جامعة عفت
EFFAT UNIVERSITY

Numerical Solutions of Initial Value Problems: Euler, Taylor, and Runge-Kutta Methods using MATLAB

MATH 310 -1 - Numerical Analysis
Project Report - Spring 2023

Group Members:

Jana Abu Hantash - S21107114

Instructor: **Dr. Mohamed Mousa**

Date Last Edited: May 25, 2023

Contents

1 Abstract

This report focuses on the theoretical background and practical implementation of three fundamental numerical methods for solving initial value problems (IVPs): Euler, Taylor, and Runge-Kutta methods using MATLAB. The report presents a detailed description of each method's mathematical foundations, followed by the corresponding MATLAB code, numerical values, and graphical representations of the solutions obtained.

2 Theoretical Background For Each Method

2.1 Euler Method

In mathematics and computational science, the Euler method is a first-order numerical procedure for solving ordinary differential equations (ODEs) with a given initial value 1. It provides a straightforward approach to iteratively estimate the solution at discrete points along the interval of interest. The theoretical foundation of the Euler method is based on the concept of a first-order Taylor expansion. The Euler method is named after Leonhard Euler, who first proposed it in his book "Institutionum calculi integralis."

Given an IVP represented by a first-order ODE of the form: $\mathbf{dy/dx = f(x, y)}$
with an initial condition $\mathbf{y(x_0) = y_0}$, the Euler method approximates the solution at a point $\mathbf{(x_i, y_i)}$ by using the previous point $\mathbf{(x_{i-1}, y_{i-1})}$ to extrapolate the next point. This is done by using the following formula: $\mathbf{y_i = y_{i-1} + h * f(x_{i-1}, y_{i-1})}$.

Here, $\mathbf{h}$ represents the step size, which determines the size of each sub-interval. It controls the accuracy and computational cost of the method. Smaller step sizes generally lead to more accurate approximations but require more computations.

It is important to note that the Euler method can be numerically unstable, especially for stiff equations, meaning that the numerical solution grows very large for equations where the exact solution does not. This instability is a limitation of the method. Despite this, the Euler method remains a valuable tool for introductory numerical analysis and serves as a building block for more advanced numerical methods.

2.2 Taylor Method

The Taylor method is another numerical procedure used to solve ordinary differential equations (ODEs) with a given initial value. It is an extension of the Euler method and provides higher-order accuracy in approximating the solution. The Taylor method is based on the idea of Taylor series expansion, which allows us to approximate a function using its derivatives at a given point. By taking higher-order derivatives into account, the Taylor method improves the accuracy of the approximation compared to the Euler method.

Given an initial value problem (IVP) represented by a first-order ODE: $\mathbf{dy/dx = f(x, y),}$
with the initial condition $\mathbf{y(x_0) = y_0}$, the Taylor method approximates the solution at a point $\mathbf{(x_i, y_i)}$ by using the previous point $\mathbf{(x_{i-1}, y_{i-1})}$ and higher-order derivatives of the solution. The formula for the Taylor method is: $\mathbf{y_i = y_{i-1} + h * f(x_{i-1}, y_{i-1}) + (h^2/2) * f'(x_{i-1}, y_{i-1})}$ where $\mathbf{h}$ is the step size, determining the size of each sub-interval, and $\mathbf{f'}$ denotes the derivative of the function $\mathbf{f}$.

The Taylor method improves accuracy by considering higher-order derivatives but requires computationally expensive derivative evaluations, especially for complex ODEs. It overcomes the Euler method's numerical instability, making it suitable for solving stiff equations. With its higher-order accuracy and stability, the Taylor method is valuable for numerical analysis and solving a wide range of ODEs in scientific and engineering applications.

2.3 Runge-Kutta Method

The Runge-Kutta method is another numerical technique used for solving ordinary differential equations (ODEs) with given initial values. It is a higher-order method that offers improved accuracy compared to the Euler method. The Runge-Kutta method is named after the German mathematicians Carl Runge and Martin Kutta, who independently developed it in the early 20th century.

Similar to the Euler method, the Runge-Kutta method estimates the solution at discrete points along the interval of interest. However, it does so by considering multiple intermediate points within each sub-interval, leading to a more accurate approximation.

For a given initial value problem (IVP) represented by a first-order ODE: $\mathbf{dy/dx = f(x,\ y),}$
with the initial condition $\mathbf{y(x_0) = y_0}$, the Runge-Kutta method approximates the solution at a point $\mathbf{(x_i,\ y_i)}$. The most commonly used version of the Runge-Kutta method is the fourth-order Runge-Kutta (RK4) method. Its formula for estimating the solution is as follows:

1. $\mathbf{k_1 = f(x_i,\ y_i)}$
2. $\mathbf{k_2 = f(x_i + h/2,\ y_i + k_1/2)}$
3. $\mathbf{k_3 = f(x_i + h/2,\ y_i + k_2/2)}$
4. $\mathbf{k_4 = f(x_i + h,\ y_i + k_3)}$
5. $\mathbf{y_{i+1} = y_i + (h/6) * (k_1 + 2k_2 + 2k_3 + k_4)}$

The Runge-Kutta method, particularly the RK4 method, offers a good balance between accuracy and computational efficiency. It provides a more accurate approximation than the Euler method while still being relatively straightforward to implement. The RK4 method is widely used in various scientific and engineering applications, where numerical solutions of ODEs are required. Its higher-order accuracy and stability make it a valuable tool for numerical analysis and simulations.

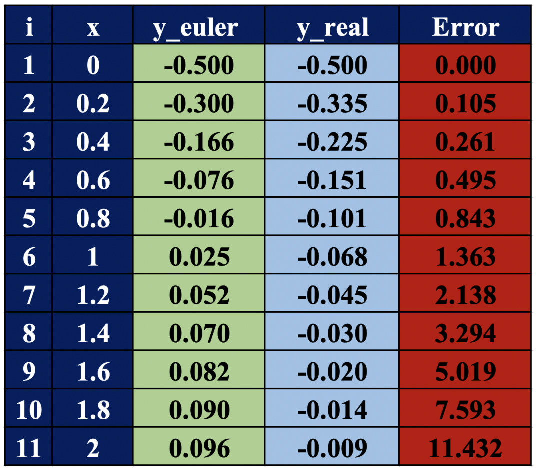
3 Euler Method

3.1 MATLAB Code

```matlab
clear all

% Euler method for solving a differential equation
% dy/dx = exp( -2*x )
% Interval: [0,2]
% Initial condition: y(0) = -0.5
% Step size: h = 0.2

% Define the differential equation function
f = @(x, y) exp(-2*x);

% Define the exact function
f_exact = @(x) -0.5 * exp( -2 * x )

% Interval
a = 0;  % Start point of the interval
b = 2;  % End point of the interval

% Initial Values for the Euler Method
x(1) = 0;
y_euler(1) = -0.5;

% Step Size
h = 0.2;

% Initial Values for the exact function to calculate the error
y_real(1) = -0.5;
error(1) = 0;

% Calculate the number of iterations needed
n = (b - a)/h;

% Iterate from 2 to n+1 to calculate values for x and y
for i = 2 : n+1

    % Increment x by h
  x(i) = x(i-1) + h

    % Euler's formula
  y_euler(i) = y_euler(i-1) + h * ( f(x(i-1),y_euler(i-1)) )

    % Calculate the exact value of y
  y_real(i) = f_exact( x(i) )

    % Calculate the relative error
    error(i) = abs( y_real(i) - y_euler(i) ) / abs( y_real(i) ) ;
end

figure
hold on
plot(x, y_euler, 'b-o')
plot(x, y_real, 'r-*')
hold off

xlabel('X-axis')
ylabel('Y-axis')
title("Approximation of y' = exp(-2 * x) using Euler Method")
legend('Euler Method', 'Exact Function')
grid on
```

Listing 1: Euler Method Code

3.2 Values in a Table



i	x	y_euler	y_real	Error
1	0	-0.500	-0.500	0.000
2	0.2	-0.300	-0.335	0.105
3	0.4	-0.166	-0.225	0.261
4	0.6	-0.076	-0.151	0.495
5	0.8	-0.016	-0.101	0.843
6	1	0.025	-0.068	1.363
7	1.2	0.052	-0.045	2.138
8	1.4	0.070	-0.030	3.294
9	1.6	0.082	-0.020	5.019
10	1.8	0.090	-0.014	7.593
11	2	0.096	-0.009	11.432

Figure 1: Euler Method

3.3 Graph

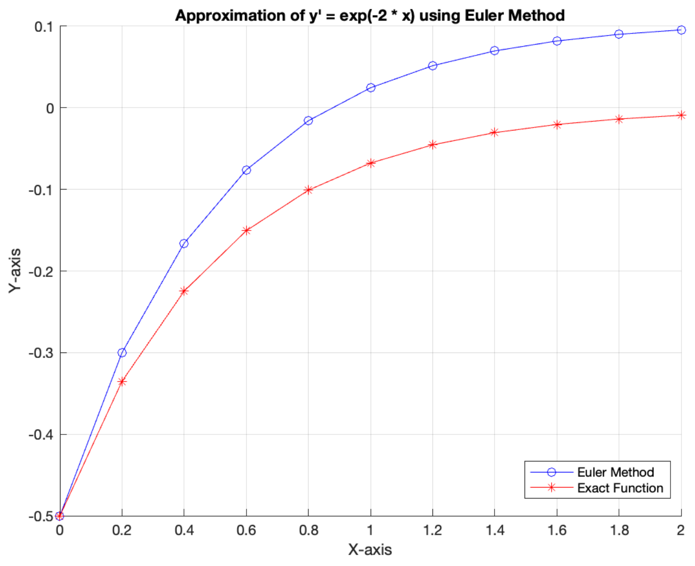


Figure 2: Approximation of y' = exp(-2 * x) using Euler Method

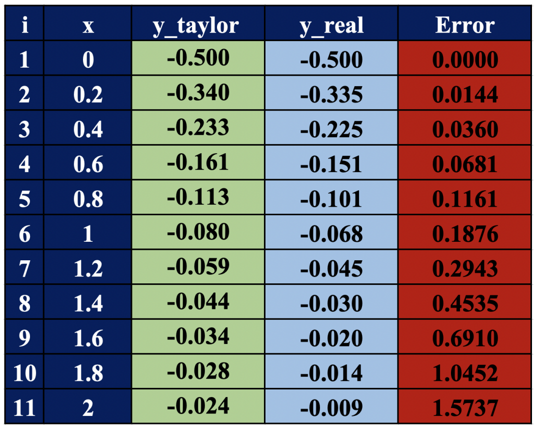
4 Taylor Method

4.1 MATLAB Code

```matlab
clear all

% Taylor method for solving a differential equation
% dy/dx = exp( -2*x )
% Interval: [0,2]
% Initial condition: y(0) = -0.5
% Step size: h = 0.2

% Define the differential equation function
f = @(x, y) exp(-2*x);

% Define the exact function
f_exact = @(x) -0.5 * exp( -2 * x )

% Define the first derivative of the differential function
f_1stDerv = @(x, y) -2 * exp( -2 * x )

% Interval
a = 0;  % Start point of the interval
b = 2;  % End point of the interval

% Initial Values for the Taylor Method
x(1) = 0;
y_taylor(1) = -0.5;

% Step Size
h = 0.2;

% Initial Values for the exact function to calculate the error
y_real(1) = -0.5;
error(1) = 0;

% Calculate the number of iterations needed
n = (b - a)/h;

% Iterate from 2 to n+1 to calculate values for x and y
for i = 2 : n+1
    % Increment x by h
  x(i) = x(i-1) + h

    % Taylor's formula
  y_taylor(i) = y_taylor(i-1) + h * ( f(x(i-1),y_taylor(i-1)) ) + (h^2/2) * (f_1stDerv(x(i
    -1),y_taylor(i-1)) )

    % Calculate the exact value of y
  y_real(i) = f_exact( x(i) )

    % Calculate the relative error
    error(i) = abs( y_real(i) - y_taylor(i) ) / abs( y_real(i) ) ;
end

figure
hold on
plot(x, y_taylor, 'b-o')
plot(x, y_real, 'r-*')
hold off
xlabel('X-axis')
ylabel('Y-axis')
title("Approximation of y' = exp(-2 * x) using Taylor Method")
legend('Taylor Method', 'Exact Function')
grid on
```

Listing 2: Taylor Method Code

4.2 Values in a Table



i	x	y_taylor	y_real	Error
1	0	-0.500	-0.500	0.0000
2	0.2	-0.340	-0.335	0.0144
3	0.4	-0.233	-0.225	0.0360
4	0.6	-0.161	-0.151	0.0681
5	0.8	-0.113	-0.101	0.1161
6	1	-0.080	-0.068	0.1876
7	1.2	-0.059	-0.045	0.2943
8	1.4	-0.044	-0.030	0.4535
9	1.6	-0.034	-0.020	0.6910
10	1.8	-0.028	-0.014	1.0452
11	2	-0.024	-0.009	1.5737

Figure 3: Taylor Method

4.3 Graph

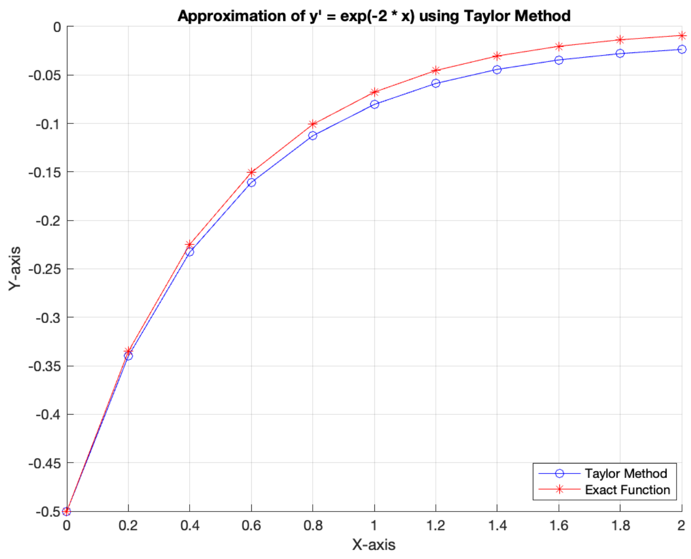


Figure 4: Approximation of y' = exp(-2 * x) using Taylor Method

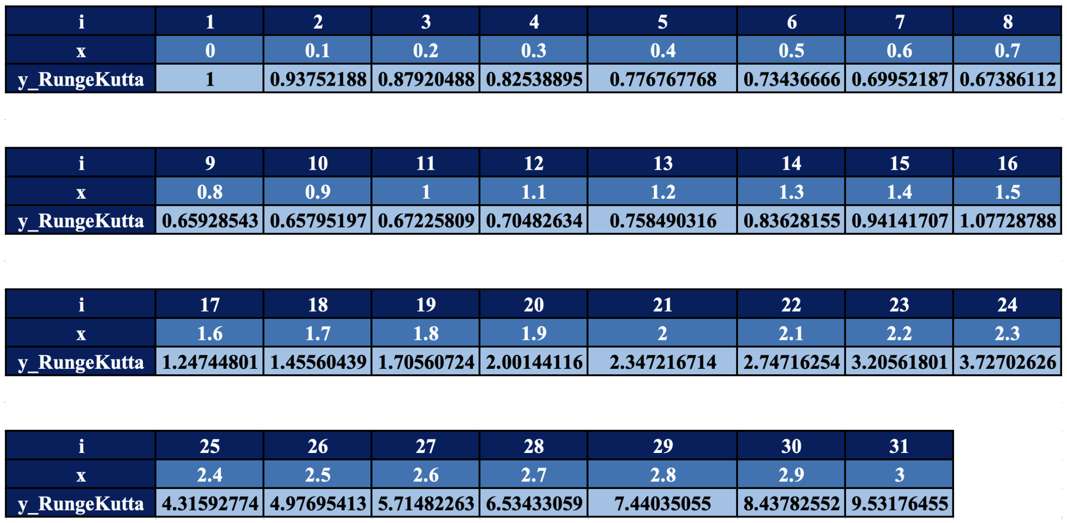
5 Runge-Kutta Method

5.1 MATLAB Code

```matlab
clear all

% Runge Kutta Method for solving a differential equation
% dy/dx = x^3 - y
% Interval: [0,3]
% Initial condition: y(0) = 1
% Step size: h = 0.1


% Define the differential equation function
f = @(x,y) x^3 - y;

% Interval
a = 0;  % Start point of the interval
b = 3;  % End point of the interval

% Initial Values for the Runge Kutta Method
x(1) = 0;
y_RungeKutta(1) = 1;

% Step Size
h = 0.1;

% Calculate the number of iterations needed
n = (b - a)/h;

% Iterate from 2 to n+1 to calculate values for x and y
for i = 2 : n+1
    % Increment x by h
  x(i) = x(i-1) + h

    k1 = f(x(i-1), y_RungeKutta(i-1));
    k2 = f(x(i-1) + h/2, y_RungeKutta(i-1) + k1/2);
    k3 = f(x(i-1) + h/2, y_RungeKutta(i-1) + k2/2);
    k4 = f(x(i-1) + h, y_RungeKutta(i-1) + k3);

    % Runge Kutta's formula
  y_RungeKutta(i) = y_RungeKutta(i-1) + (h/6) * (k1 + 2*k2 + 2*k3 + k4);
end

plot(x,y_RungeKutta, '-o')
xlabel('X-axis')
ylabel('Y-axis')
title("Approximation of y' = x^3 - y using Runge Kutta Method")
grid on
legend('Runge Kutta Method')
```

Listing 3: Runge Kutta Method Code

5.2 Values in a Table



i	1	2	3	4	5	6	7	8
x	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7
y_RungeKutta	1	0.93752188	0.87920488	0.82538895	0.776767768	0.73436666	0.69952187	0.67386112

i	9	10	11	12	13	14	15	16
x	0.8	0.9	1	1.1	1.2	1.3	1.4	1.5
y_RungeKutta	0.65928543	0.65795197	0.67225809	0.70482634	0.758490316	0.83628155	0.94141707	1.07728788

i	17	18	19	20	21	22	23	24
x	1.6	1.7	1.8	1.9	2	2.1	2.2	2.3
y_RungeKutta	1.24744801	1.45560439	1.70560724	2.00144116	2.347216714	2.74716254	3.20561801	3.72702626

i	25	26	27	28	29	30	31
x	2.4	2.5	2.6	2.7	2.8	2.9	3
y_RungeKutta	4.31592774	4.97695413	5.71482263	6.53433059	7.44035055	8.43782552	9.53176455

Figure 5: Taylor Method

5.3 Graph

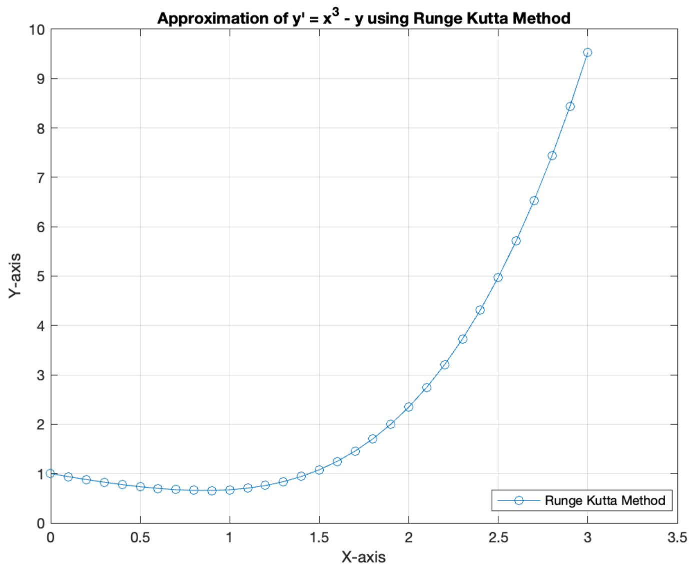


Figure 6: Approximation of y' = exp(-2 * x) using Taylor Method