Name: Eng Sci 53
Date: Lab 1
TF:

Matlab Tutorial 1

What is Matlab? It's a combination of a simple calculator, a plotting program, a data analysis tool, a programming language, and much, much more – and it does all of these pretty well! Nowadays, it's probably the most commonly used programming language in the pure and applied sciences. Learning how to use Matlab early on in your college career will not only give you a really good tool to deal with the data analysis demands you'll encounter while here at Harvard, but it will also prepare you well for wherever you go afterwards. The two Matlab tutorials we'll be doing in this lab section will hopefully be a clear and concise walk-through for some of the basics of the program.

Part 1 – Personal folder

Create a folder with your initials in the ES53 folder.

Part 2 – Matlab windows

Open up Matlab and notice the different windows:

Command Window – Where you enter in commands, such as vector definitions, functions on those variables, etc.

Workspace – Every time you create a vector or variable, Matlab stores that in the workspace, and as long as that variable is in the workspace, you can access it, manipulate, etc.

Command History – Self-explanatory

Part 3 – Creating Vectors

Begin by creating a vector, called *x*, of all the integers from 1 to 10.

```
>> x = 1:1:10
```

'x' is the variable name, '=' signifies assignment of value, and '1:1:10' is *beginning_value:increment_value:end_value*. (For example, w = 1:2:10 assigns the vector [1 3 5 7 9] to w, and v = 1:3:10 assigns [1 4 7 10] to v.)

A strength of Matlab is that through this vector notation / array defining, programmers can avoid using lengthy "for-loops" to define variables. One alternative way, using those for loops, to define *x* would be:

```
>> for i = 1:10
     x(i) = i
   end
```

It gets the job done, but it's using the same vector notation as before to increment 'i', so we might as well use the vector notation for 'x' to save space. There are also times where "for-loops" are necessary, as we will investigate later.

Notice how when we defined 'x' above, we got the following output:

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

'x' here is defined as a row vector – this is the default form. To define a column vector instead (which might be necessary for certain functions), all you need to do is stick an apostrophe on it, which transposes it.

```
>> x = (1:1:10)'

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

What you can also do is to transpose it AFTER it's been defined as a row vector.

```
>> x = 1:1:10

x =
     1    2    3    4    5    6    7    8    9    10

>> x = x'

x =
     1
     2
     3
     4
     5
     6
     7
     8
```

```
     9
    10
```

Normally, we don't want Matlab to output our variable assignments because if we're running a script and we're defining thousands of variables, having to print out all of the values on the screen can slow the program down considerably. A simple way to suppress the output is to stick a semi-colon at the end of the variable assignment.

```
>> x = 1:1:10;
```

Situations when you *do* want Matlab to output the values of the variables is if you're using Matlab as a glorified calculator, or if you're debugging a program, and…pretty much, that's it. There might be some other uses for it, but they won't come up that often. If you want to display the value of a previously assigned variable *x*, you can do it any time just by typing "x" in the command window.

Another item of note – the default increment in vector notation is 1, so you can also define 'x' as

```
>> x = 1:10;
```

Now, let's say we want to define our vector from 10 down to 1. We first try

```
>> x = 10:1;
```

and get

```
x =

   Empty matrix: 1-by-0
```

Remember, the default increment is +1, so of course, there's no way to get from 10 to 1 using only positive increments. How might we define the vector [10 9 .. 1] using vector notation?

```
>> x = ____________________ (?)
```

As well, we can use the function fliplr to help us out. Type

```
>> help fliplr
```

In general, 'help' is an extremely useful command that you'll be using a lot. The normal format is to type '`help ###`', where '`###`' is a command of some sort. In order to see what commands there are, you can press F1 and a prototypical Help window will pop up, where you can search for functions, etc.

How would we use `fliplr` to define the vector [10 9 .. 1]?

```
>> x = fliplr(_________) (?)
```

Part 4 – Vector algebra

First, define the vector 'x' as [4 4.5 5 5.5 6 ... 13.5]. (Where's the starting point, what's the increment, where's the end point?

```
>> x = __________ (?)
```

Next, define the vector 'y', starting from -4, going to -23, with 20 total entries. (Hint: type '`help linspace`').

```
>> y = linspace(____________) (?)
```

Now, let's find the inner product of these two vectors.

```
>> x*y

??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Hmm…so apparently that didn't work. Why's that? Well, Matlab wants to do matrix multiplication. (Matlab stands for "MATrix LABoratory") And when we do matrix multiplication, the inner dimensions of the matrices must agree; for instance, we could do (1-by-3 vector)*(3-by-2 vector), or (2-by-5 vector)*(5-by-1 vector).

So let's check the dimensions of these vectors. ('`help size`')

```
>> size(x)
>> size(y)
```

We see that both of them are row vectors, with size 1x20. Obviously, we can't multiply two row vectors together, right? We can't do (1-by-20)*(1-by-20). So what do we want to do? We need to transpose one of them to make it a column vector, so that we do (1-by-20)*(20-by-1).

```
>> x*(y')
```

You should get -2695 as the answer. Now, what if we wanted to multiply the i^{th} elements of the two vectors together? We *could* do this by:

```
>> z = [];
>> for i = 1:length(x)
     z(i) = x(i) * y(i);
   end
```

But that's pretty tedious. Matlab has a shortcut notation to do this element-by-element multiplication.

```
>> x.*y
```

Note the period right before the multiplication sign. Whenever you want to do this element-by-element editing, you *have* to include this period or else it won't work. It goes in front of '*', '/', and '^'.

Part 5 – Matrix algebra

Define matrix A:

```
>> A = [[2 6]; [4 8]]

A =
     2     6
     4     8
```

Define matrix B:

```
>> x = [4 -1];
>> y = [-4 1];
>> B = [x ; y];
```

Work out by hand what A*B should be (do you all remember how to do this?)

`A*B =` _____ (`?` – should be a 2x2 matrix).

Check your answers with Matlab.

Now, how would we multiply the (i^{th}, j^{th}) elements of each matrix together?

```
>> A ___ B (?)
```

Part 6 – Plotting

Plotting is an extremely powerful feature of Matlab. One item to note first: Matlab DOES NOT plot functions – it plots data points. If we were to plot y = x^2 on our graphing calculators, we could just enter y = x^2, and we would get a parabola. However, for Matlab, we would need to create a vector 'x', and then a vector 'y' that was defined as 'x.^2'. Note that I put a period before the hat preceding the 2 – that is important, as you will see later on.

For this section, we want to:
1) plot a horizontal line y = 0 in the range x = 0:5
2) plot a line with slope of 1 going through (6,0) in the range x=6:10, and

3) plot another horizontal line y = 4 in the range x=11:15.

We first need to define each of these segments. To clear the workspace, type "`clear all`". (By the way, if you want to access the last command entered, you can double click on it in the Command History, or you can press the up-arrow-key.)

```
>> x1 = _____________ (?)
>> x2 = _____________ (?)
>> x3 = _____________ (?)
```

It might be helpful to investigate the use of '`zeros`' and '`ones`'.

```
>> y1 = _____________ (?)
>> y2 = _____________ (?)
>> y3 = _____________ (?)
```

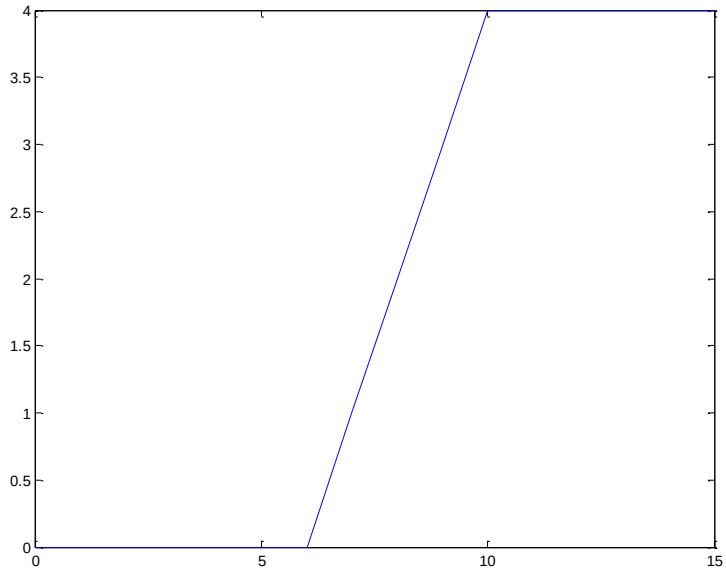
Once we've defined these segments, we can append them together.

```
>> x_tot = [x1 x2 x3];
>> y_tot = [y1 y2 y3];
```

If there are error messages, check the dimensions of the variables to make sure they can be concatenated (concatenate means to put together).

Let's plot this vector. (`help plot`)

```
>> plot(x_tot, y_tot)
```



We can plot different colors:

```
>> plot(x_tot, y_tot, 'g');
```

```
>> plot(x_tot, y_tot, 'r');
```

With different marker shapes:

```
>> plot(x_tot, y_tot, 'go');
>> plot(x_tot, y_tot, 'gx');
```

With different markers and a line:

```
>> plot(x_tot, y_tot, 'g-o');
>> plot(x_tot, y_tot, 'g-x');
```

And with different line widths:

```
>> plot(x_tot, y_tot, 'g-x', 'linewidth', 4);
```

Now, plot it in blue, with the data points marked by little circles while still having the line, and Line Width of 2.

```
>> plot(____, _____, _____, _____, ____) (?)
```

Note how the lines from x=0:5 and 11:15 are hard to see because they are right on the edge of the graph. In the figure, click on the little hand and move the graph around to better see the lines. As well, you can zoom in, zoom out, and select individual data points.

We can also adjust the scale of the graph beforehand. Set the x-range to -1:16, the y-range to -1:5, label the x- and y-axis, give a title, and turn the grid on.
(`help xlim, ylim, xlabel, ylabel, title, grid`).

Note that you can also plot log scales using `semilogx` (which has the x-axis as the log scale), `semilogy`, and `loglog` (both axes are log scales). See Matlab\graphics in the help window for very good explanations of many of the image-related features.

Part 7 – Save and load

Importantly, we can save our data (variables) and then load them later.

Let's save our variables into a dat file that we'll call my_var:

```
>> save('my_var', 'x_tot');
```

Alternatively, we could save all of our current variables into a dat file:

```
>> save('all_var');
```

As long as this dat file is in the current directory, from now on we can import the variables back in:

```
>> load('my_var');
>> load('all_var');
```

Part 8 – Scripts

Up to this point, we've been typing all of our commands into the command window. While this works, if we want to do multiple lines of code, it isn't time efficient to have to retype or paste in these lines of code each time we want to run them. Matlab allows us to create scripts that it will run just as if we were typing the lines into the command prompt, except that they're a separate file which we can edit at our leisure.

Go to 'File', 'New', 'M-File'.

An M-file editor should now be open on your screen. The first step is to save the script as 'tutorial_1###.m', where "###" are your initials. IF YOU ARE SAVING IT ON ONE OF THE LAB COMPUTERS, MAKE SURE TO USE YOUR OWN DIRECTORY!!

Now, each line in this script represents a new entry in the command prompt. With this understanding, we want to write a script that plots a sine wave and a cosine wave on the same graph in two ways.

1st Method

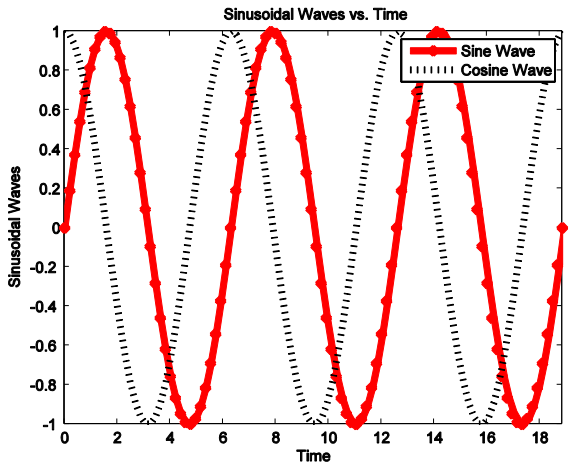
Define a time vector from 0 to 6pi. (hint: use `linspace`) with 100 elements.
Define a sine vector and a cosine vector over that time interval.

Plot both of them using one plot function, with the following attributes:

Sine wave – red line with 'x' markers, line width of 4,
Cosine wave – black dashed line, line width of 4

```
plot(_______,_______,_______,_____,_______,_______,_______,_______)
```

Fully label this plot (`xlabel, ylabel, title, legend`)
Set appropriate axis limits
To run the script, save it again and then press F5 or go to "Debug" in the menu bar and click on 'Run'



2nd Method

Define a time vector, t1, from 0 to 6pi with 100 elements.
Define a time vector, t2, from 0 to 6pi with 110 elements.

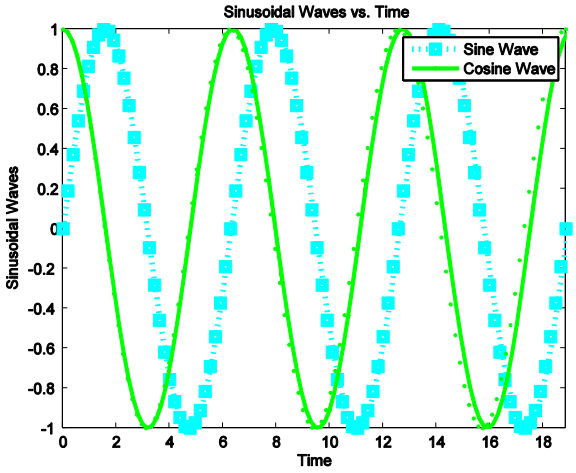
Define the sine wave with t1 and the cosine wave with t2.

Plot both of them on the same graph (hint: because the time vectors are different sizes, you can't plot them in the same `plot` function - you'll have to use '`hold on`').

New attributes:

Sine wave – cyan dotted line with square markers, line width of 3
Cosine wave – green dash dot line with pentagram markers, line width of 2

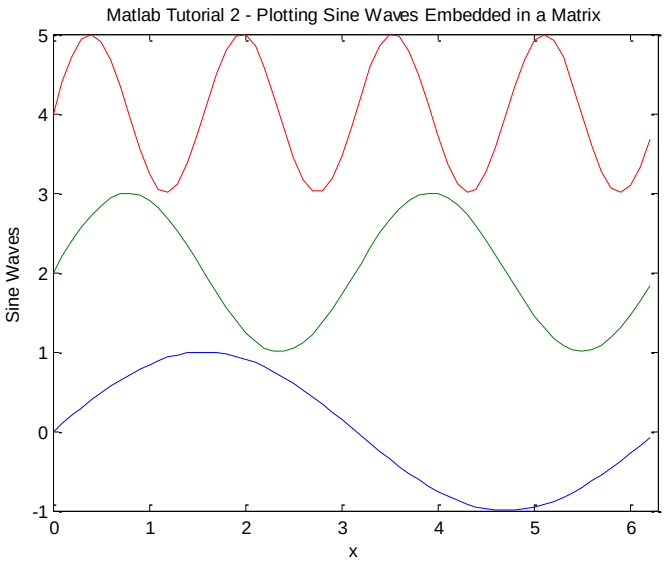
Again, fully label and set appropriate axis limits.



Part 9 – Plotting vectors and matrices

Using column or row vectors makes a difference in linear algebra operations, such as multiplying two vectors or a vector and a matrix (parts 4 and 5). When plotting, we can get different results depending on whether we use one form or the other.

Up to this point, we've been plotting vectors. Plotting a row vector on x versus a column vector on y or the other way round doesn't make any difference for Matlab. What happens when we need to plot multiple vectors in a single plot, like in the following graph?



To investigate this, first create three row vectors representing sine waves with the following attributes:

Sine Wave 1: offset of 0, frequency of 1
Sine Wave 2: offset of 2, frequency of 2
Sine Wave 3: offset of 4, frequency of 4

To start, try concatenating the three vectors in a row, like this: `[s_1 s_2 s_3]`. Plot this vector. Not exactly what we wanted, right? Of course, we could instead plot each wave with a separate `plot` command, and use `hold on`. There is an easier way, however. Matlab can also plot matrices, and does this by plotting each *column* as a different line. Check this: create a matrix with three columns, with the columns representing one of the above sine waves – remember you need to convert the sine waves in column form first!

Label and adjust axes accordingly – the end result is shown in the previous graph. Note how Matlab automatically assigns the different colors.

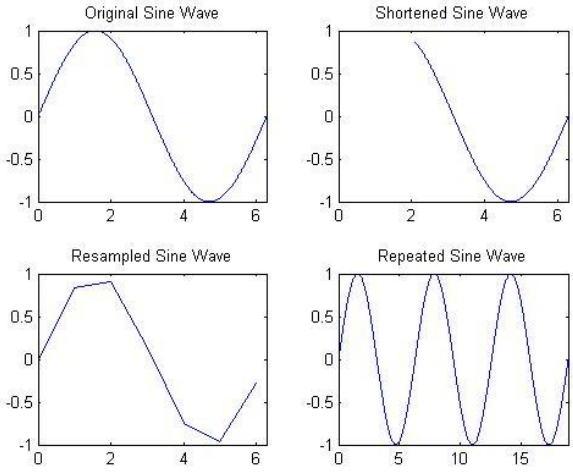
Part 10 – Matrix Manipulation

Oftentimes, we'll want to manipulate vectors or matrices in certain ways to aid in computation, such as shortening them (maybe only a small part of the vector is of interest), resampling them (perhaps there are more data points than we need, and so we want to reduce the resolution a bit to make things more manageable), or repeating them (this comes in handy when we're fitting a model to a set of data).

Check out 'help colon' to learn ways to use the colon to manipulate matrix data.

Create a script titled 'matrix_manip' that plots a figure with four subplots. (type `help subplot`) The first subplot is a sine wave from 0 to 2*pi with increment of 0.01. The second subplot should be a picture of the latter 2/3rds of the sine wave. The third subplot should be a resampling of the original wave (take every 100th

point and plot that). The fourth subplot should be repeating the initial sine wave three times.



Save your 'tutorial_1###.m' and "matrix_manip.m" scripts. You can copy, paste, print, or write them down by hand, and submit them to your lab TF.

Part 11 – Nice Shortcuts

In the Command Window, if you press the up-arrow-key, it will pull up the last command entered there.

In the Command Window, if you start typing a variable or function name, before you complete it, you can press 'tab' and it will give you a list of potential names that you can choose from, or if there's only one option, it will automatically fill that in for you. For example, if you type in 'tutor' and press tab, it'll fill in 'tutorial_1.'

Ctrl-R comments the whole line.
Ctrl-T uncomments the whole line.
Ctrl-I adjusts the indentation to the proper position.

As always you can use '`help`' to learn more about the following commands.
'`figure`' creates a figure window
'`close`' closes a figure window
'`clear`' clears variables and functions from the workspace
'`shg`' brings the current figure window forward.
'`why`' gives answers to life's questions.

F5 – Runs the script or program you have open in the M-file editor.
F9 – Runs the script that is currently selected inside the M-file editor.

MAKE SURE TO TURN THIS TUTORIAL IN WITH ALL OF YOUR GROUP MEMBERS' NAMES BEFORE YOU LEAVE!