APPM 2460
FUNCTIONS IN MATLAB

1. Introduction

We have seen in the previous worksheet that there are many situations in which we would want to repeat a chunk of code over and over again for different inputs. For example, you wrote a script that calculates n choose k. To test our code for different n and k we used a lot of copy and paste. Wouldn't it be nice if, instead of copy-and-pasting all that code, you could write something so that when you type `choose(n,k)` Matlab would return the value n choose k?

Such a construct exists, of course. It is called a **function**. In calculus you often think of a function as a plot, or a curve on some axes. We should now begin thinking of a function as a machine that takes in some stuff (numbers, strings, whatever) and returns some other stuff. For example, the function $f(x) = \exp(x)$ takes in a number between $-\infty$ and ∞ and returns a number between 0 and ∞.

To write functions in Matlab, we create `.m` files that tell the function what to do. Up to now, we have only seen `.m` files used as scripts. A script executes a series of commands (i.e. lines of code). We will now see how we can also use `.m` files to create functions. If we think of our script as one person doing a list of commands, a function is another person who does a specific job.

Matlab has many built-in functions. One example is the `plot` function. In our analogy, the `plot` function would be a person who can create plots for you, if you tell them the two vectors, and any special options. They go off, create the plot, and come back with it. The important difference between this and simply doing the task in the main script is that **the function has no knowledge of any variables not passed in to it.**

2. Creating a function

Let's see hwo to create our own functions. First, open a new `.m` file. For the file to be treated as a function, the top line must follow the below form:

```
function [ outputs ] = function_name( inputs )
```

We can have multiple outputs or inputs, separated by commas. For today, we'll only play with functions that have a single output.

As an example, let's say that I wanted a function to compute the value of a polynomial for me. I can create the following function:

```
function [y] =  poly_eval(x)
y = x^2+5*x-10;
end
```

Create a file called `poly_eval.m`, and copy and paste the above code into it. Save it, then try running `poly_eval(10)` in the command window. It should return the value of the polynomial at $x = 10$. Once we've created and saved the function, we could also use it in another script by simply typing the line `y = poly_eval(10)`. This is equivalent to telling my friend, who knows the polynomial, to tell me the value of it at 10, then we assign it to the variable y. **Note that you MUST give `.m` file the same name as the function. Otherwise, your function will not work.**

The above example is very simple; we don't save very much energy by doing this. But consider if we know we want to plot several different quadratic functions. Then the following function could save us a lot of typing:

```
function [y] = poly_eval(a,b,c,x)
y = a*x.^2+b*x+c;
end
```

Now, we can run the function multiple times for each of the different quadratic functions we want to plot. For example, if I wanted to plot $y_1(x) = x^2 + 2$, $y_2(x) = 3x^2 - x$, and $y_3 = 2$, I could create a script:

```
close all
clear all

x = 0:0.01:10;

y1 = poly_eval(1,0,2,x);
y2 = poly_eval(3,-1,0,x);
y3 = poly_eval(0,0,2,x);

plot(x,y1,x,y2,x,y3)
```

Note that rather than turn hold on and use separate `plot` lines for each plot, I was able to call them all in one command. This is very useful sometimes, but makes it more difficult to use options on your plots such as `'LineThickness'`.

3. Anonymous Functions

If we have a one-line function (like `poly_eval` above), it may seem like overkill to create an entire `.m` file just for that one line. There is a nice way to create single-line functions within a script, and functions created in this way are referred to as *anonymous functions.*

Let's take a look at an example. We'll make a new **script** (not a function). In this script, we will make an anonymous function for $f(x) = 6\exp(x)\sin(x)$, and plot said function.

```
close all
clear all

% Here we define our anonymous function:
my_fun = @(x) 6*exp(x).*sin(x);

% Now we get to plotting
x = 0:0.01:2*pi;
plot(x,my_fun(x))
```

Notice the syntax. We define anonymous functions by

```
function_name = @(inputs) ...do stuff here... ;
```

That is to say, if we wanted to rewrite our polynomial plotting script above using anonymous functions, we could do so as follows:

```
close all
clear all

% Define anonymous function:
ploy_eval = @(a,b,c,x) a*x.^2+b*x+c;

% Plot stuff:
x = 0:0.01:10;
y1 = poly_eval(1,0,2,x);
y2 = poly_eval(3,-1,0,x);
y3 = poly_eval(0,0,2,x);
plot(x,y1,x,y2,x,y3)
```

Next week you will see anonymous functions used in the process of numerically solving differential equations.

4. Naming Functions

It is important to realize that we **cannot** give our functions name like `plot` or `factorial`, which are already used as default functions in Matlab. We also should not name our scripts this, or use these terms as variable names. If I want to make a function that calculates a factorial, I'll call it `my_factorial`, so that it does not conflict with the built-in Matlab function.

5. Homework

Write a function that, given a list, returns the size of the list, the maximum and minimum numbers in that list, and prints the string `'Go Buffs!'`. To get started, your function should use the opening line

```
function [ list_len, list_max, list_min ] = my_function( input_list )
```

Once you have built this function, write a script with examples of the function running on two separate lists. This script should assign the outputs of the function to variables, and then display them. An example of assigning the output of this function would look like

```
[l_len, l_max, l_min] = my_function([0 -1 10 2 pi])
```

After running this line, the variable `l_max` shoudl have the value `10`, and so on. You can then use `disp` to show the values of these variables so that I know your function works.