

Tutorial Notes

1 Athena and 6.555

Athena is MIT's UNIX-based campus-wide academic computing facility. In this tutorial will quickly discuss aspects of the Athena environment that are relevant to this course. For more information about Athena visit <http://web.mit.edu/olh/>.

1.1 6.555 data and directories

All the data, code, and working directories you will need to access for the labs are located in the 6.555 course locker on the athena.mit.edu AFS¹ cell. The root of the course locker is `/afs/athena.mit.edu/course/6/6.555/`. When you first log onto an athena machine you can type:

```
athena% add 6.555
```

This will create a directory at `/mit/6.555/` which is a symbolic link to the course locker.

```
athena% cd /mit/6.555/  
athena% pwd  
/afs/athena.mit.edu/course/6/6.555
```

In most labs we will specify paths to files using this link. The data and Matlab code you will use for each lab is located in the appropriate subdirectory of `/mit/6.555/data/` and `/mit/6.555/matlab/` respectively.

Each pair of lab partners will form a group and be assigned a working directory in `/mit/6.555/groups/`. Your working directory is a nice place to keep your code and other course related documents if you don't want to use up space on your Athena home directory. Group/lab partners will be setup during the first real lab.

1.2 Copying files to your personal computer

It is recommended that you use the Athena machines during lab. All labs were designed and tested for the Athena environment and current Matlab version on Athena. However, you are free to use your personal computer or laptop if you have your own copy of Matlab and are more comfortable with it.

You can copy files from the athena course locker to your personal computer using secure FTP. On Windows you can download SecureFX from <http://web.mit.edu/software/>. Instructions on how to use it are located at <http://itinfo.mit.edu/article.php?id=6178>. In Linux or on a Macintosh (in Terminal.app) you can just use `sftp`:

```
sftp username@ftp.dialup.mit.edu:/afs/athena.mit.edu/course/6/6.555/path_to_file
```

¹AFS is a distributed networked filesystem. For more information about AFS visit <http://itinfo.mit.edu/article.php?id=6845>

Note that when using secure FTP you should use the full path to the course locker rather than /mit/6.555. ftp.dialup.mit.edu redirects you to a new machine that may not have /mit/6.555 linked.

2 Matlab

Matlab is short for MATrix LABoratory. The Mathworks website states: Matlab is a high-level language and interactive environment that enables you to perform computationally intensive tasks faster than with traditional programming languages such as C, C++, and Fortran [*sic*]. It is a great tool that eliminates some of the not-so-fun parts of programming such as worrying about memory management. However, as we will see, while it is very efficient with vectors/matrices it is not so great with loops.

2.1 Starting Matlab

You can add Matlab to your path on Athena by typing

```
athena% add matlab
```

and then run it with the full desktop interface with

```
athena% matlab -desktop
```

Once it starts up you will get a simple prompt >>

2.2 Matrix, vector, and signal generation and manipulation

Let's start of with the basics. We will start by creating a constant

```
>> c = 1.23
c =
    1.2300
```

Matlab will print out the result of a command if you do not put a semi-colon (;) at the end as shown above. You can see what variables you have in your environment at any point in time by using the **whos** function.

```
>> whos
  Name      Size      Bytes  Class  Attributes
  c         1x1         8  double
```

Note that right now we only have 1 variable which is a 1x1 matrix of class double. By default everything is double. For information on converting to other datatypes you can type **help datatypes**. You can remove a variable using the **clear** function.

For help on a particular function type **help function_name**. You can also type **helpdesk** to bring up HTML documentation. If there is HTML documentation for a function **doc function_name** will also bring it up.

Next we will create some vectors:

```
>> v1 = [1 2 3 4];
>> v2 = [1;2;3;4];
>> v3 = [1 2 3 4]';
>> v4 = 1:4;
```

Use **whos** or the **size** function to find out the size of each of these vectors. Type each variable name without a semi-colon to print them out. Note that ' is the transpose operation, and a ; within brackets indicates a new row. The : operator allows you to quickly make vectors of a particular range. You can even specify an increment as well. For example, try:

```
>> n = -4:4
>> n = -4:2:4
>> n = 0:.4:2
```

Some functions you may find useful when operating on double vectors are:

```
>> round(n)
>> ceil(n)
>> floor(n)
```

Making a matrix is just as easy. Try:

```
>> m1 = [1 2 3 4; 5 6 7 8]
>> m2 = m1'
>> m3 = ones(3,3)
```

The **ones** function creates a matrix of 1s of a certain size. Similarly **zeros** can be used to create matrix of zeros.

Ok, let's start making some more interesting variables. Say that we have a discrete-time signal, $x[n] = \sin(2\pi 0.02n)$, that we want to view over the interval $0 \leq n \leq 1023$ samples. First create the n-values:

```
>> n=[0:1023];
```

Use this vector together with the pre-defined Matlab constant π to generate a Matlab vector **x**, equal to $x[n]$ over the desired range:

```
>> x=sin(2*pi*0.02*n);
```

In a similar manner, create **y** to represent $y[n] = \cos(2\pi f_y n)$ for some arbitrary choice of $0.02 \leq f_y \leq 0.15$, and over the same range of n (you can use the same **n** from before). Preferably, choose f_y so that it is different from $f_x = 0.02$ (for example, $f_y = \frac{1}{9}$).

You can look at certain elements of a vector or matrix. Just index the matrix with the elements you want. For example, $\mathbf{x}(1:2:10)$ shows the 1,3,5,7, and 9th values of \mathbf{x} . Now create a matrix, such as $\mathbf{A}=\text{randn}(10,10)$; which is a 10x10 matrix of Gaussian distributed random variables. $\mathbf{A}(2,3)$ shows the item in the second row third column, and $\mathbf{A}(:,1)$ shows the entire first column (The $:$ indicates all indices). Another useful note is that $\mathbf{x}(1:2:\text{end})$ takes every other element until the end of the vector.

The **find** function is useful for indexing elements in a vector that meet a certain condition. For example, $\text{indx} = \text{find}(\mathbf{x} > 0)$; will return a vector **indx** which contains all the indices of \mathbf{x} where \mathbf{x} is greater than zero. If you want to half-wave rectify a signal you can type:

```
>> xp = x;
>> xp(find(x<0)) = 0;
```

Note that **find** actually returns the indices of the nonzero values of a vector. $\mathbf{m} = (\mathbf{x} < 0)$ makes **m** a logical vector the length of \mathbf{x} with only ones and zeros (true or false). It has ones where the condition is true and zero otherwise.

Now create the vectors **s** and **t** to represent $s[n] = x[n] + y[n]$ and $t[n] = x[n]y[n]$ Note that the latter requires the use of Matlab's point-wise multiply **.*** which multiplies two identically-sized matrices or vectors element-by-element:

```
>> t = x.*y;
```

Just for fun, try to perform the operation $\mathbf{x}*\mathbf{y}$. This operation, which performs the matrix multiplication, is undefined, since both \mathbf{x} and \mathbf{y} are row vectors. If either of them were transposed into a column vector it would be a valid matrix multiplication: $\mathbf{x}*\mathbf{y}'$ is the vector inner-product and $\mathbf{x}'*\mathbf{y}$ is the vector outer product.

This is a good time to show the speed difference between using Matlab's built-in operations and doing things brute force with loops. Make two long random vectors of the same size:

```
>> v1 = randn(1,1e6);
>> v2 = randn(1,1e6);
```

Now compare the times returned by:

```
>> tic; v3 = v1 + v2; toc
and
>> tic; for k=1:length(v1) v3(k) = v1(k) + v2(k); end; toc
```

Note that **tic** starts a timer, **toc** prints the time since **tic**. The reason for this time difference is that when you index a vector or matrix Matlab does bounds checking to make sure you don't address bad memory. This is performed in every iteration of the loop above. The built-in functions do bounds checking only once before they start.

2.3 Loading and saving data

You can use the **save** and **load** functions to save out and load in variables (or entire workspaces). For example, the following saves out \mathbf{x} , clears it from matlab, and then reads it back in:

```

>> whos x
      Name      Size      Bytes  Class
      x         1x1024     8192   double array
Grand total is 1024 elements using 8192 bytes
>> save('mydata.mat','x');
>> clear x;
>> whos x
>> load('mydata.mat');
>> whos x
      Name      Size      Bytes  Class
      x         1x1024     8192   double array
Grand total is 1024 elements using 8192 bytes

```

2.4 Plotting

Consider plotting the vectors **x**, **y**, **s** and **t**. This can be accomplished using the plot function:

```
>> plot(x);
```

The **plot** function will open a new figure window if there is none already open. Check out the **figure** function to see how to bring up new ones or make old figures active. To plot these ‘signals’ along their axis, we need to include the time vector **n** in the argument:

```
>> plot(n,x);
```

A string vector may also be included to specify line color or type. For example, **'y'** = yellow, **'r'** = red, **'-'** = solid, and **'.'** = dotted. For a complete list, type **help plot** or **doc linespec**.

An alternative plot function that is useful for plotting DT signals is **stem**. Try **stem(x(1:100))**.

2.4.1 Axis scaling

When you plot a vector, sometimes ‘too much’ is shown for useful viewing. In this case, **axis** can be used to re-scale the figure axis so that you can zoom in on a certain portion of the plot. The basic format is: **axis([xmin xmax ymin ymax])**. For example, plot **x** and focus in on $120 \leq n \leq 185$:

```

>> plot(n,x,'r');
>> axis([120 185 -1.1 1.1]);

```

Also **zoom** (the function or toolbar button) lets you draw a rectangle around a region to be zeroed in on without specify particular parameters.

2.4.2 Plotting multiple vectors

Multiple vectors may be plotted in the same figure in two ways. The first method involves ‘holding’ the current plot using **hold on**. When you are finished, **hold off** is required to ‘release’ the plot:

```
>> plot(n,x,'g');
>> hold on;
>> plot(n,y,'r');
>> hold off;
```

The second method uses a single plot command:

```
>> plot(n,x,'g',n,y,'r');
```

Plot all four vectors and zoom in to $120 \leq n \leq 185$:

```
>> h = plot(n,x,'g',n,y,'r',n,s,'b',n,t,'c');
>> axis([120 185 -2 2]);
```

Use this plot for the next set of steps

2.4.3 Labeling plots

Now, the plots must be made pretty. To do this, we first label the plot axes and give the plot a title using the functions **xlabel**, **ylabel** and **title**. For example,

```
>> xlabel('Time (samples)');
>> ylabel('Signal Value');
>> title('Plot of x[n],y[n],s[n],and t[n]');
```

Remember to surround the string inputs with single quotation marks! Another source of ‘messiness’ in plots of multiple vectors is that it is confusing to distinguish between the various plots. The **legend** command helps with this. For example,

```
>> legend(h,'x[n]','y[n]','s[n]','t[n]');
```

where **h** is the handle of the figure you are adding the legend to. If you don’t include **h** it will just add the legend to the current figure (see **gcf**).

2.4.4 Subplots

Another way of plotting multiple vectors on the same page is to use the **subplot** function. It breaks the figure window into a matrix specified by the user and allows a different plot in each of the smaller windows. Try:

```
>> subplot(2,2,1); plot(n,x);
>> subplot(2,2,2); plot(n,y);
>> subplot(2,2,3); plot(n,s);
>> subplot(2,2,4); plot(n,t);
```

The arguments to **subplot** are the number of rows, number of columns, and the index for the windows counting left to right, top to bottom. If you wish to work on a particular subplot, simply specify the window you want with **subplot(2,2,index)** and then use **axis**, **title**, etc.. as you would regularly. Try zooming in on a single period of signal **s** and adding a title and labels for both axes.

2.4.5 Outputting plots

If you want to output your plot to a printer use:

```
>> print -Pprinter_name
```

The printer in 14-0637 is named neos. You can also output to a postscript file using.

```
>> print -dps filename
```

See **help print** for more details. The print command is nice when you are writing a script or function that needs to save a figure for your report (we will talk about scripts and functions in a bit). However, if you just want to save or print the current figure it is sometimes more intuitive to just use the GUI. In the figure window you can just use File→Print to print and File→Save As to save the figure in various formats.

2.5 Frequency analysis

Matlab provides several tools that can be used to analyze the behavior of signal vectors. One of the most convenient is the **fft** command, which performs the Fast Fourier Transform on a given signal. Use it to analyze the signals we made before:

```
>> X = fft(x); Y = fft(y); S = fft(s); T = fft(t);
```

Given that the vectors contain $l = \text{length}(\mathbf{n})$ elements, **fft** computes the l -point FFT. As was quickly discussed in the last lecture, the l -point FFT computes the Fourier transform at l equally-spaced radian frequencies between 0 and 2π . For example in this example $l = 1024$, and we can generate a vector of the radian frequencies $\omega = 2\pi f$:

```
>> omega = n*(2*pi/length(n));
```

This can be converted into Hz by dividing by 2π ($F_s = 1$):

```
>> f = omega/(2*pi);
```

Use **f** to plot and observe the magnitude of the FFTs **X**, **Y**, **S**, and **T**. Use **abs(X)** to get the magnitude. Zoom in on the peaks (those below **f=0.5**) of the various FFTs. As expected, **X** has a peak at $f=0.02$, which is the frequency of the sinusoid $x[n]$. Similarly, **Y** should have a peak at the f_y chosen. **S** should have peaks at both 0.02 and f_y . Finally, **T** should have peaks at $f_y + 0.02$ and $f_y - 0.02$. (why is this?)

Note: In order to compute a specific **N**-point FFT, use **fft([signal],N)**.

2.6 Digital filtering

Matlab also has many functions for designing and simulating digital filters. Let us design a filter to separate $x[n]$ out of $s[n] = x[n] + y[n]$.

2.6.1 Filter design

This signal $s[n]$ has two frequency components one at $f_x = 0.02$ and one at your chosen f_y . Since $f_x < f_y$, let us design a low-pass filter that preserves f_x and cancels f_y . We can use the function **fir1** to design a low-pass filter with a cutoff of $\frac{0.02+f_y}{2}$.

```
>> h = fir1(N,2*cutoff)
```

where **N** is the order of the filter and **cutoff** is the cutoff frequency. Note, the factor of 2 applied to the cutoff arises due to the way in which Matlab specifies digital frequencies. Specifically, Matlab uses frequencies in terms of radian frequency divided by π (rather than divided by 2π). Therefore, it is necessary to multiply the cutoff frequency by 2. Pick **N** > 50 for a high order filter with a sharp cutoff.

2.6.2 Filter behavior

The standard tool for viewing filter frequency responses is **freqz**. To get the transfer function of an FIR filter, **h**, use:

```
>> H = freqz(h,1,2*pi*f);
```

In **freqz**, the first argument is the set of transform function numerator coefficients **h**. The second argument is the set of transfer function denominator coefficients (1 for FIR). The third argument is a vector of radian frequencies at which to evaluate the transfer function.

Plot the frequency response as with the FFT magnitude, **plot(f,abs(H))**

2.6.3 Filter simulation

The filter can be applied to signal using the Matlab **fftfilt** function

```
>> sf = fftfilt(h,s);
```

This function filters **s** using the overlap-add method, which will be discussed in class. Alternatively, for this FIR filter you can just directly convolve **s** and **h**:

```
>> sf2 = conv(h,s)
```

In general, **fftfilt** is faster on long inputs, such as speech.

Compare **sf** to **x**. Since f_y has been filtered out the signal **sf** should be a sinusoid with the same frequency as **x**. Note that **sf** will not equal **x** due to the inherent delay in the digital filter. Note also that **conv**, which performs the direct convolution, results in **sf2** being **N** samples longer than the original **s**, where **N** is the filter order chosen.

Finally, plot the frequency representation of **sf** using **fft**, and compare it to the frequency representation of **s**. It should be clear what the difference is and why.

2.7 Functions and scripts

So far we have been typing Matlab command directly into the Matlab command window. However, usually it is a better idea to create scripts or functions. A script is simply a file with a list of commands to run. These commands run within the scope of whatever called the script. For example if you run a script from the Matlab prompt it is equivalent to you typing all those commands in, one by one. This means that variables created within the script will be accessible after it is finished, and that the script can use variables you have specified prior to calling the script. Matlab scripts are saved as M-files. That is they are a text files saved with the extension `‘.m’`, *e.g.* `‘myscript.m’`. You can run this script by simply typing **myscript** if it is in your current path.

When you type

```
>> edit myscript
```

Matlab will open `‘myscript.m’` in its editor. It will create it if it does not already exist. You can type a set of commands in this file, save it, and then run it from the command window.

You can also define functions by creating an M-file, and typing

```
function [out1, out2] = myfunction(in1,in2)
```

on the first line. The **out1,out2** are the output variable names, **myfunction** is the name of the file, and **in1** and **in2** are the input variable names. The number of inputs and output arguments is not limited. Generally the name of the M-file is the same as the name of the function. Functions have their own scope. They cannot access any variables used by the caller unless they are passed in as arguments. All variables created within the function will be lost when the function exits unless they are passed as an output argument.

Here is a simple example of a function:

```
% STAT useful statistic calculator
%   [AVG,STDEV] = STAT(X) returns the average/mean and standard deviation of the
%   elements in X. X is treated as a vector.
%
%   See also MEAN, STD
%
function [avg,stdev] = stat(x)

% Put x in vector form
% For example, turn an n x m matrix into a n*m length vector
x = x(:);
```

```
n = length(x);
avg = sum(x)/n;
stdev = sqrt(sum((x-avg).^2)/n);
```

ALWAYS DOCUMENT/COMMENT YOUR CODE. Commenting will help you debug and think through your code. It will also be useful when or if you ever look back at your code weeks, months or years later. But, the MOST important reason to comment your code is that it will keep your TA and those who will grade your lab happy :). Note that you don't have to comment every line. Code can be self documenting (i.e. `n=length(x);`). The comments you put in a function that occur before the function header will be displayed when you type **help yourfunctionname**.

Matlab has all the basic control statements you find in other languages such as C/C++ or java. Type **help** on **if,else,elseif,end,for,while,** and **switch** to find out more.

2.8 Pointers to other useful functions

There are many, many useful functions in Matlab. Almost any function you can imagine is somewhere. We will tell you about functions useful to each specific lab when that lab is first introduced. Some functions that are generally useful are:

- **min,max,median,var,mean:** Do what you think they should do on vectors. Check help before you apply them to a matrix to see what dimension they operate over or how to change it.
- **any:** returns true (logical 1) if any element of the vector passed in is nonzero.
- **all:** returns true (logical 1) if all elements of a vector are nonzero.
- **cumsum:** performs a cumulative sum on the columns of a matrix
- **reshape:** allows you to change the space of a matrix
- **type:** lets you print out the contents of a file
- **repmat:** Replicate or tile an array. This lets you avoid lots of loops (trades memory for speed). For example, if you want to add a vector **v** to every column of a matrix **m** you can do **mnew = m + repmat(v,1,size(m,2));** rather than looping through each column.