MATLAB Tutorial

EEO303

Prof. Fowler

We will be using MATLAB in EEO303 to illustrate how to perform DSP operations.

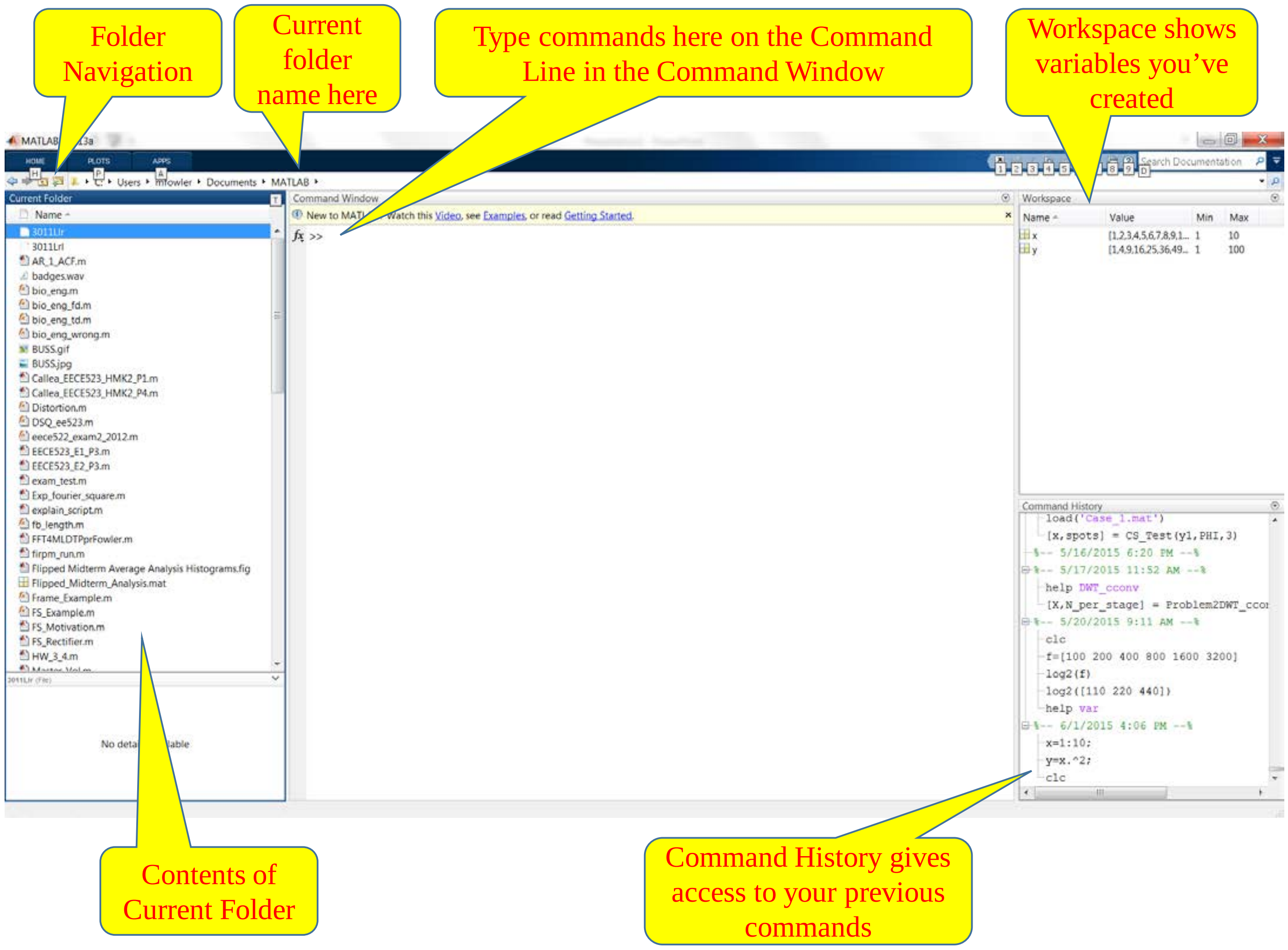
Be aware that there are lots of parts of MATLAB that we won't need, so if you run across something that sounds unfamiliar don't worry about it (for example, there are lots of matrix commands such eig and svd that we won't be needing; there are also lots of fancy plotting functions that we won't need, either).

Before going through the tutorial I give below you should check out the "tutorials" and "videos" available here:

http://www.mathworks.com/help/matlab/getting-started-with-matlab.html

<u>MATLAB Desktop</u>

When you start up MATLAB it should come up in the default "Desktop" setting, which is shown below:


Folder Navigation
Current folder name here
Type commands here on the Command Line in the Command Window
Workspace shows variables you've created
Contents of Current Folder
Command History gives access to your previous commands
HOME
PLOTS
APPS
Users ▸ mfowler ▸ Documents ▸ MATLAB ▸
Current Folder
Command Window
Workspace
Command History
Search Documentation
New to MATLAB? Watch this Video, see Examples, or read Getting Started.
No details available

```
3011Llr
3011Lrl
AR_1_ACF.m
badges.wav
bio_eng.m
bio_eng_fd.m
bio_eng_td.m
bio_eng_wrong.m
BUSS.gif
BUSS.jpg
Callea_EECE523_HMK2_P1.m
Callea_EECE523_HMK2_P4.m
Distortion.m
DSQ_ee523.m
eece522_exam2_2012.m
EECE523_E1_P3.m
EECE523_E2_P3.m
exam_test.m
Exp_fourier_square.m
explain_script.m
fb_length.m
FFT4MLDTPprFowler.m
firpm_run.m
Flipped Midterm Average Analysis Histograms.fig
Flipped_Midterm_Analysis.mat
Frame_Example.m
FS_Example.m
FS_Motivation.m
FS_Rectifier.m
HW_3_4.m
```

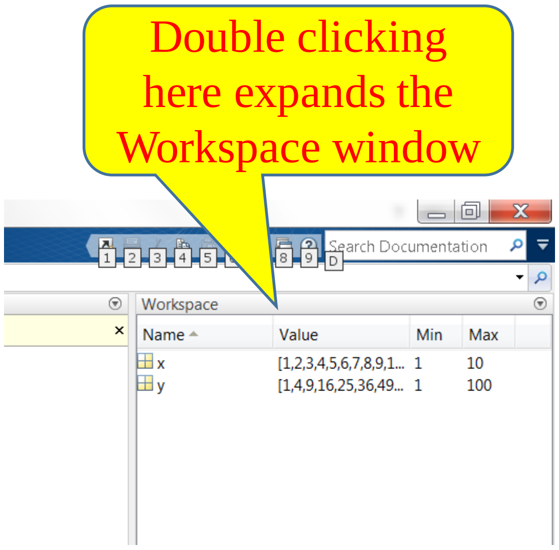
```
fx >>
```

Name	Value	Min	Max
x	[1,2,3,4,5,6,7,8,9,1...	1	10
y	[1,4,9,16,25,36,49...	1	100

```
load('Case_1.mat')
[x,spots] = CS_Test(y1,PHI,3)
%-- 5/16/2015 6:20 PM --%
%-- 5/17/2015 11:52 AM --%
help DWT_cconv
[X,N_per_stage] = Problem2DWT_ccor
%-- 5/20/2015 9:11 AM --%
clc
f=[100 200 400 800 1600 3200]
log2(f)
log2([110 220 440])
help var
%-- 6/1/2015 4:06 PM --%
x=1:10;
y=x.^2;
clc
```

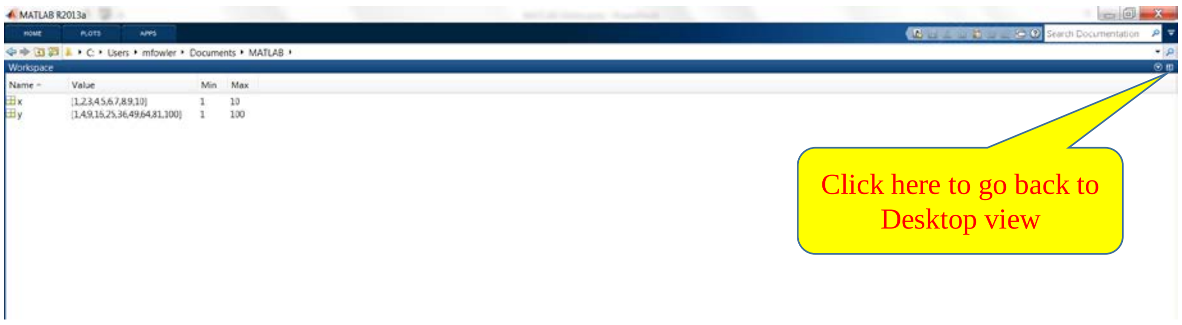
- You can use the folder navigation tools to make sure you are pointing to the desired folder on your computer.
    - This folder is where you should store your MATLAB programs as "m-files".
    - You can open m-files (which are MATLAB programs) by double clicking on them.
- Commands are entered at the command line and the result is displayed after hitting return (unless you put a ";" at the end of the line, which will suppress the output (although computation is completed).
- The Workspace window lists all the variables you've created and gives some details about them such as the values and the min and max values.
    - If the variable holds only a small number of values these values are displayed but for variables that hold many values the display shows the size of the array and the type of variable: (double, complex double, logical, etc.)

Each of the windows in the Desktop can be expanded by double clicking on the bar at the top:



Name	Value	Min	Max
x	[1,2,3,4,5,6,7,8,9,1...	1	10
y	[1,4,9,16,25,36,49...	1	100

The result is shown below… with an indication of where to click to go back to the Desktop view:


MATLAB R2013a
HOME
PLOTS
APPS
C: ▸ Users ▸ mfowler ▸ Documents ▸ MATLAB ▸
Search Documentation
Workspace
Click here to go back to Desktop view

Name	Value	Min	Max
x	[1,2,3,4,5,6,7,8,9,10]	1	10
y	[1,4,9,16,25,36,49,64,81,100]	1	100

More details on the Desktop functionality is given in:

http://www.mathworks.com/support/2015a/matlab/8.5/demos/WorkingInTheDevelopmentEnvironment.html

Using Commands

MATLAB has two ways it can be used: (1) as a command line environment and (2) as a program-running environment; in fact, the two environments can be used together.

Command Line Environment: You issue commands at a prompt and MATLAB returns results; the results can be stored in variables that can be named as desired (within certain limitations).

Program-Running Environment: You call a program from the prompt; the program does a series of commands that have been stored in a so-called “m-file”. An m-file can, if desired, accept multiple inputs variables and return multiple output variables.

In this tutorial handout I will mostly discuss the command-line environment. Commands will be entered in the command window at the command line prompt that looks like this:

```
>>
```

<u>Getting Help</u>

```
>> help command_name
```

will provide help on the various commands. For example,

```
>> help sum
```

provides help on the "sum" command that sums the elements in a vector. There is even help on help:

```
>> help help
```

give help on using help. Also just typing help gives some information useful for finding help on the various types of MATLAB commands there are:

```
>> help
```

gives a list of topics or categories into which MATLAB commands are grouped. Then you can get help on a specific topic by entering:

```
>> help topic_name
```

For example, one of the topics listed when the command "help" is issued is "elfun," which stands for "elementary math functions" so that entering

```
>> help elfun
```

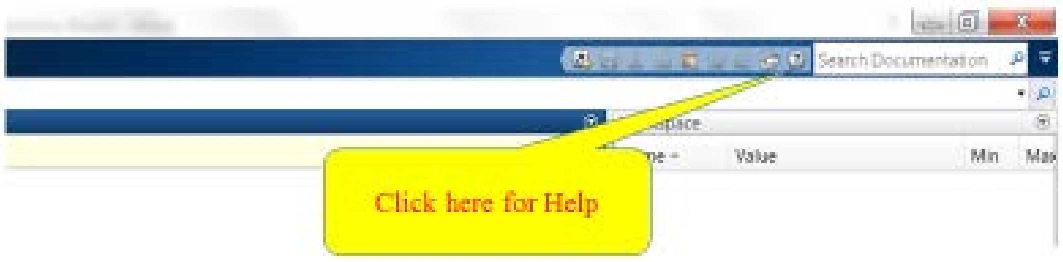
gives a list of trigonometric functions, exponential functions, numeric functions (such as "round" and "sign"), and functions for complex numbers (such as "abs", "angle", "conj", "real", and "imag").

Another way to get help is to use the "lookfor" command. It searches for commands whose help files contain a specified word. For example,

```
>> lookfor cosine
```

provides a list of functions that mention the word "cosine" in their help files.

You can also get help by clicking on the question mark at the top of the Desktop:



MATLAB is Vector/Matrix-Oriented

Much of the power of MATLAB comes from the fact that it can operate simultaneously on all the elements in a vector or matrix (unlike C, which operates only on individual elements at a time).

Create vectors:

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

creates the 1x10 row vector of the first 10 integers and assigned it to the variable. The ; at the end of this command suppressed the result of this assignment . To see the result of the assignment do this:

```
>> x
x =

    1     2     3     4     5     6     7     8     9    10

>> y=1:2:10
y =

    1     3     5     7     9
```

allows the step size to be varied.

Column vectors can also be created. The easiest way is to "transpose" a row vector:

```
>> y=(1:2:10)'

y =

    1
    3
    5
    7
    9
```

Warning: for complex-valued vectors use x.' instead of x' because the later will ***also*** conjugate it:

```
>> y=[1+j*3  2-j*4  -2+j*5]

y =

  1.0000 + 3.0000i   2.0000 - 4.0000i  -2.0000 + 5.0000i

>> y'

ans =

  1.0000 - 3.0000i
  2.0000 + 4.0000i
 -2.0000 - 5.0000i
```

Operate on Vectors:
This is where MATLAB really shines. There is usually no need, as in C, to write loops to operate in vectors and matrices; **in fact, using excessive loops in MATLAB causes the program to run VERY slowly!** Instead, most operations work simultaneously on all elements of the vector or matrix.

```
>> x=1:10;
>> y=1:2:20;
>>z=x+y
z =

    2    5    8   11   14   17   20   23   26   29
```

gives the element-by-element sum of the two vectors x and y.

We can use vectors to represent functions of time:

First define a series of time points in a row vector t:

```
>> t=0:0.01:5;   % creates a vector of times spaced every 1/100th of a second
```

The choice of the spacing depends on the function of t you will be computing. In DSP we often will choose the spacing to be the sampling interval, so sampling theory is often used to determine an appropriate value for the time spacing.

We can now use that time vector to create a function vector by using operations on the vector t. For example, to create a vector y that takes on values of a quadratic function of t we could do:

```
>> y=3*(t.^2) + t + 5;
```

which creates a vector y having the same size as t whose elements are the values of $y(t) = 3t^2 + t + 5$ evaluated at the time points specified in the vector t.

Notice the dot before the ^ (the power sign). That indicates that it operates on the vector t on an element-by-element basis. This is called “array power” in MATLAB. Without the dot it is called “matrix power” and performs a more complicated power operation defined in matrix theory (we won’t need that here). Type “help arith” for more info. This “dot” modifier to imply element-by-element operations also holds for multiplication (e.g., x.*y is element-by-element multiplication of vectors x and y, where as x*y is matrix multiplication) and for division (x./y is element-by-element division; without the dot it is a more complicated matrix mathematics operation that we won’t need here).

ramp function:
$r(t) = t\,u(t)$

Here is another example: to compute $y(t) = \dfrac{r(t)}{2+\sin(5\pi t)}$ we do the following.

```
>> t=-5:0.01:5;      % create a vector of time points from –5 seconds to 5 seconds
>> u=sign(t+eps);        % computes unit step function – the eps variable is the
          % smallest  number that MATLAB can represent; adding it to t here avoids
          % sign(0) for which the result is 0, and instead ensures that u is 1 at t=0.
>> r=t.*u;           % computes the ramp– the u function forces it to zero for neg. t
>> y=r./(2+sin(5*pi*t));
```

Notice the use of the “dot” in the .* and ./ commands… this is essential to get the point-by-point operation that is intended by multiplication and division of functions.

Plotting Vectors:
To create a sinewave of period 2 second:

```
>> t=0:0.01:8;   % creates a vector of times spaced every 1/100th of a second
>> fo=1/2;  % defines a frequency corresponding to the 2 sec period
>> s=sin(2*pi*fo*t);  % computes the value of the sinewave at all the times in t
>> plot(t,s)  % plots the sinewave vs. t; this should open a window for the figure
```

Note: Subsequent plots to an open figure window may require you to click on the figure window to bring it to the front for viewing.

```
>> help plot
```

will give more details in plotting.

When making plots it is ***ESSENTIAL*** to label your axes and show units if necessary.

```
>> xlabel(‘time (sec)’)
>> ylabel(‘y(t)  (V)’)
```

Managing the Workspace:
To find out what variables exist in the current workspace: there is a GUI display that shows what variables are in the current workspace

To erase (or clear) a particular variable from the workspace:

```
>> clear variable_name
```

To clear more than variable:

```
>> clear variable_name_1 variable_name_2 ….  variable_name_N
```

To clear all the variables in the workspace:

```
>> clear
```

To save the workspace for use later:

```
>> save  fname % saves into a file called fname.MAT that is put into the current folder.
```

To later load a saved workspace:

```
>> load  fname  % loads fname.MAT from the current folder.
```

Enter “help general” for more info.

Creating Matrices or Arrays
Matrices (or arrays) can be created several ways. Here are some examples:

1. Enter lines separated by semicolons:

```
>> A=[1 2 3;4 5 6;7 8 9]

A =

    1    2    3
    4    5    6
    7    8    9
```

2. Use MATLAB commands to create commonly used matrices and vectors:

```
>> x=ones(1,5)

x =

    1    1    1    1    1

>> x=ones(2,3)

x =

    1    1    1
    1    1    1


>> x=zeros(3,2)

x =

    0    0
    0    0
    0    0
```

3. Multiply a column vector by a row vector:

```
>> t=0:10

t =

    0    1    2    3    4    5    6    7    8    9    10
```

```
>> m=(1:4)'  % Note: the ‘ after this row vector turns it into a column vector

m =

     1
     2
     3
     4

>> B=m*t

B =

     0     1     2     3     4     5     6     7     8     9    10
     0     2     4     6     8    10    12    14    16    18    20
     0     3     6     9    12    15    18    21    24    27    30
     0     4     8    12    16    20    24    28    32    36    40
```

Note that the element at the intersection of the 2nd row and the 3rd column is the product of the 2nd element of y and the 3rd element of x; this holds in general. We can consider the ith row of this matrix to be the ith element of m multiplied by t.

A common use of matrices is to consider each row to be a different function of t. Thus, the first row of B is the function t, the second row is 2t (because the second element of m is 2), and so on.

For another example, the following puts the function $x(t) = t$ into the first row of A and the function $y(t) = t^2$ into the second row of A:

```
>> t=0:0.01:5;
>> x=t;
>> y=t.^2;
>> A=[x;y];
>> plot(t,A)   % makes two plots: row #1 vs t and row #2 vs t
```

Summation:
MATLAB has a summation command. How it works depends on whether what it works on is a vector or a matrix:

Given x, y, and A as defined above:

```
>> sum(x)  %  sums of all the elements x (regardless if row or column vector)
>> z=sum(A)  % sums down the columns of matrix A
```

For example matrix A above we view the result of sum(A) as summing together x(t) and y(t) on a time-point by time-point basis. Thus the row vector z contains values of the polynomial $z(t) = t^2 + t$

Indexing (also called subscripts):

We use indexing to either extract certain elements from a vector or matrix or to insert values into certain places in a vector or matrix.

Extraction:
When we have a matrix or vector it is often needed to extract some of its elements. For example, we may want to extract the element at the intersection of the 3rd row and the 1st column. Or we may want to extract its 2nd row. Or its 3rd column.

For example:
```
>> x=1:3
>> y=(1:4)'  % Note: the ' after this row vector turns it into a column vector
>> B=y*x

B =

    1    2    3
    2    4    6
    3    6    9
    4    8   12

>> z=B(2,:);   % extracts the second row of B
z =

    2    4    6
```

Note that a single number (e..g., 2) is used to specify the specific row of interest, where as the symbol : is used to specify that we want the elements in that row from all the columns. If we only wanted the first two elements in that row we would use this:

```
>> z=B(2,1:2);   % extracts the second row of B
```

```
z =

     2     4
```

If we want to extract an entire column we do this:

```
>> z=B(:,3);   % extracts the third column of B
z =

     3
     6
     9
    12
```

We can extract elements from a vector:

```
>> x=1:2:20
x =

     1     3     5     7     9    11    13    15    17    19

>> x(1:5)   % extracts the first 5 elements
ans =

     1     3     5     7     9
>> x([1:2  7:8])  % extracts the 1st, 2nd, 7th, and 8th elements
ans =

     1     3    13    15
```

<u>Insertion</u>:
It is also possible to use indexing for insertion rather than extraction.

```
>> B
B =

     1     2     3
     2     4     6
     3     6     9
     4     8    12

>> B(1:2,:)=[21 22 23;44 45 46]
B =

    21    22    23
    44    45    46
     3     6     9
     4     8    12

>> B(3:4,2:3)=[101 102;205 210]
B =

    21    22    23
    44    45    46
     3   101   102
     4   205   210
```

Looping:
It is possible, although not always necessary due to the vector operations of MATLAB, to perform loops in MATLAB programs or even at the command line.   Things that are often done with loops can many times be done using the vector operations; however, sometimes loops are needed or convenient. Here is a simple loop that shows how they are done (after which the same thing is done without loops):

```
>> N=10;
>> for n=1:N
X(n)=5/(pi*n^2)
end
>>
```

Note that after entering the "for" command that the prompt goes away until the corresponding "end" command is entered.

This computes a vector X whose ten elements are given by $5/\pi n^2$ for n=1,2,3,…,10.

The same thing could be done using the following:

```
>> N=10;
>> n=1:N;
>> X=5./(pi*n.^2);
```

which is interpreted as follows: form a vector n whose elements are the integers 1 to 10; in the denominator of the last line the elements of this vector are each squared (element-by-element squaring due to the dot before the ^2); each element of this result is multiplied by pi; and then each element of that result is divided into 5 (element-by-element division due to the dot before the / for division).

Usually it is desirable to avoid loops if possible… but sometimes you just have to use a loop!

A Signals & Systems Example[1]

Say we wish to compute the signal that consists of the sum of 10 sinusoidal terms each with some specified amplitude. Say the signal is given by:

$$x(t) = \sum_{n=1}^{10} \frac{-5}{\pi n^2} \cos(2\pi n f_o t)$$

The nth sinusoid has a frequency of $n\,f_o$

where f_o = 500 Hz and we wish to show the signal for 10 ms.

We need to create a vector of time instants that are spaced closely enough that everything looks smooth when we plot it. To ensure this we need to make sure that the fastest changing term in this sum looks smooth… if we do that the slower changing ones will also look smooth. The fastest one has a frequency of $10f_o$ = 5 kHz and that means that its period is 1/5000 = 0.2 ms. To make a smooth plot of this term we'll make sure that we get 10 time points per 0.2 ms period of this 5000 Hz sinusoid – thus we need a spacing of (0.2 ms)/(10 samples) = 0.02 ms/sample. (If you've studied sampling theory already that means that we are oversampling this highest frequency term by a factor of 5 since our sampling frequency is 1/0.02ms = 50,000 Hz and is 5 times twice the highest frequency.)

This is done as follows:

```
>> del_t = 0.02e-3
>> t=0:del_t:10e-3; % creates a 1 x 501 row vector of time points
>> n=(1:10).'; % creates a 10x1 column vector of integers 1 to 10
>> COS=cos((n*t)*2*pi*500);
```

This here is tricky. In the last line the term n*t is a column vector (n) times a row vector (t); note that there is no dot here in front of the * so it is NOT element-by-element, but rather is treated as a matrix multiply: experiment with various column*row to see how this works. In this case n*t gives a 10x501 matrix with the first row being 1*t, the second row being 2*t, the third row being 3*t, …. The 1oth row being 10*t. Multiplying this n*t matrix by 2*pi*500 gives a 10x501 matrix whose rows are the 10 different arguments for the cosine in

$$x(t) = \sum_{n=1}^{10} \frac{-5}{\pi n^2} \cos(2\pi n f_o t)$$

Then… taking cos of this matrix creates 10 different cosine functions, one each row of the matrix COS, which is a 10x501 matrix as can be seen by checking its size:

```
>> size(COS)

ans =
```

[1] This example is related to a topic called "Fourier Series" that is covered in Signals & Systems courses.

```
    10   501
```

Now, what's in this matrix? Each row consists of a sinewave for a particular value of n and all the values of t. For example, the first row of COS could be computed as cos(1*t*2*pi); likewise, the second row of COS is cos(2*t*2*pi), etc.

We can see this by plotting these rows. Note that the indexing scheme for a matrix A is A(3,4:10) extracts the 4th through 10th elements in the 3rd row. When you want all the values in a given row a short cut notation is, for example, A(3,:) which extracts all the values on the 3rd row of A. Thus the next two commands plot the numbers on the 1st and 2nd rows of COS, respectively:

```
>> plot(t,COS(1,:))
>> plot(t,COS(2,:))
```

Note also that A(:,5) would give all the values in the 5th *column* of A.

Now we compute the 10 sinusoids' amplitudes as a column vector (a column vector because that is consistent with the structure of COS, namely that the times are varying over the columns and the harmonic number (or coefficient index) is varying over the rows) :

```
>> A=-5./(pi*n.^2)

A =

  -1.5915
  -0.3979
  -0.1768
  -0.0995
  -0.0637
  -0.0442
  -0.0325
  -0.0249
  -0.0196
  -0.0159

>> x=sum(A(:,ones(1,501)).*COS);
>> plot(t,x)
```

This replicates the column vector A 501 times to give a 10x501 matrix that has all 501 columns equal to A. This is done as follows: ones(1,501) creates a 1x501 vector that has all ones in it; this is used to index the first (and only) column of A 501 times; the : selects all ten rows of A. Experiment with smaller versions of this to see what is happening. Then this 10x501 matrix of A's is multiplied element-by-element (via the .*) with COS; that is every element of the 1st row of COS is multiplied by A(1), every element of the 2nd row of COS is multiplied by A(2), every element of the 3rd row of COS is multiplied by A(3), etc. This is in effect multiplying each harmonic component by its corresponding coefficient. Finally, the sum(…) function sums down each column of the 10x501 matrix to give a 1x501 element function of time that represents the desired partial Fourier series sum. Plotting this versus time vector t shows the result of the approximation.

Extracting Data for Inclusion in HW or Reports

Commands and Data:

The GUI interface for MATLAB provides a way to get a history of the commands that have been entered. Commands can be copied from this history and put into word processing files. You can also copy the commands back into the command line to re-run an earlier command.

Graphics of Plots:
There are two ways to do this:
Generate Hardcopy of Figures: To print figures on the computer's printer use the printer icon in the Figure's GUI Window.

Then include the hardcopy in your HW or report.

Generate Softcopy of Figures: Once you have created a plot (and labeled it!!!) you can

- Save it as a MATLAB *.fig file
- Copy-and-Paste it into a document

To save a *.fig File just click on the "Save" icon on the figure's GUI window. A saved *.fig file can be opened at later time and modified using the figure window GUI controls.

To copy-and-paste into a word processing program click on "Edit" and then click on "Copy Figure". Note that there is a choice called "Copy Options" that you might find helpful.

Writing MATLAB Programs

There are two main types of programs within MATLAB: script and function.

See this tutorial on-line: http://www.mathworks.com/videos/writing-a-matlab-program-69023.html

Script Files: A script file is simply a way to have MATLAB run a series of commands just as if they were issued at the command line. It takes no input arguments nor does it return any output arguments.

A script file has access to every variable that was in the workspace before it starts. It also leaves all of its computed results in the workspace after it finishes.

A script file is created by putting MATLAB commands in a text file with a *.m extension and saving them to a folder that is in MATLAB's path.
So… if you put the following in a *.m text file:

comments start with %

```
t=0: 0.02e-3:10e-3;          % creates a 1 x 501 row vector of time points
n=(1:10).';                  % creates a 10x1 column vector of integers 1 to 10
COS=cos((n*t)*2*pi*1);
```

After this runs: t, n, and COS are all available in the workspace.

Function Files: A function file is a way of creating your own MATLAB commands. It can take input arguments to allow passing variables to it from the workspace. It can also return output arguments to allow passing some of its computed results back the work space.

A function file has access only to the variables that are created within it or that are passed to it as input arguments. The only variables that it returns to the workspace are those that are output arguments.

Like the script file, a function file is created by putting MATLAB commands in a text file with a *.m extension and saving them to a folder that is in MATLAB's path. **BUT**… to make it a function file its first line must include a "function line" (see example below).

We can make the above into a function file that allows a user to specify how many cosine terms get created by passing the input variable N:

Function Line specifies function name & In/Out variables

```
function COS=cos_mat(N)
t=0: 0.02e-3:10e-3;          % creates a 1 x 501 row vector of time points
n=(1:N).';                   % creates a Nx1 column vector of integers 1 to N
COS=cos((n*t)*2*pi*1);
```

This should be saved in a file called cos_mat.m

You could call it at the command line like this: X=cos_mat(10);

Note that when called, a function can assign its output result to any variable name

Debugging MATLAB Programs

Once you've written your script or function you may discover that when you attempt to run it you get an error. MATLAB has built-in debugging tools to help you find and correct errors. The following video is a nice tutorial on how to use the debugging features of MATLAB:

https://www.youtube.com/watch?v=PdNY9n8lV1Y

Coding Style for MATLAB Programs in DSP

In DSP (and other areas) we generally want to implement some DSP algorithm or processing and then we need to test it. I strongly recommend the following:

- Write the DSP algorithm as a function that:
    - takes as input variables the signal to be processed and any parameters needed to set the operating point of the algorithm
    - gives as output variables the processed signal (or other data results) and any auxiliary parameters of importance
- Then write a "main" function that:
    - Creates the test signal(s) and any other data needed
    - Calls the algorithm function you are testing
    - If needed, computes performance assessments and logs the results

By separating these two parts (the signal generation for the test and the processing of the signal) it is easier for someone else to see what you have done. It is also easer to reuse the algorithm function in some other test scenario or application.