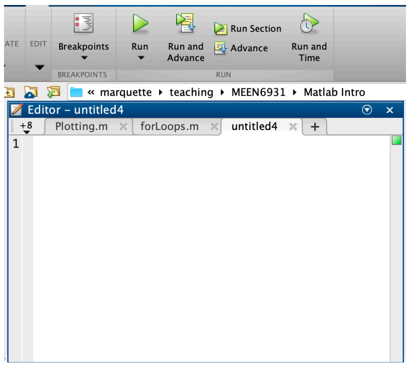
There are many mathematical notations for vectors and matrices and many people notate a vector using the curled brackets {}, but this text will refer to all arrays (i.e. matrices and vectors) using square brackets [].

Getting Started

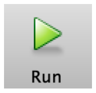
A good way to start each Matlab Script (i.e., your code) is with `clear variables, close all, clc.` This will clear all of the variables in your workspace, close all of the figures you have plotted, and clear the command line. Some older code will use `clear all` rather than `clear variables`. This is fine but `clear variables` is better in most cases.
Lets try it:

```
clear variables; close all; clc;

a = 1
b = 2;
b
```

Type the above into the editor shown below (titled Editor -untitled4 for this example)



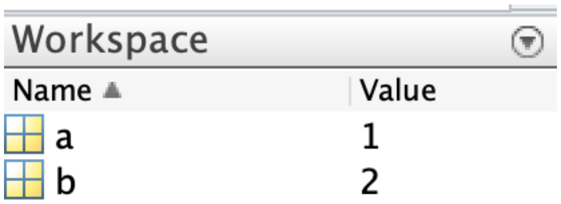
And then hit the run button



Run this program and the output should be

```
a =

     1

b =

     2
```

In the first line the semicolon denotes a new line, using the semicolon we could put three commands on one line (in generally this is not a good idea, but for simple commands it's ok). Also note that the semicolon suppresses the output of a line. The code above assigned the value of 1 to the variable a and the value of 2 to the variable b. Also, note that by simply writing b without a semicolon Matlab displays the value of b but does not do any other operation to it. You can see the value of a and b in the workspace



Workspace

Name ▲	Value
a	1
b	2

Hello Zorld

A common first program to write is the Hello World! program that prints out the character string "Hello World". Matlab offers several ways to do this, one is shown below.

```
clear variables; close all; clc;

disp('Hello World!');
```

In general, it is good form to end every line with a semicolon; even if you want to use the `disp` function to display something, it's good form to end that line with the semicolon. Note that character strings (i.e., text that is not meant to have a numerical value) are enclosed with apostrophes.

User Exercise
Write a Matlab script that displays the character string Hello Zorld!

The output should be

Hello Zorld!

Matlab Arrays

Let's program an array (a variable that holds multiple numbers similar to a matrix in math) with components given below
$[a] = \begin{bmatrix} 70 & 80 & 1 & 2 & 4 & 7 \end{bmatrix}$
Matlab (unlike Python and c++) numbers its arrays starting from 1 giving

	$[a] = [70$	80	1	2	4	$7]$
Index	1	2	3	4	5	6
Call Statement	a(1)	a(2)	a(3)	a(4)	a(5)	a(6)

So, let's write a script to hold this array

```
clear variables; close all; clc;

a = [70 80 1 2 4 7];
```

Note that square brackets are used to enclose the array. This array is a row (horizontal list) of numbers, you can use commas to separate these values, but they are not required.

To make a column (vertical list) of numbers you could use semicolons to go to the next line or just go to the next line. Alternatively, you could write a row array and take the transpose as shown below.

```
% write a column array: Method 1
a = [70; 80; 1; 2; 4; 7];
% write a column array: Method 2
a = [70
     80
      1
      2
      4
      7];
% write a column array: Method 3
a = [70 80 1 2 4 7];
a = a';
```

Note that the apostrophe is this case takes the transpose of the array (i.e., it switches the rows and columns). Also, note that the percentage sign % creates comments. This are not executed but help to organize and document your code. You should use comments, you should use comments often, you should use comments for short code. You should use comments for long code. You should use comments!

Also, note that the arrays overwrite each other, for instance if we write

```
a = [70 80 1 2 4 7];
a = 5;
a = [1; 3];
disp(a)
```

it will display

```
    1
    3
```

This is because the value of *a* is overwritten, each time it is assigned. This is also why it is a good idea to keep different codes separate. For example, if you have 3 homework problem, its good form to write three separate scripts. HW1.m, HW2.m and HW3.m. Combining these all into one homework assignment and running sections of the code is dangerous because you can inadvertently overwrite variables, it is hard to debug, and it is not a feature that works in all programming languages.

Now let's access the values in the first *a* array we assigned

```
a = [70 80 1 2 4 7];
a(2)
a(6)
a(end)
a(1:3)
```

The output should be

```
ans = 80

ans = 7

ans = 7

ans = 70    80     1
```

Thus, the parentheses let us access the value for a certain index (spot) in the array. For instance, the 2nd spot contains the value 80, so a(2) is 80. Note that we can also tell Matlab we want the last index a(end) and it will give us the value 7. You can also subtract from end, so a(end-1) would give the 2nd to last value which is 4. You can also access several value using the :, so a(1:3) gives the first through 3rd value of a. 70,80 and 1.

Now let's assign some values to the components of a 3x3 matrix

$$[A] = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}$$

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

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

The output is

```
ans =    1
ans =    5
ans =    6
ans =  1    2    3
```

Note that we now need two indices representing the rows and columns, so A(1,1) is the first row and first column value. A(3,2) is the third row and the 2nd column. A(end, end) is the last row and the last column. A(1,:) is the first row and all of the columns (represented by a :)

User Exercise
Write a Matlab script that displays the following values from the array

$$[B] = \begin{bmatrix} 1 & 2 & 3 & 8 & 10 \\ 2 & 4 & 5 & 14 & 15 \\ 3 & 5 & 6 & -1 & 0 \end{bmatrix}$$

a) The first row, first column of B
b) The last column of B
c) The 3rd row, 4th column of B

The output should be

```
ans =   1

ans =

   10
   15
    0

ans =  -1
```

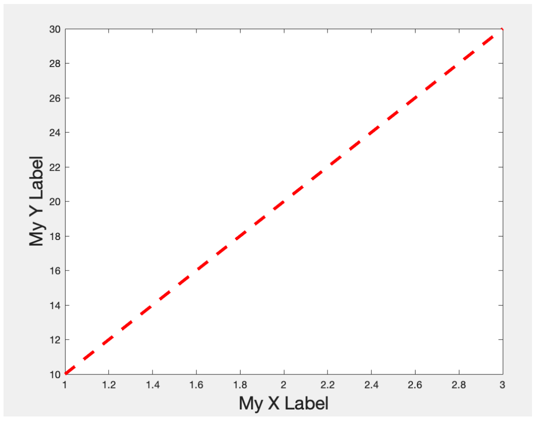
Plotting

Matlab, unlike some programming languages, has simple data visualization features that are easy to use and generally produce better plots than Excel. These plots are also easier to manipulate and deal with large sets of data much better than Excel. Matlab's plotting command is `plot(x,y)`.

Let's make of plot of y vs x.

```
clear variables; close all; clc;

x = [1 2 3];
y = [10 20 30];

plot(x,y, 'r--', 'lineWidth', 3)
xlabel('My X Label','fontSize',18)
ylabel('My Y Label','fontSize',18)
```

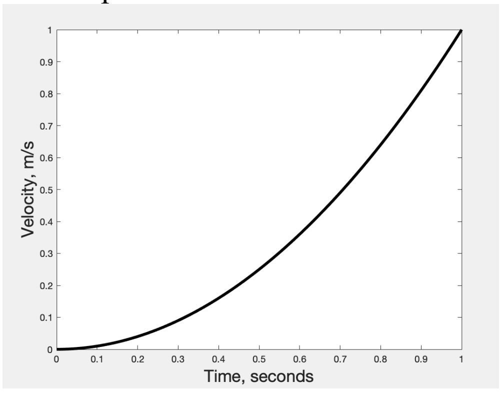
The values of x and y are required, the rest are optional options. The characters string 'r--' tells Matlab you want a red line that is dashed (i.e., --), if you wanted a blue solid line you would write 'b-', if you wanted black dotted lined you would write 'k:' (yeah, black is *k* for some reason). The LineWidth option is also optional but sets the width to something a human can see. Try it without these options and see how the plot looks. The labels 'My X Label' is also your choice, if you were plotting mechanical stress you might want this to read 'Stress, MPa', if you were plotting velocity you might want this to read 'Velocity, m/s', etc. The fontSize option is optional but helps make the font a size humans can read. Try it without these options and see how the plot looks.



Now let's plot an independent variable t that goes from 0 to 1 and a dependent variable v that that is the square of t.

```
t = linspace(0,1,100);
v = t.^2;

figure(2)
plot(t,v, 'k-', 'lineWidth', 3)
xlabel('Time, seconds','fontSize',18)
ylabel('Velocity, m/s','fontSize',18)
```

The output is



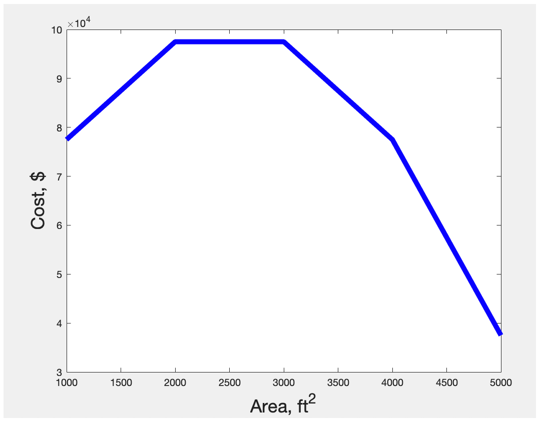
Here we used `linspace(start,finish, number of points)` to equally space 100 points between the starting value of 0 and the ending value of 1. The value of v was also calculated from t, using the .^ operator, the dot tells Matlab to square (as notated by the ^) each entry of t. This works for multiplication, division, etc. If you want to multiply all entries of an array *a* by all entries of an array *b* you would write a.*b , if you wrote a*b it would try to do matrix multiplication.

Also note that the `figure` command lets you tell it what figure you want, if you run the plot command three times without the `figure` command it will overwrite your figure each time.

User Exercise

Write a Matlab script that plots an independent variable *area* vs a depend variable *cost* at 5 equally spaced points where *area* goes from 1000 to 5000 and
$cost = 100000 + -(0.1 * (area - 2500))^2$.
Plot using a blue line with a line width of 5.

The output should be



For Loops

The `for` loop is a control flow statement that allows you to repeat a task multiple times.
The syntax is

`for` *counter* = *counter start* : *counter end*
        … task to do over and over
`end`

An example is below that prints the word "Ni!" 3 times

```
for i = 1:3
    disp('Ni!');
end
```

The `for` command tells Matlab to start the loop. You must pick a counter (i,j,k are common), this example chooses i as the counter. You then set i equal to a range of integer numbers i = 1:3. i can start from any integer (1, 10, -100) and go to any integer (3, 20, 100) as long as the end counter is larger than the start counter (it can count backwards too, but let's not do that now). Then you write the code for the task you want to repeat over-and-over followed by the `end` command which tell Matlab the loop is done.
The output is below

```
Ni!
Ni!
Ni!
```

Let's walk through what Matlab does in the `for` loop.

First time through the loop
i = 1
Matlab displays "Ni!"
The loop ends and starts from the top again
Second time through the loop
i = 2
Matlab displays "Ni!"
The loop ends and starts from the top again

Third time through the loop
i = 3
Matlab displays “Ni!”
The loop ends
The counter is now equal to the maximum value (i.e., 3) so the loop stops and the next lines of code are run.

Note that the value of i did not really matter, it could have gone from 12-14 and the output would have been the same. Therefore, this example is simple but uncommon. A more common example of a `for` loop is to do something over-and-over which is a function of the counter number (i.e., it’s a function of how many times you have been through the loop). For example, the code below prints out the numbers 1 through 3. On the first time through the loop it prints 1 because the counter is 1 and the second time it prints 2 because the counter is 2 and on the third time it prints 3 because the counter is 3

```
for i = 1:3
    disp(i);
end
```

The output is:

```
    1
    2
    3
```

Let’s walk through what Matlab does in the `for` loop.
First time through the loop
i = 1
Matlab displays the value of i which for this time through the loop is 1
The loop ends and starts from the top again
Second time through the loop
i = 2
Matlab displays the value of i which for this time through the loop is 2
The loop ends and starts from the top again
Third time through the loop
i = 3
Matlab displays the value of i which for this time through the loop is 3
The loop ends
The counter is now equal to the maximum value (i.e., 3) so the loop stops and the next lines of code are run.

The counters do not need to increase in increments of 1, the following code increases the counter by increments of 2. So i starts as 1, then increases 2 until it reaches the end value of 6.

```
for i = 1:2:6
    disp(i)
end
```

The output is:

```
 1
 3
 5
```

An even more common example of `for` loops is to use the counter as an index in an array. In the following example, we want to print the values in an array x (not the counter values) where [x] = [1 12 15]. So, we want to individually print the 1st value in the array, then the second value, then the third value.

```
x = [1,12,15];

for i = 1:length(x)
    disp(x(i));
end
```

The output is:

```
 1
12
15
```

Note that Matlab has a command `length()` that determines the length of a 1D array. If a row array has 57 entries then length will return 57, if a column array has 2 entries then length will return 2. By setting the counter to go from 1 to the length of the array x we make our code more flexible. If we were to add five values to x so that `x = [1,12,15,1,2,3,4,5]` then the for loop would start with i = 1 and go until i = 8 without having to change any of the code in the for loop. If we had written it as `for i = 1:3`, then we would have to manually change the for loop code to `for i = 1:8` when we had five new entries, in fact, we would have to manually change the for loop every time the length of x changes.

Let's walk through what Matlab does in the `for` loop.
First time through the loop
i = 1
Matlab displays the ith value of the array x, since i=1 this is the first value of the array x. The first value of the array x is 1.
The loop ends and starts from the top again
Second time through the loop
i = 2
Matlab displays the ith value of the array x, since i=2 this is the second value of the array x. The second value of the array x is 12.
The loop ends and starts from the top again
Third time through the loop

i = 3
Matlab displays the ith value of the array x, since i=3 this is the third value of the array x. The third value of the array x is 15.
The loop ends
The counter is now equal to 3 which is the length of the array, so the loop stops and the next lines of code are run.

The nice thing about using the counter as an index in an array is that we can now iterate through value that are not integers. For example, x could hold double precision floating point numbers (i.e. decimals)

```
x = [1.0,12.2,15.5];

for i = 1:length(x)
    disp(x(i));
end
```

The output is:

```
    1
  12.2000
  15.5000
```

In fact, we are not even limited to numbers, x could contain a list of my favorite farm animals and we can still iterate through the list and print the values.

```
x = [{'cow'},{'pig'},{'dragon'}];

for i = 1:length(x)
    disp(x(i));
end
```

The output is

```
   'cow'
   'pig'
   'dragon'
```

Note the characters string are enclosed in {} to tell Matlab each entry is a whole word. If this was not done, it would just see an array of letter x = cowpigdragon

While these examples show instances of program output, in computational/numerical problems it is more common to use for loops to do some sort of computation. For example, you may want to write a loop that calculated 3 times the value of i where is goes from 1 to 3 in increments of 1.

```
for i = 1:3
    a = 3*i;
    disp(a);
end
```

The output is

```
  3
  6
  9
```

Perhaps this calculates the number of pieces of silverware needed (a) based on the number of guests at a dinner (i). We could write this without a for loop as:
a = 3*1
a = 3*2
a = 3*3
but this is not very flexible (you have to re-write it every time the number of i values changes) and becomes cumbersome if i goes from 1 to 10000. It would also be more difficult if *a* was something like the cost of a house and *i* was the number of bedrooms. In this case calculating *a* (the cost of a house) as a function of *i* (the number of bedrooms) might be very complex and it's better to write the computation out one time in the for loop than many times for every possible value of i.

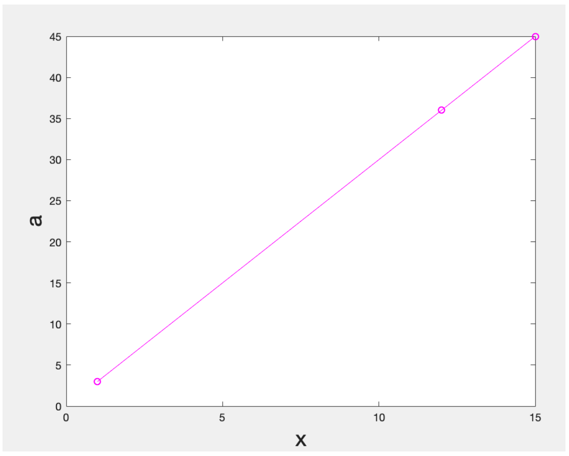
Again, it is more common in computational/numerical problems to use i as an index for an array as in

```
x = [1,12,15];

for i = 1:length(x)
    a(i) = 3*x(i);
end

plot(x,a,'m-o')
xlabel('x','fontSize',20)
ylabel('a','fontSize',20)
```

Here an array (a) holds 3 multiplied by each value in the array x.
So, on the first time through the loop i=1 and the ith value of x is 1 so the first value of *a* is set to 3*1. Likewise, when i=2 ith value of x is 12 and the second value of *a* is set to 3*12, etc. This is particularly useful when calculating a function because the for loop produce a dependent variable *a* that is the same length as the independent variable *x* and evaluated at each value of x.

The output is

```
    3
    6
    9
```



Note that since, a and x are the same length they can be plotted against each other and in the plot the line is magenta, and each value is indicated by a round marker. The option `'m-o'` tells the plot command to use magenta ‘m’, plot a solid line ‘-’ and have the marks be circles ‘o’.

The previous code will run fine, but Matlab may give you a warning that it is inefficient. The reason for this, is that the length of array *a* increase each time through the loop. Let’s walk through what Matlab does in the `for` loop.
First time through the loop
i = 1
x(1) = 1
a = 3
The loop ends and starts from the top again
Second time through the loop
i = 2
x(2) = 12
a = [3 36]
The loop ends and starts from the top again
i = 3
x(3) = 15
a = [3 36 45]
The loop ends
The counter is now equal to 3 which is the length of the array, so the loop stops and the next lines of code are run.

Note that on each time through the loop the length of *a* increases by 1. This means (in very general terms) that on the first time through Matlab has to ask your computer for enough memory to hold one value. On the second time through Matlab has to ask your computer for enough memory to hold two values, and on the final time through Matlab has to ask your computer for enough memory to hold three values. So Matlab has to interact with your computer and its memory allocations 3 times. It is more efficient if Matlab can do this once. So, the following code pre-allocates the size of *a* once, then the loop simply fills in the memory locations in *a* with values rather than having to ask for memory each time through.

```
x = [1,12,15];
a = zeros(length(x));
for i = 1:length(x)
    a(i) = 3*x(i);
end
```

The result is the same, but its better programming form in general and can increase the speed of your code for large computations.

Looping come in especially handy when trying to add things up. In the example below all of the value of i from 10 to 12 are added and the result is displayed

```
sum = 0;

for i = 10:12
    sum = sum + i;
end
disp(['sum = ', num2str(sum)])
```

The output is
sum = 33

Note that we had to initialize the variable *sum* as 0. If we did not do this, Matlab would not know what *sum* was on the first loop and would produce an error. Also note that this time I displayed a character string ‘sum =’ concatenated with the value of sum. Since you cannot concatenate (add together) string and numbers I had to convert the value of sum to a character string using the *num2str* command. Also note that I had to enclose the whole displayed statement in an array using []. Without this it would try to display ‘sum =’ with a display option num2str which does not make any sense as display has no option called num2str.

Let’s conclude our discussion of for loops with a bit more complex problem that is very common in numerical methods. In this case we have a *local* matrix

$$a = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

And we want add this matrix to the diagonals of a 4x4 *global* matrix *A* (the diagonals are the center values of a matrix where the row number equals the column number). So, we want to start in the upper left hand corner of A and add in the 2x2 matrix (a) to rows 1 through 2 and columns

1 through 2 then move down one spot on the diagonal to A(2,2) and add in the 2x2 matrix (a) to rows 2 through 3 and columns 2 through 3, etc until all the diagonal values are populated. All other value in A should be zeros.

The following code does this

```
a = [1 -1
    -1 1];

sizeA = 4;

A = zeros(sizeA);

for i = 1:sizeA-1
    A(i:i+1,i:i+1) = A(i:i+1,i:i+1) + a;
end
A
```

The output is

```
A =
   1  -1   0   0
  -1   2  -1   0
   0  -1   2  -1
   0   0  -1   1
```

Note first that we set the number of rows and columns of A using the variable sizeA. This allows A to be any dimension by changing the value of sizeA and no other code. Now we initialize all of the values in A to zero using the `zeros` command which, given one entry, creates a square array (in this case a 4x4). Now we use the *i* counter to designate the spot on the diagonal of *A* where the upper left hand corner of *a* will be located. Since *a* is a 2x2, if its upper left hand corner is a location A(1,1) then its bottom right hand corner will be at A(2,2) and in general it will take up space in row and columns i and also i + 1. Therefore if we are at diagonal entry 4 we cannot insert the 2x2 because there is now row and column 5. So we must loop between the first diagonal and the second to last diagonal (i.e., 1:sizeA-1). Now *a* is a 2x2 so we must give it a location in A that is also 2x2, its first row and column will be at i and its last row and column will be at i+1 so we want *a* to be inserted into A at rows i through i+1 and columns i through i+1that is why we use A(i:i+1,i:i+1). Since we want *a* to be added to *A* each time we must do `A(i:i+1,i:i+1) + a`. If we walk through this step by step
On loop 1
i =1

```
A =
   1  -1   0   0
  -1   1   0   0
   0   0   0   0
   0   0   0   0
```

On loop 2
i =2
A =

```
   1   -1    0    0
  -1    2   -1    0
   0   -1    1    0
   0    0    0    0
```

Note how the diagonal value from the previous lower right hand corner of *a* adds to the current upper left hand corner of *a*.
On loop 3
i =3
A =

```
   1   -1    0    0
  -1    2   -1    0
   0   -1    2   -1
   0    0   -1    1
```

The counter is now equal to 3 which is it maximum value (1 minus the length of the array), so the loop stops and the next lines of code are run.

User Exercise

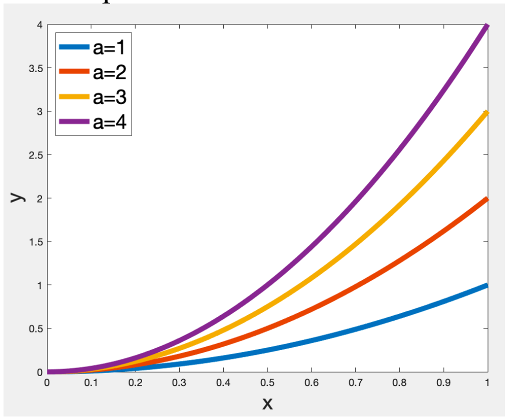
For parameters $a = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$
Use a for loop to loop through all 4 values of *a* and plot
$y = ax^2$
For 100 values of x ranging from 0 to 1. Use a font size of 20 for the x and y labels and a line width of 5.

You can use the

```
legend('a=1','a=2','a=3','a=4','fontsize',20,'location','northwest')
```

to label your plot. You should plot inside the for loop and use the hold on command so that plots do not overwrite each other.
For example
plot(x,y), hold on;

The output should be



If statement

The `if` command is a conditional statement that tells Matlab: *if* something happens *then* do something. Older codes used to have `if,then` statements, but the `then` was somewhat redundant so it is dropped in modern codes, the syntax of an `if` statement is

`if` *condition*

… do something

`end`

for example

```
a = 4;
if a > 3
    disp('a > 3')
end
```

The output is
a >3
likewise

```
a = 1;
if a > 3
    disp('a > 3')
end
```

has an output of nothing because the if statement is never true so it never does anything. This might be a bit perplexing if you are running the code, did it work? Did it run at all? Therefore, an else command can be used to tell Matlab what to do if the if statement is never true

```
a = 1;
if a > 3
    disp('a > 3')
else
    disp('a <= 3')
end
```

This time the output is
a <= 3

Note that the `if` statement depends on some condition. This condition can be
Equal ==
Not Equal ~=
Greater than >
Less Than <
Greater or equal >=
Less than or equal <=

For example

```
a = 1;
if a == 1
    disp('a = 1')
end
```

will output
a = 1
since the value of a is 1 and the condition is true.

It’s also possible to have conditions that are checked only if the previous condition is not true. For example, the following checks if a = 0 if so, it outputs “a is 0”, if this is not true then it knows a is positive or negative so it can check if a is less than zero and print “a is negative”. If a is neither equal to zero or less than zero it must be positive so the code prints “a is positive”. Note that if a is zero it does not do the other checks

```
a = 1;
if a == 0
    disp('a is zero');
elseif a < 0
    disp('a is negative');
else
    disp('a is positive');
end
```

The output is
a is positive

Then try

```
a = -1;
if a == 0
    disp('a is zero');
elseif a < 0
    disp('a is negative');
else
    disp('a is positive');
end
```

The output is
a is negative

Then try

```
a = 0;
if a == 0
    disp('a is zero');
elseif a < 0
    disp('a is negative');
else
    disp('a is positive');
end
```

The output is
a is zero

This is good debugging practice to check all the conditions and make sure they work. In generally getting your code to run is only the first step, code can run and still be wrong, so check as many cases as is needed for each problem.

One thing to note is the above example, is that Matlab has been good to us and sees that we entered a = 0, so it know a is truly equal to zero; in reality, your computer can only hold so many digits and so a is not exactly equal to zero. For my computer numbers are precise up to plus or minus 2.2204e-16. To figure this out I use the `eps` command. The value will differ between different computers. So, the zero value might actually be 2.1204e-16, but again Matlab is smart and knows that 2.1204e-16 is approximately zero and make the a == 0 condition result as true. However, letting your software do this check is often dangerous in numerical methods, we lose some control of how tightly these conditions are being enforce. For example

```
a = 0 + eps;
if a == 0
    disp('a is zero');
elseif a < 0
    disp('a is negative');
else
    disp('a is positive');
end
a
```

The output is
a is positive
a =2.2204e-16

This is strictly true, but is probably not what you are looking for because 2.2204e-16 is pretty small and you probably wanted to round this to zero. A better way to do this type of if statement is to avoid equal and not equal if you are dealing with floating point numbers (i.e., decimals)

```
a = 0 + eps;
tol = 1e-9;
if abs(a) <= tol
    disp('a is zero');
elseif a < 0
    disp('a is negative');
else
    disp('a is positive');
end
a
```

The output is
a is zero
a = 2.2204e-16

which is probably what you where looking for. This also allows you – as the user of this program – to control how tightly you want the conditions to be enforced. If you are trying to find a values that makes a function zero and it takes 2 days for the program to get the function less than 1e-12, you could change the tolerance (`tol`) to 1e-6 and maybe then the code runs faster, perhaps in minutes rather than days and 1e-12 and 1e-6 are both very small numbers so your loss of accuracy is negligible.

Combined with for loops, if statements are very powerful tools in programming. For example consider the following code

```
for i = 1:100
    if i == 12
       a = i^4 + i^3 - i^2 -i + 2;
       disp(a)
    end
end
```

The output is
22310

Which is computed (from a rather complex formula) only when i=12. If we only wanted this calculation when i=12, then this saved us a huge amount of computations because it did not waste time computing it for the 99 values that were not i=12. Now, this is a but contrived, in that we could just done this calculation for i=12 and avoid the loop. But, let's assume we want to do this calculation when i is a factor of 24 and when we don't know off hand what the factors are.

```
for i = 1:100
    if mod(i,24) == 0
       a = i^4 + i^3 - i^2 -i + 2;
       disp(['i = ', num2str(i) ': a = ', num2str(a)])
    end
end
```

The output is

```
i = 24: a = 345002
i = 48: a = 5416658
i = 72: a = 27241850
i = 96: a = 85810082
```

it computed both the factors of 24 and the values of *a*. Note that the `mod` command is the modulo operation which gives the remainder after division (for example mod(4,2) is 0 because 2 divides into four with no remainder, but mod(4,3) is 1 because 3 divides into 4 once with a remainder of 1). Especially if the computation of *a* was much more complex, perhaps taking minutes or hours to compute, this if statement would have saved a lot of time because it only computed *a* four time (when we cared about it) not 100 times (96 times when we did not care about it).

Finally, if statement can be useful in populating arrays. For example, the following (somewhat inefficient code) inserts the value of 5 on the diagonal of a 4x4 matrix.

```
A = zeros(4);

for i = 1:size(A,1)
    for j = 1:size(A,2)
        if i == j
            A(i,j) = 5;
        end
    end
end
A
```

The output is

```
A =

     5     0     0     0
     0     5     0     0
     0     0     5     0
     0     0     0     5
```

Note the command `size` is used size(A,1) gives the number of rows, size(A,2) gives the number of columns in an array.

This is an example of how you might build a large matrix with specific values in specific spots. This example is inefficient because you don't really need both for loops (why?) and there is a command is Matlab that make this type of array (which is 5 times and identity matrix)

```
A = 5*eye(4)
```

Produces the same result.

```
A =

    5    0    0    0
    0    5    0    0
    0    0    5    0
    0    0    0    5
```

But in general, you might want something more complicated as is shown in the user exercise.

User Exercise

Using if statements and for loops to write a Matlab script that builds a 4x4 matrix which is all zeros with the exception of:

- Each diagonal's value is its row number (remember the diagonal is where the rows and columns numbers are the same, in the example above all diagonal values are 5)
- Each value directly to the right of the diagonal is -20
- Each value directly to the left of the diagonal is 3
- If the row number divided by the column number is exactly 4 then the entry is 6

The output should look like

```
A =

    1  -20    0    0
    3    2  -20    0
    0    3    3  -20
    6    0    3    4
```

While statement

While statements execute while something is true, for example the following code outputs the value of *a* while it is less than 4

```
a = 1;
while a < 4
   a = a + 1;
   disp(a);
end
```

The output is

```
  2
  3
  4
```

While statement are commonly used but I avoid them at all cost for two reasons. The first is that they are prone to running indefinitely, causing the code to crash, your machine to freeze, memory to leak, and all kinds of bad stuff to happen.

For example, the following code

```
a = 1;
while a < 4
   a = a - 1;
   disp(a);
end
```

Has an error where I meant to write a = a + 1, but accidentally wrote a = a -1 and now the code runs indefinitely, and needs killed.

A better implementation would use a for loop

```
a = 1;
endValue = 100;
for i = 1:endValue
   a = a + 1;
   if a > 4
       break
   end
   disp(a);
end

if i == endValue
    disp('something is wrong')
end
```

The output is the same

```
2
3
4
```

However, now there is a check. Note that the break command exits the for loop.

If I make the same typo (minus when I meant plus)

```
a = 1;
endValue = 100;
for i = 1:endValue
   a = a - 1;
   if a > 4
       break
   end
   disp(a);
end

if i == endValue
    disp('something is wrong')
end
```

Then its spits out a bunch of wrong numbers, but does not run indefinitely. And also tells me that 'something is wrong'

The while loop also does not keep track of how many loops it took. For example, if you are searching for a value that minimizes a function and its take 3 loops or 30000 loops whiles does not record this naturally. With a for loop it is easy to see the value of the counter when the break statement is called. It is also easy to limit the number of iterations you would like to take. For example you could set endValue to 300, if its take more than 300 loops than there is something wrong and you should check your code. With a while statement you could be running 300000 loops every time you call the while statement and never realize that why your code is running slow.

In short, I do not recommend while statements.

Final Thoughts

Matlab is a powerful tool that can increase your productivity, creativity, and viability as an engineer. However, it can be frustrating, and even the most seasoned Maltab programmers rarely have a script run on the first try. On some computers Matlab actually beeps when you do something wrong (I turn this off, it can be maddening). If you work through the exercises here, and really understand each piece of code, you will be a step closer to fully using Matlab's many abilities. If fact, many numerical methods and computational codes can be programmed with simply the tools used here. However, Matlab has 1000s of commands, so its often helpful to look at others code and learn from their techniques. Matlab takes time to learn, but with practice, patience, and a desire to learn, Matlab can become an invaluable engineering tool.