CS1114: Matlab Introduction

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 CS1114.

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 course 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 TAs during your first lab period, and accounts will be established.

To start up Matlab from the Linux machines in the lab, open up a terminal (Applications->Accessories->Terminal) and type matlab at the command prompt. Once Matlab has started, you can enter commands (for instance, the example commands given below) on the Matlab command-line. The command line is in the largest portion of the Matlab main window, and is marked by the "`>>`", which prompts you to enter information.

In order for Matlab to be able to use the code that the course staff provides for you, you need to add the course directory to Matlab's path. To do that go to File->Set Path and then click Add with Subfolders and add the directory /courses/cs1114/lib.

2.2 Matlab Syntax

2.2.1 Variables

Creating and assigning to a new variable in Matlab is done with the "equals" operator ('='):

```
>> x = 2

>> y = 2.56

>> myvar = 'hello world'
```

The commands above assign different values to the variables `x`, `y`, and `myvar`. Note that there is no need to declare variables as in other languages such as Java (Matlab is a *dynamically-typed* language). A variable can take on several types of values, including:

- **Numbers**
  ```
  >> x = 2
  >> y = 2.56
  ```
- **Strings of text**
  ```
  >> myvar = 'hello world'
  ```
- **Matrices (described in the next section)**
  ```
  >> mymatrix = [ 1 2; 3 4 ]   % Creates a 2x2 matrix
  ```

Note that the '`%`' (percent) symbol denotes that the rest of the line is a *comment*—a bit of text that has no effect on what Matlab does, but can help someone reading the code understand what it is doing (for this reason, comments are an important part of computer programs).

When using the command line, the variable to which you are assigning is printed out to the command window after the assignment occurs. For instance, if you type:

```
>> x = 10
```

Matlab prints:

```
x =
```

```
10
```

Matlab will do this whenever you assign to a variable. If you want to suppress this output (and you most likely will want to do so when you are writing Matlab functions), you can end the line with a semi-colon (`;`).

```
>> x = 10;
>> x
```

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

2.2.2 Matrices

Matlab represents images (and many other things) as matrices. A matrix can be thought of as a table (or a two-dimensional array) consisting of rows and columns. An entry is addressed by provided its row and column. The matrix below demonstrates how we would refer to each cell by its (row, column) pair.

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

In Matlab, a new matrix can be created with the functions `zeros` and `ones`. Both functions take two parameters, or arguments. The first is the number of rows the new matrix should have, while the second is the number of columns.

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

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
```

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). 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 expression: it has a value of either true or false (a matrix entry is either greater than 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, as you may have come to expect in other languages).

2.3 Statements in Matlab

The commands in the previous sections are examples of simple kinds of *statements*, such as assignment statements (for instance '`x = zeros(5, 3)`' or '`x(1, 1) = 5`') and arithmetic expressions (for instance '`y + 3`'). A Matlab program is made up of a sequence of statements; with simple statements such as these, the statements are simply executed in order. However, Matlab also supports more complex statements that control a program's *flow*, that is, the order in which statements are executed. Two particularly important types of statements are *if-statements* and *loops*.

2.3.1 if-statements

An if-statement allows you to execute different sub-statements based on a condition. The most basic type of if-statement has the syntax:

```
if <logical-expression>
  <statements>
end
```

When an if-statement is executed, if <*logical-expression*> is true (i.e., 1), then <*statements*> are executed. If <*logical-expression*> is false (i.e., 0), then <*statements*> are not executed. For example, consider this program:

```
>> a = 10;
```

```
>> b = 5;

>> if a > b

     fprintf('a is larger than b\n')

   end
```

(`fprintf` is a function for printing text to the display—see the next section). This if-statement prints out '`a is larger than b`' if (and only if) '`a > b`' is true (for this program, it happens to be true).

A second type of if-statement, called an *if-then-else statement*, has a second branch (called the 'else' branch) that is executed if the condition is false. Here's an example:

```
>> if a > b

     fprintf('a is larger than b\n')

   else

     fprintf('a is not larger than b\n')

   end
```

Each branch can contain multiple statements, for instance:

```
>> if a > b

     fprintf('a is larger than b\n')

     c = 1;

   else

     fprintf('a is not larger than b\n')

     c = 0;

   end
```

2.3.2 For loops

Another useful type of statement is a loop, which executes a set of statements multiple times. A simple type of loop is a *for loop*. A for loop has the syntax:

```
for <var> = <exp1>:<exp2>
   <statements>
end
```

A for loop executes <*statements*> multiple times, each time assigning <*var*> a different value from <*exp1*> to <*exp2*>. Here's an example of a for loop:

```
>> sum = 0;

   for i = 1:10

     sum = sum + i;

   end
```

This loop executes the statement '`sum = sum + i`' 10 times. For the first iteration of the loop, the variable `i` (called the loop index variable) is assigned the value 1; during the second iteration, `i` is assigned the value 2; and so on, until the last iteration, when `i` has a value of 10. This loop computes the sum $1 + 2 + 3 + 4 + \ldots + 10$, and assigns the result to the variable `sum`.

2.4 Functions in Matlab

2.4.1 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. Unlike other languages, though, Matlab is not object-oriented, so functions simply exist in a global namespace. That is, you can call any function without putting a class name or other qualifier 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 be a comma, 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. 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 (that is, how many of them you assign to a variable). To retrieve multiple output arguments, you can use the following notation:

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

3 Matlab Image Representation

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

To represent this information, Matlab uses a three-dimensional matrix. Representing images as matrices imposes a coordinate system, in which the top left pixel (that is, the top left entry in the matrix) is located at $(1, 1)$, with the x-axis increasing as you go right and the y-axis increasing as you go down. It's important to remember that this y-axis is *inverted* with respect to the Cartesian coordinates you're probably used to (where the y-axis increases as you go *up*).

Unfortunately, manipulating three-dimensional matrices in Matlab is a little messy. To make it easier to access color information in images, we have provided several convenience functions.

3.1 `image_rgb`

the `image_rgb` function is designed to make it easy to get each of the three images "channels": the red, green, and blue components. A call to `image_rgb` would look like:

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

Where `my_image` is a three-dimensional matrix representing an image (we will show you how to get one of these in the next section).

Each of the (two-dimensional) matrices `R`, `G`, `B` has the same size as the others, matching the two-dimensional size of the image you passed in to `image_rgb`. Each entry corresponds to the intensity of a pixel in the image within the particular channel, where 0 is the lowest intensity and 255 is the highest. For example, black is represented by red green and blue intensities $\langle 0, 0, 0 \rangle$, and white is represented by $\langle 255, 255, 255 \rangle$. You can actually get the intensity values of the different channels of the pixel located at $(1, 1)$ by executing:

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

3.2 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('/courses/cs1114/images/wand1.bmp')
>> [rows, cols] = image_size(img)
```

The above code snippet 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.

3.3 Black and White Images

Matlab also has support for black and white images (also called binary images). We use black and white images to represent the output of thresholding (each pixel either meets the threshold criteria, in which case it has a value of 1 (on), or it does not and has a value of 0 (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 red, green, and blue intensities), 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 individual pixels can be addressed like this:

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

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

```
>> my_image = imread('/courses/cs1114/images/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.

3.3.1 `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('/courses/cs1114/images/wand_thresholded1.bmp');
>> image(bimage);
>> [ysel, yxel] = find(image_bw(bimage));
```

This will return two vectors, `xsel` and `ysel`, each of the same size. Each entry represents once 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 (x, y) is selected (has a binary value of 1). The same will be true of any pixel with coordinate `(xsel(n), ysel(n))`, for any choice of `n`.

3.3.2 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 to do this. To experiment with `for` loops 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 `count50` 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.

3.3.3 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 in 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 in the directory `/courses/cs1114/images/`.

```
>> bimage = imread('/courses/cs1114/images/wand_thresholded1.bmp');
>> image(bimage);
```

The first possibility is to extend our `count50` example to actually perform two-dimensional iteration:

```
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 next two separate `for` loops: the outer one counts over each row, while the inner one counts 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 advantage of the fact that `find` returns two vectors which hold the y- and x-coordinates of each selected pixel. By simply taking the length of one of these vectors, we were able to easily determine the number of selected pixels.

4 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. When you attempt to call a function `function_name`, Matlab will search a list of directories called the `PATH` for a file named `function_name.m`. We have set the `PATH` for you to include all of our custom functions.

Happily, 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" drop-down box 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 CS1114 lab, *you should always save your work on the `H` drive*. If you do not do this, there is no guarantee that you will be able to retrieve your work at a later date.

Having created a new directory, and having changed to the new directory with the "Change Directory" drop-down box 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.
- Change 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 simple, we're just going to have two input 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 so that it adds together the two input arguments. We're left with the following:

```
function [ sum ] = myplus( a, b )
%MYPLUS 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 built-in function:

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

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

5 Source control

The code in this tutorial have been simple enough that you could just write them out line-by-line. But as you work through projects in this course (and in the future), you'll find that you'll be writing programs with multiple functions and multiple files. You may also be working with other people. This all calls for better ways to manage *versions* of your code—in other words *version control*. This is also a great opportunity to remind you that you'll find important information about the course on Piazza; there you'll find a very helpful note that Rocky Li has written about version control. Please take a look; you'll be glad you did (even if it's not obvious why now). See the note here:

`http://piazza.com/class#spring2012/cs1114/6`

6 Conclusion

Again, if you have any trouble with any of the content in this tutorial, you shouldn't hesitate to ask 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.