Matlab Tutorial for
Computational Methods CE 30125

prepared by
Aaron S. Donahue
adonahu1@nd.edu

Disclaimer: Programming in Matlab is a very long and deep subject. The following is a synopsis of statements that will help with what is done in this class, but this is by no means whatsoever a complete synopsis of what Matlab is capable of. Similarly this document does not go into detail about how to optimize code or how to take advantage of built in Matlab functions to create faster more effecient code. You are encouraged to take a Matlab course, or look through one of the many Matlab references for information on how to construct high level code for complex problems. The internet is also an excellent source on how to optimize code or as a reference for built in Matlab functions. The company Mathworks, that created Matlab, keeps track of the community built functions, these can be found at the website

http://www.mathworks.com/matlabcentral/fileexchange/

Often if you are trying to do something basic in your code, like find the Lagrange basis functions for a certain set of points, someone in the Matlab community has written a code to do this and has uploaded here. So if you becoming frustrated with a section of your code you may be able to get motivation here. Just like in all academics, if you use someone elses code to do your own work you must cite them and their code, otherwise this is plagerism just like stealing passages from a book or article without proper citation.

4 Logic Statements

It is possible to perform a series of tasks in Matlab quickly and methodically using a variety of logic statements. The most commonly used logic statements are "if", "for" and "while".

4.1 The "if" statement

The syntax for the "if" command is as follows

```
if expression1
    statements1
elseif expression2
    statements2
else
    statements3
end
```

if "expression1" is true then the group of "statements1" will be executed, otherwise Matlab will check if "expression2" is true, if so "statements2" will be executed. If neither of these two expressions is true then "statements3" will be executed. For example

```
if a > 0
    display('a is positive')
elseif a < 0
```

```
    display('a is negative')
else
    display('a equals zero')
end
```

Note: It is not necessary to use either of the "else" statements in an "if" statement. The above is the full syntax, an "if" statement only requires the "if", "expression1", "statements1" and "end" components. Similarly it is possible to include more than one "elseif" command if you have a longer string of "if"'s you wish to use.

4.2 The "for" statement

The syntax for the "for" statement is as follows

```
for index = values
   program statements
          :
end
```

for a given named index, say "n", the "for" statement will march the index through the "values", each time running through "program statements". For example

```
for n = 1:10
    display(n)
    a(n) = n^2;
end
```

This will march "n" through the numbers 1 to 10. Each step it will display the value of "n" to the command window, it will also create a vector that has 10 elements, each element being the square of its indice. Producing the vector

```
a =

     1     4     9    16    25    36    49    64    81   100
```

Note: The name of "index" can be anything elegible to be used as a variable name. When writing a program it is important to not use a variable name you are using elsewhere, because the "'for" statement will redefine the "index" variable as it goes through the loop.

Note: A "for" statement is not confined to only running through values one by one, for example

```
for n = 1:0.5:10
    .
    .
    .
end
```

or

```
for n = [-1 pi 6 9 19]
    .
    .
    .
end
```

are both allowable uses of "for".

4.3 The "while"' statement

The syntax for the "while" statement is as follows

```
while expression
   statements
end
```

While the "expression" is true the "statements" will be executed. For example

```
n = 1;
while n < 100
    n = n*(n+1);
    display(n)
end
```

The "while" loop in this case will only stop when "n" becomes greater than 100. In this case the output produced is.

```
n =

     2


n =

     6


n =

    42


n =

        1806
```

Note: It is very important to make sure that when using a "while" loop that it is not possible to get trapped in an endless loop. For instance if we replaced the "n=1" statement with "n=0" in the above example the "while" loop would never terminate and we would have written a never-ending program. There are some tricks that can be used to protect against this that we will discuss later.

4.4 Nested Loops

In a basic sense when writing codes and logic statements consider that Matlab will be reading your code from top to bottom executing commands it encounters. In this way it is possible to "nest" loops within loops. For instance one could write a "for" loop that executes an "if" statement or a "while" statement, or both within the loop. For example, it is possible to exectue all three of the above examples in one complete logic statement, as follows

```
for n = 1:10
    if n > 5
        m = n;
        while m < 100
            m = m*(m-1);
        end
    elseif n < 5
        m = n;
        while m < 100
            m = m*(m+1);
        end
    else
        m = 100;
    end
    a(n) = m;
end
```

which produces the output

```
a =

  Columns 1 through 7

        1806        1806         156         420         100         870        1722

  Columns 8 through 10

        3080        5112        8010
```

In other words there is no limit to the number of nested loops one can create. A simple rule of thumb is to consider how you would logically work a problem out and then program that logic directly into Matlab. This does not always produce the most effecient code, but it often produces code that works. As you become more preficient with programming in Matlab you will begin to understand how to "optimize" your code for speed. But for now it is important to get code that works.

4.5 Auxillary Logic Statements

4.5.1 Break

The "break" command literally "breaks" out of a "for" or "while" loop. Usually a "break" command is included with a nested "if" statement and will allow the premature ending of the loop due to some circumstance. For instance in the "while" loop example above it was mentioned that if "n" were 0 then the loop would never end. One way to avoid this endless loop would be to include a "break", for example

```
while n < 100
    if n <= 0
       break
    end
    n = n*(n+1);
    display(n)
end
```

4.5.2 Continue

The "continue" command continues on to the next iteration in a "for" or "while" loop. This can be used to halt the execution of all the statements in a loop if they are unnessary. Just like in the "break" command, this is usually initiated using an "if" statement. For instance

```
for n = 1:10
    if n > 5
        continue
    end
    a(n) = n^2;
end
```

produces the output

```
a =

     1     4     9    16    25
```

the for loop skips over all "n" greater than 5.

4.6 Truth Statements

In all of these examples we have spoken of when something is "true", this refers to a truth expression. There are many of these in Matlab, but the most common are the same expressions we would use in writing down equations. When a statement is true it returns a 1, when it is false it returns a 0. The following are examples of these basic truth statements.

- "==" is used between two statements to determine if they are equal, for instance "4 == 3*2" returns a 0 and "4 == 2*2" returns a 1. *Note:* Notice the use of two "=" signs back to back. To use "=" in a truth statement it must be doubled up.

- "< " and ">" are used between two statements to determine if they are less than or greater than each other, for instance "pi>3" returns a 1, while "pi<3" returns a 0. *Note:* These truths are for strictly greater than or less than, for instance "3>3" will return a 0.

- "∼" is used as a "not" statement, for instance "4∼=3" reads as "is 4 not equal to 3". In this case a 1 would be returned. The "∼" component can be combined with any other truth statement to negate that statement.

- The "=" and "<" or ">" statements can be combined to be "less than or equal" and "greater than or equal" respectively, as follows. "4>=3" returns a 1, similarly "3>=3" returns a 1.

- "&&" is used to string together a series of 'and' logic statements. This will return a 1 only if all statements strung together are true, if any of them are false then a 0 is returned, for example."2==2 && 2>=0" reads as "is 2 equal to 2 and is 2 greater than or equal to 0?", of course this is true and this statement will return a 1. However if we change it to "2==1 && 2>=0" which read as "is 2 equal to 1 and is 2 greater than or equal to 0?", not true and thus a 0 is returned.

- "|| " is used to string together a series of 'or' statements. This will return a 1 if any of the statements are true, and a 0 if all of them are false. For example, "2==1 || 2>=0" reads as "is 2 equal to 1 or is 2 greater than or equal to 0?", which is true since 2 is greater than 0, thus a one is returned. However if we have "2==1 || 2<=0" reads as "is 2 equal to 1 or is 2 less than or equal to 0?", this is false for both statements and thus a 0 is returned.

4.7 Some other helpful commands

- "display(...)" Displays something directly to the command window. This is useful during the debugging phase of code development to see if your code is working properly. Aux syntax is "disp(...)".

- "pause(...)" Pauses the program for a set amount of time, if "..." is blank than the program will pause until a key is pressed on the keyboard.

- "a = input('...')" Will prompt the user to enter in some sort of input. This input will be assigned to the variable "a", the prompt in the command window to the user will be whatever is written in the "('...')", notice that as in all string inputs this must be bordered by two apostrophes.

Note: The logic statement 'if' will initiate as long as 1 is returned from the expression it is evauating. For instance if you where to type

```
if 2 == 2
    .
    .
    .
end
```

This statement would alway run since "2==2" is always true and always returns a 1.

5 The Editor Window

The editor window in Matlab can be accessed in a variety of ways. The most straightforward is to go "File" in the menu bar and select "New" from the drop-down menu, then select "Blank M-File". This will open up a new window called the editor window.

Note: It is possible to dock the editor window in the main Matlab window, to do this go to the "Desktop" tab in the menu bar of the editor and select "Dock $\cdots$".

The purpose of the editor is to write either "scripts" or "functions". A script is a series of Matlab commands that you want executed in order from top to bottom. This is convienent if you intend to run the same series of commands over and over again, or would like to edit certain commands and rerun a series. By producing a script you save yourself from the trouble of having to type everything in the command window everytime. When a script is run all the variables you name will appear in the Matlab workspace.

A function is similar to a script in that it is a series of commands that will be initiated by Matlab from top to bottom. The distinction in a function is that can recieve variables "passed into" it. It does not access the variables in the workspace. Similarly it does not write variables to the workspace unless instructed to do so. This is useful when constructing programs with many different modules. You can construct a function for each module and call upon these functions within the main body of the code. *Note:* all of the complicated commands you use in Matlab are actually functions. If you are at all curious type something like "edit polyfit" into the command window and see the syntax for the function "polyfit".

Regardless of if you are writing a script or a function the extension on the file will be a ".m", for example "myscript.m" or "myfunction.m". Now lets construct a script and a function to plot really nice figures.

5.1 Script

We want to write a script that will take the 'x' and 'y' vectors from the workspace and plot them, with the correct labels, as well as determine the maximum and minimum values of y. In the command window we would type something like

```
>> plot(x,y)
>> xlabel('x','fontsize',16)
>> ylabel('y','fontsize',16)
>> legend('y(x)')
>> title('My x-y plot','fontsize',16)
>> a = min(y);
>> b = max(y);
```

To put this into a script simply copy and paste all of these commands into a blank editor file and then save as something like "myplottingscript.m". This would look like

```
plot(x,y)
xlabel('x','fontsize',16)
ylabel('y','fontsize',16)
legend('y(x)')
title('My x-y plot','fontsize',16)
a = min(y);
b = max(y);
```

Now everytime you want to plot 'x' and 'y' and get the max and min of 'y' you can initiate this script from the command window by typing "myplottingscript" into the command prompt.

There is a shortcoming to this approach. This script will only plot the vectors 'x' and 'y'. If you had a vector 'f' you wanted to plot instead you would need to either change your script or change 'f' to 'y'. More versatility can be gained from using a function, as follows

5.2 Functions

The actual text in a function is very similar to the text of a script aside from the first line. Since a function does not access the workspace you need to specify which variables (if any) will be passed into the function. To change the above script example into a function we add the following line to the top of the text file

```
function myplottingfunction(x,y)
```

Now in order to use the function you must type "myplottingfunction(x,y)" into the command prompt. The result is the same, however now if you wish to plot the vector 'f' instead you can type "myplottingfunction(x,f)" into the command prompt. This is because regardless of the name Matlab will interpret the first entry into the function as 'x' and the second entry as 'y' when exectuing commands.

As mentioned before when a function is run it will not save the variables to the workspace. In order to save variables to the workspace for later use we change the first line of the function file to something like

```
function [a b] = myplottingfunction(x,y)
```

Now whatever 'a' and 'b' are in the function will be written as output. This also necesitates a change in how we call upon the function, instead of simply writing "myplottingfunction(x,y)" we may write something like "[mymax mymin] = myplottingfunction(x,y)" into the command prompt. Note that the output variable names when we call a function do not need to be the same name as the output names defined in the function. Our plotting function may look like this now

```
function [a b] = myplottingfunction(x,y)

plot(x,y)
xlabel('x','fontsize',16)
ylabel('y','fontsize',16)
legend('y(x)')
```

```
title('My x-y plot','fontsize',16)

a = min(y);
b = max(y);
```

5.3 Commenting

There is a subtle, and often lost, art to commenting your code. A comment in either a script or a function is a line of text that won't be executed by Matlab. In order to include a comment the comment must begin with the '%' symbol. If anything begins with this symbol it will be ignored by Matlab.

Why would we do this you ask? Two common reasons are as follows. The first is to add comments to the code for the user (or you) to read. These usually describe what is going on in the code at this point. When we write long code it is easy to get lost in all of the syntax, and if we step away from the code for awhile and return to it later it can be really really difficult to figure out what is going on at a particular line. So we include comments as sort of a "map" of the logic.

Another use would be to "comment out" a line that we don't want to execute but don't necessarily want to remove from the code, in case we need it later. This can be nice when debugging code. You can simplfy the actions taken in a code by commenting out certain parts so that you can focus on just one section. If we wanted to comment the above code it may look something like this.

```
function myplottingfunction(x,y)
%% Function to Plot the vectors x and y
plot(x,y)   % Plot the two vectors
xlabel('x','fontsize',16)   % Create the x-axis label
ylabel('y','fontsize',16)   % Create the y-axis label
legend('y(x)')  % Create the legend
title('My x-y plot','fontsize',16)  % Create the title

a = min(y)  % Find the minimum of y
b = max(y)  % Find the maximum of y
```

Notice that below the "function" command there is a brief description of what the function does. After each command we have included a comment on what that line of code actually does in words.