Leigh J. Little SUNY Brockport

Contents

1 Some Other MATLAB Basics

1.1 Comments

You can add comments to your script files. In MATLAB, the comment symbol is the percent (`%`) sign. Any text that occurs after a comment symbol is ignored. Unfortunately, there is no way to do block commenting.

1.2 Variables

We have used some variables in MATLAB, but it is instructive to provide some firm rules that MATLAB uses:

1) A variable must start with a letter.

2) A variable can contain letters, numbers and the underscore (`_`) character.

3) A variable must be 32 characters or less in length.

4) A variable must not have the same name as an existing MATLAB function, a function that you have written yourself or an existing variable name (unless you want to overwrite the existing variable value with a new one).

1.3 The diary Command

You can use the `diary` command to keep a record of all the commands you have entered. You can also use the history window in the lower left portion of the screen. The commands

```
>> diary on
>> diary off
```

turns the diary on and off respectively. By default, the diary will be stored in a file called `diary` in the current directory. You can give a specific file to use for the diary by entering

```
>> diary('filename')
```

1.4 Starting Over; The clear Command

When you want to completely remove one or more variables from the workspace use the clear command:

`clear` - clears all variables from the workspace

`clear A b X` - clears only the variables `A`, `b` and `X`.

1.5 Formatting the Output

MATLAB can display the result of calculations in several different precisions. The standard format is 5 digit open formatting. This is not always the best way to display the results of numerical calculations hence this can be changed to one of several alternate formats using the `format` command.

`format` - default 5 digit open formatting

`format short` - same as above

`format short e` - 5 digit exponential formatting

`format long` - 15 digit open formatting

`format long e` - 15 digit exponential formatting

See `help format` for more options.

2 Wring Your Own MATLAB Functions

In the last lecture, we looked at using MATLAB to compute the best-fit line through a set of data. MATLAB was able to compute the best-fit line, but there are important auxillary quantities associated with the best-fit line that are needed in order to assess the quality of the best-fit line. The functions to compute these quantities are not built into MATLAB (unless you shell out extra money for the Statistics add-on package). We did see that the Pearson correlation coefficient (r) and the R^2 value could easily be computed from their mathematical formulas using MATLAB's whole array operations.

One of the homework problems you had to do was to take the generic sequence of statements that we developed for computing r and R^2 and incorporate them into the existing `sc1.m` script for computing the best-fit line. However, what you observed is that you could not just paste in the few statements that perform the calculation. This was because the variable names used in the generic process (`x` and `y`) did not match the variable names that already existed in the script file (`xval` and `yval`).

Consider a more complex example. Suppose you developed a generic process for performing some computation, but instead of 6 or 7 lines of code, your process was 20, 40, or 200 lines of code. If you wanted to make use of your generic process by pasting it into an existing script you would likely encounter this same problem only on a much larger scale. You would have to take your generic process and make many changes to variable names in order to incorporate your generic process into an existing script file. This would be a terrible way to do things. You would almost certainly forget to change some of the variable names and there would be a high probability of introducing errors into the process. Moreover, you would need to do this everytime you needed to use your generic process in a different script.

It would be much better if there was a way you could write your own MATLAB function for computing the Pearson correlation coefficient. It would be nice if there was a way you could just do

```
>> r = pearson(xval,yval)
```

MATLAB would send your arrays of coordinates (`xval` and `yval`) to this function and return the value of r without you having to manually adjust the sequence of statements everytime you needed this perform this calculation.

Fortunately, we have this capability. MATLAB (and every other programming language) allows you to write your own functions. Before diving into how to do this, we should examine a function we have not looked at in detail before

2.1 The `max` Function

As you saw in the previous homework assignment, MATLAB has a `max` function that will return the maximum value of a vector.

```
>> x = [-3 4 -2 6 -1 6]
x =
   -3     4    -2     6    -1     6
>> max(x)
ans =
     6
```

When used in this way, the `max` function returns the largest element of the vector `x`. However, there is another way to call this function. Consider the following:

```
>> [xmax,xloc] = max(x)
xmax =
     6
xloc =
     4
```

In this case, the `max` function returned 2 outputs. The first is the maximum value of `x`. What is the second output (`xloc`)? This second output is the *first* location in the vector where the maximum value occurs.

```
>> x(xloc)
xmax =
     6
```

As we will see later in the semester, when working with maximum values of vectors and matrices we are often more interested in where the maximum value occurs than what the maximum value is. The `max` function is written in this way so that we can obtain both the maximum value and the location where it occurs.

This example illustrates an important feature of MATLAB functions. We have seen that MATLAB functions can have a variable number of inputs (for example, the `norm` function). MATLAB functions can also return a *variable number of outputs* (for example, the `max` and `min` functions).

What if you only wanted the location of the maximum value? The ordering of outputs is important. We need a way to tell MATLAB that we want to ignore the maximum value itself and return only the location of the maximum. You can do this using the (`~`) operator. For example,

```
>> [~,xloc] = max(x)
xloc =
     4
```

Essentially, you use the (`~`) operator as a placeholder for output values you want to ignore.

For completeness, we should examine what happens with you send a matrix as an input the to `max` function

```
>> A = [4 3 2; 6 -1 3; 2 0 9]
A =
     4     3     2
     6    -1     3
     2     0     9
>> [amax,aloc] = max(A)
amax =
     6     3     9
aloc =
     2     1     3
```

In this case, the `max` function returns two row vectors. The first vector (`amax`) is the maximum value of each column of `A`. The second vector (`aloc`) is the *row* where the maximum value in each column occurs. Note that I chose the variable names to store the output from the `max` function. Any variable name could be used here.

2.2 Writing Your Own MATLAB Functions

Functions that you create yourself comprise a critical element to programming (in all languages, not just MATLAB). Writing a complex program such as MS Word or an operating system like Linux without this capability would technically be feasible, but it would also be a nightmare.

The following basic rules apply when you write your own function in MATLAB.

1) You must choose a name for your function. This cannot be the same as an existing MATLAB function, some other function that you have written yourself or a variable name you are likely to use in your programs. For example, choosing to name your function `n` would be a terrible idea because `n` is commonly used as a variable.

   For this dicussion, we will assume the name of the function is `fname`

2) Your function must be saved in a file called `fname.m`.

3) Your function must be in your working directory (there are ways around this, but we will use this rule for now).

A MATLAB function has the generic template

```
function [list of ouptut arguments] = fname(list of input arguments)
%
%  Comments (not required, but recommended)
%
 |
 |
 | MATLAB statements to be executed by the funtion
 |
```

Some important items to note:

1) The function starts with the `function` keyword.

2) The list of output arguments is enclosed in square brackets, not parentheses.

3) You can include a comment section at the top (and anywhere else in your function).

4) You can have any number of input arguments, of any type (scalar, vector, matrix, *etc.*). You can also have a function with no inputs.

5) Item 4 also applies to output arguments.

6) The ordering of the arguments is important. We will see more on this shortly.

7) This example function would need to be saved in a file called `fname.m`

One of the easiest ways to see how a function works is to write a function to do something that you already know how to do.

2.3 The Quadratic Formula

For the quadratic equation

$$ax^2 + bx + c = 0,$$

you know that there are two solutions (counting multiplicities) and that the formula for these roots is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

We can easily write a MATLAB function that will compute the roots of a quadratic equation given the values of a, b and c.

Enter the following statements into a file called `solvquad.m`:

```
function [x] = solvquad(a,b,c)
%  [x] = solvquad(a,b,c)
%
%  This function solves the quadratic equation
%
%      a * x^2 + b * x + c = 0
%
%  for the two roots and returns the result in a
%  vector of length 2.

x(1) = (-b + sqrt(b^2-4*a*c)) / (2*a);
x(2) = (-b - sqrt(b^2-4*a*c)) / (2*a);
```

This function computes the two roots and stores them in a vector of length 2. `x(1)` corresponds to the positive root and `x(2)` corresponds to the negative root.

Notice that we have named the function `solvequad`. This does not conflict with an existing MATLAB function name, so it is safe to use this name for the function.

Some other items to note:

1) The user will supply values of a, b and c when they use (or *call*) the function.

2) When the function is called by the user, the first argument is assumed to be a, the second is assumed to be b and the third is assumed to be c (see example below).

3) There is a comment section that the top that describes what the function does.

4) We can immediately start storing the computed roots into the vector `x`. We don't need to tell the function that `x` is a vector.

We can test this function to see if it does what it should by having it compute the roots of a quadratic equation whose roots we know. Consider the equation

$$x^2 + x - 6 = 0.$$

For this equation, we know the two roots are $x = 2$ and $x = -3$. Also, the coefficients are $a = 1, b = 1$ and $c = -6$. We can compute the roots of this equation by doing

```
>> soln = solvquad(1,1,-6)
soln =
     2    -3
```

Notice that the function correctly returns 2 and -3 as the roots. Similarly for the equation

$$x^2 + 9 = 0$$

we know the two roots are $x = 3i$ and $x = -3i$ and that the coefficients are $a = 1, b = 0$ and $c = 9$. We can compute the roots of this equation by doing

```
>> soln = solvquad(1,0,9)
soln =
   0.0000 + 3.0000i   0.0000 - 3.0000i
```

One critical item to note is that even though we called the output `x` in the function body, we can call the output anything we want when we actually use it. In this case, in the main workspace, we called the output of the function `soln`. What this means is that the variables we use when we write function (*i.e.,* in the `solvequad.m` file) are *dummy variables.*

To further illustrate this, consider the following example:

```
>> hat = 1
>> cat = 0
>> football = 9
>> ratsnest = solvquad(hat,cat,football)
ratsnest =
   0.0000 + 3.0000i   0.0000 - 3.0000i
```

This example illustrates two important concepts:

1) Arguments to functions can be hard-coded numbers (like in the first two examples) or they can be the names of other variables.

2) We are using the variable names `hat`, `cat` and `football` in the workspace. These are different names than we use in the `solvequad.m` file. When the function is invoked, the function will use `hat` as the a variable, `cat` as the b variable and `football` as the c variable.

This is what is meant when we say that the variables in the function body are *dummy variables*. They are placeholder names for the real names that we are using in the main workspace.

This is what makes functions so useful. We can write a process for solving a quadratic equation regardless of the variable names we send in from the main workspace.

For example, the following version of the `solvequad.m` file contents would perform exactly the same:

```
function [cow] = solvquad(orange,apple,banana)
%  [cow] = solvquad(orange,apple,banana)
%
%  This function solves the quadratic equation
%
%      orange * x^2 +  apple * x + banana = 0
%
%  for the two roots and returns the result in a
%  vector of length 2.

cow(1) = (-apple + sqrt(apple^2-4*orange*banana)) / (2*orange);
cow(2) = (-apple - sqrt(apple^2-4*orange*banana)) / (2*orange);
```

Finally, if you ask for help on the `solvquad` function, MATLAB will echo the comment section at the top

```
>> help solvquad
   [x] = solvquad(a,b,c)

   This function solves the quadratic equation

       a * x^2 + b * x + c = 0

   for the two roots and returns the result in a
   vector of length 2.
```

2.4 A Second Solvequad Example

Enter the following into a file called `solvequad2.m`:

```
function [x1,x2] = solvquad2(a,b,c)
%  [x1,x2] = solvquad2(a,b,c)
%
%  This function solves the quadratic equation
%
%      a * x^2 + b * x + c = 0
%
%  for the two roots and returns the result in
%  the variables x1 and x2

x1 = (-b + sqrt(b^2-4*a*c)) / (2*a);
x2 = (-b - sqrt(b^2-4*a*c)) / (2*a);
```

Notice that this function performs the same task as the first one. The difference is that instead of returning the two roots in a vector of length 2, the roots are returned individually as separate scalars. For example,

```
>> [soln1,soln2] = solvquad2(1,1,-6)
soln1 =
     2
soln2 =
    -3
```

Both versions of the `solvquad` function are valid and can be used to solve quadratic equations. Generally, the first one would be preferred from a data organization point of view. This illustrates the flexibility you

have when you write functions. As long as it produces the necessary output and works properly, you can set up the function calling sequence however you wish.