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:

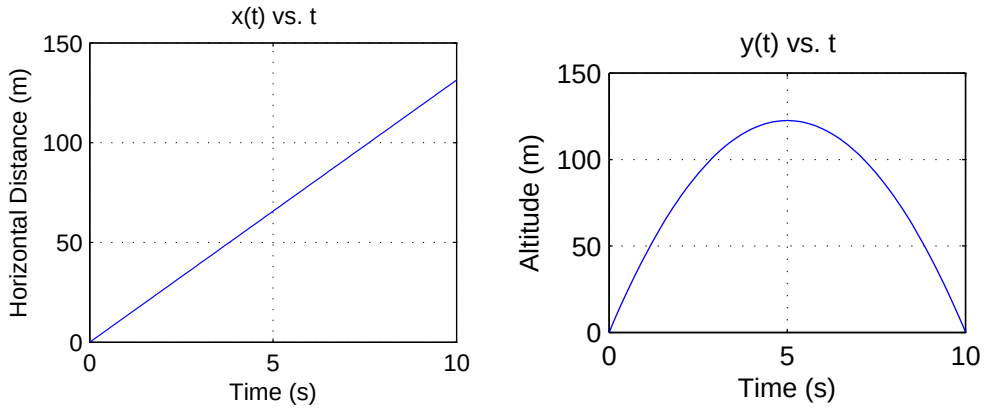
$$\begin{aligned} y(t) &= y_0 - \frac{1}{2}gt^2 + (v_0 \sin(\theta_0))\, t, \\ x(t) &= x_0 + (v_0 \cos(\theta_0))\, t, \end{aligned}$$

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

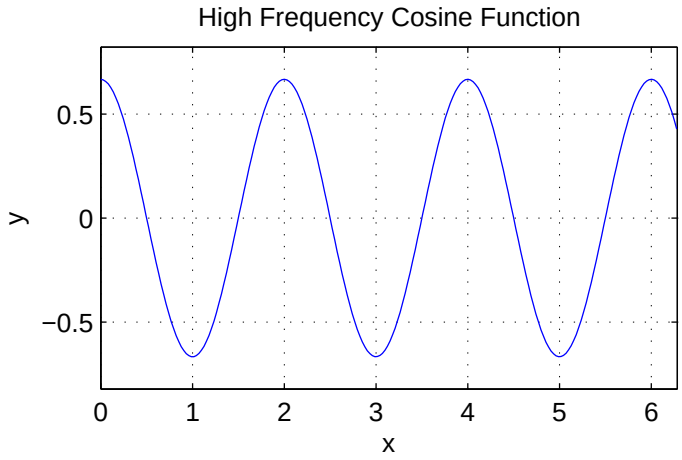
a) Using the `plot` command for multiple plots, plot $y = \sin(x)$ and $y = \cos(x)$ on the same graph for values of `x` defined by: `x = 0:pi/30:2*pi`.

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

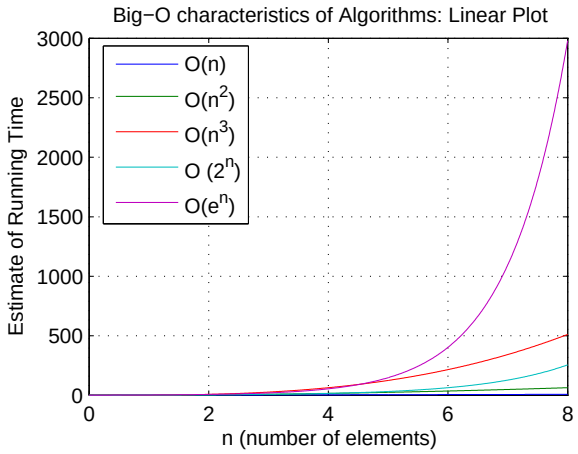
characteristics:

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

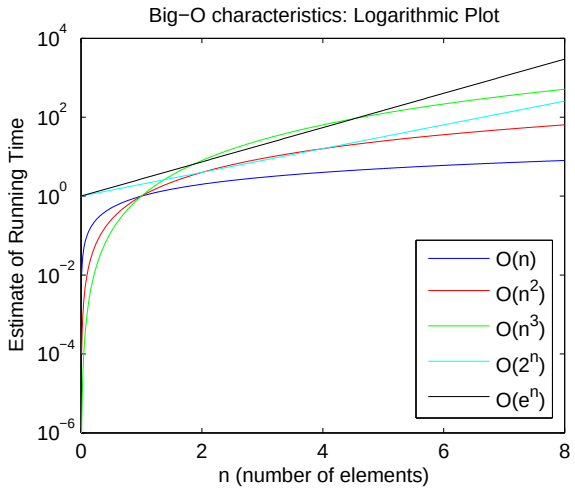
After generating an initial graph with ranging from 0 to 8, use logarithmic scaling on the y-axis of a second graph to make it more readable. You can also use the mouse to change the y-axis scale. Go to the main menu of the figure, click Edit>Axes Properties..., the property editor dialogue will pop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

2.1.2 Your Turn

The moment you finish implementing the trigonometric functions, you realize that you have forgotten your favorite function: the exponential! The Taylor Series for the exponential function is

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \ldots$$

a) Using a for loop, compute and display the error of the Taylor series when using 12 and 15 terms, each as compared to the exponential function when evaluated at $x = 2$.

b) Using a while loop, compute and display the number of terms necessary for this function to have an error within your allowed range of 0.001 when evaluated at $x = 3$. Display this error.

2.2 Symbols and Strings: Differentiation and Integration

Commands:

`syms x y z`	Declares variables `x`, `y`, and `z` as symbolic variables.
`diff(f)`	Symbolically computes derivative of the single-variable symbolic expression `f`.
`diff(f,n)`	Symbolically computes the `n`-th derivative of the single-variable symbolic expression `f`.
`diff(f,'x')`	Symbolically differentiates `f` with respect to `x`.
`diff(f,'x',n)`	Symbolically computes the `n`-th derivative with respect to `x` of `f`.
`int(f)`	Symbolically integrates the single-variable symbolic expression `f`.
`int(f,x)`	Symbolically integrates `f` with respect to `x`.

2.2.1 Example

Matlab is capable of symbolically computing the derivatives and integrals of functions. This can be very convenient to check your pencil and paper solutions; in some cases Matlab may be able to integrate functions which require very difficult change of variables. We will also learn in the next section how Matlab can be used to symbolically solve differential equations. For now we focus on derivatives and integrals.

The first step is to define symbolic variables using the `syms` command. Note that symbolic variables are different from numerical variables. They can not assume any numerical value. They are simply symbols or letters that can be manipulated symbolically.

Once these variables are declared we can define symbolic functions, differentiate and integrate them as is shown in the following examples:

a) Define the following function using `syms`:

$$f(x) = x^2 e^x - 5x^3.$$

b) Compute the integral, and the first and second derivatives of the above function symbolically.

c) Define the following function using `syms`:

$$f(x, y, z) = x^2 e^y - 5z^2.$$

Compute the integral with respect to x and second derivative with respect to z.

Solution:

a)
```
>> syms x
>> f = x^2*exp(x) - 5*x^3;
```

b)
```
>> int(f)
ans =
x^2*exp(x)-2*x*exp(x)+2*exp(x)-5/4*x^4


>> diff(f)
ans =
2*x*exp(x)+x^2*exp(x)-15*x^2


>> diff(f,2)
ans =
2*exp(x)+4*x*exp(x)+x^2*exp(x)-30*x
```

c)
```
>> syms x y z
>> f = x^2*exp(y) - 5*z^2;


>> int(f,'x')
ans =
1/3*x^3*exp(y)-5*z^2*x


>> diff(f,'z',2)
ans =
-10
```

2.2.2 Your Turn

a) Compute $\displaystyle\int \frac{1}{(x-1)(x-3)} dx$.

b) Compute $\displaystyle\int x^2 e^{-3x} dx$.

c) Compute the first and the second derivatives of $\operatorname{atan}\left(\frac{x}{5}\right)$.

2.3 Solving Differential Equations Symbolically

Commands:

`dsolve('eqn', 'cond1','cond2', ...)`	Symbolically compute the analytical solution of the ODE specified by the string `eqn` using optional initial conditions specified by the strings `cond1`, `cond2` etc. If no initial conditions are specified, the general solution of the ODE containing arbitrary constants is computed. If initial conditions are specified then the particular solution of the ODE is computed.

2.3.1 Example

Before considering the examples, the following points are to be noted:

- The letter `D` denotes differentiation with respect to the independent variable. The symbolic specification of $\frac{dy}{dt}$ is `'Dy'`. Higher order derivatives such as $\frac{d^2y}{dt^2}$, $\frac{d^3y}{dt^3}$ etc. are specified as `'D2y'`, `'D3y'` and so on.
- By default, the independent variable is `t`. The independent variable may be changed from `t` to some other symbolic variable by including that variable as the last input argument of the `dsolve` command. For example, the following equation has `x` as the independent variable:

  $$\frac{dy}{dx} = k * x$$

  and to solve the above for its general solution, the syntax is:

  ```
  dsolve('Dy = k*x','x')
  ```

- An initial condition such as $y(0) = 0$ is specified symbolically as `'y(0) = b'` and conditions like $y'(0) = 0$ and $y''(0) = 0$ are specified as `'Dy(0) = 0'` and `'D2y(0) = 0'` respectively.
- The general solution of an nth order ODE contains n arbitrary constants. To solve for the arbitrary constants and obtain a particular solution of such an ODE, we need n initial conditions. These constants appear as `C1`, `C2`, ... in your Matlab solution.
- Use the `pretty` command to print a symbolic expression in a readable format. For example,
  ```
  y = dsolve('Dy-y = 2*t*t')
  pretty(y)
  ```

a) Find the general solution of the following first order ODE:

$$\frac{dy}{dt} + y\tan(t) = \sin(2t)$$

b) Find the general solution of the following second order ODE:

$$\frac{d^2y}{dt^2} - \frac{dy}{dt} = 1 + t\sin(t)$$

Observe the number of arbitrary constants in the general solutions of the above two ODEs.

c) Find the particular solution of the following ODE:

$$\begin{aligned} \frac{dy}{dt} &= e^{-y} \\ y(0) &= 0 \end{aligned}$$

d) Find the particular solution of the following ODE:

$$\begin{aligned} \frac{dy}{dt} + Ky &= A + B\cos\left(\frac{1}{12}\pi t\right) \\ y(0) &= 0 \end{aligned}$$

where A, B and K are constants.

Solution:

a)
```
>> dsolve('Dy+y*tan(t) = sin(2*t)')
ans =
-2*cos(t)^2+cos(t)*C1
```

b)
```
>> dsolve('D2y-Dy=1+t*sin(t)')
ans =
-1/2*cos(t)-1/2*t*sin(t)-sin(t)+1/2*cos(t)*t+exp(t)*C1-t+C2
```

c)
```
>> dsolve('Dy=exp(-y)','y(0)=0')
ans =
log(t+1)
```

d)
```
dsolve('Dy+K*y = A+B*cos((1/12)*pi*t)','y(0)=0')
ans =

exp(-K*t)*(-144*B*K^2-A*pi^2-144*A*K^2)/(144*K^3+K*pi^2)
+(144*A*K^2+A*pi^2+144*B*K^2*cos(1/12*pi*t)
+12*B*pi*sin(1/12*pi*t)*K)/K/(144*K^2+pi^2)
```

2.3.2 Your Turn

a) Find the general solution of the following first order ODE:

$$\frac{dy}{dt} - y = e^{2t}.$$

b) Find the general solution of the following second order ODE:

$$\frac{d^2y}{dt^2} + 8\frac{dy}{dt} + 16y = 0.$$

c) Find the particular solution in a) given the following initial conditions:

$$y(0) = 5.$$

3 Forward Euler

Commands:

`norm(v)`	Calculates the Euclidean norm of a vector `v`.
`length(v)`	Returns the length of vector `v`.
`axis equal`	Sets the aspect ratio so that each axis has the same scale.

Forward Euler presents the most basic way to solve a differential equation numerically. Even though it has the most stringent stability criterion and the lowest accuracy of all numerical solvers, it is the simplest and easiest to program. In this section, we will explore:

- The algorithm and code used to program Forward Euler
- Application of Forward Euler to solving differential equations
- Computing local and global error
- Forward Euler's order of accuracy
- Forward Euler's stability limitations

Algorithm for Forward Euler

$$\boxed{y_{n+1} = y_n + h \cdot y'_n = y_n + h \cdot f(t_n, y_n)}$$

The forward Euler algorithm is an explicit means of obtaining the value of y at each time step using the previous time step and the local slope or derivative. The equation is derived from approximating the derivative y' with the difference quotient

$$\frac{y_{n+1} - y_n}{h} \approx y'_n = f(t_n, y_n).$$

As the boxed algorithm expresses, we solve for each subsequent time step y_{n+1} by adding the approximate change in y to the previous value of y. The last statement in the equation follows our convention of designating the slope or derivative as the function $f(t_n, y_n)$. Thus, by decreasing the time step h by which we march across our interval, we can make our solution more and more accurate. This algorithm is explicit because all values for the next time step are readily isolable from previous known values, making evaluation simple and systematic.

Code for Forward Euler

To solve any ordinary differential equation that is expressible in the explicit form

$$y'_n = f(t_n, y_n),$$

we **pass** a Matlab function `yprime.m` that contains the code for computing $f(t_n, y_n)$ to another function, say, `forwardEuler.m`. (The latter performs the numerical integration according to the forward Euler method.) Once we have the forward Euler algorithm implemented in Matlab, we

need alter only the function `yprime.m` when we change differential equations. (Copy the code below and use it for future problems.)
Although not shown in the definition of the function `forwardEuler.m` (below), when you run the code, Matlab uses a "handle" to pass the function `yprime.m` to `forwardEuler.m`. "Handles" are denoted using the "at" (or "around") character, "@"; e.g., to call the function `forwardEuler.m` (after creating your own function `yprime.m`), type

```
>> forwardEuler(@yprime, tspan,h,y0)
```

in the Matlab command window. (Make sure your "current directory" setting in Matlab is set to the directory containing `yprime.m` and `forwardEuler.m`.) *Inside* of `forwardEuler.m`, you can directly use your function using `yprime(t,y)`.
Use the following forward Euler script to solve any first-order ODE given a domain and an initial condition (these should be input parameters). Name the file, say, `forwardEuler.m`. The ODE must assume the following form:

$$y' = f(t, y)$$
$$y(t_0) = y_0$$

CODE from `forwardEuler.m`

```
function [T,Y] = forwardEuler(yprime,tspan,h,y0)

% function [T,Y] = forwardEuler(yprime, tspan,h,y0)
% Computes the numerical solution of an ODE using the forward Euler method.
% yprime: handle of function defining the ODE.
% tspan = [tmin tmax] : interval over which solution has to be computed
% h : time step
% y0 : initial condition
% -------------------------
% Output:
% T: sequence of time steps
% Y: values of y at each time step

% Initializing our interval with the first time value:
T(1) = tspan (1);

% Initializing our solution with the initial condition or guess:
Y(1) = y0;

% Initializing n:
n=1;

while T(n) < tspan(2)
% Advancing time by the input time step:
    T(n+1) = T(n) + h ;
% Evaluating the solution at the next time step:
Y(n+1) = Y(n) + h* yprime(T(n),Y(n)) ;
% Incrementing n:
n=n+1;
end
```

3.1 Example

Implementation of Forward Euler

a) Solve the following ODE using your script and plot the solution along with the exact solution on the same plot.

$$y' = t^2 + y, \quad 0 \le t \le 2$$
$$y(0) = 1$$
$$h = 0.25$$
$$y_{exact} = 3e^t - t^2 - 2t - 2$$

b) Compute solutions to the ODE in the above example for $h = 0.5, 0.25, 0.1$, and 0.01 with all other parameters remaining the same. Plot these four solutions along with the exact analytical solution on the same graph. Adjust line styles using the figure menus.

Solution:

a) CODE from `yprime.m`

```
% ODE
function yprime = yprime(t,y)
yprime = t^2 + y;
```

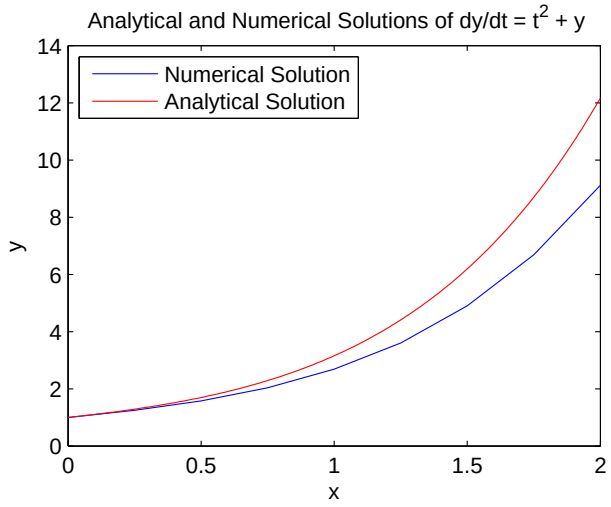
CODE from `yexact.m`

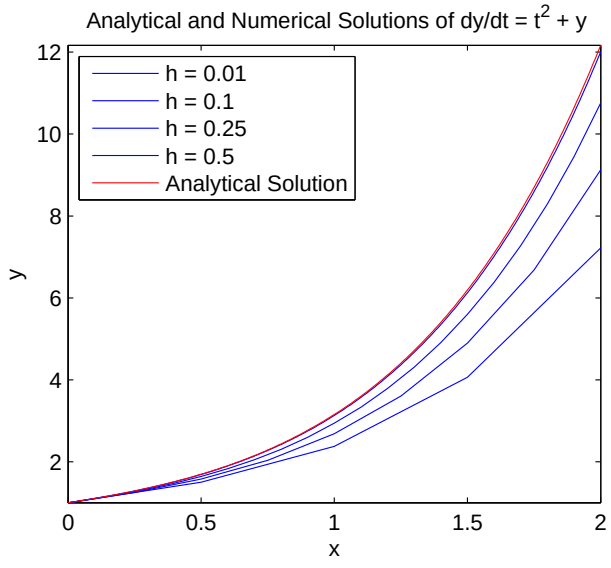
```
% Exact solution
function yexact = yexact(t)
yexact = 3*exp(t) - t.^2 - 2*t - 2;
```

CODE from M-file

```
[T,Y] = forwardEuler(@yprime, [0 2],0.25,1);
plot(T,Y)
hold on
% Computing and plotting the exact solution

t = 0 : 0.01 : 2;
plot(t, yexact(t),'r')
title('Analytical and Numerical Solutions of dy/dt = t^2 + y')
xlabel('x')
ylabel('y')
legend('Numerical Solution', 'Analytical Solution',...
'Location','NorthWest')
hold off
clear;
h = [0.01 0.1 0.25 0.5];
for i=1:4;
    [T,Y] = forwardEuler(@yprime,[0 2],h(i),1);
```

```
    plot(T,Y)
    hold on;
end

% Computing and plot the exact solution
t = 0 : 0.01 : 2;
plot(t,yexact(t),'r')
axis tight
title('Analytical and Numerical Solutions of dy/dt = t^2 + y')
xlabel('x')
ylabel('y')
hold off

legend('h = 0.01', 'h = 0.1', 'h = 0.25', 'h = 0.5',...
'Analytical Solution','Location','NorthWest')
```





Accuracy

As the results from the above example illustrate, numerical solutions introduces error, not only in each single step that we take but also across the entire interval. Thus, to measure the accuracy of the numerical method, we designate the single step error as the *local error* and the accumulated error over an interval as the *global error*. To calculate the local error at a given step, we simply calculate the absolute value of the difference between the exact solution and the numerical solution for that step. For example, for the ith step:

$$\text{Local (or) Single step error} \quad = \quad \left| y^{\text{exact}}(t_0 + ih) - y_i^{\text{numerical}} \right|.$$

Calculating the global error involves a bit more complication as to how to combine the effects of the single step errors. A widely used method is to calculate the Euclidean norm, which involves estimating the accumulation of error as follows:

$$\begin{aligned} \text{Global error} \quad &= \quad \sqrt{\frac{\sum_{i=1}^{n} \left| y^{\text{exact}}(t_0 + ih) - y_i^{\text{numerical}} \right|^2}{n}} \\ &= \quad \texttt{norm(y\_exact} - \texttt{y\_numerical)/sqrt(n)} \text{ in Matlab,} \end{aligned}$$

where `n = length(y_exact)`.

3.2 Example

a) Compute the global error in 8.1.1 a) using the Euclidean norm. Use the `length` command to calculate the number of time steps `n`.

b) Compute and display the global error for `h = 0.1, 0.01, 0.001, 0.0001`.

Solution:

a) CODE from M-file

```
clear;
%tspan = [tmin tmax]: interval over which solution has to be computed.
tspan = [0 2];
h = 0.25;
[T,Y] = forwardEuler(@yprime,tspan,h,1);

% Compute the number of time steps
n = length(Y);

% Calculate global error using Euclidean norm
ErrG_Euc = norm(yexact(T)-Y)/sqrt(n);
fprintf('Global Error: %.4f',ErrG_Euc);
```

Output

```
Global Error: 1.3321
```

b) CODE from M-file

```
clear;
tspan = [0 2];
h = [0.1 0.01 0.001 0.0001];
for j = 1:4
  [T,Y] = forwardEuler(@yprime,tspan,h(j),1);
  % Recompute number of time steps for each iteration
  n = length(Y);
  % Calculate global error using the Euclidean norm
  TimeStep = h(j);
  ErrG_Euc(j) = norm(yexact(T)-Y)/sqrt(n);
  Global_Error = ErrG_Euc(j);
  fprintf('TimeStep = %.4f, Global Error = %.4f\n',TimeStep, Global_Error);
end
```

Output

```
TimeStep = 0.1000, Global Error = 0.5670
TimeStep = 0.0100, Global Error = 0.0591
TimeStep = 0.0010, Global Error = 0.0059
TimeStep = 1.0001, Global Error = 0.0006:
```

Order of Accuracy

The accuracy that we calculated in the previous exercise applies only to the specific differential equation we solved. To describe the accuracy of the forward Euler method more generally, we estimate a quantity called the *order of accuracy* that describes the relationship between accuracy and time step. For example, a numerical scheme with a global accuracy of $O(h)$ has an order of accuracy 1. Such a scheme will generally produce a solution that is half as accurate when we double the step size. Likewise, when we halve the step size, the accuracy increases by a factor 2. Thus a higher order of accuracy is generally preferable, since only a small improvement in step size is required to effect a greater decrease in error.

Once we determine the local order of accuracy, the global order of accuracy is usually one order lower. For example, if the local order of accuracy is 2, then the global order of accuracy is 1. The global accuracy's lower order results from the fact that errors accumulate with each step. Let us look at this more precisely.

For our example before, the local error can be shown to be $O(h^2)$ using a Taylor's series expansion of the solution. The global error is the accumulation of local errors over n steps:

$$\begin{aligned} \text{Global error} &= n.O(h^2) \\ &= \left(\frac{t_f - t_0}{h}\right) O(h^2) \\ &= O(h). \end{aligned}$$

In general, if $n = \frac{t_f - t_0}{h}$ is the number of (time) steps and the local error at each step is $O(h^n)$, then:

$$\begin{aligned} \text{Global error} &= n.O(h^n) \\ &= \left(\frac{t_f - t_0}{h}\right) O(h^n) \\ &= O(h^{n-1}). \end{aligned}$$

Thus the global order of accuracy is $n - 1$.

Stability

As it results in an explicit equation, the forward Euler method has a finite stability interval, outside of which the solution diverges (blows up to ∞). To obtain an equation for the stability interval, we use the model differential equation:

$$y' = \lambda y$$

Applying the forward Euler scheme to the model equation, we get:

$$\begin{aligned} y_{n+1} &= y_n + h.y'_n \\ &= y_n + h.(\lambda y_n) \\ &= y_n (1 + h\lambda) \\ \Rightarrow y_n &= (1 + h\lambda)^n y_0 \end{aligned}$$

To prevent the numerical solution from blowing up, the factor $(1 + h\lambda)$ should not exceed unity; that is $|1 + h\lambda| \leq 1$. It can be seen that the numerical solution will quickly approach infinity once the absolute value of this factor exceeds unity.

3.3 Example

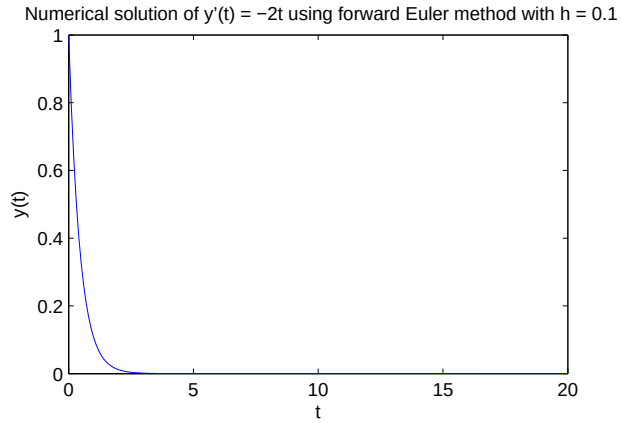
Let us consider the model equation:

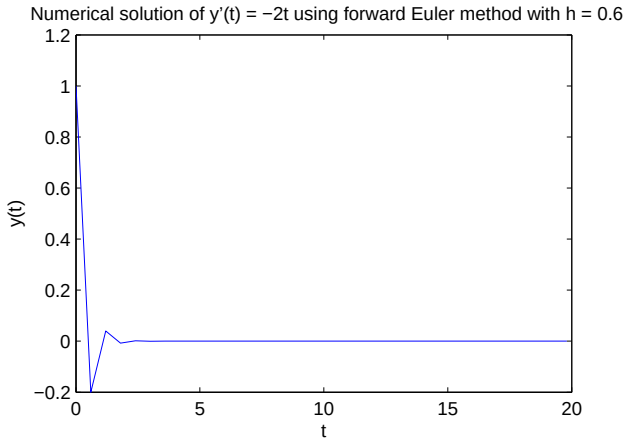
$$\frac{dy(t)}{dt} = -2y(t) \quad (\lambda = -2).$$

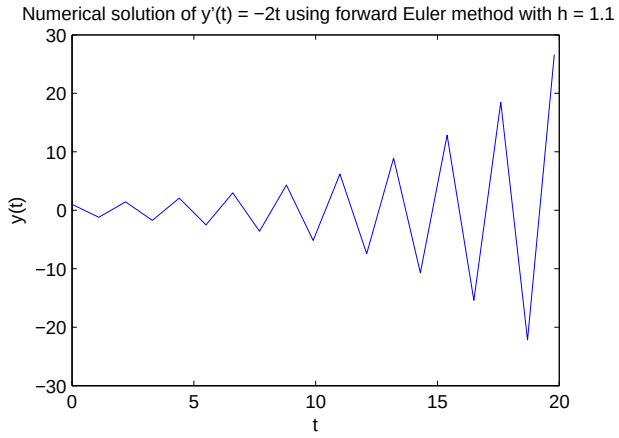
To determine the stability interval for the forward Euler method, we make use of the fact that $|1 + h\lambda| < 1$ for a decaying solution. This means:

$$|1 - 2h| < 1 \;\Rightarrow\; 0 < h < 1.$$

Given below are plots of the numerical solution for $h = 0.1,\ 0.6$ and 1.1 with $t = 0$ to 20 and $y(0) = 1$.







It can be clearly seen from the above plots that for $h > 1$, the numerical solution oscillates and diverges (is unbounded). This means the forward Euler scheme applied to this ordinary differential equation is not stable for $h > 1$. For both $h = 0.6$ and $h = 0.1$, the forward Euler scheme is stable. For $h = 0.6$, the numerical solution oscillates in an un-physical way but it decays as t increases. Therefore it is still stable.

3.4 Your Turn

In this problem you will compute a population model to determine how the future population will look based on the population in the first twenty months. The population at the beginning of each of the first twenty months appears in the table below.

Month	Population	Month	Population
0	10	10	513
1	18	11	560
2	29	12	595
3	47	13	629
4	71	14	641
5	119	15	651
6	174	16	656
7	257	17	660
8	350	18	662
9	441	19	662

a) The software package loaded on your spaceship specifies the following model for the population:

$$P(t) = \frac{K \cdot P_0}{P_0 + (K - P_0)e^{-rt}}$$

where K is the carrying capacity (maximum value) of the population, P_0 is the initial population and r is the reproductive rate. Based on experimental data, you can approximate these parameters of the model using the following:

- Carrying Capacity: $K = \lim_{t\to\infty} P(t)$.
- Initial Population: $P_0 = P(t = 0)$.
- Reproductive Rate: $r = \frac{1}{\Delta t} \cdot \ln \frac{P(2\Delta t)\cdot(P(\Delta t)-P_0)}{P_0(P(2\Delta t)-P(\Delta t))}$.

Note: The computed reproductive rate r is more accurate if points that are farther apart are used for computation(i.e. points which are separated by a large Δt).

Using these formulas, compute the parameters $K, P_0,$ and r for this model. Write down the best fit $P(t)$ using the computed values of the parameters. Evaluate the goodness of the fit by comparing the difference between the data and the model.

Plot the experimental data and the model on the same figure. In which month is the difference between the data and the model largest? How big is this difference?

b) The population model given above corresponds to the logistic growth model given by

$$P'(t) = r \cdot P(t) \cdot \left(1 - \frac{P(t)}{K}\right)$$

Solve the logistic growth model using Forward Euler with a time step of 0.1, and plot it along with the previous analytical model which was specified by the software.

c) Assuming that the model specified by the software in part a) is the analytical solution of the differential equation in part b), what is the Euclidean global error of your numerical logistic model obtained using Forward Euler? The expression for the Euclidean global error is given by:

$$\begin{aligned} \text{Global Error} &= \sqrt{\frac{\left(\sum_{i=1}^{n} |y_i^{\text{exact}} - y_i^{\text{numerical}}|^2\right)}{n}} \\ &= \frac{||y^{\text{exact}} - y^{\text{numerical}}||}{\sqrt{n}} \end{aligned}$$

where $||v||$ denotes the norm of the vector v.

4 Improved Euler

Improved Euler represents an improved version of Forward Euler not only because it is more accurate but also because it boasts a larger stability interval. At the expense of increased algorithmic complexity, each time step produces a closer solution to the exact result than Forward Euler. In this section, we will explore:

- The algorithm and code used to program Improved Euler
- Application of Improved Euler to solving differential equations
- Improved Euler's order of accuracy
- Improved Euler's stability limitations

Algorithm for Improved Euler

$$\boxed{\begin{array}{c} k_1 = f(t_n, y_n \\ k_2 = f(t_{n+1}, y_n + k) \\ y_{n+1} = y_n + h \cdot \frac{1}{2}(k_1 + k_2) \end{array}}$$

The Improved Euler algorithm is still explicit, as the next step is explicitly expressed in terms of values at the current step. However, unlike Forward Euler, the Improved Euler algorithm introduces the slopes at two different points.

Code for Improved Euler

As before, we pass in the function $f(t_n, y_n)$ so that we can solve any ordinary differential equation expressible in the explicit form $y'_n = f(t_n, y_n)$. Once we have programmed Improved Euler, we need alter only the function `yprime.m` when we change differential equations.
Use the following Improved Euler script to solve any first-order ODE given a domain and an initial condition (these should be input parameters). Name the file `ImprovedEuler.m`.

CODE from ImprovedEuler.m

```
function [T,Y] = ImprovedEuler(yprime,tspan,h,y0)

% function [T,Y] = ImprovedEuler(yprime,tspan,h,y0)
% tspan = [tmin tmax] : interval over which solution has to be computed
% h : time step
% y0 : initial condition
% --------------------------
% Output:
% T: sequence of time steps
% Y: values of y at each time step

T(1) = tspan(1);
Y(1) = y0;
n=1;

while T(n) < tspan(2)
```

```
    T(n+1) = T(n) + h;
    k1 = h * yprime(T(n),Y(n));
    k2 = h * yprime(T(n+1),Y(n)+k1);
    Y(n+1) = Y(n) + (1/2) * (k1+k2);
    n=n+1;
end
```

4.1 Example

Implementation of Improved Euler

a) Solve the following ODE using your script and plot the solution along with the exact solution on the same plot. On the same plot, graph the solutions for $h = 2, h = 4, h = 5$.

$$y' = e^{-y}, \quad y(0) = 0$$
$$t_0 = 0, \quad t_f = 20$$
$$h = 2, 4, 5$$
$$y_{\text{exact}} = \ln(t + 1)$$

Solution:

CODE from yprime.m

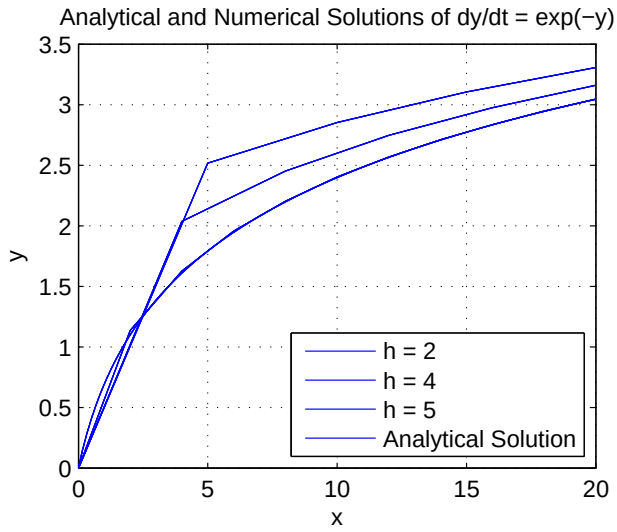
```
function yprime = yprime(t,y)
% dy/dt = exp(-y)
yprime = exp(-y);
```

CODE from yexact.m

```
% Exact solution
function yexact = yexact(t)
yexact = log(t+1);
```

CODE from M-file

```
clear;
h = [2 4 5];
for i=1:3;
 [T,Y] = ImprovedEuler(@yprime,[0 20],h(i),0);
    plot(T,Y)
    hold on;
end

% Computing and plotting the exact solution
t = 0:.01:20;
plot(t,yexact(t))
grid on
title('Analytical and Numerical Solutions of dy/dt = exp(-y)')
xlabel('x')
ylabel('y')
legend('h = 2', 'h = 4', 'h = 5','Analytical Solution')
```



b) Compute the global error and the local error of Improved Euler by solving the above equation with $h = 5$. Display both errors. Use `length` in calculating the number of time steps.
Solution:

CODE from M-file

```
clear;
% tspan = [tmin tmax] : interval over which solution has to be computed
tspan = [0 20];
h = 5;
[T,Y] = ImprovedEuler(@yprime,tspan,h,0);

% Compute the number of time steps
n = length(Y);
% Calculate global error using the Euclidean norm.
ErrG_Euc = norm(yexact(T) - Y)/sqrt(n);
% Calculate local error.
ErrL = abs( yexact( T(2) ) - Y(2) );

fprintf('\nGlobal Error: %g\n',ErrG_Euc);
fprintf('\nLocal Error : %g\n',ErrL);
```

Output

```
Global Error: 0.427327
Local Error : 0.725085
```

Order of Accuracy

The Improved Euler method represents an improvement in accuracy over the Forward Euler method. The Improved Euler method has a local order of accuracy of $O(h^3)$ and more importantly a global order of accuracy of $O(h^2)$. This is an order better than the Forward Euler method, which has a

global order of accuracy of $O(h)$. This improvement in accuracy is achieved through the introduction of intermediate calculations. The Improved Euler method is sometimes also referred to as the "Runge-Kutta 2" method for its second order global accuracy.

Stability

Just like the Forward Euler method, the Improved Euler method also results in an explicit equation. This results in a finite stability interval, outside of which the solution diverges (blows up to $\pm\infty$). To obtain an equation for the stability interval, we use the model differential equation:

$$y' = \lambda y.$$

Applying the Improved Euler scheme to the model equation, we get:

$$\begin{aligned}
k1 &= h.f(t_n, y_n) = h.(\lambda y_n), \\
k2 &= h.f(t_{n+1}, y_n + k1) \\
&= h\lambda.(y_n + k1) \\
&= h\lambda.(y_n + h\lambda y_n) = h\lambda y_n + h^2\lambda^2 y_n, \\
y_{n+1} &= y_n + \frac{1}{2}(k1 + k2) \\
&= y_n + \frac{1}{2}(h\lambda y_n + h\lambda y_n + h^2\lambda^2 y_n) \\
&= y_n + h\lambda y_n + \frac{h^2\lambda^2}{2} y_n \\
&= y_n \left(1 + h\lambda + \frac{h^2\lambda^2}{2}\right), \\
\Rightarrow y_n &= \left(1 + h\lambda + \frac{h^2\lambda^2}{2}\right)^n y_0.
\end{aligned}$$

To prevent the numerical solution from blowing up, the factor $\left(1 + h\lambda + \frac{h^2\lambda^2}{2}\right)$ should not exceed unity; that is:

$$\left|1 + h\lambda + \frac{h^2\lambda^2}{2}\right| \leq 1.$$

4.2 Example

c) Plot the stability intervals for the Forward Euler method and the Improved Euler methods.

$$\begin{aligned}
\text{Forward Euler method:} \quad & |1 + h\lambda| \leq 1 \\
\text{Improved Euler:} \quad & |1 + h\lambda + \frac{h^2\lambda^2}{2}| \leq 1
\end{aligned}$$

Note: The interior of the interval represents the stability interval.

Solution

c) The stability interval for Forward Euler method is given by

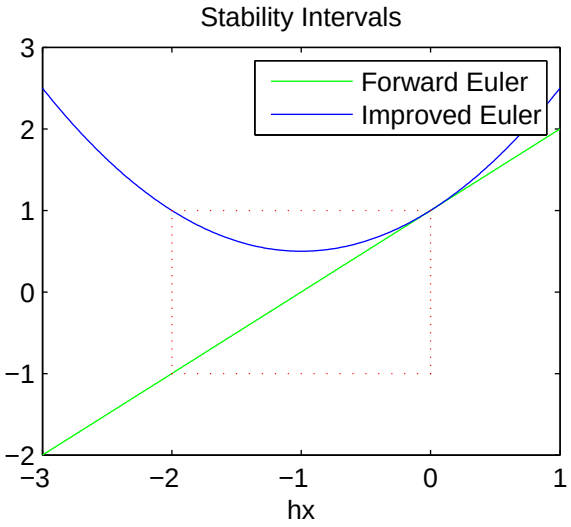
$$\begin{aligned} & |1 + h\lambda| \leq 1 \\ \implies & -1 \leq 1 + h\lambda \leq 1 \\ \implies & -2 \leq h\lambda \leq 0 \end{aligned}$$

The stability interval for Improved Euler is given by

$$\begin{aligned} & |1 + h\lambda + \frac{h^2\lambda^2}{2}| \leq 1 \\ \implies & -1 \leq \frac{2 + 2h\lambda + h^2\lambda^2}{2} \leq 1 \\ \implies & -1 \leq \frac{(1 + h\lambda)^2 + 1}{2} \leq 1 \\ \implies & -2 < 0 \leq (1 + h\lambda)^2 \leq 1 \\ \implies & -1 \leq (1 + h\lambda) \leq 1 \\ \implies & -2 \leq h\lambda \leq 0 \end{aligned}$$

CODE from M-file

```
clear;
x = -3:0.01:1;
% We assume x = h*lambda. Now x lies in the range (-2,0). Selecting a
% slightly larger range for x.
plot(x,1+x,'g',x,1+x+0.5*x^2,'b');
title('Stability Intervals);
xlabel('hx');
legend('Forward Euler method','Improved Euler');
hold on;
% Now we plot the boundaries of the stability interval
plot ([-2], [-1:0.1:1],'r:', [0],[-1:0.1:1],'r:', [-2:.1:0],[-1],'r:'...
,[-2:.1:0],[1],'r:');
```



4.3 Your Turn

In 1978 Ludwig *et al.* published a paper (*Journal of Animal Ecology* **47**: 315-332) detailing the interaction between the sprice budworm and the balsam fir forests in eastern Canada. They derived a first order nonlinear differential equation for the (non-dimensional) budworm population index μ and two constants R and Q:

$$\frac{d\mu}{dt} = R\mu\left(1 - \frac{\mu}{Q}\right) - \frac{\mu^2}{1+\mu^2},\ \mu > 0$$

The first term on the right hand side corresponds to a logistic model of population growth. The second term is a rate of population decay due to predation (mainly in the form of birds and parasites). R corresponds to the intrinsic growth rate of the organism (a budworm) and Q corresponds to the carrying capacity (largely determined by foliage)

a) Small μ limit: Using $\mu_0 = 1$, $R = 0.4$ and $Q = 15$, numerically solve the differential equation using the Improved Euler method with a time step of $h = 1$ for the interval $[0, 50]$. Plot the result. What is the limiting value of μ?

b) Large μ limit: Using $\mu_0 = 10$, $R = 0.4$ and $Q = 15$, numerically solve the differential equation using the Improved Euler method with a time step of $h = 1$ for the interval $[0, 50]$. Plot the result. What is the limiting value of μ?

c) Interpret the behaviour of these two different solutions in terms of a population model. Assume that predator growth is limited by other available resources, i.e. the predators effect on the budworm population (relative to the current size of the population) decreases as the budworm population increases.

5 Runge-Kutta 4 (RK4)

Runge-Kutta 4 extends the tactic seen in Improved Euler even further by introducing four intermediate slope calculations to determine each step with even greater accuracy and stability. While the method demands more calculation per time step, the resulting solution stays much closer to the exact solution because the intermediate calculations determine the net change more accurately. However, because Runge-Kutta 4 is still an explicit method like Forward Euler and Improved Euler, it, too, is only conditionally stable. In this section, we will explore:

- The algorithm and code used to program Runge-Kutta 4
- Application of Runge-Kutta 4 to solving differential equations
- Runge-Kutta 4's order of accuracy
- Runge-Kutta 4's stability limitations

Algorithm for Runge-Kutta 4

$$\begin{gathered} k_1 = h\, f\left(t_n, y_n\right) \\ k_2 = h\, f\left(t_n + \tfrac{h}{2}, y_n + \tfrac{k_1}{2}\right) \\ k_3 = h\, f\left(t_n + \tfrac{h}{2}, y_n + \tfrac{k_2}{2}\right) \\ k_4 = h\, f\left(t_n + h, y_n + k_3\right) \\ y_{n+1} = y_n + \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \end{gathered}$$

The Runge-Kutta 4 algorithm is still explicit, as the next step is explicitly expressed in terms of values at the current step. However, unlike the Euler methods, Runge-Kutta 4 takes several half-steps to gauge the slope between points before returning to make a full step. The final calculation uses a complex averaging of several derivatives.

Code for Runge-Kutta 4

As before, we pass in the function $f(t_n, y_n)$ so that we can solve any ordinary differential equation expressible in the explicit form $y'_n = f(t_n, y_n)$. Once we have programmed Runge-Kutta 4, we need alter only the function `yprime.m` when we change differential equations.
Use the following Runge-Kutta 4 script to solve any first-order ODE given a domain and an initial condition (these should be input parameters). Name the file `RK4.m`.

CODE from RK4.m

```
function [T,Y] = RK4(yprime,tspan,h,y0)
% function [T,Y] = RK4(yprime,tspan,h,y0)
% tspan = [tmin tmax] : interval over which solution has to be computed
% h : time step
% y0 : initial condition
% --------------------------
% Output:
% T: sequence of time steps
% Y: values of y at each time step

T(1) = tspan(1);
Y(1) = y0;
n=1;

while T(n) < tspan(2)
    T(n+1) = T(n) + h;
    k1 = h * yprime(T(n),Y(n));
    k2 = h * yprime(T(n)+h/2,Y(n)+(1/2)*k1);
    k3 = h * yprime(T(n)+h/2,Y(n)+(1/2)*k2);
    k4 = h * yprime(T(n)+h,Y(n)+k3);
    Y(n+1) = Y(n) + (1/6) * (k1+2*k2+2*k3+k4);
    n=n+1;
end
```

5.1 Code Example

Implementation of Runge-Kutta 4 method

a) Solve the following ODE using the Runge-Kutta 4 method and plot the solutions for `h` = 0.5

and 0.05 along with the exact solution on the same plot.

$$
\begin{aligned}
y' &= t(y+2), \quad y(0) = 0, \\
t_0 &= 0, \quad t_f = 20, \\
h &= 0.5,\ 0.05, \\
y_{exact} &= 3e^{\frac{t^2}{2}} - 2.
\end{aligned}
$$

Solution:

a) **CODE from yprime.m**

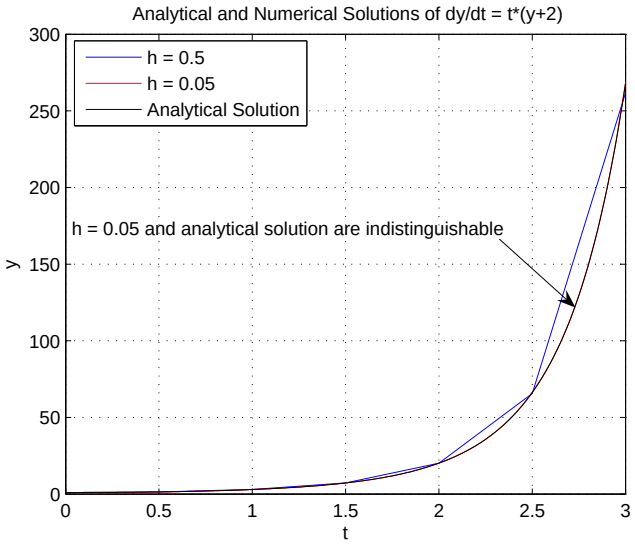
```
% ODE
function yprime = yprime(t,y)
% dy/dt = t*(y+2)
yprime = t*(y+2);
```

CODE from yexact.m

```
% Exact solution
function yexact = yexact(t)
yexact = 3*exp(t.^2/2) - 2;
```

CODE from M-file

```
clear;
h = [0.5 0.05];
[T,Y] = RK4(@yprime,[0 3],0.5,1);
plot(T,Y,'b');
[T,Y] = RK4(@yprime,[0 3],0.05,1);
hold on;
plot(T,Y,'r');

% Computing and plotting the exact solution:
t = 0 : 0.01 : 3;
plot(t,yexact(t),'k');
grid on;
title('Analytical and Numerical Solutions of dy/dt = t*(y+2)')
xlabel('t');
ylabel('y');
legend('h = 0.5', 'h = 0.05', 'Analytical Solution');
```



Order of Accuracy

Runge-Kutta 4 (RK4) represents an improvement in accuracy over both the Forward and Improved Euler methods because of its relatively better approximation of the function slope during every time step. It has a local accuracy that is $O(h^5)$ and therefore a global accuracy that is $O(h^4)$. This means, by dividing the time step by 10, we can effectively increase the global accuracy by a factor of 10^4. In fact the fourth order global accuracy gives RK4 its name. Even though it requires more calculations at each time step than the Forward and Improved Euler methods, the end result boasts a great improvement in terms of accuracy. The following exercises will demonstrate the improved accuracy of RK4.

5.2 Accuracy Example

b) Compute and display the global error in Example (a) for time steps `h = 0.001, 0.01` and `0.1`. Create plots of global error vs. `h` using logarithmic scale for both the axes of your plot. You can do this using the `loglog` command or the figure menus. Compute the slope of the line in your plot. You may use the `length` command to calculate the number of time steps. The slope you determine for the global error should coincide with the order of the RK4 method, proving that RK4 is fourth-order accurate.

Solution:

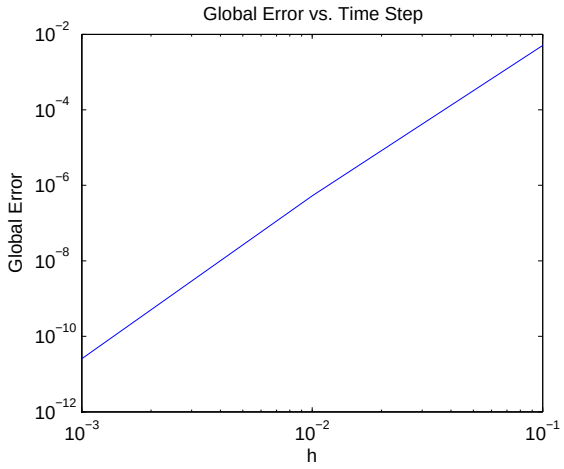
b) CODE from M-file

```
clear;
% tspan = [tmin tmax] :interval over which solution has to be computed
tspan = [0 3];
h = [0.001 0.01 0.1];

for j = 1:3
  [T,Y] = RK4(@yprime,tspan,h(j),1);
```

```
  % Calculating global error using the Euclidean norm.
  % Recompute number of time steps for each iteration
  n = length(Y);
  ErrG_Euc(j) = norm(yexact(T)-Y)/sqrt(n);
  fprintf('Time Step h = %5g, Global Error = %10g\n',h(j),ErrG_Euc(j));
end

loglog(h,ErrG_Euc);
xlabel('h');
ylabel('Global Error');
title('Global Error vs. Time Step');

% Slope calculation:
Global_Error_Slope = (log(ErrG_Euc(3))-log(ErrG_Euc(2))) /(log(h(3))-log(h(2)));
fprintf('Global error slope = %g\n',Global_Error_Slope);
```

Output

```
Time Step h = 0.001, Global Error = 2.56081e-011
Time Step h =  0.01, Global Error = 5.26977e-007
Time Step h =   0.1, Global Error = 0.00506376
Global error slope = 3.98268
```



Stability

Since it is explicit, RK4 has a finite stability region. The extra expense involved in the improved slope calculations widens the stability interval of RK4 in comparison to that of the Forward and Improved Euler methods. To obtain an equation for this stability interval, let us use the following model differential equation:

$$y' = \lambda y.$$

Applying the RK4 scheme to the model equation, we calculate the four slopes as follows:

$$k1 = h\, f(t_n, y_n)$$

$$
\begin{aligned}
&= h\,(\lambda y_n),\\
k2 &= h\lambda\left(y_n + \frac{1}{2}\,k1\right)\\
&= h\lambda\left(y_n + \frac{1}{2}h\lambda y_n\right)\\
&= h\lambda y_n + \frac{1}{2}h^2\lambda^2 y_n,\\
k3 &= h\,f\left(t_n + \frac{h}{2}, y_n + \frac{1}{2}\,k2\right)\\
&= h\lambda\left(y_n + \frac{1}{2}\,k2\right)\\
&= h\lambda\left(y_n + \frac{1}{2}\left(h\lambda y_n + \frac{1}{2}h^2\lambda^2 y_n\right)\right)\\
&= h\lambda\left(y_n + \frac{1}{2}h\lambda y_n + \frac{1}{4}h^2\lambda^2 y_n\right)\\
&= h\lambda y_n + \frac{1}{2}h^2\lambda^2 y_n + \frac{1}{4}h^3\lambda^3 y_n,\\
k4 &= h\,f(t_n + h, y_n + k3)\\
&= h\,\lambda\,(y_n + k3)\\
&= h\lambda\left(y_n + h\lambda y_n + \frac{1}{2}h^2\lambda^2 y_n + \frac{1}{4}h^3\lambda^3 y_n\right)\\
&= h\lambda y_n + h^2\lambda^2 y_n + \frac{1}{2}h^3\lambda^3 y_n + \frac{1}{4}h^4\lambda^4 y_n.
\end{aligned}
$$

With these new slopes we calculate the new value of y as:

$$y_{n+1} = y_n + \frac{1}{6}(k1 + 2\,k2 + 2\,k3 + k4)$$

After simplification and substitution, we arrive at the following expression:

$$y_{n+1} = \left(1 + h\lambda + \frac{1}{2}h^2\lambda^2 + \frac{1}{6}h^3\lambda^3 + \frac{1}{24}h^4\lambda^4\right)\,y_n.$$

To prevent the numerical solution from blowing up to infinity, we require that:

$$\left|1 + h\lambda + \frac{1}{2}h^2\lambda^2 + \frac{1}{6}h^3\lambda^3 + \frac{1}{24}h^4\lambda^4\right| \leq 1.$$

5.3 Stability Example

c) Based on the following inequalities, plot the stability intervals for the RK4, Forward and Improved Euler methods:

$$\text{Forward Euler method} : |1 + h\lambda| \leq 1,$$

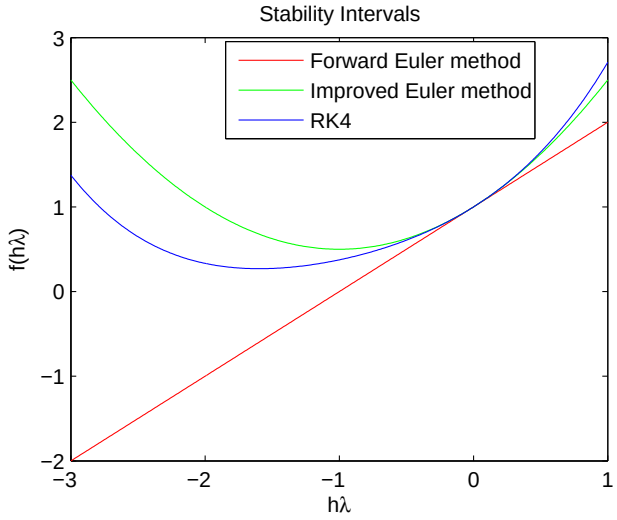
$$\text{Improved Euler} : \left|1 + h\lambda + \frac{h^2\lambda^2}{2}\right| \leq 1,$$

$$\text{RK4} : \left|1 + h\lambda + \frac{1}{2}h^2\lambda^2 + \frac{1}{6}h^3\lambda^3 + \frac{1}{24}h^4\lambda^4\right| \leq 1.$$

Solution

c) CODE from M-file

```
clear;
x = -3:0.01:1;
% We assume x = h*lambda and select a large range for x to find the
% stability interval.
plot(x,1+x,'r',x,1+x+(1/2)*x.^2,'g',x,1+x+(1/2)*x.^2+(1/6)*x.^3+(1/24)*x.^4,'b');
title('Stability Intervals');
xlabel('h\lambda');
ylabel('f(h\lambda)');
legend('Forward Euler method','Improved Euler method','RK4');
```



It can be seen from the above plot that the stability interval for:

- Forward and Improved Euler methods is: $-2 \leq h\lambda \leq 0$
- RK4 method is: $-2.8 \leq h\lambda \leq 0$

RK4, because of its bigger stability interval is more stable than the Forward and Improved Euler methods.

5.4 Your Turn

Air resistance or drag force is proportional to the square of the instantaneous speed at high velocities. Thus the speed v of a mass m is governed by the equation

$$m\frac{dv}{dt} = mg - kv^2$$

Suppose you seek to drop a 5-slug (a slug is a measure of mass) ballistic missile from the cargo hamper of a jet plane. Atmospheric conditions and the missile's head shape dictate that $k = 0.125$. Assume $g = 32$ ft/sec^2.

a) Use Forwared Euler and Runge-Kutta 4 with a time step of $h = 1$ to approximate the speed of the falling mass at $t = 5$. Graph the solutions on the same plot.

b) Use `dsolve` to determine the exact solution, and add it to your plot. Which numerical solution provides the best approximation?

c) Can you determine the terminal speed (*i.e.* the speed reached after a long time) from the plot? How long does it take to reach this speed?

d) Find an analytical expression for the terminal speed in terms of m, g and k. Does it agree with your answer for part c)?

6 Built-In ODE Solvers and Systems of ODEs

Commands:

`@`	Establishes a function handle that allows for a function to be passed as a parameter.
`ode45`	Explicit non-stiff solver, based on Runge-Kutta 4.
`ode113`	Explicit stringent high-accuracy solver.

Input Parameters

To call `ode45` or `ode113`, the following lines must be typed at the command line or in a script.

```
>> [t,y] = ode45(@yprime,tspan,y0,options)
>> [t,y] = ode113(@yprime,tspan,y0,options)
```

These are input parameters that are given to `ode45` or `ode113`.

`@yprime`	Handle of the function defining the ODE to be solved - yprime *must* output a column vector!
`tspan`	Row vector containing the initial and final time across which we must solve the ODE
`y0`	Row vector containing all necessary initial conditions $y(0)$, $y'(0)$, $y''(0)\ldots$, or simply $y(0)$ for 1st order ODEs.
`options`	Optional values specifying error tolerance where beforehand we set `options = odeset('reltol',reltol,'abstol',abstol)`. The inputs `reltol` and `abstol` are scalar values for the relative tolerance and absolute tolerance, respectively.

Introduction

When we write a numerical ODE solver, it should be able to solve any differential equation that we can define with a function $y' = f(x, y)$. Function handles allow us to input any differential

equation `@yprime` into our solver as a variable without changing the script or code of our solver. Without function handles, we would have to create separate solvers for every differential equation we encounter, since we would be unable to treat the function `yprime` as a variable in our solver. Thus, when we type an input with a handle (`@yprime`), we essentially pass a function into a program, which then uses the function as a variable.
Matlab's built-in ODE solvers function no differently. In order to solve a differential equation numerically, we must write a function `yprime` to represent the ordinary differential equation we wish to solve; this function takes the form $y' = f(x, y)$. Once we have written our differential equation in the function `yprime`, we simply pass that function into the ODE solver with a function handle in front.
For example, suppose we want to solve the differential equation

$$y'(t) + y(t) - 2t - t^2 = 0.$$

First, we write the function `yprime` by isolating the derivative $y'(t)$:

$$y'(t) = t^2 + 2t - y(t)$$

Next, we enter Matlab and write a short and simple function (.m file) for this differential equation:

```
function yprime = yprime(t,y)
yprime = t^2 + 2*t - y;
```

To solve this equation from $t = a$ to $t = b$, where $y(a) = y0$, we simple pass the function into a solver with the function handle in front:

```
>> [t,y] = ode45(@yprime, [a b], y0);
```

If our desired interval of solution is $t = 3$ to $t = 8$, where $y(3) = 1$, then we would type:

```
>> [t,y] = ode45(@yprime, [3 8], 1);
```

In response, Matlab will solve the equation and store the t-interval as a column vector and the solution y as the second column vector. Thus, we type `[t,y]` to denote these outputs.
Note that we never needed to change the solver (`ode45`) itself. When we wish to change the differential equation that we want to solve, we simply change the function that we pass into the solver. We could even create a new file titled `yprime2` or `derivative` or `dfdt` or whatever we wished, so that we can keep `yprime` intact. Thus, function handles allow us to write ODE solvers that function independently of the input function.

Accuracy Settings

The two parameters `reltol` and `abstol` are used to control the accuracy. Matlab adaptively chooses the time step such that the error at step n satisfies

$$e_n \leq \max\{r|y_n|, a\},$$

where r is the relative tolerance and a is the absolute tolerance. Matlab bounds the error with either an absolute tolerance or the product of relative tolerance and $|y_n|$. The accuracy constraint depends on the larger of the two: a categorical, scale-independent error known as 'AbsTol,' or an error scale relative to the solution known as 'RelTol.' Matlab will determine the larger of the two. To change the relative and absolute tolerance, we use the odeset function:

```
opts = odeset('reltol',1e-12,'abstol',1e-12);
```

By setting both tolerance levels to 10^{-12}, we force the solver to use a small time step to preserve our desired accuracy (or error tolerance). Thus, by changing the tolerance levels (or margins of error), we can control the size of the time step that `ode45` takes. Once we have defined `opts`, we can then pass it into the solver as the fourth input parameter. `ode45` does not work well for very stringent accuracies.
For very accurate solutions `ode113` is preferable. For example if we set an accuracy of 10^{-12}, `ode45` takes around 9s with 100,000 evaluations of `yprime` whereas `ode113` takes 1s with only 10,000 evaluations of `yprime`.

Solving a system of First Order ODEs

Assume we want to solve the system of equations

$$\begin{aligned} \frac{dR}{dt} &= R(t) - .5F(t)R(t) \\ \frac{dF}{dt} &= -F(t) + .25F(t)R(t) \end{aligned}$$

This is a simplified version of the predator prey model proposed in 1925 by Lotka and Volterra. $R(t)$ is the prey (or rabbit) and $F(t)$ is the predator (or fox). The $F(t)R(t)$ terms in the two equations model the interaction of the two populations. The number of encounters between predator and prey is assumed to be proportional to the product of the populations. Since any such encounter tends to be good for the predator and bad for the prey, the sign of the $F(t)R(t)$ term is negative in the first equation and positive in the second.
To solve this system using Matlab, you first need to define a function taking as input the time `t` and a vector `y` containing R and F:

```
function yp = predator_prey(t,y)
% y(1) is the prey R
% y(2) is the predator F
yp = [y(1) - 0.5*y(1)*y(2); -0.75*y(2) + 0.25*y(1)*y(2)];
```

In this function the first entry in `y` contains R and the second contains F. The function returns the variable `yp`. Its first entry is $\frac{dR}{dt}$ and its second entry is $\frac{dF}{dt}$. The system can now be easily solved by typing

```
>> [t,y] = ode45(@predator_prey,[0 30],[2 1]);
```

To plot the solution, use the command

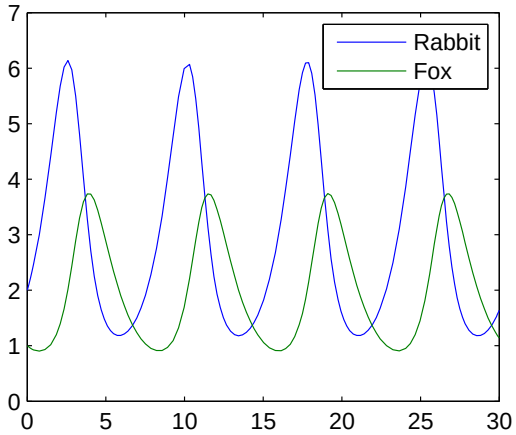
```
plot(t,y(:,1),t,y(:,2))
legend('Rabbit','Fox')
```

The resultant matrix `y` has two columns. The first column of `y` contains the solution $R(t)$ and the second $F(t)$. The number of rows is clearly equal to the number of time steps. The solution looks like this:

6.1 Code Example

a) Write a Matlab function to solve the following differential equation using `ode45`

$$y_1'(t) = y_2,$$



$$y_2'(t) = -y_1.$$

Use as initial conditions $y_1(0) = 0,\ y_2(0) = 1$. Plot your solution for $0 \le t \le 4\pi$.

b) Write a Matlab code to solve, using a nested function, the following ODE:

$$y_1'(t) = -y_1 + y_2,$$
$$y_2'(t) = y_1 - a y_2.$$

where a is set by the user. Use as initial conditions $y_1(0) = 0,\ y_2(0) = 1$. Set $a = 0.9$ and solve for $0 \le t \le 4\pi$. Plot the solution.

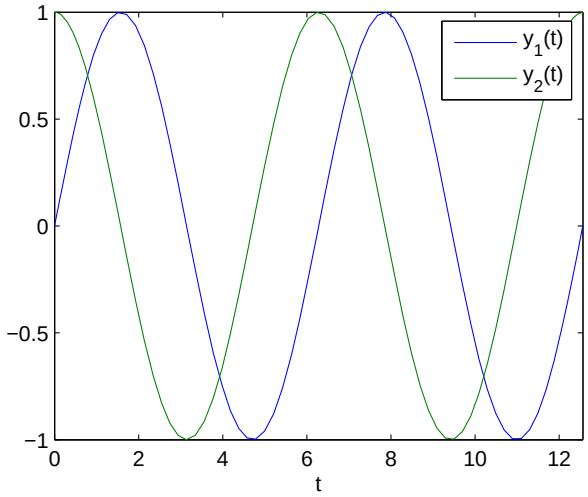
c) Solve the following ODE using `ode45` with relative and absolute error of 10^{-12}:

$$y'(t) = 8\cos(t) - 3y^3,$$
$$y(0) = 1.$$

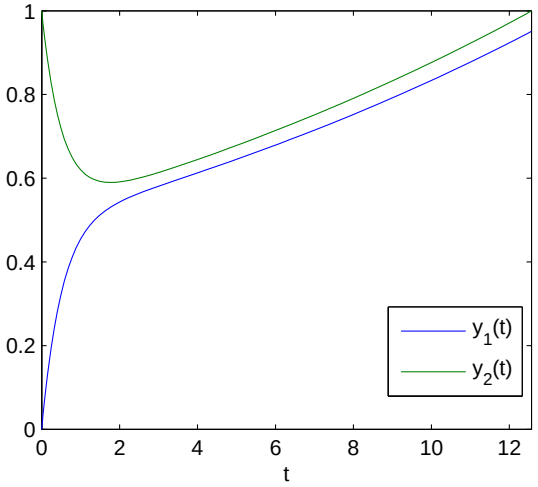
The initial time is $t = 0$, and the end time is $t = 10$. Plot the solution.

Solution:

a)
```
function exa
y0=[0 1];
tspan=[0 4*pi];
[t,y] = ode45(@yprime,tspan,y0);
plot(t,y(:,1),t,y(:,2));
xlabel('t');
legend('y_1(t)','y_2(t)');
end
function yp = yprime(t,y)
yp = [y(2);-y(1)];
end
```

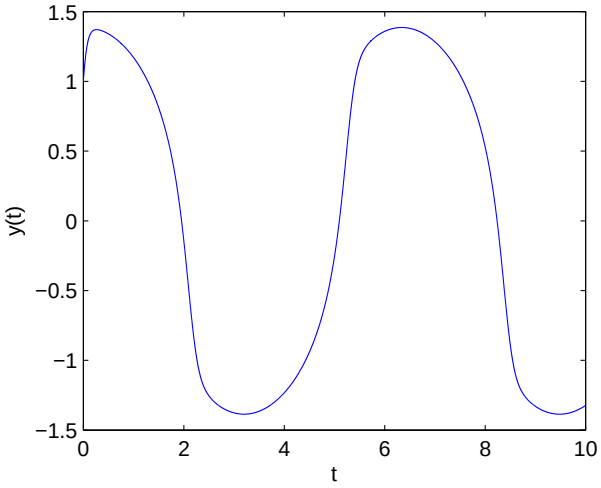


b)
```
function exb
y0=[0 1];
tspan=[0 4*pi];
a = 0.9;
[t,y] = ode45(@yprime,tspan,y0);
plot(t,y(:,1),t,y(:,2));
xlabel('t');
legend('y_1(t)','y_2(t)');
    function yp = yprime(t,y)
        yp = [-y(1)+y(2); y(1)-a*y(2)];
    end
end
```



c)
```
function exc
y0=1;
tspan=[0 10];
```

```
options=odeset('AbsTol',1e-12,'RelTol',1e-12) ;
[t,y] = ode45(@yprime,tspan,y0,options);
plot(t,y);
xlabel('t');
ylabel('y(t)');
end
function yp = yprime(t,y)
yp = 8*cos(t)-3*y.^3;
end
```



6.2 Your Turn

The problem in this section is concerned with a system of first Order ODEs. Here, we will study the effect of burning fossil fuels on the carbon dioxide content in the earth's atmosphere. We present a mathematical model that simulates the interaction of the various forms of carbon that are stored in three regimes: the atmosphere, the shallow ocean, and the deep ocean. Our main concern here will be to understand how to solve the model numerically and to draw meaningful conclusions from our solution.

The five principal variables in the model are all functions of time:

p	partial pressure of carbon dioxide in the atmosphere,
σ_s	total dissolved carbon concentration in the shallow ocean,
σ_d	total dissolved carbon concentration in the deep ocean,
α_s	alkalinity in the shallow ocean,
α_d	alkalinity in the deep ocean.

The exchange between the atmosphere and the shallow ocean involves a constant characteristic transfer time d and a source term $f(t)$ which models the pollution due to human activities.

The rate of change of the principal variables is given by the following ordinary differential equations:

$$\frac{dp}{dt} = \frac{p_s - p}{d} + \frac{f(t)}{\mu_1},$$

$$\begin{aligned}
\frac{d\sigma_s}{dt} &= \frac{1}{\nu_s}\left((\sigma_d - \sigma_s)\, w - k_1 - \frac{p_s - p}{d}\mu_2\right), \\
\frac{d\sigma_d}{dt} &= \frac{1}{\nu_d}\left(k_1 - (\sigma_d - \sigma_s)\, w\right), \\
\frac{d\alpha_s}{dt} &= \frac{1}{\nu_s}\left((\alpha_d - \alpha_s)\, w - k_2\right), \\
\frac{d\alpha_d}{dt} &= \frac{1}{\nu_d}\left(k_2 - (\alpha_d - \alpha_s)\, w\right).
\end{aligned}$$

where ν_s and ν_d are the volumes of the shallow and deep oceans respectively.

The following three additional quantities are involved in equilibrium equations in the shallow ocean:

- h_s hydrogen carbonate concentration in the shallow ocean,
- c_s carbonate concentration in the shallow ocean,
- p_s partial pressure of gaseous carbon dioxide in the shallow ocean.

The equilibrium between carbon dioxide and the carbonates dissolved in the shallow ocean is described by the following three nonlinear algebraic equations:

$$\begin{aligned}
h_s &= \frac{\sigma_s - \left(\sigma_s^2 - k_3\alpha_s\,(2\sigma_s - \alpha_s)\right)^{\frac{1}{2}}}{k_3}, \\
c_s &= \frac{\alpha_s - h_s}{2}, \\
p_s &= k_4\frac{h_s^2}{c_s}.
\end{aligned}$$

The constants involved in the above equations are:

$$\begin{array}{lcllcl}
d & = & 8.64 & w & = & 10^{-3} \\
\mu_1 & = & 4.95 \times 10^{2} & k_1 & = & 2.19 \times 10^{-4} \\
\mu_2 & = & 4.95 \times 10^{-2} & k_2 & = & 6.12 \times 10^{-5} \\
\nu_s & = & 0.12 & k_3 & = & 0.997148 \\
\nu_d & = & 1.23 & k_4 & = & 6.79 \times 10^{-2}
\end{array}$$

We will mainly be interested in studying how the levels of carbon dioxide in the atmosphere (p) and carbon concentrations in the shallow (σ_s) and deep ocean (σ_d) vary with time.

a) Type the following line in the command window:

```
>> load SourceTermTable.mat
```

This opens the file `SourceTermTable.mat` and creates two arrays `tdata` and `fdata` in your workspace. Plot the data with

```
>> plot(tdata,fdata)
```

Here `fdata` indicates the level of pollution $f(t)$ released in the atmosphere as a function of time.

b) The file `SourceTerm.m` includes the function `SourceTerm` which can be used to get the value of $f(t)$ at any given t (where t is in units of years). We will first verify that the above function gives the desired output. Plot $f(t)$ as follows. Verify that the plot you obtain is almost the same as in part a) but much smoother.

```
>> t = linspace(1000,5000,10000);
>> plot(t,SourceTerm(tdata,fdata,t))
```

Verify that $f(t) = 5.3852$ at $t = 2007$ (the current year!). Evaluate $f(t)$ at $t = 2200$ and $t = 2400$. When does the pollution function $f(t)$ peak?

c) Write a function called `carbonODE` which defines the ODE to be solved. This function will use the variables `tdata` and `fdata`. The variable `y` in `carbonODE` should contain the following variables in that exact order: $p, \sigma_s, \sigma_d, \alpha_s, \alpha_d$ The technique of nested functions must be used. The function `carbonODE` needs to be defined inside the function `carbon`. Follow this template:

```
function carbon(tdata,fdata)

format long
carbonODE(1000,[1 2.01 2.23 2.20 2.26])
    function yp = carbonODE(t,y)
        f = SourceTerm(tdata,fdata,t);
        % Write the content of carbonODE here
    end
end
```

You should obtain the following output:

```
carbon(tdata,fdata)
ans =
   0.010264729371666
  -0.004225867532479
  -0.000000813008130
  -0.000010000000000
   0.000000975609756
```

d) Solve the ODE using `ode45` by passing `carbonODE` as a function handle. You need to solve from $t = 1000$ to $t = 5000$ with the following initial values at $t = 1000$:

$$\begin{aligned} p &= 1.0 \\ \sigma_s &= 2.01 \\ \sigma_d &= 2.23 \\ \alpha_s &= 2.20 \\ \alpha_d &= 2.26 \end{aligned}$$

Plot p in one figure, σ_s and σ_d together in another figure.

e) From the graph find out the year when the concentration of CO_2 peaks for, the release from pollution (the function $f(t)$), the atmosphere, the shallow ocean.

f) Explain the observed changes in carbon concentration. Does it make sense? Why is the deep ocean important for the global carbon cycle?

g) Suppose there is an industrialist who claims that burning fossil fuels does not result in the increase of carbon levels in the environment. He supports his claim by saying that even though year 2004 and 2005 witnessed highest levels of fossil fuel consumption the change in carbon levels observed in the environment was very low. How would you use the results from our model to refute his claim?

h) Lets assume that the president calls you and asks you some advice on CO_2 emission caps to prevent global warming. You know that the CO_2 in the atmosphere should stay below 4 in order to avoid major problems such as the rise of the sea level. You want to decide on a policy to cap emissions between now and 2150. You can cap emissions, say at 10, in that time range with the command:

```
fdata(6:10) = 10;
```

inside the function `carbon`. Find a value to cap emissions (i.e. adjust the number 10 above) such that the CO_2 concentration stays below 4.

7 Systems of Second Order Ordinary Differential Equations

Solving Second Order ODEs

In this section we learn how to solve Second Order ODEs by reducing them to a system of coupled first order ODEs. Consider the Second Order ODE:

$$my''(t) + \gamma y'(t) + Ky(t) = F_0 \cos(\omega t)$$

Let us introduce two auxiliary functions $y_1(t)$ and $y_2(t)$ such that

$$y_1(t) = y(t) \text{ and } y_2(t) = y'(t).$$

Therefore,

$$y_2'(t) = y''(t).$$

Hence we can write our Second Order ODE as

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \frac{1}{m}\big(F_0 \cos(\omega t) - \gamma y_2 - K y_1\big). \end{aligned}$$

Solving the above set of first order ODEs will give us the solution to our Second Order ODE. Let us define our Second Order ODE with a function myode:

```
function yp = myode(t,y)
    yp = [0;0];
    yp(1) = y(2);
    yp(2) = (F0*cos(w*t) - gamma*y(2) - k*y(1))/m;
end
```

The `myode` function takes two input arguments, scalar `t` (time) and column vector `y` (state) and returns the output argument `yp`, a column vector of state derivatives. Note that other variables such as `F0`, `w`, `gamma`, `k`, and `m` appear in this function. The technique of nested functions needs to be used to make this work. We call `ode45` to solve our Second Order ODE defined by `myode` from $t = a$ to $t = b$, where the initial conditions $y(a) = y_0$ and $y'(a) = y'_0$ are given as the first and second entries of `[y0 yp0]`, respectively.

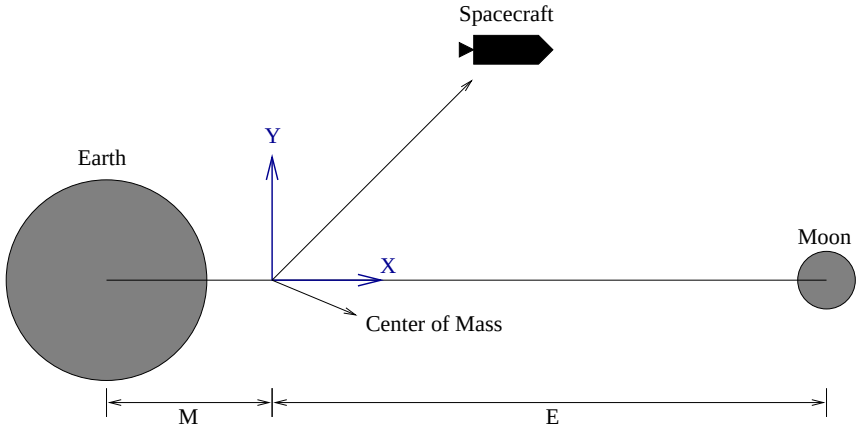
```
[t,y] = ode45(@myode,[a b],[y0 yp0]);
```

The first column of `y` contains our solution. The second column contains $y'(t)$. With this approach, it is always possible to express a system of nonlinear differential equations as a set of first order differential equations.

Solving a System of Second Order ODEs

In this section we will learn how to reduce a system of Second Order ODEs to a system of First Order ODEs and solve it the same way we have done before.

To do this, let us consider an example of the motion of a space probe in the gravitational field of two bodies (such as the earth and the moon). The equations governing the motion of the spacecraft is a system of second order differential equations.



We have a system as shown above. Both bodies impose a force on the spacecraft according to the gravitational law, but the mass of the spacecraft is too small to significantly affect the motion of the bodies. We therefore neglect the influence of the spacecraft on the two stellar bodies. Our co-ordinate system has its origin at the center of mass of earth and moon. The governing equations are given by

$$\begin{aligned} \frac{d^2x}{dt^2} &= 2\frac{dy}{dt} + x - \frac{E(x+M)}{r_1^3} - \frac{M(x-E)}{r_2^3} \\ \frac{d^2y}{dt^2} &= -2\frac{dx}{dt} + y - \frac{Ey}{r_1^3} - \frac{My}{r_2^3} \end{aligned}$$

where $r_1 = \sqrt{(x+M)^2 + y^2}$, $r_2 = \sqrt{(x_E)^2 + y^2}$, and $E = 1 - M$.

The earth and moon are assumed to have circular orbits around the center of mass of the system. To simplify the problem, we therefore consider a coordinate system which is rotating with the earth

and the moon. In that system, the earth does not move and is located at $(-M, 0)$ and the moon is at $(E, 0)$. The governing equations contain terms in $\frac{1}{r^2}$, which correspond to the gravitational force. However, the equations also contain terms $\frac{dy}{dt}$ and $-\frac{dx}{dt}$, which correspond to the Coriolis force, as well as terms x and y, which correspond to the Centrifugal force.

We first need to reduce the system of second-order differential equations into a first-order system of four equations. To do this, we define

$$z_1 = x,\ z_2 = \frac{dx}{dt},\ z_3 = y,\ \text{and}\ z_4 = \frac{dy}{dt}.$$

In vector form, we get

$$Z = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = \begin{bmatrix} x \\ x' \\ y \\ y' \end{bmatrix}.$$

Therefore,

$$Z' = \begin{bmatrix} z_1' \\ z_2' \\ z_3' \\ z_4' \end{bmatrix} = \begin{bmatrix} x' \\ x'' \\ y' \\ y'' \end{bmatrix} = \begin{bmatrix} z_2 \\ 2z_4 + z_1 - \frac{E(z_1+M)}{r_1^3} - \frac{M(E-z_1)}{r_2^3} \\ z_4 \\ 2z_2 + z_3 - \frac{Ez_3}{r_1^3} - \frac{M-z_3}{r_2^3} \end{bmatrix}$$

where $r_1 = \sqrt{(z_1+M)^2 + z_3^2}$ and $r_2 = \sqrt{(z_1-E)^2 + z_3^2}$. Now we can easily write the function for defining our ODE as follows:

```
function zp = ZPrime(t,z)

zp = [0 0 0 0]';

% We define new variables z1 through z4 to simplify the expressions later on
z1 = z(1);
z2 = z(2);
z3 = z(3);
z4 = z(4);

R1 = sqrt((z1+M)^2 + (z3)^2);
R2 = sqrt((z1-E)^2 + (z3)^2);

zp(1) = z2;
zp(2) = 2*z4 + z1 - E*(z1+M)/R1^3 - M*(z1-E)/R2^3;
zp(3) = z4;
zp(4) = -2*z2 + z3 - E*z3/R1^3 - M*z3/R2^3;
end
```

7.1 Code Example

a) Write a Matlab program to solve the example of spacecraft motion from $t = 0$ to $t = 24$ for the following initial data and plot your solution:

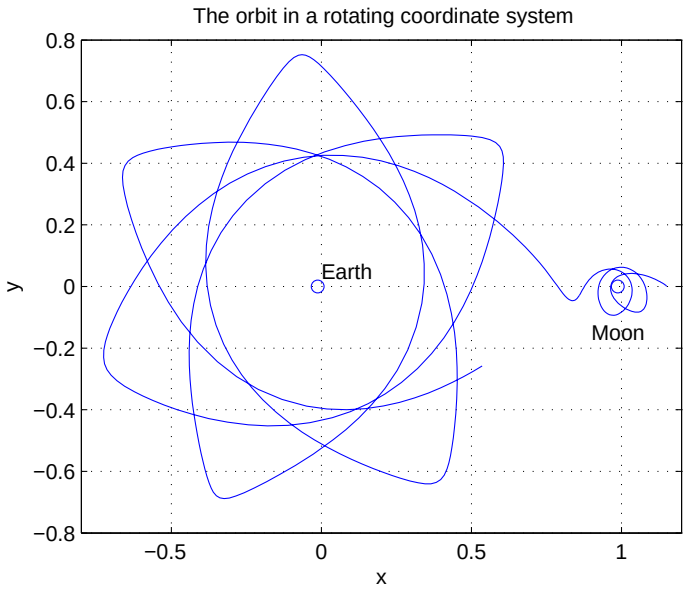
$$M = 0.012277,$$

$$
\begin{aligned}
x(0) &= 1.15, \\
\frac{dx}{dt}(0) &= 0, \\
y(0) &= 0, \\
\frac{dy}{dt}(0) &= 0.008688.
\end{aligned}
$$

Solution:

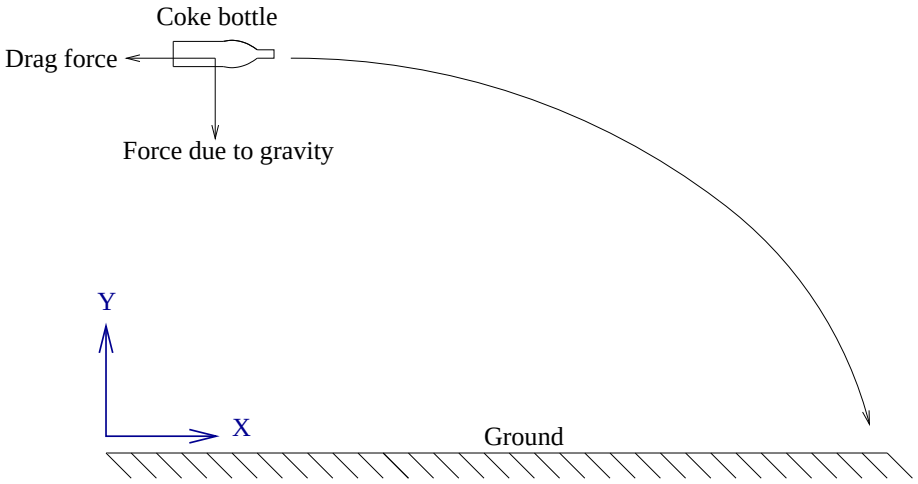
a)
```
function spacecraft
M = 0.012277; E = 1-M;

tspan = [0 24];

x0 = 1.15; xp0 = 0;
y0 = 0; yp0 = 0.008688;

z0 = [x0 xp0 y0 yp0];

[t,z] = ode45(@ZPrime,tspan,z0);

plot(z(:,1),z(:,3),-M,0,'bo',E,0,'bo')
grid on;
axis([-.8 1.2 -.8 .8]);
xlabel('x'); ylabel('y');
title('The orbit in a rotating coordinate system');
text(0,0.05,'Earth'); text(0.9,-0.15,'Moon');

    function zp = ZPrime(t,z)
        zp = zeros(4,1);

        % We define new variables z1 through z4 to simplify the expressions
        % later on
        z1 = z(1);
        z2 = z(2);
        z3 = z(3);
        z4 = z(4);

        R1 = sqrt((z1+M)^2 + (z3)^2);
        R2 = sqrt((z1-E)^2 + (z3)^2);
        % M and E are defined in the spacecraft() function.
        % We are using the technique of nested function.

        zp(1) = z2;
        zp(2) = 2*z4 + z1 - E*(z1+M)/R1^3 - M*(z1-E)/R2^3;
        zp(3) = z4;
        zp(4) = -2*z2 + z3 - E*z3/R1^3 - M*z3/R2^3;
    end
```

```
end
```



7.2 Your Turn: The Gods Must Be Crazy

In the 1980 James Uys movie "The Gods Must Be Crazy", the pilot of a airplane passing over the Kalahari desert drops a bottle of Coke which is picked up by a bushman in the desert. In this problem we will study the dynamics of the falling bottle. We will take into account the drag force due to the friction with the air.



Consider the trajectory of the bottle. We can break Newton's law of motion into two components: the horizontal(x) and the vertical(y).

$$mx'' = -C_{\mathrm{d}} \cdot ||\vec{v}|| \cdot x'$$
$$my'' = -mg - C_{\mathrm{d}} \cdot ||\vec{v}|| \cdot y'$$

where

m = mass of the bottle

g = acceleration due to gravity
C_d = Drag Coefficient (a constant)
$||\vec{v}||$ = speed of the bottle = $\sqrt{(x')^2 + (y')^2}$
$x' = \dfrac{dx}{dt}$ and $y' = \dfrac{dy}{dt}$ denote the velocities in x and y directions respectively
$x'' = \dfrac{d^2x}{dt^2}$ and $y'' = \dfrac{d^2y}{dt^2}$ denote the accelerations in the x and the y directions.

Note that the friction scales according to $||\vec{v}||^2$ and thus is larger when the bottle moves faster. Suppose a bottle of mass 0.3 kg is released at $t = 0$ from an aircraft moving at a speed of 350 m/s and at an altitude of $y_0 = 3000$ m. We want to compute the time of the flight of the bottle of Coke and where it will land on the ground. Let $y = y_0$ and $x = 0$ at $t = 0$.

a) Compute the drag coefficient using the following formula:

$$C_\mathrm{d} = \frac{mg}{{v_\mathrm{t}}^2}$$

given

$$\begin{aligned} g = &\quad \text{acceleration due to gravity } = 9.81 m/s^2 \\ v_\mathrm{t} = &\quad \text{terminal velocity of the bottle} = 300\text{m/s} \end{aligned}$$

b) Define a new vector

$$\vec{Z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = \begin{bmatrix} x \\ x' \\ y \\ y' \end{bmatrix}$$

and redefine the equations of motion as:

$$\vec{Z}' = \begin{bmatrix} z_1' \\ z_2' \\ z_3' \\ z_4' \end{bmatrix} = \begin{bmatrix} x' \\ -\frac{C_\mathrm{d}}{m}||\vec{v}||x' \\ y' \\ -g - \frac{C_\mathrm{d}}{m}||\vec{v}||y' \end{bmatrix} = \begin{bmatrix} z_2 \\ -\frac{C_\mathrm{d}}{m}\sqrt{{z_2}^2 + {z_4}^2}\, z_2 \\ z_4 \\ -g - \frac{C_\mathrm{d}}{m}\sqrt{{z_2}^2 + {z_4}^2}\, z_4 \end{bmatrix}$$

Write a function `YPrime` which takes t and Z as input and returns Z' as output. C_d, g, m are parameters in this function. Use the technique of nested functions.

c) Write a matlab program to solve the ODE using `ode45` and estimate the time of impact from the graph of $y(t)$.

d) Plot the trajectory of the bottle and hence find the total horizontal distance travelled by the bottle.

e) Plot the speed of the bottle $||\vec{v}||$ as a function of time. What is the velocity at the point of impact? What do you notice about the behaviour of the velocity? How can you explain this observation?
Hint: You have to use column 2 and column 4 of the output `y` of `ode45` for the velocities.

f) Say instead of a glass bottle, the pilot drops a can with similar weight. How does the change in the drag coefficient affect the time of impact and the horizontal distance travelled by the falling object? (You can try varying the drag coefficient by changing the terminal velocity.)

This concludes our Matlab workbook!