Programming with MATLAB

Goals

1. Learning how to create well-documented M-files in the edit window and invoke them from the command window.
2. Understanding how script and function files differ.
3. Understanding how to incorporate help comments in functions.
4. Knowing how to set up M-files so that they interactively prompt users for information and display results in the command window.
5. Understanding the role of subfunctions and how they are accessed.
6. Knowing how to create and retrieve data files.
7. Learning how to write clear and well-documented M-files by employing structured programming constructs to implement logic and repetition.
8. Recognizing the difference between if...elseif and switch constructs.
9. Recognizing the difference between for...end and while structures.
10. Knowing how to animate MATLAB plots.
11. Understanding what is meant by vectorization and why it is beneficial.
12. Understanding how anonymous functions can be employed to pass functions to function M-files.

MATLAB Environment

An *M-file*, stored with `.m` extension, consists of a series of statements that can be run all at once. They come in two flavors: *script files* and *function files*.

A *script file* is merely a series of MATLAB commands that are saved on a file.

Function files are M-files that start with the word `function`. And can accept input arguments and return outputs.

Remember to add current working folder to the MATLAB path. Go to Home, choose Set Path and add the folder with its subfolders.

Script files:

Open the editor with the selection: **New, Script**. Type in the following statements and save it as `testScript.m` in the working folder located in the MATLAB path.

```
    g = 9.81; m = 68.1; t = 12; cd = 0.25;
    v = sqrt(g*m/cd) * tanh(sqrt(g*cd/m) * t)
```

Then, in the command line, type

```
    >> testScript
```

and see the output.

Function files:

They are M-files that start with the word `function` and analogous to user-defined functions in Fortran, Visual Basic or C. The syntax for the function file is:

```
    function outvar = funcname(arglist)
    % helpcomments
    statements
    outvar = value;
    end            % this could be removed
```

where `outvar` = the name of the output variable, `funcname` = the functions name, `arglist` = the functions argument list (i.e., comma-delimited values that are passed into the function), `helpcomments` = text that provides the user with information regarding the function (these can be invoked by typing Help `funcname` in the command window), and `statements` = MATLAB statements that compute the value that is assigned to `outvar`.

Note. The M-file should be saved as `funcname.m`. The function can then be run by typing `funcname` in the command window.

Example. Type the following function in the editor:

```
function v = freefall(t, m, cd)
%freefall: bungee velocity with second-order drag
%v=freefall(t,m,cd) computes the free-fall velocity of an
%object with second-order drag
%input: t = time (s)   m = mass (kg)  cd = second-order drag
%coefficient (kg/m)
%output: v = downward velocity (m/s)
g = 9.81; % acceleration of gravity
v = sqrt(g * m / cd)*tanh(sqrt(g * cd / m) * t);
```

Save the file as `freefall.m`. Type the followings and see the outputs:

```
>> freefall(12, 68.1, 0.25)

>> freefall(10, 95, 0.25)

>> help freefall

>> g
```

Note. The variables within a function are said to be *local* and are erased after the function is executed. In contrast, the variables in a script retain their existence after the script is executed.

Example. Write a function, and name it `stats`, that computes the mean and the standard deviation of an input vector x. Then, provide an output for the vector x=[2, 4.5, 3, 7.4, 5]

```
function [mean, stdev] = stats(x)
n = length(x);
mean = sum(x)/n;
stdev = sqrt(sum((x-mean).^2/(n - 1)));
```

Now, in the command line, we type the followings:

```
>> y=[2 4.5 3 7.4 5];

>> [mean_y, SD_y]=stats(y)
```

Remarks:

1. Any variables defined through the command line are within the MATLAB workspace.

2. Workspace variables are not directly accessible to functions but rather are passed to functions via their arguments.

Try the followings:

```
>> n=19;

>> y=[2 4.5 3 7.4 5];

>> [mean_y, SD_y]=stats(y)

>> n         % the output for n will not be the one defined in the function statements.
```

Global variables: It might be convenient to have access to a variable in several contexts by defining it as global. Here is the syntax if you would like to declare x, y, and p variables as global:

```
Global x, y, p
```

Note. Functions can call other functions. In this case, the first function is called the *main* or *primary* function.

```
function v = freefallsubfunc(t, m, cd)
v = vel(t, m, cd);
end
function v = vel(t, m, cd)
g = 9.81;
v = sqrt(g * m / cd)*tanh(sqrt(g * cd / m) * t);
```

Input-output functions:

Syntax for input function:

`n = input('promptstring')` % The function displays the promptstring, waits for keyboard input to be assigned to n

The input function can also return user input as a string. To do this, an 's' is appended to the function's argument list.

```
name = input('Enter your name: ','s')
```

Syntax for output (disp) function:

```
disp(value)
```

where `value` = the value/variable/string you would like to display.

Example. Type and save the following statements in the editor. Then run the function in the command line.

```
function freefalli
g = 9.81; % acceleration of gravity
m = input('Mass (kg): ');
cd = input('Drag coefficient (kg/m): ');
t = input('Time (s): ');
disp(' ')
disp(‘Velocity (m/s):')
disp(sqrt(g * m / cd)*tanh(sqrt(g * cd / m) * t))
```

Save it as `freefalli.m`.

```
>> freefalli
```

Note. Another output function `is fprintf` function that provides more control over the display of information. The syntax is:

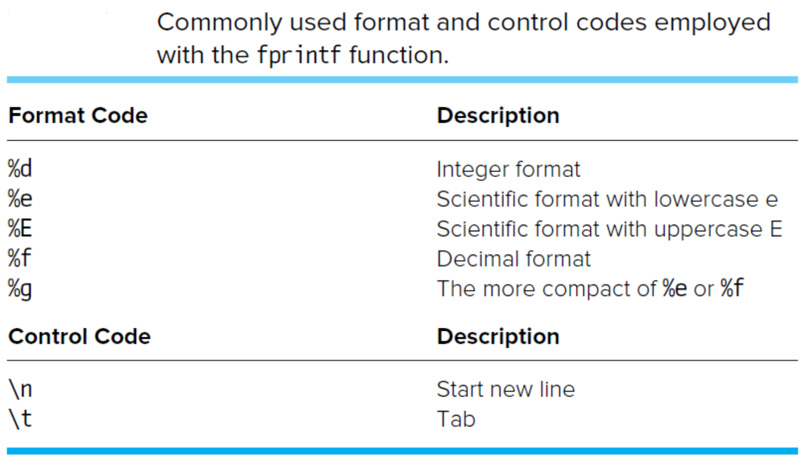
```
fprintf('format', x, ...)
```

For example:

```
>> fprintf('The velocity is %8.4f m/s\n', velocity)
```

displays the value of variable `velocity` using eight digits with four decimal digits along with a message. The following table provides more common formats for `fprintf` function. Try the following and see the output:

```
>> fprintf('%5d %10.3f %8.5e\n',100,2*pi,pi);
```



Commonly used format and control codes employed with the `fprintf` function.

Format Code	Description
%d	Integer format
%e	Scientific format with lowercase e
%E	Scientific format with uppercase E
%f	Decimal format
%g	The more compact of %e or %f
Control Code	**Description**
\n	Start new line
\t	Tab

Remark. The `clear` command removes all variables from the workspace. However, one can save and retrieve functions or variables any time using the following commands:

```
save filename var1 var2 ... varn

load filename var1 var2 ... varn
```

Structures:

Conditional statements: Its syntaxes are

```
if condition
     statements
end
```

or

```
if condition
     statements_1
else
     statements_2
end
```

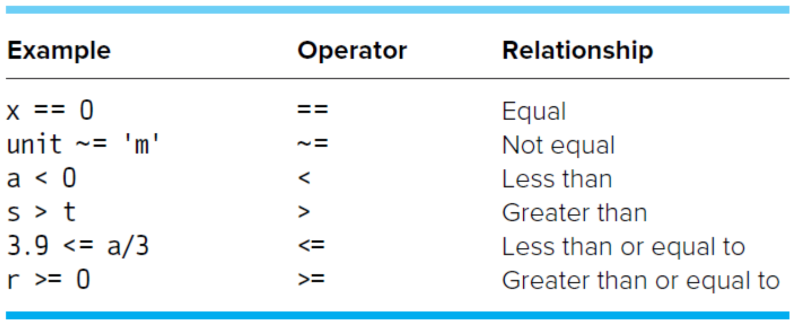
or

```
if condition_1
     statements_1
elseif condition_2
     statements_2
elseif condition_3
.
.
.
else
     statements_else
end
```

A simple M-file to evaluate whether a grade is passing:

```
function grader(grade)
  if grade >= 60
     disp('passing grade')
end
```

The following table provides a summary of relational operators in MATLAB.



Example	Operator	Relationship
`x == 0`	`==`	Equal
`unit ~= 'm'`	`~=`	Not equal
`a < 0`	`<`	Less than
`s > t`	`>`	Greater than
`3.9 <= a/3`	`<=`	Less than or equal to
`r >= 0`	`>=`	Greater than or equal to

MATLAB also uses logical operators to test logical conditions:

- ~(Not). Logical negation on an expression.
```
~ expression
```
  Returns the opposite value of an expression

- & (And). Logical conjunction on two expressions.
```
Expression1 & expression2
```
  Returns true only if both expressions are true

- || (Or). Logical disjunction on two expressions.
```
Expression1 || expression2
```
  Returns false only if both expressions are false.

Example. A simple code that returns the sign function
```
function sgn = mysign(x)
     if x > 0
          sgn = 1;
     elseif x < 0
          sgn = -1;
     else
          sgn = 0;
     end
end
```

Switch structure: Its performance is similar to the `if...elseif` structure. Rather than testing individual conditions, the branching is based on the value of a single test expression. Depending on its value, different blocks of code are implemented. It has the general syntax
```
switch testexpression
     case value1
          statements1
```

```
        case value2
             statements2
        .
        .
        .
        otherwise
             statementsotherwise
    end
```

Example. Display a proper message based on the string variable grade:

```
    grade = 'B';
    switch grade
         case 'A'
              disp('Excellent')
         case 'B'
              disp('Good')
         case 'C'
              disp('Mediocre')
         case 'D'
              disp('Whoops')
         case 'F'
              disp('Would like fries with your order?')
         otherwise
              disp('Huh!')
    end
```

Loops

The for ... end structure: The syntax is:

```
    for index = start:step:finish
         statements
    end
```

For example, try the following:

```
    for j = 10:-1:1
         disp(j)
    end
```

Example. Write a function that computes the factorial of a whole number.

```
    function fout = factor(n)
    x = 1;
    for i = 1:n
```

```
          x = x * i;
     end
     fout = x;
     end
```

The while and while … break structures: The syntaxes are:

```
     while condition
          statements
     end
```

and

```
     while (1)
          statements
          if condition,
               break,
          end
          statements
     end
```

For example, try the following:

```
     x = 238;
     while (1)
          If x < 0,
               break,
          end
          x = x - 5;
     end
```

Animations. there are special functions, `getframe` and `movie`, that allow you to capture a sequence of plots and then play them back. The `getframe` captures a snapshot (*pixmap*) of the current axes or figure. It is usually used in a for loop to assemble an array of movie frames for later playback with the `movie` function.

```
     movie (m,n,fps)
```

where m = the vector/matrix holding the sequence of frames, n = an optional variable specifying how many times the movie is to be repeated (if it is omitted, the movie plays once), and fps = an optional variable that specifies the movie's frame rate (if it is omitted, the default is 12 frames per second).

Example. In the absence of air resistance, the Cartesian coordinates of a projectile launched with an initial velocity (v_0) and angle (θ_0) can be computed with

$$x = v_0 \cos(\theta_0) t$$
$$y = v_0 \sin(\theta_0) t - 0.5 g t^2$$

where $g = 9.81$ m/s^2. Develop a script to generate an animated plot of the projectile's trajectory given that v_0 = 5 m/s and $\theta_0 = 45^\circ$.

```
clc,clf,clear
g = 9.81; theta0 = 45*pi/180; v0 = 5;
t(1) = 0;x = 0;y = 0;
plot(x,y,'o','MarkerFaceColor','b','MarkerSize',8)
axis([0 3 0 0.8])
M(1) = getframe;
dt = 1/128;
for j = 2:1000
t(j) = t(j - 1) + dt;
x = v0*cos(theta0)*t(j);
y = v0*sin(theta0)*t(j) - 0.5*g*t(j)^2;
plot(x,y,'o','MarkerFaceColor','b','MarkerSize',8)
axis([0 3 0 0.8])
M(j) = getframe;
if y< = 0, break, end
end
pause
movie(M,1)
```

Anonymous functions allow us to create a simple function without creating an M-file. Use the following syntax:

```
fhandle = @(arglist) expression
```

where arglist = a comma separated list of input arguments to be passed to the function, and expression = any single valid MATLAB expression.
For example, try the following

```
>> f1= @(x,y) x^2 + y^2;
>> f1(2,3)
```

Also try the following and see the output:

```
>> vel = @(t) ...
     sqrt(9.81*68.1/0.25)*tanh(sqrt(9.81*0.25/68.1)*t);
>> fplot(vel,[0 12])
```

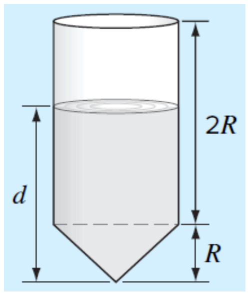
Practice problem:

1. Develop an M-file function to determine the average value of a function over a range. Illustrate its use for the bungee jumper velocity over the range from t = 0 to 12 s:

$$v(t) = \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}}\, t\right)$$

where g = 9.81, m = 68.1, and c_d = 0.25.

2. The following figure shows a cylindrical tank with a conical base. If the liquid level is quite low, in the conical part, the volume is simply the conical volume of liquid. If the liquid level is midrange in the cylindrical part, the total volume of liquid includes the filled conical part and the partially filled cylindrical part.



Write an M-file to compute the tank's volume as a function of given values of R and d. Design the function so that it returns the volume for all cases where the depth is less than 3R. Return an error message ("Overtop") if you overtop the tank—that is, d > 3R. Test it with the following data: (Note that the tank's radius is R)

R	0.9	1.5	1.3	1.3
d	1	1.25	3.8	4.0

3. The sine function can be evaluated by the following infinite series:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots$$

Create an M-file to implement this formula so that it computes and displays the values of $\sin x$ as each term in the

Series is added. In other words, compute and display in sequence the values for

$$\sin x = x$$

$$\sin x = x - \frac{x^3}{3!}$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!}, \quad \text{etc.}$$

up to the order term of your choosing. For each of the preceding, compute and display the percent relative error as

$$\%\text{ error} = \frac{\text{true}-\text{series approximation}}{\text{true}} \times 100\%$$

As a test case, employ the program to compute $\sin(0.9)$ for up to and including eight terms–that is, up to the term $\frac{x^{15}}{15!}$.

4. Develop an M-file function that is passed a numeric grade from 0 to 100 and returns a letter grade according to the scheme shown in the following table. The first line of the function should be

```
function grade = lettergrade(score)
```

Design the function so that it displays an error message and terminates in the event that the user enters a value of score that is less than zero or greater than 100. Test your function with 89.9999, 90, 45, and 120.

Letter	Criteria
A	$90 \leq$ numeric grade ≤ 100
B	$80 \leq$ numeric grade < 90
C	$70 \leq$ numeric grade < 80
D	$60 \leq$ numeric grade < 70
F	numeric grade < 60

Resources:

Chapra, Steven C. (2017). Numerical Methods with MATLAB for Engineers and Scientists, 4th Ed. McGraw Hill.