Lab 1: Introduction to MATLAB

due Feb. 1, 2016

Goal:

In this lab we will become familiar with the MATLAB programming environment and the ENVI software suite that we will use in this class. You will learn some of the basics of MATLAB syntax and functionality and review the basic ENVI display interface. You will also write your own program to do simple manipulations to an image in MATLAB.

Introduction:

MATLAB is a mathematical computer package from The MathWorks. "MATLAB" stands for "Matrix Laboratory". MATLAB's functions operate on *matrices* and *vectors*, as well as *scalar* numbers, so it is very useful for *linear algebra* applications. With it, you can perform any number of simple or complicated mathematical *operations*, including image processing. As a general-purpose *programming language*, it is easy to learn and use, but powerful enough for a wide range of science and engineering applications.

The best way to learn MATLAB is by using it, which is what we will be doing in this lab and throughout the semester. The built-in and online documentation is excellent and there is also a complete set of manuals for you to use. Please keep the manuals in the lab!

To access MATLAB, log in to one of the PCs and double-click on the MATLAB icon on the desktop (or in the Windows Start Menu). The MATLAB environment will open. It is a window with several panes. The largest one is the *command line*. The others allow you to view the current directory, see what *variables* are in *memory*, etc. The menu bar at the top and the "Start" menu at the bottom of the window allow you to access many of MATLAB's functions as well as help and tutorials.

For our purposes, MATLAB is ideal since an image is basically a matrix. Remember that digital images have been *discretized* and are composed of pixels that can be thought of as *rectangular, 2-D arrays* of small dots on the screen. Each dot has a particular location associated with it, its *coordinates* (expressed as a pair of numbers) in the rectangular array. Each of the dots also has a value associated with it (a number "stored" at that location) that specifies the brightness (or color) of that dot. These values (*DN*, or *digital numbers*) are the *elements* of the matrix. The matrix has a size, m x n, which represents the number of *rows* (m), the number of *columns* (n), and therefore the total number of pixels (m x n) that makes up the image. Each pixel also has a *logical* size related to the *quantization*, e.g. how many *binary digits* (0's and 1's) it takes to record the number stored. (Note that this is different than a physical or *spatial* size!) For example, to record *grayscale* between 0 and 255 (256 total possible values, in this case, shades of gray), you need to quantize the image at 8-*bit* ($2^8 = 256$). Thus, if you multiply the number of bits/pixel by the total number of pixels (m x n), you can figure out how much memory or *disk space* your image will fill.

Since we can treat our image as a matrix, a *mathematical object*, we can do all sorts of things to it. Essentially any matrix and scalar *operation* can be done. We can multiply, add, divide, subtract, etc. images. We can apply *transformations* (in shape, size, quantization, etc.). Digital images can be mathematically manipulated as we see fit… and we can do much of that with MATLAB.

Instructions:

1. Log in and start MATLAB. The first thing you should do when starting MATLAB is to navigate (using the pull-down menu at the top of the MATLAB window) to your *home directory* in the class folder in order to set it as your *working directory*.

   In what follows, *commands* to type into the MATLAB command line are shown in `this font`. Throughout this document, terminology you should understand the specific meaning of are given in *italics*.

   Some notes on help… If you type `help` followed by the name of a command, you will get a detailed description of the command, what it does, and how to use it. **When you have a question, using `help` should be the very first thing you do.** Make liberal use of this function, as it is the best way to explore and learn to use MATLAB. Notice also that MATLAB is *case-sensitive*. Also, pay close attention to error messages that MATLAB gives you – **do not ignore them**. Do your best to interpret them – as with `help`, reading and understanding the error messages should be the very first thing you do when trying to troubleshoot.

2. Create a matrix with the following *line*:

   ```
   A = [ 1 , 2 ; 3, 4 ]
   ```

   Note that *commas* (`,`) separate elements and *semicolons* (`;`) separate rows (the commas can be omitted and it will still work). How many rows does this matrix have? How many columns? Note that this matrix is 2-dimensional because it has rows and columns (e.g. the location of each number in the matrix is defined by its row and column). What would a 3-dimensional matrix look like? A 4-dimensional one?

3. Calculate the *"square"* of your matrix. This could mean two very different things:

   ```
   Asquare1 = A .* A
   Asquare2 = A * A
   ```

   Note that using `.*` (as opposed to simply `*`) forces MATLAB to perform the operation *element-by-element*. With `*` instead what happened? Why? Think about the difference between conventional multiplication (element-by-element in the case of a vector or matrix) and *matrix multiplication*. I make a habit of always using `.*` unless I specifically want matrix multiplication. The same holds for division (`./` vs. `/`) and for raising to a power (`.^` vs. `^`). Addition (`+`) and subtraction (`-`) are always element-by-element operations.

4. As you can see, MATLAB *prints* the result of everything you do. This can slow things down, however, especially in a long program. You can suppress the *output* by placing a semicolon at the end of the line. If you type:

   ```
   B = [ 5 , 6 ; 7 , 8 ] ;
   ```

   the vector `B` will not be printed. If you now type:

   ```
   B
   ```

   the vector will be printed. The first line assigned the variable B to the vector of numbers, so that from then on you can refer to those numbers by the variable B. Note that we did the same in step 3 when we defined the variables `Asquare1` and `Asquare2` to contain the results of mathematical operations by writing an equation.

5. Now let's do a little more manipulation and math using MATLAB. Recall the difference between regular arithmetic and linear algebra operations. Let's make some more variables. First, we define, a 2-dimensional matrix:

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

   Now a 1-dimensional matrix, or *vector*, with one row and several columns:

   ```
   D = [ 1 , 3 , 5 ]
   ```

   And define another vector, this time with several rows and one column:

   ```
   E = [ 2 ; 4 ; 6 ]
   ```

   Remember what the commas and semicolons tell MATLAB.

   And, finally, create a single number, or a *scalar* (one row and one column):

   ```
   F = 9 ;
   ```

   We can ask MATLAB about the variables in memory right now. Type:

   ```
   whos
   ```

MATLAB should list all the variables in memory right now, in addition to some basic information about them (data type, size, etc.). Note in particular the sizes that MATLAB reports.

To get rid of specific variable(s) from memory, for example `Asquare2 and Asquare2`, type:

```
clear Asquare1 Asquare2
```

Try `whos` again to check what happened. To clear all variables from memory (don't do this now), you can type `clear` without any arguments (no need to list them all). Note that if you just want to clear the display (the command line can get cluttered and messy after a while) just type `clc`. The window will be wiped clean, but memory will remain intact.

Now let's do some math. Try the following commands. Note that, for simplicity (and to save typing and memory space) we are not assigning the results of these operations to variables (although we could by writing them in equation form). First try:

```
A .* B
```

What happened? Why did you get the answer you got? Try this instead:

```
A * B
```

What happened? Why did you get this answer this time (as compared to what you got before)? Hint: see step 3. Now try this:

```
B * A
```

Did you get something different? Would that be true in general? Why or why not? How about this:

```
A .* C
```

What happened? Why? And this:

```
A * C
```

What happened? Why? Remember that for some operations to work, the matrix sizes must be compatible. Let's look at another example of that which is specific to matrix algebra. Define another matrix, this time not a square one:

```
G = [ 1 2 3 ; 4 5 6 ]
```

To determine its size, we can use `whos`, or, alternatively, we can use the `size` command:

```
size G
```

Now try this (note we are *not* doing element-by-element multiplication):

```
A * G
```

What did you get? Why? Would this work:

```
G * A
```

Why? Try each of the following (or any others you want) and see what happens (don't type these all at once – do them one at a time):

```
F .* C
F * C
D * E
E * D
B * C
B .* C
```

There are other permutations, but you probably get the point now.

6. So far you have made and manipulated small 1- and 2-dimensional matrices. As alluded to earlier, matrices, however, can be of arbitrary size and dimension. As you now know, a single number is called a *scalar*. And, as you have seen, a single row or column is a *vector*. A two-dimensional *matrix* will have m x n (rows x columns) elements. Three-dimensional (m x n x p) and higher-order matrices (cubes, tensors, or whatever you want to call them) are all possible, too. Large (in terms of size and dimensionality) matrices can be easily constructed in MATLAB without typing in every element. Below are some examples. Try them and see what they do. (Before doing so though, you will need to define the *matrix dimensions*, `m` and `n` and so on, scalars which denote the number of rows and columns, respectively). For example:

```
H = ones( m , n )
I = 0.5 .* ones( m , n )
J = zeros( m , n )
```

Does the above work without first making `m` and `n` equal to something?

7. *Diagonal matrices* are also easy to create:

```
L = diag( v , k )
```

Again, before using this command, you need to define the vector `v` that you want on the diagonal of the square matrix `L`. The parameter `k` tells MATLAB which *diagonal* of `L` to put the vector `v` on: `k` = 0 is the main diagonal, `k` = 1 is 1 spot above, `k` = -1 is one spot below, etc. Vector `v` should have at least 2 elements (otherwise it would be a scalar). You may want to play with the size of `v` as you vary `k`. Note that MATLAB automatically adjusts the size (number of rows/columns) of `L` size depending on how you specify `k`.

8. To create vectors, you can type all the elements in manually, or use the tricks you learned in step 6. The following example will create a vector of 1's, of `m` x 1 dimensions:

```
P = 0.9 .* ones( m , 1 )
```

9. The vector, `P`, you just made is a column vector. You can make it into a row vector by *transposing* it. Transposition can be done using the `'` operator or with `transpose` command:

```
Q = P'
```

or

```
Q = transpose( P )
```

This trick also works with higher-order matrices. Try it with some of the variables you have in memory right now. (Note one convention, that we haven't really been following is to use lower case variable names for vectors and scalars and upper case for higher-order matrices.) If you use your matrix `L` (which you will note is still in MATLAB's *memory*) you get an interesting transposed matrix (e.g. `L'`), provided that vector `v` that you used to create it was not located on the main diagonal. Otherwise the transpose won't do very much – can you see why? So you may need to re-define `k` and recreate `L` to make the result of transposing `L` more interesting.

10. Now, the power of MATLAB is that you can string together commands (MATLAB's built-in *functions*) into pre-written *programs* (your own functions, each stored in a simple *text file* called a `.m` file) that you can invoke with a single command. You can write functions using MATLAB's built-in *text editor* (which is very useful because it color-codes your program), but you can use any other text editor. Just make sure you name your program using the `.m` suffix. Moreover, these functions can be used on any *platform* that MATLAB runs on (Windows, Mac OS X, UNIX, etc.).

Try writing the program below. It computes the *mean* and *standard deviation* of a *series*:

```
function [ mean , stdev ] = stat( x )
% Comments look like this (they start with a percent sign).
% They are not interpreted by MATLAB – they are ignored.
% They are handy for putting notes in your program.
% The "function" line above defines your program
% as a function that accepts input (x) and
% returns two outputs (mean and stdev).
% The name of the function is stat.
% You invoke the program by typing stat(x) at the
% command line.  The program will then compute
% the mean and standard deviation of x.

% The next line returns the number of rows and
% columns in x.

[ m , n ] = size( x ) ;

% Below is a structure called an if-then-else
% statement. The double equal sign does not
% assign a value to m.  It is a test of equality.
% Indenting makes your program easier to read.

if m == 1
      m = n ;
end

% What do you think these next two lines do?
% What about the lack of semicolons?

mean = sum( x ) ./ m

stdev = sqrt(  ( sum( ( x - mean ) .^ 2 ) ) ./ ( m – 1 ) )

% At this point the program stops and will have
% returned two values, the mean and standard deviation.
```

(Note the use of comments in the example – comment your code extensively when you write your own programs!) When you are done typing it in, save this program in your working directory as `stat.m`. But don't run it yet.

11. Again using MATLAB’s text editor, write a second program (see below) and call it `quart.m`. Save it, but don't run it yet.

```
function [ quart1 , quart3 ] = quart( x )
% This function returns the first and
% third quartiles of the vector x.

ymed = median( x ) ;

% What do these lines do?

y1 = x( x <= ymed ) ;
y2 = x( x >= ymed ) ;

quart1 = median( y1 )
quart3 = median( y2 )
```

12. In the MATLAB command window, type:

```
clear
```

This will clear all the variables currently in memory. If you type:

```
clc
```

you will clear the command line window so that you have a fresh slate. Don't worry, though, there are ways to *save* variables in memory to disk, too!

13. Now create a series of numbers stored in a vector:

```
x = [ 1 , 48 , 81 , 2 , 10 , 25 , 14 , 18 , 53 , 41 ]
```

14. Run the `stat.m` program. Remember, this program that we wrote is now a MATLAB function, thus it needs an *argument* to execute the desired computations. Since we want to obtain statistics for vector `x`, we run the command:

```
stat( x )
```

What happened? Why?

15. It turns out that MATLAB has a lot of built-in functions and *toolboxes*. These include calculator functions like `sin`, `cos`, `tan`, etc. but also include a host of more complex things. Explore the MATLAB help and documentation to see what is there. Among these are built-in functions that will calculate the mean and the standard deviation (both are part of the Statistics Toolbox). Use them to check the results of your `stat.m` program:

```
mean( x )
std( x )
```

16. Another built-in statistical function is the *median*:

```
median( x )
```

This should give you the number halfway between the largest and smallest in the series. An intuitive way to check the output of `median` is with:

```
y = sort( x )
```

which will return a vector `y` that includes the elements of `x`, sorted from largest to smallest. Since there are 10 elements in `x` (and in `y`), the median value is halfway between the 5th and 6th elements of `y`.

17. Now try your `quart.m` program. To check the answers, look at your vector `y`. The first quartile should be the 3rd element in `y`. The third quartile should be the 8th element in `y`.

18. Now try writing your own program to calculate the median of a series. Note that it should work for both an even and odd number of elements, so think about and implement both *cases*! Make sure you save it in your directory with your other programs.

19. Here are some notes on working with an image (basically just a matrix).

a. MATLAB has built-in functions (through the Image Processing Toolbox) for *opening*, *displaying*, and *writing* images. See the handout and this url for more information:

http://amath.colorado.edu/courses/4720/2000Spr/Labs/Worksheets/Matlab_tutorial/matlabimpr.html

However, some of the images you may work with are in *file formats* other than those that the functions discussed at that website can handle (e.g. *ENVI format*). Here is a snippet of *code* for opening an ENVI format image:

```
fid = fopen( ' myfile ' , ' r ' , ' l ' ) ;
myimage = fread( fid , [n , m] , 'uint8' ) ;
fclose( fid ) ;
```

The first line opens the *file* containing your image. The `fid` is a number that MATLAB assigns to the file as a reference. The second line reads the file and puts the *contents* into the variable `myimage`. The variables `n` and `m` denote the *size* of the image in rows and columns and `uint8` denotes the *type* of data (8-bit *unsigned integers* in this case). The third line *closes* the file.

Similarly, to *save* an image in ENVI-format:

```
fid = fopen ( 'outputfilename' , ' wb ' ) ;
fwrite( fid , myimage , 'int16' ) ;
fclose( fid ) ;
```

See the MATLAB help for more information on `save`, `load`, `fopen`, `fread`, `fwrite`, and `fclose`.

b. To display images (or any other data) as a *figure*, MATLAB has tons of built-in functions. Here is a snippet of code with just one example:

```
figure( 1 ) % Makes a new figure window.
hold off        % Allows the figure to be
                % overwritten.
imagesc( myimage )      % Plots the image.
colormap( pink )  % Sets the colormap.
```

```
axis image        % Formats the axes.
title( ' My Image  ' )  % Sets the title.
xlabel( ' column ' )    % Labels the x axis
ylabel( ' row ' )       % Labels the y axis
hold on           % Prevents the figure
                  % from being overwritten.
```

There are many, many more options and functions for plotting (e.g. `plot`, `subplot`, `image`, `colormap`, `pcolor`, `shading`, `demcmap`, etc.) Explore the help for MATLAB and the Image Processing Toolbox (and others) for more.

20. *Indexing* is an **extremely** important and useful part of MATLAB. The reading has an extensive discussion of how to do it. Here is a little example that selects only the first row of some random matrix `mymatrix` (note that you'd need to first define `mymatrix` for this line to do anything for you):

```
firstrow = mymatrix( 1 , : )
```

The `1` in the argument denotes the first row and the *colon* (`:`) indicates all of the column elements of that row. How would you change this syntax to select some other row? How about to select, instead, a certain column? Or to select just a part of a row (or column)? How would you select a square or rectangular subset of a matrix?

21. Let's say your matrix is called `mymatrix`. You can *rotate* it by 90 degrees with this command:

```
mymatrix = rot90( mymatrix )
```

(Again, this line won't work unless you have a matrix called `mymatrix` in memory already.) Note that this line will *overwrite* the original `mymatrix` with the rotated one since you are assigning the result back to the original variable. This can be a handy technique for minimizing the creation of new variables and for efficient use of memory, but it can cause problems if you aren't careful!

22. Using the snippets of code above and the functions you've learned about in this lab (and in the reading), write a MATLAB program to do the following:

c. Open the ENVI-format image `test.img`. This file is available in the *`instructors`* directory in the class folder. Copy it to your working directory before working on it!
d. Display the image (correctly – transposition will be required) and give it a colormap.

- e. **Calculate the mean, standard deviation, and median pixel values.**
- f. **Cut out a 100 x 100 square of pixels from the lower right corner and save the *subsetted image* as a new ENVI-format image.**
- g. **Open and display the new, subsetted image (correctly) and give it a colormap.**

23. Now for some more advanced concepts: conditional statements; loops; graphics; user input; and file I/O.

    a. Conditional statements. Sometimes you need to make decisions. These decisions can alter the course of your program and make it do different things depending on the situation. One way to do this is with a `if-(else)-then` structure. Here is a simple example:

```
%Example of using an if-(else)-then statement
%(and some other useful things)

clear %clear memory
clc %clear the display

n = 1 ; %size of the matrix we will create
%keep it n = 1 for this program to work
%would need modification if we chose other values for n

x = rand( n ) ;
%generates n x n matrix containing numbers between 0 and 1

x = 10 .* x ;
%multiply by 10 to get a number between 1 and 10

x = round( x ) ; %round to nearest integer

x %just prints x to the display so that we can see it

if x == 0
%this checks if x is exactly equal to 0
%it does not set x equal to 0
       disp( 'This number is exactly 0!' )
       %prints a message to the screen
       y = x + 10 ; %we can do something random here

elseif x > 0 && x < 5
%checks if x is greater than 0 AND if it is less than 5
%note the use of relational (>, < in this case)
%and logical operators (AND in this case)
%OR would be written:||
%==, >, <, >=, <= are all valid relational operators
       disp( 'This number is smaller than 5 (but not 0)!' )
       y = x .* exp( i .* pi ) ; %exponentials… too easy!

elseif x == 5 %checks if x is exactly equal to 5
       disp( 'This number is exactly 5!' )
       y = sqrt( x ) ; %square root… why not?

elseif x > 5 %checks if x is greater than 5
```

```
        disp( ‘This number is greater than 5!’ )
        y = ( ( 2 .* x ) .^ 2 ) + x ; %quadratic… yawn

else
%I’m not sure what other case there can be…
%but this covers the unexpected!
        disp( ‘I should never get here… but here I am! ‘ )
        y = 1e6 ;
        %say this number with your pinky to your mouth

end
%all flow control structures must have an end
%also note the indenting used in this program
%(omitting the comments makes the indenting clearer)
```

b. Loops. There are two common types of loops: `while` and `for`. This example uses both.

```
%Example of using loops
%(and some other useful things)

clear
clc

i = 0 ;
j = 0 ;
%i and j are counters that we have created and initialized to 0
%loops work by counting and incrementing these counters
%each time through the loop, the counter(s) should be changed
%when the counter reaches the right value, it triggers a behavior

A = zeros( 1000 ) ;
%we make a big 1000 x 1000 matrix of zeros to put stuff into later

while i <= 1000
%i started at 0 and we do whatever is in this loop until i is 1000

        disp( ‘This is my ‘ , num2str( i ) , ‘ time through the
        while loop!‘ )
        %what does this do? look up disp and num2str…

        n = 5 ; %a constant we will use below

        for j = 1 : n : 1000
        %this for loop is nested within the while loop
        %this entire for loop is done in each while iteration
        %j is reinitialized to 1 and increments by n each time
        %the for loop terminates when j reaches (or exceeds) 1000

                str = sprintf( ‘This is my %d through the for loop!’
        , j ) ;
                disp( str ) ;
                %what does this do? see the help for disp…

                k = i + j ;
                %some random, meaningless math using our counters
                %more commonly, the counters are used to index (see
        19)
                %like this…

                A( i , j ) = k ;
                %we store our meaningless math result in specific
        location
        end %end of the for loop; note the indenting

        i = i + 1 ;
        %this is a critical line
        %it increments i by 1 each time through the while loop
        %what would happen if we didn't do this?

    end % end of the while loop; note the indenting
```

c. Graphics. MATLAB is great for making plots. Look at the help for `plot`.

d. User input. MATLAB can include code for asking the user for input. Look at the help for `input`.

e. File I/O. MATLAB can read and write files of a variety of different formats. Look at the help for the Image Processing Toolbox for the various supported image formats. Also look at the help for `fopen`, `fread`, `fwrite`, `fclose`, `save`, `load`, and other related commands.

24. Write a program that reads in an image and tests each pixel sequentially for whether the value stored there is greater than, equal to, or less than the average brightness value in the image. The result should be recorded as either -1, 0, or +1 depending on if the tested pixel is less than, equal to, or greater than the image average, respectively. Results should be stored in a new matrix the same size as the input image. The result of testing pixel `(i,j)` in the input image should be stored at location `(i,j)` in the results matrix, e.g. the same location. This program will require you to work with an image, calculate statistics, exercise indexing, use conditional statements, and implement loops.

25. <u>(You don't have to write MATLAB code for this step)</u> In this step, you will use ENVI to open and display both `test.img` and your subsetted image (output of your program from step 22). ENVI is also installed on the PCs and can be launched from the desktop icon or the Windows Start Menu. When it launches, you will see the ENVI menu bar. Explore the menus and the online help (ENVI is fairly intuitive, although some functions are buried). There are also a set of tutorials you may have worked with if you took the remote sensing course from me. For now, you just need to familiarize yourself with the user interface and how to open, display, and close images. Try opening your images using the command under the *`File`* menu. (Note that ENVI format includes two files: the image data itself and an accompanying `.hdr` file that contains metadata such as number of rows and columns, etc.. The `.hdr` file is text file you can open in a text editor. When opening an image in ENVI, you should choose the image data file, not the `.hdr` file.). Note that opening an image file in ENVI simply reads the image file into ENVI's memory – it does not display automatically! Instead, you will now see a new window called `Available Bands List`. The image you opened should be listed there. You can now display the image by highlighting the *band* containing it, specifying a grayscale *display* in the dropdown menu, and clicking the `Load` button. Note that three windows will now appear, which together comprise a display: `Scroll`, `Image`, and `Zoom`. `Scroll` shows the entire extent of the image. `Image` shows the image at full resolution. `Zoom` shows a magnified view. Note that `Zoom` and `Image` can be resized to adjust their extent. Also note that the extent of `Zoom` is shown in `Image` by a red box, and the extent of `Image` is shown in `Scroll` by a red box. Resizing `Scroll` changes the resolution of the image displayed in the window. Note that `Scroll` has its own menu bar with commands. Among these are commands for manipulating the display (not necessarily the actual data!). The main ENVI menu bar has commands for manipulating the actual data. Experiment with ENVI on your own.

What to Turn In and How:

You will turn in and demo your code for (a) the median calculation **(step 18)**, (b) your program for the image processing **(step 22)** and (c) your "loop" program **(step 24)**. This is a total of three, separate MATLAB programs. **Make sure your code is commented well enough for me to understand what you are doing! Submit your programs electronically** in three, separate `.m` files (NOT Word files! NOT hardcopy! One file for each program) that can be executed in MATLAB. Your code should be turned in via the *`drop box`* in the class folder. Do not turn in either your input or output image from MATLAB. For your experimentation with ENVI, please also electronically submit **screen-captures of your ENVI displays** (as reasonably-sized JPEGs!).