CS100R: Matlab Introduction

August 25, 2007

1 Introduction

The purpose of this introduction is to provide you a brief introduction to the features of Matlab that will be most relevant to your work in this course. Even if you have experience with Matlab, we still recommend that you review this handout, because it does discuss some functionality that is specific to CS100R and not Matlab in general.

You should try actually entering any code that you see in type-writer style into the Matlab interpreter (discussed below).

```
>> code like this should be entered into the Matlab interpreter
```

If you have any problems or are confused about any part of this lab, the TA staff is available to help you with this, either during regularly scheduled lab times or whenever they are in the lab.

2 Getting Started with Matlab

2.1 Lab Use

Matlab is available on all of the lab computers. If you have not yet received a lab user account, you will need to speak with the lab TA's during your first lab period, and accounts will be established.

To startup Matlab, you can simply double click the icon on the desktop.

2.2 Matlab Syntax

2.2.1 Matrices

Matlab represents images (and many other things) as matrices. Recall from class that a matrix can be though of as a table (or a 2 dimension array), which consists of rows and columns. An entry (or cell) is addressed by providing its row and column. The matrix below demonstrates how we would refer to each cell by simply providing the row and the column.

$$\begin{pmatrix} 1,1 & 1,2 & 1,3 & 1,4 \\ 2,1 & 2,2 & 2,3 & 2,4 \\ 3,1 & 3,2 & 3,3 & 3,4 \\ 4,1 & 4,2 & 4,3 & 4,4 \end{pmatrix}$$

In Matlab, a new matrix can be created with the `zeros` or `ones` functions.

Both functions take two function parameters, or arguments. The first is the number of rows the new matrix should have, while the second is the number of columns.

You can experiment with this at the Matlab command-line (this command line is in the largest portion of the Matlab main window, and is marked by the "`>>`", which prompts you to enter information).

Note that a vector (which is the Matlab version of an array), is simply a sepecial case of a matrix: it is a matrix with one column and many rows.

You should try creating a new matrix, editing, and printing entries. For example, try the following code:

```
>> x = zeros(5, 3)
>> x(1,1) = 5
>> x(2,1) = 3
>> x(3,1) = 1
```

Notice that when using the command line, the variable which you are assigning to is printed out to the command window after it is assigned to. Matlab will do this whenever you assign to a variable. If you want to surpress this output (which you most likely will want to when writing Matlab functions), you can end a line with the semi-colon (;).

```
>> x(3,2) = 10;
>> x
```

Notice that just typing the name of the variable without a semi-colon causes Matlab to print it to the screen.

Note that unlike many other languages, matrices in Matlab are one-indexed, rather than zero-indexed. This means that the first entry in the matrix is (1,1), rather than (0,0) as it would be in many other languages (C, C++, C#, Java, Perl, etc.). This is an important difference that you will need to be mindful of if you have experience coding in another language.

Unlike other languages, Matlab also allows you to perform operations on an entire matrix. For example, try:

```
>> x * 5
>> x > 0
```

Notice that in the first line, we multiplied the entire matrix x by a scalar (a non-vector, non-matrix quantity). Multiplication is an arithmetic operation.

Matlab supports all of the standard arithmetic operations addition (the + operator), subtraction (the - operator), multiplication (the * operator) and division (the / operator).

```
>> y = 6
>> y + 3
>> y * y
>> x / y
```

The second line (`x>0`) is a logical operation: it has a value of either true or false (a cell is either greater then 0, or it is not). Matlab represents false as 0, and true as 1 (although any non-zero value will evaluate to true in Matlab, just like in C/C++).

2.3 Functions in Matlab

2.3.1 Function Basics

Matlab provides functions (you may have heard them called "methods") just like many other languages. Recall that the purpose of functions is to encapsulate code that you plan to use in many places in your code. Unlike other languages though, Matlab is not object oriented, so functions simply exist in a global namespace. That is, you can call any function with out a class name in front of it.

```
>> fprintf('you can call the fprintf function to print out a string\n')
```

Where functions take multiple arguments, they are delimited by a comma just as in many other languages. Matlab contains many built-in functions that allow you to do interesting things without having to write any code yourself. You can find out more about what a function does, using the help command on the command line. For example:

```
>> help sum
```

will provide you with a variety of important information about the `sum` function, including its input and output arguments. Observe that functions in Matlab can return more than one value (unlike many other languages that you may have used).

For an example of this, look at the built-in help for the `find` command. You will notice that the meaning of the output arguments changes based on how many of the arguments you receive (assign to a value). To assign to multiple output arguments, you can use the following notation:

```
>> i = eye(5)
>> [J,I] = find(i);
```

2.4 Matlab Image Representation

Recall from lectures that color images are represented as a series of pixels. For each pixel, we have a value for its red, green and blue component.

To represent this information, Matlab uses a 3-dimensional matrix. To make it easier to access this information we provide you several functions.

2.4.1 image_rgb

The `image_rgb` function is designed to make it easy to get each of the 3 image "channels," the red, green amd blue components. A call to `image_rgb` would look like:

```
>> [R,G,B] = image_rgb(my_image);
```

Where `my_image` is a handle to an image (we will show you how to get a handle to an image in the next section).

Matlab treats the top left-hand corner as location $\langle 1, 1\rangle$, with the x axis increasing as you go right (like ordinary cartesian coordinates) and the y axis increasing as you go down (the opposite of cartesian coordinates).

Each of the matrices `R,G`and `B` are the same size, and each of the values for a chanel at a certain pixel will represent an intensity between 0 and 255, with 255 being the brightest, and zero being the darkest. Thus, the color white is $\langle 255, 255, 255\rangle$ (255 red, 255 green, 255 blue) while the color black is $\langle 0, 0, 0\rangle$.

We can actually get the intensity of the different channels of the pixel at $\langle 1, 1\rangle$ by executing:

```
>> [R,G,B] = image_rgb(my_image)
>> red = R(1,1)
>> green = G(1,1)
>> blue = B(1,1)
```

2.5 Matlab Image Manipulation

Images in Matlab can be loaded with the `imread` function. This function takes a string argument, which is the path to the image to open. Try the following code:

```
>> img = imread('m:\imgs\lab0\part1\wand1.bmp');
>> [rows, cols] = image_size(img)
```

The above code snipped actually opened an image, and then determined its size. The matlab `image_size` function actually returns an array, and the syntax that was used above binds rows to the first element of the array and cols to the second.

2.6 Black and White Images

Matlab also has support for black and white images (also called binary images). We use black and white images to actually represent the output of thresholding (each pixel either meets the threshold criterea, in which case it has a value of 1 (on), or it does not and has a value of zero (off)).

Unlike color images, black and white images can be represented by two-dimensional matrices: instead of having multiple components for each pixel (as in color where each pixel had a red, green and blue value), each pixel has only one value (1 or 0).

Thus a new binary image can be created by doing the following:

```
>> cols = 420;
>> rows = 300;
>> bw_img = zeros(rows, cols);
```

And thus individual pixels can be addressed by doing:

```
>> bw_img(1,1)
```

We also provide a function to return a 2-dimensional array from an image handle:

```
>> my_image = imread('m:\imgs\lab1\part1\wand_thresholded1.bmp')
>> bw = image_bw(my_image)
>> bw(1,1)
```

This code snippet opened a black and white image and then loaded it into the matrix `bw`. Finally, we checked the top left pixel to see if it was on or off (it should be off).

2.6.1 find

One important function to experiment with in Matlab is the `find` command. You can use the `find` command to determine which pixels are selected in the binary image:

```
>> bimage = imread('m:\imgs\lab0\part2\wand_thresholded1.bmp');
>> image(bimage);
>> [ysel,xsel] = find(image_bw(bimage));
```

This will return two vectors, `ysel` and `xsel`,each of the same size. Each entry represents one component of a selected pixel. For example:

```
>> num_selected = length(ysel)
>> length(xsel)
>> x = xsel(1)
>> y = ysel(1)
```

From this, we can tell that the pixel $\langle x, y \rangle$ is selected (has a binary value of 1). The same will be true of any pixel with coordinate $\langle xsel(n), ysel(n) \rangle$, for any choice of n.

2.6.2 Vector Iteration

Once you have a vector of values, you will probably want to iterate over it. Recall that you can use Matlab's `for` loop over all values. To experiment with this, and Matlab conditionals, we will build a function that will count the number of entries over 50 in a Matlab vector. Remember that since we are making a new Matlab function, we will want to place it in a new file called `count50.m` in the current directory.

```
function [ total ] = count50(vect)
   total = 0;
   for i=1:length(vect)
      if(vect(i) > 50)
         total = total + 1;
      end
end
```

You should already be familiar with most of the concepts used in this code. It simply loops over all of the elements of a vector (recall the discussion of the for loop in the previous lab). The Matlab built-in `length` function is used to determine how many elements are present in the vector. Where `count_50` finds an element over 50, it increments the `total` variable.

Remember that since the last value of `total` is what is returned by the function, the fact that we use total as both a temporary value (to hold the intermediate values of our count) and as the final return value is perfectly legal.

2.6.3 Matrix Iteration

Now, suppose we wanted to build a slightly more complicated function to count what percentage of the image our wand occupies. We can do this two ways, and we will demonstrate both. Here we assume that our function takes as input a binary image. If you would like to get a binary image, to play with, you can load it using the `imread` function in Matlab. There are a variety of binary images on the `M` drive, under `M:\imgs\lab0\part2\wand_thresholded*.bmp`.

```
>> bimage = imread("m:\imgs\lab0\part2\wand_thresholded1.bmp");
>> image(bimage);
```

The first possibility is to extend our `count50` example to actually perform two dimensional iteration rather than just one:

```
function [ fg_perc ] = percent_wand(bimage)
   [ rows, cols ] = image_size(bimage);

   tot = 0;
   for y=1:rows
      for x=1:cols
         tot = tot + bimage(y, x);
```

```
    end
end

fg_perc  = tot / (rows * cols);
```

What we did here was nest two seperate `for` loops, the outer one to count over each row, while the inner one counted over each column. Now, instead of using an `if` statement to see if a certain pixel was selected, we instead just added its value to `tot`. Observe that if a pixel is selected (has a value of 1), it will increase the value of `tot`, while if it is not selected, it will not increase the value.

Finally, to obtain a percentage, we divide `tot` (the number of selected pixels) by the total number of pixels.

We can, however implement this function with much less code using the Matlab builtin `find` that we spoke of before. Observe:

```
function [ fg_perc ] = percent_wand2(bimage)
   [rows, cols] = size(bimage);
   [j,i] = find (bimage);
   fg_perc = length(j) / (rows * cols);
```

Here we took the fact that `find` retured to us two vectors which told us the y and x coordinates of each selected pixel, and simply took the length of one of these vectors.

2.7 Defining Your Own Functions

In addition to calling existing functions, Matlab allows you to define your own custom functions. Matlab requires that functions be defined in a file that has the same name as the function name, with a `.m` extension. Matlab will search what it calls the *PATH*, which we have set for you. The *PATH* is a list of locations that Matlab will search for functions when they are called. Luckily the current directory is always included in Matlab's PATH, so if you want to define your own function, you can just put it in the current directory (the current directory is indicated by the "Current Directory" dropdown at the top of the main Matlab window).

To create a new function, we should first create a directory to contain some of the work that we have done in this lab. When working in the CS100R lab, **you should always save your work on the *H* drive**. If you do not do this, there is no guarentee that you will be able to retrieve your work.

Having created a new directory, and having changed to the new directory with the "Change Directory" dropdown at the top of the screen, you should now right click in the top left pane which shows the contents of the current directory. From the menu select "New" and then "M-File". You will now be able to name the file. We are going to create a function called `myplus` , so you will want to call the file `myplus.m`.

Now double click on the newly created file. Upon opening it, you will see the following:

```
function [ output_args ] = Untitled1( input_args )
%UNTITLED1 Summary of this function goes here
%   Detailed explanation goes here
```

For our function, we will need to change several things:

- Change the function name from `Untitled1` to `myplus`.
- Change the output arguments from `output_args` to something that makes sense for our function.
- Chagne the input arguments from `input_args` to something that makes sense for our function.

The first change is quite easy to make. As for the second, we are going to have a single return value called `sum`, since this function is going to sum two input arguments.

The third change is also quite easy, we will need to replace `input_args` with our own input arguments. Since our function is doing something quite simple, we're just going to have to input argument arguments, `a` and `b`.

We will also put an explanation in so that our function works with the `help` command. Finally, we want to actually fill in the body of this function, to add together the two input arguments. We're left with the following:

```
function [ sum ] = myplus( a, b )
%SUM Add together two arguments
%   The myplus function adds together its two input arguments
%   and returns this value as sum.
   sum = a + b;
```

We can try our new function on the command line just as we would try any other builtin function.

```
>> myplus(3, 2)
```

Notice that unlike other languages, Matlab does not have an explicit `return` statement to return a value. Instead, values are returned by assigning to the variables that you named as output arguments. In our case, the `sum` variable was defined to be an output variable, so whatever value it has at the end of the execution of the `myplus` function is returned.

3 Conclusion

Again, if you have any trouble with any of the content in this tutorial, you should as a TA for help. We will be relying on this material heavily throughout the course of the semester, so make sure you are very comfortable with it now.