MATLAB Workshop Tutorial 1

Ethan J. Eldridge

January 30, 2012

Contents

List of Figures

1 Topics Covered In this Tutorial

This is the first of a few MATLAB tutorials intended to introduce key concepts to those interested in learning how to program in MATLAB. This tutorial assumes a basic knowledge of general programming concepts and that the user has downloaded and installed MATLAB itself.[1] After an introduction to the key data types and general format of the MATLAB language, conditionals, loops, and functions will be covered. General plotting will then be touched on. The next MATLAB tutorial will pick up where this one leaves off and will cover vectorization, animation, matrix manipulation, structures in MATLAB, and GUIS.

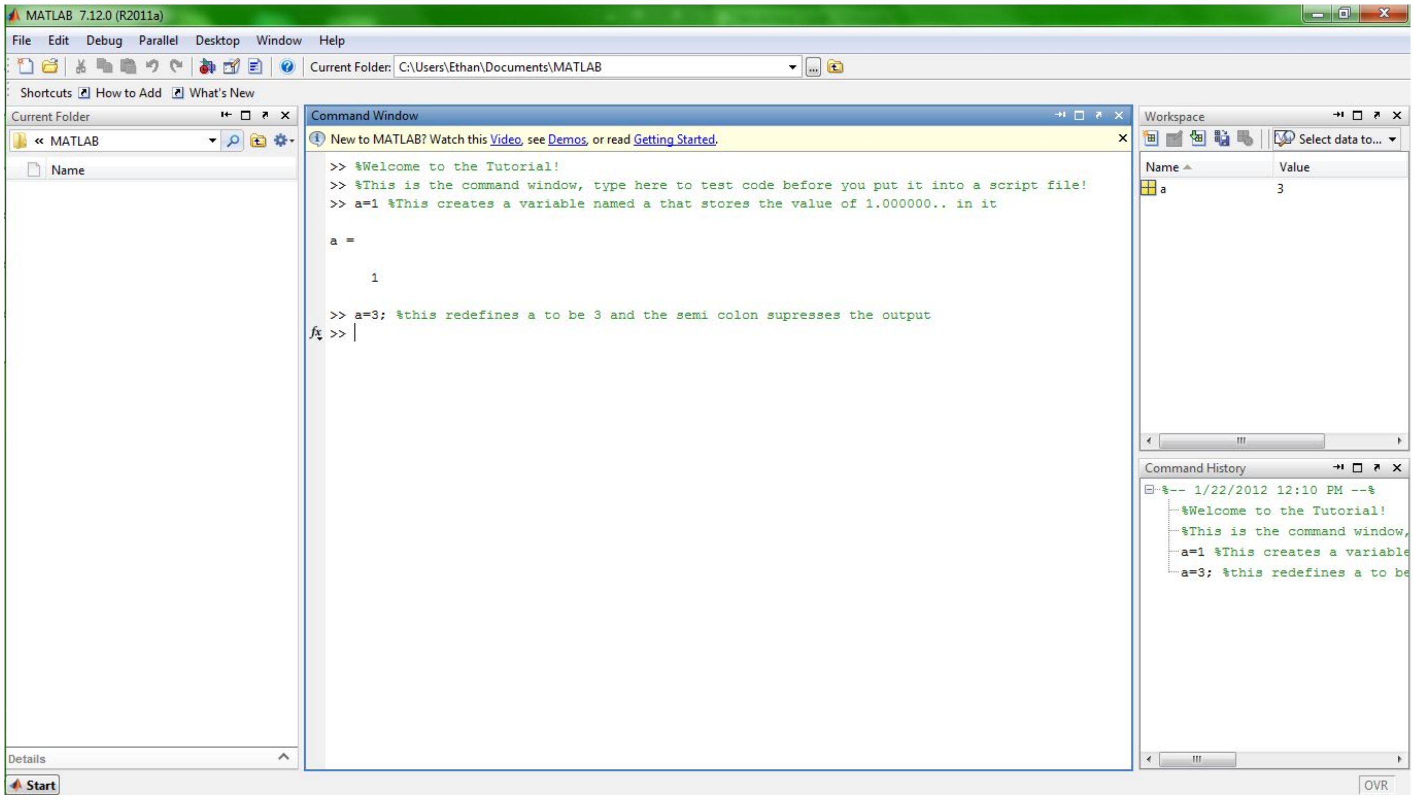
2 Key Differences of MATLAB from other Languages

Matlab is a very powerful language for working with Matrices and provides very easy File IO. The command window provides a quick way to execute code and is useful when you just want to type something out and not create an entirely new project to test one thing. (Ahem, Visual Studio, Codeblocks, Eclipse, etc) Below are a list of the few grievances that I have with Matlab's differences from other languages, with a brief explanation of the reasons why.

Matlab indexs at one
This is by far my largest complaint with the language. For those unfamiliar with the Computer Science Discipline, data structures that contain information can be indexed at a location by an offset amount from the location of the structure in memory. This knowledge of the low level view allows algorithms built at the higher level to work efficiently and effectively. Matlab indexs arrays and matrices starting with one instead of the typical computer science standard of zero. This is a mild inconveniance and will not bug engineers or math majors because starting at one is the norm for them.
To access a vector you use parentheses, not brackets.

Matlab is interpreted
Matlab is whats known as an interpreted language, which means that all the code can be tested in real time without compiling the souce files down into machine language. This is exactly what languages such as Python and Lisp do, and is very useful for what is known as the REPL process.[2] By providing an interactive environment for development, Matlab allows for fast prototyping of projects and quick modelling of anything one can program.
Matlab works with Double Precision numbers
Without getting too technical, Matlab's number system represents all numbers as double precision, this means that for a simple integer there's a lot of wasted space in memory. Because of this large amounts of matrix operations can result in the matlab program taking up a large amount of resources on your computer and running fairly slow compared to C++ or another compiled language. However, despite these slow ups, speed can be recovered through the process of vectorization of code.[3]
Matlab is weakly typed
For those coming from a C,C++, or Java background with no experience with Python or Lisp, this means that the type of a variable does not need to be defined at creation time. The Matlab interpreter takes care of using type inference to define the type of a variable when you create it.
When working with objects, the 'this' or 'self' parameter is not an implicit parameter
In python, C, C++, Java, and most other languages that provide Object Oriented programming, an

[1]For those who have not downloaded MATLAB, `http://www.mathworks.com` is the official place to download it, if one is affiliated with a university, there is a good chance that your software archive contains licenses and downloadable versions of the product.
[2]REPL stands for Read-Eval-Print loop
[3]Covered in the end of this tutorial


MATLAB 7.12.0 (R2011a)
File Edit Debug Parallel Desktop Window Help
Current Folder: C:\Users\Ethan\Documents\MATLAB
Shortcuts How to Add What's New
Current Folder
« MATLAB
Name
Details
Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
Workspace
Select data to...
Command History
Start
OVR

```
>> %Welcome to the Tutorial!
>> %This is the command window, type here to test code before you put it into a script file!
>> a=1 %This creates a variable named a that stores the value of 1.000000.. in it

a =

     1

>> a=3; %this redefines a to be 3 and the semi colon supresses the output
>> |
```

Name	Value
a	3

```
%-- 1/22/2012 12:10 PM --%
%Welcome to the Tutorial!
%This is the command window,
a=1 %This creates a variable
a=3; %this redefines a to be
```

Figure 1: Matlabs Primary Interface

instance pointer or reference is passed along to methods of the object by default. According to mathwork's help documentation this is not the case with Matlab and objects must be passed explicitly. This difference doesn't particularly matter for the scope of this tutorial, but it is good to note for those planning on using Matlab for OOP.

Matlab passes all variables by value

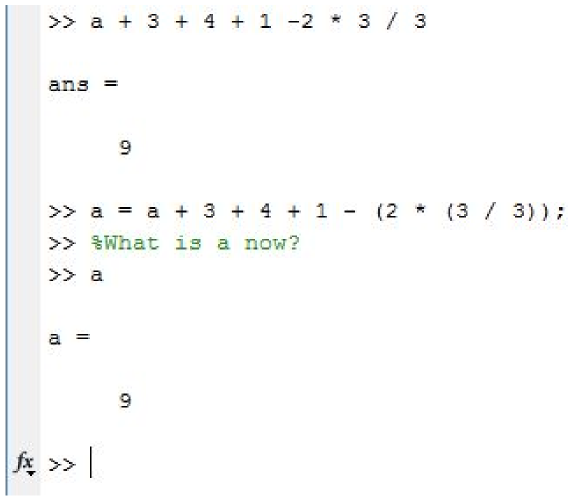
To the beginner this won't mean much, but to intermediate programmers or those with any C derived language background this can be important when working with programming that isn't strictly functional.

For more differences Mathworks has documentation that can be found very quickly just by searching in your favorite search engine with the right key phrases.

3 The Command Window and typical data types

When you start Matlab, the main console containing the Command Window, Command History, Current Folder, and the Workspace environment will be the primary focus at first, and also a secondary window will open that allows for editing m-files. An m-file is the source code for a Matlab program. M-files can contain scripts that just rattle off commands, or be functions that can be called from other m-files. Observing figure 1, you can see the primary screen in all of its glory. I've already written a few things into the prompt, lets take a moment to check out what's happening.

The % symbol begins a *comment*, this is just code that does nothing but provide insight into why you wrote something when you look at it a few months down the road. Moving down, I define a *variable* named **a**. Variables are just named spaces in memory that you can store the data that you would like to work with in; in order to access the data you've stored you use it's name to reference it from places in your program. Notice the only difference between when I defined **a** and when I redefined it, is the semi-colon at the end of the command (Yes the value changed to, but ignore that). The semi-colon supresses the output of a command from the command window. In layman's terms, if you'd like to see



```
>> a + 3 + 4 + 1 -2 * 3 / 3

ans =

     9

>> a = a + 3 + 4 + 1 - (2 * (3 / 3));
>> %What is a now?
>> a

a =

     9

>> |
```

Figure 2: Matlabs basic math operations for scaler values

Matlab spew out what it just did from any given command, don't place a semi colon at the end of the command. If you prefer your workspace a little bit neater, include the semicolon. You can put as many as you like, but one will suffice.

Taking a look at figure 2 we can start to get a feel for Matlabs basic math operations. Possibly because Matlab was created primarily for mathematicians, the order of operations is already defined into the standard methods. But if one would like to be absolutely sure of what order any operations are being down in, you can abuse parentheses to enforce the order. In the figure, the parentheses only enforce the natural order of operations. Also, note that the first time we use the operations we did not assign the returned value into a variable, and after hitting enter the value appears as 'ans = 9'. This ans variable is a default variable that matlab uses to show the output of the command window. When we want to save the value of a computation, as we will most of the time, we use the = sign to assign the right hand side to the left hand side.

In general math operators are pretty straight forward, when a . appears before an operation, the operation is applied to each component of a matrix versus the typical operations for the matrix. This is important to remember because it can save a lot of headaches later on. When you work with scaler numbers, you can also think of this as working with a 1x1 matrix. This tutorial won't cover matrix specific functions, but will use vectors in numerous examples, which are 1xn matrices in a respect.

Common Math Operations	
$+$	Addition
$-$	Subtraction
$.*$	Component-wise multiplication
$./$	Right division
$.\backslash$	Left division
$.\hat{}$	Component=wise power
$.'$	Transpose a matrix
$*$	Matrix multiplication

Moving away from numbers for a moment, we can create text by declaring strings. To create a string you simply surround the text with single qoutes. This is shown in figure 3, also despite the counter intuitive notion of apples and oranges, we can add an integer to a string. So what happens when we do this? Well, since strings are stored as a vector of single characters, and each character is associated with an ASCII number value, when we add 9 to the string, Matlab adds 9 to each ASCII value and if we turn this vector back into a string, well result with the text in figure 4.

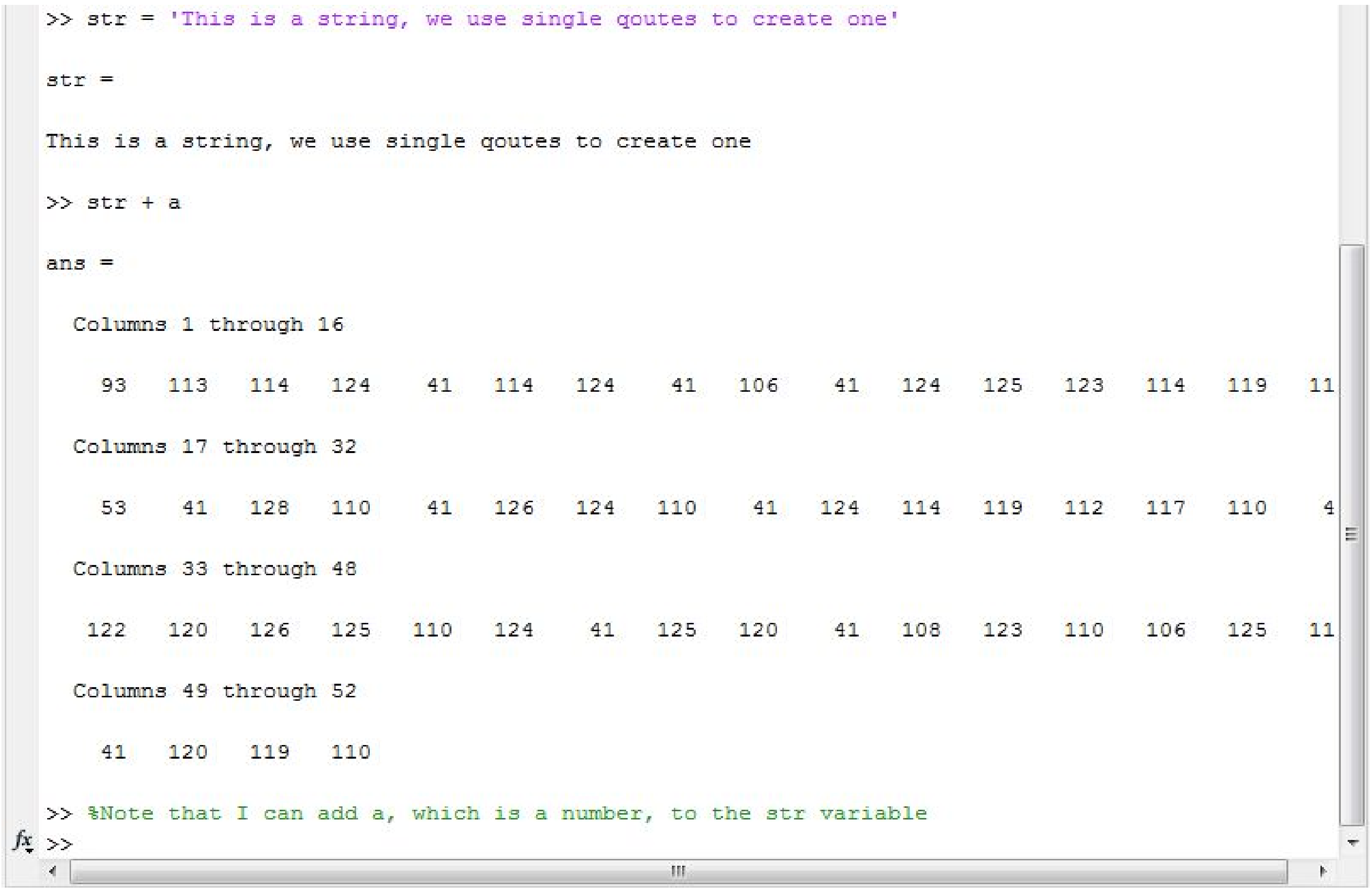

```
>> str = 'This is a string, we use single qoutes to create one'

str =

This is a string, we use single qoutes to create one

>> str + a

ans =

  Columns 1 through 16

    93   113   114   124    41   114   124    41   106    41   124   125   123   114   119   11

  Columns 17 through 32

    53    41   128   110    41   126   124   110    41   124   114   119   112   117   110    4

  Columns 33 through 48

   122   120   126   125   110   124    41   125   120    41   108   123   110   106   125   11

  Columns 49 through 52

    41   120   119   110

>> %Note that I can add a, which is a number, to the str variable
>>
```

Figure 3: The use of strings in Matlab

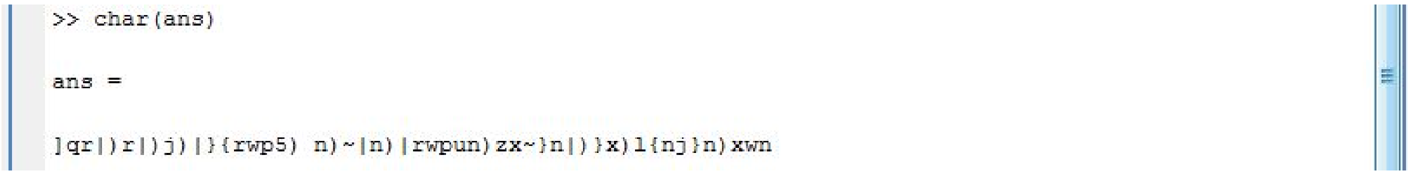
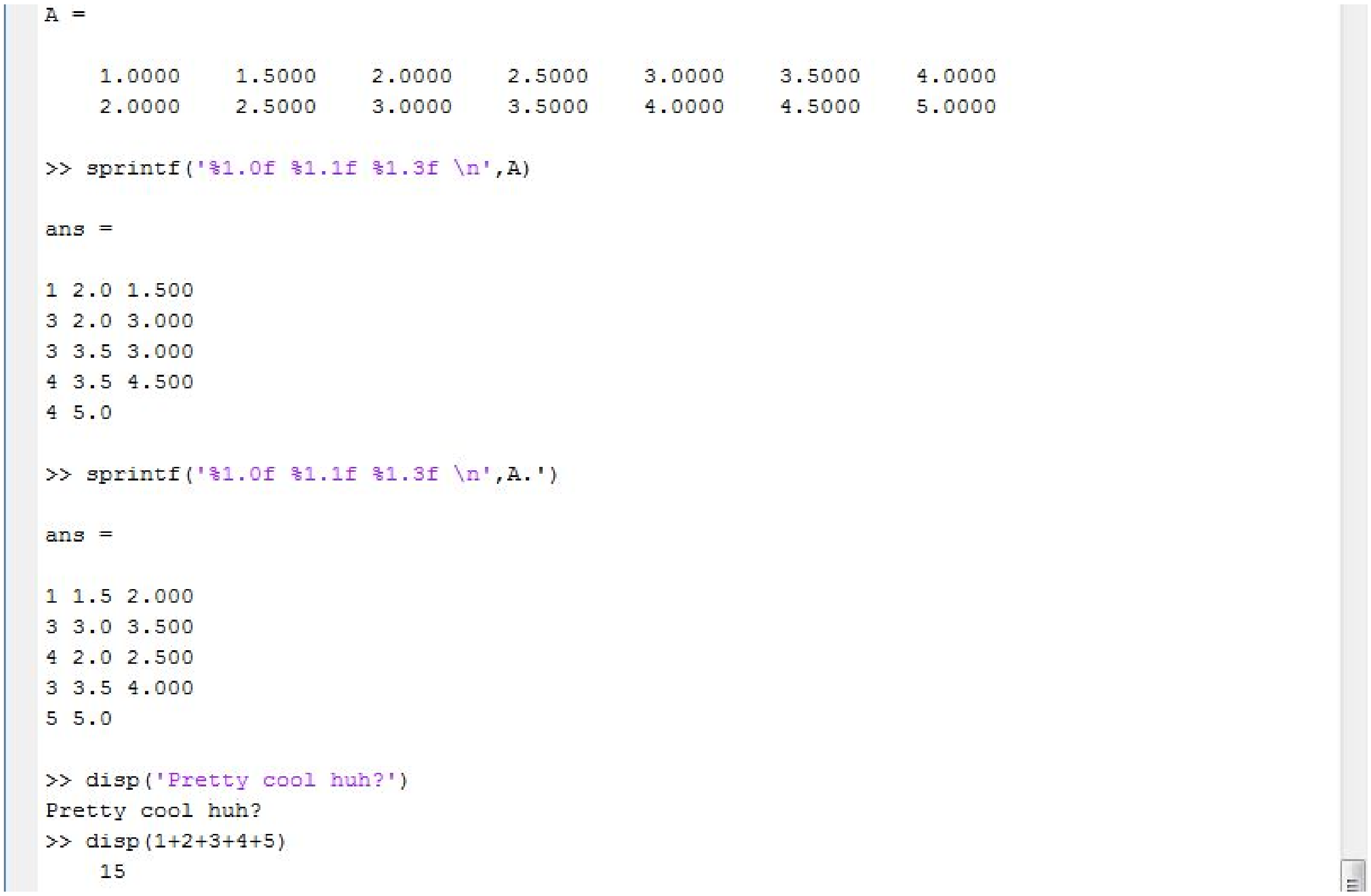

```
>> char(ans)

ans =

]qr|)r|)j)|}{rwp5) n)~|n)|rwpun)zx~}n|)}x)l{nj}n)xwn
```

Figure 4: The str variable converted back into a string after an addition takes place



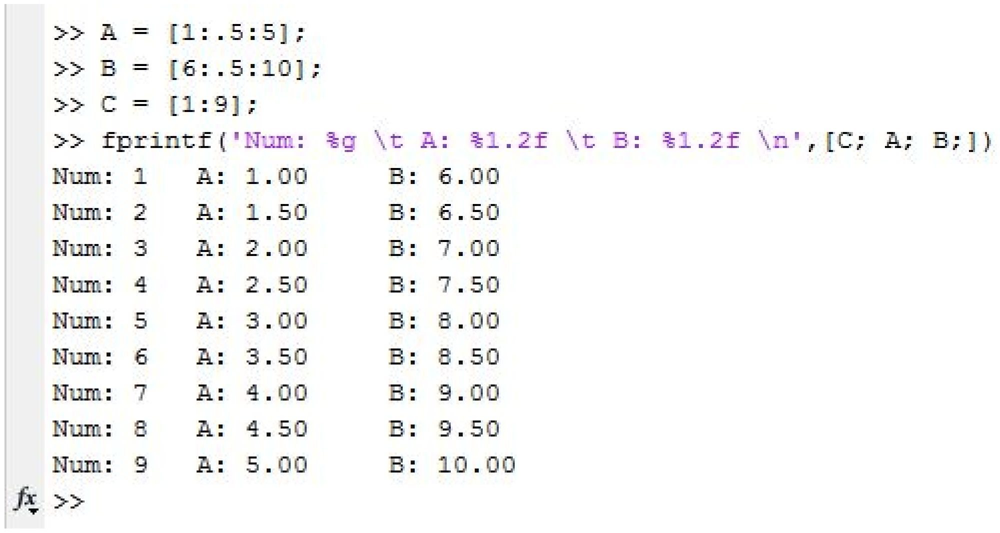
```
A =

    1.0000    1.5000    2.0000    2.5000    3.0000    3.5000    4.0000
    2.0000    2.5000    3.0000    3.5000    4.0000    4.5000    5.0000

>> sprintf('%1.0f %1.1f %1.3f \n',A)

ans =

1 2.0 1.500
3 2.0 3.000
3 3.5 3.000
4 3.5 4.500
4 5.0

>> sprintf('%1.0f %1.1f %1.3f \n',A.')

ans =

1 1.5 2.000
3 3.0 3.500
4 2.0 2.500
3 3.5 4.000
5 5.0

>> disp('Pretty cool huh?')
Pretty cool huh?
>> disp(1+2+3+4+5)
    15
```

Figure 5: disp, fprintf and sprintf commands

When you want to convert from a number to a string, or vice versa, you can use the command *num2str* and *str2num*. Note that in the previous example we used the *char* command to convert the ASCII values back to their character counterparts. If we had called *num2str* on **series ans** then, we would have ended up with a vector of ASCII values converted to strings, which is not the same as the symbols themselves.
Whenever you would like to display information to the console, the *disp* command will do the trick. Also, to format data into a specific form, the commands *sprintf* and *fprintf* allow for some handy tricks. Figure 5 shows disp and sprintf in action. The syntax for these commands are:

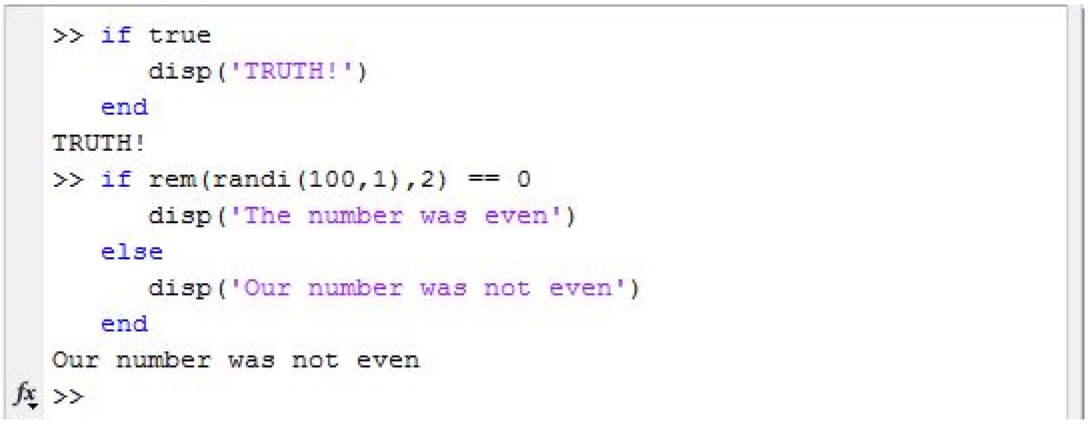
(Formatting String, Data to be passed in seperated by commas)

There are two things to remember when using sprintf and fprintf. First, they read the data in a column at a time, downwards. You can see this in the figure becuase in the first sprintf, the numbers are displayed from the first row first column, then the second row, first column. When trying to format data into a table the tab charater \$t\$ will help immensely to create nicely lined up columns. A complete list of formatters can be found by typing doc sprintf at the command line, or by looking through the documentation for the function on mathwork's website. In the example above, three numbers are read in perline and displayed as floating point numbers; the first of which has 0 trailing terms, the second having 1 decimal place, and the third having 3 places to the right of the decimal. Commonly, the formatter \%g is used to remove any trailing zeroes on a number.
A common task that one normally does with Matlab is to pass a multitude of data vectors to fprintf in order to display a neatly formatted table on the console, or to a file. Because of the way fprintf reads in files, each data vector should be placed on it's own row. To do this, a handy trick that I normally use is to construct a vector within the function argument itself and then apply the function. Figure 6 shows an example of this.



```
>> A = [1:.5:5];
>> B = [6:.5:10];
>> C = [1:9];
>> fprintf('Num: %g \t A: %1.2f \t B: %1.2f \n',[C; A; B;])
Num: 1   A: 1.00     B: 6.00
Num: 2   A: 1.50     B: 6.50
Num: 3   A: 2.00     B: 7.00
Num: 4   A: 2.50     B: 7.50
Num: 5   A: 3.00     B: 8.00
Num: 6   A: 3.50     B: 8.50
Num: 7   A: 4.00     B: 9.00
Num: 8   A: 4.50     B: 9.50
Num: 9   A: 5.00     B: 10.00
>>
```

Figure 6: Using fprintf to create a formatted table



```
>> if true
       disp('TRUTH!')
   end
TRUTH!
>> if rem(randi(100,1),2) == 0
       disp('The number was even')
   else
       disp('Our number was not even')
   end
Our number was not even
>>
```

Figure 7: A couple of If Statements, and the rem and randi functions

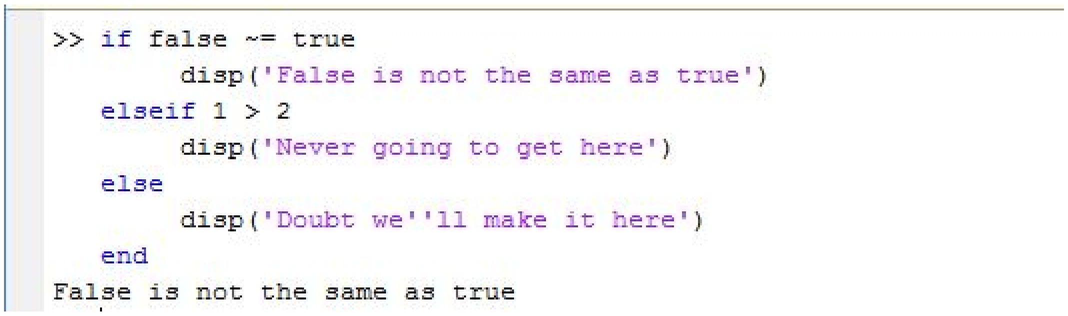
The last bit of information before we move on to the next section is the use of the range operator in the last example. To define the Matrix A (which is a 2 row, 7 column matrix) we used the range operator **:** to define the starting number, the steps between numbers, and the number to count up to. Key things to note about this operator is the following:

- The default step is 1, so 1:10 will count from 1 to 10
- The operator is *inclusive* so 1:10 will result in 10 numbers
- negative steps are permitted so 10:-1:1 will have the numbers one through ten in descending order

4 Conditionals

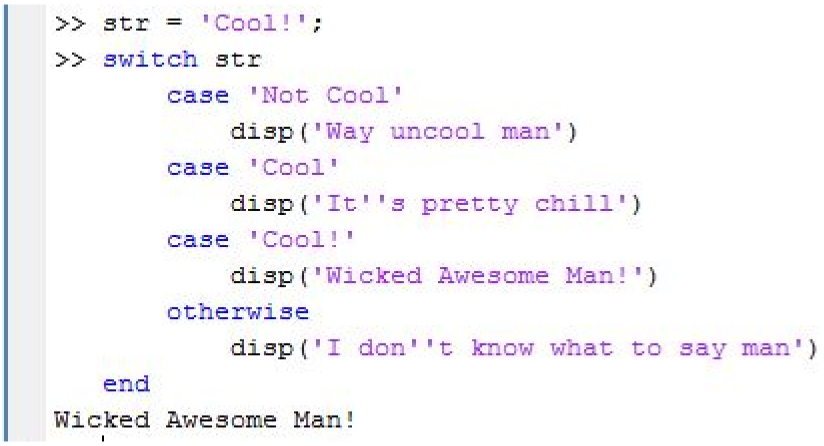
Now that we've covered some common basics of Matlab, lets move on to being able to control the execution of our programs. Matlab, like most languages, have a built in expression for true and false. Unsurprisingly, these are *true* and *false*, note that these built in variables are lowercase, not uppercase like most other languages. Let's consider the simplest conditional, the if statement in Figure 7.

The first statement in the figure shows a rather silly example, if the statement "true" is true, then we'll display the word TRUTH!, obviously this statement always evaluates to true, and not surprisingly Matlab returns our statement back to us the moment we hit enter. The second statement shows the use of two new functions as well as the addition of an else clause in our conditional statement. If the



```
>> if false ~= true
        disp('False is not the same as true')
   elseif 1 > 2
        disp('Never going to get here')
   else
        disp('Doubt we''ll make it here')
   end
False is not the same as true
```

Figure 8: The use of elseif



```
>> str = 'Cool!';
>> switch str
       case 'Not Cool'
           disp('Way uncool man')
       case 'Cool'
           disp('It''s pretty chill')
       case 'Cool!'
           disp('Wicked Awesome Man!')
       otherwise
           disp('I don''t know what to say man')
   end
Wicked Awesome Man!
```

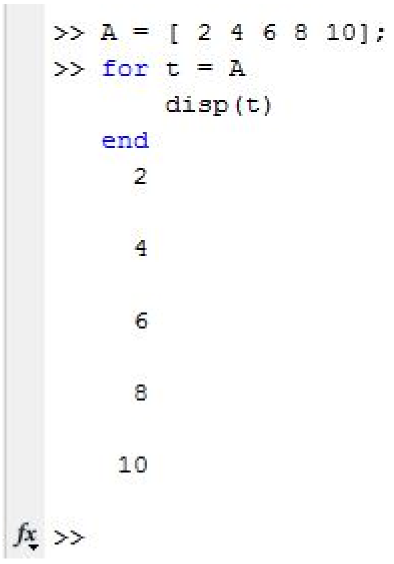
Figure 9: An example of a Switch Statement

first condition is found to be false, whatever code beneath the else statement, known as the body, is executed. The function 'rem' returns the remainder of it's first argument divided by its second. The function 'randi' returns random numbers from a uniform distribution of the first argument in a square matrix with the size of the second parameter. The two equal signs is the equality operator, to test if something is not equal you use ~=. This is different from the != operator normally used in most languages. The final form of the if statement is shown in Figure 8. The *elseif* allows for testing of as many possibilties as one feels like typing out. Also, the figure shows the use of both the inequality operator and the not-equal operator.

The if statement is very nice, and you really don't require many other boolean statements, but as a standard, the *switch* statement is useful when you what to match an argument against some known value. This is accomplished with the case and otherwise commands shown in Figure 9. The example uses a hard coded value of 'Cool!' as the variable to switch on, but normally one would find a switch statement in a function that acts different depending on a key parameter or parameters passed to it. You can switch on any variable so long as it's a single variable and can be matched against. Also, one of the interesting things is that a single case statement can contain multiple matches! This is accomplished through curly braces {} with each case seperated by commas.

5 Looping

There are two kinds of loops in Matlab, the for loop, which repeats a group of statements a specified number of times, and the while loop which continues executing while it's conditional is true. All loops rely on a conditional statement to determine whether or not they can continue running. The syntax for a for loop is the following:



```
>> A = [ 2 4 6 8 10];
>> for t = A
       disp(t)
   end
     2

     4

     6

     8

    10

>>
```

Figure 10: The For loop in Matlab functions like a For Each construct

```
for n=1:2:10
      Do Stuff
end
```

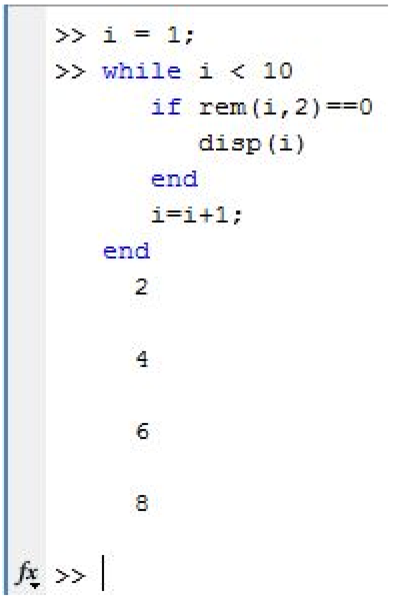
While it may not look like the for is using a conditional, it's really checking to see if the stepping variable, n, is equal to 10 yet. Once it is, we stop executing the statements inside the loop. The for loop is very simple, and easy to use. The range operator enables us to define the array which we will move through. That being said, we can do something rather interesting with the for loop. By using a previously defined array, we can create the construct generally known as a for each loop. Figure 10 shows an example of this.

We begin by creating a vector named A and filling it with a few even numbers up to ten. The synax for the for loop hasn't changed, the range operator creates an array for us to move through, by already having this vector available we skip one step. At the beginning of each iteration of the loop, the variable t is set to be equal to the next value in A. Easy!

The second kind of loop, the while loop, repeats a group of commands until it's conditional is false. There's really nothing else to say about it. As long as the expression passed to the while loop's conditional part evaluates to a boolean, then it will work. You can place boolean expressions, integer comparisons, tests on vectors and matrices, and our next subject, functions, all into the conditional, so long as they evaluate to a yes or no answer. For a small code example that does the same thing as the previous foor loop, see Figure 11.

6 Functions

Frankly put, writing everything in a script is just a bad idea. Especially if you're doing the same bunch of statements over and over and just changing variables. Lucky for us programmers, we know how to write functions. And doing this in Matlab is fairly straightforward. However, there are a couple kinks which I personally find irritating and would like to adress. *You cannot declare multiple functions inside a single m-file* or at least, not in the way a programmer normally uses functions. In matlab, if you include multiple functions in a single m-file, *only the first function will be visible outside of that m-file.* I repeat this because it is important. *Only the first function in an m-file will be visible from outside of that file.* The other functions declared in that file will be treated as local to that m-file and have private access exclusively from that file. This is fine when you're writing helper functions to support a larger construct, but can become annoying when half of your directory is filled with function files.



```
>> i = 1;
>> while i < 10
     if rem(i,2)==0
        disp(i)
     end
     i=i+1;
  end
   2

   4

   6

   8

fx >> |
```

Figure 11: A while loop in matlab displaying even numbers

Lucky for us, there are ways around this strange implementation choice. For those unfamiliar with object oriented programming, this will seem a little bit like black magic, but I was told by a Physic professor once to take anything I could for granted, so I will now. We can create static methods inside of a class and call these methods from an instance of that class. However, to do this your version of Matlab must support the classdef keyword. The following code segment allows this to happen:

```
classdef holder
    methods (Static)
        function res = f1(...)
             Some code here
        end
        function res = f2(...)
            Some more code here
        end
    end
end
```

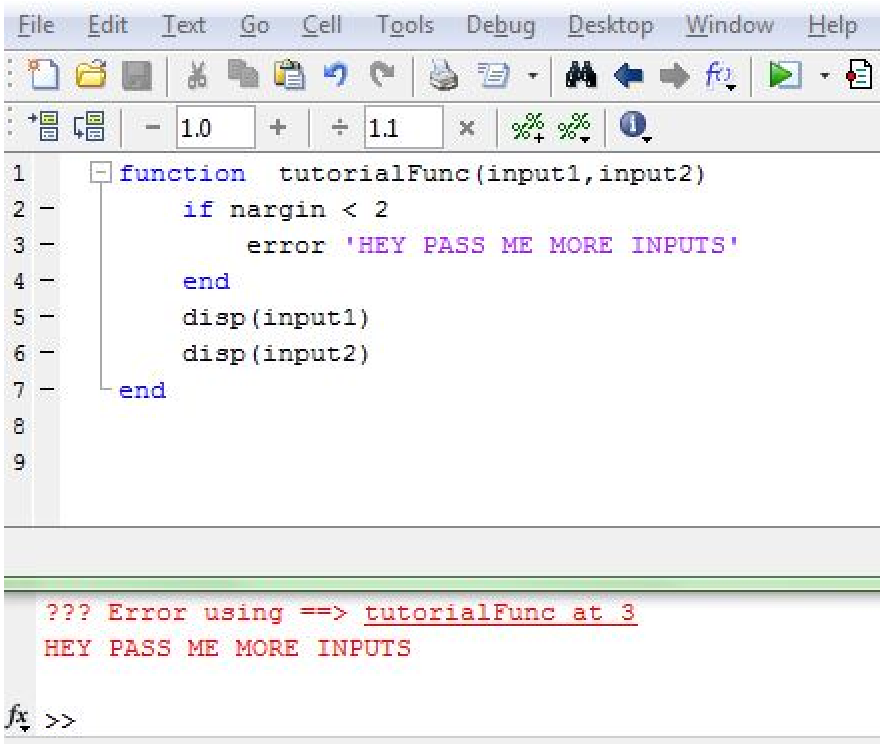
Strange oddities aside, let's get at the syntax of the function first. The general syntax is this:

function [output1, output2,...] = functionName(input1,input2,...)

Fairly straightforward, the function keyword at the beginning states that we'll be creating a new functio and we then follow it by it's output arguments. If there's more than one output we have to place them in brackets. After the equals sign we specify the name for the function[4] and list off the parameters the function requires to operate on. I don't need to stress the importance of good descriptive names for functions and parameters/arguments, it should be pretty clear to you when you make the code, and when you look at it three months down the line what you were doing.

A good command to know when creating functions is the built-in function *nargin*. This strange looking word stands for number of arguments in, and stores the number of arguments actually passed to your functions when you execute them. This is extremely useful for debugging purposes, as you can check for inproper function calls and display an error message appropriately. An example function and use of these two commands is shown in Figure 12.

[4]This name should be less than your Matlabs built in variable namelengthmax, which is normally 63



```
function  tutorialFunc(input1,input2)
    if nargin < 2
        error 'HEY PASS ME MORE INPUTS'
    end
    disp(input1)
    disp(input2)
end
```

```
??? Error using ==> tutorialFunc at 3
HEY PASS ME MORE INPUTS

>>
```

Figure 12: An example of nargin & functions

Notice that I didn't specify any output arguments. If your function doesn't return any outputs for some reason, then you shouldn't try to include them somehow, if it doesn't make sense, don't do it. The function tutorialFunc takes two inputs as parameters, since Matlab is weakly typed[5] we don't have to specify the type of the value we're passing to the function. On the second line, we use the function nargin to determine if the proper number of inputs has been passed to the function, if for some reason this doesn't occur, then we raise an error using the error function with a very informative message to the use. Besides this, nothing is new, lines 2-6 account for the body of the function, and despite this being the only function in the file, I close the function with an end for the sake of it. Note that if you have multiple functions declared in a file (keeping in mind about what the first paragraph of this section adressed) that you *must* end each functions with an end statement. If you only have a single function declared, then it's fairly normal not to end the file with an end.

6.1 Anonymous Functions

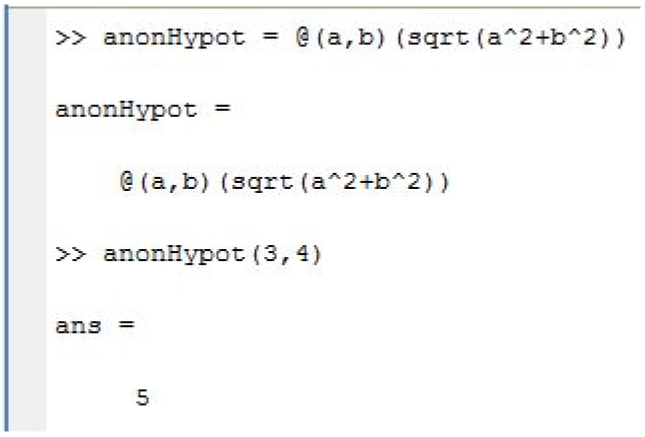
For the more advanced programmer, Matlab fully supports anonymous functions in a similar way to Python. To create an anonymous function one simply constructs a function handler to an expression. This expression should be one line, and a generous use of parentheses is recommended[6]. The syntax for creating a function handler is shown in Figure 13. The function created computers the hypotenuse of a right triangle given the other two sides a and b.

Anonymous functions can return outputs as well, although this can be trick sometimes. I've seen two common ways to deal with multiple outputs from an anonymous function. Here are two code examples:

```
f = @(x)deal(x+2,x+3)      %Using Deal
g = @(x)[x+2,x+3]          %Using a vector
```

[5]Discussed in the Key Differences section, the input parameters types will be determined and evaluated at runtime.

[6]Matlab's online documentation states the space characters within the expression can be ambiguous for Matlab to read, and this could cause problems that might not be obvious during runtime.



```
>> anonHypot = @(a,b)(sqrt(a^2+b^2))

anonHypot =

    @(a,b)(sqrt(a^2+b^2))

>> anonHypot(3,4)

ans =

     5
```

Figure 13: Anonymous Function Example

Both functions do the exact same thing, however how they return their arguments is completely different. The first function, f, uses the built in function deal to send what it recieves as input to outputs. There is a catch to using this method though. When calling function f, you *must* assign f to the same number of variables as deal recieves. If you fail to do this, you will get an error referencing the deal function.

The second anonymous function, g, returns a vector of the two arguments. This will work if you call the function with something like, a=g, but not [a,b]=g. Why? Because the function isn't neccesarily returning two arguments, just a vector of the arguments, because of this, different types of outputs will cause problems, and this is why you should probably document your code properly and make sure that you use the deal function correctly. If you know you'll be returning only integer types, then this vector form will work pretty well, and your code will just have to adapt to fit it.

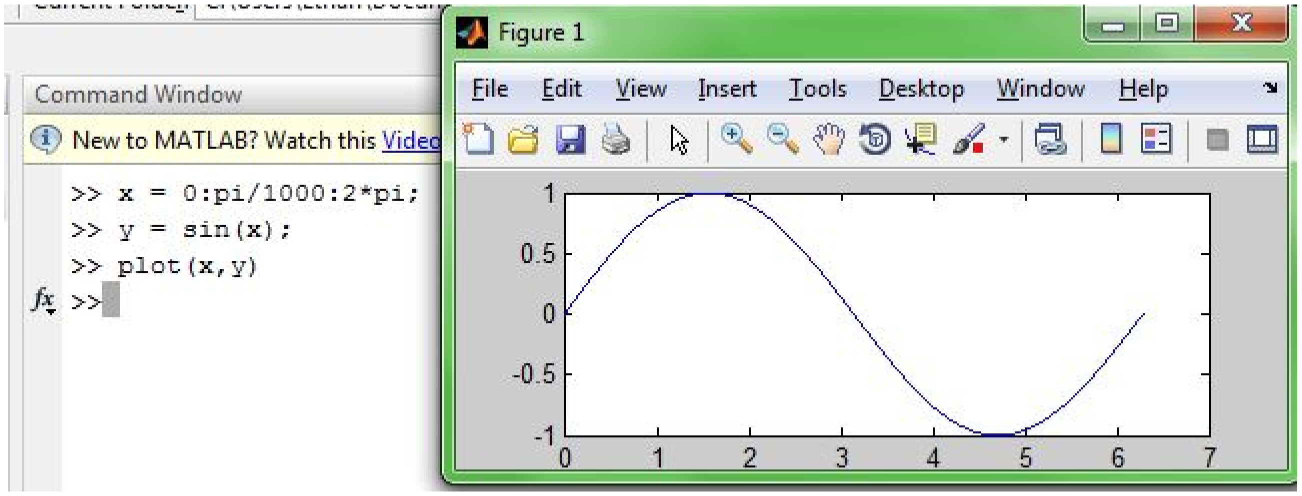
The last note that I'd like to say on anonymous functions regards the state of variables. anonymous function can use local variables inside of their body. *And continue using them after those variables have gone out of scope.* How? Because the variable that's stored inside of the anonymous functions body is a copy of the value of the variable used. This means that state effects like declaring an integer, using the integer in an anonymous function, and changing the integer will *not* change the value used by the function. This should make functional programmers happy, and those with less caring programming conventions sigh. This means that if you're using an anonymous for say, graphing an equation, you must redeclare the function if you want to change the value of some constant in the function, or declare the function as a parameter to fplot.[7]

7 General Plotting

Let's face it, one of Matlabs primary purposes in a lot of applications is to display a lot of data in some visual format, and as such, supports a rediculous multitude of built-in functions. The easiest way to use this functionality is by right clicking variables in the workspace and selecting the plot catalog. However, as coders, we like to automate things and not do them ourselves, even if it means we have to do a little bit more work then right clicking. An easy function to use is the plot command. This function takes two same-sized vectors to plot against one another. For example, to display a graph of the sine wave from zero to 2π, we can use the code segment in Figure 14 to produce the shown plot;

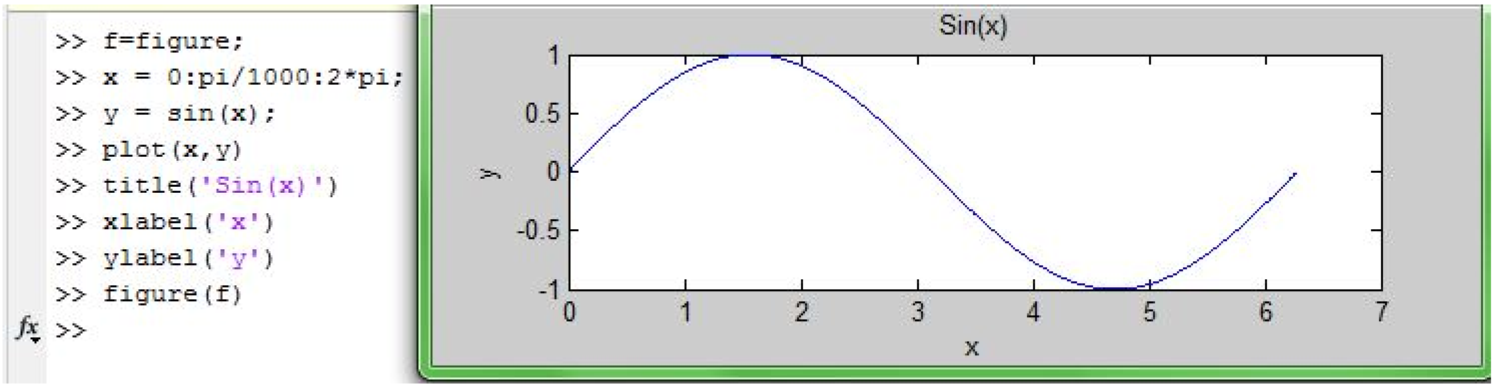
Despite how very nice this plot is, we could use some labeling to add information to the plot. If we have just executed the shown code, then we can input the next set of commands without worrying, if it's been a while and something may have created a new plot, we need to make sure we can get back to the figure we were editing before. We can do this by assigning the output of the command figure to a

[7]Mathworks has a nice example of this on their help documentation for examples of anonymous functions, declare integers a,b, and c and type fplot(@(x) a*x.^2 + b*x + c, [-25 25]) to see how it works.


Command Window
New to MATLAB? Watch this Video
Figure 1
File Edit View Insert Tools Desktop Window Help

```
>> x = 0:pi/1000:2*pi;
>> y = sin(x);
>> plot(x,y)
>>
```

Figure 14: An Example of Basic Plotting in Matlab



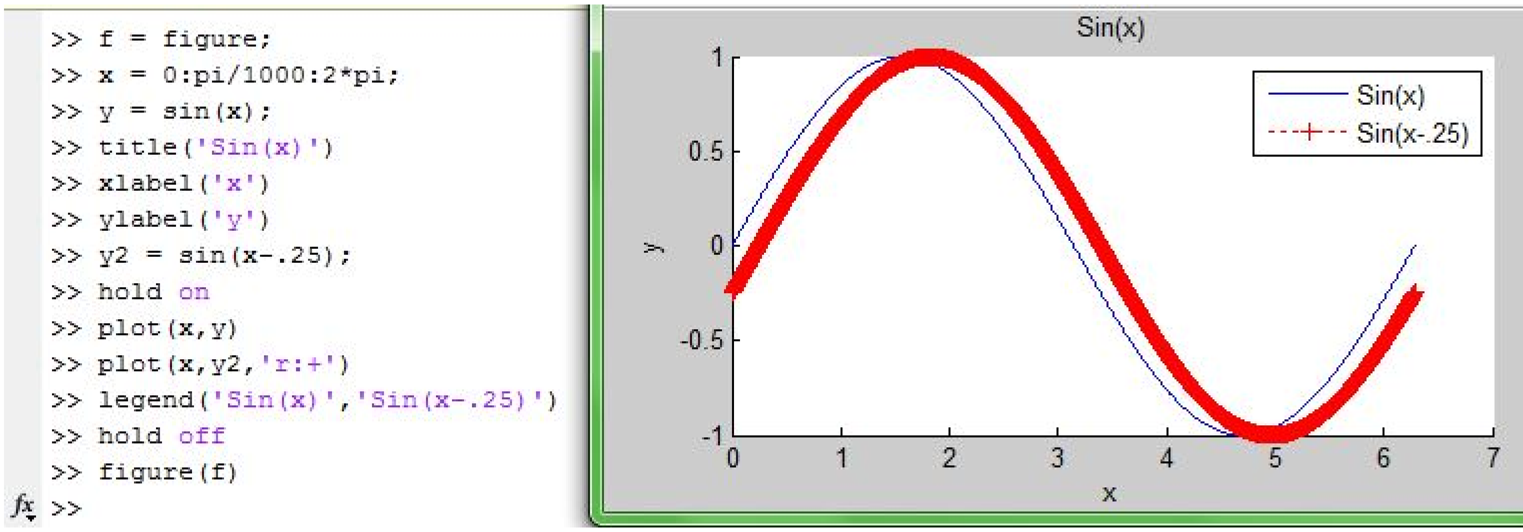
```
>> f=figure;
>> x = 0:pi/1000:2*pi;
>> y = sin(x);
>> plot(x,y)
>> title('Sin(x)')
>> xlabel('x')
>> ylabel('y')
>> figure(f)
>>
```

Figure 15: Passing the plot function multiple pieces of data

variable, and whenever we want to make sure we're using that figure, we pass it to the figure function again. This is made clear in Figure 14. A few other commands are pretty self explanatory and are shown in the figure as well.

- title creates the title Sin(x) on top of the displayed graph
- xlabel creates the label along the x axis
- ylabel creates the label along the y axis

There are a couple ways of showing multiple plots on a single figure. You can call the plot command, passing in pairs of vectors, and allow Matlab to handle coloring and such for you, or you can use the *hold on* command to cause all plotting to be done on the current figure until you specify *hold off*. This can be useful if you have to manipulate the data in some way before plotting it, and you don't feel like placing giant lines of vectorized code into a single plot function call. Figure 15 shows an example of this first way of doing things, and Figure **??** shows how to use the hold command. Notice that in the latter, we had to specify how to format the line for the second plot command. If we had not done this, we would have ended up with two blue lines and wouldn't have been able to tell the diference. The format "r:+" creates a dotted line and places plus signs along each datapoint, the reason the line is so thick in the image is because we have 1000 points clustered close together. To clear the plot, you can use the *clf* command to remove most properties from the figure, to clear the background and any other changes that you might make, you can use *clf reset.*



```
>> f = figure;
>> x = 0:pi/1000:2*pi;
>> y = sin(x);
>> title('Sin(x)')
>> xlabel('x')
>> ylabel('y')
>> y2 = sin(x-.25);
>> hold on
>> plot(x,y)
>> plot(x,y2,'r:+')
>> legend('Sin(x)','Sin(x-.25)')
>> hold off
>> figure(f)
>>
```

Figure 16: Using hold to plot multiple functions

8 Wrapping Up

Hopefully, this rather brief exhibition of code was enough to get you started in Matlab, and answer common questions. As mentioned previously, this is a two part Workshop document, and thus the next session will cover other concepts, with the assumption that one is modestly familiar with Matlab. The next session will cover vectorization, animation, matrix manipulation, structures in MATLAB, and GUIS.