MATLAB WORKBOOK

CME 102 **Winter 2008-2009**

Eric Darve
Hung Le

Introduction

This workbook aims to teach you Matlab and facilitate the successful integration of Matlab into the CME 102 (Ordinary Differential Equations for Engineers) curriculum. The workbook comprises three main divisions; **M**atlab Basics, Matlab Programming and **N**umerical Methods for Solving ODEs. These divisions are further subdivided into sections, that cover specific topics in Matlab. Each section begins with a listing of Matlab commands involved (printed in boldface), continues with an example that illustrates how those commands are used, and ends with practice problems for you to solve.
The following are a few guidelines to keep in mind as you work through the examples:

a) You must turn in all Matlab code that you write to solve the given problems. A convenient method is to copy and paste the code into a word processor.

b) When generating plots, make sure to create titles and to label the axes. Also, include a legend if multiple curves appear on the same plot.

c) Comment on Matlab code that exceeds a few lines in length. For instance, if you are defining an ODE using a Matlab function,explain the inputs and outputs of the function. Also, include in-line comments to clarify complicated lines of code.

Good luck!

1 Matlab Basics

1.1 Matrix and Vector Creation

Commands:

`;`	Placed after a command line to suppress the output.
`eye(m,n)`	Creates an `m` × `n` matrix with ones on the main diagonal and zeros elsewhere (the main diagonal consists of the elements with equal row and column numbers). If `m` = `n`, `eye(n)` can be used instead of `eye(n,n)`. Creates the `n`-dimensional identity matrix.
`ones(m,n)`	Creates an `m`-by-`n` matrix of ones (`m` rows, `n` columns).
`zeros(m,n)`	Creates an `m`-by-`n` matrix of zeros (`m` rows, `n` columns).
`a:b:c`	Generates a row vector given a start value `a` and an increment `b`. The last value in the vector is the largest number of the form `a+nb`, with `a+nb` ≤ `c` and `n` integer. If the increment is omitted, it is assumed to be 1.
`sum(v)`	Calculates the sum of the elements of a vector `v`.
`size(A)`	Gives the two-element row vector containing the number of row and columns of `A`. This function can be used with `eye`, `zeros`, and `ones` to create a matrix of the same size of `A`. For example `ones(size(A))` creates a matrix of ones having the same size as `A`.
`length(v)`	The number of elements of `v`.
`[]`	Form a vector/matrix with elements specified within the brackets.
`,`	Separates columns if used between elements in a vector/matrix. A space works as well.
`;`	Separates rows if used between elements in a vector/matrix.

Note: More information on any Matlab command is available by typing "`help command_name`" (without the quotes) in the command window.

1.1.1 Example

a) Create a matrix of zeros with 2 rows and 4 columns.

b) Create the row vector of odd numbers through 21,

```
L =
     1     3     5     7     9    11    13    15    17    19    21
```

Use the colon operator.

c) Find the sum `S` of vector `L`'s elements.

d) Form the matrix

```
A =
     2     3     2
     1     0     1
```

Solution:

a)
```
>> A = zeros(2,4)
A =
     0     0     0     0
     0     0     0     0
```

b)
```
>> L = 1 : 2 : 21
L =
     1     3     5     7     9    11    13    15    17    19    21
```

c)
```
>> S = sum(L)
S =
   121
```

d)
```
>> A = [2, 3, 2; 1 0 1]
A =
     2     3     2
     1     0     1
```

1.1.2 Your Turn

a) Create a 6 x 1 vector `a` of zeros using the zeros command.

b) Create a row vector `b` from 325 to 405 with an interval of 20.

c) Use sum to find the sum `a` of vector `b`'s elements.

1.2 Matrix and Vector Operations

Commands:

`+`	Element-by-element addition. (Dimensions must agree)
`-`	Element-by-element subtraction. (Dimensions must agree)
`.*`	Element-by-element multiplication. (Dimensions must agree)
`./`	Element-by-element division. (Dimensions must agree)
`.^`	Element-by-element exponentiation.
`:`	When used as the index of a matrix, denotes "ALL" elements of that dimension.
`A(:,j)`	`j`-th column of matrix `A` (column vector).
`A(i,:)`	`i`-th row of matrix `A` (row vector).
`.'`	Transpose (Reverses columns and rows).
`'`	Conjugate transpose (Reverses columns and rows and takes complex conjugates of elements).
`*`	Matrix multiplication, Cayley product (row-by-column, not element-by-element).

1.2.1 Example

a) Create two different vectors of the same length and add them.

b) Now subtract them.

c) Perform element-by-element multiplication on them.

d) Perform element-by-element division on them.

e) Raise one of the vectors to the second power.

f) Create a 3×3 matrix and display the first row of and the second column on the screen.

Solution:

a)
```
>> a = [2, 1, 3]; b = [4 2 1]; c = a + b

c =

     6     3     4
```

b)
```
>> c = a - b

c =

    -2    -1     2
```

c)
```
>> c = a .* b

c =

     8     2     3
```

d)
```
>> c = a ./ b

c =

    0.5000    0.5000    3.0000
```

e)
```
>> c = a .^ 2

c =

     4     1     9
```

f)
```
>> d = [1 2 3; 2 3 4; 4 5 6]; d(1,:), d(:,2)

ans =

     1     2     3
```

```
ans =

     2
     3
     5
```

1.2.2 Your Turn

a) Create the following two vectors and add them.

```
a =

     5     3     1

b =

     1     3     5
```

b) Now subtract them.

c) Perform element-by-element multiplication on them.

d) Perform element-by-element division on them.

e) Raise one of the vectors to the second power.

f) Create a 3×3 matrix and display the first row of and the second column on the screen.

1.3 Basic 1D Plot Commands

Commands:

`figure`	Creates a figure window to which MATLAB directs graphics output. An existing figure window can be made current using the command `figure(n)`, where `n` is the figure number specified in the figure's title bar.
`plot(x,y,'s')`	Generates a plot of y w.r.t. x with color, line style and marker specified by the character string `s`. For example, `plot(x,y,'c:+')` plots a cyan dotted line with a plus marker at each data point. The string `s` is optional and default values are assumed if `s` is not specified. For default values, list of available colors, line styles, markers and their corresponding character representations, type `help plot`.
`axis([xmin,xmax, ymin,ymax])`	Specifies axis limits for the x- and y- axes. This command is optional and by default MATLAB determines axes limits depending on the range of data used in plotting.
`title('...')`	Adds a title to the graph in the current figure window. The title is specified as a string within single quotes.
`xlabel('...')`	Adds a label to the x-axis of the graph in the current figure window. This is again specified as a string within single quotes.

`ylabel('...')`	Similar to the `xlabel` command.
`grid on`	Adds grid lines to the current axes.
`grid off`	Removes grid lines from the current axes.

1.3.1 Example

Let us plot projectile trajectories using equations for ideal projectile motion:

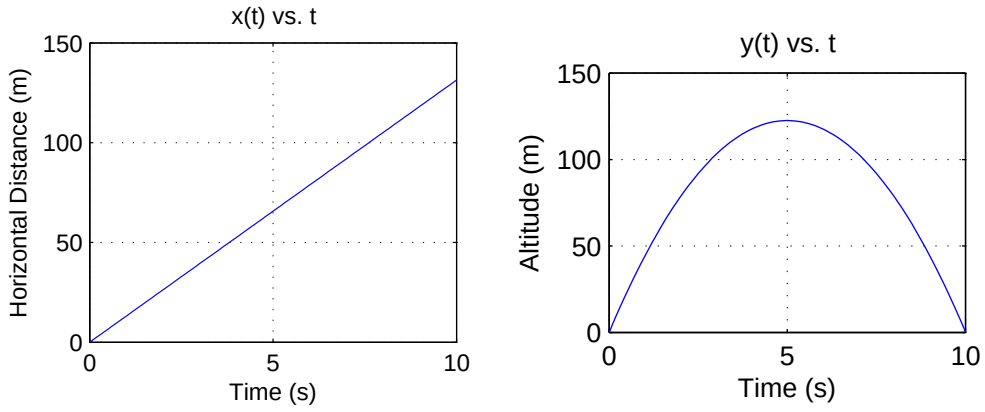
$$\begin{aligned} y(t) &= y_0 - \frac{1}{2}gt^2 + (v_0 \sin(\theta_0))\, t, \\ x(t) &= x_0 + (v_0 \cos(\theta_0))\, t, \end{aligned}$$

where $y(t)$ is the vertical distance and $x(t)$ is the horizontal distance traveled by the projectile in metres, g is the acceleration due to Earth's gravity $= 9.8$ m/s^2 and t is time in seconds. Let us assume that the initial velocity of the projectile $v_0 = 50.75$ m/s and the projectile's launching angle $\theta_0 = \frac{5\pi}{12}$ radians. The initial vertical and horizontal positions of the projectile are given by $y_0 = 0$ m and $x_0 = 0$ m. Let us now plot `y` vs. `t` and `x` vs. `t` in two separate graphs with the vector: `t=0:0.1:10` representing time in seconds. Give appropriate titles to the graphs and label the axes. Make sure the grid lines are visible.

Solution:

We first plot `x` and `y` in separate figures:

```
>> t = 0 : 0.1 : 10;
>> g = 9.8;
>> v0 = 50.75;
>> theta0 = 5*pi/12;
>> y0 = 0;
>> x0 = 0;
>> y = y0 - 0.5 * g * t.^2 + v0*sin(theta0).*t;
>> x = x0 + v0*cos(theta0).*t;
>>
>> figure;
>> plot(t,x);
>> title('x(t) vs. t');
>> xlabel('Time (s)');
>> ylabel('Horizontal Distance (m)');
>> grid on;
>>
>> figure;
>> plot(t,y);
>> title('y(t) vs. t');
>> xlabel('Time (s)');
>> ylabel('Altitude (m)');
>> grid on;
```



1.3.2 Your Turn

The range of the projectile is the distance from the origin to the point of impact on horizontal ground. It is given by $R = v_0 \cos(\theta_0)$. To estimate the range, your trajectory plots should be altered to have the horizontal distance on the x-axis and the altitude on the y-axis. This representation will clearly show the path of the projectile launched with a certain initial angle. This means you will have to plot `y` vs. `x`.
Observing the formula for the projectile's range, we see that to increase the range we will have to adjust the launching angle. Use the following adjusted angles to create two more trajectory plots (`y` vs. `x`), one for each angle, and determine which launching angle results in a greater range:

$$\theta_0^1 = \left(\frac{5\pi}{12} - 0.255\right) \text{ radians and}$$
$$\theta_0^2 = \left(\frac{5\pi}{12} - 0.425\right) \text{ radians.}$$

The time vectors for these angles should be defined as `t = 0:0.1:9` and `t = 0:0.1:8` respectively.

1.4 Plotting Multiple Functions I

Commands:

`plot(x,y)`	Creates a plot of `y` vs. `x`.
`plot(x,y1,x,y2, ...)`	Creates a multiple plot of `y1` vs. `x`, `y2` vs. `x` and so on, on the same figure. MATLAB cycles through a predefined set of colors to distinguish between the multiple plots.
`hold on`	This is used to add plots to an existing graph. When `hold` is set to `on`, MATLAB does not reset the current figure and any further plots are drawn in the current figure.
`hold off`	This stops plotting on the same figure and resets axes properties to their default values before drawing a new plot.
`legend`	Adds a legend to an existing figure to distinguish between the plotted curves.
`ezplot('f(x)', [x0,xn])`	Plots the function represented by the string `f(x)` in the interval `x0` $\leq$ `x` $\leq$ `xn`.

Note: Make sure that when you use the `"hold"` command to make multiple plots, you should specify the color and/or line style in the `plot` command. Otherwise all the plots will be of the same default (blue) color and line style. Check this out.

1.4.1 Example

a) Using the `plot` command for multiple plots, plot $y = \sin(x)$ and $y = \cos(x)$ on the same graph for values of `x` defined by: `x = 0:pi/30:2*pi`.

b) Using the `plot` command for a single plot and the `hold` commands, plot $y = \sin(x)$ and $y = \cos(x)$ on the same graph for values of `x` defined by: `x = 0:pi/30:2*pi`.

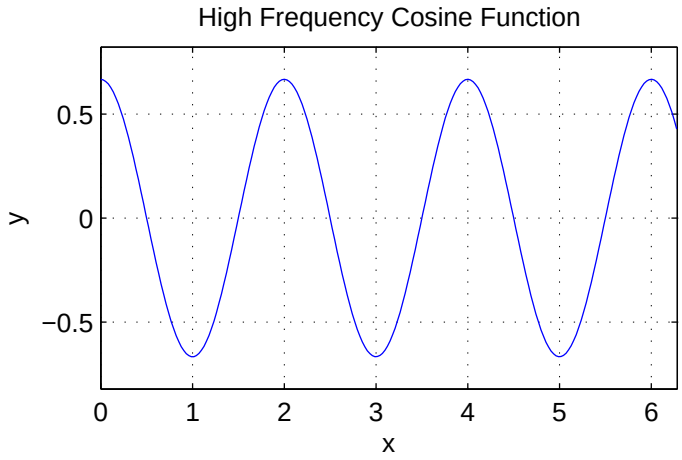
c) Using the `ezplot` command, plot $y = \frac{2}{3}\cos(\pi x)$ for values of `x` such that $0 \leq x \leq 2 * pi$.

Solution:

```
>> x = 0 : pi/30 : 2*pi;
>> plot(x,sin(x),x,cos(x));
>> title('y = sin(x) and y = cos(x)');
>> xlabel('x');
>> ylabel('y');
>> legend('y = sin(x)','y = cos(x)');
>> grid on;

>> x = 0 : pi/30 : 2*pi;
>> plot(x,sin(x));
>> title('y = sin(x) and y = cos(x)');
>> xlabel('x');
>> ylabel('y');
>> grid on;
>> hold on;
>> plot(x,cos(x),'r');
>> legend('y = sin(x)','y = cos(x)');

>> ezplot('(2/3)*cos(pi*x)',[0,2*pi]);
>> title('High Frequency Cosine Function');
>> xlabel('x');
>> ylabel('y');
>> grid on;
```



1.4.2 Your Turn

a) Using the `plot` command for multiple plots, plot $y = \text{atan}(x)$ and $y = \text{acot}(x)$ on the same graph for values of `x` defined by `x = -pi/2:pi/30:pi/2`.

b) Using the `plot` command for a single plot and the `hold` commands, plot $y = \text{atan}(x)$ and $y = \text{acot}(x)$ on the same graph for values of `x` defined by `x = -pi/2:pi/30:pi/2`.

c) Using the `ezplot` command, plot $y = \frac{2}{3}\sin(9\pi x)$, for values of `x` such that `0` $\leq$ `x` $\leq$ `2` $*$ `pi`.

1.5 Plotting Functions II

Commands:

`log(n)`	Calculates the natural logarithm (base `e`) of `n`.
`semilogy(x,y)`	Graphs a plot of `y` vs. `x` using a logarithmic scale (powers of ten) on the `y`-axis.
`semilogx(x,y)`	Graphs a plot of `y` vs. `x` using a logarithmic scale (powers of ten) on the `x`-axis.
`loglog(x,y)`	Graphs a plot of `y` vs. `x` using a logarithmic scale (powers of ten) on both axes. The logarithmic scales prove most useful when the value spans multiple orders of magnitude.

1.5.1 Example

Graph the efficiency of several programming algorithms according to big-O notation, a method of describing the running time of algorithms. Each expression represents the scale by which an algorithm's computation time increases as the number of its input elements increases. For example, $O(n)$ represents an algorithm that scales linearly, so that its computation time increases at the same rate as the number of elements. The algorithms you must graph have the following big-O

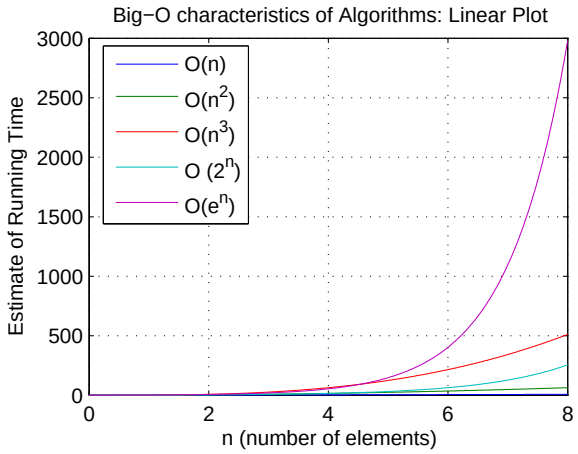
characteristics:

$$\begin{aligned}
\text{Algorithm \#1:} &\quad O(n) \\
\text{Algorithm \#2:} &\quad O(n^2) \\
\text{Algorithm \#3:} &\quad O(n^3) \\
\text{Algorithm \#4:} &\quad O(2^n) \\
\text{Algorithm \#5:} &\quad O(e^n)
\end{aligned}$$

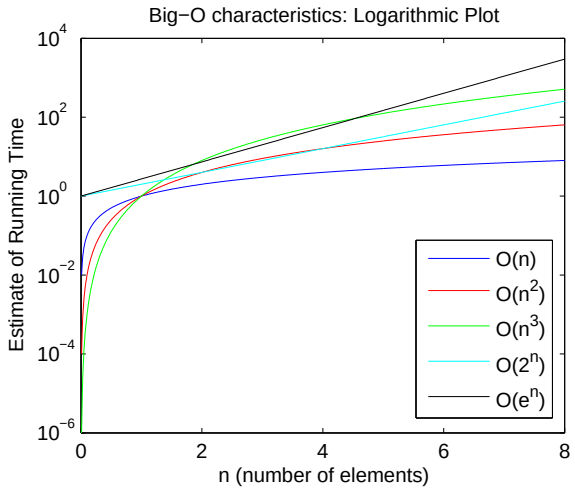
After generating an initial graph with ranging from 0 to 8, use logarithmic scaling on the y-axis of a second graph to make it more readable. You can also use the mouse to change the y-axis scale. Go to the main menu of the figure, click Edit>Axes Properties..., the property editor dialogue will pop out. There, you can also change the font, the range of the axes, ... Try to play with it.

Solution:

```
>> n=0:0.01:8;
>> plot(n,n,n,n.^2,n,n.^3,n,2.^n,n,exp(n))
>> title('Big-O characteristics of Algorithms: Linear Plot')
>> ylabel('Estimate of Running Time')
>> xlabel('n (number of elements)')
>> legend('O(n)','O(n^2)','O(n^3)', 'O (2^n)','O(e^n)')
>> grid on;
```




```
>> n = 0:0.01:8;
>> semilogy(n,n,'b',n,n.^2,'r',n,n.^3,'g',n,2.^n,'c',n,exp(n),'k')
>> title('Big-O characteristics: Logarithmic Plot')
>> ylabel('Estimate of Running Time')
>> xlabel('n (number of elements)')
>> legend('O(n)','O(n^2)','O(n^3)', 'O(2^n)','O(e^n)')
```



1.5.2 Your Turn

Your task is to graph algorithms with the following big-O characteristics:

$$\begin{aligned} \text{Algorithm } \#1: &\quad O(n \ln n) \\ \text{Algorithm } \#2: &\quad O(\sqrt{n}) \\ \text{Algorithm } \#3: &\quad O(\ln n) \end{aligned}$$

Note: The ln function in Matlab is given by `log(_)`.
Print both the linear and logarithmic plots, using a domain from `n = 1` to `n = 500` to observe the considerable improvement in readability that a logarithmic scale for the `y`-axis will provide. The logarithmic scale is very useful when attempting to compare values that are orders of magnitude apart on the same graph.
Do not use a grid for the logarithmic scale.

2 Matlab Programming

2.1 `for` and `while` Loops

Commands:

`for i = a:b`	The `for` loop repeats statements a specific number of times, starting with `i = a` and ending with `i = b`, incrementing `i` by 1 each iteration of the loop. The number of iterations will be `b - a + 1`.
`while` *condition*	The `while` loop repeats statements an indefinite number of times as long as the user-defined condition is met.
`for i = a:h:b`	The `for` loop works exactly the same except that `i` is incremented by `h` after each iteration of the loop.
`clear`	Clears all previously defined variables and expressions.
`fprintf`	Outputs strings and variables to the Command Window. See below for an example.
`abs(x)`	Returns the absolute value of the defined variable or expression `x`.
`factorial(n)`	Returns the factorial of the defined variable or expression `n`.

`...`	The ellipses can be used to break up long lines by providing a continuation to the next line. Strings must be ended before the ellipses but can be immediately restarted on the next line. Examples below show this.

Note: Neglecting the command `clear` can cause errors because of previously defined variables in the workspace.

`fprintf:`
This is an example of how to use `fprintf` to display text to the command window.

```
fprintf ('\nOrdinary Differential Equations are not so ordinary.\n');
fprintf ('-------------------------------------------------------'...
    '---------------\n');
fprintf ('This course is CME %g: ODEs for Engineers.  My expected'...
    ' grade is %g\n',102,100);
x = 100; y = 96;
fprintf ('The previous course was CME %g: Vector Calculus for '...
    'Engineers.  My grade was: %g\n',x,y);
```

The Matlab command window displays:

```
Ordinary Differential Equations are not so ordinary.
------------------------------------------------------------------------
This course is CME 102: ODEs for Engineers.  My expected grade is 100
The previous course was CME 100: Vector Calculus.  My grade was: 96
```

The command `fprintf` takes a string and prints it as is. The character `\n` is one of several "Escape Characters" for `fprintf` that can be placed within strings given to `fprintf`. `\n` specifies a new line. `%g` is one of many "Specifiers" that `fprintf` uses and it represents a placeholder for a value given later in the call to `fprintf`. The order of the arguments given to `fprintf` determine which `%g` is replaced with which variable or number. Experiment with the code above to see what `\n` can do and how `%g` can be used.

M-Files/Scripts:
Everything we have done so far has been in MATLABs interactive mode. However, MATLAB can execute commands stored in a regular text file. These files are called scripts or 'M-files'. Instead of writing the commands at the prompt, we write them in a script file and then simply type the name of the file at the prompt to execute the commands. It is almost always a good idea to work from scripts and modify them as you go instead of repeatedly typing everything at the command prompt.
A new M-file can be created by clicking on the "New M-file" icon on the top left of the Command Window. An M-file has a .m extension. The name of the file should not conflict with any existing MATLAB command or variable.
Note that to execute a script in an M-file you must be in the directory containing that file. The

current directory is shown above the Command Window in the drop down menu. You can click on the "..." icon, called "Browse for folder", (on the right of the drop-down menu) to change the current directory. The % symbol tells MATLAB that the rest of the line is a comment. It is a good idea to use comments so you can remember what you did when you have to reuse an M-file (as will often happen).
It is important to note that the script is executed in the same workspace memory as everything we do at the prompt. We are simply executing the commands from the script file as if we were typing them in the Command Window. The variables already existing before executing the script can be used by that script. Similarly, the variables in the script are available at the prompt after executing the script.

2.1.1 Example

After your 30 years of dedicated service as CEO, TI has transfered you to a subdivision in the Himalayas. Your task as head of the subdivision is to implement transcendental functions on the Himalayan computers. You decide to start with a trigonometric function, so you find the following Taylor Series approximation to represent one of these functions:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \ldots$$

However, since the computers in the Himalayas are extremely slow (possibly due to the high altitudes), you must use the Taylor Series as efficiently as possible. In other words, you must use the smallest possible number of terms necessary to be within your allowed error, which is 0.001. You will use $x = 3$ as the value at which to evaluate the function.

a) Compute and display the exact error of using the first 2 and 5 terms of the series as compared to the actual solution when the function is evaluated at $x = \frac{\pi}{3}$.

b) Compute and display the number of terms necessary for the function to be within the allowed error.

Solution:

a) CODE from M-file

```
clear;
x = pi/3;

% Iterate 2 terms for our approximation.
SIN_Approx2 = 0;
for j=0:2
   SIN_Approx2 = SIN_Approx2 + (-1)^j*x^(2*j+1)/factorial(2*j+1);
end
SIN_Error2 = abs(SIN_Approx2 - sin(x));

% Iterate 5 terms for our approximation.
SIN_Approx5 = 0;
for j=0:5
   SIN_Approx5 = SIN_Approx5 + (-1)^j*x^(2*j+1)/factorial(2*j+1);
```

```
end
SIN_Error5 = abs(SIN_Approx5 - sin(x));

fprintf('\nError with 2 terms:\n')
fprintf ('-------------------------\n')
fprintf ( 'sin(pi/3): %g\n',SIN_Error2 )

fprintf ('\nError with 5 terms: \n')
fprintf ('-------------------------\n')
fprintf ( 'sin(pi/3): %g\n',SIN_Error5)
```

OUTPUT:

```
Error with 2 terms:
-------------------------
sin(pi/3): 0.00026988

Error with 5 terms:
-------------------------
sin(pi/3): 2.90956e-010
```

b) CODE from M-file:

```
clear;
SIN_APP = 0; % This is the sine approximation.
n = 0; x = 3;

% Iterate until our approximation is below the error tolerance.
while abs( SIN_APP - sin(x) ) >= 0.001
    SIN_APP = SIN_APP + (-1)^n*x^(2*n+1)/factorial(2*n+1);
    n = n + 1;
end
SIN_Terms = n;
SIN_Error = abs( SIN_APP - sin(x) );
% Output
fprintf ('\nNumber of Terms Needed for the function to be within the'...
    ' allowed error:\n');
fprintf ('-------------------------------------------------------------'...
    '------------\n');
fprintf ('sin(3): %g terms | Error = %g\n',SIN_Terms,SIN_Error);
```

OUTPUT:

```
Number of Terms Needed for the function to be within the allowed error:
-----------------------------------------------------------------------
sin(3): 6 terms | Error = 0.000245414
```