International Islamic University Chittagong (IIUC)

Department of Computer Science Engineering (CSE)

PROJECT REPORT

Course title : **Numerical Method Lab**

Course code : **CSE-4746**

Session : **Spring-2024**

Submitted To:

Mohammed Shamsul Alam

Professor, Dept. of CSE

International Islamic University Chittagong

Submitted By:

Name : Afif Hossain Irfan

Matric ID : C211005

Semester : 7th

Section : 7AM

Mobile no. : 01521536082

Marks :

TITLE: ODE Visualization Using MATLAB

Introduction :

Ordinary Differential Equations (ODEs) are pivotal in modeling various engineering systems and processes, encompassing fields such as mechanics, electrical circuits, and chemical reactions. This project, titled "ODE Visualization Using MATLAB" aims to develop a MATLAB-based tool for numerically solving and visualizing the behavior of solutions to ODEs. By implementing multiple numerical methods and providing graphical representations of the results, this tool will assist engineers and researchers in understanding and analyzing ODEs more effectively.

Features :

1. Functionality: Implement core functionalities to solve and visualize ODEs using MATLAB.
2. Method Support: Support multiple numerical methods for solving ODEs.
3. Visualization: Provide clear and informative plots of the solutions.
4. User Control: Allow users to adjust parameters such as initial values, step sizes, and the point of evaluation.
5. Comparison and Analysis: Offer features to compare the performance and accuracy of different methods.

Methodology :

1. Input Handling: Prompt users to enter the ODE function, initial conditions, step size, and endpoint using MATLAB's input functions.

2. Numerical Methods Implementation:

i) Implement the following methods in MATLAB:

- **Euler's Method:** A straightforward numerical approach using a simple iterative process.
- **Heun's Method:** An improved version of Euler's method with better accuracy.
- **Midpoint Method:** Uses the midpoint to achieve a more accurate result than Euler's method.
- **Runge-Kutta Method:** A higher-order method providing excellent accuracy by averaging several intermediate steps.

ii)Visualization:

- Use MATLAB's plotting functions to create graphs that depict the solutions obtained from each method.
- Ensure each method's results are plotted with distinct markers and colors for clear differentiation.

iii) Comparison:

- Plot all methods on the same graph to allow visual comparison of the different solution trajectories.

iv) Documentation:

- Provide detailed comments in the code to explain each step and method.
- Include usage instructions and theoretical background in the project documentation.

Implementation :

The implementation of the ODE Visualization Using MATLAB involves several steps, from handling user input to plotting the results of various numerical methods. Below is a detailed breakdown of each step, including flowcharts and a short description of the functions used.

1. Flowchart of the Process:

Step 1: Initialize the MATLAB Environment –
Clear any existing variables and the command window to ensure a clean start.

Step 2: Handle User Input -
Prompt the user to input the ODE function, initial values, step size, and the evaluation point.

Step 3: Calculate the Number of Steps -
Determine the number of steps required to reach the evaluation point using the given step size.

Step 4: Implement, Execute and Plotting Euler Method -
Define the Euler method function and call it to solve the ODE. Plot the results obtained from the Euler method.

Step 5: Implement, Execute and Plotting Heun's Method -
Define the Heun's method function and call it to solve the ODE. Plot the results obtained from the Heun's method.

Step 6: Implement, Execute and Plotting Midpoint Method -
Define the Midpoint method function and call it to solve the ODE. Plot the results obtained from the Midpoint method.

Step 7: Implement, Execute and Plotting Runge-Kutt Method -
Define the Runge-Kutt method function and call it to solve the ODE. Plot the results obtained from the Runge-Kutt method.

Step 12: Compare All Methods -
Plot all methods on the same graph to compare their solutions.

2. Function Descriptions:

- **Euler Method**: A basic iterative approach that uses the slope at the current point to estimate the next value.

$$y_{i+1} = y_i + h\, f(x_i, y_i)$$

Steps:

1. Start with the initial conditions (x_0, y_0)
2. Compute the slope at the initial point using the function $f(x_i, y_i)$.
3. Move a step h along the xxx-axis to get the new point (x_1, y_1)
4. Repeat the process for the desired number of steps.

Pros:

- Simple and easy to implement.

Cons:

- Low accuracy, especially for large step sizes.

- **Heun's Method**: An improved version of Euler's method that uses the average of slopes at the current point and the predicted next point for better accuracy.

$$k_1 = hf(x_i, y_i)$$

$$k_2 = hf(x_i + h, y_i + k_1)$$

$$y_{i+1} = y_i + \frac{1}{2}(k_1 + k_2)$$

Steps:

1. Start with the initial conditions (x_0, y_0)
2. Compute the slope at the initial point $k_1 = hf(x_i, y_i)$
3. Estimate the slope at the next point using $k_2 = hf(x_i + h, y_i + k_1)$
4. Compute the average slope and use it to update the solution.
5. Repeat for the desired number of steps.

Pros:

- Higher accuracy than Euler's Method.
- Still relatively simple to implement.

Cons:

- Can still accumulate errors over large intervals, although less so than Euler's Method.

- **Midpoint Method**: A method that uses the slope at the midpoint of the interval to estimate the next value, offering improved accuracy over Euler's method.

$$k_1 = hf(x_i, y_i)$$

$$k_2 = hf\left(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}\right)$$

$$y_{i+1} = y_i + k_2$$

Steps:

1. Start with the initial conditions (x_0, y_0)
2. Compute the slope at the initial point $k_1 = hf(x_i, y_i)$
3. Estimate the slope at the midpoint using $k_2 = hf(x_i + h/2\,, y_i + k_1/2)$
4. Use this midpoint slope to update the solution.
5. Repeat for the desired number of steps.

Pros:

- Better accuracy compared to both Euler's and Heun's methods for the same step size.
- Still relatively straightforward to implement.

Cons:

- Requires more function evaluations per step compared to Euler's Method.

- **Runge-Kutta Method**: A higher-order method that computes multiple intermediate slopes and combines them to provide a highly accurate solution.

$$k_1 = hf(x_i, y_i)$$

$$k_2 = hf\left(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}\right)$$

$$k_3 = hf\left(x_i + \frac{h}{2}, y_i + \frac{k_2}{2}\right)$$

$$k_4 = hf(x_i + h\,, y_i + k_3)$$

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4\,)$$

Steps:

1. Start with the initial conditions (x_0, y_0)
2. Compute intermediate slopes k_1 , k_2, k_3 and k_4.
3. Combine these slopes in a weighted average to update the solution.
4. Repeat the process for the desired number of steps.

Pros:

- Very high accuracy compared to the previous methods for the same step size.
- Generally used when accuracy is crucial or when large step sizes are required.

Cons:

- More complex and computationally intensive compared to Euler's and Heun's methods.

Conclusion

- **Euler's Method** is the simplest but least accurate.
- **Heun's Method** improves upon Euler's accuracy by using a better slope estimation.
- **Midpoint Method** further improves accuracy by evaluating the slope at the midpoint of the interval.
- **Runge-Kutta Method** provides the highest accuracy among the discussed methods but is more computationally intensive.

Used Dataset :

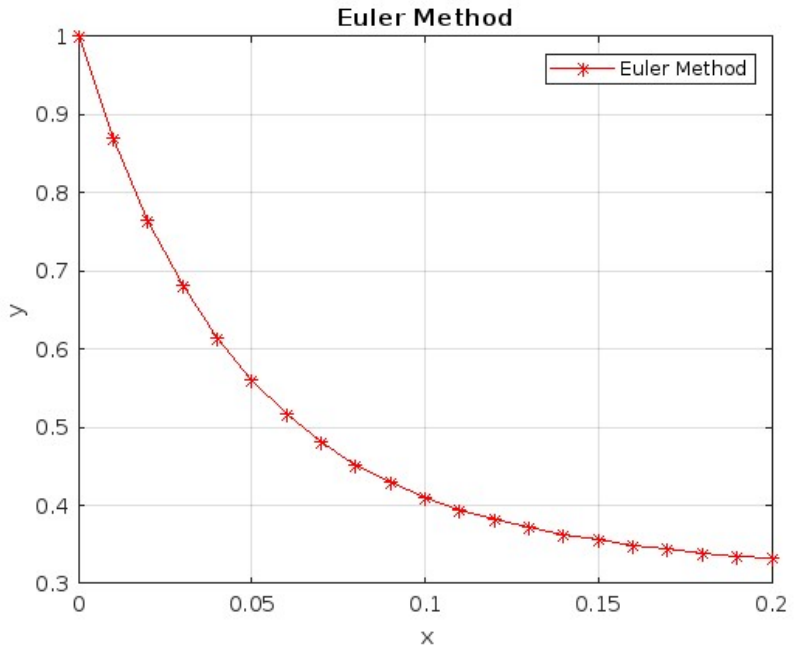
The Equation $\frac{dy}{dx} + 20y = 7e^{-0.5x}$, the initial condition $y(0) = 1$

Now, Computing $y(0.2)$ using 4 methods by taking $h = 0.01$

Result :

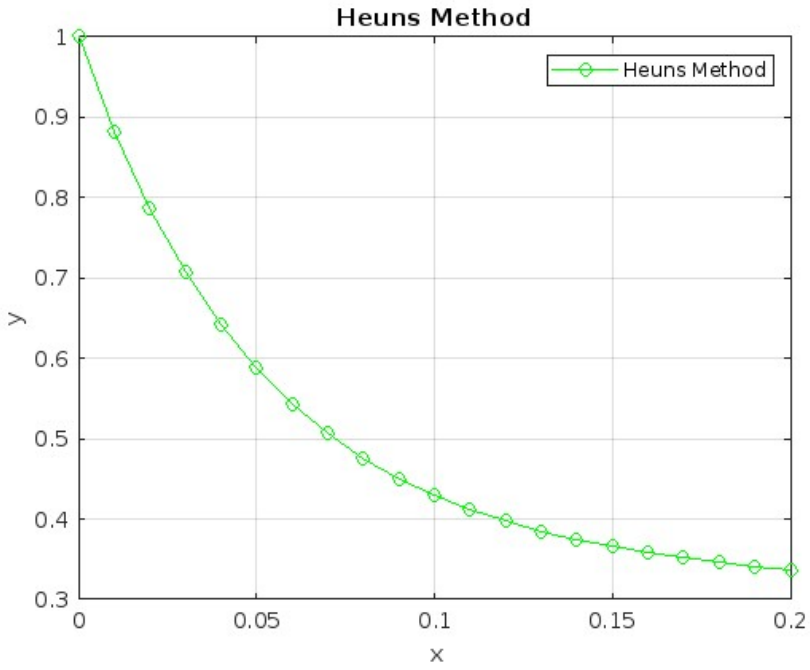
- **Euler Method**:

```
y(  0.0100) =   0.870000
y(  0.0200) =   0.765651
y(  0.0300) =   0.681824
y(  0.0400) =   0.614417
y(  0.0500) =   0.560148
y(  0.0600) =   0.516390
y(  0.0700) =   0.481043
y(  0.0800) =   0.452427
y(  0.0900) =   0.429197
y(  0.1000) =   0.410277
y(  0.1100) =   0.394808
y(  0.1200) =   0.382100
y(  0.1300) =   0.371604
y(  0.1400) =   0.362878
y(  0.1500) =   0.355570
y(  0.1600) =   0.349398
y(  0.1700) =   0.344136
y(  0.1800) =   0.339605
y(  0.1900) =   0.335659
y(  0.2000) =   0.332183
```



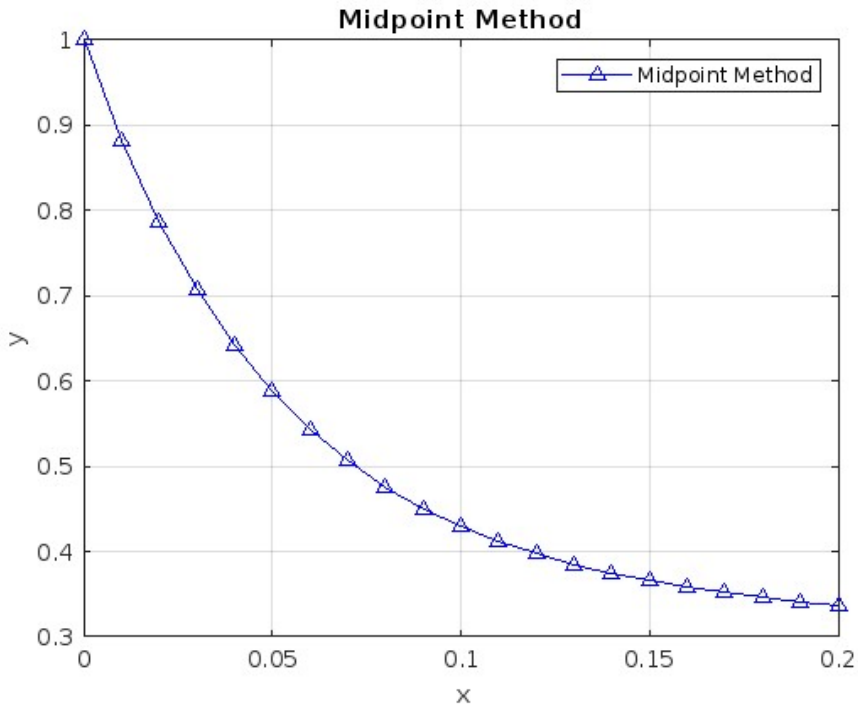
- **Heun's Method**:

```
y(  0.0100) =   0.882825
y(  0.0200) =   0.786429
y(  0.0300) =   0.707072
y(  0.0400) =   0.641689
y(  0.0500) =   0.587767
y(  0.0600) =   0.543243
y(  0.0700) =   0.506428
y(  0.0800) =   0.475935
y(  0.0900) =   0.450629
y(  0.1000) =   0.429577
y(  0.1100) =   0.412014
y(  0.1200) =   0.397315
y(  0.1300) =   0.384965
y(  0.1400) =   0.374543
y(  0.1500) =   0.365703
y(  0.1600) =   0.358163
y(  0.1700) =   0.351689
y(  0.1800) =   0.346091
y(  0.1900) =   0.341212
y(  0.2000) =   0.336926
```



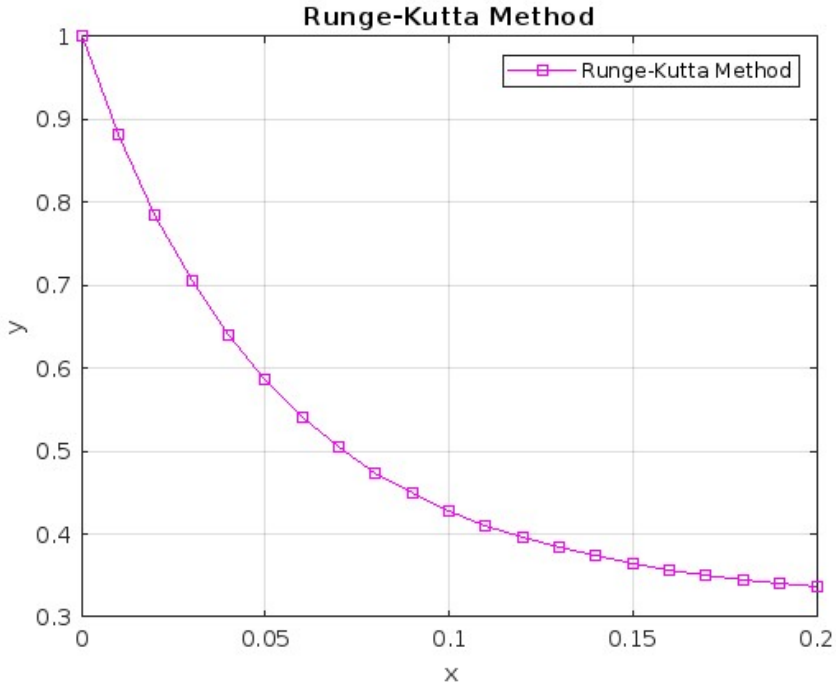
- **Midpoint Method**:

```
y(  0.0100) =   0.882825
y(  0.0200) =   0.786429
y(  0.0300) =   0.707072
y(  0.0400) =   0.641689
y(  0.0500) =   0.587766
y(  0.0600) =   0.543242
y(  0.0700) =   0.506427
y(  0.0800) =   0.475934
y(  0.0900) =   0.450628
y(  0.1000) =   0.429576
y(  0.1100) =   0.412013
y(  0.1200) =   0.397314
y(  0.1300) =   0.384964
y(  0.1400) =   0.374542
y(  0.1500) =   0.365702
y(  0.1600) =   0.358162
y(  0.1700) =   0.351687
y(  0.1800) =   0.346089
y(  0.1900) =   0.341211
y(  0.2000) =   0.336925
```

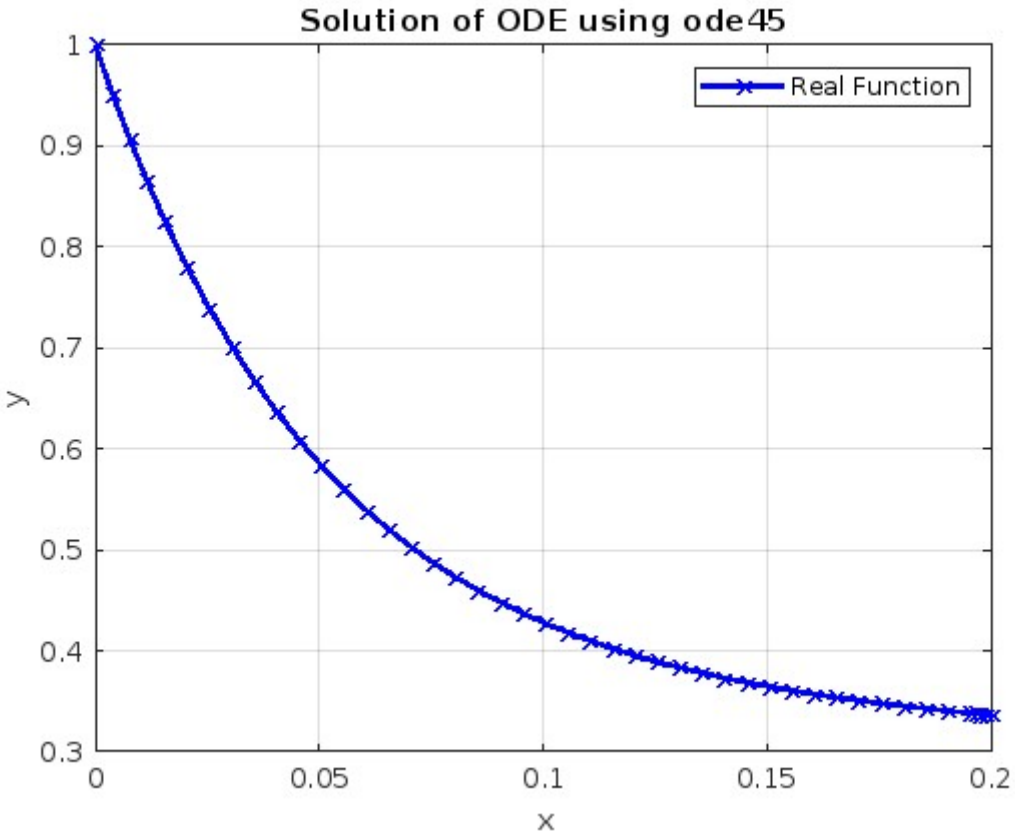


- **Runge-Kutta Method**:

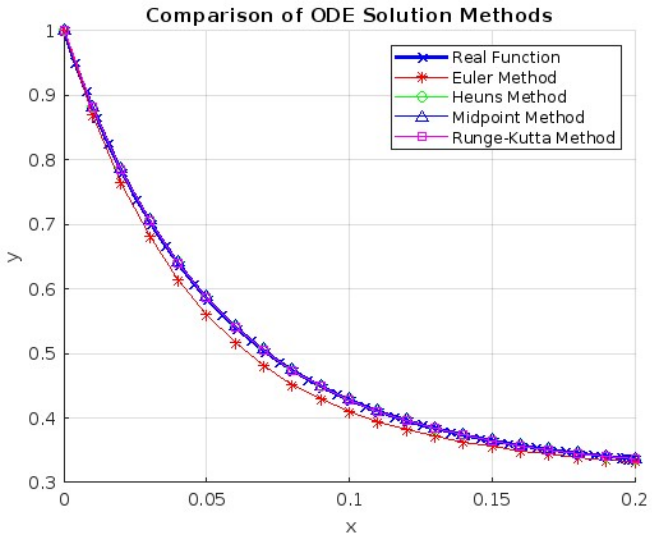
```
y(  0.0100) =   0.882013
y(  0.0200) =   0.785098
y(  0.0300) =   0.705436
y(  0.0400) =   0.639901
y(  0.0500) =   0.585935
y(  0.0600) =   0.541442
y(  0.0700) =   0.504706
y(  0.0800) =   0.474323
y(  0.0900) =   0.449142
y(  0.1000) =   0.428223
y(  0.1100) =   0.410794
y(  0.1200) =   0.396224
y(  0.1300) =   0.383996
y(  0.1400) =   0.373688
y(  0.1500) =   0.364952
y(  0.1600) =   0.357506
y(  0.1700) =   0.351117
y(  0.1800) =   0.345594
y(  0.1900) =   0.340783
y(  0.2000) =   0.336555
```

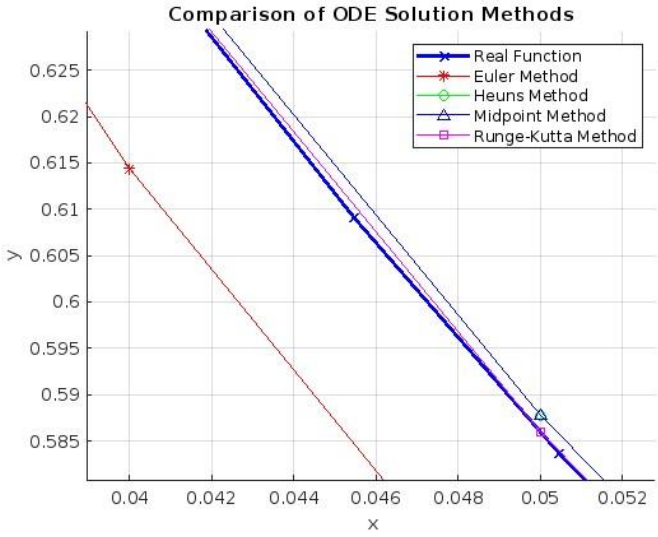


- **Real Solution**:



Analysis :





Here,

Accuracy of Euler Method: 96.99%
Accuracy of Heun's Method: 99.78%
Accuracy of Midpoint Method: 99.78%
Accuracy of Runge-Kutta Method: 99.99%

Conclusion :

The "ODE Visualization Using MATLAB" project aims to provide a comprehensive tool for solving and visualizing ordinary differential equations using various numerical methods. By implementing and comparing methods like Euler's, Heun's, Midpoint, and Runge-Kutta, the project not only aids in solving ODEs but also serves as an educational resource for understanding the nuances of these numerical techniques. This tool will be valuable for both engineers and researchers, enabling them to model, solve, and analyze ODEs effectively.

Source Code :

```
clear all;
clc;

% Define the function handle input
f = input('Enter your function: ');

% Input initial values, step size, and end point
x0 = input('Enter initial value of independent variable (x0): ');
y0 = input('Enter initial value of dependent variable (y0): ');
h = input('Enter step size (h): ');
xn = input('Enter point at which you want to evaluate the solution (xn): ');

% Calculate the number of steps
n = round((xn - x0) / h);

% Euler Method Function
function [x, y] = euler_method(f, x0, y0, h, n)
    x = zeros(1, n+1);
    y = zeros(1, n+1);
    x(1) = x0;
    y(1) = y0;
    for i = 1:n
        y(i + 1) = y(i) + h * f(x(i), y(i));
        x(i + 1) = x0 + i * h;

        fprintf('y(%8.4f) = %10.6f\n',x(i+1),y(i+1))
    end
end

% Euler Method Call
[x_euler, y_euler] = euler_method(f, x0, y0, h, n);
% Plot the results
figure;

plot(x_euler, y_euler, 'r-*', 'DisplayName', 'Euler Method');
grid on;
legend('show');
xlabel('x');
ylabel('y');
title('Euler Method');


% Heun's Method Function
function [x, y] = heuns_method(f, x0, y0, h, n)
    x = zeros(1, n+1);
    y = zeros(1, n+1);
    x(1) = x0;
    y(1) = y0;
    for i = 1:n
        k1 = h * f(x(i), y(i));
        k2 = h * f(x(i) + h, y(i) + k1);

        y(i + 1) = y(i) + (1/2) * (k1 + k2);
        x(i + 1) = x0 + i * h;

        fprintf('y(%8.4f) = %10.6f\n',x(i+1),y(i+1))
    end
end

% Heun's Method Call
[x_heun, y_heun] = heuns_method(f, x0, y0, h, n);

% Plot the results
figure;
```

```
plot(x_heun, y_heun, 'g-o', 'DisplayName', 'Heuns Method');
grid on;
legend('show');
xlabel('x');
ylabel('y');
title('Heuns Method');


% Midpoint Method Function
function [x, y] = midpoint_method(f, x0, y0, h, n)
    x = zeros(1, n+1);
    y = zeros(1, n+1);
    x(1) = x0;
    y(1) = y0;
    for i = 1:n
        k1 = f(x(i), y(i));
        k2 = f(x(i) + (h / 2), y(i) + (h / 2) * k1);

        y(i + 1) = y(i) + h * k2;
        x(i + 1) = x0 + i * h;

        fprintf('y(%8.4f) = %10.6f\n',x(i+1),y(i+1))
    end
end

% Midpoint Method Call
[x_mid, y_mid] = midpoint_method(f, x0, y0, h, n);

% Plot the results
figure;

plot(x_mid, y_mid, 'b-^', 'DisplayName', 'Midpoint Method');
grid on;
legend('show');
xlabel('x');
ylabel('y');
title('Midpoint Method');


% Runge-Kutta Method Function
function [x, y] = rk_method(f, x0, y0, h, n)
    x = zeros(1, n+1);
    y = zeros(1, n+1);
    x(1) = x0;
    y(1) = y0;
    for i = 1:n
        k1 = h * f(x(i), y(i));
        k2 = h * f(x(i) + (h / 2), y(i) + (k1 / 2));
        k3 = h * f(x(i) + (h / 2), y(i) + (k2 / 2));
        k4 = h * f(x(i) + h, y(i) + k3);

        y(i + 1) = y(i) + (1/6) * (k1 + 2 * k2 + 2 * k3 + k4);
        x(i + 1) = x0 + i * h;

        fprintf('y(%8.4f) = %10.6f\n',x(i+1),y(i+1))
    end
end

% Runge-Kutta Method Call
[x_rk, y_rk] = rk_method(f, x0, y0, h, n);

% Plot the results
figure;

plot(x_rk, y_rk, 'm-s', 'DisplayName', 'Runge-Kutta Method');
grid on;
legend('show');
```

```
xlabel('x');
ylabel('y');
title('Runge-Kutta Method');


[x, y] = ode45(f, [x0 xn], y0);

% Plot the solution
figure;
plot(x, y, 'b-x', 'LineWidth', 2);
grid on;
xlabel('x');
ylabel('y');
title('Solution of ODE using ode45');
legend('Real Solution');


figure;
hold on
plot(x, y, 'b-x', 'LineWidth', 2, 'DisplayName', 'Real Solution');
plot(x_euler, y_euler, 'r-*', 'DisplayName', 'Euler Method');
plot(x_heun, y_heun, 'g-o', 'DisplayName', 'Heuns Method');
plot(x_mid, y_mid, 'b-^', 'DisplayName', 'Midpoint Method');
plot(x_rk, y_rk, 'm-s', 'DisplayName', 'Runge-Kutta Method');
grid on;
legend('show');
xlabel('x');
ylabel('y');
title('Comparison of ODE Solution Methods');
hold off;


% Compute relative errors for each method
relative_error_euler = abs((y_euler - interp1(x, y, x_euler)) ./ interp1(x, y,
x_euler)) * 100;
relative_error_heun = abs((y_heun - interp1(x, y, x_heun)) ./ interp1(x, y, x_heun)) *
100;
relative_error_mid = abs((y_mid - interp1(x, y, x_mid)) ./ interp1(x, y, x_mid)) * 100;
relative_error_rk = abs((y_rk - interp1(x, y, x_rk)) ./ interp1(x, y, x_rk)) * 100;

fprintf('Accuracy of Euler Method: %.2f%%\n', 100 - mean(relative_error_euler));
fprintf('Accuracy of Heun''s Method: %.2f%%\n', 100 - mean(relative_error_heun));
fprintf('Accuracy of Midpoint Method: %.2f%%\n', 100 - mean(relative_error_mid));
fprintf('Accuracy of Runge-Kutta Method: %.2f%%\n', 100 - mean(relative_error_rk));
```