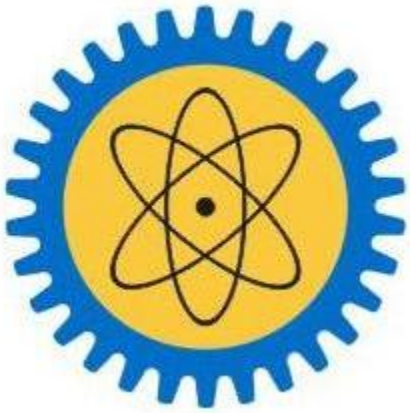


BUSITEMA UNIVERSITY

Pursuing Excellence

FACULTY OF ENGINNEERING AND TECHNOLOGY

COURSE UNIT: COMPUTER PROGRAMMING

MATLAB

LECTURER: MR. BENDICTO. S. MASERUKA

A REPORT ABOUT THE MATLAB ASSIGNMENT.

SUBMITTED BY GROUP ONE

GITHUB LINK: https://github.com/MATLABGROUPONE2025/Group-1-assignments.git

DATE OF SUBMISSION:……../……./……

DECLARATION.

We group one members hereby certify and confirm that the information in this report is out of our own efforts, research and it has never been submitted in any institution for any academic award.

NAME	REG. NO	COURSE	GITHUB LINK	SIGNATURE
IKEO JESCA	BU/UP/2024/1027	WAR	https://github.com/Ikeo-1234/Group-one-assignment.git	
MUTARYEBWA LUKE AHEBWA	BU/UP/2024/1042	WAR	https://github.com/mutaryebwa/group-one-assignments-.git	
ATUSIMIRWE OLIVIA	BU/UG/2024/2597	WAR	https://github.com/oliviaatusiimirwe/atusiimirwe.git	
SSENTUDE IBRAHIM	BU/UG/2024/2588	WAR	https://github.com/ssentudde/ssentuddeibrahim.git	
ATIM GLORIA VERONICA	BU/UP/2024/1017	WAR	https://github.com/Atim658/group-one-assignments.git	
MUGISHA PETER	BU/UP/2024/3225	AMI	https://github.com/MugishaPeter/Assignments-group-1.git	
NAKABUGU HAULA KASULE	BU/UP/2024/3741	AMI	https://github.com/haula-git/Nakabugo-haula-.git	
MAKAAYI HAKIM	BU/UP/2024/3819	PTI	https://github.com/Makaayi/Makaayi-Hakiim	
MUTAKA AMOS	BU/UP/2024/4451	WAR	https://github.com/mutaka03/Mutaka-Amos.git	
LUCKY DOROTHYN	BU/UP/2024/0982	MEB	https://github.com/Lucky-1234-max/Group-1-assignment-.git	

DATE……

APPROVAL.

This is to confirm that this report has been written and presented by GROUP ONE, giving the details of the MATLAB assignments and what they learnt.

LECTURER.

NAME: …….

SIGNATURE: ……....

DATE: …….

ACKNOWLEDGMENT.

First and foremost, we would like to thank the Almighty God for giving us the strength to carry on with our assignment as group one. We would love to extend our gratitude to all the persons with whose help we managed to make it this far

The willingness of each one of us to invest time and provide constructive feedback has been immensely valuable in this assignment. Finally, we would like to express our gratitude to all the sources and references that we used.

DEDICATION.

We dedicate this report to all the individuals especially Group One members, who have been there for us in the process of formulating and producing this report. To our lecturer Mr. Bendicto. S. Maseruka, whose guidance and expertise have been invaluable, your mentorship and insightful feedback have shaped our understanding.

ABSTRACT.

We started our assignment in the university library out of which were we exposed to various codes in the different sections through Group One members, we generated the codes necessary from the work that we were taught in the lecture and also researched on more information that can help us complete the assignment.

1 Contents

DECLARATION. **2**
APPROVAL. **3**
ACKNOWLEDGMENT. **4**
DEDICATION. **5**
ABSTRACT. **6**
2 CHAPTER ONE **9**
2.1 QUESTION ONE 9
3 USING THE LAGRANGE METHOD **9**
3.1 LAGRANGE METHOD OF NUMERICAL APPROXIMATION. 9
3.2 FLOWCHART FOR LAGRANGE METHOD 10
3.3 MATLAB CODE FOR LAGRANGE EXPLAINED 11
3.4 PLOT 13
3.5 OUTPUT 14
4 USING THE FIXED POINT ITERATION METHOD **14**
4.1 A FLOW CHART OF THE WORK 14
4.2 A pseudo code of the work 15
4.3 THE MATLAB CODE 16
4.4 EXPLANATION FOR THE CODE 16
4.5 STEPS OF ANALYSIS 17
5 USING THE SECANT METHOD **19**
What is the Secant Method? 19
5.1 A flow chart of the number 20
5.2 A pseudo code for the work 20
5.3 THE MATLAB CODE 22
5.4 EXPLANATION OF THE CODE 23
6 USING TRAPEZOIDAL RULE **24**
7 NEWTON RAPHSON METHOD **28**
8 CHAPTER TWO **31**
8.1 QUESTION TWO 31
9 USING THE D OPERATOR METHOD **31**
9.1 Flow chart of the number 31

9.2 A PSEUDO CODE OF THE WORK 31
9.3 STEPS OF ANALSIS OF DATA 32
10 USING EULER METHOD 35
10.1 PSUEDOCODE 35
11 USING THE RUNGE-KUTTA 2ND ORDER METHOD 42
11.1 RUNGE KUTTE 2ND ORDER ALGORITHM DEVELOPMENT. 42
11.2 FLOW CHART FOR RUNGE KUTTE 2ND ORDER 43
11.3 MATLAB CODE FOR RUNGE KUTTE EXPLAINED. 44
11.4 PLOT 45
11.5 OUTPUT 46
12 LAPLACE TRANSFORM SOLUTION FOR RLC CIRCUIT 47
12.1 Flowchart 47
12.2 MATLAB Code 48
12.3 THE OUTPUT OF THE CODE 49

2 CHAPTER ONE

2.1 QUESTION ONE

In your groups , utillize the knowledge of algorithm development , control structures and modules 1 to4 on the following problems;

a). All numerical approximation methods, for finding the solutions to functions but not limited to; newtons raphsons method , secant method ,etc.

3 USING THE LAGRANGE METHOD

3.1 LAGRANGE METHOD OF NUMERICAL APPROXIMATION.

The **Lagrange interpolation method** is a numerical technique used to estimate the value of a function when we only know it at a finite number of points.

- ✓ Suppose you have data pairs (x0, y0), (x1, y1) …, (xn, yn) (x_0, y_0), (x_1, y_1), \dots, (x_n, y_n) (x_0, y_0), (x1, y1) …, (xn, yn).
- ✓ The Lagrange method builds a single polynomial Pn(x)P_n(x)Pn(x) of degree '**n**' that passes exactly through all those points.

Given a set of $k+1$ nodes $\{x_0, x_1, \ldots, x_k\}$, which must all be distinct, $x_j \neq x_m$ for indices $j \neq m$, the **Lagrange basis** for polynomials of degree $\leq k$ for those nodes is the set of polynomials $\{\ell_0(x), \ell_1(x), \ldots, \ell_k(x)\}$ each of degree k which take values $\ell_j(x_m) = 0$ if $m \neq j$ and $\ell_j(x_j) = 1$. Using the Kronecker delta this can be written $\ell_j(x_m) = \delta_{jm}$. Each basis polynomial can be explicitly described by the product:

$$\ell_j(x) = \frac{(x-x_0)}{(x_j-x_0)} \cdots \frac{(x-x_{j-1})}{(x_j-x_{j-1})} \frac{(x-x_{j+1})}{(x_j-x_{j+1})} \cdots \frac{(x-x_k)}{(x_j-x_k)}$$

$$= \prod_{\substack{0 \leq m \leq k \\ m \neq j}} \frac{x-x_m}{x_j-x_m}.$$

Notice that the numerator $\prod_{m \neq j}(x-x_m)$ has k roots at the nodes $\{x_m\}_{m \neq j}$ while the denominator $\prod_{m \neq j}(x_j-x_m)$ scales the resulting polynomial so that $\ell_j(x_j) = 1$.

The Lagrange interpolating polynomial for those nodes through the corresponding *values* $\{y_0, y_1, \ldots, y_k\}$ is the linear combination:

$$L(x) = \sum_{j=0}^{k} y_j \ell_j(x).$$

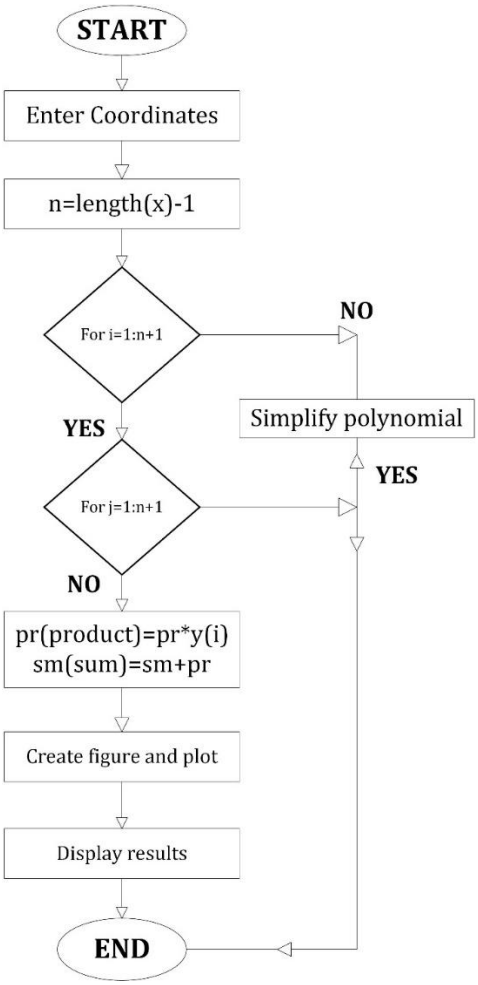
Figure 1 Basic Lagrange method of numerical approximation.

3.2 FLOWCHART FOR LAGRANGE METHOD

A flow chart refers to a visual representation of an algorithm that uses standardized symbols, shapes, and arrows to illustrate the sequence of steps, decisions, and flow of control in a program or system.

Flow charts consist of several key elements:

- **Oval**: Start/End points
- **Rectangle**: Process steps
- **Diamond**: Decision points
- **Arrows**: Flow direction



3.3 MATLAB CODE FOR LAGRANGE EXPLAINED

```
x = input("Enter list of abscissas:");
y = input("Enter list of ordinates:");
n = length(x)-1; %size of the data pairs minus 1
sm = 0; %Summation variable
```

This reads two vectors that the user inputs. It read the X and Y coordinates. For this code to run, x any y must be same length and correspond point wise.

Sets '`n = length(x)-1`' so later loops run over all data points.

```
tic
for i = 1:n+1
    pr = 1;
    for j = 1:n+1
        if j ~= i
            xp = sym("xp");
            pr = pr * (xp-x(j))/(x(i)-x(j));
        end
```

```
    end
    pr = pr*y(i); %multiply the current y with pr
    sm = sm+pr; %add current term to polynomial
end
toc
```

This starts timing with `tic`.

Outer loop over data points **`i`**. For each **`i`** it computes the ***`i-th`*** Lagrange basis polynomial

Multiplies **L_i`(xp)`** by **`y(i)`** and adds to **`sm`**. After the loop **`sm`** is the full symbolic Lagrange interpolating polynomial in the symbolic variable **`xp`**.

`toc` prints elapsed time.

```
sm;

disp("=== Lagrange Interpolating polynomial ===")
sm = simplify(sm); %simplify the equation

figure("Name","Lagrange")
hold on;
plot(x,y,"bo") %plotting the coordinates
fplot(sm,[min(x)-1,max(x)+1],"g") %plotting the function

disp("=== Evaluate sm at point xp ===")
xp_apx = input("Enter the point to be approximated:");
y_val = subs(sm,sym('xp'),xp_apx); % substitute 'xp' with xp_val

% Plot the specific point approximated
plot(xp_apx,y_val,"c*", 'MarkerSize', 10, 'LineWidth', 2)

%Annotating the plot
```

```
xlabel('x')
ylabel('y')
title('Numerical approximation using Lagrange method')
legend('Cordinates','Function','Approximated point')
grid on;

% Display the result
fprintf('At x = %.4f, y = %.4f\n', xp_apx, double(y_val));

hold off
```

- Adds labels, title, legend, and grid to the plot.
- Prints the approximated numeric result to console (formats to 4 decimal places).

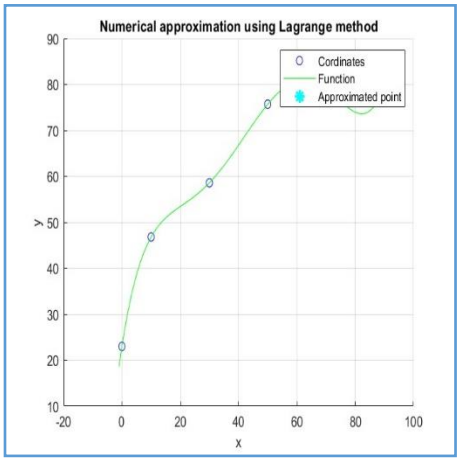
`hold off` ends plot additions.

Legend order must match plotted objects

‘fprintf’ Prints the approximated numeric result to console (formats to 4 decimal places).

The output is shown as below.

3.4 PLOT

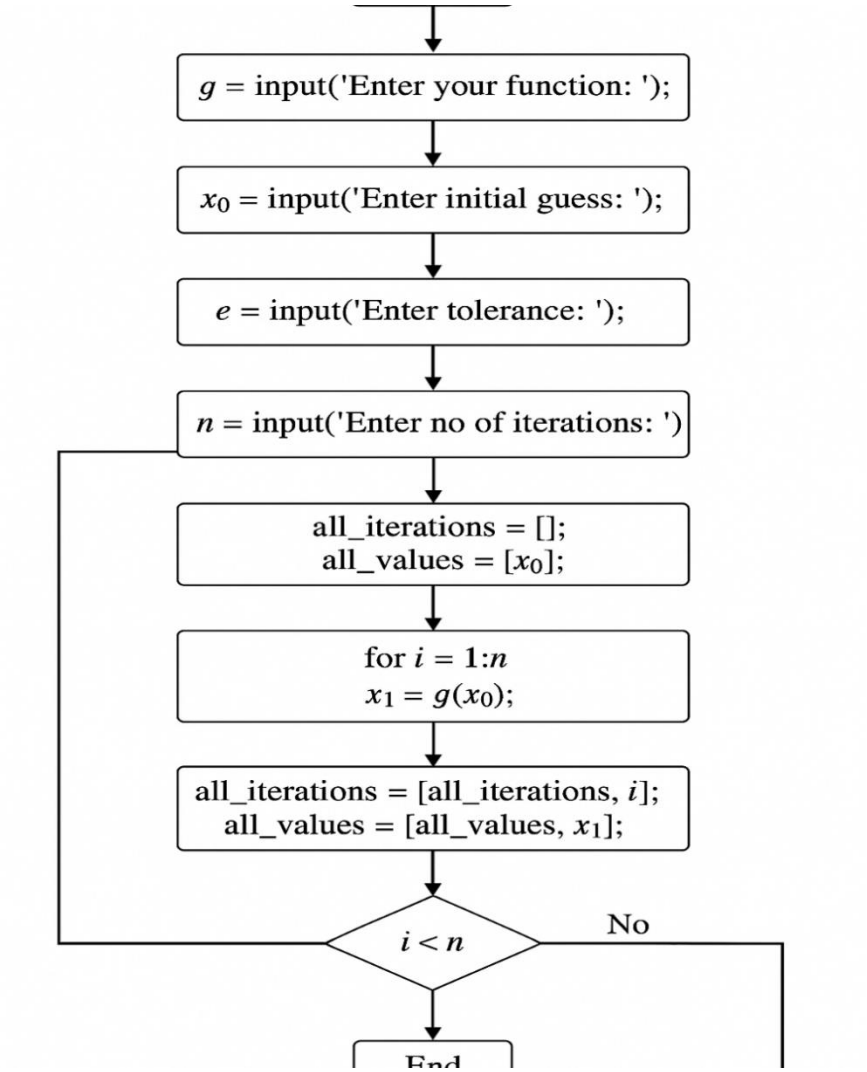


3.5 OUTPUT

```
LaGrange Example
Enter list of abscissas: [0 10 30 50 70 90]
Enter list of ordinates: [23.0 46.8 58.6 75.7 78.9 80.2]
Elapsed time is 1.257555 seconds.
=== Lagrange Interpolating polynomial ===
=== Evaluate sm at point xp ===
Enter the point to be approximated:60
At x = 60.0000, y = 80.6577
```

4 USING THE FIXED POINT ITERATION METHOD

4.1 A FLOW CHART OF THE WORK



4.2 A pseudo code of the work

BEGIN

INPUT g // function

INPUT x0 // initial guess

INPUT e // tolerance

INPUT n // number of iterations

all_iterations ← []

all_values ← [x0]

FOR i ← 1 TO n DO

x1 ← g(x0)

all_iterations ← all_iterations + [i]

all_values ← all_values + [x1]

x0 ← x1

END FOR

PLOT all_values against iteration numbers

END

4.3 THE MATLAB CODE

```
%input data
g = input('Enter your function: ');
xO = input('Enter initial guess: ');
e = input('Enter tolerance: ');
n = input('Enter no of iterations: ');
% Store all iterations
all_iterations = [];
all_values = [xO];
% Start with initial guess
for i = 1:n
    x1 = g(xO);
% Store each iteration
all_iterations = [all_iterations, i];
all_values = [all_values, x1];
    xO = x1;
end
% Plot all points
figure;
plot(0:n, all_values, 'bo-', 'LineWidth', 2, 'MarkerFaceColor', 'blue', 'MarkerSize',
6);
xlabel('Iteration Number');
ylabel('x value');
title('Fixed-Point Iteration Convergence');
grid on;
```

4.4 EXPLANATION FOR THE CODE

```
%input data
g = input('Enter your function: ');
xO = input('Enter initial guess: ');
e = input('Enter tolerance: ');
n = input('Enter no of iterations: ');
```

What each line does:

g = input('Enter your function: ');

Prompts the user to enter a mathematical function

Example: User might type @(x) x^2 - 2 or @(x) exp(-x)

Stores the function in variable g

xO = input('Enter initial guess: ');

Prompts the user to enter a starting value for the iteration

Example: User might type 1.5

Stores the initial guess in variable xO

e = input('Enter tolerance: ');

Prompts the user to enter the desired accuracy level

Example: User might type 0.001 or 1e-6

Stores the tolerance in variable e

n = input('Enter no of iterations: ');

Prompts the user to enter the maximum number of iterations allowed

Example: User might type 20 or 50

Stores the iteration limit in variable

4.5 STEPS OF ANALYSIS

Step 1: Define the Function

f = @(x) x^2 - 4;

Creates a function f(x) = x^2 - 4

We're trying to find where this function equals zero (the roots)

The roots are x = 2 and x = -2 (since $2^2 - 4 = 0$ and $(-2)^2 - 4 = 0$)

Step 2: Initial Guesses

x0 = 1; x1 = 3;

x0 = 1 is our first guess

x1 = 3 is our second guess

The secant method needs two starting points to begin

Step 3: Initialize Storage Arrays

iterations = [];

values = [x0, x1];

values stores all the x-values we'll compute

Starts with our two initial guesses: [1, 3]

The Secant Method Loop (Runs 5)

```
 for i = 1:5
    x2 = x1 - f(x1)*(x1 - x0)/(f(x1) - f(x0));
    x0 = x1; x1 = x2;
    values = [values, x1];
end
```

First Iteration (i=1):

$f(x1) = f(3) = 3^2 - 4 = 5$

$f(x0) = f(1) = 1^2 - 4 = -3$

x2 = 3 - 5*(3-1)/(5 - (-3)) = 3 - 5*2/8 = 3 - 1.25 = 1.75

Now: x0 = 3, x1 = 1.75

values = [1, 3, 1.75]

Second Iteration (i=2):

$f(x1) = f(1.75) = 1.75^2 - 4 = -0.9375$

f(x0) = f(3) = 5

x2 = 1.75 - (-0.9375)*(1.75-3)/(-0.9375 - 5) = 2.2093

Now: x0 = 1.75, x1 = 2.2093

values = [1, 3, 1.75, 2.2093]

This continues until we get close to the actual root x = 2

Step 5: Plotting the Results

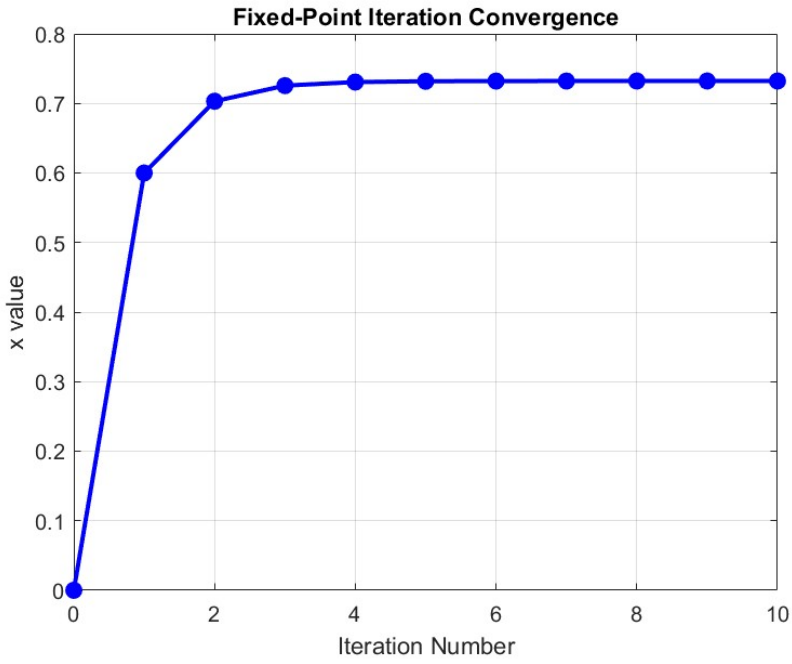
plot(0:length(values)-1, values, 'ro-', 'LineWidth', 2);

0:length(values)-1 creates x-axis: [0, 1, 2, 3, 4, 5, 6] (iteration numbers)

values contains our computed x-values: [1, 3, 1.75, 2.2093, ...]

'ro-' means red circles connected by lines

Shows how each iteration gets closer to the true root



5 USING THE SECANT METHOD

What is the Secant Method?

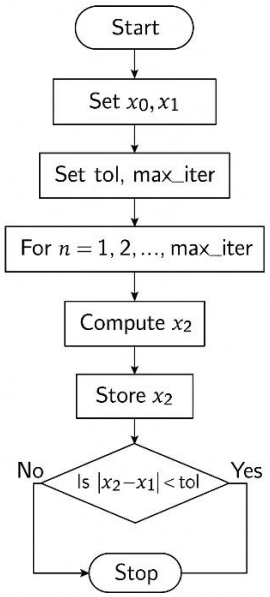
- The **Secant Method** is a numerical method to find roots of a nonlinear equation f(x)=0f(x) = 0f(x)=0.

- Instead of using derivatives (like Newton-Raphson), it approximates the derivative using a **secant line** between two points.
- Formula:

$$x_{n+1} = x_n - f(x_n)\frac{(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}$$

Question:
Given a non-linear equation 2x-5x+2=0. Use Secant Method to find the root of this equation correct upto 4 decimal places (Ɛ=10-4)

5.1 A flow chart of the number



5.2 A pseudo code for the work

BEGIN

```
    DEFINE f(x) = 2x^2 - 5x + 2
    SET x0 = 0
    SET x1 = 1
    SET tol = 1e-4
    SET max_iter = 100

    STORE iters = [x0, x1]
    STORE fx_vals = [f(x0), f(x1)]

    FOR n = 1 TO max_iter DO
        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
        APPEND x2 to iters
        APPEND f(x2) to fx_vals
        PRINT iteration number, x2, f(x2)

        IF |x2 - x1| < tol THEN
            PRINT root found at x2
            EXIT loop
        END IF

        x0 = x1
        x1 = x2
    END FOR

    PLOT f(x) over interval
    PLOT horizontal line y = 0
    PLOT iters vs fx_vals
END
```

5.3 THE MATLAB CODE

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

% Initial guesses
x0 = 0;
x1 = 1;

% Tolerance
tol = 1e-4;
max_iter = 100;

% Store iterations for plotting
iters = [x0, x1];
fx_vals = [f(x0), f(x1)];

% Secant Method iterations
for n = 1:max_iter
    % Compute next approximation
    x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0));

    % Store values for plotting convergence
    iters (end+1) = x2;
    fx_vals (end+1) = f(x2);

    % Display current iteration
    fprintf('Iteration %d: x = %.6f, f(x) = %.6f\n', n, x2, f(x2));

    % Check stopping criteria
    if abs(x2 - x1) < tol
        fprintf('\nRoot found at x = %.6f with f(x) = %.6f\n', x2, f(x2));
        break;
    end

    % Update for next iteration
    x0 = x1;
    x1 = x2;
end

%% Plot the function and root
x = linspace(-2, 3, 500);
y = f(x);

figure;
plot(x, y, 'b-', 'LineWidth', 1.5); hold on;
yline(0, 'k--');
plot(iters, fx_vals, 'ro-', 'LineWidth', 1.5, 'MarkerSize', 6);
xlabel('x');
ylabel('f(x)');
```

```
title('Secant Method Convergence');
legend('f(x)', 'y=0', 'Secant Iterations');
grid on;
```

5.4 EXPLANATION OF THE CODE

1. **Define the function**
2. f = @(x) 2*x.^2 - 5*x + 2;

   This is the equation we want to solve.

3. **Choose two starting points**
4. x0 = 0;
5. x1 = 1;

   These are the initial guesses for the root.

6. **Set tolerance and max iterations**
7. tol = 1e-4;
8. max_iter = 100;

   The loop stops when the root is accurate to 4 decimal places or after 100 steps.

9. **Secant Method loop**
10. x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0));

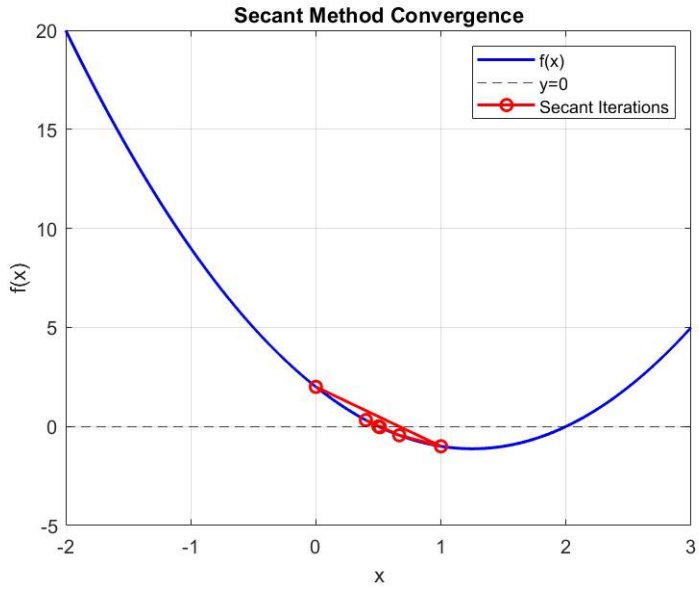
    This formula finds a better root estimate by drawing a secant line between the two guesses and finding where it cuts the x-axis.

    - If the new guess is close enough (abs(x2 - x1) < tol), the loop stops.
    - Otherwise, update the guesses (x0 = x1; x1 = x2;) and continue.

11. **Display the root**
12. fprintf('Root found at x = %.4f\n', root);

    Prints the final root to 4 decimal places.

13. **Plot the function and root**
    - The curve of f(x)f(x)f(x) is drawn.
    - A horizontal line at y=0y=0y=0 marks the x-axis.
    - The root is shown as a red point.



In short: The code keeps improving guesses with the **Secant Method** until the root is found within the required accuracy, then it shows the result and plots it.

6 USING TRAPEZOIDAL RULE

One of the real world applications of trapezoidal rule is in the estimation of distances, estimation of total rainfall from precipitation data ,Calculating drug concentration in bloodstream from metabolic rate data etc

Example

A drone's velocity was recorded every 2 seconds for a total of 8 seconds and the velocities include 5, 8, 12, 10 and 6m/s. Use trapezoidal rule to estimate the total distance travelled by the drone

```
%% Drone Distance Estimation using Trapezoidal Rule
% Given data: Time and velocity measurements
time = [0, 2, 4, 6, 8];
velocity = [5, 8, 12, 10, 6];
%% METHOD 1: Trapezoidal Rule Calculation
% Using the formula: Distance = (Δt/2) * [v0 + 2v1 + 2v2 + ... + 2vn-1 + vn]
dt = time(2) - time(1);
```

```
n = length(velocity) - 1;
% Apply trapezoidal rule
distance_trapz = (dt/2) * (velocity(1) + velocity(end) + 2*sum(velocity(2:end-1)));
```

This code explains the trapezoidal rule in a real world example of estimating the distance moved by a drone. There was given data which included time (seconds) and velocity (m/s) and this had to be put in a formula above

```
%% METHOD 2: Using MATLAB's built-in trapz function
distance_builtin = trapz(time, velocity);
```

this code is the second trapezoidal method which is mat lab inbuilt.

```
%% METHOD 3: Numerical integration with verification
distance_alt = 0;
for i = 1:length(time)-1
    segment_distance = (velocity(i) + velocity(i+1)) * dt / 2;
    distance_alt = distance_alt + segment_distance;
end
```

this code explains the third trapezoidal method that calculates the total distance by summing the area of each trapezoid between consecutive time points. For each 2-second interval, it computes the average velocity × time and accumulates these segment distances.

```
%% Visualization
figure('Position', [100, 100, 1200, 800]);
% Plot 1: Velocity vs Time with trapezoids
subplot(2,2,1);
plot(time, velocity, 'bo-', 'LineWidth', 2, 'MarkerSize', 8, 'MarkerFaceColor', 'b');
hold on;
% Shade the trapezoids
for i = 1:length(time)-1
    x = [time(i), time(i+1), time(i+1), time(i)];
    y = [0, 0, velocity(i+1), velocity(i)];
    fill(x, y, 'r', 'FaceAlpha', 0.3, 'EdgeColor', 'r');
end
grid on;
xlabel('Time (s)');
ylabel('Velocity (m/s)');
title('Velocity vs Time with Trapezoidal Areas');
legend('Velocity data', 'Trapezoidal areas', 'Location', 'northeast');
```

This code creates a plot showing velocity data points connected by lines and shades the area under the velocity curve using red trapezoids. The trapezoidal areas represent the distance traveled during each time interval, visually demonstrating how the total distance is calculated. The subplot plots the graph on the left upper side of the same figure window

```
% Plot 2: Cumulative distance
subplot(2,2,2);
cumulative_distance = zeros(1, length(time));
for i = 2:length(time)
```

```
cumulative_distance(i) = cumulative_distance(i-1) + ...
                            (velocity(i-1) + velocity(i)) * dt / 2;
end
plot(time, cumulative_distance, 'g-s', 'LineWidth', 2, 'MarkerSize', 6,
'MarkerFaceColor', 'g');
grid on;
xlabel('Time (s)');
ylabel('Cumulative Distance (m)');
title('Cumulative Distance Traveled');
for i = 1:length(time)
end
```

This plot shows how the total distance accumulates over time. It starts at zero and increases at each time point by adding the trapezoidal area from the previous interval. The green stepped line visually displays the running total distance traveled by the drone at each measurement time. The subplot plots the graph on the left downer side of the same figure window

```
% Plot 3: Comparison of methods
subplot(2,2,3);
methods = {'Manual Trapz', 'MATLAB trapz', 'Segment Calc'};
distances = [distance_trapz, distance_builtin, distance_alt];
bar(distances);
set(gca, 'XTickLabel', methods);
ylabel('Distance (m)');
title('Comparison of Calculation Methods');
grid on;
% Add value labels on bars
for i = 1:length(distances)
    text(i, distances(i)+0.1, sprintf('%.3f', distances(i)), ...
         'HorizontalAlignment', 'center', 'FontWeight', 'bold');
end
```

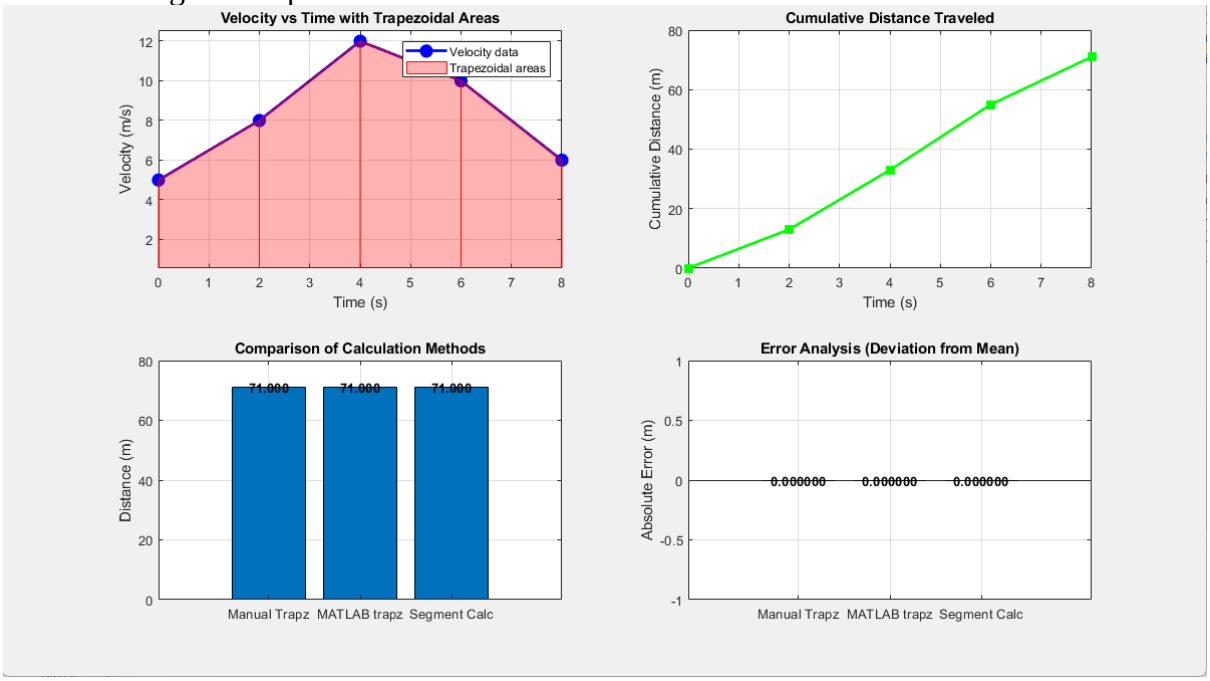
This plot creates a bar chart comparing the distance results from three different calculation methods. It displays the numerical values above each bar to verify that all methods give identical results, confirming the accuracy of the trapezoidal rule implementation. . The subplot plots the graph on the right upper side of the same figure window

```
% Plot 4: Error analysis
subplot(2,2,4);
errors = abs(distances - mean(distances));
bar(errors);
set(gca, 'XTickLabel', methods);
ylabel('Absolute Error (m)');
title('Error Analysis (Deviation from Mean)');
grid on;
for i = 1:length(errors)
    text(i, errors(i)+0.001, sprintf('%.6f', errors(i)), ...
         'HorizontalAlignment', 'center', 'FontWeight', 'bold');
end
```

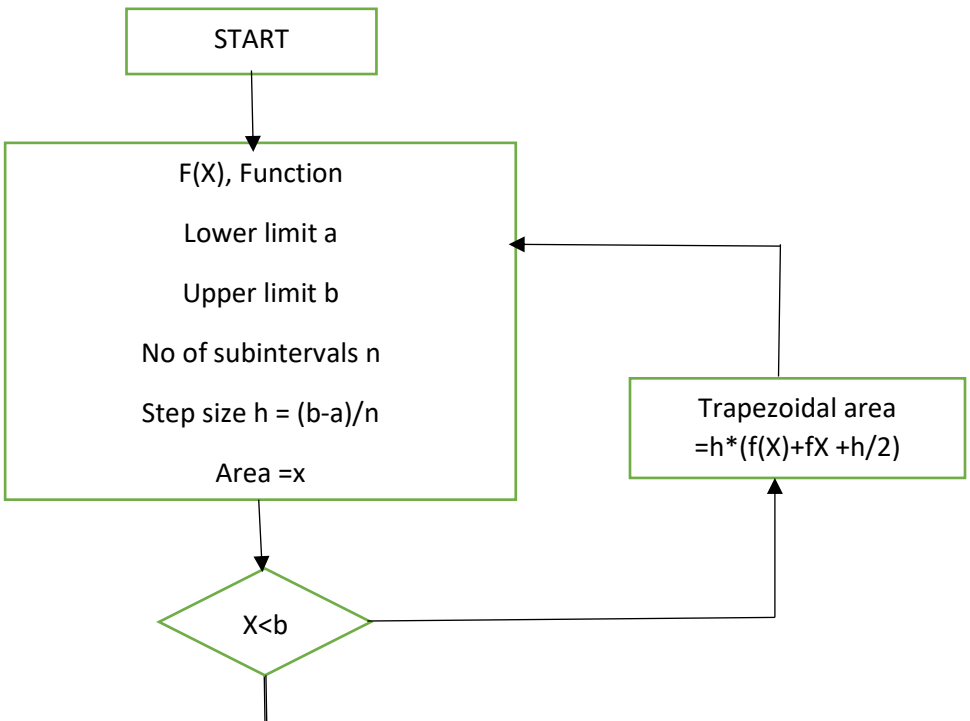
This plot shows the error analysis by calculating how much each method's result deviates from the average distance. Since all three methods give identical results, the errors are zero,

confirming perfect agreement between the different calculation approaches. . The subplot plots the graph on the right lower side of the same figure window

The following are the plots



FLOW CHART



7 NEWTON RAPHSON METHOD

The Newton–Raphson method is an iterative numerical method used to find approximate roots of nonlinear equations f(x)=0f(x) = 0f(x)=0.

Formula

Starting with an initial guess x0x_0x0, the iteration is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- $f(x)$ = the function
- $f'(x)$ = derivative of the function
- Repeat until the change $|x_{n+1} - x_n|$ is very small (within tolerance).

Steps

Choose an initial guess x0x_0x0.

Compute f(x0)f(x_0)f(x0) and f′(x0)f'(x_0)f′(x0).

Update using the formula.

Repeat until convergence.

CODE EXPLANATION

1 Define the function and its derivative

```
f  = @(x) 2.^x - 5.*x + 2;
df = @(x) log(2).*2.^x - 5;
```

- The equation $f(x) = 2^x - 5x + 2$.
- Its derivative $f'(x) = \ln(2) \cdot 2^x - 5$.

2 set initial guess and parameters

```
x = 0.5; tol = 1e-4; maxIter = 50;
```

- Start at $x = 0.5$, stop when error < 10^{-4}, or after 50 steps.

3 Iterate with Newton–Raphson formula

```
x_new = x - f(x)/df(x);
```

Improves the guess each step.

Loop stops when the change between guesses is very small.

4 Print results

Shows iteration number, new guess, and f(x)f(x)f(x).

Prints the final root once found.

5 Plot the function and root

Blue curve = function.

Dashed line = x-axis.

Red dot = the computed root.

The code uses Newton–Raphson iterations to solve 2x−5x+2=02^x - 5x + 2 = 02x−5x+2=0, prints the steps, and plots the function with the root marked.

8 CHAPTER TWO

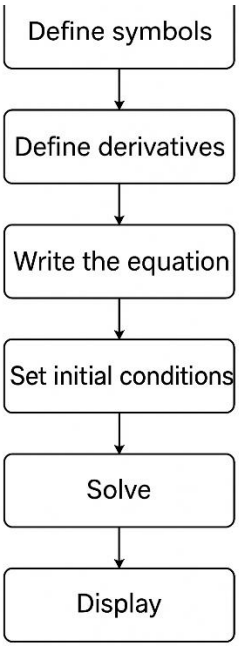
8.1 QUESTION TWO

All methods for the solving differential equations numerically there include but not limited to; euler methods etc

9 USING THE D OPERATOR METHOD

Solving y" + 3y' + 2y = 0 in matlab;

9.1 Flow chart of the number



9.2 A PSEUDO CODE OF THE WORK

BEGIN

DEFINE symbolic variables t, y(t)

DEFINE first derivative D = dy/dt

DEFINE second derivative D2 = d^2y/dt^2

SET ODE: D2 + 3*D + 2*y = 0

SET initial conditions:

y(0) = 1

D(0) = 0

SOLVE ODE with initial conditions → ySol(t)

DISPLAY ySol(t)

PLOT ySol(t) for t in [0, 5]

END

9.3 STEPS OF ANALSIS OF DATA

1: Tell MATLAB we're working with symbols

syms t y(t)

What this does: Creates symbolic variables t (time) and y(t) (a function of time). Think of this as telling MATLAB "we're doing math with letters, not numbers."

STEP 2: Define derivatives using D operator

D = diff(y, t);

D2 = diff(y, t, 2);

What this does:

D = diff(y, t) means "D represents the first derivative of y with respect to t" (dy/dt)

D2 = diff(y, t, 2) means "D2 represents the second derivative of y with respect to t" (d^2y/dt^2)

Analogy: If y is position, then:

D is velocity (first derivative)

D2 is acceleration (second derivative)

STEP 3: Write the differential equation

ode = D2 + 3*D + 2*y == 0;

What this does: Creates the equation y'' + 3y' + 2y = 0 using our D operators.

hence:

D2 = y'' (second derivative)

3*D = 3 × y' (3 times first derivative)

2*y = 2 × y (2 times the function)

== 0 means the sum equals zero

STEP 4: Set initial conditions

```
cond1 = y(0) == 1;    % At time t=0, y=1
cond2 = D(0) == 0;    % At time t=0, derivative y'=0
conds = [cond1, cond2];
```

What this does:

y(0) == 1 means "when t=0, y=1"

D(0) == 0 means "when t=0, the derivative y'=0" (starting with zero slope)

conds = [cond1, cond2] groups both conditions together

Real-world example: If y is the position of a car:

y(0) == 1 means the car starts at position 1

D(0) == 0 means the car starts with zero velocity (stopped)

STEP 5: Solve the equation

```
ySol(t) = dsolve(ode, conds);
```

What this does: dsolve means "differential equation solver" - it finds the exact mathematical formula that satisfies both the equation AND the initial conditions.

Output: It will find something like y(t) = 2e^(-t) - e^(-2t) (the exact solution)

STEP 6: Display the solution

```
fprintf('Analytical Solution:\n')
pretty(ySol)
```

What this does:

fprintf prints text to screen

pretty(ySol) displays the solution in a nice mathematical format

STEP 7: Plot the solution

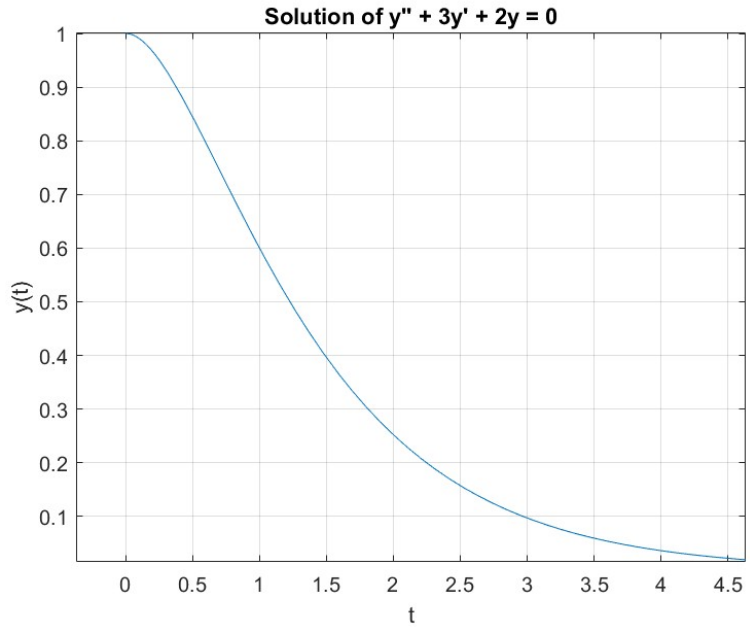
fplot(ySol, [0 5])

xlabel('t')

ylabel('y(t)')

title('Solution of y"" + 3y" + 2y = 0')

grid on



What this does:

fplot(ySol, [0 5]) plots the solution from t=0 to t=5

xlabel, ylabel, title add labels

grid on adds grid lines for easier reading

What This Equation Represents:

This is a damped oscillator equation:

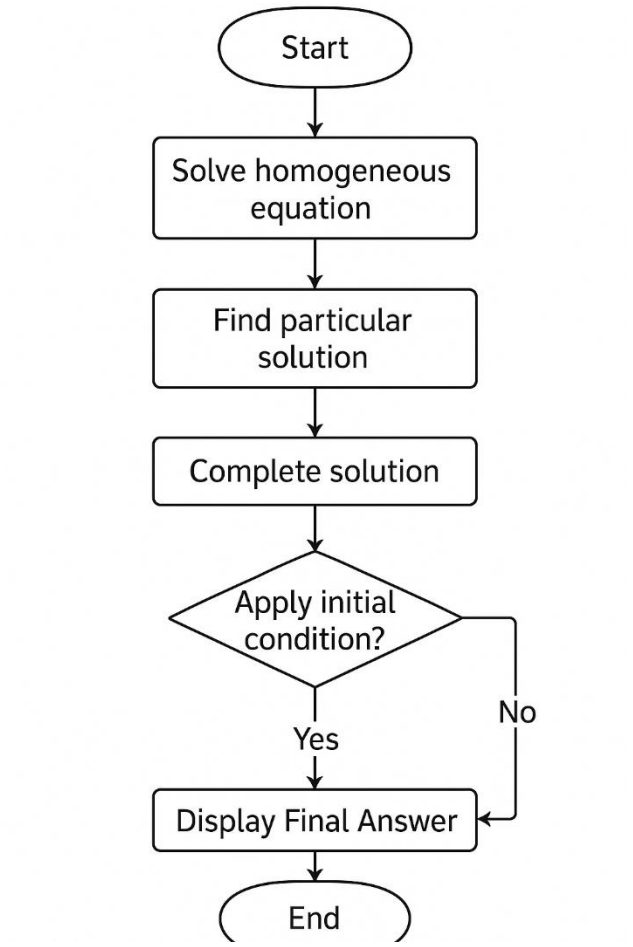
A spring-mass system with friction

The solution shows how the position changes over time

The system starts at position 1 and gradually comes to rest at position 0

10 USING EULER METHOD

A flow chart of the data



10.1 PSUEDOCODE

Problem: Solve damped driven oscillator with zero initial conditions
Differential Equation: x'' + 2x' + 5x = 3cos(2t)
Initial Conditions: x(0) = 0, x'(0) = 0

Step 1: Solve homogeneous equation
Characteristic equation: $r^2 + 2r + 5 = 0$
Roots: r = -1 ± 2i
Homogeneous solution: x_h(t) = e^(-t)[A·cos(2t) + B·sin(2t)]

STEP 2: Find particular solution using complex method
Complexified equation: z" + 2z' + 5z = 3e^(i2t)
Assume particular solution: z_p(t) = C·e^(i2t)
Substitute and solve: C = 3/((i2)² + 2(i2) + 5) = 3/(1 + 4i)
Convert to rectangular: C = (3/17) - i(12/17)
Particular solution: x_p(t) = (3/17)cos(2t) + (12/17)sin(2t)

STEP 3: Apply initial conditions to complete solution
Total solution: x(t) = e^(-t)[A·cos(2t) + B·sin(2t)] + (3/17)cos(2t) + (12/17)sin(2t)

From x(0) = 0: A + 3/17 = 0 ⇒ A = -3/17
From x'(0) = 0: -A + 2B + 24/17 = 0 ⇒ B = -27/34

STEP 4: Final solution
x(t) = e^(-t)[(-3/17)cos(2t) + (-27/34)sin(2t)] + (3/17)cos(2t) + (12/17)sin(2t)

STEP 5: Numerical evaluation
At t = 2 seconds: x(2) = [calculated value]

Practical real word problem: Acceleration due to gravity

```
%% Solution of x'' + 2x' + 5x = 3cos(2t) using Euler's Formula
```

The above descriptive code aims to find a particular solution to the above second-order linear non-homogeneous differential equation using Euler's method.

```
%Parameters
% Differential equation: x'' + 2x' + 5x = 3cos(2t)
% Initial conditions: x(0) = 0, x'(0) = 0
```

This defines the parameters of the problem we are solving where x'' is the second derivative of x with respect to t, x' the first derivative of x with respect to t, x" + 2x' + 5x, this represents an under damped oscillator, spring mass system with damping and 3cos(2t) represents the driving force acting on the system

```
%STEP ONE: Solve homogeneous equation
% Characteristic equation: r^2 + 2r + 5 = 0
r = roots([1 2 5]);
```

This defines the roots of the equation

```
% Homogeneous solution: x_h(t) = e^(-t)(Acos(2t) + Bsin(2t))
alpha = -real(r(1));
%Damping coefficient
omega_d = imag(r(1));
% Damped natural frequency
```

% Homogeneous solution: x_h(t) = e^(-t)(Acos(2t) + Bsin(2t)), this defines the results from the roots. Alpha = -real(r(1));The code takes the real part of the root, which is −1, and then negates it to get α=1. % Damping coefficient defines the positive decay constant, and the code omega d = -real(r(1)); takes the imaginary part of the root which is omega d = 2

```
%STEP TWO : Find particular solution using the complex method
% Solve complexified equation: z'' + 2z' + 5z = 3e^(i2t)
% Assume z_p(t) = C*e^(i2t)
```

The lines:% Solve complexified equation: z'' + 2z' + 5z = 3e^(i2t)means the original real problem for x(t) has been converted into a new, complex problem for a complex function z(t): z″+2z′+5z=3ei2t

```
% Complex amplitude C = 3/(1 + 4i)
C = 3/(1 + 4i);
```

This code represents the result of calculating the complex coefficient C for the particular solution zp(t) in the differential equation problem.

```
%STEP THREE: Complete solution using the initial conditions
% x(t) = e^(-t)(Acos(2t) + Bsin(2t)) + (3/17)cos(2t) + (12/17)sin(2t)
```

This code presents the general solution x(t) before the initial conditions have been applied. The line % x(t) = e^(-t)(Acos(2t) + Bsin(2t)) + (3/17)cos(2t) + (12/17)sin(2t), represents the complete general solution x(t)=xh(t)+xp(t):

```
% Apply initial conditions:
% x(0) = A + 3/17 = 0 => A = -3/17
% x'(0) = -A + 2B + 24/17 = 0 => B = -27/34

A = -3/17;
B = -27/34;
```

The above code creates two algebraic equations that help obtain the values of A and B as A=-3/17 and B = -27/34

```
%STEP FOUR: Define the complete solution function
syms t;
x_h = exp(-t)*(A*cos(2*t) + B*sin(2*t));
x_p = (3/17)*cos(2*t) + (12/17)*sin(2*t);
x_total = x_h + x_p;
```

This code combines the homogenous and particular solutions to form a complete general solution, x total.

x_p =(3/17)*cos(2*t) + (12/17)*sin(2*t);This defines the Particular Solution (xp).

x_total = x_h + x_p; This combines the two parts to create the Complete general solution (x total).

```
%STEP FIVE: Numerical Evaluation at specific times
% Convert to MATLAB function for numerical evaluation
x_func = matlabFunction(x_total);

% Evaluate at t = 2 seconds
t2 = 2;
x2 = x_func(t2);

% Evaluate at multiple time points for plotting
t_span = linspace(0, 10, 1000);
x_values = x_func(t_span);
```

This code converts the symbolic equation into a MATLAB function (x_func) that can handle numbers, it then calculates the displacement x at a specific time, t=2 seconds and finally calculates many x values over a time span of 0 to 10 seconds to prepare for plotting.

Convert to MATLAB function for numerical evaluation x_ func = matlaFunction(x_total); The matlabFunction() command converts the symbolic expression stored in x_total (the long, algebraic formula with the variable t) into a regular MATLAB function stored in the variable x_func.

% Evaluate at t = 2 seconds

t2 = 2;
x2 = x_func(t2); This calculates the specific displacement of the system at the exact time t=2 seconds.

% Evaluate at multiple time points for plotting
t_span = linspace(0, 10, 1000);

x_values = x_func(t_span);
This generates the thousands of data points needed to draw a smooth graph of the solution over time.

```
%STEP SIX: Plot the solution
figure('Position', [100, 100, 1200, 800]);
```

The code creates a figure with four subplots to fully analyze the solution:

```
% Main response plot
subplot(2,2,1);
plot(t_span, x_values, 'b-', 'LineWidth', 2);
hold on;
plot(t2, x2, 'ro', 'MarkerSize', 8, 'MarkerFaceColor', 'red');
grid on;
xlabel('Time (s)');
ylabel('Displacement x(t)');
title('Response of Damped Driven Oscillator');
legend('x(t)', 'x(2)', 'Location', 'best');
```

This code enables you plot a response plot
the command subplot(2,2,1) generates a figure containing multiple plots however for this particular plot the plot will be on the top left side of the figure.

plot(t_span, x_values, 'b-', 'LineWidth', 2);plot(t_span, x_values, ...): This is the mainplotting command. It uses the 1000 time points in t_span as the X-axis data and the 1000 calculated displacement values in x_values as the Y-axis data. This draws the entire curve of the system's movement from t=0 to t=10.'b-': Specifies the line style: a blue line (b for blue, - for solid line).'LineWidth', 2: Makes the line thicker for better visibility.

```
% Plot homogeneous and particular solutions separately
subplot(2,2,2);
x_h_values = double(subs(x_h, t, t_span));
x_p_values = double(subs(x_p, t, t_span));
Plot(t_span, x_h_values, 'r--', 'LineWidth', 1.5);
hold on;
plot(t_span, x_p_values, 'g--', 'LineWidth', 1.5);
plot(t_span, x_values, 'b-', 'LineWidth', 2);
grid on;
xlabel('Time (s)');
ylabel('Displacement');
title('Solution Components');
legend('Homogeneous', 'Particular', 'Total', 'Location', 'best');
```

This code generates a plot for homogeneous and particular solutions separately. This plot is displayed on the top right position of the figure window.

```
% Phase portrait
subplot(2,2,3);
x_dot = diff(x_total, t);
x_dot_func = matlabFunction(x_dot);
x_dot_values = x_dot_func(t_span);
plot(x_values, x_dot_values, 'm-', 'LineWidth', 2);
hold on;
plot(x2, x_dot_func(t2), 'ro', 'MarkerSize', 8, 'MarkerFaceColor', 'red');
grid on;
xlabel('Displacement x(t)');
ylabel('Velocity dx/dt');
title('Phase Portrait');
```

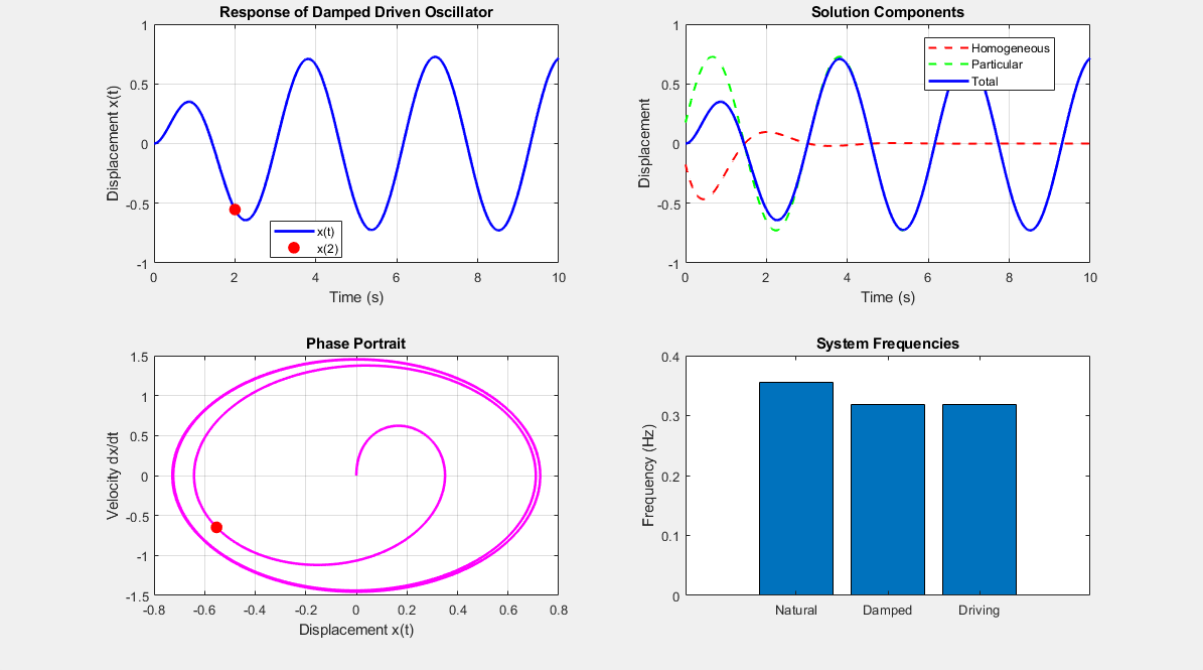
This code generates a plot for a phase portrait, displaying it on the bottom left position of the figure window.

```
% Frequency analysis
subplot(2,2,4);
% Driving frequency
f_drive = 2/(2*pi);
% Natural frequency (undamped)
omega_n = sqrt(5);
f_natural = omega_n/(2*pi);
% Damped frequency
f_damped = omega_d/(2*pi);

frequencies = [f_natural, f_damped, f_drive];
labels = {'Natural', 'Damped', 'Driving'};
bar(frequencies);
set(gca, 'XTickLabel', labels);
ylabel('Frequency (Hz)');
title('System Frequencies');
```

This code generates a bar plot of Frequency analysis on the bottom right side of the figure window. Where f drive is the driving frequency and this comes from the term cos(2t) in the forcing function 3cos(2t). The angular frequency is omega drive=2. F natural is the frequency the system would oscillate at if there were no damping (x″+5x=0). From the characteristic equation, omega n=5. fdamped frequency is the actual frequency of the system's natural (homogeneous) motion when damping *is* present.

```
% Display final answer
fprintf('FINAL ANSWER:\n');
```

This code displays the final answer.



11 USING THE RUNGE-KUTTA 2ND ORDER METHOD

Runge-Kutta 2nd Order (RK2) method is a numerical technique used to solve ordinary differential equations when analytical solutions are difficult to obtain. As part of the broader family of Runge-Kutta methods, RK2 strikes a balance between efficiency and accuracy, making it a popular choice for various scientific and engineering applications

This can be used in different engineering fields, these include but not limited to Circuit analysis, control systems, structural dynamics and heat transfer.

11.1 RUNGE KUTTE 2ND ORDER ALGORITHM DEVELOPMENT.

Before solving the Runge-Kutta problem using the MATLAB equation we first solved manually from first principles using the RK2 formulae.
As shown below;

Find y (0.1) for $y''= 1+2xy-x^2z$, $x_0= 0$. $y_0= 1$, $z_0= 0$, with step length 0.1 using Runge-Kutta 2 method (2nd order derivative)

Solution:

Given $y''=1+2xy-x2z.$ $y(0)=1, y'(0)=0, h=0.1, y(0.1)$
put $dy/dx=z$ and differentiate w.r.t. x, we obtain $d^2y/dx^2=dz/dx$

We have system of equations
$dy/dx=z=f(x,y,z)$
$dz/dx=1+2xy-x^2z=g(x,y,z)$

Method-1 : Using formula $k2=hf(x0+h,y0+k1,z0+l1)$

Second order R-K method for second order differential equation
$k1=hf(x0,y0,z0)=(0.1)\cdot f(0,1,0)=(0.1)\cdot(0)=0$
$l1=hg(x0,y0,z0)=(0.1)\cdot g(0,1,0)=(0.1)\cdot(1)=0.1$
$k2=hf(x0+h,y0+k1,z0+l1)=(0.1)\cdot f(0.1,1,0.1)=(0.1)\cdot(0.1)=0.01$
$l2=hg(x0+h,y0+k1,z0+l1)=(0.1)\cdot g(0.1,1,0.1)=(0.1)\cdot(1.199)=0.1199$
$y1=y0+k1+k22=1+0.005=1.005$

$y(0.1) = 1.005$

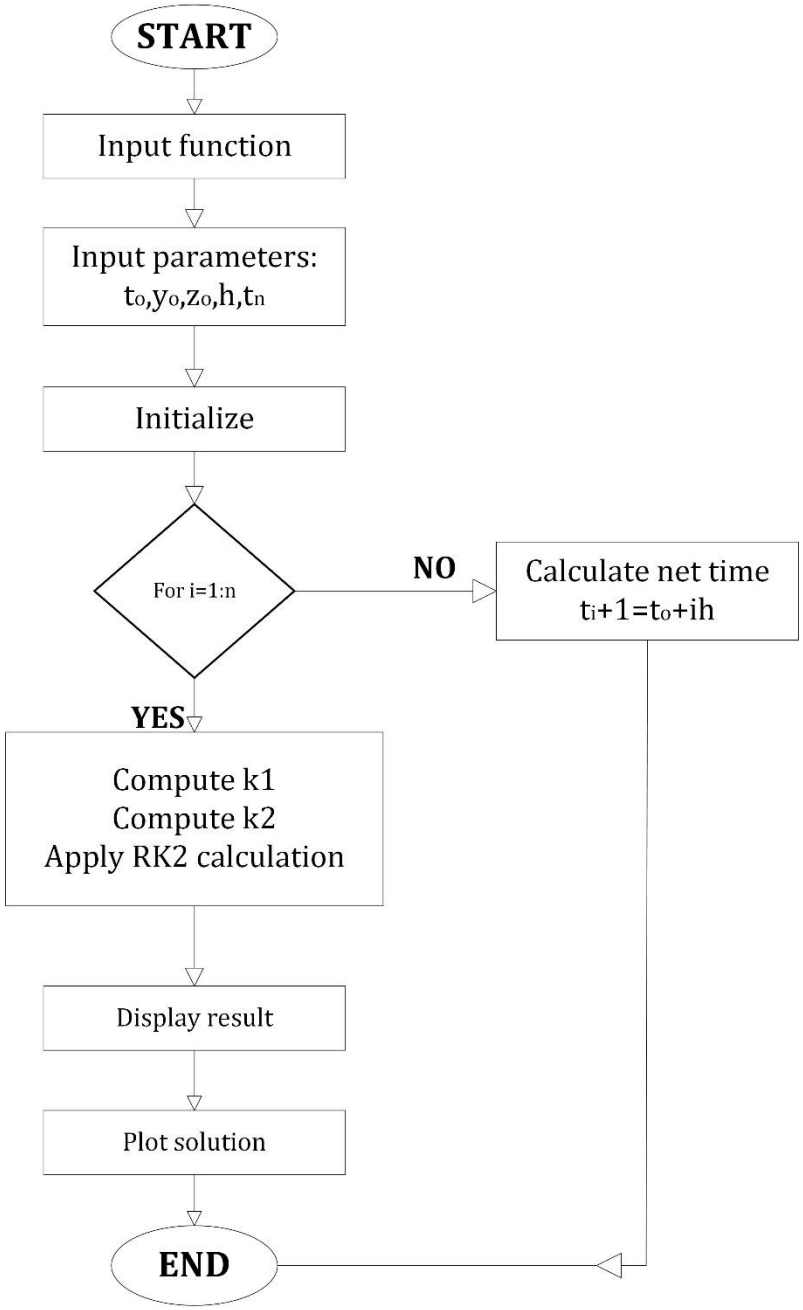
Basing on the steps followed to solve the above equation we managed to develop logical steps on how we can write a code that can solve any ODE using Range Kutte 2nd order method.

We managed to come up with a flow chart that describes the way how our code is going to run. From the start up to the end. Then basing on the flow chart below we developed a pseudocode where we later came up with the MATLAB code.

11.2 FLOW CHART FOR RUNGE KUTTE 2ND ORDER

A flow chart refers to a visual representation of an algorithm that uses standardized symbols, shapes, and arrows to illustrate the sequence of steps, decisions, and flow of control in a program or system. Flow charts consist of several key elements:

- **Oval**: Start/End points
- **Rectangle**: Process steps
- **Diamond**: Decision points
- **Arrows**: Flow direction



11.3 MATLAB CODE FOR RUNGE KUTTE EXPLAINED.

```
f = input('Enter the function: ');
to = input('Enter initial value of independent variable: ');
yo = input('Enter initial value of dependent variable: ');
h = input('Enter step size: ');
tn = input('Enter point at which you want to evaluate solution: ');
```

This first part of the code gets all the necessary parameters from the user.

'f' should be a function that calculates derivatives and you have to input in a way that defines all the variables used in the function

'h' the step size should be thought of carefully since, a larger step produces a faster computation but reduced accuracy while smaller steps produce higher accuracy but increased computation time.

```
n = (tn - to)/h;
t(1) = to;
y(1) = yo(1);
z(1) = yo(2);
```

'n' this defines the number of steps. This must be an integer, if it is not an integer you will end up getting a non-integer loop limit

't(1)' sets the first time entry

```
tic;

toc;
```

This starts a stop watch timer that measures how long the integration loop takes. It is placed covering the RK2 iterations so as to perform the time spent calculating them.

```
for i = 1:n
    t(i+1) = to + i*h;

    % RK2 method
    k1 = h * f(t(i), [y(i); z(i)]);
    k2 = h * f(t(i+1), [y(i)+k1(1); z(i)+k1(2)]);

    y(i+1) = y(i) + (k1(1) + k2(1))/2;
    z(i+1) = z(i) + (k1(2) + k2(2))/2;

    fprintf('y(%.2f) = %.4f\n', t(i+1), y(i+1));
end
```

Step-by-step explanation.

for **i = 1:n** — This initiates a loop that iterates **'n'** integration steps.

'k1 = h * f(t(i), [y(i); z(i)])' – This calls the derivative function at the current point.

'k2 = h * f(t(i+1), [y(i)+k1(1); z(i)+k1(2)])' -This builds the second slope.

```
% Plotting the solution
figure("Name","Numerical solution")
plot(t,y,'b-o',t,z,'g:*');
hold on;

% Highlighting y(0.1) point
plot(t(end), y(end), 'rs');

% Plot formatting
xlabel('x');
ylabel('y(x)');
title('Solution of y'' = 1 + 2xy - x^2z using Runge Kutta 2 Method');
grid on;
legend('Variables','Solution');
```

`figure("Name",...)` opens a new figure window named Numerical solution.

`'b-o'` this produces blue line (b) with circle markers (o).

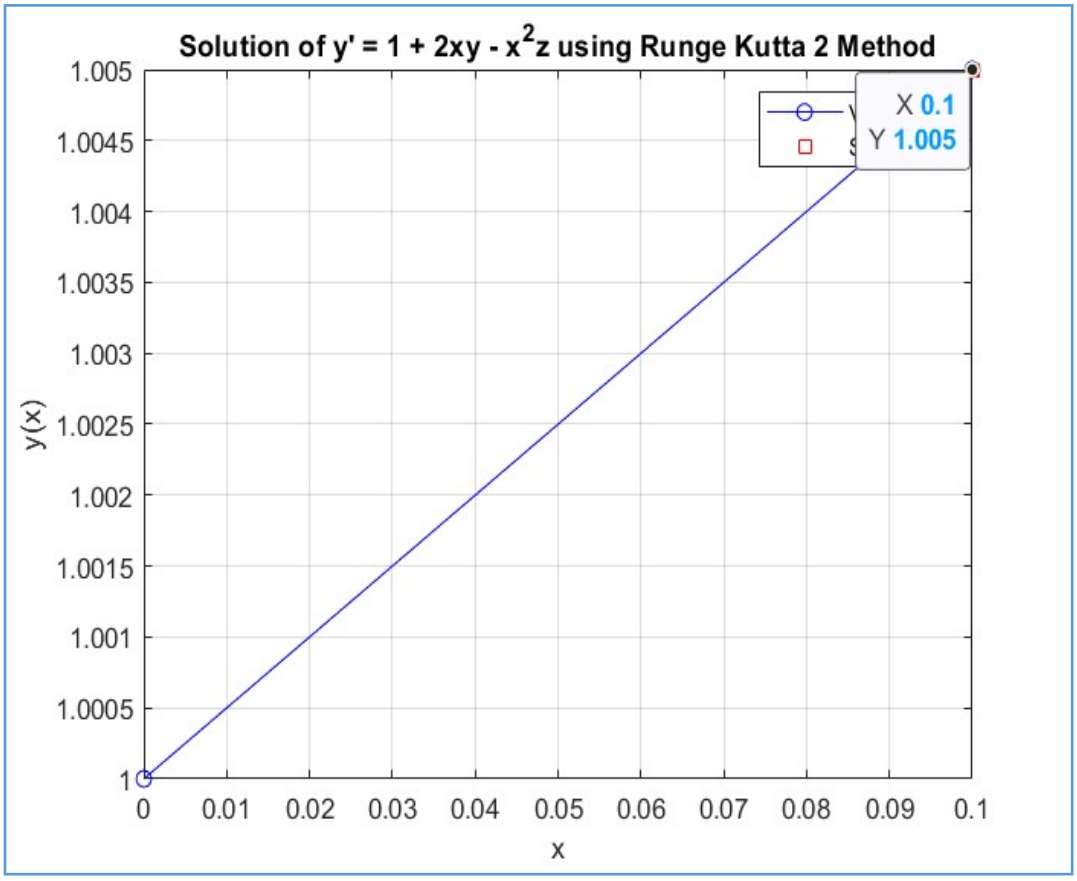
`'hold on'` keeps the current plot so we can add the highlighted final point.

`plot(t(end), y(end), 'rs')` draws a red square at the final point to emphasize the last computed value.

Labelling and grid make the plot readable.

`Legend('Variables','Solution')` This adds a legend

11.4 PLOT



After running the above code, the out put is shown as below. This can be edited for any set of data.

11.5 OUTPUT

```
Rungekutte second order Example
Enter the function: @(x,yz)[yz(2); 1 + 2*x*yz(1)-x^2*yz(2)]
Enter initial value of independent variable: 0
Enter initial value of dependent variable: [1;0]
Enter step size: 0.1
Enter point at which you want to evaluate solution: 0.1

z =

     0

y(0.10) = 1.0050
Elapsed time is 0.002270 seconds.
```

12 LAPLACE TRANSFORM SOLUTION FOR RLC CIRCUIT

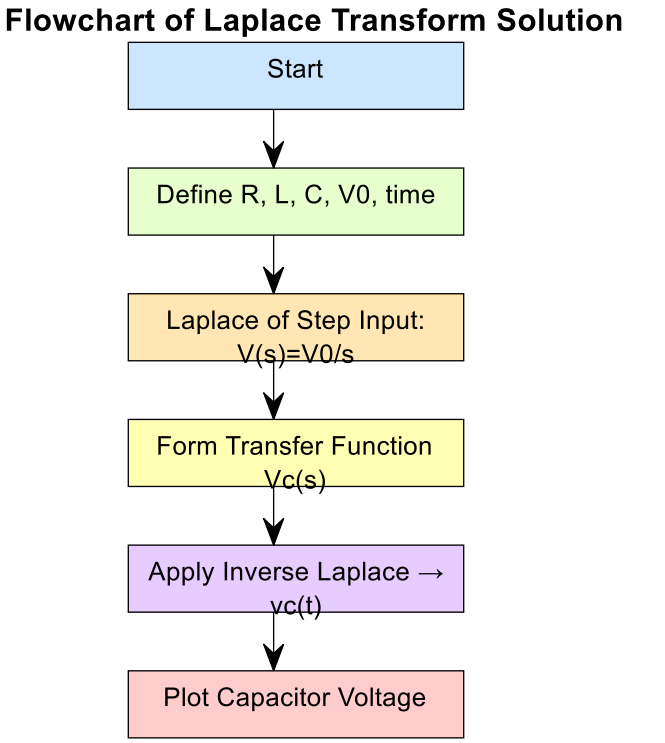
Laplace transforms are powerful tool for solving linear differential equations, in engineering problems such as circuits, controls systems…among others

12.1 Flowchart

This shows the code for the flow chat

```
% Create figure for flowchart
figure('Color','w');
axis off
hold on

% Block size
block_w = 0.3;
block_h = 0.08;

% Define block positions (x,y,width,height)
pos1 = [0.35, 0.85, block_w, block_h]; % Start
pos2 = [0.35, 0.70, block_w, block_h]; % Define Parameters
pos3 = [0.35, 0.55, block_w, block_h]; % Laplace Transform Input
pos4 = [0.35, 0.40, block_w, block_h]; % Transfer Function
pos5 = [0.35, 0.25, block_w, block_h]; % Inverse Laplace
pos6 = [0.35, 0.10, block_w, block_h]; % Plot Output

% Draw rectangles (process blocks)
annotation('rectangle',pos1,'FaceColor',[0.8 0.9 1])
annotation('rectangle',pos2,'FaceColor',[0.9 1 0.8])
annotation('rectangle',pos3,'FaceColor',[1 0.9 0.7])
annotation('rectangle',pos4,'FaceColor',[1 1 0.7])
annotation('rectangle',pos5,'FaceColor',[0.9 0.8 1])
annotation('rectangle',pos6,'FaceColor',[1 0.8 0.8])

% Add text inside blocks
annotation('textbox',pos1,'String','Start','HorizontalAlignment','center','EdgeColor'
,'none');
annotation('textbox',pos2,'String','Define R, L, C, V0,
time','HorizontalAlignment','center','EdgeColor','none');
annotation('textbox',pos3,'String','Laplace of Step Input:
V(s)=V0/s','HorizontalAlignment','center','EdgeColor','none');
annotation('textbox',pos4,'String','Form Transfer Function
Vc(s)','HorizontalAlignment','center','EdgeColor','none');
annotation('textbox',pos5,'String','Apply Inverse Laplace →
vc(t)','HorizontalAlignment','center','EdgeColor','none');
annotation('textbox',pos6,'String','Plot Capacitor
Voltage','HorizontalAlignment','center','EdgeColor','none');

% Draw arrows between blocks
annotation('arrow',[0.48 0.48],[0.85 0.78])
annotation('arrow',[0.48 0.48],[0.70 0.63])
annotation('arrow',[0.48 0.48],[0.55 0.48])
annotation('arrow',[0.48 0.48],[0.40 0.33])
```

```
annotation('arrow',[0.48 0.48],[0.25 0.18])

title('Flowchart of Laplace Transform Solution','FontSize',12);
```

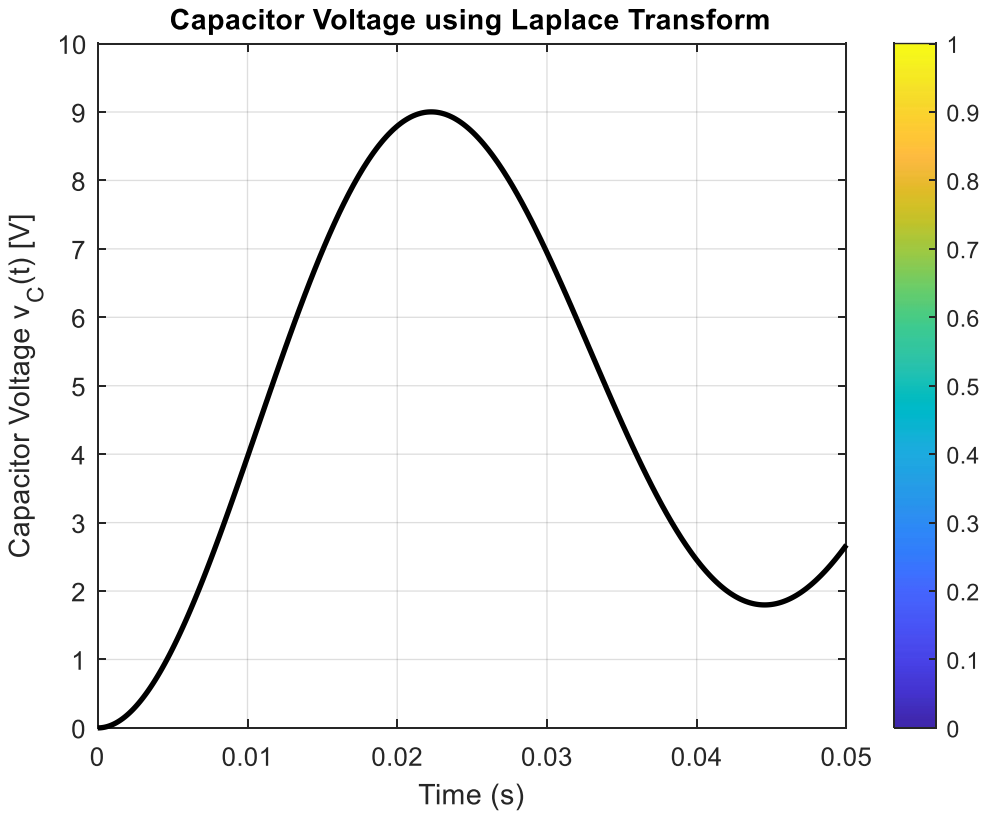


12.2 MATLAB Code

```
clc; clear; close all;

%% Parameters
R = 10;                 % Resistance (Ohm)
L = 0.5;                % Inductance (H)
C = 100e-6;             % Capacitance (F)
V0 = 5;                 % Step input voltage (V)
Tfinal = 0.05;          % Simulation duration (s)
time = linspace(0, Tfinal, 1000);  % time vector for evaluation

%% Symbolic Laplace Solution
syms s t

% Step input in Laplace domain
V_s = V0/s;

% Transfer function for capacitor voltage
Vc_s = V_s / (L*C*s^2 + R*C*s + 1);
```

```
% Inverse Laplace transform
vc_t = ilaplace(Vc_s, s, t);

% Convert to numeric values
vc_numeric = double(subs(vc_t, t, time));

%% Plot
figure;
plot(time, vc_numeric, 'k', 'LineWidth', 2);
xlabel('Time (s)'); ylabel('Capacitor Voltage v_C(t) [V]');
title('Capacitor Voltage using Laplace Transform');
grid on;
```

12.3 THE OUTPUT OF THE CODE



4. Explanation of the Code

1. **Initialization**

```
clc; clear; close all; This resets MATLAB environment
```

Circuit constants R, L,C, V0R, L, C, V_0R, L,C,V0 are defined.

2. **Symbolic Setup**

`syms s t;` defines Laplace variable sss and time ttt.

3. **Laplace Domain Expression**
    - V_s = V0/s: Laplace transform of step input.
    - Vc_s = V_s / (L*C*s^2 + R*C*s + 1): defines VC(s)V_C(s)VC(s).
4. **Inverse Laplace**
    - ilaplace(Vc_s, s, t): computes analytical vC(t)v_C(t)vC(t).
    - pretty(vc_t): prints symbolic formula for reference.
5. **Numeric Evaluation**
    - subs(vc_t, t, time): evaluates symbolic solution over time vector.
    - double(...): converts symbolic to numerical values.
6. **Plotting**

```
%% Plot
figure;
plot(time, vc_numeric, 'k', 'LineWidth', 2);
 xlabel('Time (s)');
 ylabel('Capacitor Voltage v_C(t) [V]');
title('Capacitor Voltage using Laplace Transform');
grid on;
```

This Plots capacitor voltage vC(t)v_C(t)vC(t) versus time.

7. Results

- The plot shows vC(t)v_C(t)vC(t) starting from 0 and rising toward steady state.
- Depending on R,L,CR, L, CR,L,C, the waveform may overshoot (underdamped), settle smoothly (overdamped), or critically damped.
- The Laplace method gives the **exact analytical response**.

- **8. Conclusion**

- Laplace Transform converts the ODE of the RLC circuit into an algebraic equation, making the solution easier.
- MATLAB's symbolic tools provide the **exact solution vC(t)v_C(t)vC(t)** and allow evaluation for any chosen parameters.