Computational Engineering LAB
Laboratory Manual

NAME: Shyam Vadariya

Roll No.: 21BME002

Semester: 6

Github Id: https://github.com/shyamvadalia

(B.Tech. Mechanical Engineering)

School of Technology Pandit

Deendayal Energy University

Gandhinagar Gujarat

INDEX

Contents

LAB 1: Introduction to Numerical Methods

Objective: To understand basics of numerical methods for engineers.

Theory:

Numerical methods are techniques by which mathematical problems are formulated so that they can be solved with arithmetic and logical operations. The implementation of a numerical method with an appropriate convergence check in a programming language is called a numerical algorithm.

In the pre–computer era, the time and work of implementing such calculations seriously limited their practical use. However, with the advent of fast, inexpensive digital computers, the role of numerical methods in engineering and scientific problem solving has exploded.

Example – Suppose a bungee-jumper is falling freely as shown in the figure-

The governing equation based on newton's laws are given as below,

And *net F = mg-F*$_{drag}$

$= mg - F_{drag}$

Where, v = velocity of falling man, g is acceleration due to gravity = 9.81, cd = drag coefficient = 0.25, m = mass of the bungee jumper = 68.1 kg, t = time.

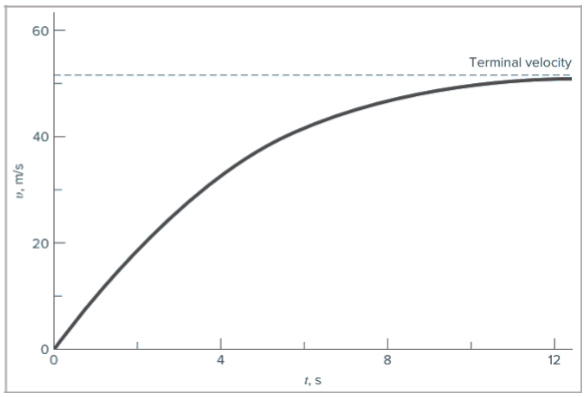
$$\frac{dv}{dt} = g - \frac{c_d}{m}v^2$$

if the jumper is initially at rest (υ = 0 at t = 0), calculus can be used to solve Equation with varying t values. (for solution-https://www.youtube.com/watch?v=VxNCIut1huw)

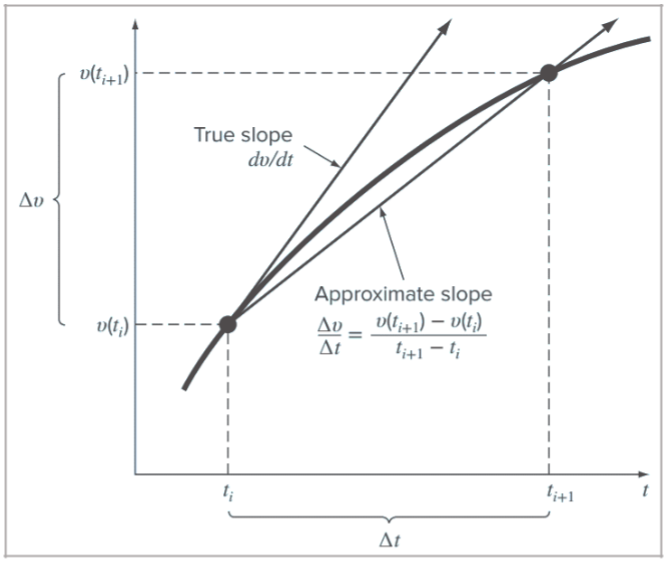
$$v(t) = \sqrt{\frac{gm}{c_d}}\tanh\left(\sqrt{\frac{gc_d}{m}}t\right)$$

t, s	υ, m/s
0	0
2	18.7292
4	33.1118
6	42.0762
8	46.9575
10	49.4214
12	50.6175
∞	51.6938

Plot the Solution →



Alternatively, we can solve above problem numerically. The use of a finite difference to approximate the first derivative of υ with respect to t.



$$\frac{d\upsilon}{dt} \cong \frac{\Delta\upsilon}{\Delta t} = \frac{\upsilon(t_{i+1}) - \upsilon(t_i)}{t_{i+1} - t_i}$$

$$\upsilon(t_{i+1}) = \upsilon(t_i) + \left[g - \frac{c_d}{m}\upsilon(t_i)^2\right](t_{i+1} - t_i)$$

$$\upsilon_{i+1} = \upsilon_i + \frac{d\upsilon_i}{dt}\Delta t$$

We can now see that the differential equation has been transformed into an equation that can be used to determine the velocity algebraically at t_{i+1} using the slope and previous values of υ and t. If you are given an initial value for velocity at some time t_i, you can easily compute velocity at a later time t_{i+1}. This new value of velocity at t_{i+1} can in turn be employed to extend the computation to velocity at t_{i+2} and so on. Thus at any time along the way,

New value = old value + slope × step size

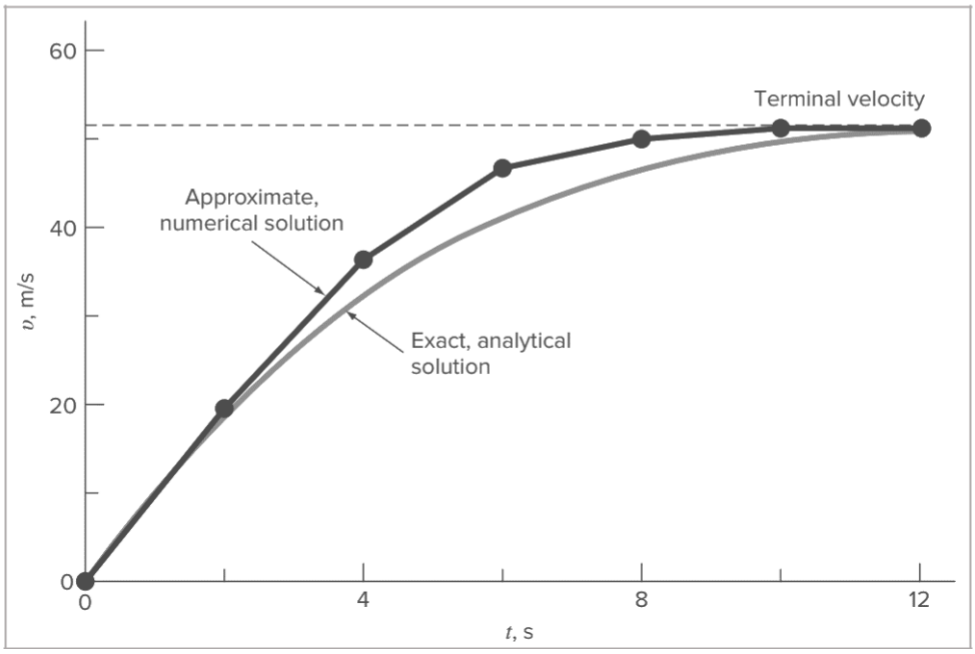
This approach is formally called **Euler's method**. Employ a step size of 2 s for the calculation, and initial condition as (υ = 0 at t = 0).

Solution. At the start of the computation ($t_0 = 0$), the velocity of the jumper is zero. Using this information and the parameter values from Example 1.1, Eq. (1.12) can be used to compute velocity at $t_1 = 2$ s:

$$v = 0 + \left[9.81 - \frac{0.25}{68.1}(0)^2\right] \times 2 = 19.62 \text{ m/s}$$

For the next interval (from $t = 2$ to 4 s), the computation is repeated, with the result

$$v = 19.62 + \left[9.81 - \frac{0.25}{68.1}(19.62)^2\right] \times 2 = 36.4137 \text{ m/s}$$



Few other real physical problems,

Assignment -

Any numerical as suggested by Faculty.

Sample numerical Problem-

Newton's law of cooling says that the temperature of a body changes at a rate proportional to the difference between its temperature and that of the surrounding medium (the ambient temperature),

$$\frac{dT}{dt} = -k(T - T_a)$$

where T = the temperature of the body (°C), t = time (min), k = the proportionality constant (per minute), and Ta = the ambient temperature (°C).

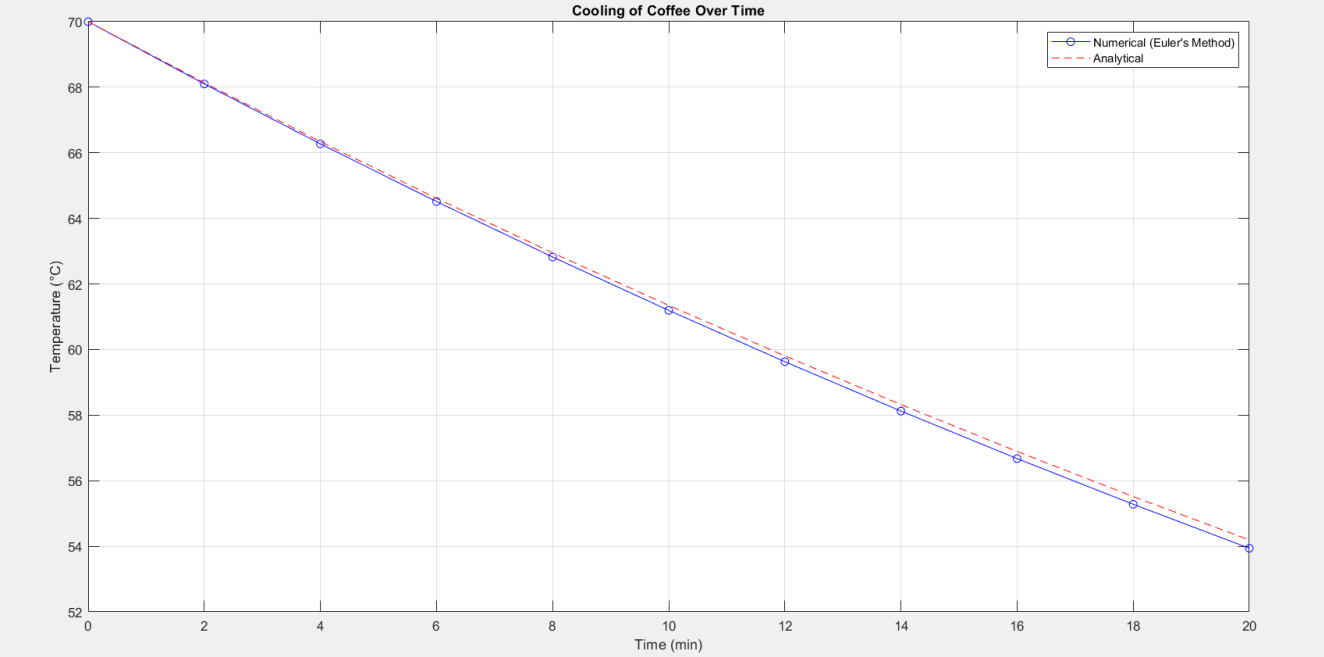
Suppose that a cup of coffee originally has a temperature of 70 °C. Use Euler's method to compute the temperature from t = 0 to 20 min using a step size of 2 min if T a = 20 °C and k = 0.019/min. Also validate the numerical results/plot with that of analytical solution.

MATLAB Code:

```
% Parameters
T0 = 70;          % Initial temperature of the coffee (°C)
Ta = 20;          % Ambient temperature (°C)
k = 0.019;        % Proportionality constant (per minute)
t_start = 0;      % Start time (min)
t_end = 20;       % End time (min)
dt = 2;           % Step size (min)

% Number of steps
num_steps = (t_end - t_start) / dt;

% Initialize arrays
time = zeros(num_steps + 1, 1);   % Time array
temperature = zeros(num_steps + 1, 1);  % Temperature array

% Set initial conditions
time(1) = t_start;
temperature(1) = T0;

% Euler's method to approximate the solution
for i = 1:num_steps
    dTdt = -k * (temperature(i) - Ta);
    temperature(i + 1) = temperature(i) + dt * dTdt;
    time(i + 1) = time(i) + dt;
end

% Analytical solution for comparison
% T(t) = Ta + (T0 - Ta) * exp(-k * t)
analytical_temperature = Ta + (T0 - Ta) * exp(-k * time);
```

```
% Plotting results
figure;
plot(time, temperature, 'b-o', time, analytical_temperature, 'r--');
xlabel('Time (min)');
ylabel('Temperature (°C)');
legend('Numerical (Euler''s Method)', 'Analytical');
title('Cooling of Coffee Over Time');
grid on;
```

Output:



LAB 2: Basics of Computational Programming and Software

Objective: Understand the basics of computational programming and software.

Theory: The primary objective is to provide basic overview of MATLAB software.

MATLAB (an abbreviation of "**MAT**rix **LAB**oratory") is a proprietary multi-paradigm programming language and numeric computing environment developed by MathWorks. MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages.

Uses of MATLAB software –

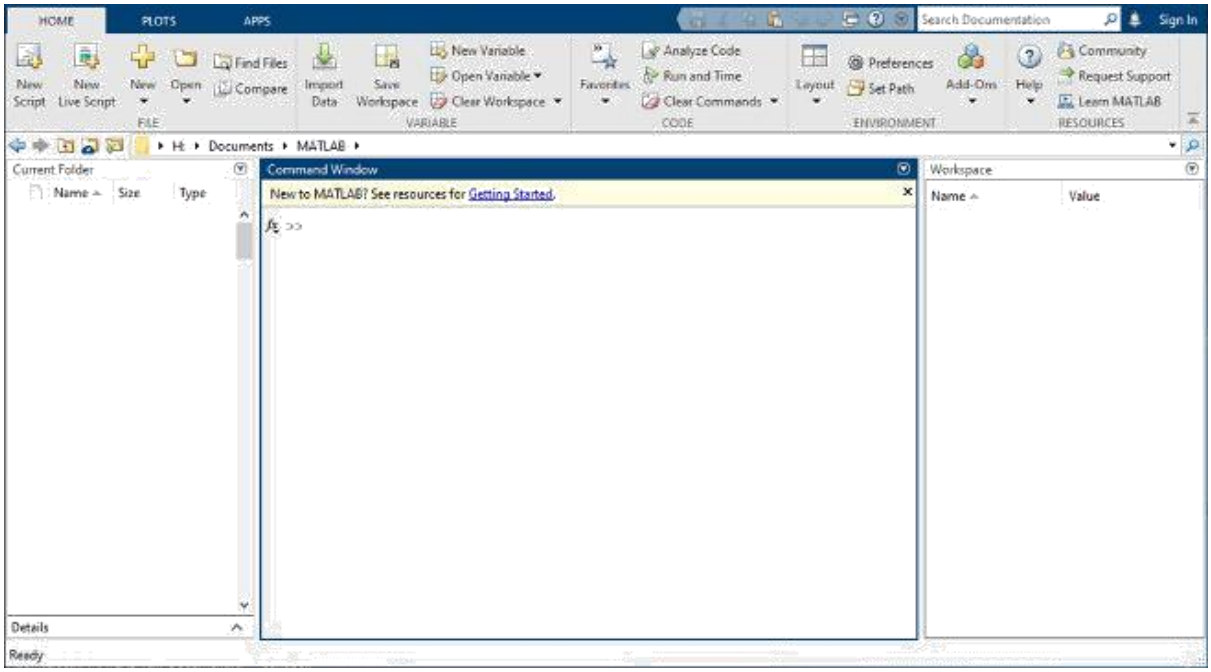
- Dealing with Matrices and Arrays
- 2-D and 3-D Plotting and graphics
- Linear Algebra
- Algebraic Equations
- Non-linear Functions
- Statistics
- Data Analysis
- Calculus and Differential Equations
- Numerical Calculations
- Integration
- Transforms
- Curve Fitting
- Various other special functions

Application of MATLAB

MATLAB is widely used as a computational tool in science and engineering encompassing the fields of all engineering streams–

- Signal Processing and Communications
- Image and Video Processing
- Control Systems
- Test and Measurement
- Computational fluid dynamics
- Computational Biology, finance
- Solid mechanics, Finite element method

When you start MATLAB®, the desktop appears in its default layout.



The desktop includes these panels:

Current Folder — Access your files.

Command Window — Enter commands at the command line, indicated by the prompt (>>).

Workspace — Explore data that you create or import from files.

As you work in MATLAB, you issue commands that create variables and call functions. For example, create a variable named a by typing this statement at the command line:

```
a = 1
```

MATLAB adds variable a to the workspace and displays the result in the Command Window.

```
a =

    1
```

Create a few more variables.

```
b = 2
b =

    2
c = a + b
c =

    3
d = cos(a)
d =

    0.5403
```

Note that the argument in the trigonometric functions should be in radians !!

Example – sin(45 degree) must be converted to→ sin (45*pi/180 radians).

When you do not specify an output variable, MATLAB uses the variable ans, short for *answer*, to store the results of your calculation.

```
sin(a)
ans =

    0.8415
```

If you end a statement with a semicolon, MATLAB performs the computation, but suppresses the display of output in the Command Window.

e = a*b;
You can recall previous commands by pressing the up- and down-arrow keys, ↑ and ↓. Press the arrow keys either at an empty command line or after you type the first few characters of a command. For example, to recall the command b = 2, type b, and then press the up-arrow key.
clc- Clear Command
Syntax clc
Description - clc clears all the text from the Command Window, resulting in a clear screen.
clear - Remove items from workspace, freeing up system memory.
clear removes all variables from the current workspace, releasing them from system memory.

close - Close one or more figure.
Syntax - close
close(fig), close all
Description - close - closes the current figure.
Commonly used Operators and Special Characters
MATLAB supports the following commonly used operators and special characters −

Operator	Purpose
+	Plus; addition operator.
-	Minus; subtraction operator.
*****	Scalar and matrix multiplication operator.
.*	Array multiplication operator.
^	Scalar and matrix exponentiation operator.
.^	Array exponentiation operator.
\	Left-division operator.
/	Right-division operator.
.\	Array left-division operator.
./	Array right-division operator.
:	Colon; generates regularly spaced elements and represents an entire row or column.
( )	Parentheses; encloses function arguments and array indices; overrides precedence.
[ ]	Brackets; enclosures array elements.
.	Decimal point.
...	Ellipsis; line-continuation operator
,	Comma; separates statements and elements in a row
;	Semicolon; separates columns and suppresses display.

%	Percent sign; designates a comment and specifies formatting.
_	Quote sign and transpose operator.
.'	Nonconjugated transpose operator.
=	Assignment operator.

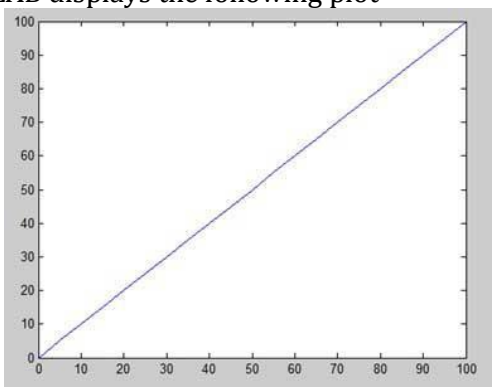
Special Variables and Constants
MATLAB supports the following special variables and constants −

Name	Meaning
ans	Most recent answer.
eps	Accuracy of floating-point precision.
i,j	The imaginary unit √-1.
Inf	Infinity.
NaN	Undefined numerical result (not a number).
pi	The number π

To plot the graph of a function, you need to take the following steps −
Define **x**, by specifying the **range of values** for the variable **x**, for which the function is to be plotted
Define the function, **y = f(x)**
Call the **plot** command, as **plot(x, y)**
Following example would demonstrate the concept. Let us plot the simple function **y = x** for the range of values for x from 0 to 100, with an increment of 5.
Create a script file and type the following code −

```
x = [0:5:100];
y = x;
plot(x, y)
```

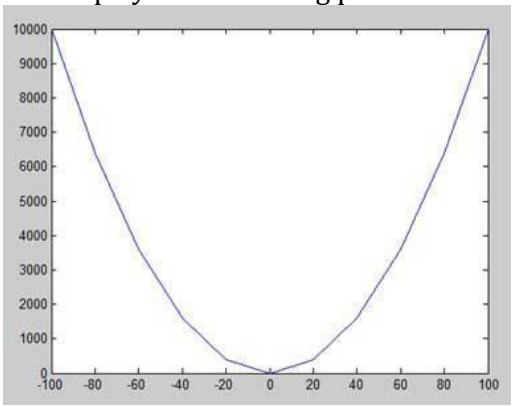
When you run the file, MATLAB displays the following plot −



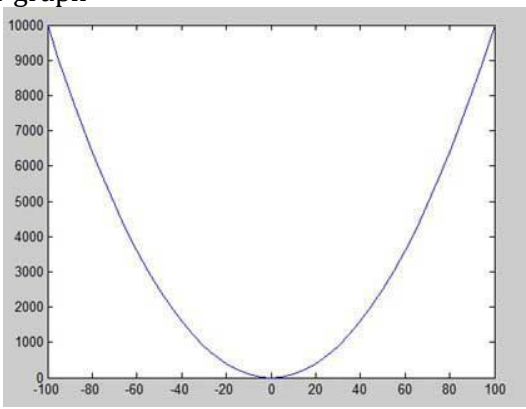
Let us take one more example to plot the function $y = x^2$. In this example, we will draw two graphs with the same function, but in second time, we will reduce the value of increment. Please note that as we decrease the increment, the graph becomes smoother. Create a script file and type the following code −

```
x = [1 2 3 4 5 6 7 8 9 10];
x = [-100:20:100];
```

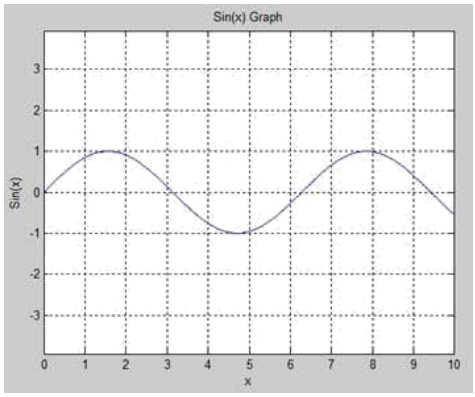
y = x.^2;
plot(x, y)
When you run the file, MATLAB displays the following plot –



Change the code file a little, reduce the increment to 5 –
x = [-100:5:100];
y = x.^2;
plot(x, y)
MATLAB draws a smoother graph –



Adding Title, Labels, Grid Lines and Scaling on the Graph
MATLAB allows you to add title, labels along the x-axis and y-axis, grid lines and also to adjust the axes to spruce up the graph.
The **xlabel** and **ylabel** commands generate labels along x-axis and y-axis.
The **title** command allows you to put a title on the graph.
The **grid on** command allows you to put the grid lines on the graph.
The **axis equal** command allows generating the plot with the same scale factors and the spaces on both axes.
The **axis square** command generates a square plot.
Example
Create a script file and type the following code –
x = [0:0.01:10];
y = sin(x);
plot(x, y), xlabel('x'), ylabel('Sin(x)'), title('Sin(x)
Graph'), grid on, axis equal
MATLAB generates the following graph –



Drawing Multiple Functions on the Same Graph

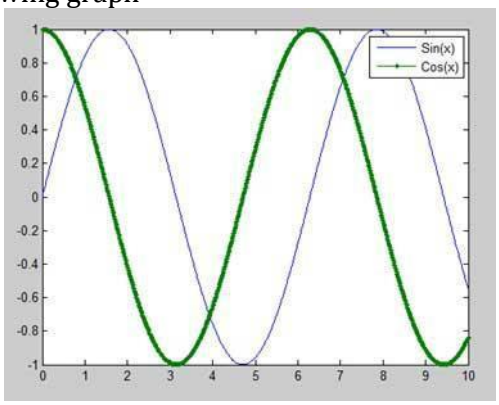
You can draw multiple graphs on the same plot. The following example demonstrates the concept –

Example

Create a script file and type the following code –

```
x = [0 : 0.01: 10];
y = sin(x);
g = cos(x);
plot(x, y, x, g, '.-'), legend('Sin(x)', 'Cos(x)')
```

MATLAB generates the following graph –

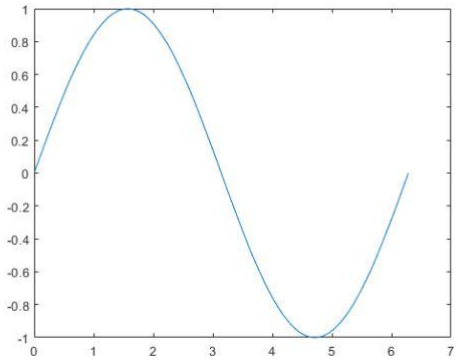


2-D and 3-D Plots

Line Plots

To create two-dimensional line plots, use the plot function. For example, plot the sine function over a linearly spaced vector of values from 0 to 2π: x = linspace(0,2*pi,50);

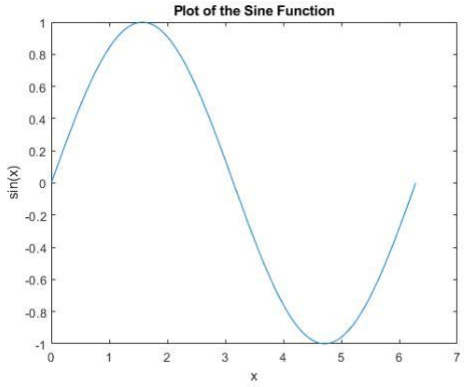
```
y = sin(x);
plot(x,y)
```



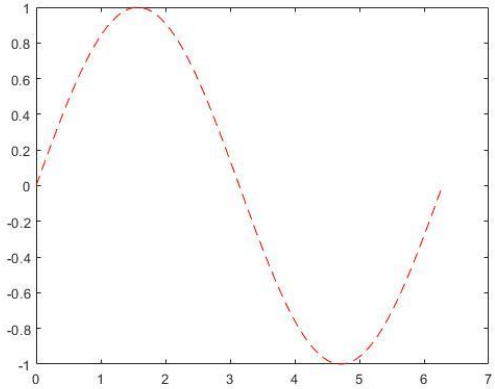
You can label the axes and add a title.

```
xlabel("x")
```

ylabel("sin(x)")
title("Plot of the Sine Function")

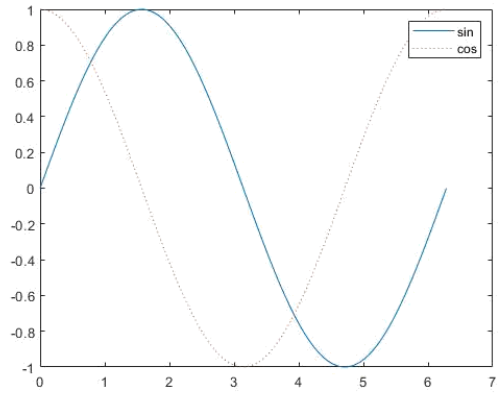


By adding a third input argument to the plot function, you can plot the same variables using a red dashed line.
plot(x,y,"r--")



"r--" is a *line specification*. Each specification can include characters for the line color, style, and marker. A marker is a symbol that appears at each plotted data point, such as a +, o, or *. For example, "g:*" requests a dotted green line with * markers.
Notice that the titles and labels that you defined for the first plot are no longer in the current figure window. By default, MATLAB® clears the figure each time you call a plotting function, resetting the axes and other elements to prepare the new plot.
To add plots to an existing figure, use hold on. Until you use hold off or close the window, all plots appear in the current figure window.

```
x = linspace(0,2*pi);
y = sin(x);
plot(x,y)
hold on
y2 = cos(x);
plot(x,y2,":")
legend("sin","cos")

hold off
```

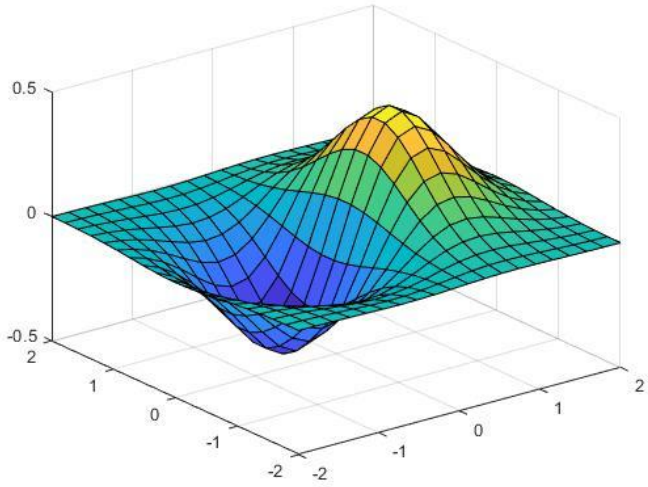


3-D Plots

Three-dimensional plots typically display a surface defined by a function in two variables, $z=f(x,y)$. For instance, calculate $z=xe^{-x2-y2}$ given row and column vectors x and y with 20 points each in the range [-2,2].

```
x = linspace(-2,2,20);
y = x';
z = x .* exp(-x.^2 - y.^2);
```

Then, create a surface plot as - surf(x,y,z)



for

for loop to repeat specified number of times

Syntax

```
for index = values
  statement
end
```

Description

for *index* = *values*, *statements*, end executes a group of statements in a loop for a specified number of times. *values* has one of the following forms:

initVal:*endVal* — Increment the *index* variable from *initVal* to *endVal* by 1, and repeat execution of *statements* until *index* is greater than *endVal*.

initVal:*step*:*endVal* — Increment *index* by the value *step* on each iteration, or decrements *index* when *step* is negative.

Example – Create an identity matrix using FOR loop command.

```
A = zeros(4,4);
   for i = 1:4
```

```
    A(i,i) = 1;
  end
```

while

while loop to repeat when condition is true

Syntax

while *expression*
  statements
end

Description

while *expression*, *statements*, end evaluates an expression, and repeats the execution of a group of statements in a loop while the expression is true. An expression is true when its result is nonempty and contains only nonzero elements (logical or real numeric). Otherwise, the expression is false.

Example - Use a while loop to calculate factorial(10).

```
n = 10;
f = n;
while n > 1
  n = n-1;
  f = f*n;
end
disp(['n! = ' num2str(f)])
n! = 3628800
```

if, elseif, else

Execute statements if condition is true

Syntax

if *expression*
  statements
elseif *expression*
  statements
else
  statements
end

Description

if *expression*, *statements*, end evaluates an expression, and executes a group of statements when the expression is true. An expression is true when its result is nonempty and contains only nonzero elements (logical or real numeric). Otherwise, the expression is false.

The elseif and else blocks are optional. The statements execute only if previous expressions in the if...end block are false. An if block can include multiple elseif blocks. Examples

Use if, elseif, and else for Conditional Assignment

Create a tridiagonal matrix using if else assignment.

First create a matrix of 1s.

```
nrows = 4;
ncols = 6;
A = ones(nrows,ncols);
```

Loop through the matrix and assign each element a new value. Assign 2 on the main diagonal, -1 on the adjacent diagonals, and 0 everywhere else.

```
for c = 1:ncols
  for r = 1:nrows
```

```
      if r == c
         A(r,c) = 2;
      elseif abs(r-c) == 1
         A(r,c) = -1;
      else
         A(r,c) = 0;
       end

   end
end
A
```

A=*4×6*

```
   2    -1    0    0    0   0
  -1     2   -1    0    0   0
   0    -1    2   -1    0   0
   0     0   -1    2   -1   0
```

Arrays in MATLAB

In this section, we will discuss some functions that create some special arrays. For all these functions, a single argument creates a square array, double arguments create rectangular array. The **zeros()** function creates an array of all zeros − For example −

```
zeros(5)
```

MATLAB will execute the above statement and return the following result

− ans =

```
      0   0   0   0   0
      0   0   0   0   0
      0   0   0   0   0
      0   0   0   0   0
      0   0   0   0   0
```

```
ones(4,3)
```

MATLAB will execute the above statement and return the following result −

ans =

```
    1   1   1
    1   1   1
    1   1   1
    1   1   1
```

The **eye()** function creates an identity matrix.

For example −

```
eye(4)
```

MATLAB will execute the above statement and return the following result

− ans =

```
      1   0   0   0
      0   1   0   0
      0   0   1   0
      0   0   0   1
```

The **rand()** function creates an array of uniformly distributed random numbers on (0,1) − For example −

```
rand(3, 5)
```

MATLAB will execute the above statement and return the following result

− ans =

```
  0.8147  0.9134  0.2785  0.9649  0.9572
```

```
  0.9058  0.6324  0.5469  0.1576  0.4854
  0.1270  0.0975  0.9575  0.9706  0.8003
```

Multidimensional Arrays

An array having more than two dimensions is called a multidimensional array in MATLAB. Multidimensional arrays in MATLAB are an extension of the normal two-dimensional matrix. Generally to generate a multidimensional array, we first create a two-dimensional array and extend it.

For example, let's create a two-dimensional array a.

```
a = [7 9 5; 6 1 9; 4 3 2]
```

MATLAB will execute the above statement and return the following result –

```
a =
  7  9  5
  6   1   9
  4   3   2
```

The array *a* is a 3-by-3 array; we can add a third dimension to *a*, by providing the values like –

```
a(:, :, 2)= [ 1 2 3; 4 5 6; 7 8 9]
```

MATLAB will execute the above statement and return the following result –

```
a =
ans(:,:,1) =

  0 0 0
  0 0 0
   0 0 0
ans(:,:,2) =
   1 2 3
   4 5 6
   7 8 9
```

Assignment -

One numerical as suggested by Faculty.

Sample numerical.

1. Draw the functions

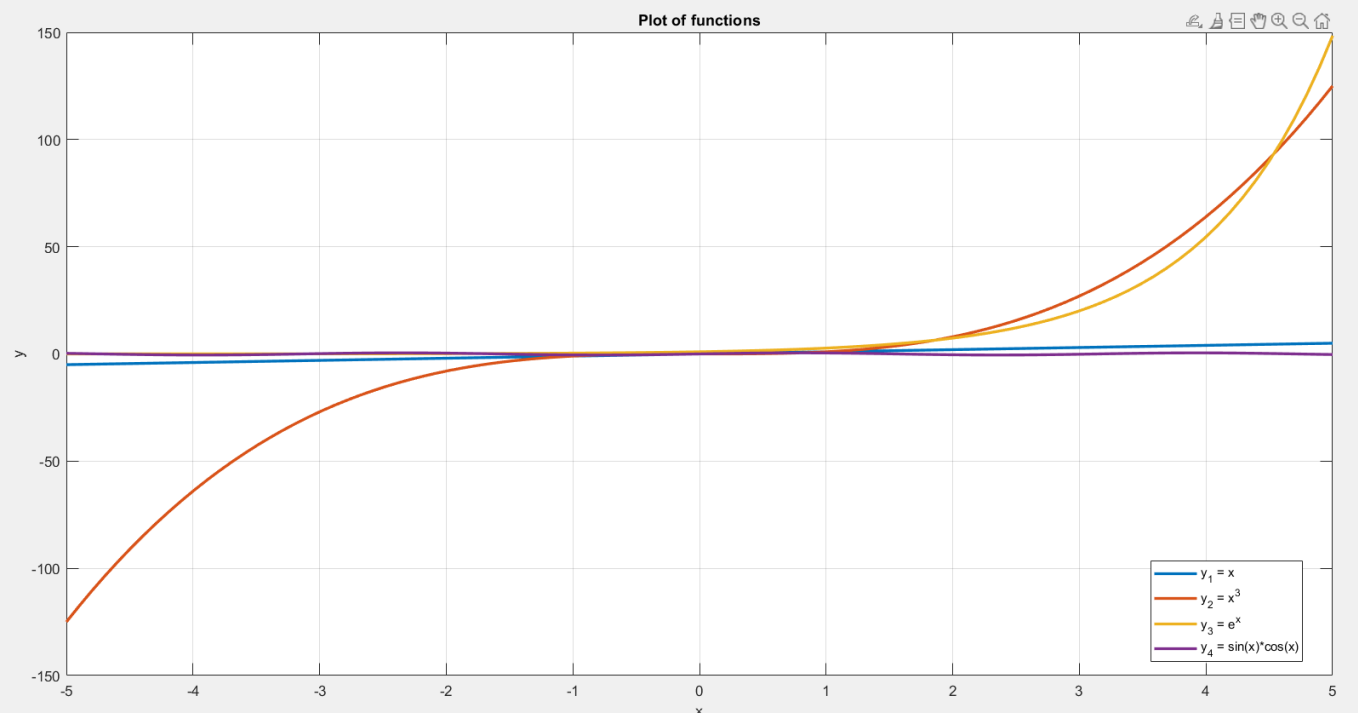
on same plot for the range of x from -5 to +5. Also label the plots aesthetically.

MATLAB Code:

```
% Define the range of x values
x = linspace(-5,5,100);

% Define all the functions
y1 = x;
y2 = x.^3;
y3 = exp(x);
y4 = sin(x).*cos(x);

% Plot the functions
figure;
```

```
plot(x,y1,'LineWidth',2,'DisplayName','y_1 = x');
hold on;
plot(x,y2,'LineWidth',2,'DisplayName','y_2 = x^3');
plot(x,y3,'LineWidth',2,'DisplayName','y_3 = e^x');
plot(x,y4,'LineWidth',2,'DisplayName','y_4 = sin(x)*cos(x)');

% Label the axes and add a legend
xlabel('x');
ylabel('y');
title('Plot of functions');
legend('Location','best');
grid on;

% Display the plot
hold off;
```

Output:



2. Make a MATLAB program that sums number from 1 to 100 using FOR loop statement.

MATLAB Code:

```
% Initialize the sum
totalSum = 0;

% Use a FOR loop to iterate from 1 to 100
for i = 1:100
    totalSum = totalSum+i;
end

% Display the result
fprintf('The sum of numbers from 1 to 100 is: %d\n',totalSum);
```

Output:

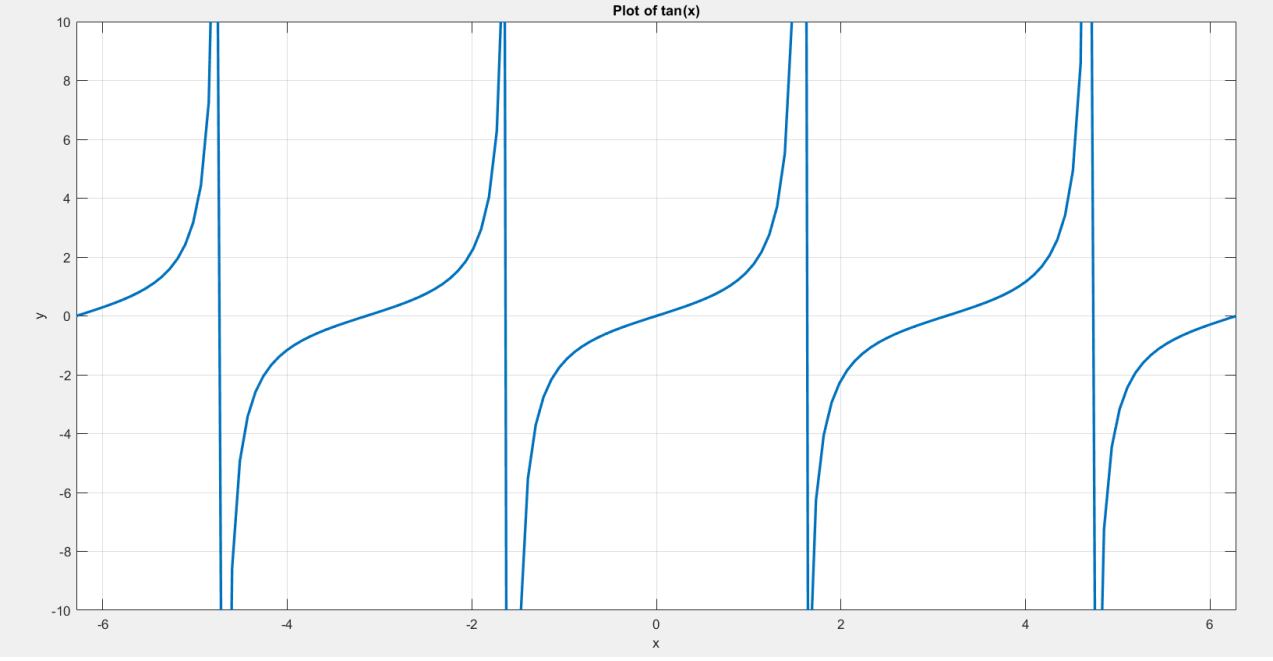
```
>> Lab_2_Numerical2
The sum of numbers from 1 to 100 is: 5050
```

2. Write a MATLAB code to plot 'tan' function with x domain as -2 to 2 with 150 number of points. (Hint – when you plot, you will notice the plot is not aesthetic. To make your plot aesthetic, use – xlim and ylim functions.
   Example - xlim([-2*pi 2*pi]); ylim([-10 10]);

MATLAB Code:

```
% Define the x values
x = linspace(-2*pi,2*pi,150);

% Calculate the tangent values
y = tan(x);

% Plot the tangent function
figure;
plot(x,y,'LineWidth',2);
title('Plot of tan(x)');
xlabel('x');
ylabel('y');
grid on;

% Set xlim and ylim for better aesthetics
xlim([-2*pi,2*pi]);
ylim([-10,10]);
```

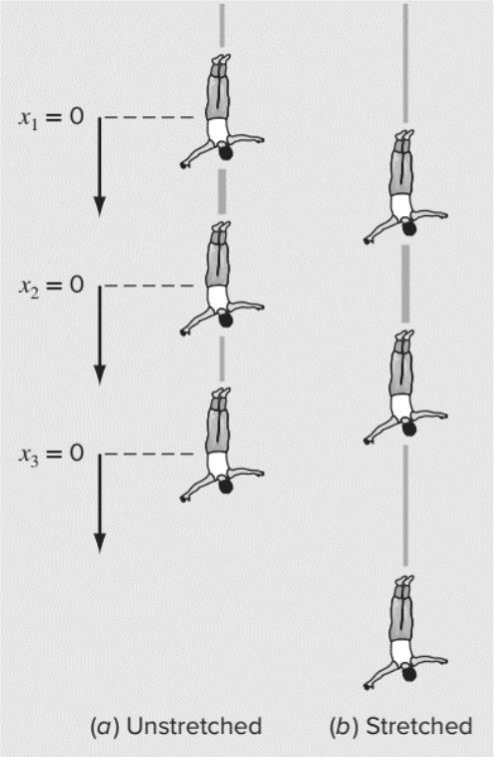
Output:

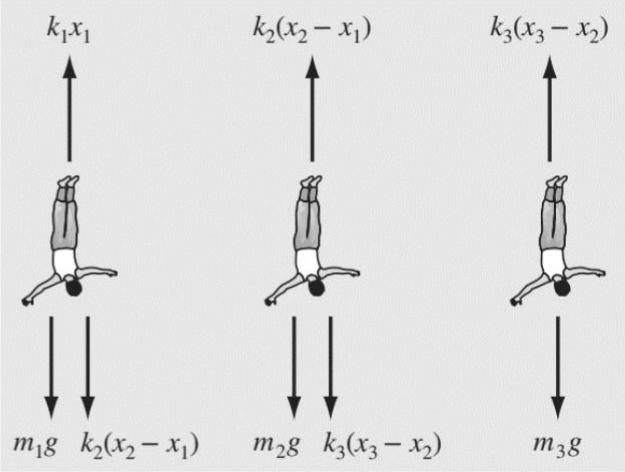


LAB 3: Systems of Linear Algebraic Equations

Objective: To represent a system of linear algebraic equations in matrix form and obtain the solution.

Theory: Suppose that three jumpers are connected by bungee cords. Being held in place vertically so that each cord is fully extended but unstretched. We can define three distances, x_1, x_2, and x_3, as measured downward from each of a shows them their unstretched positions. After they are released, gravity takes hold and the jumpers will eventually come to the equilibrium positions shown in Suppose that you are asked to compute the displacement of each of the jumpers. If we assume that each cord behaves as a linear spring and follows Hooke's law, free-body diagrams can be developed for each jumper as depicted-





Free body diagram (using Hooke's law)

Using Newton's second law, force balances can be written for each jumper:

$$m_1 \frac{d^2x_1}{dt^2} = m_1 g + k_2(x_2 - x_1) - k_1x_1$$

$$m_2 \frac{d^2x_2}{dt^2} = m_2 g + k_3(x_3 - x_2) + k_2(x_1 - x_2)$$

$$m_3 \frac{d^2x_3}{dt^2} = m_3 g + k_3(x_2 - x_3)$$

where m_i = the mass of jumper i (kg), t = time (s), k_j = the spring constant for cord j (N/m), x_i = the displacement of jumper i measured downward from the equilibrium position (m), and g = gravitational acceleration (9.81 m/s^2). Because we are interested in the steady-state solution, the second derivatives can be set to zero. Collecting terms gives

$$\begin{aligned}(k_1 + k_2)x_1 \qquad - k_2x_2 \qquad &= m_1g\\ -k_2x_1 + (k_2 + k_3)x_2 - k_3x_3 &= m_2g\\ -k_3x_2 + k_3x_3 &= m_3g\end{aligned}$$

Thus, the problem reduces to solving a system of three simultaneous equations for the three unknown displacements. Because we have used a linear law for the cords, these equations are linear algebraic equations. MATLAB provides two direct ways to solve systems of linear algebraic equations. The most efficient way is to employ the backslash, or "left-division," operator as in

```
>> x = A\b
```

The second is to use matrix inversion:

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

However, the matrix inverse solution is less efficient than using the backslash.

Use MATLAB to solve the bungee jumper problem described. The parameters for the problem are

Jumper	Mass (kg)	Spring Constant (N/m)	Unstretched Cord Length (m)
Top (1)	60	50	20
Middle (2)	70	100	20
Bottom (3)	80	50	20

The system of equations

$$\begin{aligned}(k_1 + k_2)x_1 \qquad - k_2x_2 \qquad &= m_1g\\ -k_2x_1 + (k_2 + k_3)x_2 - k_3x_3 &= m_2g\\ -k_3x_2 + k_3x_3 &= m_3g\end{aligned}$$

are written as,

$$\begin{bmatrix} 150 & -100 & 0 \\ -100 & 150 & -50 \\ 0 & -50 & 50 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 588.6 \\ 686.7 \\ 784.8 \end{Bmatrix}$$

Start up MATLAB and enter the coefficient matrix and the right-hand-side vector:

```
>> K = [150 -100 0;-100 150 -50;0 -50 50]

K =
   150  -100     0
  -100   150   -50
     0   -50    50

>> mg = [588.6; 686.7; 784.8]

mg =
  588.6000
  686.7000
  784.8000
```

Employing left division yields

```
>> x = K\mg

x =
   41.2020
   55.9170
   71.6130
```

Alternatively, multiplying the inverse of the coefficient matrix by the right-hand-side vector gives the same result:

```
>> x = inv(K)*mg

x =
   41.2020
   55.9170
   71.6130
```

Because the jumpers were connected by 20-m cords, their initial positions relative to the platform is

```
>> xi = [20;40;60];
```

Thus, their final positions can be calculated as

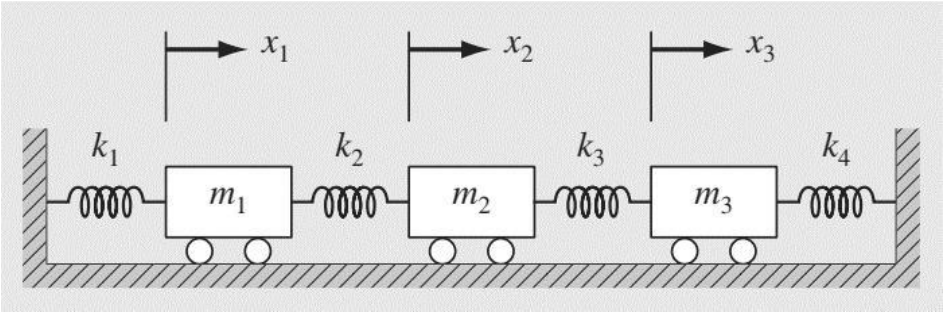
```
>> xf = x+xi

xf =
   61.2020
   95.9170
  131.6130
```

Assignment -

One numerical as suggested by Faculty.

Sample numerical.

Consider the three mass-four spring system in Fig. Determining the equations of motion from $\Sigma F_x = ma_x$ for each mass using its free-body diagram results in the



Derive the governing equation for acceleration for all the three blocks?

where $k_1 = k_4 = 10$ N/m, $k_2 = k_3 = 30$ N/m, and $m_1 = m_2 = m_3 = 1$ kg. The three equations can be written in matrix form:

$$0 = \{\text{Acceleration vector}\} + [k/m \text{ matrix}]\{\text{displacement vector } x\}$$

At a specific time where $x_1 = 0.05$ m, $x_2 = 0.04$ m, and $x_3 = 0.03$ m, this forms a tridiagonal matrix. Use MATLAB to solve for the acceleration of each mass.

MATLAB Code:

```
% Define constants
k1 = 10;   % N/m
k2 = 30;   % N/m
k3 = 30;   % N/m
k4 = 10;   % N/m
m1 = 1;    % kg
m2 = 1;    % kg
m3 = 1;    % kg
x1 = 0.05; % m
x2 = 0.04; % m
x3 = 0.03; % m

% Construct the coefficient matrix [A] and right-hand side vector [B]
A = [m1, 0, 0; -k1, k1 + k2, -k2; 0, -k2, k2 + k3];
B = [-k1 * x1; k2 * (x2 - x1); k3 * (x3 - x2)];

% Solve for accelerations using matrix inversion
accelerations = A \ B;

% Display the accelerations
a1 = accelerations(1);
a2 = accelerations(2);
a3 = accelerations(3);

fprintf('Acceleration of mass 1 (a1) = %.4f m/s^2\n', a1);
fprintf('Acceleration of mass 2 (a2) = %.4f m/s^2\n', a2);
fprintf('Acceleration of mass 3 (a3) = %.4f m/s^2\n', a3);
```

Output:

```
>> Lab_3
Acceleration of mass 1 (a1) = -0.5000 m/s^2
Acceleration of mass 2 (a2) = -0.2180 m/s^2
Acceleration of mass 3 (a3) = -0.1140 m/s^2
```

LAB 4: Eigenvalue Problems

Objective: To obtain the eigenvalue of a dynamical system.

Theory: Previously, have dealt with methods for solving sets of linear algebraic equations of the general form

$$[A]\{X\} = [B]$$

Such systems are called nonhomogeneous because of the presence of the vector {B} on the right-hand side of the equality. In contrast, a homogeneous linear algebraic system has a right-hand side equal to zero:

$$[A]\{X\} = [0]$$

At face value, this equation suggests that the only possible solution would be the trivial case for which all x's = 0. Graphically this would correspond to two straight lines that intersected at zero. Eigenvalue problems associated with engineering are typically of the general form,

$$[A]\{X\} = \{X\},$$

$$[A]-[\ ]\{X\} = 0,$$

Where the parameter λ is the eigenvalue. Thus, rather than setting the x's to zero, we can determine the value of λ that drives the left-hand side to zero! One way to accomplish this is based on the fact that, for nontrivial solutions to be possible, the determinant of the matrix must equal zero:

$$\det[A]-[\ ]=0$$

Expanding the determinant yields a polynomial in λ, which is called the characteristic polynomial. The roots of this polynomial are the solutions for the eigenvalues.

$$\begin{aligned}(a_{11}-\lambda)x_1 + a_{12}x_2 &= 0\\ a_{21}x_1 + (a_{22}-\lambda)x_2 &= 0\end{aligned}$$

$$\begin{vmatrix} a_{11}-\lambda & a_{12} \\ a_{21} & a_{22}-\lambda \end{vmatrix} = \lambda^2 - (a_{11}+a_{22})\lambda - a_{12}a_{21}$$

$$\begin{matrix}\lambda_1 \\ \lambda_2\end{matrix} = \frac{(a_{11}-a_{22}) \pm \sqrt{(a_{11}-a_{22})^2 - 4a_{12}a_{21}}}{2}$$

Problem Statement. Use the polynomial method to solve for the eigenvalues of the following homogeneous system:

$$\begin{aligned}(10-\lambda)x_1 - 5x_2 &= 0\\ -5x_1 + (10-\lambda)x_2 &= 0\end{aligned}$$

$$\begin{matrix}\lambda_1 \\ \lambda_2\end{matrix} = \frac{20 \pm \sqrt{20^2 - 4(1)75}}{2} = 15, 5$$

We can now substitute either of these values back into the system and examine the result. For $\lambda_1 = 15$, we obtain

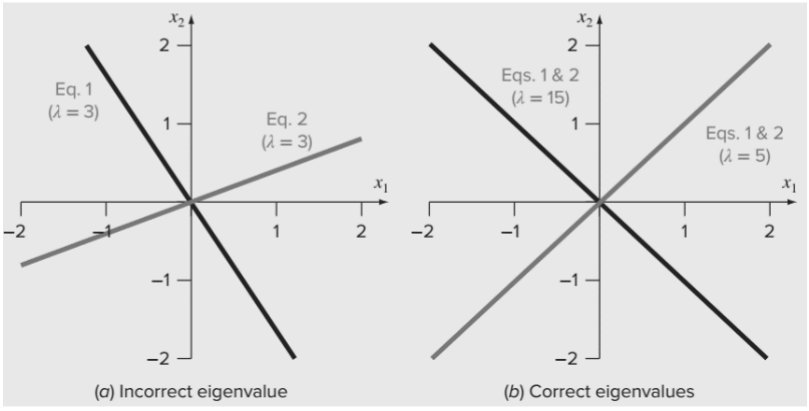
$$-5x_1 - 5x_2 = 0$$
$$-5x_1 - 5x_2 = 0$$

In essence as we move toward a correct eigenvalue the two lines rotate until they lie on top of each other. Mathematically, this means that there are an infinite number of solutions. But solving either of the equations yields the interesting result that all the solutions have the property that x 1 = −x2. Although at first glance this might appear trivial, it's actually quite interesting as it tells us that the ratio of the unknowns is a constant. This result can be expressed in vector form as,

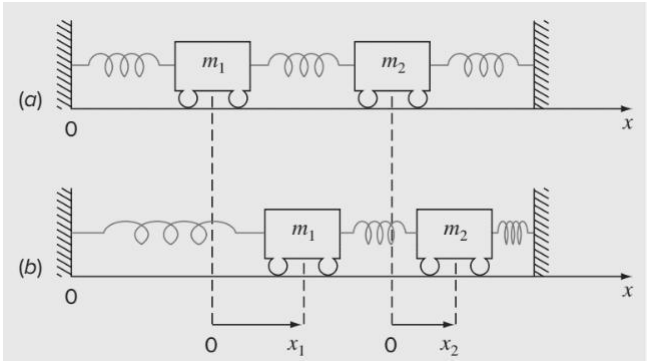
$$\{x\} = \begin{Bmatrix} -1 \\ 1 \end{Bmatrix}$$

which is referred to as the *eigenvector* corresponding to the eigenvalue $\lambda = 15$.
In a similar fashion, substituting the second eigenvalue, $\lambda_2 = 5$, gives

$$5x_1 - 5x_2 = 0$$
$$-5x_1 + 5x_2 = 0$$



Example- A two mass–three spring system with frictionless rollers vibrating between two fixed walls. The position of the masses can be referenced to local coordinates with origins at their respective equilibrium positions (a). As in (b), positioning the masses away from equilibrium creates forces in the springs that on release lead to oscillations of the masses.



Newton's second law can be employed to develop a force balance for each mass:

$$m_1 \frac{d^2x_1}{dt^2} = -kx_1 + k(x_2 - x_1)$$

$$m_2 \frac{d^2x_2}{dt^2} = -k(x_2 - x_1) - kx_2$$

where x_i is the displacement of mass i away from its equilibrium position from vibration theory, it is known that solutions can take the form,

$$x_i = X_i \sin(\omega t)$$

where X_i = the *amplitude* of the oscillation of mass i (m) and ω = the *angular frequency* of the oscillation (radians/time), which is equal to

$$\omega = \frac{2\pi}{T_p}$$

where T_p = the *period* (time/cycle). Note that the inverse of the period is called the *ordinary frequency* f (cycles/time). If time is measured in seconds, the unit for f is the cycles/s, which is referred to as a *Hertz* (Hz).

Equation can be differentiated twice and substituted into Eq. After collection of terms, the result in,

$$\left(\frac{2k}{m_1} - \omega^2\right) X_1 - \frac{k}{m_1} X_2 = 0$$

$$-\frac{k}{m_2} X_1 + \left(\frac{2k}{m_2} - \omega^2\right) X_2 = 0$$

Comparison of Eq. with general form of eigenvalue Eq. indicates that at this point, the solution has been reduced to an eigenvalue problem—where, for this case, the eigenvalue is the square of the frequency.

If $m_1 = m_2 = 40$ kg and $k = 200$ N/m,

$$(10 - \lambda)x_1 - 5x_2 = 0$$
$$-5x_1 + (10 - \lambda)x_2 = 0$$

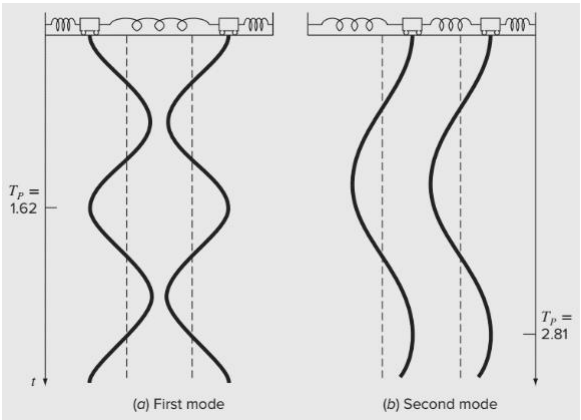


Figure - The principal modes of vibration of two equal masses connected by three identical springs between fixed walls.

As might be expected, MATLAB has powerful and robust capabilities for evaluating eigenvalues and eigenvectors. The function eig, which is used for this purpose, can be employed to generate a vector of the eigenvalues as in

```
>> e = eig(A)
```

where e is a vector containing the eigenvalues of a square matrix A. Alternatively, it can be invoked as

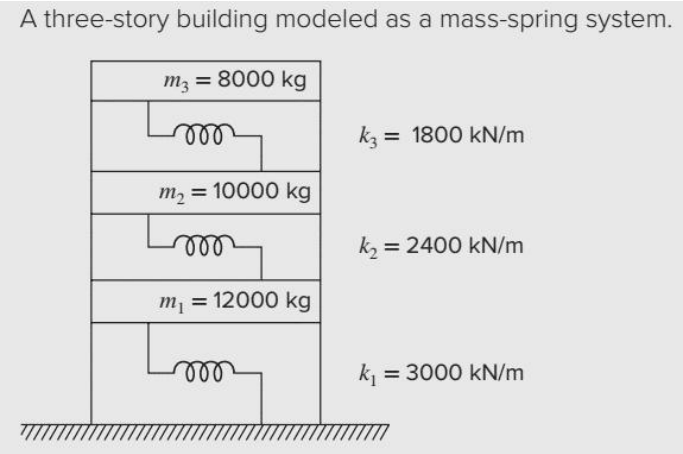
```
>> [V,D] = eig(A)
```

where D is a diagonal matrix of the eigenvalues and V is a full matrix whose columns are the corresponding eigenvectors.

Assignment -

One numerical as suggested by Faculty.

Sample numerical.



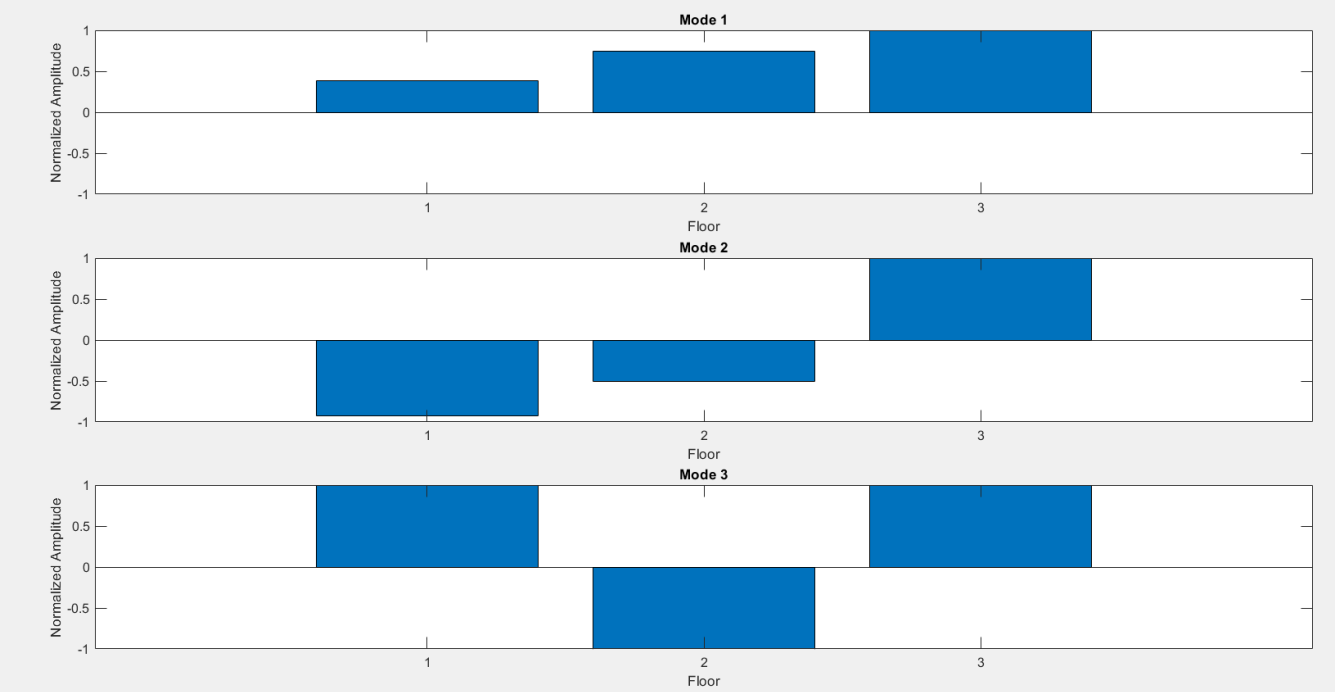
Engineers and scientists use mass-spring models to gain insight into the dynamics of structures under the influence of disturbances such as earthquakes. Use MATLAB to determine the eigenvalues and eigenvectors for this system. Graphically represent the modes of vibration for the structure by displaying the amplitudes versus height for each of the eigenvectors. Normalize the amplitudes so that the translation of the third floor is one.

MATLAB Code:

```
% Define mass and stiffness matrices
m1 = 12000; m2 = 10000; m3 = 8000;
k1 = 3000; k2 = 2400; k3 = 1800;

M = diag([m1, m2, m3]); % Mass matrix
K = [k1 + k2, -k2, 0;
     -k2, k2 + k3, -k3;
     0, -k3, k3];        % Stiffness matrix

% Solve the eigenvalue problem
[V, D] = eig(K, M);

% Extract eigenvalues (D) and eigenvectors (V)
lambda = diag(D);

% Display eigenvalues
disp('Eigenvalues:');
disp(lambda);
```

```
% Display eigenvectors
disp('Eigenvectors:');
disp(V);
% Normalize eigenvectors
normalized_V = V ./ V(3, :);

% Plot amplitudes versus height for each eigenvector
figure;
heights = [1, 2, 3]; % Floor heights
for i = 1:size(V, 2)
    subplot(size(V, 2), 1, i);
    bar(heights, normalized_V(:, i));
    title(['Mode ', num2str(i)]);
    xlabel('Floor');
    ylabel('Normalized Amplitude');
    ylim([-1, 1]); % Adjust ylim as needed
end
```

Output:




```
>> Lab_4
Eigenvalues:
   0.0569
   0.3395
   0.6986

Eigenvectors:
  -0.0031    0.0064   -0.0058
  -0.0060    0.0035    0.0071
  -0.0081   -0.0069   -0.0034
```

LAB 5: Non-linear Equations

Objective: To obtain the numerical solution of any non-linear equation.

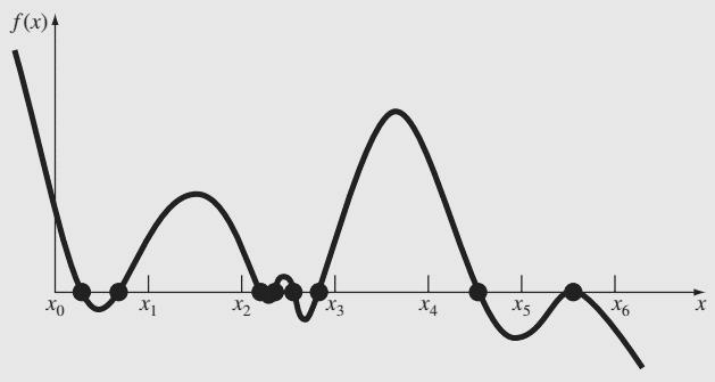
Theory: Root of the equation f(x) = 0 is to make a plot of the function and observe where it crosses the x axis. The two major classes of methods available are distinguished by the type of initial guess. They are

1. Bracketing methods - As the name implies, these are based on two initial guesses that "bracket" the root that is, are on either side of the root.
2. Open methods - These methods can involve one or more initial guesses, but there is no need for them to bracket the root.

f (x) changed sign on opposite sides of the root. In general, if f(x) is real and continuous in the interval from x_l to x_u and f(x_l) and f(x_u) have opposite signs, that is,

$$f(x_l)\, f(x_u) < 0$$

then there is at least one real root between x_l and x_u.



The Bisection method –

Example - To determine the mass of the bungee jumper with a drag coefficient of 0.25 kg/m to have a velocity of 36 m/s after 4 s of free fall. Note: The acceleration of gravity is 9.81 m/s2

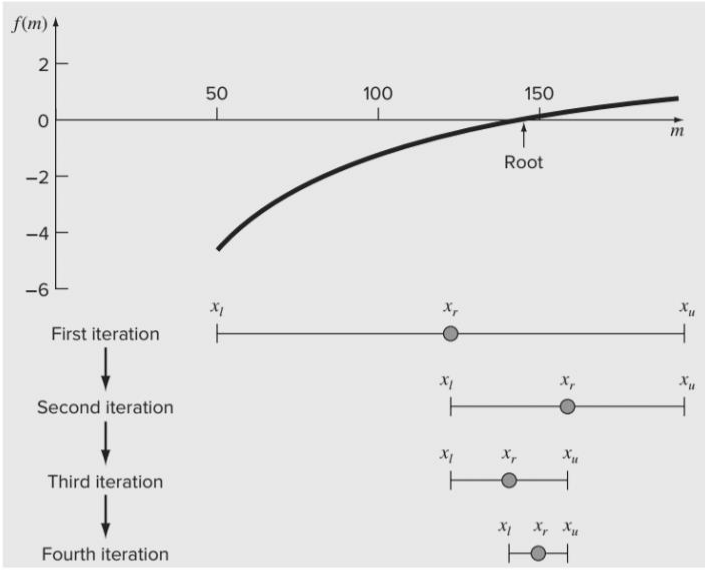
$$f(m) = \sqrt{\frac{gm}{c_d}} \tanh\left(\sqrt{\frac{gc_d}{m}}\, t\right) - v(t)$$

We can see that the function changes sign between values of 50 and 200. The plot obviously suggests better initial guesses, say 140 and 150, but for illustrative purposes let's assume we don't have the benefit of the plot and have made conservative guesses. Therefore, the initial estimate of the root x_r lies at the midpoint of the interval,

$$x_r = \frac{50 + 200}{2} = 125$$

$$|\varepsilon_t| = \left|\frac{142.7376 - 125}{142.7376}\right| \times 100\% = 12.43\%$$

$$f(50)\,f(125) = -4.579(-0.409) = 1.871$$



At this point, the new interval extends from $x_l = 125$ to $x_u = 200$. A revised root estimate can then be calculated as

$$x_r = \frac{125 + 200}{2} = 162.5$$

which represents a true percent error of $|\varepsilon_t| = 13.85\%$. The process can be repeated to obtain refined estimates. For example,

$$f(125)\,f(162.5) = -0.409(0.359) = -0.147$$

Therefore, the root is now in the lower interval between 125 and 162.5. The upper bound is redefined as 162.5, and the root estimate for the third iteration is calculated as

$$x_r = \frac{125 + 162.5}{2} = 143.75$$

which represents a percent relative error of $\varepsilon_t = 0.709\%$. The method can be repeated until the result is accurate enough to satisfy your needs.

Stopping criteria,

$$|\varepsilon_a| = \left|\frac{x_r^{\text{new}} - x_r^{\text{old}}}{x_r^{\text{new}}}\right| 100\%$$

Iteration	x_l	x_u	x_r	$\lvert\varepsilon_a\rvert$ (%)	$\lvert\varepsilon_t\rvert$ (%)
1	50	200	125		12.43
2	125	200	162.5	23.08	13.85
3	125	162.5	143.75	13.04	0.71
4	125	143.75	134.375	6.98	5.86
5	134.375	143.75	139.0625	3.37	2.58
6	139.0625	143.75	141.4063	1.66	0.93
7	141.4063	143.75	142.5781	0.82	0.11
8	142.5781	143.75	143.1641	0.41	0.30

Other methods – False position, Newton-Raphson method etc.

Example- Find the roots of equation – $( ) = ^{0.2} - ^{-0.8} - 2$ using bisection method. Take the initial guesses as root may lie between 0 and 4.

MATLAB routine for finding roots is as follows-

```
clc;close all; clear all;
x = linspace(0,10,50);
y = exp(0.2*x)-exp(-0.8*x)-2;
xl =0;
xu =4;
err= 0.01;
 iter = 1;
while abs(xu-xl)> err
    xr = (xl+xu)/2;
    yl = exp(0.2*xl)-exp(-0.8*xl)-2;
    yr = exp(0.2*xr)-exp(-0.8*xr)-2;
    yu = exp(0.2*xu)-exp(-0.8*xu)-2;
   if yl*yr <0
       xu=xr;
   else
       xl=xr;
   end
  iter = iter+1;
end
fprintf('root of equation = %f',xr);
plot(x,y);
hold on;
hline = refline([0 0]);
```

Assignment -

One numerical as suggested by Faculty.

Sample numerical.

Using Bisection method, locate the root of –

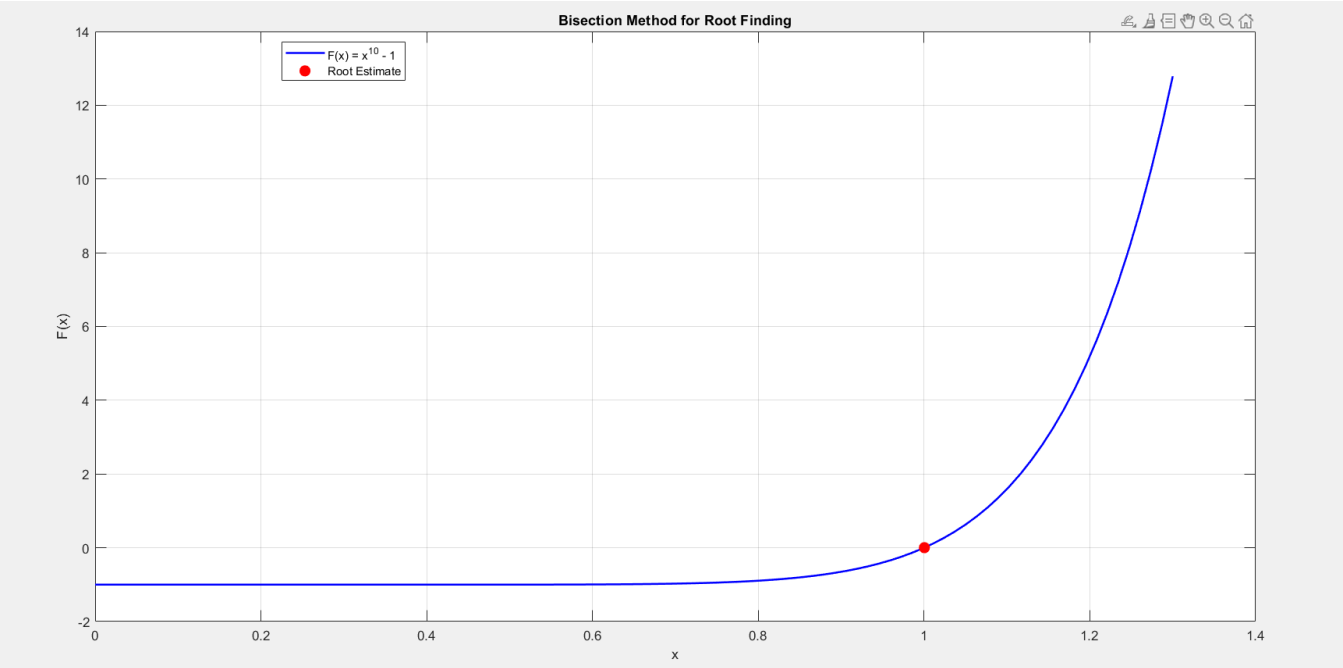
$$F(x) = x_{10} - 1, \text{ between } x = 0 \text{ to } 1.3$$

MATLAB Code:

```
% Define the function F(x)
F = @(x) x.^10 - 1;

% Define the interval [x1, xu]
x1 = 0;
xu = 1.3;
```

```
% Desired error tolerance
err = 0.01;

% Initialize iteration counter and root estimate
iter = 0;
xr = (x1 + xu) / 2;

% Perform bisection method
while abs(F(xr)) > err
    % Calculate function values at endpoints and midpoint
    y1 = F(x1);
    yr = F(xr);
    yu = F(xu);

    % Determine which subinterval to select
    if y1 * yr < 0
        % Root is in the lower subinterval
        xu = xr;
    else
        % Root is in the upper subinterval
        x1 = xr;
    end

    % Calculate new midpoint
    xr = (x1 + xu) / 2;

    % Increment iteration counter
    iter = iter + 1;
end

% Display the root and number of iterations
fprintf('Root of the equation = %.6f\n', xr);
fprintf('Number of iterations = %d\n', iter);

% Plot the function F(x) and mark the root
x_vals = linspace(0, 1.3, 100);
y_vals = F(x_vals);

figure;
plot(x_vals, y_vals, 'b-', 'LineWidth', 1.5);
hold on;
plot(xr, F(xr), 'ro', 'MarkerSize', 8, 'MarkerFaceColor', 'r');
xlabel('x');
ylabel('F(x)');
title('Bisection Method for Root Finding');
grid on;
legend('F(x) = x^{10} - 1', 'Root Estimate', 'Location', 'best');
```

Output:

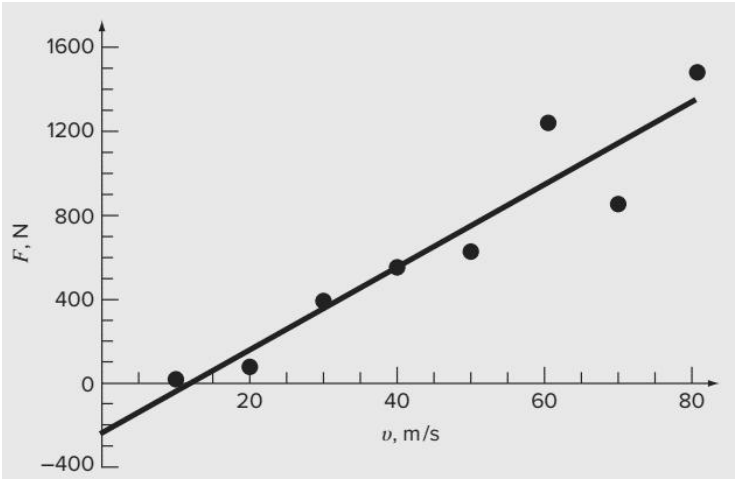



```
>> Lab_5
Root of the equation = 1.000391
Number of iterations = 7
```

LAB 6: Polynomial Approximation

Objective: To obtain the linear and polynomial approximation (curve fitting) for a given data set.

Theory: Given a set of points (x_i, y_i) for i=0,1,2,…, n, where the x_i are distinct values of the independent variable and the y_i are corresponding values of some function f, Either Approximate the value of y at some value of x not listed among the x_i or Determine a function g that in some sense approximate the data.



The mathematical expression for the straight line is

$$y = a_0 + a_1x + e$$

where a_0 and a_1 are coefficients representing the intercept and the slope, respectively, and e is the error, or residual, between the model and the observations, which can be represented by rearranging Eq. as

$$e = y - a_0 - a_1x$$

Thus, the residual is the discrepancy between the true value of y and the approximate value, $a_0 + a_1x$, predicted by the linear equation. The approach is to minimize the sum of the squares of the residuals as,

$$S_r = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - a_0 - a_1x_i)^2$$

This criterion, which is called *least square method*, has a number of advantages, including that it yields a unique line for a given set of data. To determine values for a_0 and a_1, Eq. is differentiated with respect to each unknown coefficient:

$$\frac{\partial S_r}{\partial a_0} = -2\sum(y_i - a_0 - a_1x_i)$$

$$\frac{\partial S_r}{\partial a_1} = -2\sum[(y_i - a_0 - a_1x_i)x_i]$$

all summations are from i = 1 to n. Setting these derivatives equal to zero will result in a minimum S_r. If this is done, the equations can be expressed as,

$$0 = \sum y_i - \sum a_0 - \sum a_1 x_i$$

$$0 = \sum x_i y_i - \sum a_0 x_i - \sum a_1 x_i^2$$

Upon rearranging,

$$n \quad a_0 + \left(\sum x_i\right) a_1 = \sum y_i$$

$$\left(\sum x_i\right) a_0 + \left(\sum x_i^2\right) a_1 = \sum x_i y_i$$

$$a_1 = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - \left(\sum x_i\right)^2}$$

$a_0 = \bar{y} - a_1 \bar{x}$ where $\bar{y}$ and $\bar{x}$ are the means of y and x, respectively.

Problem - Fit a straight line to the values in Table given below,

x_i	y_i
10	25
20	70
30	380
40	550
50	610
60	1,220
70	830
80	1,450

Solution – Following values needs to be calculated-

i	x_i	y_i	x_i^2	$x_i y_i$
1	10	25	100	250
2	20	70	400	1,400
3	30	380	900	11,400
4	40	550	1,600	22,000
5	50	610	2,500	30,500
6	60	1,220	3,600	73,200
7	70	830	4,900	58,100
8	80	1,450	6,400	116,000
Σ	360	5,135	20,400	312,850

The means can be computed as

$$\bar{x} = \frac{360}{8} = 45 \qquad \bar{y} = \frac{5{,}135}{8} = 641.875$$

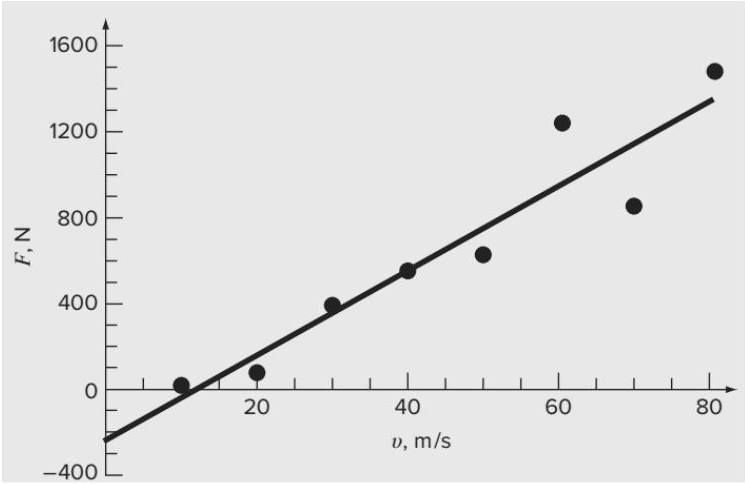
The slope and the intercept can then be calculated with Eqs. (14.15) and (14.16) as

$$a_1 = \frac{8(312{,}850) - 360(5{,}135)}{8(20{,}400) - (360)^2} = 19.47024$$

$$a_0 = 641.875 - 19.47024(45) = -234.2857$$

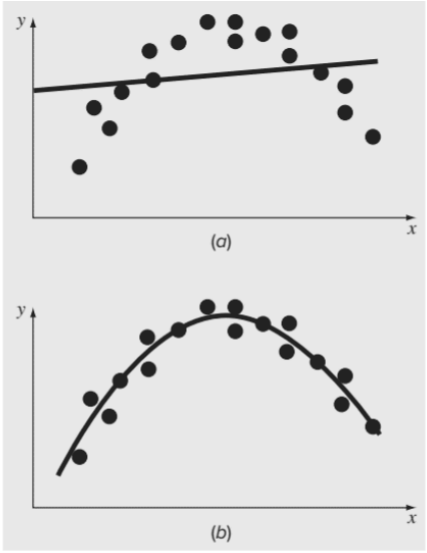
Using force and velocity in place of y and x, the least-squares fit is

$$F = -234.2857 + 19.47024v$$



In MALTAB, first define X vector and Y vector. Then type cftools (or open curve fitting tool from app menu). Explore the linear/polynomial of various degrees and observe the goodness of fit and r values.

Polynomial Regression-



The least-squares procedure can be readily extended to fit the data to a higher-order polynomial. For example, suppose that we fit a second-order polynomial or quadratic:

$$y = a_0 + a_1 x + a_2 x^2 + e$$

For this case the sum of the squares of the residuals is

$$S_r = \sum_{i=1}^{n} \left(y_i - a_0 - a_1 x_i - a_2 x_i^2\right)^2 \tag{15.2}$$

To generate the least-squares fit, we take the derivative of Eq. (15.2) with respect to each of the unknown coefficients of the polynomial, as in

$$\frac{\partial S_r}{\partial a_0} = -2 \sum \left(y_i - a_0 - a_1 x_i - a_2 x_i^2\right)$$

$$\frac{\partial S_r}{\partial a_1} = -2 \sum x_i \left(y_i - a_0 - a_1 x_i - a_2 x_i^2\right)$$

$$\frac{\partial S_r}{\partial a_2} = -2 \sum x_i^2 \left(y_i - a_0 - a_1 x_i - a_2 x_i^2\right)$$

These equations can be set equal to zero and rearranged to develop the following set of normal equations:

$$(n)a_0 + (\sum x_i)a_1 + (\sum x_i^2)a_2 = \sum y_i$$

$$(\sum x_i)a_0 + (\sum x_i^2)a_1 + (\sum x_i^3)a_2 = \sum x_i y_i$$

$$(\sum x_i^2)a_0 + (\sum x_i^3)a_1 + (\sum x_i^4)a_2 = \sum x_i^2 y_i$$

Polynomial Regression

Problem Statement. Fit a second-order polynomial to the data in the first two columns of Table 15.1.

TABLE 15.1 Computations for an error analysis of the quadratic least-squares fit.

x_i	y_i	$(y_i - \bar{y})^2$	$(y_i - a_0 - a_1 x_i - a_2 x_i^2)^2$
0	2.1	544.44	0.14332
1	7.7	314.47	1.00286
2	13.6	140.03	1.08160
3	27.2	3.12	0.80487
4	40.9	239.22	0.61959
5	61.1	1272.11	0.09434
$\sum$	152.6	2513.39	3.74657

Solution. The following can be computed from the data:

$m = 2$ $\quad \sum x_i = 15$ $\quad \sum x_i^4 = 979$

$n = 6$ $\quad \sum y_i = 152.6$ $\quad \sum x_i y_i = 585.6$

$\bar{x} = 2.5$ $\quad \sum x_i^2 = 55$ $\quad \sum x_i^2 y_i = 2488.8$

$\bar{y} = 25.433$ $\quad \sum x_i^3 = 225$

Therefore, the simultaneous linear equations are

$$\begin{bmatrix} 6 & 15 & 55 \\ 15 & 55 & 225 \\ 55 & 225 & 979 \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \end{Bmatrix} = \begin{Bmatrix} 152.6 \\ 585.6 \\ 2488.8 \end{Bmatrix}$$

```
a =
    2.4786
    2.3593
    1.8607
```

Therefore, the least-squares quadratic equation for this case is

$$y = 2.4786 + 2.3593x + 1.8607x^2$$

- The sum of squares of residuals, also called the residual sum of squares:

$$SS_{\text{res}} = \sum_i (y_i - f_i)^2 = \sum_i e_i^2$$

- The total sum of squares (proportional to the variance of the data):

$$SS_{\text{tot}} = \sum_i (y_i - \bar{y})^2$$

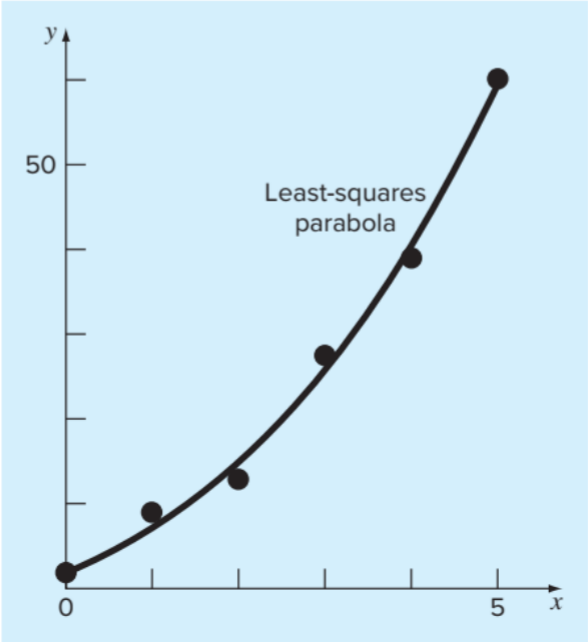
The most general definition of the coefficient of determination is

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}$$

The coefficient of determination is

$$r^2 = \frac{2513.39 - 3.74657}{2513.39} = 0.99851$$

and the correlation coefficient is $r = 0.99925$



MATLAB routine for finding curve fitting (using polyfit function) –

```
clear all; close all; clc;
X = 10:10:80;
Y = [ 25 70 380 550 610 1220 830 1450];
p = polyfit(X,Y,1); % for linear regression
% p = polyfit(X,Y,2) % for qudratic regression
% p = polyfit(X,Y,3) % for cubic regression
scatter(X,Y,'r*');
% make straight line using coefficients found using polyfit function
Y1 = p(2) + p(1)*X;
hold on;
plot(X,Y1,'-b');
```

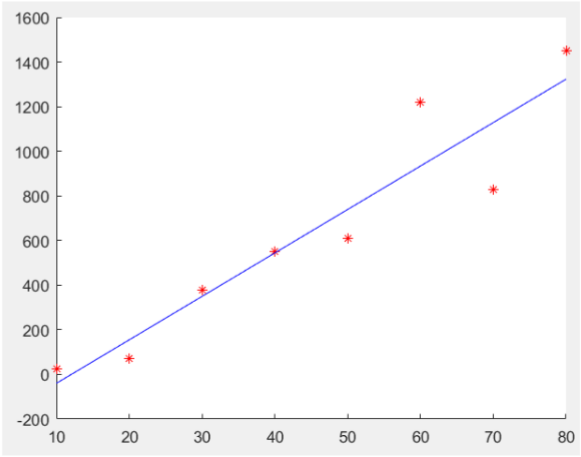


Figure – linear regression of the given data

Alternatively, you can use inbuilt curve fitting tool in MATLAB.

Introducing the Curve Fitting App

You can fit curves and surfaces to data and view plots with the Curve Fitting app.

- Create, plot, and compare multiple fits.
- Use linear or nonlinear regression, interpolation, smoothing, and custom equations.
- View goodness-of-fit statistics, display confidence intervals and residuals, remove outliers, and assess fits with validation data.
- Automatically generate code to fit and plot curves and surfaces, or export fits to the workspace for further analysis.

Fit a Curve

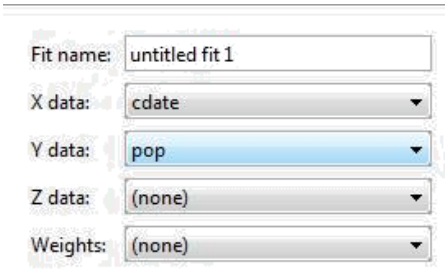
1. Load some example data at the MATLAB® command line:

   load census.mat

2. Open the Curve Fitting app by entering:

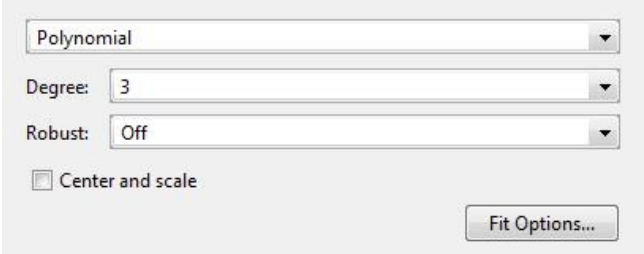
   cftool

   Alternatively, click **Curve Fitting** on the **Apps** tab.
3. Select **X data** and **Y data**. For details, see Selecting Data to Fit in Curve Fitting App.

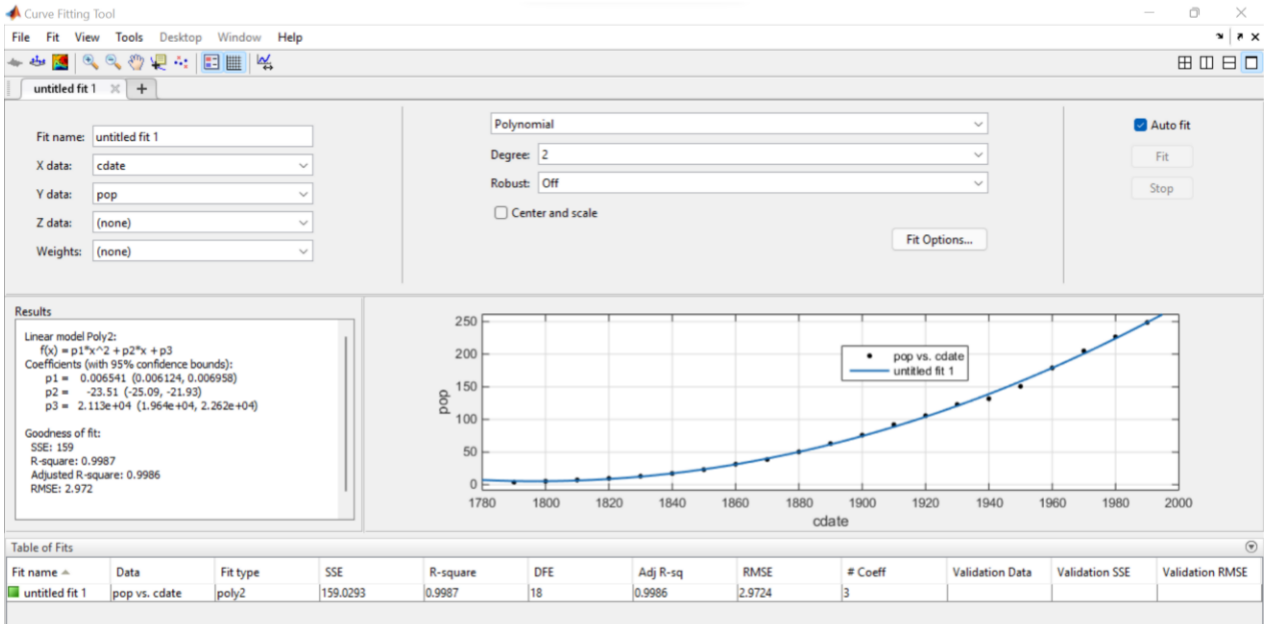


The Curve Fitting app creates a default polynomial fit to the data.

4. Try different fit options. For example, change the polynomial **Degree** to 3 to fit a cubic polynomial.



5. Select a different model type from the fit category list, e.g., **Smoothing Spline**. For information about models you can fit, see Model Types for Curves and Surfaces.



Assignment -

One numerical as suggested by Faculty.

Sample numerical.

Q. Suppose we have xdata and ydata points observed in any experiment as follows –

x	0	2	4	6	9	11	12	15	17	19
y	5	6	7	6	9	8	8	10	12	12

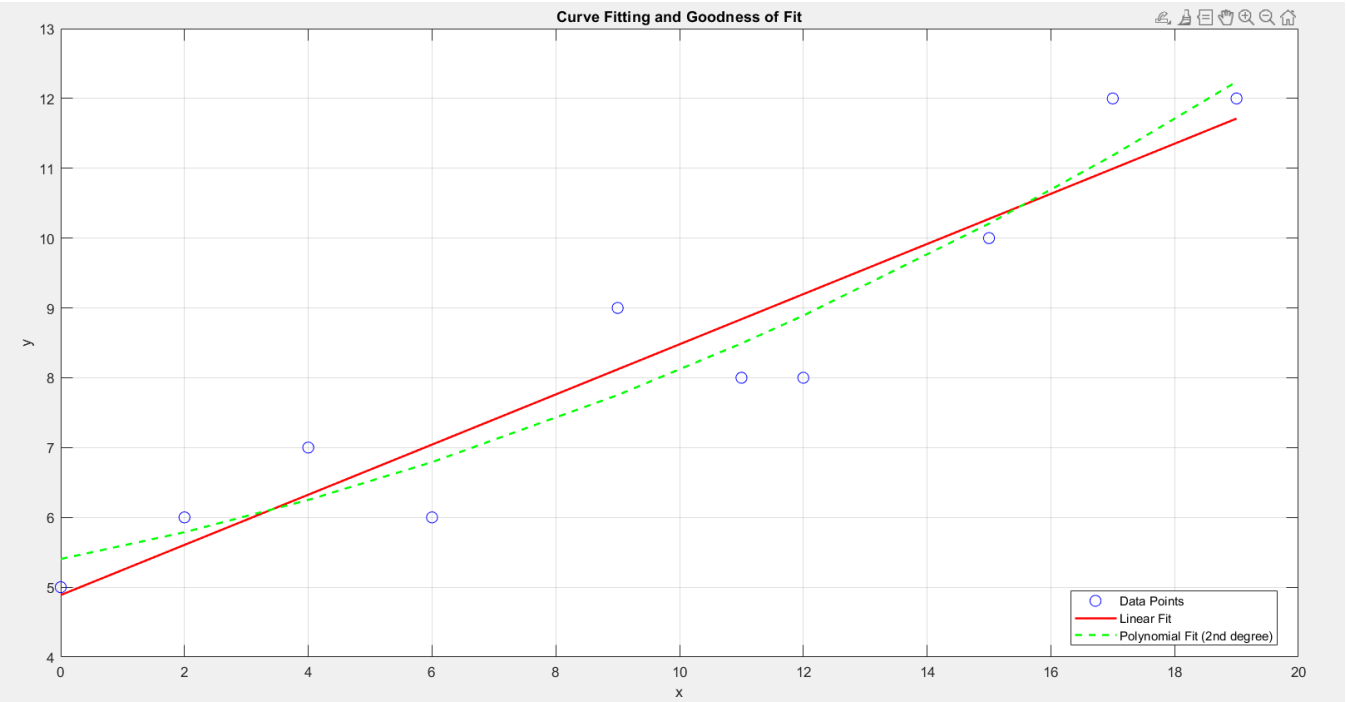
Using hand calculations, show the linear regression for the above data also using curve fitting toolbox in MATLAB, evaluate the goodness of fits with linear and polynomial fit with 2nd degree.

MATLAB Code:

```
% Given data points
x = [0, 2, 4, 6, 9, 11, 12, 15, 17, 19];
y = [5, 6, 7, 6, 9, 8, 8, 10, 12, 12];

% Linear Regression by Hand Calculation
n = length(x);
x_bar = mean(x);
y_bar = mean(y);

SS_xy = sum((x - x_bar) .* (y - y_bar));
SS_xx = sum((x - x_bar) .^ 2)
```

```
m = SS_xy / SS_xx;
b = y_bar - m * x_bar;

fprintf('Linear Regression by Hand:\n');
fprintf('Slope (m) = %.4f\n', m);
fprintf('Y-Intercept (b) = %.4f\n', b);

% Linear Regression using polyfit (degree 1) in MATLAB
p_linear = polyfit(x, y, 1);

fprintf('\nLinear Regression using polyfit:\n');
fprintf('Slope (m) = %.4f\n', p_linear(1));
fprintf('Y-Intercept (b) = %.4f\n', p_linear(2));

% Polynomial Regression using polyfit (degree 2) in MATLAB
p_poly = polyfit(x, y, 2);

% Evaluate Goodness of Fit (R-squared) for Linear and Polynomial (2nd degree)
Models
y_fit_linear = polyval(p_linear, x);
y_fit_poly = polyval(p_poly, x);

SS_tot = sum((y - y_bar) .^ 2);
SS_res_linear = sum((y - y_fit_linear) .^ 2);
SS_res_poly = sum((y - y_fit_poly) .^ 2);

R2_linear = 1 - (SS_res_linear / SS_tot);
R2_poly = 1 - (SS_res_poly / SS_tot);

fprintf('\nGoodness of Fit (R-squared):\n');
fprintf('Linear Fit: R^2 = %.4f\n', R2_linear);
fprintf('Polynomial Fit (2nd degree): R^2 = %.4f\n', R2_poly);

% Plotting
figure;
plot(x, y, 'bo', 'MarkerSize', 8);
hold on;
plot(x, y_fit_linear, 'r-', 'LineWidth', 1.5);
plot(x, y_fit_poly, 'g--', 'LineWidth', 1.5);
xlabel('x');
ylabel('y');
title('Curve Fitting and Goodness of Fit');
legend('Data Points', 'Linear Fit', 'Polynomial Fit (2nd degree)', 'Location',
'best');
grid on;
```

Output:



>> Lab_6
Linear Regression by Hand:
Slope (m) = 0.3591
Y-Intercept (b) = 4.8881

Linear Regression using polyfit:
Slope (m) = 0.3591
Y-Intercept (b) = 4.8881

Goodness of Fit (R-squared):
Linear Fit: R^2 = 0.8929
Polynomial Fit (2nd degree): R^2 = 0.9121

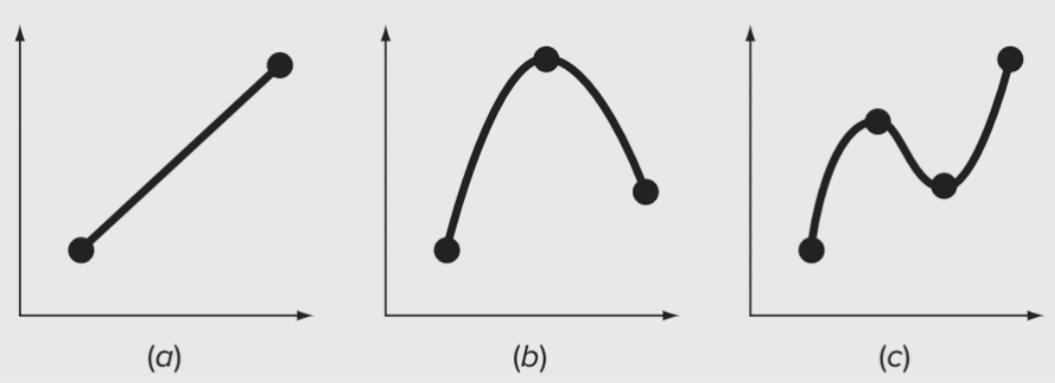
LAB 7: Polynomial Interpolation

Objective: To obtain the linear/polynomial interpolation using Newton/Lagrange's polynomial.

Theory: You will frequently have occasion to estimate intermediate values between precise data points. The most common method used for this purpose is polynomial interpolation. The general formula for an (n – 1)th-order polynomial can be written as-

$$f(x) = a_1 + a_2x + a_3x^2 + \cdots + a_nx^{n-1}$$

For n data points, there is one and only one polynomial of order (n – 1) that passes through all the points. For example, there is only one straight line (i.e., a first-order polynomial) that connects two points. Similarly, only one parabola connects a set of three points. Polynomial interpolation consists of determining the unique (n–1) order polynomial that fits n data points. This polynomial then provides a formula to compute intermediate values.



Examples of interpolating polynomials: (*a*) first-order (linear) connecting two points, (*b*) second-order (quadratic or parabolic) connecting three points, and (*c*) third-order (cubic) connecting four points.

A straightforward way for computing the coefficients, is based on the fact that n data points are required to determine the n coefficients.

Problem Statement. Suppose that we want to determine the coefficients of the parabola, $f(x) = p_1x^2 + p_2x + p_3$, that passes through the last three density values from Table 17.1:

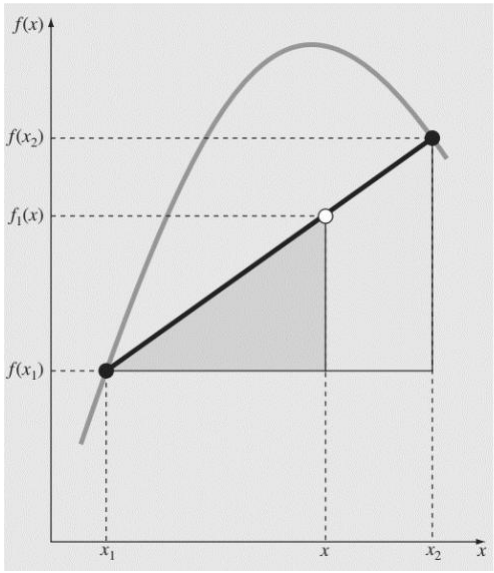
$$x_1 = 300 \quad f(x_1) = 0.616$$
$$x_2 = 400 \quad f(x_2) = 0.525$$
$$x_3 = 500 \quad f(x_3) = 0.457$$

This results in,

$$\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix} \begin{Bmatrix} p_1 \\ p_2 \\ p_3 \end{Bmatrix} = \begin{Bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \end{Bmatrix}$$

Such matrices are very ill-conditioned. That is, their solutions are very sensitive to round off errors. There are methods called, Newton's interpolating polynomial and Lagrange's interpolating polynomial, we can get rid of above problem.

The simplest form of interpolation is to connect two data points with a straight line. This technique, called linear interpolation, is depicted graphically in Fig.



Using similar triangles,

$$\frac{f_1(x) - f(x_1)}{x - x_1} = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

which can be rearranged to yield

$$f_1(x) = f(x_1) + \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_1)$$

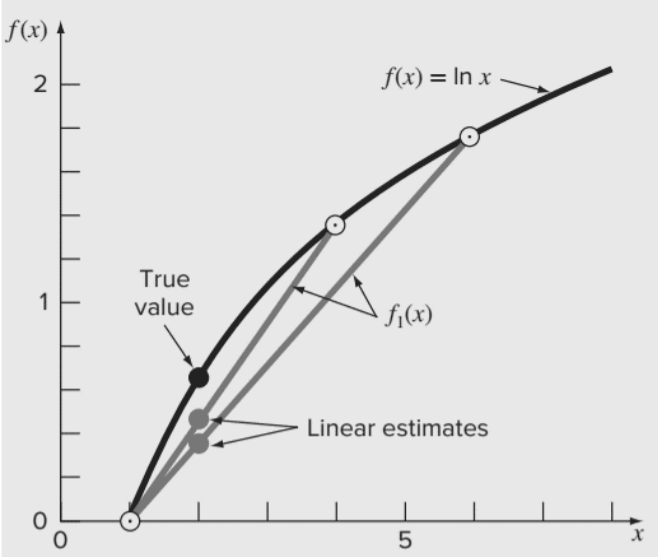
Problem Statement. Estimate the natural logarithm of 2 using linear interpolation. First, perform the computation by interpolating between ln 1 = 0 and ln 6 = 1.791759. Then, repeat the procedure, but use a smaller interval from ln 1 to ln 4 (1.386294). Note that the true value of ln 2 is 0.6931472.

Solution. We use Eq. (17.5) from $x_1 = 1$ to $x_2 = 6$ to give

$$f_1(2) = 0 + \frac{1.791759 - 0}{6 - 1}(2 - 1) = 0.3583519$$

which represents an error of $\varepsilon_t = 48.3\%$. Using the smaller interval from $x_1 = 1$ to $x_2 = 4$ yields

$$f_1(2) = 0 + \frac{1.386294 - 0}{4 - 1}(2 - 1) = 0.4620981$$



17.2.2 Quadratic Interpolation

The error in Example 17.2 resulted from approximating a curve with a straight line. Consequently, a strategy for improving the estimate is to introduce some curvature into the line connecting the points. If three data points are available, this can be accomplished with a second-order polynomial (also called a quadratic polynomial or a parabola). A particularly convenient form for this purpose is

$$f_2(x) = b_1 + b_2(x - x_1) + b_3(x - x_1)(x - x_2) \tag{17.6}$$

A simple procedure can be used to determine the values of the coefficients. For b_1, Eq. (17.6) with $x = x_1$ can be used to compute

$$b_1 = f(x_1) \tag{17.7}$$

Equation (17.7) can be substituted into Eq. (17.6), which can be evaluated at $x = x_2$ for

$$b_2 = \frac{f(x_2) - f(x_1)}{x_2 - x_1} \tag{17.8}$$

Finally, Eqs. (17.7) and (17.8) can be substituted into Eq. (17.6), which can be evaluated at $x = x_3$ and solved (after some algebraic manipulations) for

$$b_3 = \frac{\dfrac{f(x_3) - f(x_2)}{x_3 - x_2} - \dfrac{f(x_2) - f(x_1)}{x_2 - x_1}}{x_3 - x_1} \tag{17.9}$$

Problem Statement. Employ a second-order Newton polynomial to estimate ln 2 with the same three points used in Example 17.2:

$$x_1 = 1 \qquad f(x_1) = 0$$

$$x_2 = 4 \qquad f(x_2) = 1.386294$$

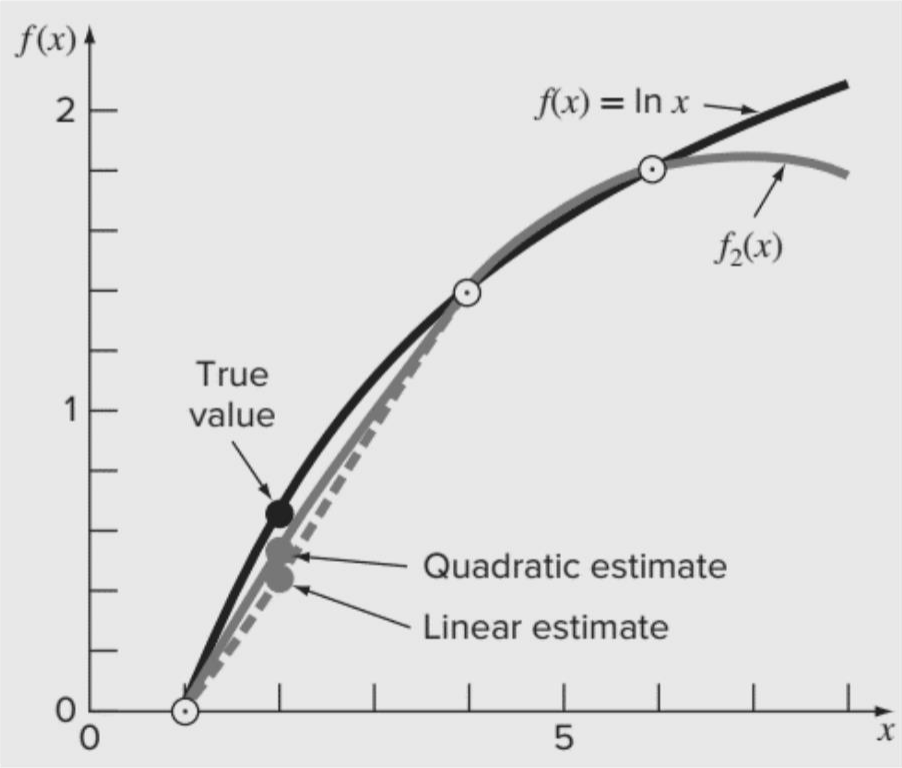
$$x_3 = 6 \qquad f(x_3) = 1.791759$$

Solution. Applying Eq. (17.7) yields

$$b_1 = 0$$

Equation (17.8) gives

$$b_2 = \frac{1.386294 - 0}{4 - 1} = 0.4620981$$



and Eq. (17.9) yields

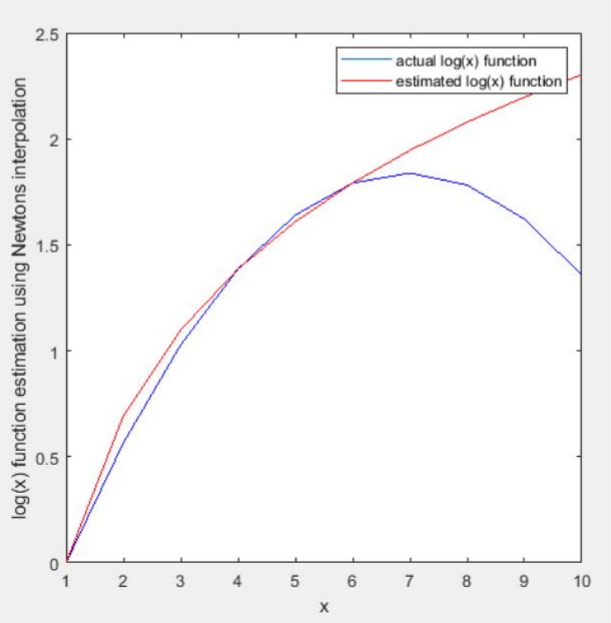
$$b_3 = \frac{\dfrac{1.791759 - 1.386294}{6 - 4} - 0.4620981}{6 - 1} = -0.0518731$$

Substituting these values into Eq. (17.6) yields the quadratic formula

$$f_2(x) = 0 + 0.4620981(x - 1) - 0.0518731(x - 1)(x - 4)$$

MATLAB Routine –

```
clc;clear all; close all;
x=1:1:10;
y = (-0.0518731*x.^2) + (0.7214635)*(x) - 0.6695904;
y1=log(x);
plot (x,y,'b');
hold on;
plot(x,y1,'r')
xlabel('x'); ylabel('log(x) function estimation using Newtons interpolation')
legend('actual log(x) function','estimated log(x) function')
```



LAGRANGE INTERPOLATING POLYNOMIAL

Suppose we formulate a linear interpolating polynomial as the weighted average of the two values that we are connecting by a straight line:

$$f(x) = L_1 f(x_1) + L_2 f(x_2) \tag{17.19}$$

where the L's are the weighting coefficients. It is logical that the first weighting coefficient is the straight line that is equal to 1 at x_1 and 0 at x_2:

$$L_1 = \frac{x - x_2}{x_1 - x_2}$$

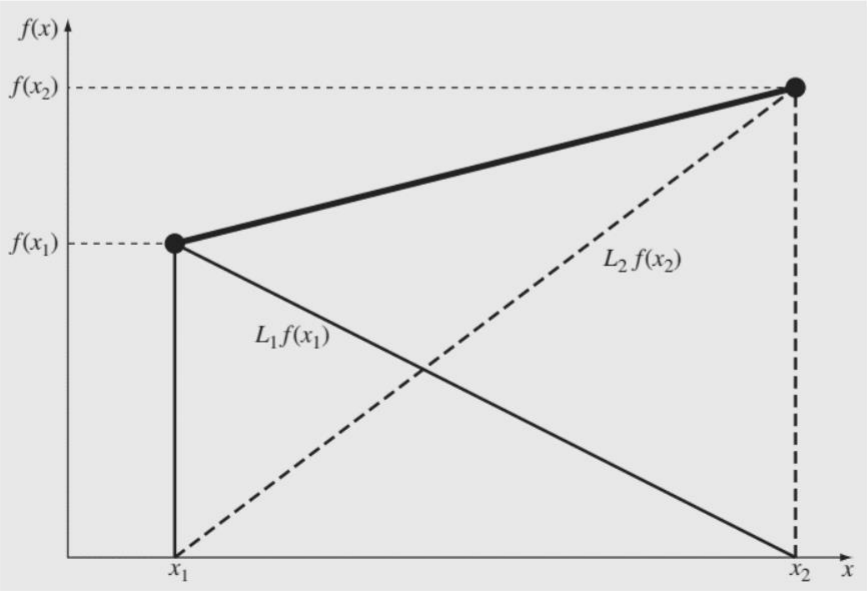
Similarly, the second coefficient is the straight line that is equal to 1 at x_2 and 0 at x_1:

$$L_2 = \frac{x - x_1}{x_2 - x_1}$$

Substituting these coefficients into Eq. (17.19) yields the straight line that connects the points (Fig. 17.8):

$$f_1(x) = \frac{x - x_2}{x_1 - x_2} f(x_1) + \frac{x - x_1}{x_2 - x_1} f(x_2) \tag{17.20}$$

where the nomenclature $f_1(x)$ designates that this is a first-order polynomial. Equation (17.20) is referred to as the *linear Lagrange interpolating polynomial.*



A visual depiction of the rationale behind Lagrange interpolating polynomials. The figure shows the first-order case. Each of the two terms of Eq. (17.20) passes through one of the points and is zero at the other. The summation of the two terms must, therefore, be the unique straight line that connects the two points.

The same strategy can be employed to fit a parabola through three points. For this case three parabolas would be used with each one passing through one of the points and equaling zero at the other two. Their sum would then represent the unique parabola that connects the three points. Such a second-order Lagrange interpolating polynomial can be written as

$$f_2(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} f(x_1) + \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} f(x_2) + \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} f(x_3) \tag{17.21}$$

Notice how the first term is equal to $f(x_1)$ at x_1 and is equal to zero at x_2 and x_3. The other terms work in a similar fashion.

where

$$L_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}$$

where n = the number of data points and $\prod$ designates the "product of."

Problem Statement. Use a Lagrange interpolating polynomial of the first and second order to evaluate the density of unused motor oil at $T = 15$ °C based on the following data:

$x_1 = 0 \qquad f(x_1) = 3.85$

$x_2 = 20 \qquad f(x_2) = 0.800$

$x_3 = 40 \qquad f(x_3) = 0.212$

Solution. The first-order polynomial [Eq. (17.20)] can be used to obtain the estimate at $x = 15$:

$$f_1(x) = \frac{15 - 20}{0 - 20} 3.85 + \frac{15 - 0}{20 - 0} 0.800 = 1.5625$$

In a similar fashion, the second-order polynomial is developed as [Eq. (17.21)]

$$f_2(x) = \frac{(15 - 20)(15 - 40)}{(0 - 20)(0 - 40)} 3.85 + \frac{(15 - 0)(15 - 40)}{(20 - 0)(20 - 40)} 0.800 + \frac{(15 - 0)(15 - 20)}{(40 - 0)(40 - 20)} 0.212 = 1.3316875$$

Assignment -

One numerical as suggested by Faculty.

Sample numerical.

17.4 Given the data

x	1	2	2.5	3	4	5
$f(x)$	0	5	7	6.5	2	0

(a) Calculate $f(3.4)$ using Newton's interpolating polynomials of order 1 through 3. Choose the sequence of the points for your estimates to attain the best possible accuracy. That is, the points should be centered around and as close as possible to the unknown.

(b) Repeat **(a)** but use the Lagrange polynomial.

MATLAB CODE:

```
% Given data
x = [1 2 2.5 3 4 5];
fx = [0 5 7 6.5 2 0];

% Point for interpolation
x0 = 3.4;

% Newton's interpolating polynomial of order 1 (linear)
n1 = 1;
coeffs1 = divided_diff_coeffs(x(1:n1+1), fx(1:n1+1));
P1 = newton_interpolation(x(1:n1+1), coeffs1, x0);

% Newton's interpolating polynomial of order 2 (quadratic)
n2 = 2;
coeffs2 = divided_diff_coeffs(x(1:n2+1), fx(1:n2+1));
P2 = newton_interpolation(x(1:n2+1), coeffs2, x0);

% Newton's interpolating polynomial of order 3 (cubic)
n3 = 3;
coeffs3 = divided_diff_coeffs(x(1:n3+1), fx(1:n3+1));
P3 = newton_interpolation(x(1:n3+1), coeffs3, x0);

% Output the results
disp('Using Newton Interpolation:');
disp(['f(3.4) (order 1): ', num2str(P1)]);
disp(['f(3.4) (order 2): ', num2str(P2)]);
```

```
disp(['f(3.4) (order 3): ', num2str(P3)]);

function coeffs = divided_diff_coeffs(x, fx)
    % Compute divided difference coefficients
    n = length(x) - 1;
    coeffs = fx;
    for j = 1:n
        for i = n:-1:j+1
            coeffs(i) = (coeffs(i) - coeffs(i-1)) / (x(i) - x(i-j));
        end
    end
end

function P = newton_interpolation(x, coeffs, x0)
    % Evaluate Newton's interpolating polynomial at x0
    n = length(x) - 1;
    P = coeffs(n+1);
    for i = n:-1:1
        P = coeffs(i) + (x0 - x(i)) * P;
    end
end

% Lagrange interpolation to estimate f(3.4)
L = lagrange_interpolation(x, fx, x0);

% Output the result
disp('Using Lagrange Interpolation:');
disp(['f(3.4): ', num2str(L)]);

function L = lagrange_interpolation(x, fx, x0)
    % Lagrange interpolation to evaluate f(x0)
    n = length(x);
    L = 0;
    for j = 1:n
        lj = 1;
        for m = 1:n
            if m ~= j
                lj = lj * (x0 - x(m)) / (x(j) - x(m));
            end
        end
        L = L + fx(j) * lj;
    end
end
```

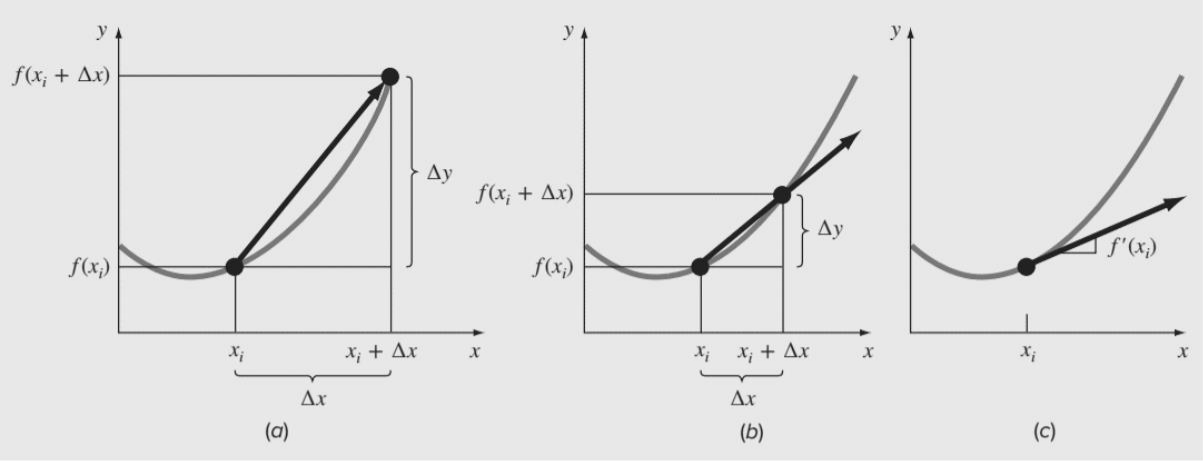
Output:

```
>> Lab_7
Using Newton Interpolation:
f(3.4) (order 1): 12
f(3.4) (order 2): 35.52
f(3.4) (order 3): 29.416
Using Lagrange Interpolation:
f(3.4): 4.8248
```

LAB 8: Numerical Differentiation

Objective: To numerically obtain the derivative of any function.

Theory: Calculus is the mathematics of change. Because engineers and scientists must continuously deal with systems and processes that change, calculus is an essential tool of our profession. Mathematically, the derivative, which serves as the fundamental vehicle for differentiation, represents the rate of change of a dependent variable with respect to an independent variable.



The graphical definition of a derivative: as Δx approaches zero in going from (*a*) to (*c*), the difference approximation becomes a derivative.

$$\frac{\Delta y}{\Delta x} = \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x}$$

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x}$$

Law	Equation	Physical Area	Gradient	Flux	Proportionality
Fourier's law	$q = -k\frac{dT}{dx}$	Heat conduction	Temperature	Heat flux	Thermal Conductivity
Fick's law	$J = -D\frac{dc}{dx}$	Mass diffusion	Concentration	Mass flux	Diffusivity
Darcy's law	$q = -k\frac{dh}{dx}$	Flow through porous media	Head	Flow flux	Hydraulic Conductivity
Ohm's law	$J = -\sigma\frac{dV}{dx}$	Current flow	Voltage	Current flux	Electrical Conductivity
Newton's viscosity law	$\tau = \mu\frac{du}{dx}$	Fluids	Velocity	Shear Stress	Dynamic Viscosity
Hooke's law	$\sigma = E\frac{\Delta L}{L}$	Elasticity	Deformation	Stress	Young's Modulus

Methodology to find derivative numerically-

Using Taylor series expansion,

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \cdots$$

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{f''(x_i)}{2!}h + O(h^2)$$

Also,

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} + O(h)$$

Similarly,

$$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2} + O(h)$$

Table for different order of accuracy-

First Derivative	Error
$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h}$	$O(h)$
$f'(x_i) = \frac{-f(x_{i+2}) + 4f(x_{i+1}) - 3f(x_i)}{2h}$	$O(h^2)$
Second Derivative	
$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2}$	$O(h)$
$f''(x_i) = \frac{-f(x_{i+3}) + 4f(x_{i+2}) - 5f(x_{i+1}) + 2f(x_i)}{h^2}$	$O(h^2)$
Third Derivative	
$f'''(x_i) = \frac{f(x_{i+3}) - 3f(x_{i+2}) + 3f(x_{i+1}) - f(x_i)}{h^3}$	$O(h)$
$f'''(x_i) = \frac{-3f(x_{i+4}) + 14f(x_{i+3}) - 24f(x_{i+2}) + 18f(x_{i+1}) - 5f(x_i)}{2h^3}$	$O(h^2)$
Fourth Derivative	
$f''''(x_i) = \frac{f(x_{i+4}) - 4f(x_{i+3}) + 6f(x_{i+2}) - 4f(x_{i+1}) + f(x_i)}{h^4}$	$O(h)$
$f''''(x_i) = \frac{-2f(x_{i+5}) + 11f(x_{i+4}) - 24f(x_{i+3}) + 26f(x_{i+2}) - 14f(x_{i+1}) + 3f(x_i)}{h^4}$	$O(h^2)$

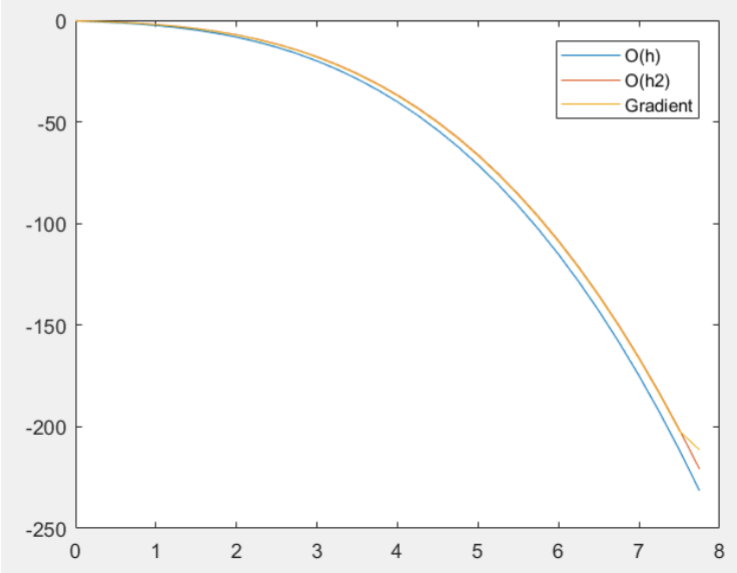
Problem Statement.

Estimate the derivative of for *O(h), O(h2)* for forward differencing-

$$f(x) = -0.1x^4 - 0.15x^3 - 0.5x^2 - 0.25x + 1.2$$

at $x = 0.5$ using finite-differences and a step size of $h = 0.25$.

```
clear all;clc;
xl = input('Enter lower limit of X: ');xu = input('Enter upper limit of X: ');
h = input('Enter step size of X: ');n = xu-xl/h;
i = 1;
x(1) = xl;
while x<xu
    a = x(i)+ 2*h;
    b = x(i)+ h;
    c = x(i);
    df1(i) = (f(b)-f(c))/h;
    df2(i) = (-f(a) + 4*f(b) - 3*f(c))/(2*h);
    y(i) = f(c);
    xp(i) = x(i);
    x(i+1) = x(i) +h;
    i = i+1;
end
dfg = gradient(y,0.25);
plot(xp(),df1())
hold on
plot(xp(),df2())
hold on
plot(xp(),dfg())
legend('O(h)','O(h2)','Gradient')
function fun = f(x)
    fun = -0.1*(x^4) -0.15*(x^3) -0.5*(x^2) -0.25*x +1.2;
end
```



Assignment -

One numerical as suggested by Faculty.

Sample numerical.

Develop an M-file to obtain first-derivative estimates for unequally spaced data. Test it with the following data:

x	0.6	1.5	1.6	2.5	3.5
$f(x)$	0.9036	0.3734	0.3261	0.08422	0.01596

where $f(x) = 5e^{-2x}\ x$. Compare your results with the true derivatives.

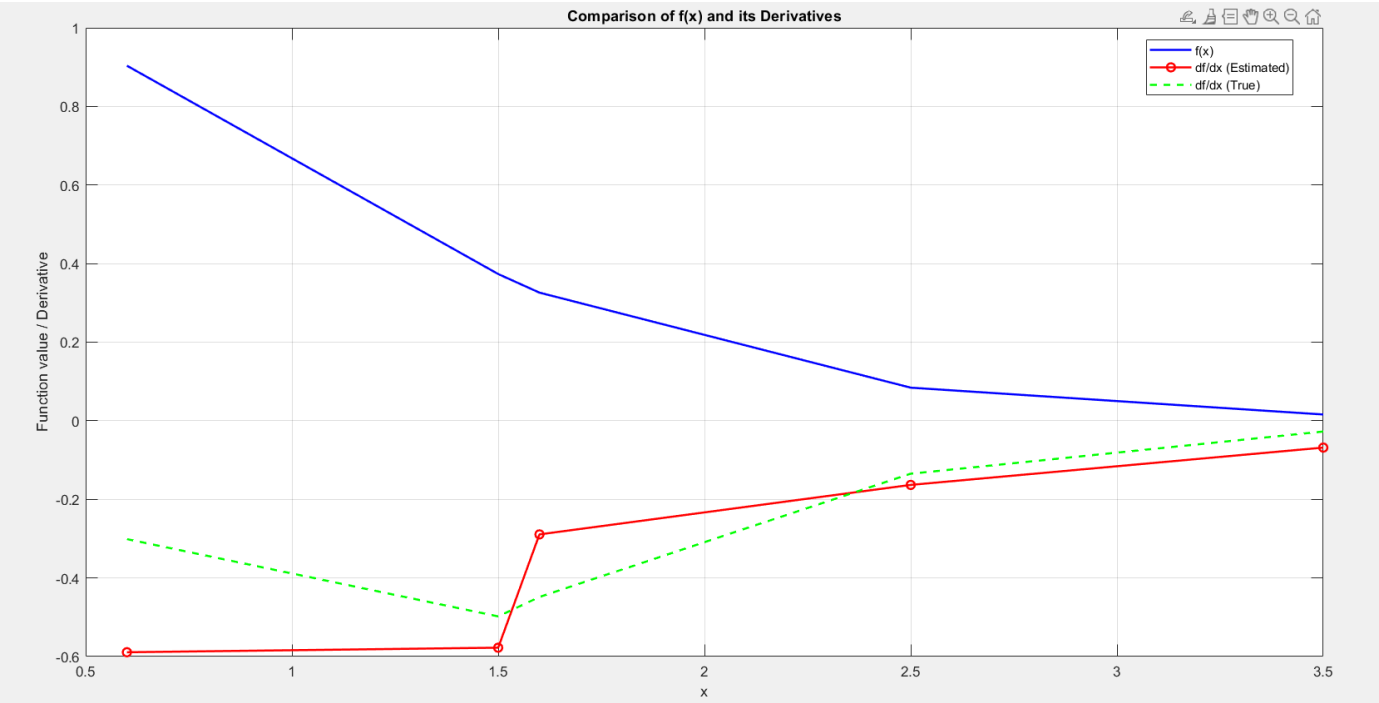
MATLAB Code:

```
% Given data
x = [0.6 1.5 1.6 2.5 3.5];
f = 5 * exp(-2*x) .* x; % Compute f(x) = 5*e^(-2*x)*x

% True derivative of f(x)
true_derivative = @(x) 5 * exp(-2*x) .* (1 - 2*x);

% Compute first-derivative estimates
n = length(x);
df_estimated = zeros(size(x));

for i = 1:n
    if i == 1
        % Forward difference at the first point
        h = x(i+1) - x(i);
        df_estimated(i) = (f(i+1) - f(i)) / h;
    elseif i == n
        % Backward difference at the last point
        h = x(i) - x(i-1);
        df_estimated(i) = (f(i) - f(i-1)) / h;
    else
        % Centered difference at interior points
        h1 = x(i) - x(i-1);
        h2 = x(i+1) - x(i);
        df_estimated(i) = (f(i+1) - f(i-1)) / (h1 + h2);
```

```
    end
end

% Compare estimated derivatives with true derivatives
disp('x       f(x)           df/dx (Estimated)     df/dx (True)');
disp('------------------------------------------------------------');
for i = 1:n
    fprintf('%.2f    %.4f          %.4f               %.4f\n', x(i), f(i),
df_estimated(i), true_derivative(x(i)));
end
```

Output:

```
>> Lab_8
x      f(x)          df/dx (Estimated)     df/dx (True)
------------------------------------------------------------
0.60    0.9036          -0.5891               -0.3012
1.50    0.3734          -0.5775               -0.4979
1.60    0.3261          -0.2892               -0.4484
2.50    0.0842          -0.1632               -0.1348
3.50    0.0160          -0.0683               -0.0274
```



LAB 9: Numerical Integration

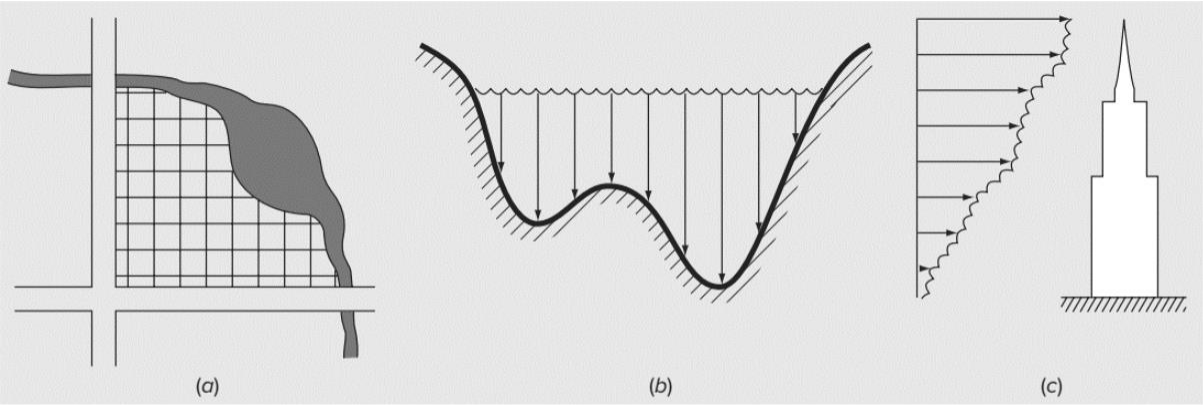
Objective: To numerically obtain the integration of any given function.

Theory: Mathematically, definite integration is represented by

$$I = \int_a^b f(x)\,dx$$

which stands for the integral of the function $f(x)$ with respect to the independent variable x, evaluated between the limits $x = a$ to $x = b$.

The meaning of integration is the total value, or summation, of a f (x) dx over the range x = a to b. In fact, the symbol ∫ is actually a stylized capital S that is intended to signify the close connection between integration and summation.



Examples of how integration is used to evaluate areas in engineering and scientific applications. (*a*) A surveyor might need to know the area of a field bounded by a meandering stream and two roads. (*b*) A hydrologist might need to know the cross-sectional area of a river. (*c*) A structural engineer might need to determine the net force due to a nonuniform wind blowing against the side of a skyscraper.

Method to numerically evaluate integration-

TRAPEZOIDAL RULE –

The *Newton-Cotes formulas* are the most common numerical integration schemes. They are based on the strategy of replacing a complicated function or tabulated data with a polynomial that is easy to integrate:

$$I = \int_a^b f(x)\,dx \cong \int_a^b f_n(x)\,dx$$

where $f_n(x)$ = a polynomial of the form

$$f_n(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + a_n x^n$$

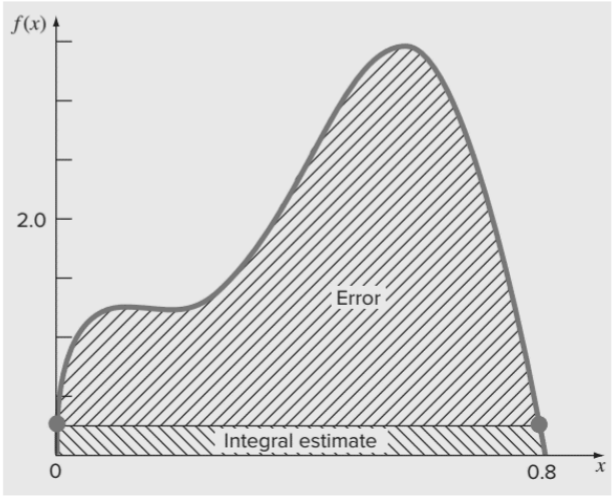
The trapezoidal rule is the first of the Newton-Cotes closed integration formulas.

$$I = \int_a^b \left[f(a) + \frac{f(b) - f(a)}{b - a} (x - a) \right] dx$$

The result of the integration is

$$I = (b - a)\frac{f(a) + f(b)}{2} \qquad I = \text{width} \times \text{average height}$$

which is called the *trapezoidal rule*.

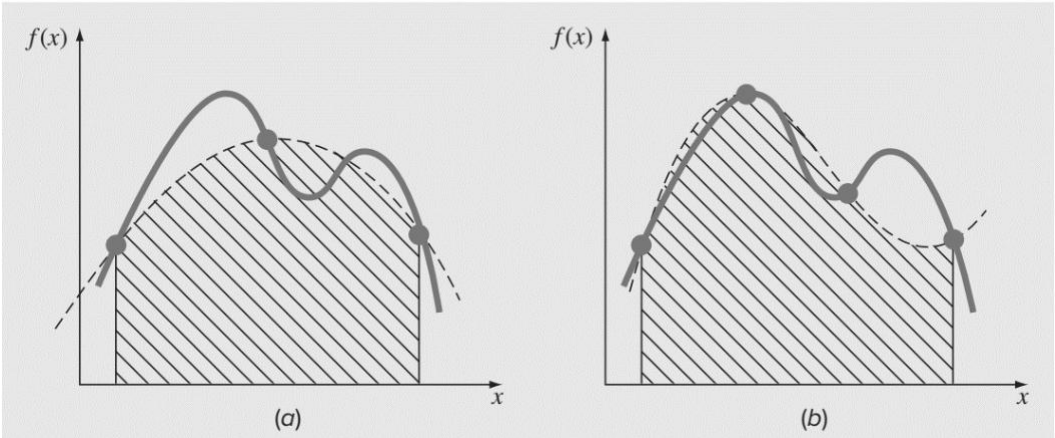


Error estimate

Simpson's 1/3 Rule –

Simpson's 1/3 rule corresponds to the case where the polynomial

(*a*) Graphical depiction of Simpson's 1/3 rule: It consists of taking the area under a parabola connecting three points. (*b*) Graphical depiction of Simpson's 3/8 rule: It consists of taking the area under a cubic equation connecting four points.



$$I = \int_{x_0}^{x_2} \left[\frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} f(x_0) + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} f(x_1) \right.$$

$$\left. + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} f(x_2) \right] dx$$

$$I = \frac{h}{3} [f(x_0) + 4f(x_1) + f(x_2)]$$

where, for this case, $h = (b - a)/2$. This equation is known as *Simpson's 1/3 rule*.

$$I = (b - a) \frac{f(x_0) + 4f(x_1) + f(x_2)}{6}$$

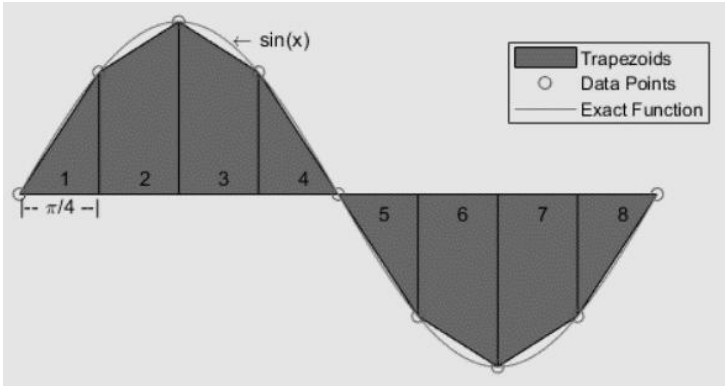
where $a = x_0$, $b = x_2$, and x_1 = the point midway between a and b, which is given by $(a + b)/2$.

Problem Statement. Use Eq. (19.23) to integrate

$$f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$$

from $a = 0$ to $b = 0.8$ using analytical, Trapezoidal and 1/3 simpson rule.

Segments (n)	Points	Name	Formula	Truncation Error
1	2	Trapezoidal rule	$(b - a) \frac{f(x_0) + f(x_1)}{2}$	$-(1/12)h^3 f''(\xi)$
2	3	Simpson's 1/3 rule	$(b - a) \frac{f(x_0) + 4f(x_1) + f(x_2)}{6}$	$-(1/90)h^5 f^{(4)}(\xi)$
3	4	Simpson's 3/8 rule	$(b - a) \frac{f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)}{8}$	$-(3/80)\, h^5 f^{(4)}(\xi)$



`trapz` performs numerical integration via the trapezoidal method. This method approximates the integration over an interval by breaking the area down into trapezoids with more easily computable areas. For example, here is a trapezoidal integration of the sine function using eight evenly-spaced trapezoids:

For an integration with `N+1` evenly spaced points, the approximation is

$$\int_a^b f(x)dx \approx \frac{b-a}{2N}\sum_{n=1}^{N}(f(x_n)+f(x_{n+1}))$$

$$= \frac{b-a}{2N}[f(x_1)+2f(x_2)+\ldots+2f(x_N)+f(x_{N+1})]\ ,$$

where the spacing between each point is equal to the scalar value $\frac{b-a}{N}$. By default MATLAB® uses a spacing of 1.

If the spacing between the `N+1` points is not constant, then the formula generalizes to

$$\int_a^b f(x)dx \approx \frac{1}{2}\sum_{n=1}^{N}(x_{n+1}-x_n)[f(x_n)+f(x_{n+1})]\ ,$$

where $a = x_1 < x_2 < \ldots < x_N < x_{N+1} = b$, and $(x_{n+1}-x_n)$ is the spacing between each consecutive pair of points.

MATLAB code for evaluating area under the sin curve.

```
clc;clear all; close all;
X = 0:pi/100:pi;
Y = sin(X);
Q = trapz(X,Y);
```

Assignment -

One numerical as suggested by Faculty.

Sample numerical.

19.4 Evaluate the following integral:

$$\int_{-2}^{4}(1-x-4x^3+2x^5)\,dx$$

(a) analytically, **(b)** single application of the trapezoidal rule, **(c)** composite trapezoidal rule with $n = 2$ and 4, **(d)** single application of Simpson's 1/3 rule, **(e)** Simpson's 3/8 rule, and

MATLAB Code:

```
% Define the integrand function
f = @(x) 1 - x - 4*x.^3 + 2*x.^5;

% Integration limits
a = -2;
b = 4;

% Analytical solution
I = integral(f,a,b);

% Trapezoidal rule (single application)
trap_approx = (b - a) * (f(a) + f(b)) / 2;

% Composite Trapezoidal rule with n = 2
n = 2;
h = (b - a) / n;
x = a:h:b;
trap_composite_approx_n2 = h * (sum(f(x)) - (f(a) + f(b)) / 2);

% Composite Trapezoidal rule with n = 4
n = 4;
h = (b - a) / n;
x = a:h:b;
trap_composite_approx_n4 = h * (sum(f(x)) - (f(a) + f(b)) / 2);

% Simpson's 1/3 rule (single application)
simpson13_approx = (b - a) * (f(a) + 4*f((a+b)/2) + f(b)) / 6;

% Simpson's 3/8 rule (single application)

simpson38_approx = (b - a) * (f(a) + 3*f((2*a+b)/3) + 3*f((a+2*b)/3) + f(b)) / 8;

% Display results
fprintf('Analytical solution: %f\n', I);
fprintf('Trapezoidal rule (single application): %f\n', trap_approx);
fprintf('Composite Trapezoidal rule with n = 2: %f\n', trap_composite_approx_n2);
fprintf('Composite Trapezoidal rule with n = 4: %f\n', trap_composite_approx_n4);
fprintf('Simpson''s 1/3 rule (single application): %f\n', simpson13_approx);
fprintf('Simpson''s 3/8 rule (single application): %f\n', simpson38_approx);
```

Output:

```
>> Lab_9
Analytical solution: 1104.000000
Trapezoidal rule (single application): 5280.000000
Composite Trapezoidal rule with n = 2: 2634.000000
Composite Trapezoidal rule with n = 4: 1516.875000
Simpson's 1/3 rule (single application): 1752.000000
Simpson's 3/8 rule (single application): 1392.000000
```

LAB 10: Ordinary Differential Equation

Objective: To numerically obtain the governing ODE of any physical system and solve it numerically.

Theory: equations, which are composed of an unknown function and its derivatives, are called differential equations. When the function involves one independent variable, the equation is called an *ordinary differential equation* (or ODE). Consider an ordinary differential equations of the form,

$$\frac{dy}{dt} = f(t, y)$$

Euler's Method –

New value = old value + slope × step size

in mathematical terms,

$$y_{i+1} = y_i + \phi h$$

where the slope ϕ is called an increment function. According to this equation, the slope estimate of ϕ is used to extrapolate from an old value yi to a new value yi+1 over a distance h. This formula can be applied step by step to trace out the trajectory of the solution into the future.

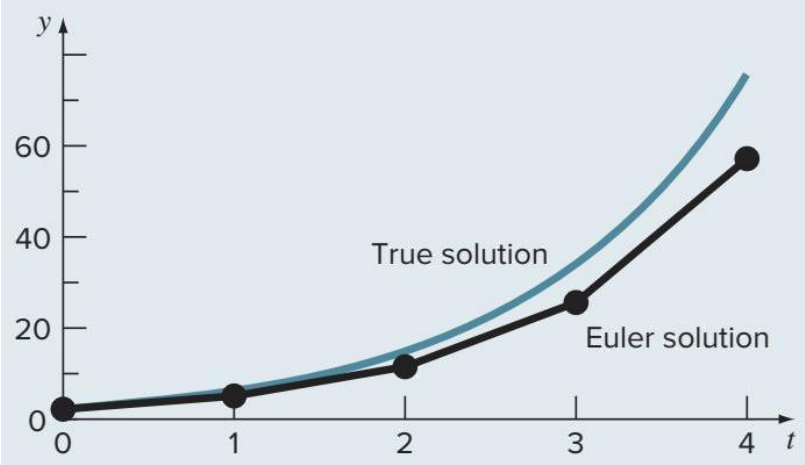
The first derivative provides a direct estimate of the slope at t_i

$\phi = f(t_i, y_i)$
where $f(t_i, y_i)$ is the differential equation evaluated at t_i and y_i.
$y_{i+1} = y_i + f(t_i, y_i)h$
This formula is referred to as *Euler's method* (or the Euler-Cauchy or point-slope method). A new value of y is predicted using the slope (equal to the first derivative at the original value of t) to extrapolate linearly over the step size h.

Problem Statement. Use Euler's method to integrate $y' = 4e^{0.8t} - 0.5y$ from $t = 0$ to 4 with a step size of 1. The initial condition at $t = 0$ is $y = 2$. Note that the exact solution can be determined analytically as

$$y = \frac{4}{1.3}(e^{0.8t} - e^{-0.5t}) + 2e^{-0.5t}$$

t	y_{true}	y_{Euler}	$\lvert\varepsilon_t\rvert$ (%)
0	2.00000	2.00000	
1	6.19463	5.00000	19.28
2	14.84392	11.40216	23.19
3	33.67717	25.51321	24.24
4	75.33896	56.84931	24.54



Assignment Problem –

Assignment –

Read about Modified Euler method and Attempt a numerical based on it.

One numerical as suggested by Faculty.

Sample numerical.

Solve the following initial value problem over the interval from $t = 0$ to where $y(0) = 1$. Display all your results on the same graph.

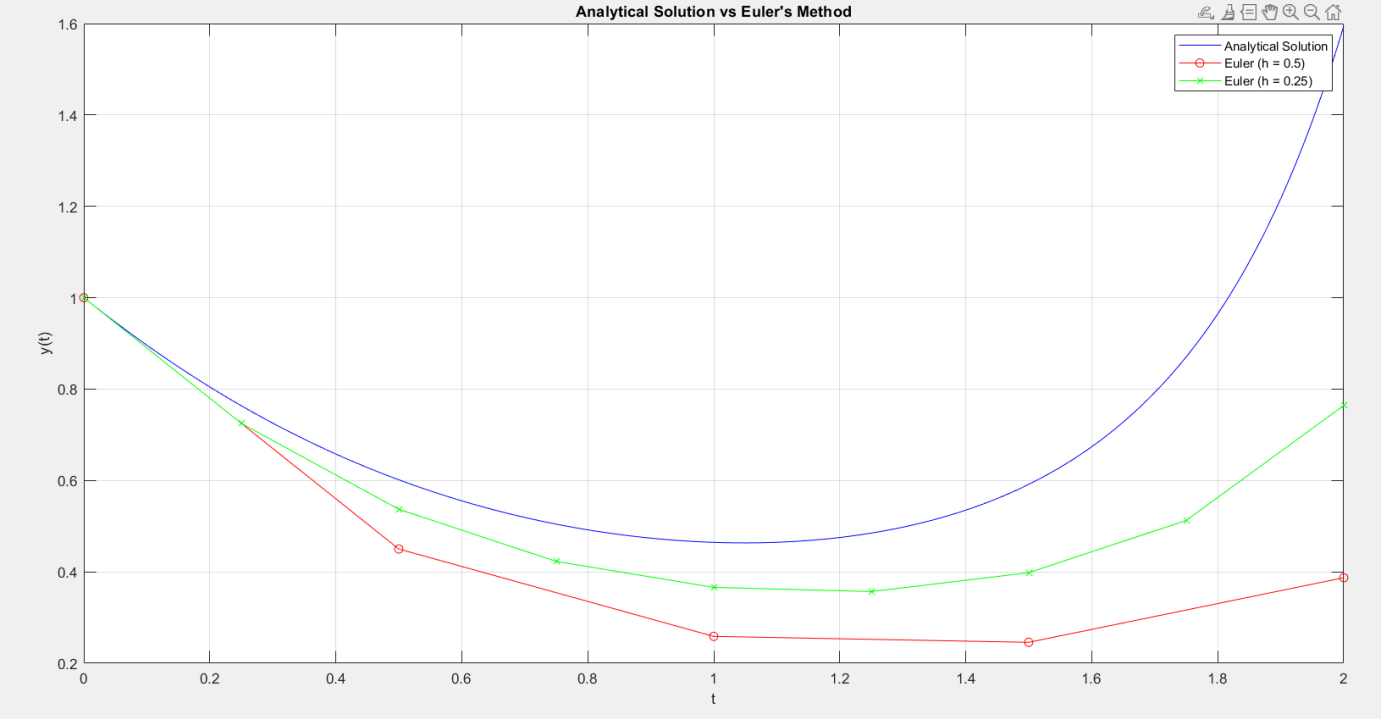
$$\frac{dy}{dt} = yt^2 - 1.1y$$

(a) Analytically.
(b) Using Euler's method with $h = 0.5$ and 0.25.

MATLAB Code:

```
% Parameters
y0 = 1;  % Initial condition y(0)
T = 2;   % Endpoint of the interval
h1 = 0.5;  % Step size 1
h2 = 0.25; % Step size 2

% Function definition
f = @(t, y) y * t^2 - 1.1 * y;

% Analytical solution
t_values = 0:0.01:T;
y_analytical = y0 * exp(t_values.^3 / 3 - 1.1 * t_values);

% Euler's method for step size h1
t_euler_h1 = 0:h1:T;
y_euler_h1 = zeros(size(t_euler_h1));
y_euler_h1(1) = y0;
for i = 1:length(t_euler_h1) - 1
    y_euler_h1(i+1) = y_euler_h1(i) + h1 * f(t_euler_h1(i), y_euler_h1(i));
end

% Euler's method for step size h2
t_euler_h2 = 0:h2:T;
y_euler_h2 = zeros(size(t_euler_h2));
y_euler_h2(1) = y0;
for i = 1:length(t_euler_h2) - 1
    y_euler_h2(i+1) = y_euler_h2(i) + h2 * f(t_euler_h2(i), y_euler_h2(i));
end

% Plotting
figure;
plot(t_values, y_analytical, 'b-', t_euler_h1, y_euler_h1, 'ro-', t_euler_h2,
y_euler_h2, 'gx-');
legend('Analytical Solution', ['Euler (h = ' num2str(h1) ')'], ['Euler (h = '
num2str(h2) ')']);
xlabel('t');
ylabel('y(t)');
title('Analytical Solution vs Euler''s Method');
grid on;
```

Output:


Analytical Solution vs Euler's Method
Analytical Solution
Euler (h = 0.5)
Euler (h = 0.25)
y(t)
t
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2

LAB 11: Partial Differential Equation

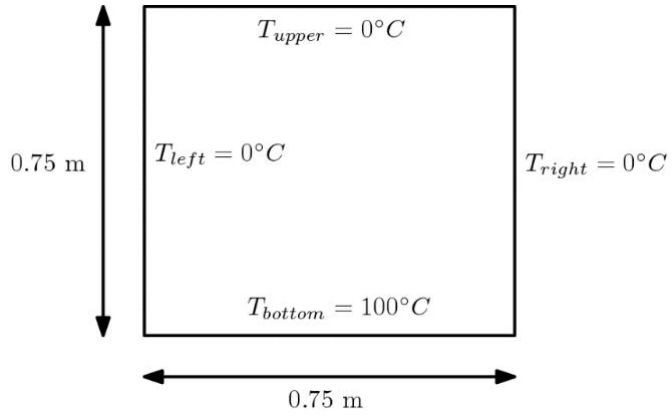
Objective: To numerically obtain the governing PDE of any physical system and solve it numerically.

Theory:

Theory - Consider a 2D metal plate with dimensions 0.75x0.75 m2 is kept at 100 degree C at one of it end and rest of the ends are initially at 0 degree C as shown in the figure.

The transient Fourier law of heat conduction for 2D metal plate is given by,

Where T(x,y,t) is the temperature at any location and time and x, y, t are space and time coordinates. Assume the thermal conductivity of plate is K = 1 W/m degree C.



Take the spatial discretization as delta x = delta y = 0.025 m.

Using explicit finite difference method, the discrete equation can be obtained using Taylor series expansion as below,

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\Delta t} = k\left(\frac{T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}}{\Delta x^2} + \frac{T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}}{\Delta y^2}\right)$$

$$\text{Suppose } \Delta x = \Delta y = \Delta h$$

$$T_{i,j}^{n+1} = T_{i,j}^{n} + \frac{k\Delta t}{\Delta h^2}(T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n} + T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n})$$

Take the spatial discretization as $\Delta = \Delta = \Delta h$ = 0.025 m and $\Delta = 0.00015$, thermal conductivity of plate is K = 1 W/m degree C.

MATLAB program-

```
% Explicit method for 2D heat equation in flat plate
close all; clear all; clc;
```

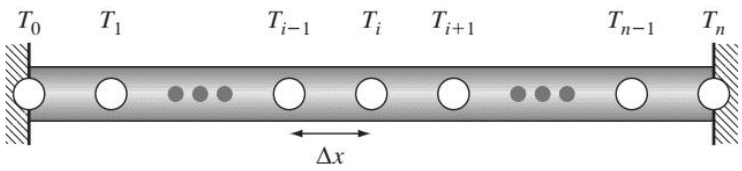
```
L =0.75;
dx = 0.05
dy=0.05;
N = L/dx +1;
x = linspace(0,L,N);
y = linspace(0,L,N);
dt = 0.00015;
epsilon = 1e-4;
T_new = zeros(N,N);
 %boundary condition
      T_new(1,:) =100;
      T_new(:,1) =0;
      T_new(N,:)=0;
      T_new(:,N)=0;

error =1;iter=0;
while(error > epsilon)
    iter=iter+1;
      T = T_new;% update T every iteration
   for i=2:N-1
      for j =2:N-1
         T_new(i,j) = dt*((T(i+1,j)-2*T(i,j)+T(i-1,j))/dx^2 +(T(i,j+1)-2*T(i,j)+T(i,j-1))/dy^2) +
T(i,j);

      end
   end
   error = max(max(abs(T-T_new)));

figure(2);
hold on;
pause(0.001);
% contourf(T_new); shading
    flat;colorbar;
    contourf(x,y,T_new,'ShowText','on')
colorbar;
xlabel('x meter');ylabel('y meter')

end
```

Assignment -

One numerical as suggested by Faculty.

Sample numerical.

Consider 1D heat conduction in metal rod as below.



1D heat conduction governing equation is given below,

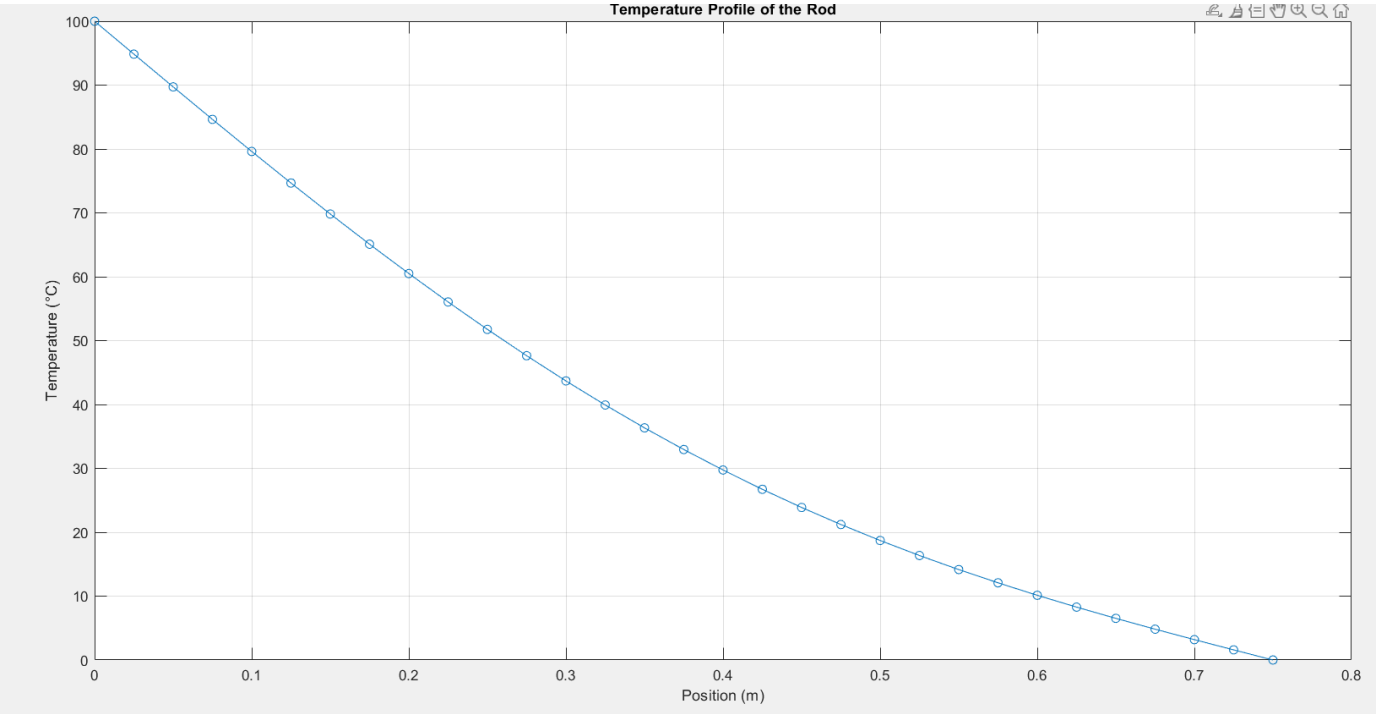
$$\frac{\partial T}{\partial t} = k\left(\frac{\partial^2 T}{\partial x^2}\right)$$

Write down the discretized equation is explicit finite difference method and obtain the Temperature profile T(x,t). Take length of rod is 0.75 m, Δx= 0.025 m, Δ = 0.00015, thermal conductivity of plate is K = 1 W/m degree

MATLAB Code:

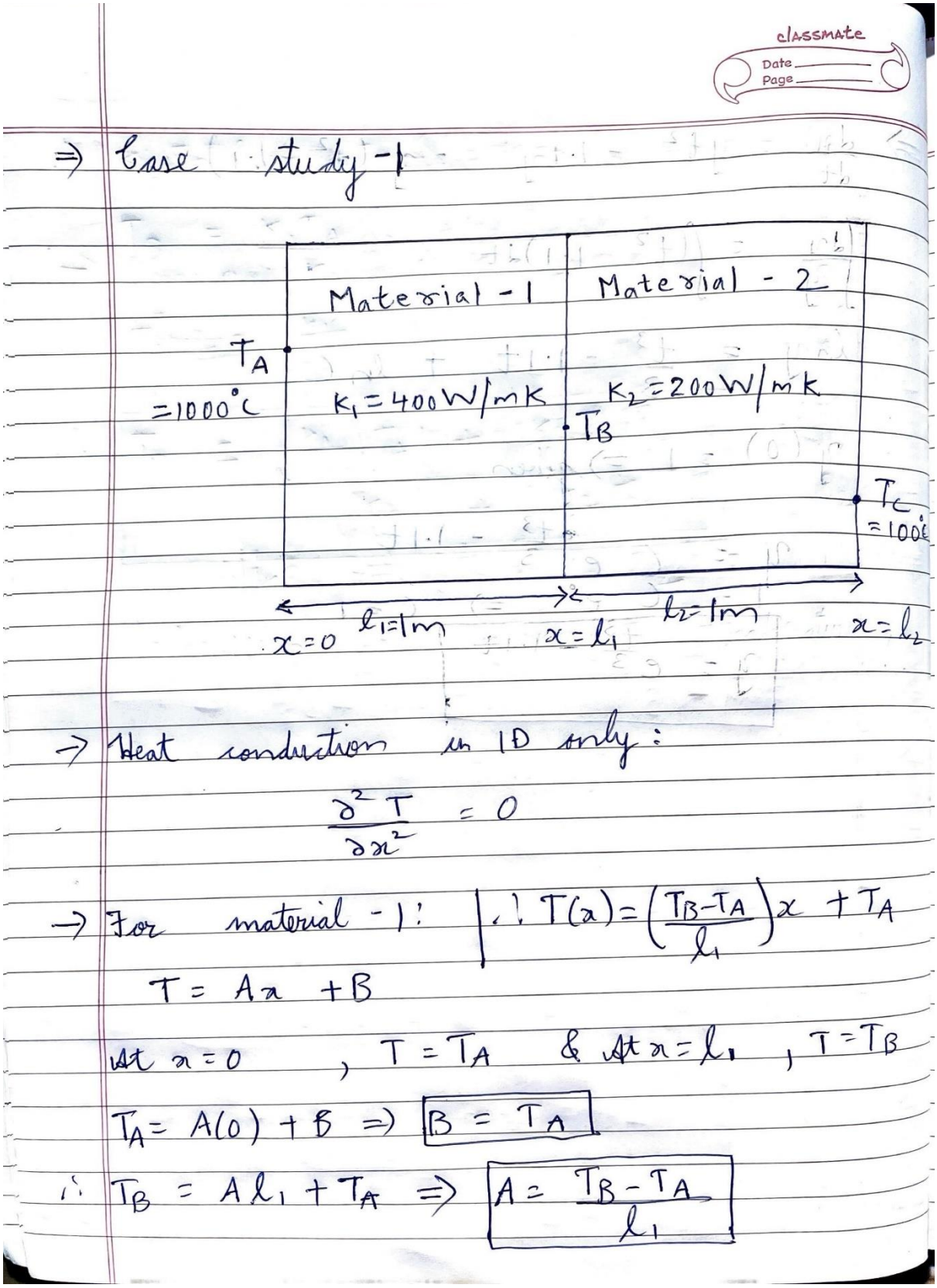
```
% Parameters and grid setup
L = 0.75;           % Length of the rod (meters)
dx = 0.025;         % Spatial step size (meters)
dt = 0.00015;       % Time step (seconds)
k = 1;              % Thermal conductivity (W/m°C)

% Number of nodes along the rod
n = round(L / dx) + 1;

% Spatial grid
x = linspace(0, L, n);

% Initial temperature profile (e.g., at t = 0)
T0 = zeros(1, n);   % Initially, temperature is zero everywhere
T0(1) = 100;        % Set a boundary condition at x = 0 (e.g., 100°C)

% Time-stepping loop
num_steps = 500;  % Number of time steps
T = T0;           % Initialize temperature profile

for step = 1:num_steps
    % Compute new temperature profile using explicit finite difference
    T_new = zeros(1, n);

    % Update internal nodes (excluding boundary nodes)
    for i = 2:n-1
        T_new(i) = T(i) + k * (dt / dx^2) * (T(i+1) - 2*T(i) + T(i-1));
    end

    % Update boundary condition (e.g., Dirichlet boundary at x = 0)
    T_new(1) = 100;  % Maintain a constant temperature at the boundary

    % Update T for the next time step
    T = T_new;
end

% Plot the temperature profile at the final time step
figure;
plot(x, T, '-o');
title('Temperature Profile of the Rod');
xlabel('Position (m)');
ylabel('Temperature (°C)');
grid on;
```

Output:


Temperature Profile of the Rod
Temperature (°C)
Position (m)
0
10
20
30
40
50
60
70
80
90
100
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8

CASE STUDY – 1

Consider a composite wall of length 2 m. For a length of 1 m, material 1 with thermal conductivity 400 W/m*K is used and material 2 with thermal conductivity of 200 W/m*K for the remaining part of the wall. Temperature at left end is 1000 ^{o}C and at right end is 100 ^{o}C. Assuming 1D steady state heat conduction, we will plot the temperature distribution profile along the length of the wall. Neglect the contact resistance between walls.



→ For material -2 :

$T = Cx + D$

At $x = l_1 \Rightarrow T = T_B$

At $x = l_1 + l_2 \Rightarrow T = T_C$

$\therefore T_B = Cl_1 + D$

$\therefore T_C = C(l_1 + l_2) + D$

$\therefore T_C - T_B = Cl_2 \Rightarrow \boxed{C = \frac{T_C - T_B}{l_2}}$

$\therefore T_B = (T_C - T_B)\frac{l_1}{l_2} + D$

$\therefore \boxed{D = T_B - (T_C - T_B)\frac{l_1}{l_2}}$

$\boxed{T(x) = \left(\frac{T_C - T_B}{l_2}\right)x + T_B - (T_C - T_B)\frac{l_1}{l_2}}$

→ $q_1 = -K_1 A \frac{dT}{dx} = -K_1 A \frac{T_B - T_A}{l_1} = K_1 A \frac{T_A - T_B}{l_1}$

→ $q_2 = -K_2 A \frac{dT}{dx} = -K_2 A \frac{T_C - T_B}{l_2} = K_2 A \frac{T_B - T_C}{l_2}$

→ Also, $q_1 = q_2$

$\therefore K_1 A \left(\frac{T_A - T_B}{l_1}\right) = K_2 A \left(\frac{T_B - T_C}{l_2}\right)$

$\frac{K_1 T_A - K_1 T_B}{l_1} = \frac{K_2 T_B - K_2 T_C}{l_2}$

$\therefore K_1 T_A l_2 - K_1 T_B l_2 = K_2 T_B l_1 - K_2 T_C l_1$

$$\therefore K_2 T_B l_1 + K_1 T_B l_2 = K_1 T_A l_2 + K_2 T_C l_1$$

$$\therefore T_B = \frac{K_1 T_A l_2 + K_2 T_C l_1}{K_2 l_1 + K_1 l_2}$$

$$\therefore T_B = \left(\frac{K_1}{l_1}\right)$$

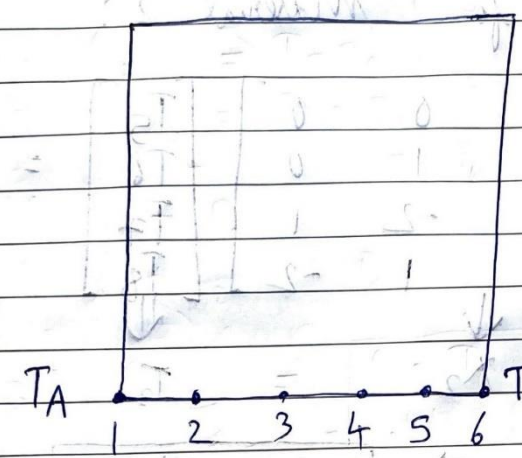
$$\therefore \boxed{T_B = \frac{\left(\frac{K_1}{l_1}\right) T_A + \left(\frac{K_2}{l_2}\right) T_C}{\left(\frac{K_1}{l_1}\right) + \left(\frac{K_2}{l_2}\right)}}$$

$\Downarrow$

Expression for intermediate temperature

⇒ Material - 1 : (6 nodes) :

1st and 6th nodes are BC's (T_A and T_B)



$$\frac{\partial^2 T}{\partial x^2} = 0 \Rightarrow \frac{T(x+h) - 2T(x) + T(x-h)}{h^2} = 0$$

$$\therefore \boxed{T(x+h) - 2T(x) + T(x-h) = 0}$$

→ For i = 2 ; $T_1 - 2T_2 + T_3 \Rightarrow -2T_2 + T_3 = -T_{a1}$

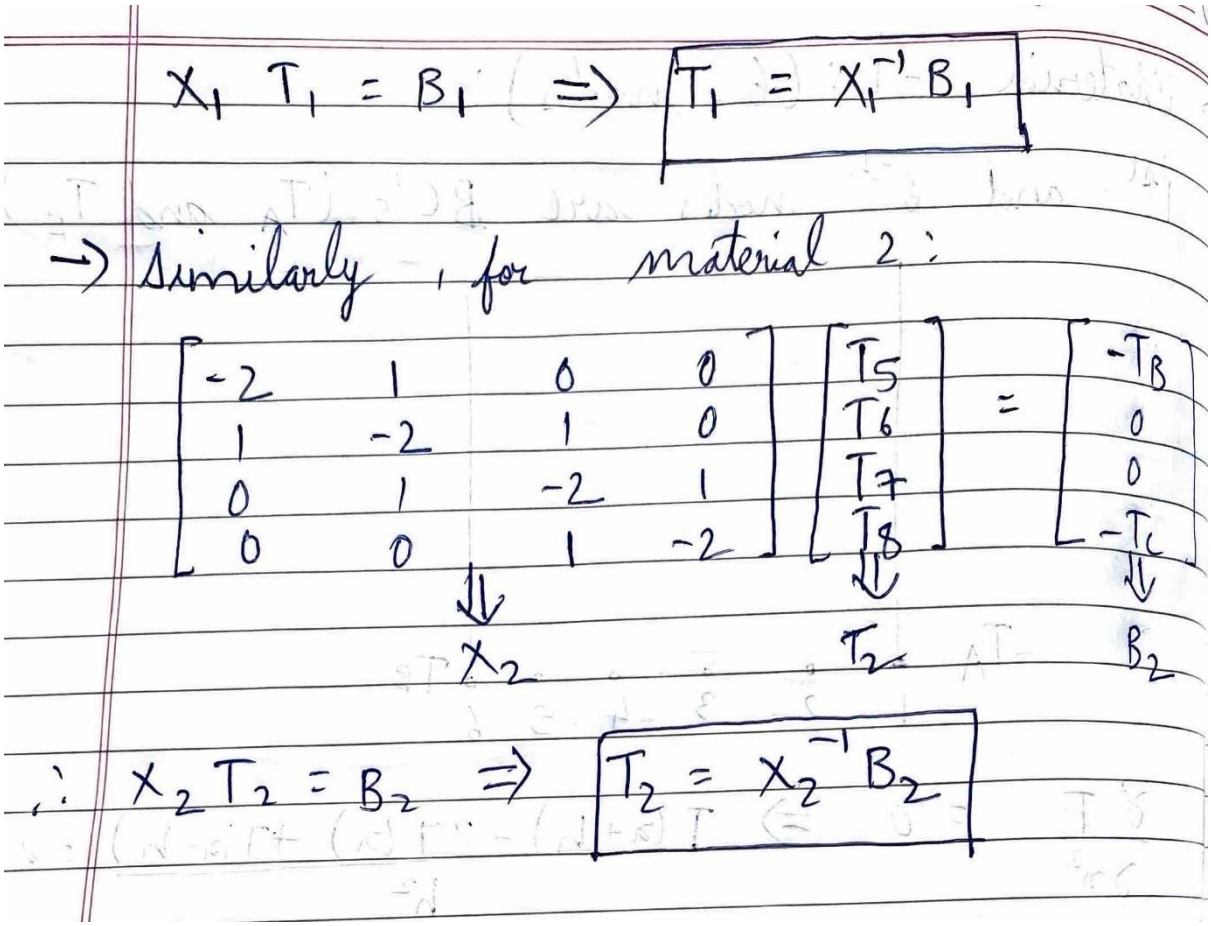
→ For i = 3 ; $T_2 - 2T_3 + T_4 = 0$

→ For i = 4 ; $T_3 - 2T_4 + T_5 = 0$

→ For i = 5 ; $T_4 - 2T_5 + T_6 = 0 \Rightarrow T_4 - 2T_5 = -T_6$

$$\begin{bmatrix} -2 & 1 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} T_2 \\ T_3 \\ T_4 \\ T_5 \end{bmatrix} = \begin{bmatrix} -T_1 \\ 0 \\ 0 \\ -T_6 \end{bmatrix}$$

X_1 T_1 B_1

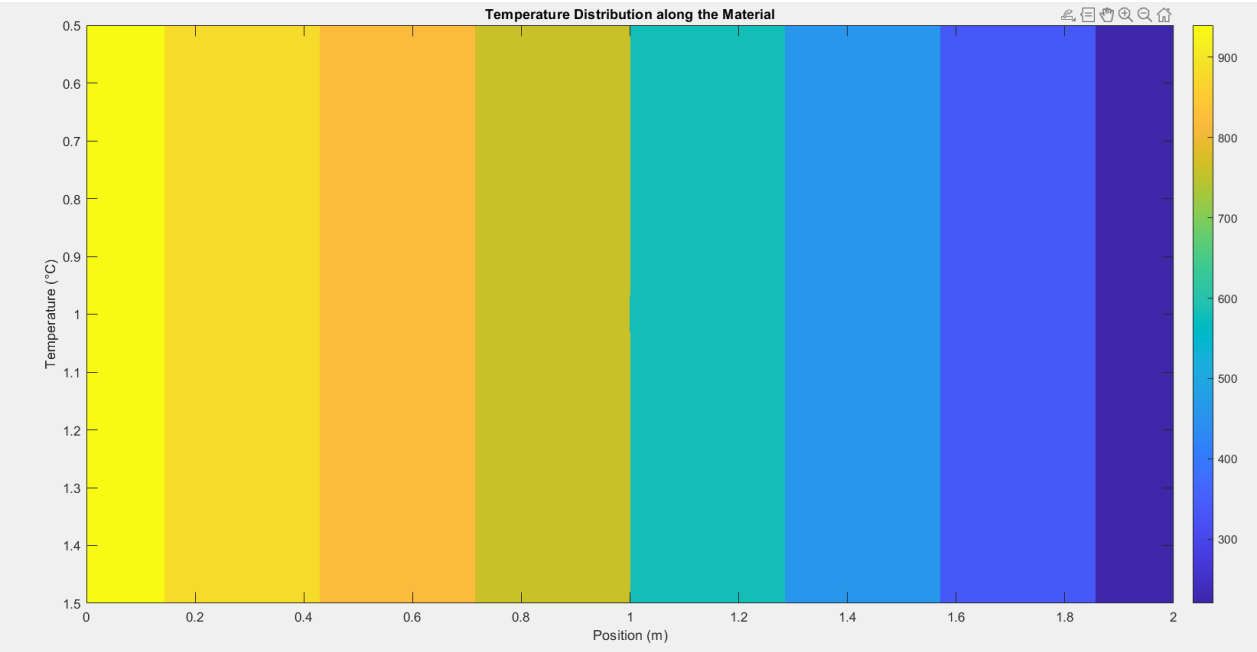


MATLAB Code:

```
% Input parameters
L1 = 1; % (m) (Material 1)
L2 = 1; % (m) (Material 2)
K1 = 400; % (W/m*k) (Thermal conductivity of material 1)
K2 = 200; % (W/m*k) (Thermal conductivity of material 2)
M1 = 5; % No. of sections / grid spacings in domain
M2 = 5; % No. of sections / grid spacings in domain
dx1 = L1/M1; % Segment length
dx2 = L2/M2; % Segment length
TA = 1000; % Temperature at left end in Celsius
TC = 100; % Temperature at right end in Celsius

%TB estimation
TB = (((K1/L1)*TA)+((K2/L2)*TC))/((K1/L1)+(K2/L2));

N1 = M1 - 1; % No. of points in domain 1
N2 = M2 - 1; % No. of points in domain 2

%Domain 1
X1 = zeros(N1,N1);
for i = 1:N1
    X1(i,i) = -2;
end
for i = 1:N1-1
    X1(i,i+1) = 1;
    X1(i+1,i) = 1;
end

B1 = zeros(N1,1);
```

```
B1(1,1) = -TA;
B1(N1,1) = -TB;

T1 = X1\B1;

% Domain 2
X2 = zeros(N2,N2);
for i = 1:N2
    X2(i,i) = -2;
end
for i = 1:N2-1
    X2(i,i+1) = 1;
    X2(i+1,i) = 1;
end

B2 = zeros(N2,1);
B2(1,1) = -TB;
B2(N2,1) = -TC;

T2 = X2\B2;

% Combine temperature distributions
T_combined = [T1; T2];

% Create x-coordinates for plotting
x = linspace(0, L1+L2, N1+N2+1);

% Plot the temperature distribution
figure;
imagesc(x, 1, T_combined');
colorbar;
xlabel('Position (m)');
ylabel('Temperature (°C)');
title('Temperature Distribution along the Material');

% Set axis limits
xlim([0, L1+L2]);
```

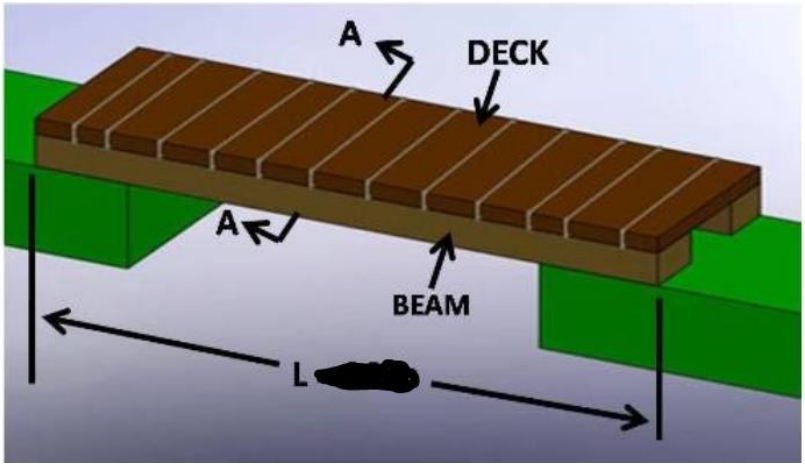
Output:



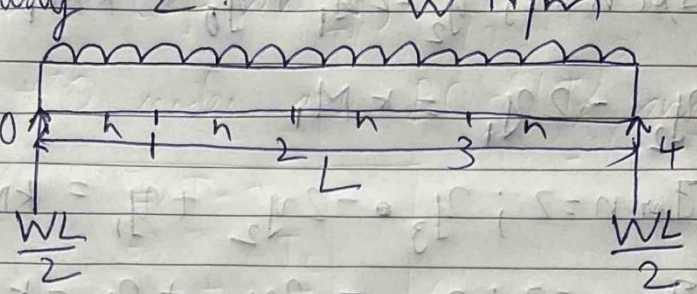
CASE STUDY – 2

Deflection of a simple supported pedestrian bridge subjected to uniformly distributed load (UDL) from traffic:

- Design for flexural stress. Shear stress will not be considered here.
- The bridge will be subjected to 4000 N/mm2 load from traffic.
- The length (L) of the bridge is 10000 mm.
- Cross section of the beam is rectangular with B x H = 3m x 0.9m
  (Moment of Inertia I = 1.8225*1011 mm4
- Material of the Beam would Steel A36.
- The beams will be located as shown figure.
- The factor of safety against failure is 1.5.



⇒ Case Study - 2 :



$$M_1 = \frac{WL}{2}h - \frac{Wh^2}{2}$$

$$M_2 = \frac{WL(2h)}{2} - \frac{W(2h)^2}{2}$$

$$M_3 = \frac{WL(3h)}{2} - \frac{W(3h)^2}{2}$$

$$M_4 = \frac{WL(4h)}{2} - \frac{W(4h)^2}{2}$$

→ We know that (y is deflection)

$$EI\frac{d^2y}{dx^2} = M_n$$

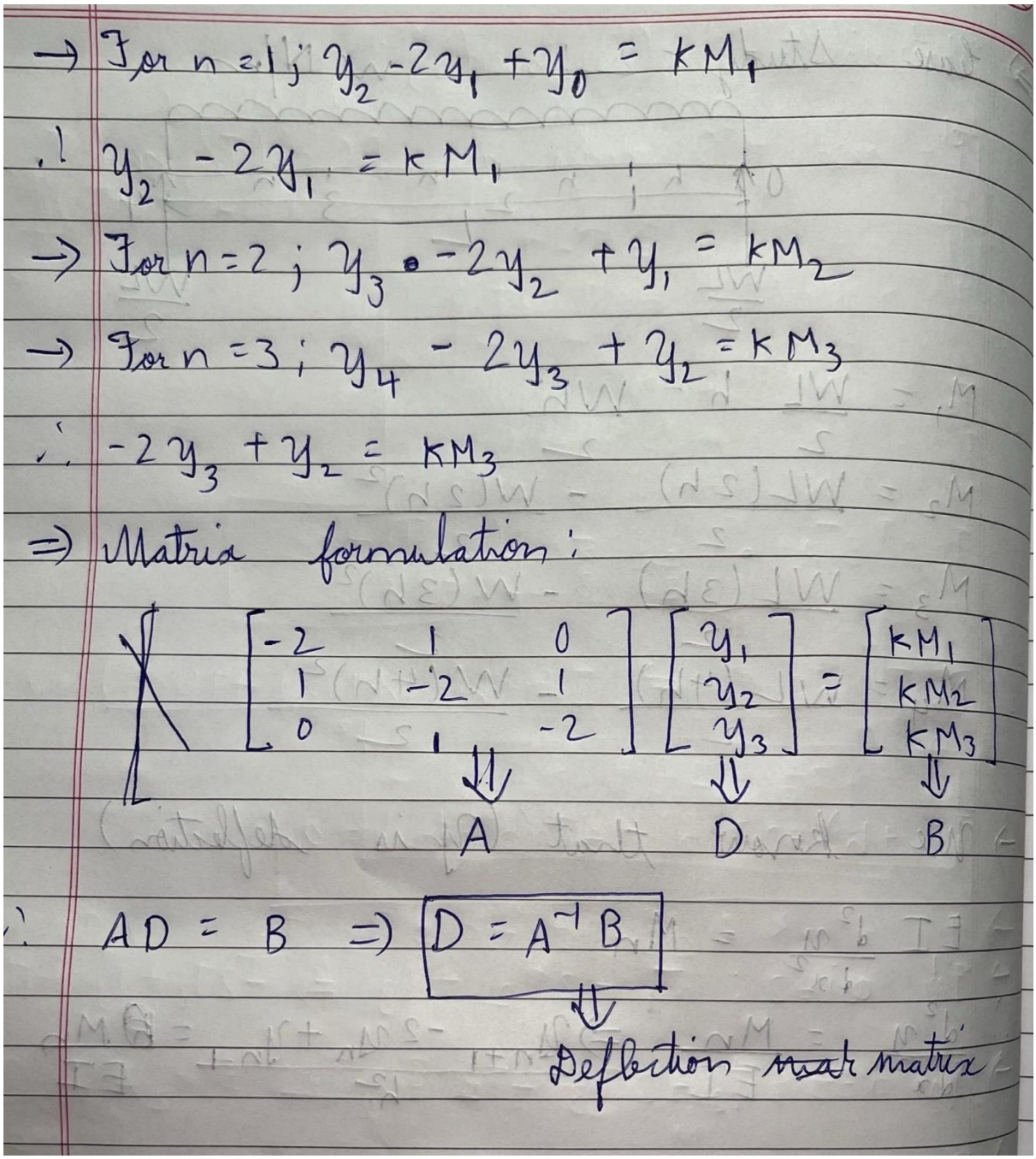
$$\therefore \frac{d^2y}{dx^2} = \frac{M_n}{EI} \Rightarrow \frac{y_{n+1} - 2y_n + y_{n-1}}{h^2} = \frac{M_n}{EI}$$

$$\therefore y_{n+1} - 2y_n + y_{n-1} = \frac{h^2 M_n}{EI}$$

$$\frac{h^2}{EI} = K = \text{some constant factor}$$

$$\therefore y_{n+1} - y_n + y_{n-1} = KM_n$$

→ Also $y_0 = y_4 = 0$ (Deflection at ends)

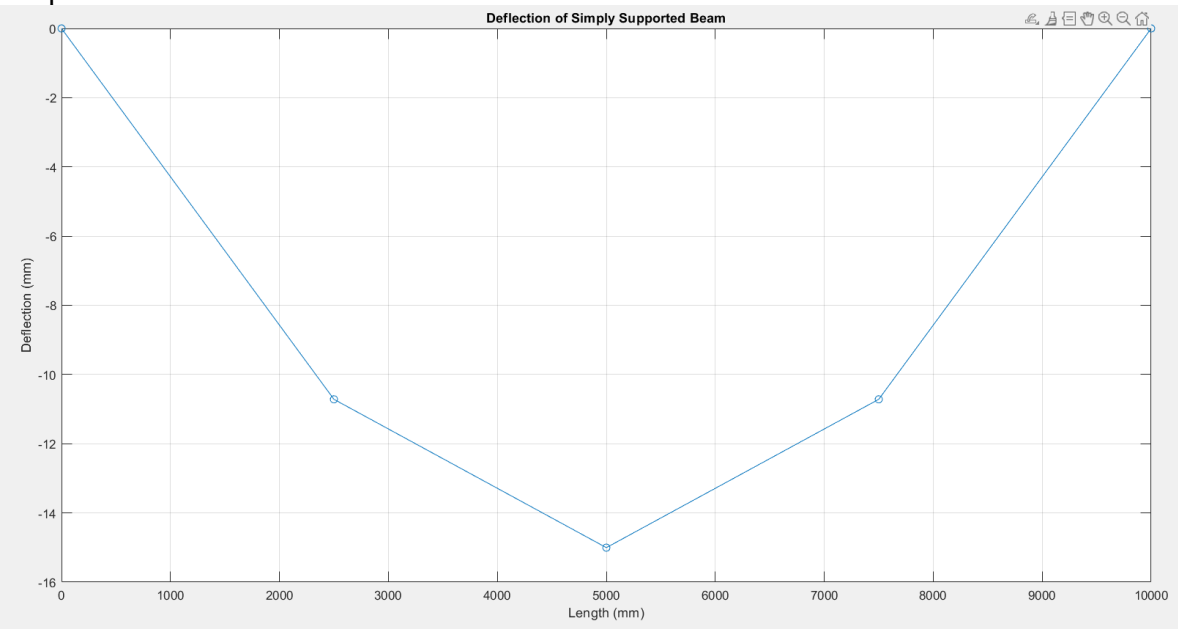


MATLAB Code:

```
% Parameters
L = 10000; % Length of the bridge (mm)
W = 4000; % Load per unit length (N/mm)
E = 200e3; % Young's modulus of A36 steel (MPa)
I = 1.8225e11; % Moment of inertia (mm^4)
FOS = 1.5; % Factor of safety
N = 5; % Number of nodes
h = L/(N-1); % Distance between two consecutive nodes
K = (h^2)/(E*I); % Constant factor

x = linspace(0,L,N); % Discretize the beam

% Bending Moment Values at each nodes
M = zeros(1, N);
```

```
for i = 1:N
    M(i) = ((W*L*(i-1)*h)/2) - ((W*((i-1)*h)^2)/2);
end

% Coefficient Matrix
A = zeros(N, N);

for j = 1:N
    A(j,j) = -2;
end

for j = 1:N-1
    A(j,j+1) = 1;
    A(j+1,j) = 1;
end

% Boundary conditions
A(1,:) = 0;
A(1,1) = 1;
A(N,:) = 0;
A(N,N) = 1;

% Constant right-hand side matrix
B = zeros(N, 1);
for i = 1:N
    B(i) = K * M(i);
end
B(1) = 0; % Deflection at the left support is zero
B(N) = 0; % Deflection at the right support is zero

% Deflection matrix
D = A \ B;

% Plot deflection curve
plot(x, D, '-o');
xlabel('Length (mm)');
ylabel('Deflection (mm)');
title('Deflection of Simply Supported Beam');
grid on;
```

Output:



Extras

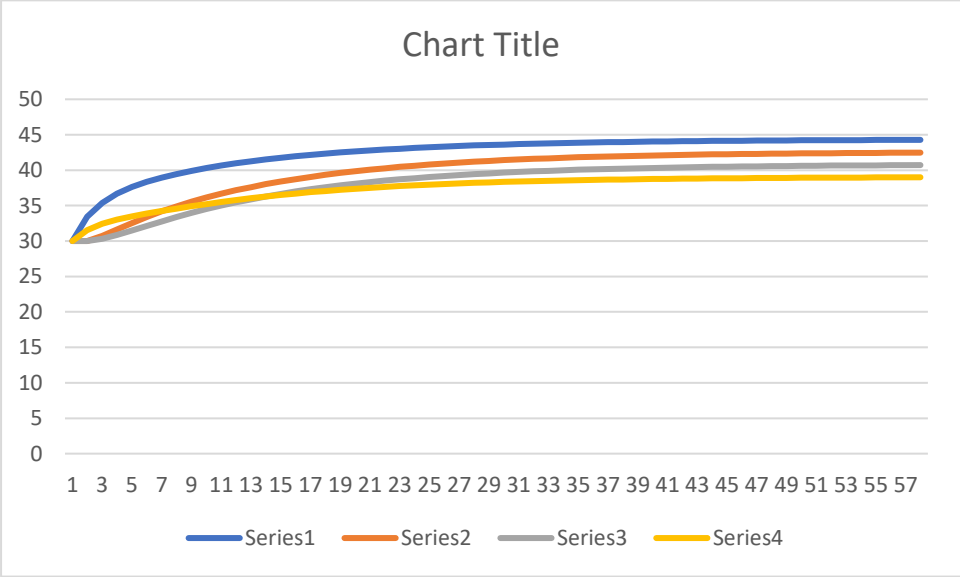
1). Boundary Conditions Feasibility Check (1D Unsteady State Heat Conduction):

delta t	50
rho	8850
cp	380
k	398
h	0.166667

i). Dirichlet – Dirichlet:

T1	T2	T3	T4	T5	T6
46.1	44.1	42.7	40.8	39.5	37.3
46.1	30	30	30	30	37.3
46.1	33.42968778	30	30	31.55508	37.3
46.1	35.39816335	30.7306061	30.33126861	32.44761	37.3
46.1	36.6836102	31.63983971	30.86716975	33.03046	37.3
46.1	37.61508343	32.549687	31.49259931	33.47914	37.3
46.1	38.34352358	33.4035533	32.14096548	33.86989	37.3
46.1	38.94350723	34.18692434	32.77823057	34.23229	37.3
46.1	39.45474584	34.90010536	33.38806508	34.57603	37.3
46.1	39.9000969	35.54825248	33.96323209	34.90324	37.3
46.1	40.29377791	36.13765264	34.5011236	35.21356	37.3
46.1	40.64528828	36.67438732	35.00151032	35.50626	37.3
46.1	40.96137571	37.16392178	35.46539666	35.78084	37.3
46.1	41.2470773	37.61104417	35.89442139	36.03726	37.3
46.1	41.50630409	38.0199248	36.29053212	36.27583	37.3
46.1	41.7421892	38.39420527	36.65580204	36.49713	37.3
46.1	41.95730664	38.73708473	36.99232358	36.70196	37.3
46.1	42.15381528	39.05139341	37.30214606	36.89121	37.3
46.1	42.33355709	39.33965217	37.58723933	37.06583	37.3
46.1	42.49812629	39.60411989	37.84947345	37.22679	37.3
46.1	42.64891908	39.846831	38.09060852	37.37503	37.3
46.1	42.78717018	40.06962556	38.31229101	37.51148	37.3
46.1	42.91398027	40.27417343	38.51605419	37.63702	37.3
46.1	43.03033677	40.46199413	38.70332121	37.75248	37.3
46.1	43.13713013	40.63447328	38.87540968	37.85865	37.3
46.1	43.23516659	40.79287655	39.03353728	37.95624	37.3
46.1	43.3251785	40.93836151	39.17882773	38.04593	37.3
46.1	43.40783281	41.07198795	39.31231691	38.12837	37.3
46.1	43.48373805	41.19472685	39.43495897	38.20411	37.3
46.1	43.55345034	41.30746842	39.54763221	38.27372	37.3
46.1	43.61747851	41.41102917	39.65114474	38.33767	37.3
46.1	43.67628853	41.50615825	39.74623979	38.39642	37.3
46.1	43.73030743	41.59354325	39.83360079	38.4504	37.3
46.1	43.77992679	41.67381535	39.91385595	38.49999	37.3
46.1	43.8255058	41.74755407	39.98758272	38.54555	37.3
46.1	43.86737408	41.81529155	40.05531176	38.58741	37.3
46.1	43.90583417	41.87751647	40.11753073	38.62586	37.3

46.1	43.94116382	41.93467764	40.1746877	38.66118	37.3
46.1	43.97361804	41.98718728	40.22719438	38.69363	37.3
46.1	44.00343101	42.035424	40.27542901	38.72344	37.3
46.1	44.03081781	42.07973557	40.3197391	38.75082	37.3
46.1	44.05597594	42.12044144	40.36044394	38.77598	37.3
46.1	44.07908683	42.15783505	40.39783681	38.79909	37.3
46.1	44.10031711	42.19218594	40.43218718	38.82032	37.3
46.1	44.11981983	42.22374171	40.46374259	38.83982	37.3
46.1	44.13773559	42.25272981	40.49273043	38.85774	37.3
46.1	44.15419354	42.27935919	40.51935962	38.87419	37.3
46.1	44.16931231	42.30382176	40.54382207	38.88931	37.3
46.1	44.18320087	42.32629386	40.56629407	38.9032	37.3
46.1	44.19595933	42.34693743	40.58693758	38.91596	37.3
46.1	44.20767966	42.36590127	40.60590138	38.92768	37.3
46.1	44.21844631	42.38332206	40.62332214	38.93845	37.3
46.1	44.2283369	42.39932535	40.63932541	38.94834	37.3
46.1	44.23742271	42.41402648	40.65402652	38.95742	37.3
46.1	44.24576922	42.42753142	40.66753144	38.96577	37.3
46.1	44.25343659	42.43993748	40.67993749	38.97344	37.3
46.1	44.26048008	42.45133408	40.69133409	38.98048	37.3
46.1	44.26695046	42.46180336	40.70180337	38.98695	37.3
46.1	44.27289435	42.47142078	40.71142079	38.99289	37.3

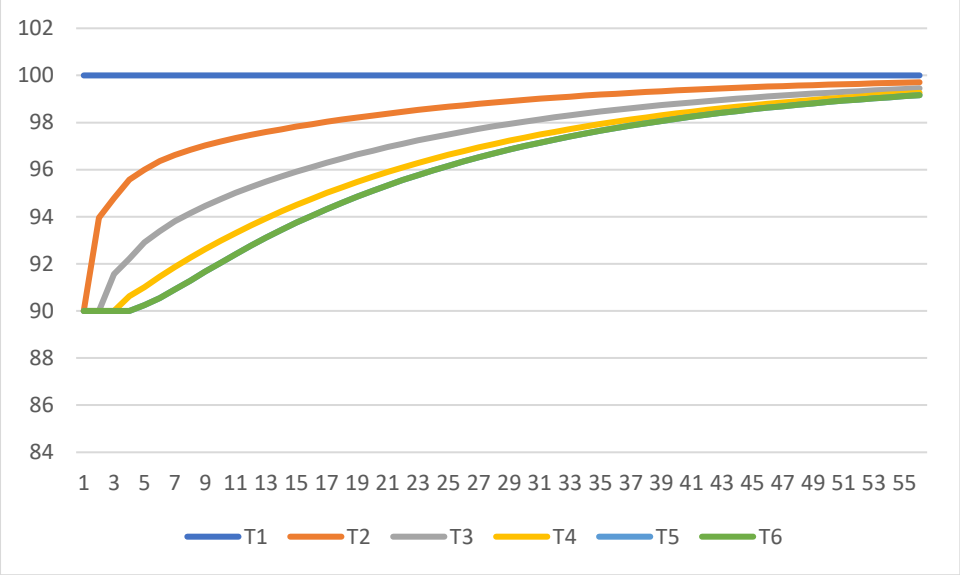


ii). Dirchlet – Neumann:

100	90	90	90	90	90
100	93.96	90	90	90	90
100	94.78368	91.56816	90	90	90
100	95.5759968	92.22051	90.62099	90	90
100	95.9991311	92.91587	91.00849	90.24591	90.24591
100	96.36250551	93.38152	91.46183	90.54789	90.54789

100	96.62248299	93.80179	91.86011	90.90981	90.90981
100	96.84298704	94.14988	92.2527	91.28613	91.28613
100	97.02669364	94.46507	92.62122	91.66889	91.66889
100	97.18971873	94.74931	92.97426	92.04601	92.04601
100	97.33618758	95.01279	93.30959	92.4136	92.4136
100	97.47099279	95.25839	93.62925	92.76841	92.76841
100	97.5962891	95.48944	93.9335	93.1093	93.1093
100	97.71384645	95.7076	94.22327	93.43568	93.43568
100	97.82468935	95.91428	94.49918	93.74757	93.74757
100	97.9295898	96.11042	94.76192	94.04521	94.04521
100	98.02908199	96.29681	95.01211	94.32903	94.32903
100	98.12358431	96.47405	95.25035	94.59953	94.59953
100	98.21342826	96.64268	95.47721	94.85725	94.85725
100	98.29889399	96.80317	95.69323	95.10275	95.10275
100	98.3802249	96.95594	95.89894	95.33658	95.33658
100	98.45763942	97.10138	96.09482	95.55928	95.55928
100	98.53133715	97.23986	96.28134	95.77135	95.77135
100	98.60150323	97.37171	96.45896	95.97331	95.97331
100	98.66831093	97.49726	96.62809	96.16563	96.16563
100	98.73192344	97.61681	96.78915	96.34876	96.34876
100	98.79249504	97.73064	96.94251	96.52315	96.52315
100	98.85017199	97.83903	97.08854	96.68922	96.68922
100	98.90509316	97.94225	97.2276	96.84735	96.84735
100	98.95739054	98.04054	97.36002	96.99793	96.99793
100	99.00718968	98.13413	97.48612	97.14132	97.14132
100	99.05461003	98.22325	97.60619	97.27786	97.27786
100	99.0997653	98.30811	97.72053	97.40788	97.40788
100	99.14276372	98.38892	97.8294	97.53169	97.53169
100	99.18370835	98.46587	97.93308	97.64958	97.64958
100	99.22269729	98.53915	98.0318	97.76185	97.76185
100	99.25982397	98.60892	98.12581	97.86875	97.86875
100	99.29517734	98.67537	98.21533	97.97054	97.97054
100	99.32884211	98.73864	98.30057	98.06748	98.06748
100	99.36089893	98.79888	98.38174	98.15978	98.15978
100	99.3914246	98.85625	98.45903	98.24768	98.24768
100	99.42049226	98.91088	98.53263	98.33137	98.33137
100	99.44817155	98.9629	98.60272	98.41107	98.41107
100	99.47452878	99.01244	98.66946	98.48697	98.48697
100	99.49962709	99.05961	98.73301	98.55923	98.55923
100	99.52352663	99.10452	98.79353	98.62805	98.62805
100	99.54628464	99.14729	98.85115	98.69358	98.69358
100	99.56795564	99.18802	98.90603	98.75598	98.75598
100	99.58859157	99.22681	98.95828	98.8154	98.8154
100	99.60824185	99.26374	99.00803	98.87198	98.87198
100	99.62695357	99.2989	99.05541	98.92586	98.92586
100	99.64477155	99.33239	99.10053	98.97716	98.97716
100	99.66173849	99.36428	99.14349	99.02601	99.02601
100	99.67789502	99.39464	99.1844	99.07254	99.07254
100	99.69327987	99.42355	99.22336	99.11683	99.11683

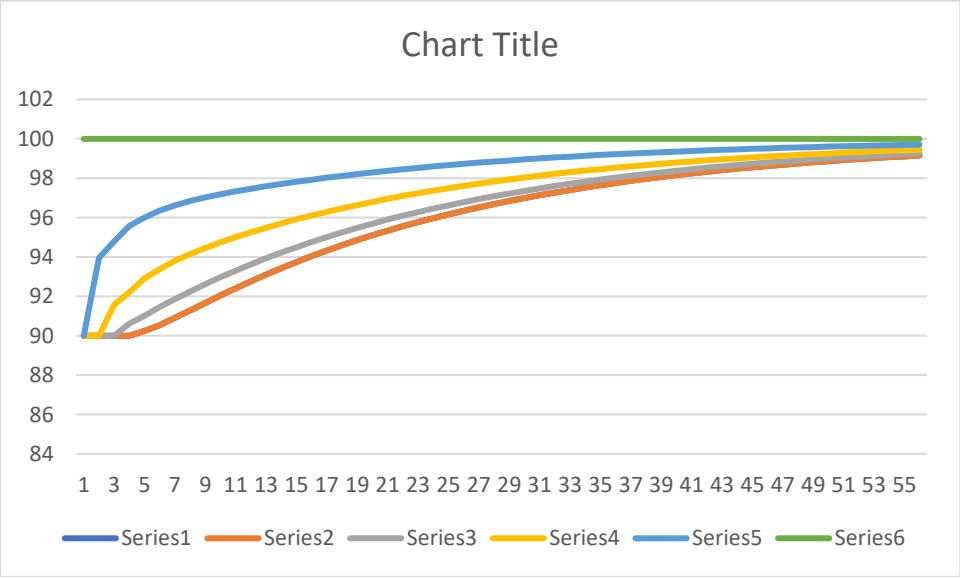
100	99.70792988	99.45109	99.26045	99.15902	99.15902



iii). Neumann – Dirchlet:

T1	T2	T3	T4	T5	T6
90	90	90	90	90	100
90	90	90	90	93.96	100
90	90	90	91.56816	94.78368	100
90	90	90.62099	92.22051	95.576	100
90.24591	90.24591258	91.00849	92.91587	95.99913	100
90.54789	90.54789323	91.46183	93.38152	96.36251	100
90.90981	90.90981359	91.86011	93.80179	96.62248	100
91.28613	91.28613054	92.2527	94.14988	96.84299	100
91.66889	91.66889182	92.62122	94.46507	97.02669	100
92.04601	92.04601437	92.97426	94.74931	97.18972	100
92.4136	92.41360031	93.30959	95.01279	97.33619	100
92.76841	92.76841389	93.62925	95.25839	97.47099	100
93.1093	93.10930384	93.9335	95.48944	97.59629	100
93.43568	93.43568469	94.22327	95.7076	97.71385	100
93.74757	93.74756855	94.49918	95.91428	97.82469	100
94.04521	94.04520693	94.76192	96.11042	97.92959	100
94.32903	94.32902576	95.01211	96.29681	98.02908	100
94.59953	94.59952666	95.25035	96.47405	98.12358	100
94.85725	94.85725195	95.47721	96.64268	98.21343	100
95.10275	95.10275442	95.69323	96.80317	98.29889	100
95.33658	95.3365835	95.89894	96.95594	98.38022	100
95.55928	95.55927585	96.09482	97.10138	98.45764	100
95.77135	95.77135086	96.28134	97.23986	98.53134	100
95.97331	95.97330802	96.45896	97.37171	98.6015	100
96.16563	96.16562602	96.62809	97.49726	98.66831	100
96.34876	96.34876248	96.78915	97.61681	98.73192	100
96.52315	96.52315426	96.94251	97.73064	98.7925	100
96.68922	96.68921804	97.08854	97.83903	98.85017	100
96.84735	96.84735096	97.2276	97.94225	98.90509	100
96.99793	96.99793148	97.36002	98.04054	98.95739	100

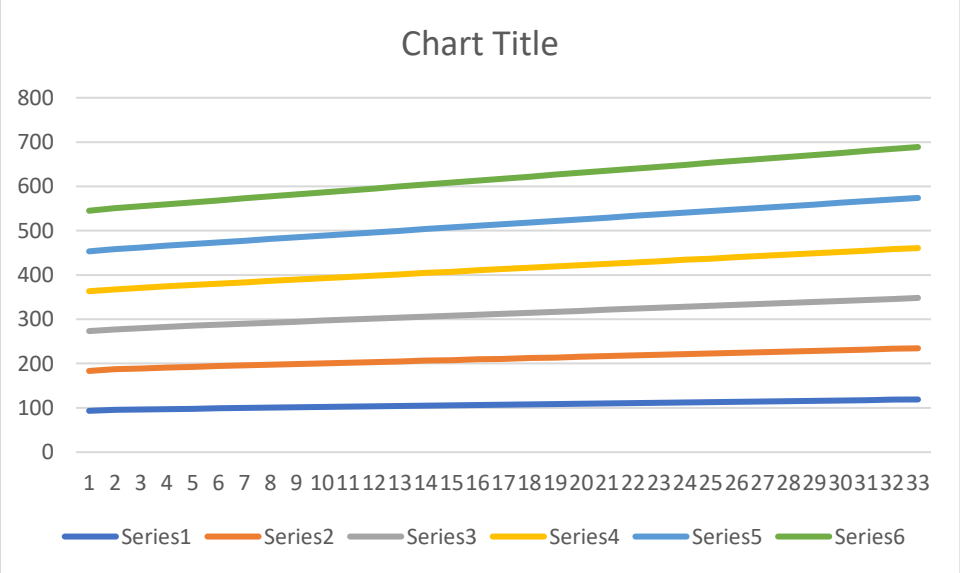
97.14132	97.1413201	97.48612	98.13413	99.00719	100
97.27786	97.27786018	97.60619	98.22325	99.05461	100
97.40788	97.40787877	97.72053	98.30811	99.09977	100
97.53169	97.53168729	97.8294	98.38892	99.14276	100
97.64958	97.64958234	97.93308	98.46587	99.18371	100
97.76185	97.76184634	98.0318	98.53915	99.2227	100
97.86875	97.86874824	98.12581	98.60892	99.25982	100
97.97054	97.97054415	98.21533	98.67537	99.29518	100
98.06748	98.06747794	98.30057	98.73864	99.32884	100
98.15978	98.15978185	98.38174	98.79888	99.3609	100
98.24768	98.247677	98.45903	98.85625	99.39142	100
98.33137	98.33137398	98.53263	98.91088	99.42049	100
98.41107	98.41107331	98.60272	98.9629	99.44817	100
98.48697	98.48696592	98.66946	99.01244	99.47453	100
98.55923	98.55923364	98.73301	99.05961	99.49963	100
98.62805	98.6280496	98.79353	99.10452	99.52353	100
98.69358	98.69357868	98.85115	99.14729	99.54628	100
98.75598	98.75597786	98.90603	99.18802	99.56796	100
98.8154	98.81539664	98.95828	99.22681	99.58859	100
98.87198	98.87197738	99.00803	99.26374	99.60824	100
98.92586	98.92585562	99.05541	99.2989	99.62695	100
98.97716	98.97716046	99.10053	99.33239	99.64477	100
99.02601	99.0260148	99.14349	99.36428	99.66174	100
99.07254	99.07253569	99.1844	99.39464	99.6779	100
99.11683	99.11683458	99.22336	99.42355	99.69328	100
99.15902	99.1590176	99.26045	99.45109	99.70793	100



iv). Neumann – Neumann:

T1	T2	T3	T4	T5	T6
93.33333	90	90	90	90	91.66667
95.31333	91.98	90	90	90.99	92.65667
96.11721	92.78388	91.17612	90.58806	91.39194	93.05861
97.1422	93.80887056	91.78182	91.41487	91.90444	93.5711
97.91814	94.5848036	92.76792	91.92364	92.60363	94.2703

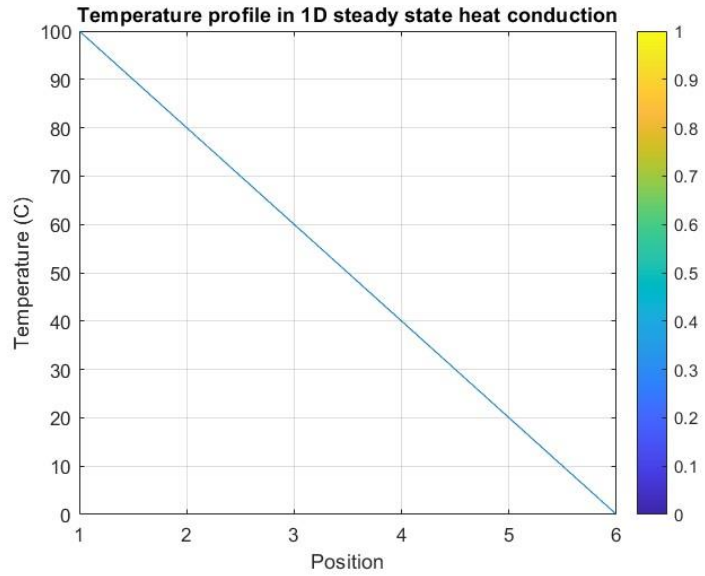
98.81891	95.48557521	93.34565	92.82906	93.18972	94.85639
99.52779	96.19445774	94.30991	93.35014	93.96549	95.63215
100.3884	97.05503727	94.85923	94.28576	94.58997	96.25664
101.0641	97.73072768	95.8229	94.8071	95.39927	97.06594
101.9108	98.57747697	96.35277	95.76223	96.03752	97.70419
102.5693	99.23599925	97.32347	96.27653	96.864	98.53067
103.4133	100.0799553	97.83763	97.24737	97.50504	99.17171
104.0613	100.7280153	98.81896	97.75104	98.34198	100.0087
104.9074	101.5740335	99.3186	98.7364	98.98097	100.6476
105.5476	102.214305	100.3125	99.2275	99.8257	101.4924
106.398	103.0646341	100.7977	100.2273	100.4604	102.127
107.0314	103.6980637	101.8055	100.7045	101.3119	102.9786
107.8872	104.553855	102.2757	101.7193	101.9411	103.6078
108.514	105.1806445	103.2984	102.1816	102.7994	104.466
109.3759	106.042594	103.7531	103.2119	103.4224	105.0891
109.996	106.662617	104.7916	103.6584	104.2874	105.954
110.8646	107.5312387	105.2299	104.7051	104.9038	106.5704
111.4776	108.1442172	106.2852	105.1348	105.7758	107.4424
112.3533	109.0199602	106.7061	106.1989	106.385	108.0517
112.9589	109.6255389	107.7792	106.6108	107.2645	108.9311
113.8422	110.5088378	108.1819	107.6931	107.8662	109.5328
114.4399	111.1066153	109.2738	108.0862	108.7534	110.4201
115.3312	111.9979124	109.6571	109.1879	109.3471	111.0138
115.9208	112.5874531	110.7689	109.5611	110.2425	111.9092
116.8205	113.4872087	111.1317	110.6833	110.8278	112.4945
117.4014	114.0680477	112.2645	111.0355	111.732	113.3986
118.3101	114.976745	112.6058	112.1792	112.3083	113.9749
118.8817	115.5483889	113.7608	112.5092	113.2216	114.8883



2). 1D Steady State Heat Conduction:
MATLAB Code:

```
% 1D steady state conduction
% Input parameter
L = 100; %(m)
N = 6; % Number of nodes
dx = L/(N-1); %distance between consecutive nodes
tol = 1e-4;

% Domain discretization
T_new = zeros(N,1); %Initialising the domain

x = linspace(0,L,N);

%BC
for i=1:N
    T_new(1,1)=100;
end

%Main Loop - Logic
error = 1;iter = 0;
while(error>tol)
    iter = iter + 1;
    T = T_new;
    for i = 2:N-1
        T_new(i) = (T(i+1)+T(i-1))/2;
    end
    error = max(max(abs(T-T_new)));
    figure(2);

    %plotting
    plot(T_new); shading flat; colorbar;
    xlabel('Position');
    ylabel('Temperature (C)');
    title('Temperature profile in 1D steady state heat conduction');
    grid on;
end
```

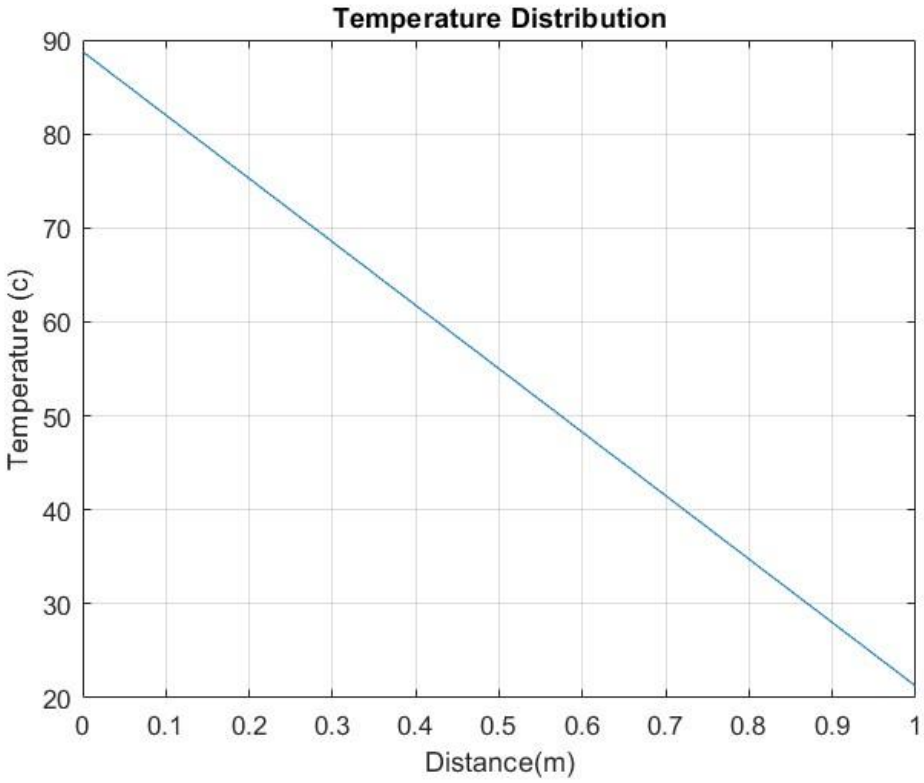
Output:



3). 1D Steady State Heat Conduction using Matrix Solver:
MATLAB Code:

```
L=1;
Nx=7;
dx=L/(Nx-1);
% k = 1;
Q = 0;
A = zeros(Nx,Nx);
A(1,1) = 2;
A(Nx,Nx)=2;
A(7,6)=-1;
for i = 2:Nx-1
    A(i,i-1) =-1;
    A(i,i) = 2;
    A(i,i+1) =-1;
    A(i-1,i) = -1;
end
disp(A)
b = zeros(Nx,1);
b(1) = 100;
b(Nx) = 10;
T= A\b;
x = linspace(0,L,Nx);
plot(x,T);
xlabel('Distance(m)');
ylabel('Temperature (c)');
title('Temperature Distribution')
grid on;
```

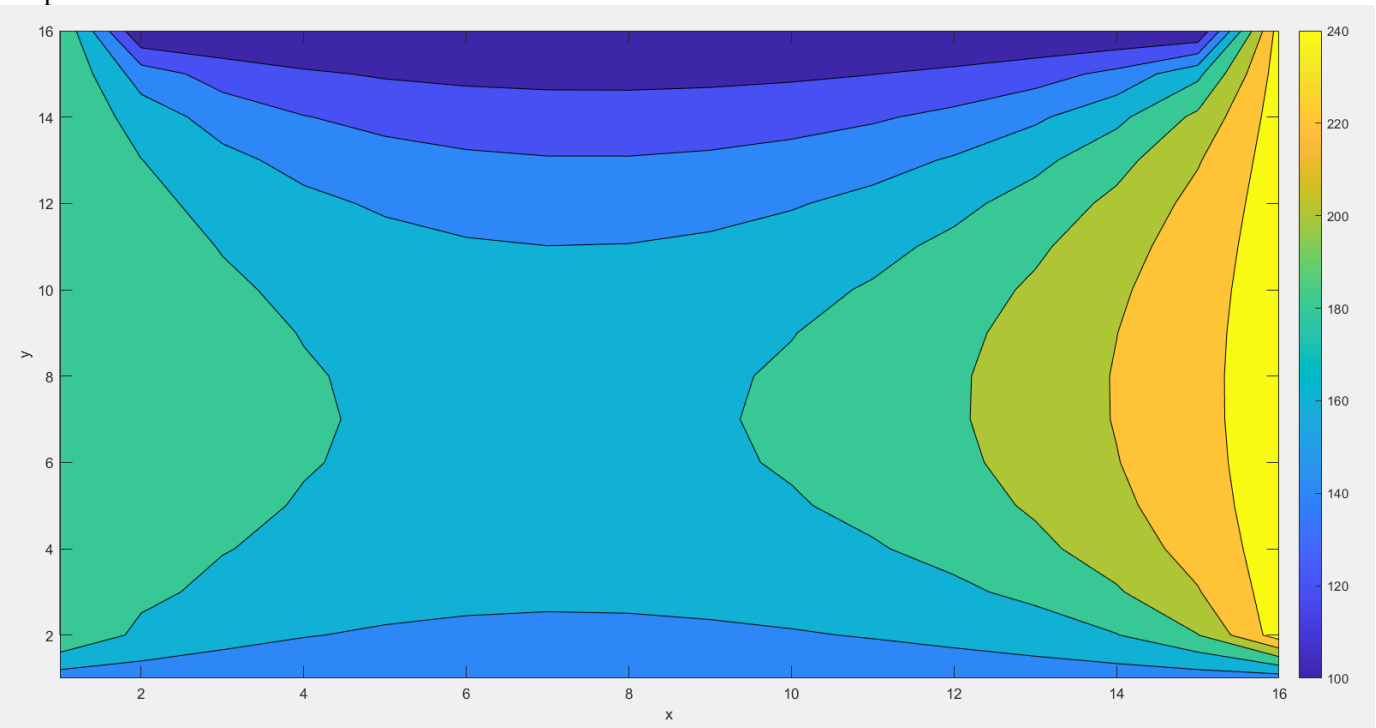
Output:



4). 2D Heat Conduction in Flat Plate:
MATLAB Code:

```
% Explicit method for 2D heat equation in flat plate
% Input Parameters
L = 0.75; %(m)
dx = 0.05; %(m)
dy = 0.05; %(m)
N = L/dx+1; %Number of Nodes
tol = 1e-4;
dt = 0.00015;

% Domain Discretisation
T_new = zeros(N,N); %Initialising the Domain

x = linspace(0,dx,N); %x-coordinate
y = linspace(0,dy,N); %y-coordinate

% Boundary Condition
for i=1:N
    for j=1:N
        T_new(1,j) =150;
        T_new(i,1) =200;
        T_new(N,j)=100;
        T_new(i,N)=250;
    end
end

%Main Loop - Logic
error =1;iter=0;
while(error > tol)
     iter=iter+1;
        T = T_new;
    for i=2:N-1
        for j =2:N-1
            T_new(i,j) = dt*((T(i+1,j)-2*T(i,j)+T(i-1,j))/dx^2 +(T(i,j+1)-
2*T(i,j)+T(i,j-1))/dy^2) + T(i,j);
        end
    end
    error = max(max(abs(T-T_new)));
figure(2);
%plotting
     contourf(T_new); shading flat;colorbar;
xlabel('x');ylabel('y')
pause(0.1);
end
```

Output:

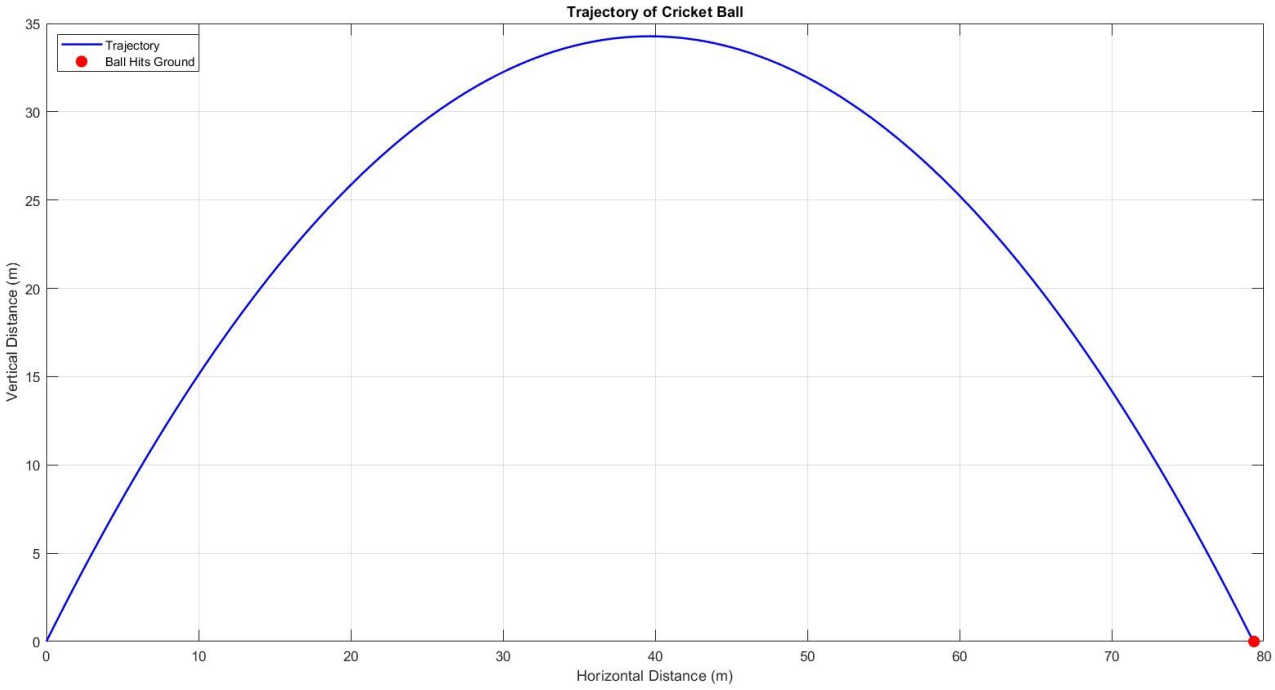


5). Calculation of distance of a cricket ball hit for a six:
MATLAB Code:

```
% Constants
g = 9.81; % Acceleration due to gravity (m/s^2)

% Initial conditions
v0 = 30; % Initial velocity (m/s)
launch_angle = deg2rad(60); % Launch angle in radians
delta_t = 0.01; % Time step (s)

% Initialize arrays
t = 0:delta_t:10; % Time array
x = zeros(size(t)); % Horizontal position
y = zeros(size(t)); % Vertical position
vx = zeros(size(t)); % Horizontal velocity
vy = zeros(size(t)); % Vertical velocity

% Initial conditions
x(1) = 0;
y(1) = 0;
vx(1) = v0 * cos(launch_angle);
vy(1) = v0 * sin(launch_angle);

% Simulation using Finite Difference Method
for i = 2:length(t)
    % Update velocities
    vx(i) = vx(i-1);
    vy(i) = vy(i-1) - g * delta_t;

    % Update positions
    x(i) = x(i-1) + vx(i) * delta_t;
    y(i) = max(0, y(i-1) + vy(i) * delta_t); % Ensure the ball doesn't go below
ground level
```

```
    % If the ball hits the ground, break the loop
    if y(i) == 0
        break;
    end
end

% Calculate distance and velocity when the ball hits the ground
distance_traveled = x(i);
horizontal_velocity = vx(i);

% Plot trajectory
plot(x(1:i), y(1:i), 'b-', 'LineWidth', 1.5);
hold on;
plot(x(i), y(i), 'ro', 'MarkerSize', 8, 'MarkerFaceColor', 'r');
xlabel('Horizontal Distance (m)');
ylabel('Vertical Distance (m)');
title('Trajectory of Cricket Ball');
grid on;
legend('Trajectory', 'Ball Hits Ground', 'Location', 'northwest');

% Display results
fprintf('Distance Traveled: %.2f meters\n', distance_traveled);
fprintf('Horizontal Velocity at Ground: %.2f m/s\n', horizontal_velocity);
```

Output:



>> CricP
Distance Traveled: 79.35 meters
Horizontal Velocity at Ground: 15.00 m/s