Introduction to MATLAB

1 Overview

MATLAB is a powerful, widely used tool for doing signal processing. The primary goal of this lab is to familiarize you with using MATLAB. It's assumed you have some experience with programming in some other language. You should already know something about the basics about variables, loops, functions, etc. This introduction shows how common programming constructs are done in MATLAB.

There are three specific goals for this lab:

1. Learn basic MATLAB commands and syntax, including the help system.
2. Learn to write and edit your own script files in MATLAB, and run them as commands.
3. Learn a little about advanced programming techniques for MATLAB, i.e., vectorization.

This lab will get you started with MATLAB. The Chapter 2 build-a-figures will introduce additional MATLAB commands and techniques as they are needed.

2 Getting Started

There have been many fine tutorial introductions to MATLAB written. The approach taken here is to use some of The Mathworks (http://www.mathworks.com/) tutorials to get a general introduction to MATLAB, its command environment, editor, etc. and then learn many specific techniques by recreating many of the figures in the text.

2.1 The Mathworks Tutorials

The Mathworks site lists several tutorials for learning MATLAB. The Interactive MATLAB Tutorial (http://www.mathworks.com/academia/student_center/tutorials/mltutorial_launchpad.html) matches well with what we are doing here. The follow table lists the various parts of the tutorial and notes which are most important to do.

Time	Link	*Description*	*Importance*
(5:38)	Introduction to MATLAB Tutorials	Directions for viewing modules and downloading the tutorial example files.	Yes
(6:10)	Navigating the MATLAB Desktop	Get comfortable with the MATLAB desktop environment	Yes
(22:55)	Using MATLAB as a Graphical Calculator	Learn to use MATLAB as a powerful tool for solving problems and visualizing functions	Very
(2:13)	Using MATLAB to Solve Problems	Explore a case study of using MATLAB to solve problems	Yes
(13:06)	Importing and Extracting Data	Work with data in MATLAB – how to bring it in and manipulate it	Not much

(10:55)	Visualizing Data	Better understand complex mathematical functions and data by visualizing it with MATLAB	Very
(12:33)	Conducting Computational Analysis in MATLAB	Analyze data through MATLAB functions and computation	Not Much
(6:42)	Fitting Data to a Curve	Model your data to an equation and interpolate upon it in MATLAB	Not Much
(9:29)	Writing MATLAB Programs	Write your first MATLAB program (script M-file)	Yes
(8:01)	Publishing MATLAB Programs	Communicate your work by publishing it as an HTML report, all from within MATLAB	Some

If time permits, do all the tutorials above. If you can't do them all, at least do the important and very important ones. These will give you the background needed for the next section.

2.2 Playing with MATLAB

The following steps will introduce you to MATLAB by letting you play with it.

(a) Run the MATLAB help desk by typing `doc`. The help desk provides a hypertext interface to the MATLAB documentation. Two links of interest are Getting Started and Getting Help in MATLAB. Both are under Documentation Set.

(b) Explore the MATLAB `helpwin` capability available at the command line. Try the following:

```
helpwin
helpwin plot
helpwin colon     %<--- a VERY IMPORTANT notation
helpwin ops
helpwin zeros
helpwin ones
lookfor filter    %<--- keyword search
```

(c) Run the MATLAB demos: type `demo` and explore a variety of basic MATLAB commands and plots.

(d) Use MATLAB as a calculator. Try the following:

```
pi*pi - 10
sin(pi/4)
ans ^ 2           %<--- "ans" holds the last result
```

(e) Do variable name assignment in MATLAB. Try the following:

```
x = sin( pi/5 );
cos( pi/5 )       %<--- assigned to what?
y = sqrt( 1 - x*x )
ans
```

(f) Complex numbers are natural in MATLAB. The basic operations are supported. Try the following:

```
z = 3 + 4i, w = -3 + 4j
real(z), imag(z)
```

```
abs([z,w])        %<-- Vector constructor
conj(z+w)
angle(z)
exp( j*pi )
exp(j*[ pi/4, 0, -pi/4 ])
```

3 Warm-Up

3.1 MATLAB Array Indexing

(a) Make sure that you understand the colon notation. In particular, explain in words what the following MATLAB code will produce

```
jkl = 0 : 6
jkl = 2 : 4 : 17
jkl = 99 : -1 : 88
ttt = 2 : (1/9) : 4
tpi = pi * [ 0:0.1:2 ];
```

(b) Extracting and/or inserting numbers into a vector is very easy to do. Consider the following definition of `xx`:

```
xx = [ zeros(1,3), linspace(0,1,5), ones(1,4) ]
xx(4:6)
size(xx)
length(xx)
xx(2:2:length(xx))
xx(2:2:end))
```

Explain the results echoed from the last four lines of the above code.

(c) Observe the result of the following assignments:

```
yy = xx; yy(4:6) = pi*(1:3)
```

Now write a statement that will take the vector `xx` defined in part (b) and replace the even indexed elements (i.e., `xx(2)`, `xx(4)`, etc) with the constant π^{π}. Use a vector replacement, not a loop.

3.2 MATLAB Script Files

(a) Experiment with vectors in MATLAB. Think of the vector as a set of numbers. Try the following:

```
xk = cos( pi*(0:11)/4 ) %<---comment: compute cosines
```

Explain how the different values of cosine are stored in the vector `xk`. What is `xk(1)`? Is `xk(0)` defined?

NOTES: the semicolon at the end of a statement will suppress the echo to the screen. The text following the `%` is a comment; it may be omitted.

(b) (A taste of vectorization) Loops can be written in MATLAB, but they are NOT the most efficient way to get things done. It's better to always avoid loops and use the colon notation instead. The

following code has a loop that computes values of the cosine function. (The index of `yy( )` must start at 1.)

Rewrite this computation without using the loop (follow the style in the previous part).

```
yy = [ ];   %<--- initialize the yy vector to be empty
for k=-5:5
yy(k+6) = cos( k*pi/3 )
end
yy
```

Explain why it is necessary to write `yy(k+6)`. What happens if you use `yy(k)` instead?

(c) Plotting is easy in MATLAB for both real and complex numbers. The basic plot command will plot a vector y versus a vector x connecting successive points by straight lines. Try the following:

```
x = [-3 -1 0 1 3];
y = x.*x - 3*x;
plot( x, y )
z = x + y*sqrt(-1)
plot( z ) %<---- complex values: plot imag vs. real
```

Use `helpwin arith` to learn how the operation `xx.*xx` works when `xx` is a vector; compare to matrix multiply.

When unsure about a command, use `helpwin`.

(d) Use the built-in MATLAB editor to create a script file called `mylab1.m` containing the following lines:

```
tt = -1 : 0.01 : 1;
xx = cos(5*pi*tt);
zz = 1.4*exp(j*pi/2)*exp(j*5*pi*tt);
plot(tt, xx, 'b-', tt, real(zz), 'r--')   %<--- plot a sinusoid
grid on
title('TEST PLOT of a SINUSOID')
xlabel('TIME (sec)')
```

Explain why the plot of `real(zz)` is a sinusoid. What is its phase and amplitude? Make a calculation of the phase from a time-shift measured on the plot.

(e) Run your script from MATLAB. To run the file mylab1 that you created previously, try

```
mylab1            %<---will run the commands in the file
type mylab1       %<---will type out the contents of
                  % mylab1.m to the screen
```

3.3 MATLAB Sound

The exercises in this section involve sound signals, so you should bring headphones to the lab for listening.

(a) Run the MATLAB sound demo by typing `xpsound` at the MATLAB prompt. If you are unable to hear the sounds in the MATLAB demo then ask for help.

When unsure about a command, use `helpwin`.

(b) Now generate a tone (i.e., a sinusoid) in MATLAB and listen to it with the `soundsc( )` command.[1]

The first two lines of code in part 3.2(d) create a vector xx of values of a 2.5 Hz sinusoid. The frequency of your sinusoidal tone should be 2000 Hz and its duration should be 0.9 sec. Use a sampling rate (`fs`) equal to 11025 samples/sec. The sampling rate dictates the time interval between time points, so the time-vector should be defined as follows:

```
tt = 0:(1/fs):dur;
```

where `fs` is the desired sampling rate and `dur` is the desired duration (in seconds). Read the online help for both `sound( )` and `soundsc( )` to get more information on using this command. What is the length (number of samples) of your `tt` vector?

4 Manipulating Sinusoids with MATLAB

Now you're on your own. Include a short summary of this Section with plots in your Lab report. Write a MATLAB script file to do steps (a) through (d) below. Include a listing of the script file with your report.

(a) Generate a time vector (`tt`) to cover a range of t that will exhibit approximately two cycles of the 4000 Hz sinusoids defined in the next part, part (b). Use a definition for `tt` similar to part 3.2(d). If we use *T* to denote the period of the sinusoids, define the starting time of the vector `tt` to be equal to *−T*, and the ending time as *+T*. Then the two cycles will include t = 0. Finally, make sure that you have at least 25 samples per period of the sinusoidal wave. In other words, when you use the colon operator to define the time vector, make the increment small enough to generate 25 samples per period.

(b) Generate two 4000 Hz sinusoids with arbitrary amplitude and time-shift.

$$x_1(t) = A_1 \cos(2\pi(4000)(t - t_{m1})) \qquad x_2(t) = A_2 \cos(2\pi(4000)(t - t_{m2}))$$

Select the value of the amplitudes and time-shifts as follows: Let A1 be equal to your age and set A2 = 1.2A1. For the time-shifts, set $t_{m1} = (37.2/M)T$ and $t_{m2} = -(41.3/D)T$ where D and M are the day and month of your birthday, and T is the period.

[1] The `soundsc(xx,fs)` function requires two arguments: the first one (`xx`) contains the vector of data to be played, the second argument (`fs`) is the sampling rate for playing the samples. In addition, `soundsc(xx,fs)` does automatic scaling and then calls `sound(xx,fs)` to actually play the signal.

Make a plot of both signals over the range of $-T \leq t \leq T$. For your final printed output in part (d) below, use `subplot(3,1,1)` and `subplot(3,1,2)` to make a three-panel figure that puts both of these plots in the same figure window. See `helpwin subplot`.

(c) Create a third sinusoid as the sum: $x_3(t) = x_1(t) + x_2(t)$. In MATLAB this amounts to summing the vectors that hold the values of each sinusoid. Make a plot of $x_3(t)$ over the same range of time as used in the plots of part (b). Include this as the third panel in the plot by using `subplot(3,1,3)`.

(d) Before printing the three plots, put a title on each subplot, and include your name in one of the titles.

See `helpwin title`, `helpwin print` and `helpwin orient`, especially `orient tall`.

4.1 Complex Amplitude

Write one line of MATLAB code that will generate values of the sinusoid $x_1(t)$ above by using the complex amplitude representation:

$$x_1(t) = \mathrm{Re}\{Xe^{j\omega t}\}$$

Use appropriate constants for X and ω.

5 Introduction to Complex Exponentials

The goal of this section is to gain familiarity with complex numbers and their use in representing sinusoidal signals such as $x(t) = A\cos(\omega t + \phi)$ as complex exponentials $z(t) = Ae^{j\phi}e^{j\omega t}$. The key is to use the appropriate complex amplitude together with the real part operator as follows:

$$x(t) = A\cos(\omega t + \phi) = \mathrm{Re}\{Ae^{j\phi}e^{j\omega t}\}$$

5.1 Overview

Manipulating sinusoidal functions using complex exponentials turns trigonometric problems into simple arithmetic and algebra. In this lab, we first review the complex exponential signal and the phasor addition property needed for adding cosine waves. Then we will use MATLAB to make plots of phasor diagrams that show the vector addition needed when adding sinusoids.

5.2 Complex Numbers in MATLAB

MATLAB can be used to compute complex-valued formulas and also to display the results as vector or

"phasor" diagrams. Here are some of MATLAB's built-in complex number operators:

`conj`	Complex conjugate
`abs`	Magnitude
`angle`	Angle (or phase) in radians
`real`	Real part
`imag`	Imaginary part
`i,j`	pre-defined as $\sqrt{-1}$
`x = 3 + 4i`	i suffix defines imaginary constant (same for j suffix)

`exp(j*theta)` Function for the complex exponential $e^{j\theta}$

Each of these functions takes a vector (or matrix) as its input argument and operates on each element of the vector. Notice that the function names `mag()` and `phase()` do not exist in MATLAB.[2]

When unsure about a command, use `helpwin`.

5.3 Vectorization

The power of MATLAB comes from its matrix-vector syntax. In most cases, loops can be replaced with vector operations because functions such as `exp()` and `cos()` are defined for vector inputs, e.g.,

```
cos(vv) = [cos(vv(1)), cos(vv(2)), cos(vv(3)), ... cos(vv(N))]
```

where vv is an N-element row vector. Vectorization can be used to simplify your code. If you have the following code that plots a certain signal,

```
M = 200;
for k=1:M
   x(k) = k;
   y(k) = cos( 0.001*pi*x(k)*x(k) );
end
plot( x, y, 'ro-' )
```

then you can replace the for loop and get the same result with 3 lines of code:

```
M = 200;
y = cos( 0.001*pi*(1:M).*(1:M) );
plot( 1:M, y, 'ro-' )
```

Use this vectorization idea to write 2 or 3 lines of code that will perform the same task as the following MATLAB script without using a `for` loop. (Note: there is a difference between the two operations `xx*xx` and `xx.*xx` when xx is a vector.)

```
%--- make a plot of a weird signal
N = 200;
for k=1:N
   xk(k) = k/50;
   rk(k) = sqrt( xk(k)*xk(k) + 2.25 );
   sig(k) = exp(j*2*pi*rk(k));
end
plot( xk, real(sig), 'mo-' )
```

[2] In the latest release of MATLAB a function called `phase()` is defined in a rarely used toolbox; it does more or less the same thing as `angle()` but also attempts to add multiples of 2π when processing a vector.

5.4 Functions

Functions are a special type of M-file that can accept inputs (matrices and vectors) and also return outputs. The keyword `function` must appear as the first word in the file that defines the function, and the first line of the M-file defines how the function will pass input and output arguments. The file extension must be lower case "`m`" as in `my_func.m`.

The following function has a few mistakes. Before looking at the correct one below, try to find these mistakes (there are at least three):

```
matlab mfile [xx,tt] = badcos(ff,dur)
%BADCOS Function to generate a cosine wave
% usage:
% xx = badcos(ff,dur)
% ff = desired frequency in Hz
% dur = duration of the waveform in seconds
%
tt = 0:1/(100*ff):dur;  %-- gives 100 samples per period
badcos = cos(2*pi*freeq*tt);
```

The corrected function should look something like:

```
function [xx,tt] = goodcos(ff,dur)
tt = 0:1/(100*ff):dur;  %-- gives 100 samples per period
xx = cos(2*pi*ff*tt);
```

Notice the word "`function`" in the first line. Also, "`freeq`" has not been defined before being used. Finally, the function has "`xx`" as an output and hence "`xx`" should appear in the left-hand side of at least one assignment line within the function body. The function name is not used to hold values produced in the function.

6 Complex Exponentials

In this part, you learned how to write M-files. In this section, you will write two functions that can generate sinusoids or sums of sinusoids.

6.1 M-file to Generate a Sinusoid

Write a function that will generate a single sinusoid, $x(t) = A\cos(\omega t+\phi)$, by using four input arguments: amplitude (A), frequency (ω), phase (ϕ) and duration (`dur`). The function should return two outputs: the values of the sinusoidal signal (x) and corresponding times (t) at which the sinusoid values are known. Make sure that the function generates 20 values of the sinusoid per period. Call this function `one_cos( )`. Hint: use `goodcos( )` from part (a) as a starting point.

Demonstrate that your `one_cos( )` function works by plotting the output for the following parameters: A = 95, ω= 200π rad/sec, $\phi = \pi/5$ radians, and `dur` = 0.025 seconds. Be prepared to explain r features on the plot that indicate how the plot has the correct period and phase. What is the expected period in millisec?

7 Debugging Skills

Testing and debugging code is a big part of any programming job. Almost any modern programming environment provides a symbolic debugger so that break-points can be set and variables examined in the middle of program execution. Of course, many programmers insist on using the old-fashioned method of inserting print statements in the middle of their code (or the MATLAB equivalent, leaving off a few semi-colons). This is akin to riding a tricycle to commute around town.

In order to learn how to use the MATLAB tools for debugging, try `helpwin debug`. Here is part of what you'll see:

```
dbstop - Set breakpoint.
dbclear - Remove breakpoint.
dbcont - Resume execution.
dbstack - List who called whom.
dbstatus - List all breakpoints.
dbstep - Execute one or more lines.
dbtype - List M-file with line numbers.
dbquit - Quit debug mode.
```

When a breakpoint is hit, MATLAB goes into debug mode and the prompt changes to a K>. Any MATLAB command is allowed at the prompt. To resume M-file function execution, use `dbcont` or `dbstep`. To exit from the debugger use `dbquit`.

One of the most useful modes of the debugger causes the program to jump into "debug mode" whenever an error occurs. This mode can be invoked by typing:

```
dbstop if error
```

With this mode active, you can snoop around inside a function and examine local variables that probably caused the error. You can also choose this option from the debugging menu in the MATLAB editor. It's sort of like an automatic call to 911 when you've gotten into an accident. Try `helpwin dbstop` for more information. Use the following to stop debugging

```
dbclear if error
```