EENG 212-M01 Prof. L. Amara
EENG 221-M01
Fall 2014

Tutorial 5

MATLAB TUTORIAL
FUNCTIONS FILES

1-Difference between a script file and a function file

1-1 Script files:

So far the tutorials of the past weeks have highlighted the creation and use of script files, basically user-created files with a sequence of Matlab commands that are saved with an extension .m and referred to as m-files. These file are run in the Matlab environment by typing their name without the extension.m next to the Matlab prompt > (EDU>> if using the student version)

In case, the value of a variable has to be entered from the keyboard, we use the Matlab command **input** :

Consider this script file: **div.m**

```
>>a=input ('input the numerator value');      % enter value of a
>>b=input ('input the denominator value');    % enter value of b
>>v=mod (a,b)                                % mod function is the modulus after division.
```

An example of a script file that illustrates how to solve a set of linear equations, very common in circuit analysis:

To solve the following system:

$$5x_1 + 2x_2 + 5x_3 = 2$$
$$-1x_1 + 4x_2 + 3x_3 = 4$$
$$10x_1 + 9x_2 + 3x_3 = 5$$

First put it in a matrix format Ax=b

$$\begin{bmatrix} 5 & 2 & 5 \\ -1 & 4 & 3 \\ 10 & 9 & 3 \end{bmatrix} \begin{bmatrix} X1 \\ X2 \\ X3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 5 \end{bmatrix}$$

Call the script file is: **systemA.m**

```
>>A= [ 5   2   5;   -1   4   3;   10   9   3 ];   % create matrix A
>>b = [ 2;  4;  5 ];                               %create vector b
>>det_A= det(A) ;                                  %  find the determinant
>>x=A\b                                            % find x
```

the last line could have been replaced by

```
>>x=inv(A)* b
```

another example highlights the use of the **disp** (display) command

```
% using disp command
disp('Hello my name is EENG212')
disp(' ')                    % display a blank line
time= fix (clock)            % get time as integers
hours=int2str(time(4))       %get the hour
min=int2str(time(5))         %get the minutes
```

1-2 Function Files:

A function file has a function definition line, as a first command, that defines the input and output explicitly. A function file is also saved as an m-file and can be called at any time from another program.
The format of a function is:

function [xout, yout] = funcname (xin, yin)
body of the function……
……

[xout, yout] is the output list while (xin, yin) is the input list. Now the output list can be omitted entirely if the computed quantities are not of any interest. This might be the case when the function displays result graphically.

Let's rewrite the **div.m** script file as a function file

```
>>function r = modfn(a,b);
>>% if a and b are integers, then r is the integer remainder of a after division by d.
>>% if a and b are integer matrices, then r is the matrix of remainders
>>r=a-b.*floor(a./b) ;    %floor(A)  function rounds A to the nearest integers less than or equal
 >>                       % to A.
```

Next save the above file as **modfn.m**. It can be invoked and executed at any time by entering **modfn(a,b)** where inputs **a** and **b** should be specified. The output **r** is returned as a result.

Let's define another function that accept an input vector x and returns its mean and standard deviation. The file should be saved with the name **stat.m**

```
>>function [mean, stdev] =stat (x);
>>n=length(x);
>>mean=sum(x)/n;
>>stdev = sqrt(sum ((x-mean).^2/n));
```

The following example defines a function to compute the sum of a geometric series. The inputs to the function are two integers rand n and the output is the sum of series

```
>>function  s= gseriessum(r,n)
>>%the series is s=1+r+r^2 + r^3 + …+r^n
>>vector =0 : n;     %vector from 0 to n
>>series=r.^vector;    %create a vector of terms in series
>>s=sum(series) ;       % sum of all elements
```

2-More on File Functions

2.1 Subfunctions

Subfunctions are functions within a function. In the above file the mean could have been defined as a subfunction.

```
>>function [mean, stdev] =stat (x);
>>n=length(x);
>>mean=avg(x,n);
>>stdev = sqrt(sum ((x-avg(x,n)).^2/n));
>>function mean = avg(x,n)
>>mean = sum(x)/n;
```

Here the subfunction **avg** is imbedded in the **stat** function body and is not visible outside the file.
As another example, let's write a function file that outputs a conversion table for Celsius and Fahrenheit temperatures. The input of the functions should be two numbers: Ti and Tf, specifying the lower and upper range of the table in Celsius. The output should be a two column Matrix Celsius -Fahrenheit. Use increments of ones.

The file is saved as **ctof.m**

```
>>function convtable =ctof(tinitial, tfinal);
>>%table to convert temperature from C to F
>>C=[ tinitial : tfinal]';    % column vector C
>>F=(9/5)*C + 32 ;          % convert to F
>>Convtable = [ C   F] ;
```

Let's also write a function factorial to compute the factorial n! for any integer n. Of course we could have used the built-in function n!=prod(1: n).

```
>>function factn = factorial(n)
>>factn=1;                    % initialize  and   0!=1
>>for k= n: -1: 1             % go from n to 1
>>       factn=factn*k;        % multiply n by n-1, n-2 etc
>>end
```

Another example with a crowded output lits:

```
>>function [t, sinfct, cosfct, expfct]= fnExample2(f1, f2, damp)
>>%
>>% Function fnExample2
>>%
>>% call: [t, sinfct, cosfct, expfct] = fnExample2(f1, f2, damp)
>>% or fnExample2(f1, f2, damp)
>>%
>>% First example of a MATLAB function
>>%
>>% Normally both a description of the input and output
>>% parameters and the important characteristics of the
>>% function are given here
>>t=(0:0.01:2);
>>sinfct=sin(2*pi*f1*t);
>>cosfct=2*cos(2*pi*f2*t);
>>expfct=exp(-damp*t);
>>plot(t,[sinfct;cosfct; expfct])
>>xlabel ('time / s')
>>ylabel('Amplitude')
>>title('three gorgeous signals')
```

2-2 Evaluation a function with <u>feval</u>

The function **feval** evaluates a function whose name is specified as a string at the given list of input variables. For example **[x , y]= feval ('Hfunction' , r, t);** evaluates the function Hfunction on the input variables r and t and returns the output in **x** and **y**. It is equivalent to typing **[x,y] =Hfunction (r, t);** this method is mostly used when you have more than one function that you want to evaluate with the same input list.

3-Working with data Files

Serious programming involves creating, importing and exportind data. Now, data is saved in files in one of two formats: text or binary. In text format, data values are ASCIIcodes, and can be viewed in any text editor. In binary format, data values are not ASCII codes and cannot be viewed in a text editor. Binary format is more efficient in terms of storage space required since numbers are stored in a continuous stream. This format is to be read by only a computer.
When working with data files, the commands **load** and **save** are often used.

3.1 Binary files:
MATLAB can read and write binary files using the `fopen`, `fread/fwrite`, and `fclose` commands.
The following are the many formats used for binary files:

'uchar' unsigned character, 8 bits
'schar' signed character, 8 bits
'int8' integer, 8 bits
'int16' integer, 16 bits
'int32' integer, 32 bits
'uint8' unsigned integer, 8 bits
'uint16' unsigned integer, 16 bits
'single' floating point, 32 bits
'double' floating point, 64 bits

Picking the right data format with which to read a particular binary file requires either knowledge of how the file was written or plain old trial and error!

3.1.1 Saving/Loading variables:
The command **save** writes the contents of one or more variables to a specified file in binary format and the command load reads (inputs) variable data values from a specified file:
For example :

```
>> save('mytest.mat', 'A1', 'B1')
```

Stores the contents of the variables A1 and B1 to the mytest.mat file; and

```
>> load('mytest.mat', 'B1')
```

creates a variable B1 in the current workspace and initializes its value from the data in the file mytest.mat

3.1.2 Binary files I/O

`fwrite(File_id, Variable, 'Precision')` writes (outputs) the contents of a variable to a file in binary format. A file must be opened with **`fopen()`** before executing any **`fwrite()`** commands.
The possible precision (data type) options are listed in 3.1

Create a matrix A and a matrix B :
A= [1 3 5; 2 5 10 ; 9 4 3; 0 2 1] and B= [4 5; 2 8]
Let's create a binary file **myfile.bin** where we will write both matrices as follows:

>> fid=fopen ('myfile.bin','w'); % open binary file for writing ; fid is a file identifier
>>fwrite (fid, A); % write in Matrix A
>>fwrite (fid, B); % write in Matrix B
>>fclose (fid); %close the file

Now we can read both matrices A and B, assuming that we know their dimensions. As we said it is important to know the dimensions of the data stored in the binary file

>>fid=fopen ('myfile.bin','r') ; % open binary file for reading
>>MatC=fread (fid, [4, 3]) % pull out a 4 by 3 matrix
>> MatC=

```
1   3    5
2   5   10
9   4    3
0   2    1
```

Which is exactly the matrix A.

>>fid= fopen (' myfile.bin' , 'r') ;
>>MatD= fread (fid, [2, 2])
>>MatD=

```
4   5
2   8
```

Which was the matrix B, last entered in the binary file.
Now if we do
>>fid=fopen('myfile.bin','r');
>>MatE=fread (fid, [1, 3])

It will return an empty matrix MatE=[ ] because the binary file has been emptied previously; no more data in it.

>> fid=fopen('myfile.bin','r');
>> mydata=fread (fid) % returns a column vector of all numbers in file
>>mydata=

```
1
2
9
0
3
5
4
2
5
10
3
1
4
```

2
5
8
>>fclose(fid);
We can always reconstruct our original matrix A from the vector **mydata**

>>MatrixA=[mydata(1:4)'; mydata (5:8)'; mydata (9:12)']'

Of course all kinds of operations can be performed on the data stored in a binary file.

4-Math Functions

3.1 elementary functions: (see text book)

abs(x)	absolute value of x
sqrt(x)	square root of x
round (x)	rounds x to the nearest integer
fix(x)	rounds (or truncates) x to the nearest integer toward 0
floor(x)	rounds x to the nearest integer toward –infinity
ceil(x)	rounds x to the nearest integer toward + infinity
sign(x)	returns a value of -1 if x<0, a value of 0 if x=0 , and a value of 1 otherwise
rem(x,y)	computes the remainder of x/y
log(x)	ln(x) natural log
log10(x)	log of x to the base of 10

4.2 Trigonometric Functions

sin(x), cos(x), tan(x),… are the common trig. functions.
The arc functions are written as asin(x), acos(x), atan(x),….
atan2(y,x) returns the angle in radians.

4-3 Logical Operators:

&	Logical AND
|	Logical OR
~	Logical complement (NOT)
xor	exclusive or

Examples:
>>if x=[0 5 3 7] and y= [0 2 8 7]
>>m=(x>y) & (x>4) results in m= [0 1 0 0] since condition is true only for x2
>>c=x|y results in c=[0 1 1 1] since xi or yi is non-zero for i= 2,3, and 4
>>d=~(x|y) results in d=[1 0 0 0] since it is the logical complement of (x|y)

4-4 Relational Operators

<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
~=	not equal

Examples: for the above two vectors x and y :

>>**k=x<y** results in [0 0 1 0] because xi<yi for only i=3

>>s=x~=y results in [0 1 1 0]

>>**r=x>=y** results in [1 1 0 1]