MATH 3670 - Intro to Scientific Computation - Fall 2017

Lab 6. Functions in MATLAB (Chapter 7)

Content

- 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 Part II**: Chapter 7, pg 252–258 # 17, 26*, 36, (Due Fri, Sep 15)

In this 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).

1. Anonymous Functions. The `fplot` command

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).

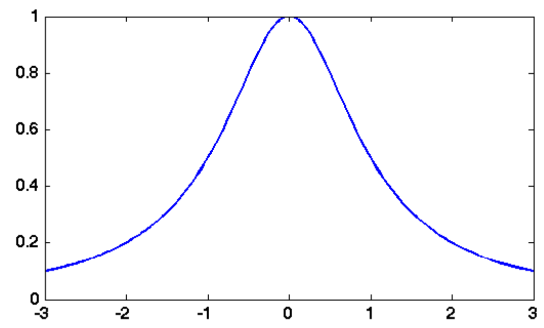
To evaluate an anonymous function at a point, say $x = 1$, one simply says

```
f(1)
```

```
ans
     = 0.5000
```

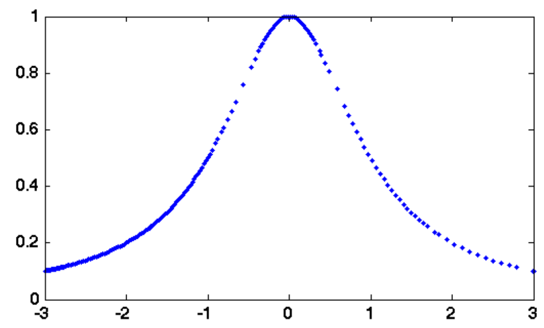
To plot an anonymous function, we use the command

```
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.

2. 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

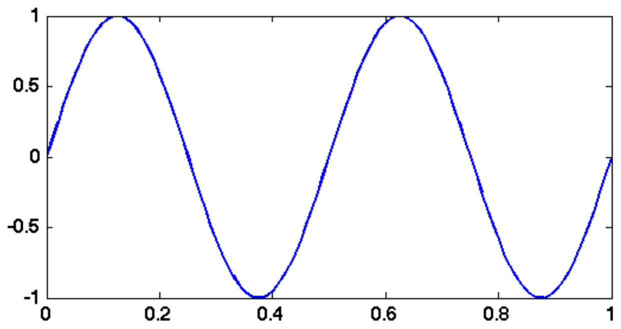
The usual mathematical functions are defined, like `sin`, `cos`, `exp`, and `log` (log means natural logarithm; log10 and log2 mean base ten and base two logarithms respectively). The argument of most MATLAB functions can be vectors, and the usual result is applying the function to element. Thus,

```
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.

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.