MATH 3670 - SCIENTIFIC COMPUTATION I
Fall 2015

Week 3: Programming in MATLAB (Chapter 6).
Functions in MATLAB (Chapter 7)

Content

Part I. Introduction to MATLAB programming
- Relational and Logical operators
- Conditional constructs
- Loop constructs
- output formatting
- Examples

Part II. Functions in MATLAB
- Anonymous functions. The `fplot` command
- Function file and its structure (arguments, return values, function body)
- Local vs. global variables
- Function files vs. scripts; when to use either
- Function functions (use of via function name or handle)
- subfunctions, nested functions

• **Homework # 3**: (Due Fri, Sept 18) (***** - mandatory for MATH 3670)
Chapter 6, pg 213 - 215 # 14, 26, **30***
Chapter 7, pg 252–258 # 17, **26***, 36

---

1. Basic Programming

So far, we have been doing MATLAB "programming" of sorts... the code snippets demonstrated in the class lectures as well as the assigned homework can be classified as "programming". Today, we formalize this notion a little bit.

First we want to define the notion of a "program". For now, let's just think of a "program" as a code snippet that is all inclusive. For our purposes, a program is a set of lines of code that can be used on any computer that can run MATLAB and produce the same results as the author of this program. For the most part, the homework that has been submitted so far qualifies as a program by this definition.

At this point, most students make use of MATLAB's editor and command windows to execute code. A typical scenario consists of entering commands (lines of code) in the

command window to see what happens, and then after a satisfactory action is observed, this/these commands is/are entered into the editor where the program is being created. Many of you have already implemented ideas such as clearing out variables via `clear` and `clc` to clear the command window. This is an example of making the program all inclusive - to ensure that any future MATLAB environment does not conflict with the environment necessary for the current program.

So far, our "programs" have, for the most part, been linear - each line of code in a script file is executed in order. Most real world solutions require a more complex execution order. Examples of non-linear execution order include:
• A loop, which may contain a few or many lines of code and may be executed over and over many times as required by the application.
• A boolean structure (such as an if - else structure) - if (some state) then perform (some task) else (perform another task). These tasks may be a few lines of code or a vast program in it's own right. The state from which this is decided upon might not be known until run time. For example, the state might depend on some input such as a user's input or an input due to a sensor on an aircraft's wing.
• A "call" to another snippet (i.e. subroutine or function) that you may write or has already been written to perform some particular task. This task may itself, be a very complicated program. Function writing and usage will be covered in a later lab session.

2. EXAMPLES to illustrate some of these concepts:

1) Compute Fibonacci Numbers
The first 2 Fibonacci numbers are 0, 1. Each subsequent Fibonacci is the sum of the previous 2 terms. The first 6 Fibonacci numbers are 0, 1, 1, 2 , 3, 5

Since Fibonacci numbers are recursively defined (each new fibonacci number is the sum of the 2 previous fibonacci numbers), an iterative computing strategy is a natural choice for computing arbitrary fibonacci numbers.
Let's say you were asked (by a teacher, boss, etc.) to write a program that produces Fibonocci numbers and display only those within a range that a user specifies. Your job is to write a program that satisfies these requirements. Therefore we need to formally specify these requirements which are derived from our boss's request. Specifically, we need to somehow obtain from the user a first and last Fibonocci number to be displayed. Also, we need to compute these numbers and print them out. Given the definition of such a sequence, we see that we must compute all Fibonocci numbers at least up to the higher of the user specified range. Now we know that we need to ask the user for the first and last Fibonocci number to be displayed and THEN compute all such numbers up to the last one specified. Additionally, only some of those that were computed (those within the specified range) can be printed out. The following will do this job:

```
f1 = 0;
f2 = 1;
first = input('Please enter the first Fibonocci to be displayed: ');
last = input('Please enter the last Fibonocci to be displayed: ');
```

```
for i = 1:last
     new = f1 + f2;
     f1 = f2;
     f2 = new;
     if ((i >= first) && (i <= last))
         fprintf ('The %3ith Fibonacci number is: %i\n', i, f1);
     end
end
```

The first 2 lines initialize the first 2 Fibonacci numbers. Please understand that the definition of Fibonocci requires that the first 2 must be defined (not computed).
The next 2 lines uses the **input()** function to prompt the user for the 1st and last numbers that are to be displayed.
The rest of this code piece consists of a **for loop** which causes the block of code within this **for loop** to execute a specific number of times. How many?? From 1 to the variable **last** which is set to the response that the user typed in.
Notice the line

```
  if ((i >= first) && (i <= last))
```

This **if** statement allows the interior **fprintf()** statement to execute ONLY if the loop iterator **i** is within the range specified by the variables **first** and **last** which were defined by the responses that the user specified which satisfies the second major (derived) requirement.

The **fprinf()** line outputs the contents within the literals (apostrophes) to the command window. Notice that there are two % characters within this literal string. When a % appears INSIDE a string literal in a **fprintf()** function, the few characters that follow represent how a variable's output is to be formatted as described on pages 103 - 110 in the text. Here, there is

```
%3i
```

and

```
%i
```

which tells **fprintf()** that there are 2 variables to be formatted; the first one will be the first variable in the format list which is **i** (the loop iterator number (and forced to use exactly 3 digits whether it needs them or not). The forced usage of 3 digits allows for the output to be lined up nicely. The 2nd is a free formatting integer meaning it will use only the required number of digits which will display the value of the variable **f1** - the ith Fibonacci number.

The line

```
  new = f1 + f2;
```

computes the next Fibonacci number.

Notice those last 2 lines in this loop. These are necessary so that during the next iteration of the **for loop**, the next Fibonacci number can be computed.
When this code is executed and 23, 27 are entered at the prompts, the following is output:

```
Please enter the first Fibonocci to be displayed: 23
Please enter the last Fibonocci to be displayed: 27
The  23th Fibonacci number is: 17711
The  24th Fibonacci number is: 28657
The  25th Fibonacci number is: 46368
The  26th Fibonacci number is: 75025
The  27th Fibonacci number is: 121393
```

More on the **if-then-else** structure:

```
if (temperature < 10)
   blockNum = 1
   status = checkHeater();
elseif (temperature >= 10) && (temperature < 30)
   blockNum = 2
elseif (temperature >= 30) && (temperature < 45)
   blockNum = 3
elseif (temperature >= 45) && (temperature < 60)
   blockNum = 4
elseif (temperature >= 60) && (temperature < 80)
   blockNum = 5
elseif (temperature >= 80) && (temperature < 95)
   blockNum = 6
else
   blockNum = 7
    status = checkAirconditioner();
end
```

(NOTE: This code only illustrates a longer if-else if block; it will not execute because the functions called on the 3rd line and the 2nd to last line are not defined)

Notice that there are several conditional blocks (most of the consisting of only one line of code). Most of them require 2 relational expressions to evaluate to true before it gets executed.
Each relational expression consists of 2 objects that are compared by a relational operator. The most common relational operators are:

```
<, > <=, >= ==, ~=
```

IMPORTANT:
A relational expression MUST EVALUATE TO TRUE (integer value 1) OR FALSE (integer value 0). This is why

```
>> if (x = y)
```

results in an error because x = y DOES NOT evaluate to either TRUE or FALSE BUT...

```
>> if (x == y)
```

does NOT RESULT in an error BECAUSE x == y DOES evaluate to either a TRUE or FALSE.
Additionally, the **elseif** statements that have 2 relational expressions also have a **logical operator** that logically relates these 2 relational expressions. The result is that the line(s) of code within a particular block will only execute if all relational expressions evaluate to true and ALSO if their aggregation as specified by the logical operators evaluate to true.
As explained on page 179 of the text, MATLAB's logical operators include:
&& for logical AND
|| for logical OR

```
~ (tilda) for logical NOT
```

IMPORTANT:
If you intend for the interior block of code to be executed ONLY IF exactly one of those relational expressions evaluate to true, then you must use the function **xor(a,b)** which stands for **exclusive or**.
Page 182 of the text lists some other MATLAB logical built-in functions.

Another common mistake is confusing = (an assignment operator) and == (a relational equivalence operator) NOTE: the line:

```
>> if (x == y)
```

This line means **if x equals y then...**
HOWEVER, replace the == with =:

```
>> if (x = y)
```

produces

```
??? if (x = y)
           |
Error: The expression to the left of the equals sign not a valid target ...
```

The reason is that though **(x = y)** is a valid line of code (the value of y is assigned to x), **if (x = y)** is NOT a valid line of code. It is important to remember that the logical equivalence operator == asks **if x equal to y, then...**, which contrasts with the assignment equality operator **x = y** which says **assign the value of y to x**

The above code piece that had several **if-then-else** blocks could, alternatively be coded as a **switch** statement as explained on page 187 - 190, 203.

2) Investment accounts (iterative computation)
This piece of code computes how long a retirement account will last given:
- a 5% per year return on the account
- a set amount the retiree will withdraw per year (which each year is increased by the 2% inflation rate.
This will be computed for 3 different people who have different amounts and withdrawal needs.
This example (modified from the example on page 201) illustrates how an array can be used as a loop iterator among other things:

```
rate = .05;
inflation = .02;
names = ['Mary  '; 'George'; 'Julia '];
accounts = [400000, 155000, 686000];
withdraws = [22000, 16000, 48000];
for idx = 1:numel(accounts)
    years = -1;
    W = 0;
    balance = accounts(idx);
    Wnext = withdraws(idx);
    balanceNext = balance*(1+rate);
    while balanceNext >= Wnext
        W = Wnext;
        balance = balanceNext - W;
        balanceNext = balance*(1+rate);
        Wnext = W * (1+inflation);
        years = years + 1;
        if (years >= 100000)
             disp('something''s wrong; 100000 years have been computed');
             break;
        end
    end
    fprintf('%s''s money will last for %i years\n', names(idx,:), years);
    fprintf('  if $%i.00 is withdrawn every year and\n', withdraws(idx));
    fprintf('  assuming a constant inflation rate of %4.3f\n', inflation);
    fprintf('  and a constant rate of return of %4.3f percent\n\n', rate);
end
```

The above code snippet produces the following output:

```
Mary's money will last for 26 years
  if $22000.00 is withdrawn every year and
  assuming a constant inflation rate of 0.020
```

```
  and a constant rate of return of 0.050 percent

George's money will last for 10 years
  if $16000.00 is withdrawn every year and
  assuming a constant inflation rate of 0.020
  and a constant rate of return of 0.050 percent

Julia 's money will last for 18 years
  if $48000.00 is withdrawn every year and
  assuming a constant inflation rate of 0.020
  and a constant rate of return of 0.050 percent
```

Here, there is a **while** loop within a **for** loop.

```
for idx = 1:numel(accounts)
```

iterates a number of times which is computed as the number of elements in the **accounts** array. Notice the MATLAB function **numel(accounts)** - this function returns the number of elements in the **accounts** array which, in this case is **3**.
CAUTION: this code snippet is not "robust" - if any of the other arrays had fewer elements, this code would error out.
Now, for each **accounts** element, several variables are computed as required for each retiree's computation.
Note the line:

```
while balanceNext >= Wnext
```

This is a **while** loop initiator. It causes all code until the corresponding **end** statement (i.e. the 2nd one) to execute AS LONG AS **balanceNext greater than or equal to Wnext** evaluates to **true** (i.e. if the next iteration of balance is at least as big as the next withdrawal amount).
Then, there are a few lines that do necessary computations.
Note the 4 lines:

```
if (years >= 100000)
      disp('something''s wrong; 100000 years have been computed');
      break;
end
```

This was included for illustrating a "just in case" issue (and not a very good one at that). Simply put, it was deemed that if the number of years has grown to 100,000 (which means this loop has executed 100,000 times), then probably something is wrong. An error message is displayed and then a **break** statement causes the loop to exit. The motivation is that if this loop has executed that many times, then there may be some sort of infinite loop problem, possibly due to some unknown logic error in the code. The **break** statement will force this loop to terminate.

Now for the **fprintf()** statements. **fprintf()** will output to the command window (or file) information with a large degree of flexibility compared to the **disp()** function. It can output text and one or more variables in many different format options (such as floating point, scientific notation, specification of how many digits are displayed, etc.). In general, the % within quotes will tell **fprintf** that a variable's value is to be formatted as specified by the format specifier. The number of %s existing within the string literal should be the same as the number of variable names listed after the literal close (i.e. the closing apostrophe) before the closing parentheses of the **fprintf()** function. Additionally, it usually is a good idea to have as the last 2 characters before the closing apostrophe to be

```
\n
```

which causes a **line feed** so that the cursor is on the next line in column one so the next output won't simply be appended to the current output's line.
The above usages are discussed:

```
>>     fprintf('%s''s money will last for %i years\n', names(idx,:), years);
```

Outputs:

```
Mary 's money will last for 26 years
```

Notice the double apostrophe - this allows for a single apostrophe to be included in the string output.
Also, note that the first variable to be formatted via the %s specifies that this variable is to be formatted as a string (because of the **s** that follows the %) and the first variable (of the 2) in the variable list is **names(idx,:)** - all characters associated with the **idx**-th (1st, 2nd or 3rd) character string. The 2nd formatter: %i depicts that the corresponding variable **years** will be displayed as an integer

```
>>     fprintf('  if $%i.00 is withdrawn every year and\n', withdraws(idx));
```

Outputs:

```
if $22000.00 is withdrawn every year and
```

```
>>     fprintf('  assuming a constant inflation rate of %4.3f\n', inflation);
```

Outputs:

```
assuming a constant inflation rate of 0.020
```

Here, there is one format specification corresponding to the variable **inflation**: %4.3f
The **f** stands for **floating point**. Here, 4 digits total are allocated for the display, 3 of which are to the right of the decimal point.

```
>>     fprintf('  and a constant rate of return of %4.3f percent\n\n', rate);
```

Outputs:

```
and a constant rate of return of 0.050 percent
```

Notice the

```
\n\n
```

(2 line feeds) - this will leave a blank line between this output and any future output.
For a more thorough discussion on this, see pages 103-110.

---

3. SUMMARY: Things to remember so far

- **Logical and Conditional Operators:** - Typically used in if-then-else structures to control if a block of code is executed
- **For loops:** For loops - A common code construct that allows a block of code to be executed repeatedly. - Loop iterations may be determined by explicit loop parameters or via iterating through an array.
- **Review of**

- **fprintf()**
- User prompting via **input()**

---

3. Anonymous Functions. The "fplot" command

In this second part of the lab we will learn one of the most effective tool in programming with MATLAB (actually any software language/package): functions. MATLAB allows one to carry out simple calculations with variables in the workspace using canned (already written) functions (such as `sin, exp,` etc). BUT also, MATLAB allows the user to write their own functions. There are two distinct ways to write user-defined functions: within the computer code (in a script file or simply at the command window) using so-called anonymous functions and in a separate .m file (called a function file).

Let's illustrate this via a task of plotting a simple function of one variable, like $f(x) = 1/(x^2+1)$, for $x \in [-3, 3]$. We have already seen two ways to accomplish this task:

```
x=-3:0.1:3;
f=1./(x.^2+1); % function evaluation is simply an array operation
plot(x,f)
```

or

```
syms x
f=1/(x^2+1) % function is defined symbolically
ezplot(f,-3,3)
```

Simple (one-line) mathematical function such as this can be represented in a MATLAB code through so-called anonymous functions (a new feature of MATLAB since 2004).
NOTE: even though we have not explicitly used the **ezplot** function, its usage is similar to the **fplot** function call which takes as its first parameter a variable depicting a function which in this case is a symbolically defined function as opposed to **fplot** which requires a function defined as a character string.

```
f = @(x) 1/(x^2+1)
```

Note the syntax of an anonymous function. It specifies the name of the independent variable x after the symbol '@', then it contains the mathematical expression involving x. Worth mentioning are: x need not be predefined (as is the case for symbolic functions).

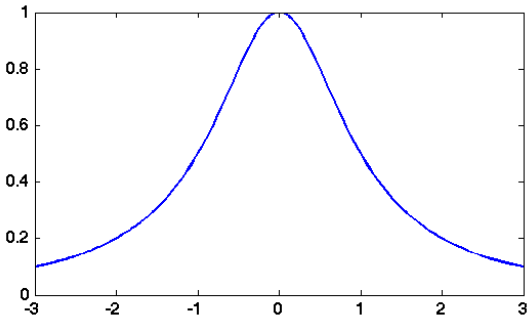
To evaluate an anonymous function at a point, say $x = 1$, one simply says

```
f(1)


ans
     = 0.5000
```

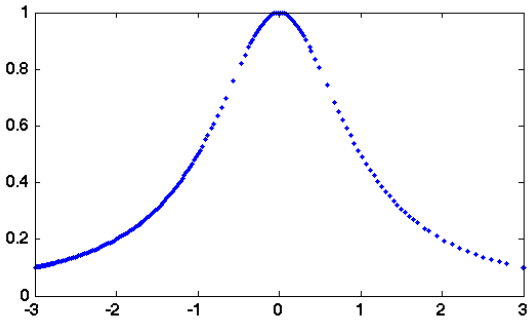
To plot an anonymous function, we use the command

```
fplot(f,[-3,3])
```



It is interesting to note how the `fplot` command actually plots the graph. It selects the values of x where the expression for f is to be evaluated in a non-uniform way. To see this, one can extract the actual values that were selected and then plot them as arrays separately

```
[x,y] = fplot(f,[-3,3]);
plot(x,y,'.')
```



NOTE: The [**x**,**y**] in the first line [**x**,**y**] = **fplot()** sets the variable **x** to the first output of **fplot** and the 2nd output is assigned to variable named **y**. As specified in the help for this function, this first variable is a vector representing the x-values that **fplot()** used and the 2nd variable is a vector consisting of the y-values associated with the above x-values.

You may be able to notice that the spacing between the x values plotted is not uniform. It is picked by the computer through an internal algorithm used by **fplot()**, out of your control. Therefore, not all plots should be done via `fplot`, especially if the graph has distinctive features at different locations.

4. Function Files

Besides the simple operations that can be represented through a one-line expression, the other way of writing user defined functions in MATLAB is a so called function file. Functions files in MATLAB are nothing else than text files saved with the .m extension (just like script files) in the current MATLAB directory on your computer, but with a specific format. Since both script files and function files have .m extension, then are both called m-files.

Compared to script files, function files are much better at compartmentalizing tasks. Each function file starts with a line such as

```
function [out1,out2] = myfun(in1,in2,in3)}
```

The variables `in1`, etc. are input arguments, and `out1` etc. are output arguments. You can have as many as you like of each type (including none) and call them whatever you want. The name **myfun** should match the name of the disk file (in this case: myfun.m). For this reason, you cannot use a name of a pre-defined function for your user-defined function.

Let's revisit the example before, this time using a function file, named myfun.m

myfun.m

```
function y = myfun(x)

        y=1/(x^2+1);
```

After this file is saved in your current MATLAB directory, call it from the command window

```
>> myfun(1)


ans
     = 0.5000
```

Notice that the semicolons are still used in the m-file. If they are not, the answer to a particular line will be printed out.

If you want the input to be an array instead of a simple number, then the function file must be written accordingly (in vectorized format)

myfun.m

```
function y = mynewfun(x)
% This function accepts inputs as arrays (vectors)
% The output is also an array

       y=1./(x.^2+1);
```

VERY IMPORTANT: Make sure you understand why the dots are in front of the division and exponentiation signs. This is so element by element division/multiplication is done instead of the standard matrix division/multiplication rules.
Now one can type at the command window (or from a different script file)

```
>> xplot = -2:0.1:2;
>> yplot = mynewfun(xplot);
>> plot(xplot,yplot)
```

The header is a descriptive portion of the function. It tells someone various information about the m-file. Some examples are a brief descriptive paragraph, a listing of inputs and outputs with associated units, a revision date, and authors name. MATLAB will always try to perform a mathematical operation on any text it sees. If it sees a descriptive paragraph, it will try to execute it and will end up finding an error. To avoid the error, comments can be added after a % sign.You may ask for this description in the command window by typing

```
>> help mynewfun


% This function accepts inputs as arrays (vectors)
% The output is also an array
```

2.1. Using a Function File

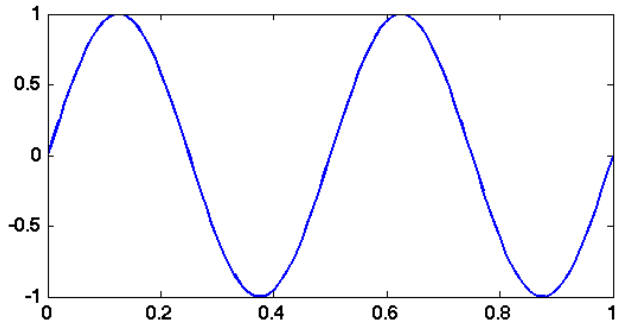
The usual mathematical functions are defined, like `sin`, `cos`, `exp`, and `log` (log means natural logarithm; log10 and log2 mean base ten and base two logarithms respectively). The argument of most MATLAB functions can be vectors, and the usual result is applying the function to element. Thus,

```
sin([pi/4 pi/2 pi])

ans =
    0.7071    1.0000         0
```

The following defines a 2 Hz sinewave over the time interval [0, 1];

```
>> t = 0:.01:1;
>> s = sin(2*pi*2*t);
>> plot(t,s)
```



As an example of how to use a function defined in a function file, we write a function that takes two numbers as an input and then outputs the sum, difference, product, and result of division. The m-file would look like:

algebra.m

```
function [sum,diff,prod,div]=algebra(a,b)
% This program takes in two numbers and performs addition,
% subtraction, multiplication and division

sum=a+b;
diff=a-b;
prod=a*b;
div=a/b;
```

The first line is the function definition line. The next 2 lines are comments describing what the function does and what the inputs & outputs are. The final four lines are the algebraic expressions that are exactly the same as what would have been typed in the workspace. Now to run the program, several examples will be given:

```
>> a=3;
>> b=2;
>>[sum,diff,prod,div]=algebra(a,b);
>>sum

    sum = 5
```

One can see that each of the output has been stored with the desired variable name.

```
>>diff

    diff = 1

>>prod

    prod = 6

>>div

    div = 1.5
```

One variation involves changing the names of the inputs and outputs:

```
>>c=3;
>>d=2;
>>[s,f,p,v]=algebra(c,d);
```

Notice that the names of the inputs do not match the names used in the m-file and the names of the outputs do not match those in the m-file. They do not need to match up. MATLAB do the change from c to a, d to b, etc. automatically. Now, if you use this m- file elsewhere you do not need to be concerned about what you name the variables you input and output, you just need to have the proper order.

A Matlab function accepts zero or more arguments (i.e. parameters) and returns zero or more outputs; each output can be any data type (i.e. scalar, string, matrix, etc.). A description of what a given function does results from typing help function-name. For example, type

```
>>help logspace

LOGSPACE Logarithmically spaced vector.
    LOGSPACE(d1, d2) generates a row vector of 50 logarithmically
    equally spaced points between decades $10^d1$ and $10^d2$.  If $d2$
    is $pi$, then the points are between $10^d1$ and $pi$.

    LOGSPACE(d1, d2, N) generates N points.

    See also LINSPACE.
```

Here is a function that implements (badly, it turns out) the quadratic formula.

quadform.m

```
function [x1,x2] = quadform(a,b,c)

d = sqrt(b2 - 4*a*c);
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
```

The reason why this is a bad implementation is that there are no sanity checks such as whether the user attempted to find the solution of inputs that are strings or other nonsensical inputs. Often, professionally written code will have such robustness checks (which may include work arounds, error handling/reporting, etc.) being the majority of the code.

From MATLAB you could call

```
>> [r1,r2] = quadform(1,-2,1)
```

2.2 Functions or scripts? That is the question!

Scripts are always interpreted and executed one line at a time. No matter how many times you execute the same script, MATLAB must spend time parsing your syntax. For example, if you had a script file that performed a task consisting of 10 lines of code and then some other script (or function) called this script file 1000 times, then for each of these 1000 "calls" to the script file, the same laborious syntax checking and many other internal activities would performed. By contrast, functions are effectively compiled into memory when called for the first time (or modified). Subsequent invocations skip the interpretation step. So if that same 10 lines in that script file was instead a function, then these laborious activities would be done once and for the remaining 999 executions, only the highly efficient "byte code" of the compiled 10 lines of code would be executed.

Most of your programming should be done in functions, which also require you to carefully state your input and output assumptions. Use scripts for drivers (main execution files) or when the task does not need to be compartmentalized. As a rule of thumb, call scripts only from the command line, and do not call other scripts from within a script.

In general, a function should perform one main task. If you find yourself adding more and more functionality to a function - a case of "function bloat", it may be time to consider splitting up this function into smaller ones.

2.3. Local and Global variables

Each function has its own workspace (scope) so the variables defined within a function execution are called local variables. The scope of a function cannot be accessed outside of the function. This means that if you somehow stopped execution within the function (such as via a break point) then the workspace will show ONLY those variables local to that function. Any variable defined by anything that called this function would not be visible; they exist but are not "visible" from within this function. As soon as execution is transfered from this called function back to the calling function, then the workspace will include only those variables that exists within this calling function (and those variables defined in the called function are now forever lost except for whatever this function passed back to the calling function). If variables are to be shared among several function files, then they need to be declared as global variables in EACH of the function file these variables are to be used. The syntax is

```
global variable_name
```

For more details, see the textbook , page 226–227. In general, usage of global variables is discouraged in professional programming environments in favor of passing information back and forth via function parameters and outputs. True, global variables are an easy way for functions to share information but experience has taught the professional community that in the long run, this creates more problems than the advantages are worth such as making it much harder to chase down obscure logic problems.

2.4. Subfunctions and Nested Functions

Often is the case that one function file calls another function file and so on, so it may be the case that you will need to work with multiple files in your directory. This may sound like a disadvantage, but it is not always the case, depending on the task at hand. In some cases several functions can be written inside the same m-file. As a general guide, one would put all functions in one .m file if it is likely that these functions will remain unique to the particular task for which this .m file was written for. If, however, if one foresees that a particular function that is being written for a particular problem may have utility for other unrelated problems, then one would put this function in its own .m file to be called by any, possibly still to be written program.

An example of nested functions is given below, where the task is to compute the vertex of a parabola given by the quadratic function

$$f(x) = ax^2 + bx + c$$

```
function [X,Y]=vertex(a,b,c)

X=-b/(2*a);
Y=fun(X, a, b, c);
```

```
    function Y1=fun(X1,a,b,c)
        Y1=a*X1^2+b*X1+c;
    end

end
```

To compute the vertex, use the command

```
[X,Y]=vertex(1,-2,5);
```

2.5. Function Functions

See textbook 234–240.

If you haven't done so, please download (to your personal z-drive location if on a UCCS computer) the zip file as explained in the Lab 6: link in the labs.html file. This zip file contains many professionally produced programs, some of them fun to execute. One can get a lot of tips from perusing these .m files to see how more complex programs are structured including the above described concepts regarding function creation and usage. This zip file was obtained from the very useful site: https://www.mathworks.com/moler/ncmfilelist.html

---

SUMMARY: Things to remember from this lab

When you save your m-file, you should save it with the same name that you gave it in the function call. If you do not you may run into problems. When you use the function in the command window, MATLAB will look for the actual .m file that is saved not for the name of the file in the function call. For example, if you had saved the example function as funky.m, MATLAB would require you to call funky in the work space and not algebra.

Another important note is that MATLAB usually starts in a default folder. Depending on the version of MATLAB, there may or may not be an obvious visual display. The implication is that when you call a function in the workspace, the m-file must be in the current folder. A few helpful commands are cd, cd pathway, and ls. The cd command will display the current directory that you are in. The cd pathway command will change you to whatever folder you specify in pathway. The `ls` command will list all of the files in the folder, whether they are MATLAB relevant or not. If you use the `ls` command and your m-file does not show up, you will need to move it into that folder or change the folder you are working out of.

---

4. HOMEWORK FORMAT (review from last week):

HOMEWORK NOTE: The assigned problem chapter 6, num 26 is poorly worded. The problem asks for you to prompt the user for weight and height inputs twice and then calculate BMIs for each set. But this will not be possible if you Publish because the

publisher will execute the program behind the scene which will cause it to hang (or stall) at the point where the prompt occurs.

Here is how I'd like you to proceed: First, get your program working with prompting the user for one weight and height input (via the **input()** function call). After you get this working, comment this line out and then create an array of the 2 heights and array of the 2 weights that are asked for. Have your program loop through these arrays to calculate the BMIs.

Organize this home work assignments as follows:
Put all problems into one .m file similar to below by making the first line identify the class, week number and YOUR NAME. Then, make a new section for each new problem (demarcated by

%%

```
%% Your Name, Math 265 WEEK 3 HOMEWORK #5 problems

%% Chapt. 6 #14
< all code for chapt. 6 #14>

%% Chapt. 6 #26
< all code for chapt. 6 #26>

%% Chapt. 6 #30 (for MATH 3670 only!)
< all code for chapt. 6 #30>
```

AGAIN: Before submitting your work, execute the .m file by pressing the little green arrow at the top of your MATLAB session. The output of this execution is what I use to determine if the assignment was done correctly. If you get any error message (red text in your command window), this means that your .m file did not execute completely. It executed up to the point where an error message was generated and then aborted. You must fix the error then re-run. Continue this process until a you get a clean run. After a clean run THEN...

Publish your work by clicking on file then publish. This will cause your .m file to execute again and create an HTML file consisting of all your output. It is THIS OUTPUT that you are to submit. Any output that has errors (such as the above mentioned red text) will not be given full credit.

Note that the homework problems are simple enough that one would normally write just one function or script to perform the requested tasks. But the point of these exercises is to learn the mechanics of writing one function (such as only performing a plot based on its input parameters) and calling it multiple times from a calling function to perform the required tasks.