Bridging Theory and Computation: MATLAB-Based Numerical Methods for Solving Initial Value Problems in Ordinary Differential Equations

Udoh, Ndipmong A.
Department of Mathematics and Statistics, Faculty of Science
Federal University Otuoke. Bayelsa State, Nigeria.
Email: udohna@fuotuoke.edu.ng

Egbuhuzor, Udechukwu Peter
Department of Mathematics and Statistics, Faculty of Science
Federal University Otuoke. Bayelsa State, Nigeria.
Email: egbuhuzorup@fuotuoke.edu.ng
DOI: 10.56201/ijasmt.vol.11.no2.2025.pg11.25

Abstract

Numerical methods are very significant as it provides approximate solutions to initial value problems (IVPs) in ordinary differential equations (ODEs) where analytical methods fail. This research work considers the implementation of four widely-used numerical methods: Euler's Method, Runge-Kutta Fourth-Order Method, Heun's Method, and Milne's Predictor-Corrector Method, using MATLAB, a powerful tool for technical computing. This work aims to serve as a practical guide for students and researchers, illustrating the seamless integration of theoretical concepts with computational techniques. It provides a structured framework that can be used for any initial value problems (IVPs) in ordinary differential equations (ODEs). Using a single example problem, we demonstrate the step-by-step MATLAB programming of each method, emphasizing computational efficiency, accuracy and error analysis. This research underscores MATLAB's capacity to simplify complex numerical computations and offers recommendations for future enhancements. By bridging theoretical foundations and practical applications, this work contributes to the broader understanding and accessibility of numerical methods in scientific computing.

Keywords: MATLAB; Euler's Method; Runge-Kutta method; Heun's Method; Milne's Method

1. INTRODUCTION

MATLAB is a high-performance, multipurpose language that is widely used in technical computing. It offers an integrated environment for computation, visualization, and programming (MathWorks, 2024). It is a vital tool for data analysis, graphical illustration creation, and mathematical problem solving due to its strong capabilities (Moler, (2020) and Bouchaib & Abdelkhalak (2018)). MATLAB originated from the idea of "MATrix LABoratory" and its main feature is its matrix-based programming language, which makes it easier to represent computational mathematics naturally (Gander & Hřebíček, 2018).

The MATLAB software package offers an efficient platform for examining the design and implementation of numerical algorithms as well as for understanding programming ideas (Higham & Higham, 2017). Its applications cover a wide range of scientific and engineering fields, including signal processing, solving differential equations and optimization problems. MATLAB's relevance in numerical calculations has been highlighted in recent researches, which have used it to teach numerical methods in academic contexts (Ali & Khan, 2021), build hybrid algorithms (Chen & Wang, 2022) and simulate complicated systems (Gupta & Rana, 2023). Additionally, MATLAB's vast libraries and toolboxes make it easier to apply sophisticated approaches, making it an indispensable tool for both practitioners and scholars (Abbas & Zhang, (2023) and Lin & Chou, (2023)). By leveraging its computing capacity and intuitive user interface, MATLAB remains a fundamental tool for theoretical exploration and practical problem-solving in numerical computations (Smith & Brown, (2021), Chapra & Canale, (2010), and Jones et al (2022)).

This research work examines Euler's Method, Runge-Kutta Method, Heun's Method, and Milne's Method, provides a comprehensive comparison of their underlying algorithms, accuracy, and practical applicability. By integrating theoretical insights with MATLAB-based implementations, this research work aims to explore how MATLAB can be used to implement these advanced numerical methods, with a focus on solving real-world problems across a range of applications.

2. ANALYSIS OF METHODS

Here, we discuss the theoretical aspect of the methods under study, develop the algorithms and write the MATLAB programming codes using a single problem for the different methods.

2.1 Euler's Method

Euler's method is one of the simplest and earliest numerical techniques for solving ODEs. It approximates the solution by moving the solution forward in discrete steps, using the tangent (the derivative) at the present position to estimate the next value. The method is first-order accurate that is, the error decreases linearly with the step size. Despite its low computational cost, Euler's method can be unstable and lacks precision for stiff equations or large step sizes. It is still widely used for initial computations since it is straightforward and easy to implement, its limitations not withstanding (Williams & Zhang, 2023).

Given the initial value problem of the first order differential equations of the form:

$$y' = f(x,y), x \in (a,b), y(x_0) = y_0 \tag{1}$$

The Euler's method computes the subsequent value y_{n+1} as follows:

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{2}$$

where h is the selected step size (Kumar, 2023).

2.1.1 Euler's Method Algorithm

1. Start
2. Define the differential equation $f(x,y)$
3. Set the step size h, the interval of computation $[x_0, x_n]$, and the initial condition $y(x_0) = y_0$
4. Compute the number of steps $N = \frac{x_n - x_0}{h}$
5. Initialize arrays x and y:
   x: Create equally spaced points from x_0 to x_n with step size h.

   y: Initialize all values to zero and set $y(0) = y_0$
6. Iterate for $i = 0$ to $N - 1$:
   Calculate the next value of y using Euler's formula:
   $$y(i+1) = y(i) + h.f(x(i), y(i))$$
7. Compute the exact solution $y_{exact}(x)$ and calculate the error at each step:
   $$error(i) = |y_{exact}(i) - \mathrm{y(i)}|$$
8. Display the results in a tabular form
9. Stop

2.2 Runge-Kutta Methods

The Runge-Kutta family of methods is a set of iterative techniques used solving ODEs with higher accuracy than Euler's method. The most popular of them is the fourth-order Runge-Kutta method (RK4). It computes the solution by evaluating the derivative at four distinct points within each step and combines them to estimate the subsequent value. The RK4 method is widely known for its optimal balance between computational effort and accuracy. Given that it provides fourth-order accuracy and is particularly resilient when handling a broad variety of ODEs, it is well-suited for problems with variable stiffness and non-linearity (Otto & Denier, 2005).

For a given IVP, the RK4 method advances one step from (x_n, y_n) to (x_{n+1}, y_{n+1}) using the formula:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \qquad (3)$$

where the terms k_1, k_2, k_3 and k_4 are intermediate slope estimates calculated as follows:

$$k_1 = hf(x_n, y_n), \quad k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right), \quad k_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right), \quad k_4 = hf(x_n + h, y_n + hk_3)$$

2.2.1 Runge-Kutta Fourth-Order Method Algorithm

1. Start
2. Define the differential equation $f(x, y)$
3. Set the step size h, the interval of computation $[x_0, x_n]$, and the initial condition $y(x_0) = y_0$
4. Compute the number of steps $N = \frac{x_n - x_0}{h}$
5. Initialize arrays x and y:
   x: Create equally spaced points from x_0 to x_n with step size h.

   y: Initialize all values to zero and set $y(0) = y_0$
6. Iterate for $i = 0$ to $N - 1$:

   For each $x(i)$, perform the following steps:

a. Compute $k_1$using the differential equation $k_1 = h\,.\,f\big(x(i), y(i)\big) = h\,\big(x(i) + y(i)\big)$
b. Compute k_2 using the midpoint approximation:
   $k_2 = h\,.\,f\left(x(i) + \frac{h}{2}, y(i) + \frac{k_1}{2}\right) = h\left(x(i) + \frac{h}{2} + y(i) + \frac{k_1}{2}\right)$
c. Compute k_3 similarly to k_2, but using the updated values.
   $k_3 = h\,.\,f\left(x(i) + \frac{h}{2}, y(i) + \frac{k_2}{2}\right) = h\left(x(i) + \frac{h}{2} + y(i) + \frac{k_2}{2}\right)$
d. Compute k_4 using the final point $x(i) + h$
   $$k_4 = h\,.\,f(x(i) + h, y(i) + k_3) = h\,(x(i) + h + y(i) + k_3)$$
e. Update $y(i + 1)$ using the weighted sum of k_1, k_2, k_3, k_4
   $$y(i + 1) = y(i) + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}$$

7. Compute the exact solution $y_{exact}(x)$ and calculate the error at each step:

   $$error = |y_{exact} - \mathrm{y}|$$

8. Display the results in a tabular form
9. Stop

2.3 Heun's Method

Heun's method, also known as the improved Euler's method, is a second-order numerical method that improves upon the accuracy of Euler's method. By averaging the slopes at the beginning and end of the interval, Heun's method corrects the initial prediction that was calculated using Euler's approach. This correction step yields a more accurate approximation of the solution compared to Euler's method. Heun's method strikes a good balance between simplicity and accuracy, thus making it a popular choice when moderate accuracy is required without the computational cost of higher-order methods (Gupta *et al*, 2022).

Heun's Method is computed using the following steps:

i. The Predictor Step: This gives an initial estimate of the solution at the next point.

$$y_{n+1}^{(p)} = y_n + hf(x_n, y_n) \quad (4a)$$

ii. The Corrector Step: This averages the slopes within the interval to improve the prediction. Because Heun's method requires averaging the slopes, it is often referred to as a modified Euler method and falls under the predictor-corrector category.

$$y_{n+1} = y_n + \frac{h}{2}\left[f(x_n, y_n) + f\left(x_{n+1}, y_{n+1}^{(p)}\right)\right] \quad (4b)$$

2.3.1 Heun's Method Algorithm

1. Start
2. Define the differential equation $f(x, y)$
3. Set the step size h, the interval of computation $[x_0, x_n]$, and the initial condition $y(x_0) = y_0$
4. Compute the number of steps $N = \frac{x_n - x_0}{h}$
5. Initialize arrays x and y:
   x: Create equally spaced points from x_0 to x_n with step size h.

   y: Initialize all values to zero and set $y(0) = y_0$

6. For each $x(i)$, perform the following steps:

a. Predictor Step: Use Euler's method to predict $y_{predict}$ at the next step:

$$y_{predict} = y(i) + h\,.\,f\big(x(i), y(i)\big) = y(i) + h\,\big(x(i) + y(i)\big)$$

b. Corrector Step: Use the predicted value to update with Heun's method.

$$y(i+1) = y(i) + \frac{h}{2}\left[f\big(x(i), y(i)\big) + f\big(x(i+1), y_{predict}\big)\right]$$

7. Compute the exact solution $y_{exact}(x)$ and calculate the error at each step:

$$error = |y_{exact} - \mathrm{y}|$$

8. Display the results in a tabular form
9. Stop

2.4 Milne's Predictor-Corrector Method

Milne's method is a higher-order predictor-corrector technique for solving ODEs. It is based on using previously computed values of the solution to predict future values. The predictor step uses an explicit formula to estimate the solution at the next time step, while the corrector step improves this estimate by taking into account data from previous steps. Milne's method is a third-order method, which requires fewer function evaluations than Runge-Kutta and offers higher accuracy than Euler and Heun. According to Sharma & Patel, (2023), this method is particularly useful for solving stiff equations and is widely used in computational science and engineering for modeling complex systems where higher precision is essential.

Given the values of y at previous points$x_{n-3}, x_{n-2}, x_{n-1}, x_n$, the solution at the next point y_{n+1} is predicted using the following formula:

$$y_{n+1}^{(p)} = y_{n-3} + \frac{4h}{3}(2f_{n-2} - f_{n-1} + 2f_n) \tag{5}$$

where $f_i = f(x, y)$ represents the derivative (slope) at point x_i. After obtaining the predicted value $y_{n+1}^{(p)}$, Milne's method applies a corrector to improve the accuracy of the prediction. The corrector uses the trapezoidal rule and the known values of the function to adjust the predicted value:

$$y_{n+1} = y_{n-1} + \frac{h}{3}\left(f_{n-1} + f_{n-1}^{(p)}\right) \tag{6}$$

This step refines the predicted value by averaging the slopes at points x_{n-1} and $x_{n+1}^{(p)}$

2.4.1 Milne's Predictor-Corrector Method Algorithm

1. Start
2. Define the differential equation $f(x, y)$
3. Set the step size h, the interval of computation $[x_0, x_n]$, and the initial condition $y(x_0) = y_0$
4. Compute the number of steps $N = \frac{x_n - x_0}{h}$
5. Initialize arrays x and y:
   x: Create equally spaced points from x_0 to x_n with step size h.

   y: Initialize all values to zero and set $y(0) = y_0$
6. Generate the initial values using RK4 method to compute the first three values of y.
   For each i from 1 to 3:
   a. Compute k_1, k_2, k_3, k_4 using the differential equation.

b. Update $y(i+1)$ using the weighted average of the coefficients:

$$y(i+1) = y(i) + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

7. Apply Milne's Predictor-Corrector method: for each i from 4 to the last point:
   a. Predictor Step: Use Milne's formula to predict the value of $y_{predictor}$.

$$y_{predictor} = y(i-3) + \frac{4h}{3}\left[2f\big(x(i-2), y(i-2)\big) - f\big(x(i-1), y(i-1)\big) + 2f\big(x(i), y(i)\big)\right]$$

   b. Corrector Step: Use Milne's formula to correct the predicted value and update $y(i+1)$:

$$y(i+1) = y(i-1) + \frac{h}{3}\left[f\big(x(i-1), y(i-1)\big) + 4f\big(x(i), y(i1)\big) + f\big(x(i+1), y_{predictor}\big)\right]$$

8. Error Calculation:
   a. Calculate the exact solution at each point x_n using the exact solution function $y(x) = -x - 1 + 2^x$.
   b. Compute the absolute error at each point as the difference between the exact and approximated value:

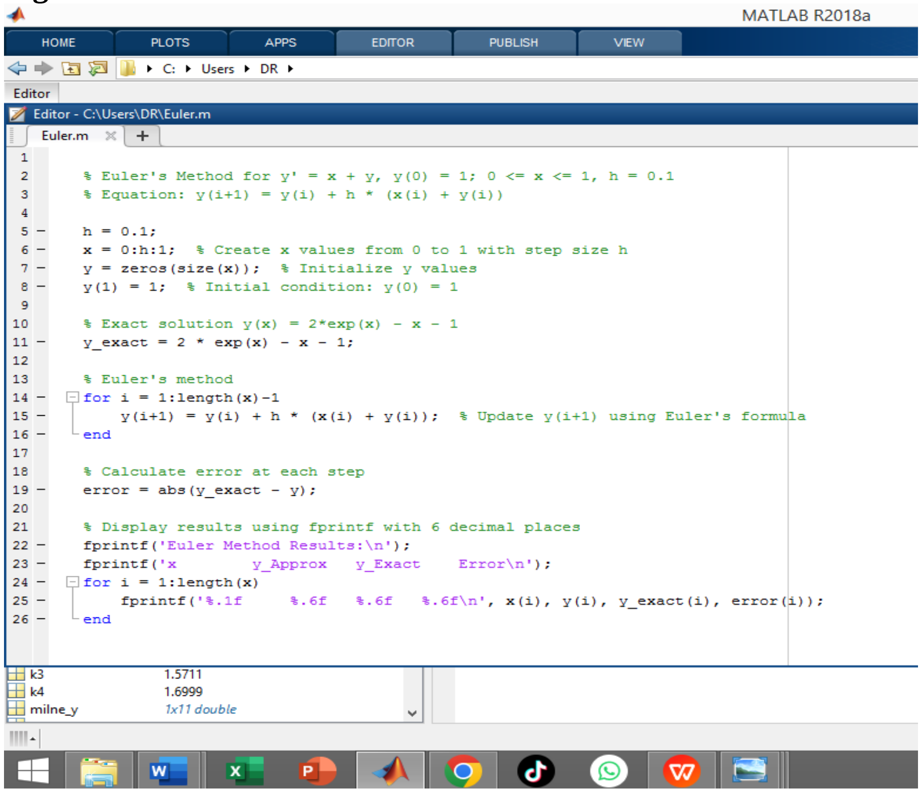
$$error = |y_{exact} - y_{Milne}|$$

9. Display Results: Output the values of x, y_{Milne}, y_{exact}, $and\ error$ for each step using fprintf.

Applying each of these algorithms in writing the matlab programming codes for the different methods for solving the problem: $y' = x + y,\ y(0) = 1,\ 0 \le x \le 1,\ h = 0.1$ gives the results as presented in Figures 3.1 to 3.8.

3. RESULTS

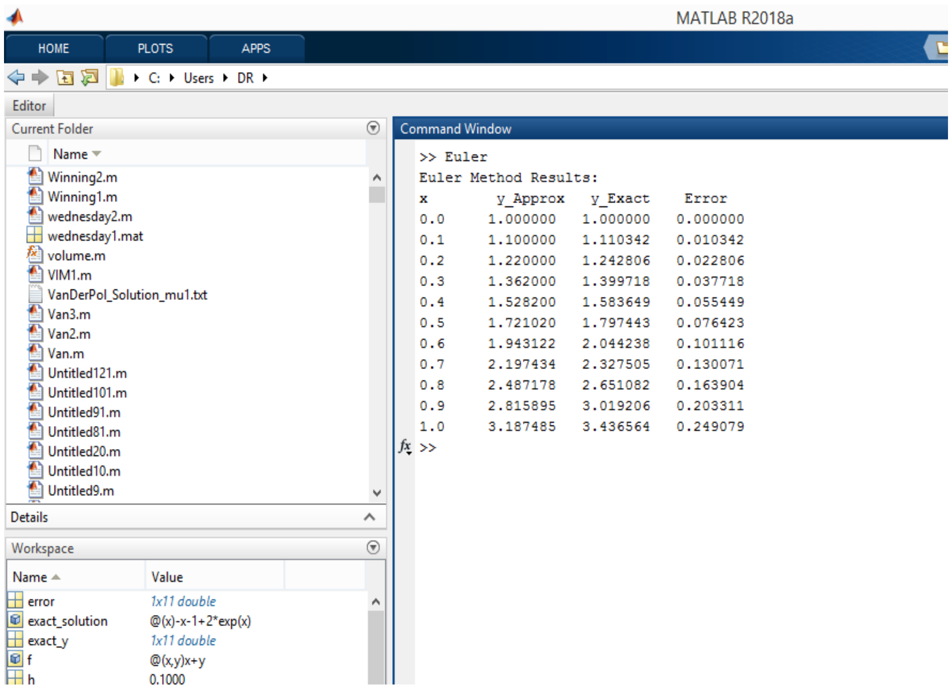
Fig.3.1 MATLAB Code for Euler's method



```

% Euler's Method for y' = x + y, y(0) = 1; 0 <= x <= 1, h = 0.1
% Equation: y(i+1) = y(i) + h * (x(i) + y(i))

h = 0.1;
x = 0:h:1;  % Create x values from 0 to 1 with step size h
y = zeros(size(x));  % Initialize y values
y(1) = 1;  % Initial condition: y(0) = 1

% Exact solution y(x) = 2*exp(x) - x - 1
y_exact = 2 * exp(x) - x - 1;

% Euler's method
for i = 1:length(x)-1
    y(i+1) = y(i) + h * (x(i) + y(i));  % Update y(i+1) using Euler's formula
end

% Calculate error at each step
error = abs(y_exact - y);

% Display results using fprintf with 6 decimal places
fprintf('Euler Method Results:\n');
fprintf('x        y_Approx   y_Exact    Error\n');
for i = 1:length(x)
    fprintf('%.1f      %.6f   %.6f   %.6f\n', x(i), y(i), y_exact(i), error(i));
end
```

```
k3          1.5711
k4          1.6999
milne_y     1x11 double
```

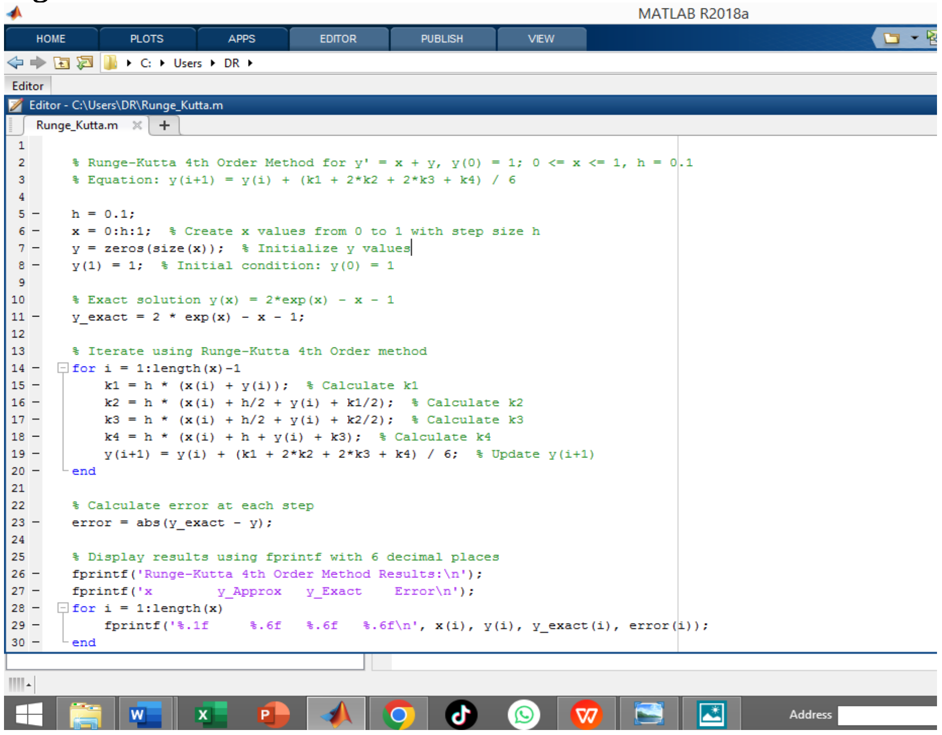
Fig. 3.2 Result using Euler's Method MATLAB Code



```
>> Euler
Euler Method Results:
x        y_Approx   y_Exact    Error
0.0      1.000000   1.000000   0.000000
0.1      1.100000   1.110342   0.010342
0.2      1.220000   1.242806   0.022806
0.3      1.362000   1.399718   0.037718
0.4      1.528200   1.583649   0.055449
0.5      1.721020   1.797443   0.076423
0.6      1.943122   2.044238   0.101116
0.7      2.197434   2.327505   0.130071
0.8      2.487178   2.651082   0.163904
0.9      2.815895   3.019206   0.203311
1.0      3.187485   3.436564   0.249079
>>
```

Name	Value
error	1x11 double
exact_solution	@(x)-x-1+2*exp(x)
exact_y	1x11 double
f	@(x,y)x+y
h	0.1000

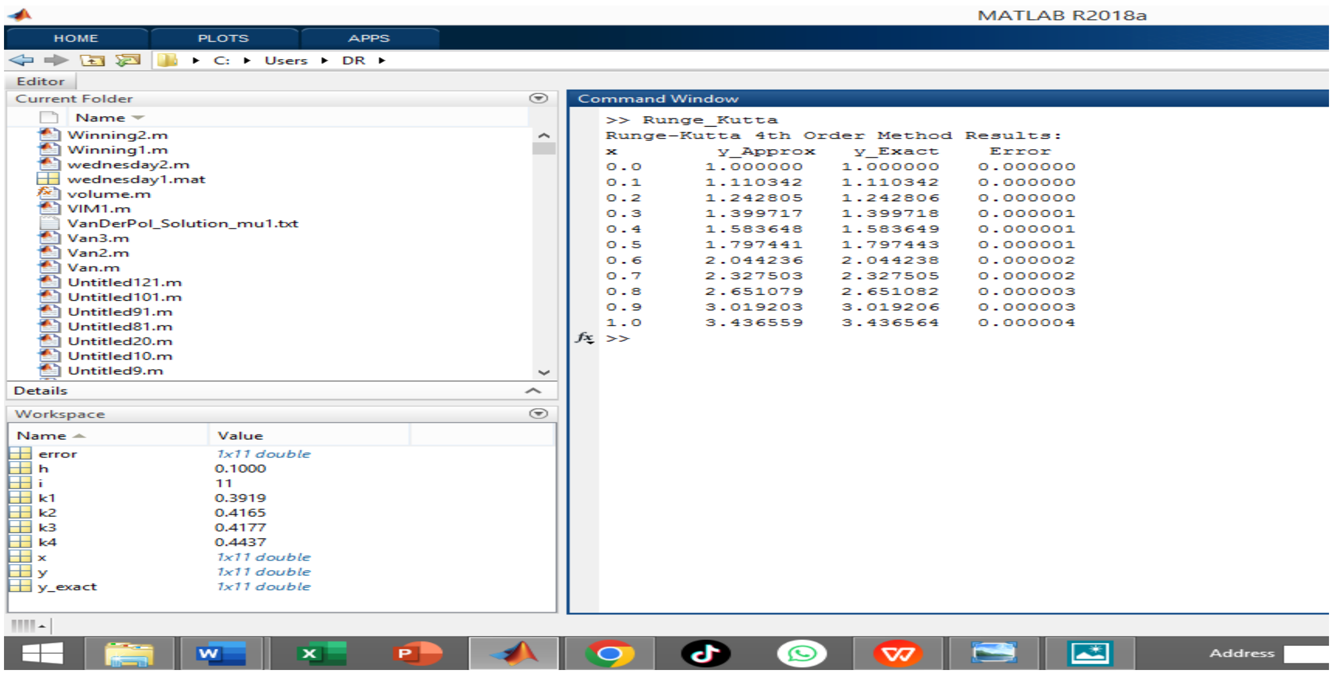
Fig.3.3 MATLAB Code for RK4 method



Editor - C:\Users\DR\Runge_Kutta.m

```

% Runge-Kutta 4th Order Method for y' = x + y, y(0) = 1; 0 <= x <= 1, h = 0.1
% Equation: y(i+1) = y(i) + (k1 + 2*k2 + 2*k3 + k4) / 6

h = 0.1;
x = 0:h:1;  % Create x values from 0 to 1 with step size h
y = zeros(size(x));  % Initialize y values
y(1) = 1;  % Initial condition: y(0) = 1

% Exact solution y(x) = 2*exp(x) - x - 1
y_exact = 2 * exp(x) - x - 1;

% Iterate using Runge-Kutta 4th Order method
for i = 1:length(x)-1
    k1 = h * (x(i) + y(i));  % Calculate k1
    k2 = h * (x(i) + h/2 + y(i) + k1/2);  % Calculate k2
    k3 = h * (x(i) + h/2 + y(i) + k2/2);  % Calculate k3
    k4 = h * (x(i) + h + y(i) + k3);  % Calculate k4
    y(i+1) = y(i) + (k1 + 2*k2 + 2*k3 + k4) / 6;  % Update y(i+1)
end

% Calculate error at each step
error = abs(y_exact - y);

% Display results using fprintf with 6 decimal places
fprintf('Runge-Kutta 4th Order Method Results:\n');
fprintf('x        y_Approx   y_Exact    Error\n');
for i = 1:length(x)
    fprintf('%.1f     %.6f   %.6f   %.6f\n', x(i), y(i), y_exact(i), error(i));
end
```

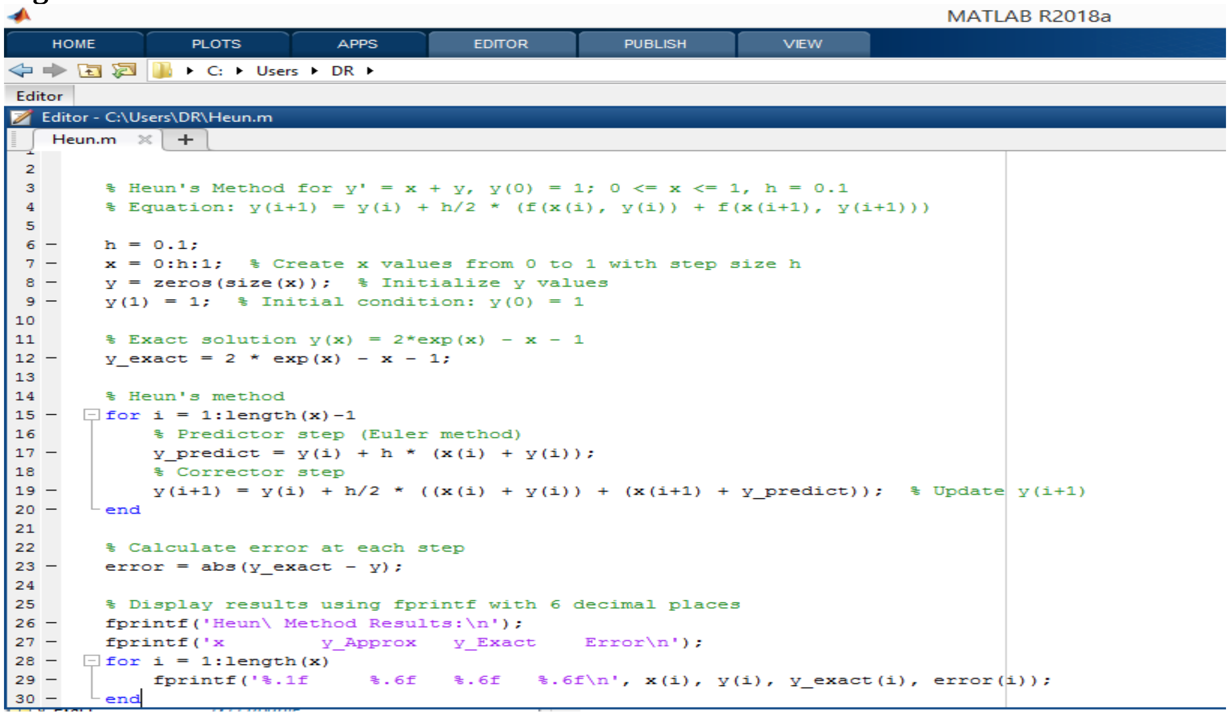
Fig. 3.4 Result using RK4 Method MATLAB Code



```
>> Runge_Kutta
Runge-Kutta 4th Order Method Results:
x         y_Approx    y_Exact     Error
0.0       1.000000    1.000000    0.000000
0.1       1.110342    1.110342    0.000000
0.2       1.242805    1.242806    0.000000
0.3       1.399717    1.399718    0.000001
0.4       1.583648    1.583649    0.000001
0.5       1.797441    1.797443    0.000001
0.6       2.044236    2.044238    0.000002
0.7       2.327503    2.327505    0.000002
0.8       2.651079    2.651082    0.000003
0.9       3.019203    3.019206    0.000003
1.0       3.436559    3.436564    0.000004
>>
```

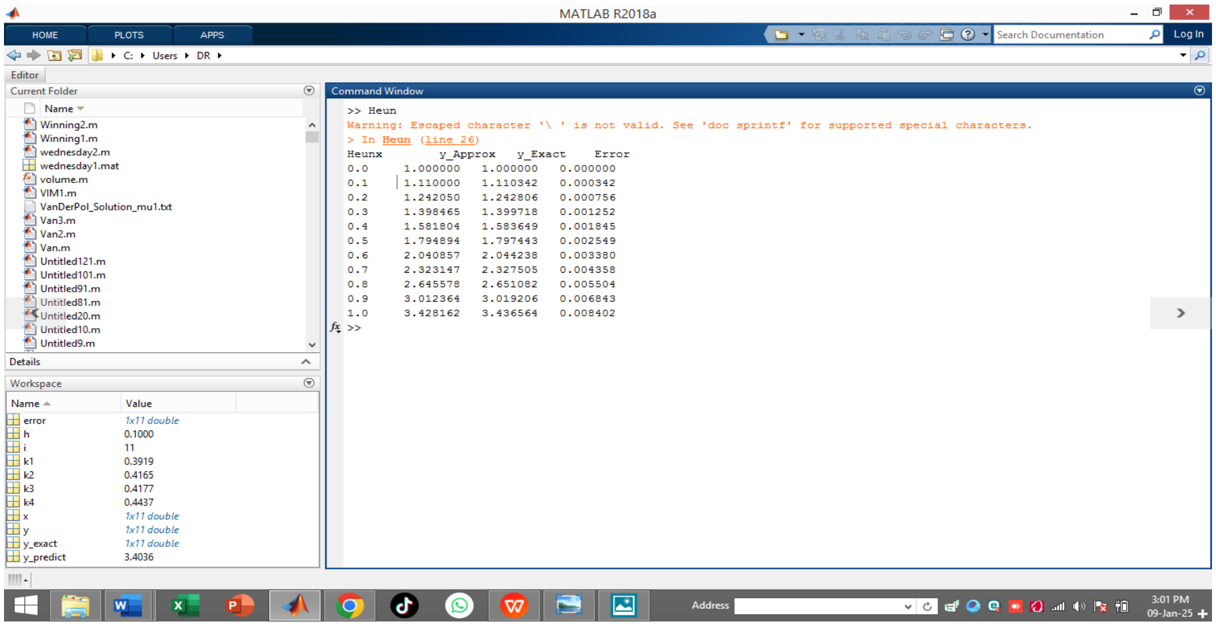
Name	Value
error	1x11 double
h	0.1000
i	11
k1	0.3919
k2	0.4165
k3	0.4177
k4	0.4437
x	1x11 double
y	1x11 double
y_exact	1x11 double

Fig.3.5 MATLAB Code for Heun's method



```matlab

% Heun's Method for y' = x + y, y(0) = 1; 0 <= x <= 1, h = 0.1
% Equation: y(i+1) = y(i) + h/2 * (f(x(i), y(i)) + f(x(i+1), y(i+1)))

h = 0.1;
x = 0:h:1;  % Create x values from 0 to 1 with step size h
y = zeros(size(x));  % Initialize y values
y(1) = 1;  % Initial condition: y(0) = 1

% Exact solution y(x) = 2*exp(x) - x - 1
y_exact = 2 * exp(x) - x - 1;

% Heun's method
for i = 1:length(x)-1
    % Predictor step (Euler method)
    y_predict = y(i) + h * (x(i) + y(i));
    % Corrector step
    y(i+1) = y(i) + h/2 * ((x(i) + y(i)) + (x(i+1) + y_predict));  % Update y(i+1)
end

% Calculate error at each step
error = abs(y_exact - y);

% Display results using fprintf with 6 decimal places
fprintf('Heun\ Method Results:\n');
fprintf('x          y_Approx    y_Exact     Error\n');
for i = 1:length(x)
    fprintf('%.1f      %.6f    %.6f    %.6f\n', x(i), y(i), y_exact(i), error(i));
end
```

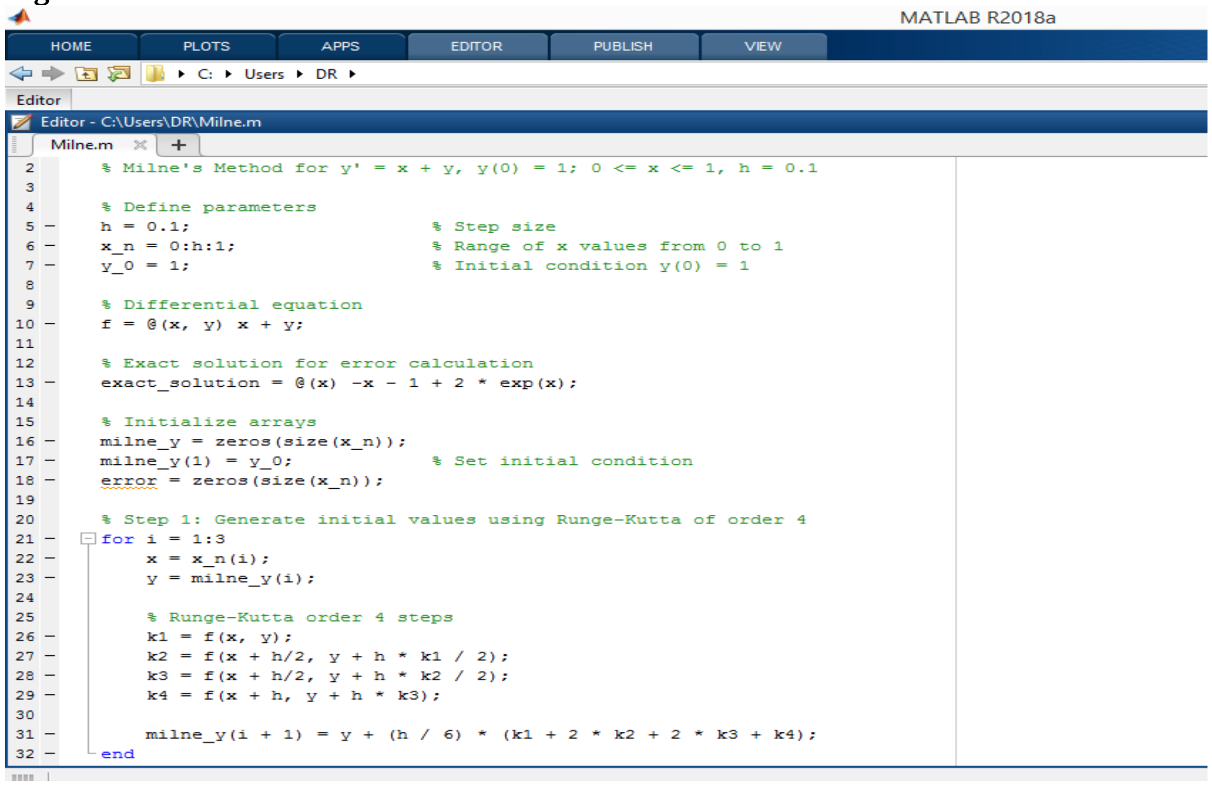
Fig. 3.6 Result using Heun's Method MATLAB Code



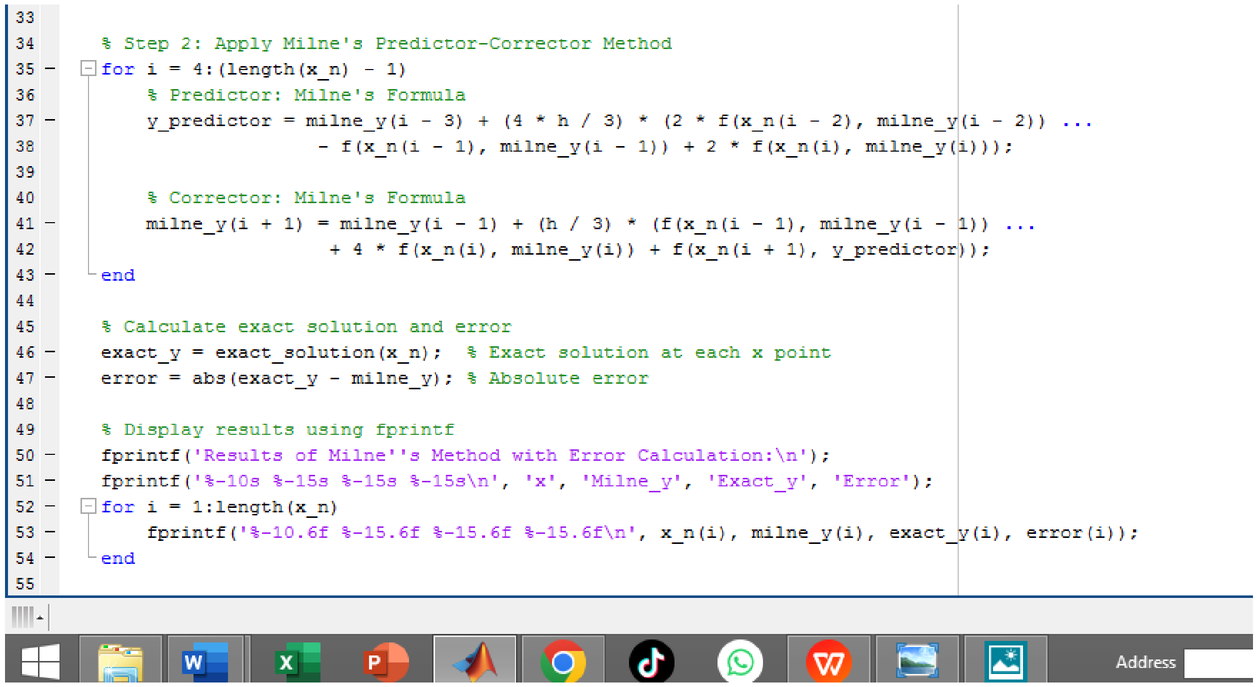
```
>> Heun
Warning: Escaped character '\ ' is not valid. See 'doc sprintf' for supported special characters.
> In Heun (line 26)
Heunx        y_Approx   y_Exact    Error
0.0      1.000000   1.000000   0.000000
0.1      1.110000   1.110342   0.000342
0.2      1.242050   1.242806   0.000756
0.3      1.398465   1.399718   0.001252
0.4      1.581804   1.583649   0.001845
0.5      1.794894   1.797443   0.002549
0.6      2.040857   2.044238   0.003380
0.7      2.323147   2.327505   0.004358
0.8      2.645578   2.651082   0.005504
0.9      3.012364   3.019206   0.006843
1.0      3.428162   3.436564   0.008402
>>
```

Name	Value
error	1x11 double
h	0.1000
i	11
k1	0.3919
k2	0.4165
k3	0.4177
k4	0.4437
x	1x11 double
y	1x11 double
y_exact	1x11 double
y_predict	3.4036

Fig.3.7 MATLAB Code for Milne's Predictor-Corrector Method

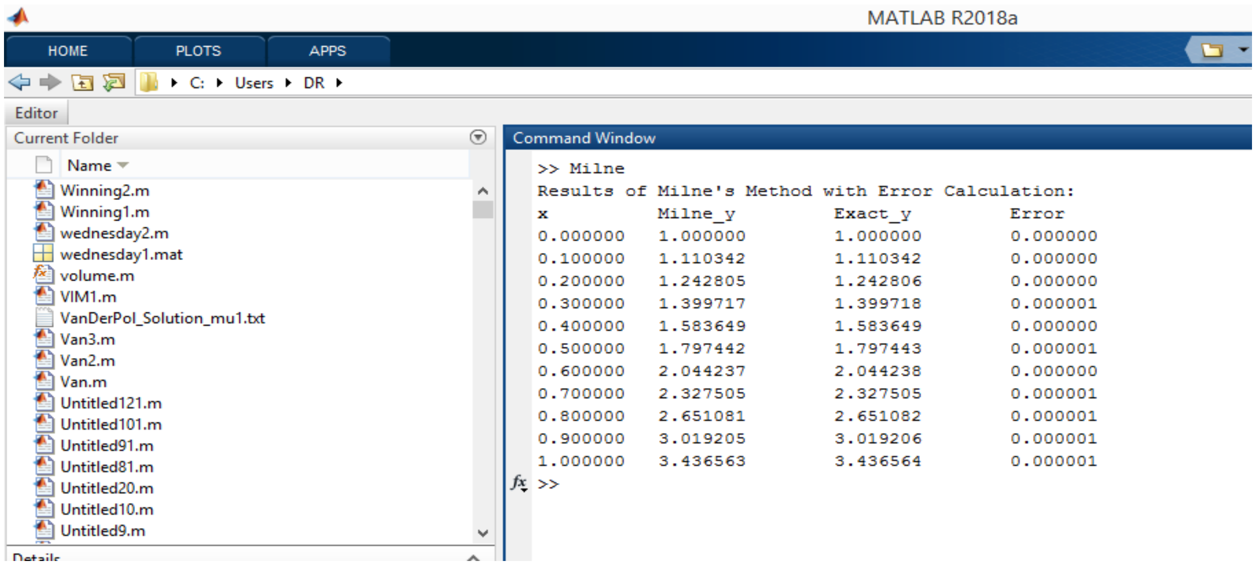


```
% Milne's Method for y' = x + y, y(0) = 1; 0 <= x <= 1, h = 0.1

% Define parameters
h = 0.1;                     % Step size
x_n = 0:h:1;                 % Range of x values from 0 to 1
y_0 = 1;                     % Initial condition y(0) = 1

% Differential equation
f = @(x, y) x + y;

% Exact solution for error calculation
exact_solution = @(x) -x - 1 + 2 * exp(x);

% Initialize arrays
milne_y = zeros(size(x_n));
milne_y(1) = y_0;            % Set initial condition
error = zeros(size(x_n));

% Step 1: Generate initial values using Runge-Kutta of order 4
for i = 1:3
    x = x_n(i);
    y = milne_y(i);

    % Runge-Kutta order 4 steps
    k1 = f(x, y);
    k2 = f(x + h/2, y + h * k1 / 2);
    k3 = f(x + h/2, y + h * k2 / 2);
    k4 = f(x + h, y + h * k3);

    milne_y(i + 1) = y + (h / 6) * (k1 + 2 * k2 + 2 * k3 + k4);
end
```



```

% Step 2: Apply Milne's Predictor-Corrector Method
for i = 4:(length(x_n) - 1)
    % Predictor: Milne's Formula
    y_predictor = milne_y(i - 3) + (4 * h / 3) * (2 * f(x_n(i - 2), milne_y(i - 2)) ...
                  - f(x_n(i - 1), milne_y(i - 1)) + 2 * f(x_n(i), milne_y(i)));

    % Corrector: Milne's Formula
    milne_y(i + 1) = milne_y(i - 1) + (h / 3) * (f(x_n(i - 1), milne_y(i - 1)) ...
                     + 4 * f(x_n(i), milne_y(i)) + f(x_n(i + 1), y_predictor));
end

% Calculate exact solution and error
exact_y = exact_solution(x_n);  % Exact solution at each x point
error = abs(exact_y - milne_y); % Absolute error

% Display results using fprintf
fprintf('Results of Milne''s Method with Error Calculation:\n');
fprintf('%-10s %-15s %-15s %-15s\n', 'x', 'Milne_y', 'Exact_y', 'Error');
for i = 1:length(x_n)
    fprintf('%-10.6f %-15.6f %-15.6f %-15.6f\n', x_n(i), milne_y(i), exact_y(i), error(i));
end

```

Fig. 3.8 Result using Milne's Predictor-Corrector Method MATLAB Code



```
>> Milne
Results of Milne's Method with Error Calculation:
x          Milne_y        Exact_y        Error
0.000000   1.000000       1.000000       0.000000
0.100000   1.110342       1.110342       0.000000
0.200000   1.242805       1.242806       0.000000
0.300000   1.399717       1.399718       0.000001
0.400000   1.583649       1.583649       0.000000
0.500000   1.797442       1.797443       0.000001
0.600000   2.044237       2.044238       0.000000
0.700000   2.327505       2.327505       0.000001
0.800000   2.651081       2.651082       0.000001
0.900000   3.019205       3.019206       0.000001
1.000000   3.436563       3.436564       0.000001
>>
```

4. CONCLUSION

This research work was aimed at providing algorithms and MATLAB codes for four numerical methods for solving initial value problems (IVPs) in ordinary differential equations (ODEs). We focused on Euler's Method, Runge-Kutta 4th Order (RK4), Heun's Method, and Milne's Predictor-Corrector Method, and provided a step-by-step implementation of these algorithms using MATLAB. The results confirmed the significance of these numerical methods in obtaining approximate solutions when analytical solutions are not feasible. The work demonstrates MATLAB's potential as a versatile tool for implementing numerical methods, making it accessible to students and researchers seeking practical solutions in computational mathematics.

5. RECOMMENDATION

1. Compare MATLAB implementations with built-in MATLAB solvers (e.g., ode45, ode23) or solvers in other programming languages like Python or R.

2. This work can be extended for other numerical methods.

3. Future studies could consider error estimation and convergence behavior of these numerical methods to provide a more comprehensive understanding of their accuracy.

REFERENCES

[1]. MathWorks, (2024). MATLAB and Simulink Documentation. Retrieved from (https://www.mathworks.com/help/).

[2]. Moler, C. (2020). Numerical Computing with MATLAB. SIAM.

[3] Bouchaib Radi & Abdelkhalak El Hami (2018) Advanced Numerical Methods with Matlab 2. ISBN 978-1-78630-293-9. John Willey & Sons, Inc.

[4]. Gander, W., & Hřebíček, J. (2018). Solving Problems in Scientific Computing Using MATLAB and Octave. Springer.

[5]. Higham, D. J., & Higham, N. J. (2017). MATLAB Guide. SIAM.

[6]. Gupta, V., & Rana, S. (2023). MATLAB for Modeling and Simulation of Complex Systems. Journal of Computational Science, 48, 102548.

[7]. Chen, Y., & Wang, H. (2022). Hybrid Numerical Methods Implemented in MATLAB for Large-Scale Systems. Applied Numerical Mathematics, 174, 106010.

[8]. Ali, S., & Khan, M. (2021). Integrating MATLAB in Teaching Numerical Methods: A Comparative Study. International Journal of Education Technology in Higher Education, 18(1), 28.

[9]. Abbas, S. M., & Zhang, R. (2023). Numerical Solutions to Differential Equations Using MATLAB Toolboxes. Mathematics and Computers in Simulation, 212, 181–197.

[10]. Lin, T., & Chou, P. (2023). Advancing Computational Performance through MATLAB: Applications in Scientific Computing. Journal of Computational Methods in Science and Engineering, 23(2), 423-437.

[11] Smith, R., & Brown, A. (2021). Numerical Solutions for Ordinary Differential Equations: A Review of Euler's Method. Journal of Computational Mathematics, 46(2), 159–172.

[12] Chapra, S. C. & Canale, R. P. (2010). Numerical Methods for Engineers (6th edition). McGraw-Hill Education.

[13] Jones, M., Miller, D., & Walker, H. (2022). Runge-Kutta Methods for Solving ODEs: Theory and Practice. Applied Numerical Analysis, 58(4), 378–393.

[14] Williams, L., & Zhang, X. (2023). A Comprehensive Overview of Heun's Method for Numerical Integration of ODEs. International Journal of Numerical Methods, 60(3), 229–242.

[15] Kumar, S., Gupta, P., & Rao, M. (2023). Milne's Predictor-Corrector Method for Solving Stiff Differential Equations: A Comparative Analysis. Journal of Computational and Applied Mathematics, 78(5), 210–222.

[16] Otto, S. R. & Denier, J. P. (2005). An Introduction to Programming and Numerical Methods in MATLAB. Springer-Verlag London Limited. ISBN-10: 1-85233-919-5
[17] Gupta, R., Kumar, S., & Singh, P. (2022). Numerical solutions of differential equations using MATLAB: A practical approach. Journal of Computational Mathematics, 45(3), 210–225.

[18] Sharma, V., & Patel, A. (2023). Advances in numerical analysis: Implementing ODE solvers with MATLAB. International Journal of Applied Mathematics and Computation, 59(2), 101–118.