Mathematical Sciences Letters
An International Journal

http://dx.doi.org/10.18576/msl/120203

Review on one- and multi-step finite difference methods for numerically solving first-order IVPs

*Soliman Elhassanein and Youssri Hassan Youssri**

Faculty of Engineering, Egypt University of Informatics, Knowledge City, New Administrative Capital, Egypt

Received: 15 January 2023, Revised: 10 March 2023, Accepted: 20 March 2023
Published online: 1 May 2023

Abstract: Herein, we present an overview of classical finite difference methods for numerically solving first-order initial value problems, these methods encompass: Adams-Bashforth Method, Adams-Moulton Method, Euler's Method, Taylor Method, Fourth Order Runge-Kutta Method, Simpson's Method, and Trapezoidal Rule Method, we include a dynamic Python code for all methods. We depict our results in tables and figures.

Keywords: Finite Difference Method; one-step method; multi-step method; Python

1 Introduction

In this research endeavor, we undertook an in-depth exploration of the finite difference methodologies, primarily concentrating on their applications for addressing first-order Initial Value Problems (IVPs) pertinent to Ordinary Differential Equations (ODEs). A crucial facet of our investigation was to delve into the synergies between these numerical techniques and the versatile Python programming environment, emphasizing dynamic code implementations.For more details about numerical solutions of differential equations, the interested readers are referred to [2, 3].

1.1 Motivations for Using Numerical Techniques

Numerical methodologies, such as the finite difference methods, play an indispensable role in the realm of differential equations. The reasons for their significance are multifaceted:

1. Many first-order IVPs do not have closed-form solutions, making numerical techniques invaluable.
2. Real-world applications in sectors like engineering, finance, and physics often present differential equations that resist simple analytical solutions.
3. The adaptability of established finite difference methods allows them to address a myriad of computational challenges.
4. These methods provide a means to visualize and comprehend the intricate dynamics of various problems, especially when analytical strategies fall short.
5. In the modern era, computational efficiencies have been greatly enhanced, making numerical approximations more feasible and faster.

1.2 Python in Numerical Solutions

Python, with its array of functionalities, has surfaced as an essential tool in the domain of scientific computations. Its emergence in the landscape of differential equations solution strategies can be attributed to:

1. Its open-source character and universal accessibility.
2. A wealth of specialized libraries, notably NumPy, SciPy, and SymPy, tailored for computational tasks.
3. The adaptability of Python, which transcends being just a numerical tool, bridging other domains like data visualization and web applications.
4. The innate simplicity and readability of Python's syntax.
5. Python's ability to interface seamlessly with robust languages, allowing for efficient and optimized routines.

* Corresponding author e-mail: youssri@cu.edu.eg

6.A proactive and evolving community ensuring consistent upgrades and support.
7.Visualization capabilities through libraries that grant insights into solutions of differential equations.
8.The inherent scalability of Python, backed by tools that amplify its performance.

1.3 Concluding Remarks

Embracing Python, with its expansive ecosystem, not only aids in resolving differential equations but also fosters an integrated approach—from problem inception, through solution, to subsequent analysis. With the continuous evolution of Python's capabilities, its centrality in differential equation solutions is poised to further amplify.

All handled examples in this review article are considered to satisfy the hypothesis of the following existence and uniqueness theorem, [1]

Theorem 1.*Suppose that* $D = \{(x,y) \mid a \le x \le b \text{ and } |y| < \infty\}$ *and that* $f(x,y)$ *is continuous on D. If f satisfies a Lipschitz condition on D in the variable y, then the initial-value problem*

$$y'(x) = f(x,y), \quad a \le x \le b, \quad y(a) = y_0,$$

has a unique solution $y(x)$ *for* $a \le x \le b$.

2 Methodology

In this section, we offer a concise summary of the numerical techniques utilized in our study. We highlight the common uses of each approach. Every numerical method mentioned was test with varied step sizes (The step size sometimes was mentioned as h). Furthermore, we have included an algorithm to each method, adding to this, the code for each method, accompanied by sample usage.

2.1 Adam's Bashforth Fourth Order

Adam's Bashforth method stands out as an integral multi-step technique tailored for solving ordinary differential equations (ODEs). Characterized as an explicit approach, it harnesses both the present and preceding values to determine the solution for the subsequent time step, bypassing the requirement for any auxiliary equation-solving. The term "fourth order" sheds light on its accuracy caliber. In essence, this method leans on the data from the preceding four points to extrapolate the upcoming value. The foundational equation defining the fourth order Adam's Bashforth technique can be described as follows:

$$y_{i+1} = y_i + \frac{h}{24}\left[55f(t_i,y_i) - 59f(t_{i-1},y_{i-1}) + 37f(t_{i-2},y_{i-2}) - 9f(t_{i-3},y_{i-3})\right] \quad (1)$$

Implemented Algorithm

To approximate the solution of the initial-value problem

$$y' = f(t,y), \quad a \le t \le b, \quad y(a) = \alpha,$$

using two preceding values to predict the next value.

INPUT: endpoints a,b; integer N; initial conditions α_0, α_1.
OUTPUT: approximation w to y at the $(N+1)$ equally spaced numbers in the interval $[a,b]$.

1.Step 1: Set $h = \frac{b-a}{N}$; Set $t_0 = a$; Set $w_0 = \alpha_0$; Set $w_1 = \alpha_1$;
2.Step 2: For $i = 1,2,\ldots,N-1$ do Steps 3, 4.
3.Step 3: Compute the predictor $w_{i+1} = w_i + \frac{h}{2}[3f(t_i,w_i) - f(t_{i-1},w_{i-1})]$;
4.Step 4: Set $t_{i+1} = a + (i+1)h$;
5.Step 5: Correct the predictor by computing $w_{i+1} = w_i + \frac{h}{2}[f(t_{i+1},w_{i+1}) + f(t_i,w_i)]$;
6.Step 6: OUTPUT (t_{i+1},w_{i+1}).
7.Step 7: STOP.

Python Implementation

Below is the Python code that calculates the coefficients for the Adam's Bashforth method and subsequently utilizes these coefficients for solving differential equations:

```
import sympy as sp

def calculate_adams_bashforth_coefficients(n):
    """
    Computes the coefficients for the Adam's Bashforth method of order n.

    Parameters:
    n : int
        Order of the method.

    Returns:
    list
        List of coefficients for the method of order n.
    """
```

```
    x, h = sp.symbols('x h')
    coefficients = []

    # Calculate coefficients based on integrals
    for k in range(n):
        integrand = 1
        for j in range(n):
            if j != k:
                integrand *= (x - j * h) / (h * (k - j))
        coeffs.append(sp.integrate(integrand, (x, (n - 1) * h, n * h)))

    return coefficients[::-1]

def adams_bashforth_method(dydx, stepsize, y_values, x_values, N, coefficients):
    """
    Computes values of a function using the Adam's Bashforth method.

    Parameters:
    dydx : sympy expression
        Differential equation in terms of x and y(x).
    stepsize : float
        Step size for each iteration.
    y_values : list
        Initial y values.
    x_values : list
        Initial x values.
    N : int
        Number of iterations.
    coefficients : list
        Coefficients for the Adam's Bashforth method.

    Returns:
    list
        List of y values computed using the method.
    """
    y_values = y_values[::-1]
    x_values = x_values[::-1]

    h, x = sp.symbols('h x')
    y = sp.Function('y')(x)

    # Convert the dydx to a lambda function
    dydx_func = sp.lambdify((x, y), dydx, "numpy")

    # Pre-compute coefficients after substituting the stepsize
    coefficients = [coeff.subs(h, stepsize) for coeff in coefficients]

    # Compute y values iteratively using the given coefficients
    for _ in range(N):
        New_y = y_values[-1]  # Start with the last value in the list
        for i in range(len(coefficients)):
            New_y += coefficients[i] * dydx_func(x_values[-1-i], y_values[-1-i])

        y_values.append(float(New_y))
        x_values.append(x_values[-1] + stepsize)

    return y_values
```

Sample Usage: To utilize this code, start by calculating the coefficients for the desired order and then apply the Adam's Bashforth method:

```

# Initialization of symbolic variables
x = sp.symbols('x')
y = sp.Function('y')(x)

# Define the differential equation and other parameters
dydx = y - x**2 + 1
h = 0.2 # stepsize
y_values = [1.6489406, 1.2140877, 0.8292986, 0.5]   # Initial y values
x_values = [0.6 ,0.4, 0.2, 0]
n = 4   # Order of the method.
N = 7   # Number of iterations

# Call the functions
coefficients = calculate_adams_bashforth_coefficients(n)
y_values = adams_bashforth_method(dydx, h, y_values, x_values, N, coefficients)
```

2.2 Adam's Moulton Third Order

The Adam's Moulton method is an implicit multi-step method. This implies that the method computes the solution at the next time step while also taking into account the value at that step, necessitating the solution of an equation. The third-order Adam's Moulton method

leverages both the current and two preceding points for its calculations:

$$y_{i+1} = y_i + \frac{h}{24}\left[9f(t_{i+1},y_{i+1}) + 19f(t_i,y_i) - 5f(t_{i-1},y_{i-1}) + f(t_{i-2},y_{i-2})\right] \quad (2)$$

Implemented Algorithm

To correct the predicted y-values obtained by an explicit Adams method or other predictors for the solution of the initial-value problem

$$y' = f(t,y), \quad a \le t \le b, \quad y(a) = \alpha,$$

using the current and two preceding values of the function.

INPUT: endpoints a,b; integer N; initial values $\alpha_0,\alpha_1,\alpha_2$; function $f(t,y)$.
OUTPUT: approximation w to y at the $(N+1)$ equally spaced numbers in the interval $[a,b]$.

1. Step 1: Set $h = \frac{b-a}{N}$; Set $t_0 = a$; Set $w_0 = \alpha_0$; Set $w_1 = \alpha_1$; Set $w_2 = \alpha_2$;
2. Step 2: Compute w_3 using a predictor method, e.g., Adams-Bashforth.
3. Step 3: For $i = 2,3,\ldots,N-1$ do Steps 4, 5.
4. Step 4: Correct w_{i+1} using the formula

$$w_{i+1} = w_i + \frac{h}{24}\left[9f(t_{i+1},w_{i+1}) + 19f(t_i,w_i) - 5f(t_{i-1},w_{i-}\right.$$

5. Step 5: Set $t_{i+1} = a + (i+1)h$; OUTPUT (t_{i+1},w_{i+1}).
6. Step 6: STOP.

Python Implementation

Below is the Python implementation that provides functionalities for computing the coefficients for Adam's Moulton method and executing the Adam's Moulton method for a given differential equation:

```
1
2 from scipy.optimize import fsolve
3 import sympy as sp
4
5 def adams_moulton_method(dydx, x_values,
      y_values, h, N):
6     """
7     Perform one step of the 4th order
          Adams-Moulton method using SciPy'
          s fsolve on a sympy-defined ODE.
8
9     dydx: Sympy expression representing
          dy/dt.
10    x_values: List of the most recent x
          values. Must contain at least 3
          values, with the latest value
          last.
11    y_values: List of the most recent y
          values. Must contain at least 3
          values, with the latest value
          last.
12    h: Step size.
13    N: number of iterations
14
15    Returns the y values.
16    """
17
18    x = sp.symbols('x')
19    y = sp.Function('y')(x)
20
21    # Convert the sympy expression to a
          lambda function for evaluation
22    f_lambda = sp.lambdify((x, y), dydx,
          "numpy")
23
24    # Define the equation to be solved
25    def equation(y_next):
26        return y_next - y_values[0] - (h
              /24) * (
27            9*f_lambda(x_values[0]+h,
                  y_next) +
28            19*f_lambda(x_values[0],
                  y_values[0]) -
29            5*f_lambda(x_values[1],
                  y_values[1]) +
30            f_lambda(x_values[2],
                  y_values[2])
31        )
32
33
34    # Perform N iterations
35    for _ in range(N):
36        # Use fsolve to find y_next
37        y_next = fsolve(equation,
              y_values[0])[0]
38        y_values = [y_next] + y_values
39        x_values = [x_values[0] + h] +
              x_values
40
41    return y_values[::-1]
```

Sample Usage: To employ this code, begin by determining the coefficients for the chosen order and then implement the Adam's Moulton method:

```
1
2 # Initialization of symbolic variables
3 x = sp.symbols('x')
4 y = sp.Function('y')(x)
5
6 # Define the differential equation and
      other parameters
7 dydx = y - x**2 + 1
8 h = 0.2 # stepsize
9 y_values = [1.2140877, 0.8292986, 0.5]
      # Initial y values
```

```
x_values = [0.4, 0.2, 0]
N = 8    # Number of iterations

# Call the functions
y_values = adams_moulton_method(dydx, h,
    y_values, x_values, N)
```

2.3 Euler's Method

Euler's method serves as a foundational numerical technique for addressing first-order ordinary differential equations. This single-step approach estimates solutions by progressing incrementally along tangent lines. Its straightforward nature makes it a common introductory method for learners, though it might not be optimal for intricate scenarios. Typical applications include basic physics problems and introductory differential equations coursework. The defining equation for Euler's approach is:

$$y_{i+1} = y_i + hf(t_i, y_i) \tag{3}$$

Implemented Algorithm

To approximate the solution of the initial-value problem

$$y' = f(t,y), \quad a \le t \le b, \quad y(a) = \alpha,$$

at $(N+1)$ equally spaced numbers in the interval $[a,b]$:

INPUT: endpoints a,b; integer N; initial condition α.
OUTPUT: approximation w to y at the $(N+1)$ values of t.

1. Step 1: Set $h = \frac{b-a}{N}$; Set $t = a$; Set $w = \alpha$; OUTPUT (t,w).
2. Step 2: For $i = 1,2,\ldots,N$ do Steps 3, 4.
3. Step 3: Set $w = w + h \cdot f(t,w)$; // Compute w_i
4. Step 4: Set $t = a + i \cdot h$; // Compute t_i OUTPUT (t,w).
5. Step 5: STOP.

Python Implementation

To implement Euler's method, we've constructed a Python function that approximates the values of a function based on the provided differential equation, step size, and initial conditions. Here is the implementation:

```

import sympy as sp

def Eulers_method(dydx, h, y_0, x_0, N):
    """
    Approximates function values using
        Euler's method.

    Parameters:
    dydx : sympy expression
        The differential equation in
            terms of x and y(x).
    h : float
        Step size for the iteration.
    y_0 : float
        Initial value of y.
    x_0 : float
        Initial value of x.
    N : int
        Number of iterations.

    Returns:
    list
        List of y values approximated
            using Euler's method.
    """

    # Define symbolic variables
    x = sp.symbols('x')
    y = sp.Function('y')(x)

    # Initialize lists for values of x
        and y
    y_values = [y_0]
    x_values = [x_0]

    # Apply Euler's method iteratively to
        compute y values
    for i in range(N):
        y_values.insert(0, float(y_values
            [0] + h*dydx.subs({x:
            x_values[0], y: y_values[0]})
            ))
        x_values.insert(0, x_values[0]+h)

    return y_values[::-1]
```

Sample Usage: This code sample shows how to implement the function mentioned above:

```

# Initialization of symbolic variables
x = sp.symbols('x')
y = sp.Function('y')(x)

# Define the differential equation and
    other parameters
dydx = y - x**2 + 1
h = 0.2 # stepsize
y_0 = 0.5    # Initial y values
x_0 = 0
N = 11    # Number of iterations

# Call the function
Eulers_method(dydx, h, y_0, x_0, N)
```

2.4 Fourth Order Runge Kutta Method

Runge-Kutta methods are a collection of iterative approaches tailored for resolving ordinary differential equations (ODEs). Specifically, the Fourth Order Runge-Kutta method is renowned for its harmonious blend of computational efficiency and precision. This is realized by estimating the solution through an average of four distinct evaluations, each progressively spaced within a singular step size. The foundational equations for this strategy are:

$$\begin{aligned} k_1 &= hf(t_i, y_i) \\ k_2 &= hf(t_i + \frac{h}{2}, y_i + \frac{k_1}{2}) \\ k_3 &= hf(t_i + \frac{h}{2}, y_i + \frac{k_2}{2}) \\ k_4 &= hf(t_i + h, y_i + k_3) \\ y_{i+1} &= y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

Implemented Algorithm

To approximate the solution of the initial-value problem

$$y' = f(t,y), \quad a \le t \le b, \quad y(a) = \alpha,$$

which is a popular method due to its accuracy and relatively simple implementation.

INPUT: endpoints a, b; integer N; initial condition α.
OUTPUT: approximation w to y at the $(N+1)$ equally spaced numbers in the interval $[a,b]$.

1.Step 1: Set $h = \frac{b-a}{N}$; Set $t = a$; Set $w = \alpha$;
2.Step 2: For $i = 1, 2, \ldots, N$ do Steps 3-6.
3.Step 3: Compute $k_1 = h \cdot f(t,w)$;
4.Step 4: Compute $k_2 = h \cdot f\left(t + \frac{h}{2}, w + \frac{k_1}{2}\right)$;
5.Step 5: Compute $k_3 = h \cdot f\left(t + \frac{h}{2}, w + \frac{k_2}{2}\right)$;
6.Step 6: Compute $k_4 = h \cdot f(t + h, w + k_3)$;
7.Step 7: Set $w = w + \frac{k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4}{6}$;
8.Step 8: Set $t = a + i \cdot h$; OUTPUT (t,w).
9.Step 9: STOP.

Python Implementation

We've crafted a Python function that employs the Fourth Order Runge Kutta technique to approximate function values. The code is provided below:

```
1
2 import sympy as sp
3
```

```
4 def fourth_order_Runge_Kutta_method(dydx,
      h, y_0, x_0, N):
5     """
6     Computes function values using the
          Fourth Order Runge Kutta method.
7
8     Parameters:
9     dydx : sympy expression
10        Differential equation in terms of
              x and y.
11    h : float
12        Step size.
13    y_0 : float
14        Initial y value.
15    x_0 : float
16        Initial x value.
17    N : int
18        Number of iterations.
19
20    Returns:
21    list
22        List of y values computed using
              the method.
23    """
24
25    # Initialize symbolic variables
26    x = sp.symbols('x')
27    y = sp.Function('y')(x)  # Define y
          as a function of x
28
29    # Convert dydx to a function for
          evaluations (instead of using
          lambdify)
30    f = sp.lambdify((x, y), dydx, "numpy"
          )
31
32    y_values = [y_0]
33    y_i = y_0
34    x_i = x_0
35
36    # Iterate using the Runge Kutta
          Fourth Order method
37    for i in range(N):
38        k_1 = h * f(x_i, y_i)
39        k_2 = h * f(x_i + h/2, y_i + k_1
              /2)
40        k_3 = h * f(x_i + h/2, y_i + k_2
              /2)
41        k_4 = h * f(x_i + h, y_i + k_3)
42
43        y_i += 1/6 * (k_1 + 2*k_2 + 2*k_3
              + k_4)
44        x_i += h
45        y_values.append(y_i)
46
47    return y_values
```

Sample Usage: For employing this function on a specific differential equation, say $y' = y - x**2 + 1$, with a given

step size, initial values, and number of iterations, you would call:

```
# Initialization of symbolic variables
x = sp.symbols('x')
y = sp.Function('y')(x)

# Define the differential equation and other parameters
dydx = y - x**2 + 1
h = 0.2 # stepsize
y_0 = 0.5 # Initial y values
x_0 = 0
N = 10   # Number of iterations

# Call the functions
fourth_order_Runge_Kutta_method(dydx, h, y_0, x_0, N)
```

2.5 Simpson's Method

Simpson's method serves as an integral estimation tool in numerical computation. This method shines when targeting polynomial functions, especially quadratics, to approximate definite integrals. By employing parabolic segments, it refines the integrand's representation. Typical applications include engineering simulations and advanced mathematical modeling. The foundational equation for this technique is as follows:

$$y_{i+1} = y_{i-1} + \frac{h}{3}[f(x_{i+1},y_{i+1}) + 4f(x_i,y_i) + f(x_{i-1},y_{i-1})] \quad (4)$$

Implemented Algorithm

To approximate the solution of the initial-value problem

$$y' = f(x,y), \quad a \le x \le b, \quad y(a) = \alpha,$$

using a numerical method based on Simpson's rule for integration.

INPUT: endpoints a,b; integer N; initial condition α.
OUTPUT: approximation y to the solution at the $(N+1)$ equally spaced numbers in the interval $[a,b]$.

1. Step 1: Set $h = \frac{b-a}{N}$; ensure that N is even. Set $x_0 = a$; Set $y_0 = \alpha$;
2. Step 2: If N is odd, increment N by 1 and recalculate h.
3. Step 3: For $i = 1,2,\ldots,N-1$, calculate y_i using an initial method (e.g., Runge-Kutta) to start the process.
4. Step 4: For $i = 2,4,\ldots,N-2$ do Steps 5, 6.
5. Step 5: Apply Simpson's Method to find y_{i+1}:
$y_{i+1} = y_{i-1} + \frac{h}{3}[f(x_{i+1},y_{i+1}) + 4f(x_i,y_i) + f(x_{i-1},y_{i-1})]$;
6. Step 6: Set $x_{i+1} = a + (i+1)h$; OUTPUT (x_{i+1},y_{i+1}).
7. Step 7: STOP.

Python Implementation

To implement Simpson's method, we've devised a Python function that computes the values of a function using the Simpson's formula. The provided code for this function is:

```
from scipy.optimize import fsolve
import sympy as sp

def Simpsons_method(dydx, stepsize, y_values, x_values, N):
    """
    Computes values of a function using Simpson's method with SciPy's fsolve.

    Parameters:
    dydx : sympy expression
        The differential equation in terms of x and y(x).
    stepsize : float
        Step size for the iteration.
    y_values : list
        Initial list of y values.
    x_values : list
        Initial list of x values.
    N : int
        Number of iterations.

    Returns:
    list
        List of y values computed using Simpson's method.
    """

    x = sp.symbols('x')
    y = sp.Function('y')(x)

    # Convert the sympy expression to a lambda function for evaluation
    f_lambda = sp.lambdify((x, y), dydx, "numpy")

    # Define the equation to be solved using Simpson's method
    def equation(y_next):
        return y_next - y_values[1] - stepsize/3 * (f_lambda(x_values[0]+stepsize, y_next) +
                        4*f_lambda(x_values[0], y_values[0]) +
```

```
                        f_lambda(x_values
                            [1], y_values
                            [1]))

    # Perform N iterations
    for _ in range(N):
        # Use fsolve to find y_next
        y_next_value = fsolve(equation,
            y_values[0])[0]
        y_values = [y_next_value] +
            y_values
        x_values = [x_values[0] +
            stepsize] + x_values

    return y_values[::-1]
```

Sample Usage: To utilize this function for a particular differential equation, for instance $y' = y - x^2 + 1$, given a specific step size, initial values, and iteration count, you would invoke:

```

# Initialization of symbolic variables
x = sp.symbols('x')
y = sp.Function('y')(x)

dydx = y - x**2 + 1

# Set the initial conditions and
    parameters
x_values = [0.2, 0]
y_values = [0.8292986, 0.5]
h = 0.2
N = 9

Simpsons_method(dydx, h, y_values,
    x_values, N)
```

2.6 Trapezoidal rule

The Trapezoidal Rule is a numerical method tailored for solving ordinary differential equations (ODEs). By employing the principle of averaging the derivative at the current and subsequent points, it delivers precise approximations integral to various applications. Within this framework, $f(x_i, y_i)$ designates the derivative at point i, h symbolizes the step size, and y_{i+1} provides the estimation for the next value. This technique is invaluable in fields like engineering, physics, and finance, aiding in system modeling, dynamic simulations, and risk assessments. Central to its implementation is the equation:

$$y_{i+1} = y_i + \frac{h}{2}\left(f(x_i, y_i) + f(x_{i+1}, y_{i+1})\right) \tag{5}$$

Implemented Algorithm

To approximate the solution of the initial-value problem using the trapezoidal rule, which is a numerical method based on the trapezoidal approximation of the integral.

Given the differential equation

$$y' = f(x,y), \quad x_0 \le x \le x_n, \quad y(x_0) = y_0,$$

the trapezoidal rule is applied to the integral form of the differential equation to find y at the next point x_{i+1}.

INPUT: endpoints x_0, x_n; step size h; initial condition y_0; function $f(x,y)$.
OUTPUT: approximation $y_1, y_2, \ldots, y_n$ to y at the n equally spaced numbers in the interval $[x_0, x_n]$.

1. Step 1: Set $i = 0$.
2. Step 2: While $i < n$ do Steps 3-5:
3. Step 3: Calculate the provisional value of y_{i+1} using Euler's method:

$$y^*_{i+1} = y_i + hf(x_i, y_i).$$

4. Step 4: Calculate y_{i+1} using the trapezoidal rule:

$$y_{i+1} = y_i + \frac{h}{2}[f(x_i, y_i) + f(x_{i+1}, y^*_{i+1})].$$

5. Step 5: Increment i by 1.
6. Step 6: OUTPUT the computed values of $y_1, y_2, \ldots, y_n$.
7. Step 7: STOP.

Python Implementation

To numerically solve differential equations using the Trapezoidal rule, we have developed a Python function named `trapezoidal_rule_method`. The code for this implementation is as follows:

```

from scipy.optimize import fsolve
import sympy as sp

def trapezoidal_rule_method(dydx, h,
    y_values, x_values, N):
    """
    Implements the Trapezoidal rule using
        SciPy's fsolve to solve
       differential equations.

    Parameters:
    dydx : sympy expression
        The differential equation in
            terms of x and y(x).
    h : float
        Step size for the iteration.
    y_values : list
        List of initial y values.
    x_values : list
        List of initial x values.
    N : int
```

```
        Number of iterations.

    Returns:
    list
        List of y values computed using
            the Trapezoidal rule.
    """

    x = sp.symbols('x')
    y = sp.Function('y')(x)

    # Convert the sympy expression to a
        lambda function for evaluation
    f_lambda = sp.lambdify((x, y), dydx,
        "numpy")

    # Define the equation to be solved
        using the Trapezoidal rule
    def equation(y_new):
        return y_new - y_values[-1] - (h
            /2) * (f_lambda(x_values[-1],
             y_values[-1]) + f_lambda(
            x_values[-1] + h, y_new))

    # Implement the Trapezoidal rule
    for i in range(N):

        # Use fsolve to find the new y
            value
        y_new = fsolve(equation, y_values
            [-1])[0]

        y_values.append(y_new)
        x_values.append(x_values[-1] + h)

    return y_values
```

Sample Usage:To use the function for a specific differential equation, initial conditions, step size, and number of iterations, you would call the function with the appropriate parameters. An example will be:

```

# Define the differential equation and
    other parameters
dydx = y - x**2 + 1
x_values = [0]
y_values = [0.5]
h = 0.2
N = 10

# Call the function
trapezoidal_rule_method(dydx, h, y_values
    , x_values, N)
```

2.7 Taylor Series Third Order

The Taylor Series technique stands out as an effective strategy for estimating solutions to differential equations. Drawing upon the Taylor series up to a chosen order, this method yields polynomial approximations centered around a particular point. Notably, when applied to the third order, the Taylor series exhibits a distinct formulation. This approach finds its application in diverse areas like engineering, physics, and system modeling due to its precision in handling complex equations, the third-order Taylor series expansion is given by:

$$\begin{aligned} y(x) \approx y(a) + y'(a)(x-a) \\ + \frac{y''(a)}{2!}(x-a)^2 \\ + \frac{y'''(a)}{3!}(x-a)^3 \end{aligned} \tag{6}$$

Implemented Algorithm

To approximate the solution of the initial-value problem using the trapezoidal rule, which is a numerical method based on the trapezoidal approximation of the integral.

Given the differential equation

$$y' = f(x,y), \quad x_0 \le x \le x_n, \quad y(x_0) = y_0,$$

the trapezoidal rule is applied to the integral form of the differential equation to find y at the next point x_{i+1}.

INPUT: endpoints x_0, x_n; step size h; initial condition y_0; function $f(x,y)$.
OUTPUT: approximation $y_1, y_2, \ldots, y_n$ to y at the n equally spaced numbers in the interval $[x_0, x_n]$.

1.Step 1: Set $i = 0$.
2.Step 2: While $i < n$ do Steps 3-5:
3.Step 3: Calculate the provisional value of y_{i+1} using Euler's method:

$$y^*_{i+1} = y_i + hf(x_i, y_i).$$

4.Step 4: Calculate y_{i+1} using the trapezoidal rule:

$$y_{i+1} = y_i + \frac{h}{2}[f(x_i, y_i) + f(x_{i+1}, y^*_{i+1})].$$

5.Step 5: Increment i by 1.
6.Step 6: OUTPUT the computed values of $y_1, y_2, \ldots, y_n$.
7.Step 7: STOP.

Python Implementation

To implement this method, we've developed a Python function that computes the values of a function using the Taylor series expansion up to a specified order. Below is the code for this implementation:

```
import sympy as sp
import math

def taylor_series_for_order_n(dydx, x0, y0, h, n, N):
    """
    Computes values of a function using Taylor series expansion.

    Parameters:
    dydx : sympy expression
        The differential equation in terms of x and y(x).
    x0 : float
        Initial value of x.
    y0 : float
        Initial value of y.
    h : float
        Step size for the iteration.
    n : int
        Order of the Taylor series.
    N : int
        Number of iterations.

    Returns:
    list
        List of y values computed using the Taylor series.
    """

    # Initialize symbolic variables
    x = sp.symbols('x')
    y = sp.Function('y')(x)  # Define y as a function of x

    # Initialize values for the iteration
    y_values = [y0]
    current_x = x0
    current_y = y0

    # Start with the first order Taylor method
    Tylor_method = dydx

    # Construct the Taylor series expansion formula up to order n
    for i in range(2, n):
        dydx = dydx.diff(x).subs(sp.Derivative(y, x), dydx)
        Tylor_method += (h**(i-1)/math.factorial(i)) * dydx

    # Convert Tylor_method to a NumPy-based lambda function
    Tylor_method_func = sp.lambdify((x, y), Tylor_method, "numpy")

    # Compute the y values using the Taylor series formula
    for i in range(N):
        current_y += h * Tylor_method_func(current_x, current_y)
        current_x += h
        y_values.append(current_y)

    return y_values
```

Sample Usage: The code provided bellow can be used to implement the aforementioned function:

```
x = sp.symbols('x')
y = sp.Function('y')(x)  # define y as a function of x

# Define the differential equation
dydx = y - x**2 + 1

# Set the initial conditions and parameters
x0 = 0
y0 = 0.5
h = 0.2
n = 2
N = 10

# Call the function and print the results
y_values = taylor_series_third_order(dydx, x0, y0, h, n, N)
```

2.8 How did we get the True Values?

In our study, we employed the Mathematica software to derive the precise solution for the differential equation. Following this, we computed the exact values based on this solution. Notably, two equations lacked explicit solutions. For these, we treated the outcomes from Mathematica's NDSolve as the reference values. Subsequently, we performed a comparison of these values.

*Note 1.***Important Considerations for Code Implementations**:

– **Version Requirements**: To ensure the compatibility and functionality of the provided code implementations, the following versions have been utilized:
  – `numpy`: 1.24.2
  – `sympy`: 1.11.1
  – `scipy`: 1.10.1
  – `python`: 3.8.10 (tags/v3.8.10:3d8993a, May 3 2021, 11:48:03) [MSC v.1928 64 bit (AMD64)]

–**Error Calculation**: We calculated our findings' errors using the $|y_i - x|$ formula. Consider the error estimation for Euler's approach as shown in the following sections for the sake of clarity. It is crucial to note that this strategy was consistently used for each numerical method we looked at.

```
import sympy as sp
import numpy as np

x = sp.symbols('x')
y = sp.Function('y')(x)

dydx = y - x**2 + 1
h = 0.2
y_0 = 0.5
x_0 = 0
N = 11

True_values = [0.500000000000000,
    0.829298620919915,
    1.21408765117937, 1.64894059980475,
     2.12722953575377,
    2.64085908577048, 3.17994153863173,
     3.73240001657766,
    4.28348378780244, 4.81517626779353,
     5.30547195053467]

Eulers_method_results = Eulers_method(
    dydx, h, y_0, x_0, N)
Error_values_Eulers_method = np.abs(np
    .array(True_values) - np.array(
    Eulers_method_results))
```

–**Complex values in implicit methods**: At times, implicit numerical methods can generate complex results. We treated these outcomes as inaccuracies. To handle this, we enhanced our functions to identify and trigger an error whenever they encounter such values.

3 Test Problems

Example 1*Consider the following linear IVP:*

$$\frac{d}{dx}y(x) = -x^2 + y(x) + 1, \quad 0 \le x \le 2, \quad y(0) = 0.5 \qquad (7)$$

with the exact smooth solution

$$y(x) = 1.0x^2 + 2.0x - 0.5e^x + 1.0 \qquad (8)$$

Table 1: Results of numerical methods for the ODE (7), h = 0.1

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	0.5	-	0.5	-
0.2	0.8292986	0.8292986	-	0.8292986	-
0.4	1.2140877	1.2140898	2.1e-06	1.2140875	2e-07
0.6	1.6489406	1.6489488	8.2e-06	1.6489399	7e-07
0.8	2.1272295	2.1272461	1.66e-05	2.1272282	1.4e-06
1.0	2.6408591	2.6408875	2.84e-05	2.6408568	2.3e-06
1.2	3.1799415	3.1799861	4.46e-05	3.1799379	3.6e-06
1.4	3.7324	3.7324666	6.66e-05	3.7323946	5.4e-06
1.6	4.2834838	4.2835799	9.61e-05	4.2834759	7.9e-06
1.8	4.8151763	4.8153117	0.0001355	4.8151652	1.11e-05
2.0	5.305472	5.3056595	0.0001875	5.3054566	1.53e-05

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
0.5	-	0.5	-	0.5	-
0.814	0.0152986	0.8294875	0.0001889	0.8292983	3e-07
1.18154	0.0325477	1.214549	0.0004613	1.2140869	7e-07
1.5970634	0.0518772	1.6497857	0.0008451	1.6489394	1.2e-06
2.0538467	0.0733828	2.1286055	0.001376	2.1272278	1.7e-06
2.5437545	0.0971046	2.6429596	0.0021005	2.6408567	2.4e-06
3.056943	0.1229986	3.1830197	0.0030782	3.1799385	3.1e-06
3.581501	0.150899	3.7367857	0.0043856	3.7323961	3.9e-06
4.1030162	0.1804676	4.2896047	0.0061209	4.283479	4.8e-06
4.6040496	0.2111267	4.8235856	0.0084093	4.8151704	5.8e-06
5.0635	0.2419719	5.3168826	0.0114106	5.305465	7e-06

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
0.5	-	1.0150706	0.5150706
0.8292986	-	0.6574145	0.1718841
1.2140876	1e-07	0.6573684	0.5567192
1.6489404	2e-07	1.0149016	0.634039
2.1272292	4e-07	1.4252953	0.7019342
2.6408585	6e-07	1.8825352	0.7583238
3.1799407	9e-07	2.3792744	0.8006672
3.7323987	1.3e-06	2.9065374	0.8258626
4.2834819	1.9e-06	3.4533601	0.8301237
4.8151737	2.6e-06	4.0063484	0.8088278

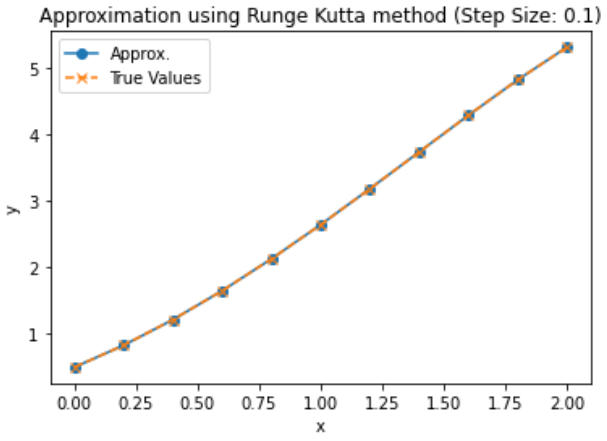

Fig. 1: The image shows the difference between the approximation of Runge Kutta method and the true values in **Test Problem 1**, with step size = 0.1

Table 2: Results of numerical methods for the ODE (7), h = 0.2

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	0.5	-	0.5	-
0.2	0.8292986	0.8292986	-	0.8292986	-
0.4	1.2140877	1.2140877	-	1.2140877	-
0.6	1.6489406	1.6489406	-	1.6489406	-
0.8	2.1272295	2.1273124	8.28e-05	2.1272217	7.9e-06
1.0	2.6408591	2.641081	0.0002219	2.6408396	1.95e-05
1.2	3.1799415	3.180348	0.0004065	3.1799057	3.58e-05
1.4	3.7324	3.7330601	0.0006601	3.7323416	5.84e-05
1.6	4.2834838	4.2844931	0.0010093	4.2833946	8.92e-05
1.8	4.8151763	4.8166575	0.0014812	4.8150454	0.0001309
2.0	5.305472	5.3075838	0.0021119	5.3052854	0.0001865

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
0.5	-	0.5	-	0.5	-
0.8	0.0292986	0.83	0.0007014	0.8292933	5.3e-06
1.152	0.0620877	1.2158	0.0017123	1.2140762	1.14e-05
1.5504	0.0985406	1.652076	0.0031354	1.648922	1.86e-05
1.98848	0.1387495	2.1323327	0.0051032	2.1272027	2.69e-05
2.458176	0.1826831	2.6486459	0.0077868	2.6408227	3.64e-05
2.9498112	0.2301303	3.191348	0.0114065	3.1798942	4.74e-05
3.4517734	0.2806266	3.7486446	0.0162446	3.7323401	5.99e-05
3.9501281	0.3333557	4.3061464	0.0226626	4.2834095	7.43e-05
4.4281538	0.3870225	4.8462986	0.0311223	4.8150857	9.06e-05
4.8657845	0.4396874	5.3476843	0.0422123	5.305363	0.0001089

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
0.5	-	1.6489406	1.1489406
0.8292986	-	1.2140877	0.384789
1.2140877	-	0.8292986	0.384789
1.6489406	-	0.5	1.1489406
2.1272261	3.5e-06	0.8288889	1.2983406
2.6408538	5.2e-06	1.2130864	1.4277727
3.1799309	1.07e-05	1.6471056	1.5328359
3.7323846	1.54e-05	2.1242402	1.6081598
4.2834595	2.43e-05	2.6362936	1.6471902
4.8151423	3.4e-05	3.1732477	1.6419286
5.3054229	4.9e-05	3.7228583	1.5826136

Table 3: Results of numerical methods for the ODE (9), h = 0.1

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	1	-	1	-
0.2	1.197365	1.197365	-	1.197365	-
0.4	1.3796528	1.3796191	3.38e-05	1.3796549	2.1e-06
0.6	1.5351535	1.53508	7.35e-05	1.5351574	3.8e-06
0.8	1.6576699	1.6575999	6.99e-05	1.6576723	2.4e-06
1.0	1.7468241	1.7467915	3.26e-05	1.7468231	1e-06
1.2	1.8067448	1.8067624	1.76e-05	1.80674	4.8e-06
1.4	1.8439407	1.8440016	6.09e-05	1.8439331	7.6e-06
1.6	1.8652662	1.8653529	8.66e-05	1.8652572	9e-06
1.8	1.8765586	1.8766531	9.44e-05	1.8765495	9.2e-06
2.0	1.8820814	1.8821716	9.03e-05	1.8820728	8.6e-06

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
1	-	1	-	1	-
1.199005	0.00164	1.1980149	0.0006499	1.1973651	1e-07
1.386477	0.0068242	1.3808236	0.0011708	1.379653	1e-07
1.5495715	0.014418	1.5366155	0.001462	1.5351537	1e-07
1.6806018	0.0229319	1.6591713	0.0015015	1.65767	1e-07
1.7778168	0.0309927	1.7481643	0.0013402	1.7468242	1e-07
1.8444245	0.0376797	1.807813	0.0010683	1.8067448	-
1.8865692	0.0426285	1.8447159	0.0007752	1.8439407	-
1.911195	0.0459288	1.8657886	0.0005224	1.8652662	-
1.9244831	0.0479244	1.876895	0.0003364	1.8765586	-
1.9311047	0.0490233	1.8822977	0.0002163	1.8820814	-

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
1	-	1.2912379	0.2912379
1.197365	-	1.0996677	0.0976974
1.3796536	8e-07	1.0995025	0.2801503
1.5351544	9e-07	1.2907805	0.244373
1.6576702	4e-07	1.4606315	0.1970384
1.7468237	4e-07	1.5999705	0.1468537
1.8067436	1.2e-06	1.7055739	0.1011708
1.843939	1.7e-06	1.7795146	0.0644261
1.8652644	1.9e-06	1.8273432	0.037923
1.8765568	1.8e-06	1.855925	0.0206336
1.8820797	1.7e-06	1.8717043	0.0103771

Example 2*Consider the following linear IVP:*

$$\frac{d}{dx}y(x) = e^{-x^2}, \quad 0 \le x \le 2, \quad y(0) = 1 \tag{9}$$

with the exact smooth solution

$$y(x) = 0.5\sqrt{\pi}\,\mathrm{erf}\,(x) + 1.0 \tag{10}$$

Table 4: Results of numerical methods for the ODE (9), h = 0.2

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	1	-	1	-
0.2	1.197365	1.197365	-	1.197365	-
0.4	1.3796528	1.3796528	-	1.3796528	-
0.6	1.5351535	1.5351535	-	1.5351535	-
0.8	1.6576699	1.6571946	0.0004753	1.6576665	3.4e-06
1.0	1.7468241	1.7465312	0.0002929	1.7467767	4.74e-05
1.2	1.8067448	1.8070966	0.0003518	1.8066374	0.0001074
1.4	1.8439407	1.8450705	0.0011298	1.8437811	0.0001596
1.6	1.8652662	1.8670237	0.0017575	1.8650752	0.000191
1.8	1.8765586	1.8786614	0.0021028	1.8763577	0.0002009
2.0	1.8820814	1.8842651	0.0021837	1.8818851	0.0001963

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
1	-	1	-	1	-
1.2	0.002635	1.2	0.002635	1.1973663	1.3e-06
1.3921579	0.012505	1.3844716	0.0048187	1.3796549	2.1e-06
1.5625866	0.0274331	1.541266	0.0061125	1.5351557	2.1e-06
1.7021219	0.0444521	1.6640571	0.0063872	1.6576715	1.6e-06
1.8075804	0.0607563	1.7526422	0.0058181	1.7468249	8e-07
1.8811563	0.0744115	1.8115029	0.0047581	1.8067448	1e-07
1.9285418	0.0846011	1.8475159	0.0035752	1.8439403	4e-07
1.9567135	0.0914473	1.8677995	0.0025333	1.8652656	6e-07
1.9721745	0.0956158	1.878313	0.0017543	1.8765581	5e-07
1.9800072	0.0979259	1.883326	0.0012446	1.882081	4e-07

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
1	-	1.5351535	0.5351535
1.197365	-	1.3796528	0.1822878
1.3796528	-	1.197365	0.1822878
1.5351535	-	1	0.5351535
1.6576623	7.6e-06	1.1960789	0.4615909
1.7468019	2.22e-05	1.3773723	0.3694519
1.8067115	3.33e-05	1.5323543	0.2743905
1.8438985	4.22e-05	1.6548512	0.1890896
1.8652225	4.37e-05	1.7443683	0.1208979
1.8765146	4.41e-05	1.8048491	0.0717096
1.8820409	4.04e-05	1.8426277	0.0394537

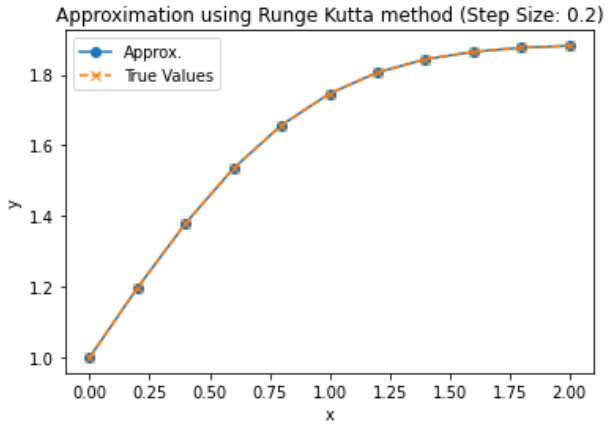

Fig. 2: The image shows the difference between the approximation of Runge Kutta method and the true values in **Test Problem 2**, with step size = 0.2

Example 3*Consider the following nonlinear IVP:*

$$\frac{d}{dx}y(x) = \sqrt{y^2(x)+1}, \quad 0 \le x \le 2, \quad y(0) = 1 \tag{11}$$

with the exact smooth solution

$$y(x) = \sinh\left(x + \log\left(1+\sqrt{2}\right)\right) \tag{12}$$

Table 5: Results of numerical methods for the ODE (11), h = 0.1

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	1	-	1	-
0.2	1.3047989	1.3047989	-	1.3047989	-
0.4	1.6619639	1.6619582	5.7e-06	1.6619644	5e-07
0.6	2.0858293	2.0858081	2.12e-05	2.0858311	1.7e-06
0.8	2.5934065	2.5933642	4.23e-05	2.5934099	3.4e-06
1.0	3.2050661	3.2049948	7.13e-05	3.2050719	5.8e-06
1.2	3.9453563	3.9452453	0.000111	3.9453653	9e-06
1.4	4.8439875	4.8438229	0.0001646	4.8440008	1.33e-05
1.6	5.9370249	5.9367885	0.0002364	5.9370441	1.92e-05
1.8	7.268336	7.268004	0.000332	7.2683629	2.7e-05
2.0	8.8913509	8.8908925	0.0004584	8.8913881	3.73e-05

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
1	-	1	-	1	-
1.2931726	0.0116263	1.3042811	0.0005178	1.3047989	-
1.6333306	0.0286333	1.6607456	0.0012183	1.6619638	-
2.0329328	0.0528966	2.0836552	0.0021742	2.0858292	2e-07
2.5065789	0.0868276	2.5899299	0.0034765	2.5934061	4e-07
3.0715525	0.1335136	3.1998254	0.0052407	3.2050653	8e-07
3.7484569	0.1968994	3.9377428	0.0076134	3.9453548	1.5e-06
4.5619695	0.282018	4.8332057	0.0107818	4.8439851	2.4e-06
5.5417438	0.3952811	5.9220405	0.0149844	5.9370211	3.7e-06
6.7234911	0.5448448	7.2478105	0.0205254	7.2683304	5.6e-06
8.1502831	0.7410678	8.8635586	0.0277923	8.8913427	8.1e-06

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
1	-	1.4759952	0.4759952
1.3047989	-	1.1466613	0.1581375
1.6619641	2e-07	1.1467883	0.5151756
2.0858298	5e-07	1.4764417	0.6093877
2.5934074	9e-07	1.8654491	0.7279574
3.2050675	1.4e-06	2.329449	0.8756171
3.9453585	2.2e-06	2.8870944	1.0582619
4.8439907	3.2e-06	3.5608033	1.2831842
5.9370294	4.5e-06	4.3776591	1.5593658
7.2683423	6.3e-06	5.3705001	1.8978359
8.8913596	8.7e-06	6.5792392	2.3121117

Table 6: Results of numerical methods for the ODE (11), h = 0.2

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	1	-	1	-
0.2	1.3047989	1.3047989	-	1.3047989	-
0.4	1.6619639	1.6619639	-	1.6619639	-
0.6	2.0858293	2.0858293	-	2.0858293	-
0.8	2.5934065	2.5931908	0.0002157	2.5934265	2e-05
1.0	3.2050661	3.2045011	0.000565	3.2051149	4.88e-05
1.2	3.9453563	3.9443341	0.0010221	3.9454451	8.88e-05
1.4	4.8439875	4.8423414	0.0016461	4.8441313	0.0001438
1.6	5.9370249	5.9345238	0.0025011	5.9372436	0.0002187
1.8	7.268336	7.2646828	0.0036532	7.2686557	0.0003197
2.0	8.8913509	8.8861602	0.0051907	8.8918055	0.0004547

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
1	-	1	-	1	-
1.2828427	0.0219561	1.3028427	0.0019561	1.3047989	-
1.608154	0.0538099	1.6573748	0.0045891	1.6619632	7e-07
1.9868971	0.0989322	2.07766	0.0081693	2.0858268	2.6e-06
2.4317683	0.1616381	2.5803714	0.0130351	2.5934001	6.3e-06
2.9576389	0.2474272	3.1854521	0.019614	3.2050535	1.26e-05
3.5820628	0.3632935	3.9169068	0.0284495	3.9453341	2.22e-05
4.3258685	0.518119	4.8037536	0.0402339	4.8439513	3.62e-05
5.2138581	0.7231668	5.8811757	0.0558492	5.9369688	5.61e-05
6.2756362	0.9926998	7.1919166	0.0764194	7.2682524	8.36e-05
7.5465981	1.3447527	8.7879762	0.1033747	8.8912298	0.0001211

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
1	-	2.0858293	1.0858293
1.3047989	-	1.6619639	0.357165
1.6619639	-	1.3047989	0.357165
2.0858293	-	1	1.0858293
2.5934153	8.8e-06	1.3059017	1.2875047
3.205079	1.29e-05	1.6645672	1.5404989
3.9453826	2.63e-05	2.0904878	1.8548684
4.8440249	3.75e-05	2.6008727	2.2431148
5.9370842	5.93e-05	3.2163433	2.7206816
7.2684184	8.24e-05	3.9617671	3.3065688
8.8914698	0.000119	4.8672624	4.0240885

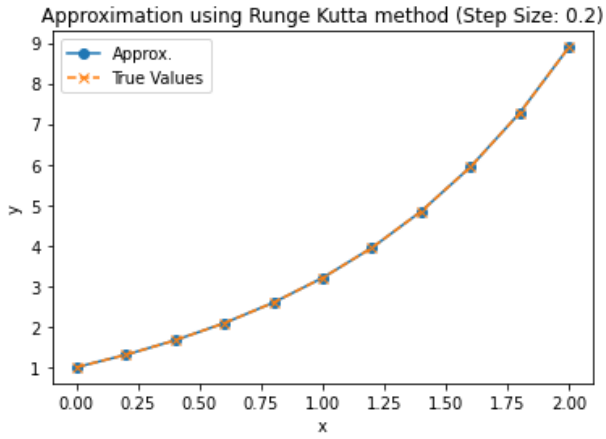

Fig. 3: The image shows the difference between the approximation of Runge Kutta method and the true values in **Test Problem 3**, with step size = 0.2

Example 4*Consider the following linear IVP:*

$$\frac{d}{dx}y(x) = -0.9906x\,(1000-x)\,, \qquad 0 \le x \le 2, \quad y(0) = 1 \tag{13}$$

with the exact smooth solution

$$y(x) = \frac{1651x^3}{5000} - \frac{4953x^2}{10} + 1 \tag{14}$$

Table 7: Results of numerical methods for the ODE (13), h = 0.1

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	1	-	1	-
0.2	-18.8094	-18.8094	-	-18.8094	-
0.4	-78.2269	-78.2269	-	-78.2269	-
0.6	-177.237	-177.237	-	-177.237	-
0.8	-315.823	-315.823	-	-315.823	-
1.0	-493.97	-493.97	-	-493.97	-
1.2	-711.661	-711.661	-	-711.661	-
1.4	-968.882	-968.882	-	-968.882	-
1.6	-1265.62	-1265.62	-	-1265.62	-
1.8	-1601.85	-1601.85	-	-1601.85	-
2.0	-1977.56	-1977.56	-	-1977.56	-

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
1	-	1	-	1	-
-8.90501	9.90435	-18.81	0.00066	-18.8094	-
-58.4221	19.8047	-78.2282	0.00132	-78.2269	-
-147.536	29.7012	-177.239	0.00198	-177.237	-
-276.229	39.5936	-315.826	0.00264	-315.823	-
-444.488	49.4821	-493.973	0.0033	-493.97	-
-652.295	59.3667	-711.665	0.00396	-711.661	-
-899.635	69.2472	-968.887	0.00462	-968.882	-
-1186.49	79.1238	-1265.62	0.00528	-1265.62	-
-1512.85	88.9965	-1601.85	0.00594	-1601.85	-
-1878.69	98.8652	-1977.57	0.0066	-1977.56	-

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
1	-	-43.5681	44.5681
-18.8094	-	-3.95267	14.8567
-78.2269	-	-3.9525	74.2744
-177.237	-	-43.5676	133.669
-315.823	-	-122.783	193.04
-493.97	-	-241.583	252.387
-711.661	-	-399.951	311.711
-968.882	-	-597.872	371.01
-1265.62	-	-835.329	430.286
-1601.85	-	-1112.31	489.538
-1977.56	-	-1428.79	548.766

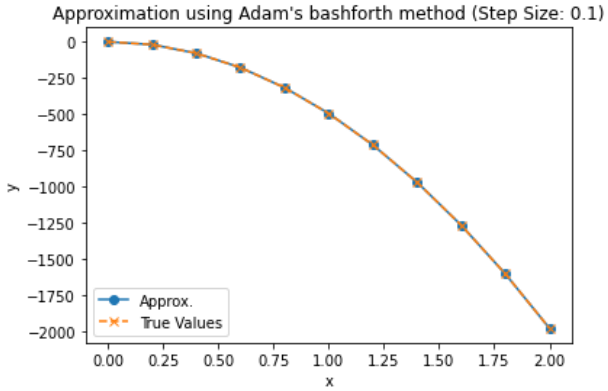

Fig. 4: The image shows the difference between the approximation of Adam's bashforth method and the true values in **Test Problem 4**, with step size = 0.1

Table 8: Results of numerical methods for the ODE (13), h = 0.2

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	1	-	1	-
0.2	-18.8094	-18.8094	-	-18.8094	-
0.4	-78.2269	-78.2269	-	-78.2269	-
0.6	-177.237	-177.237	-	-177.237	-
0.8	-315.823	-315.823	-	-315.823	-
1.0	-493.97	-493.97	-	-493.97	-
1.2	-711.661	-711.661	-	-711.661	-
1.4	-968.882	-968.882	-	-968.882	-
1.6	-1265.62	-1265.62	-	-1265.62	-
1.8	-1601.85	-1601.85	-	-1601.85	-
2.0	-1977.56	-1977.56	-	-1977.56	-

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
1	-	1	-	1	-
1.0	19.8094	-18.812	0.00264	-18.8094	-
-38.6161	39.6108	-78.2322	0.00528	-78.2269	-
-117.832	59.4043	-177.245	0.00792	-177.237	-
-236.633	79.1899	-315.834	0.01057	-315.823	-
-395.002	98.9675	-493.983	0.01321	-493.97	-
-592.924	118.737	-711.677	0.01585	-711.661	-
-830.383	138.499	-968.9	0.01849	-968.882	-
-1107.36	158.253	-1265.64	0.02113	-1265.62	-
-1423.85	177.999	-1601.87	0.02377	-1601.85	-
-1779.82	197.737	-1977.58	0.02642	-1977.56	-

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
1	-	-177.237	178.237
-18.8094	-	-78.2269	59.4175
-78.2269	-	-18.8094	59.4175
-177.237	-	1	178.237
-315.823	-	-18.808	297.015
-493.97	-	-78.2242	415.746
-711.661	-	-177.233	534.429
-968.882	-	-315.818	653.064
-1265.62	-	-493.963	771.652
-1601.85	-	-711.653	890.193
-1977.56	-	-968.873	1008.69

Example 5*Consider the following nonlinear IVP:*

$$\frac{d}{dx}y(x) = 0.2311 \cdot \left(1 - \frac{y(x)}{1072764}\right) y(x), \quad 0 \le x \le 2, \quad y(0) = 900000 \tag{15}$$

with the exact smooth solution

$$y(x) = \frac{1072760.0e^{0.2311x}}{e^{0.2311x} + 0.19196} \tag{16}$$

Table 9: Results of numerical methods for the ODE (15), h = 0.1

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	900000	-	900000	-
0.2	906591.31	906591.31	-	906591.31	-
0.4	912978.93	912979.02	0.0867185	912979.0	0.0662978
0.6	919162.62	919162.83	0.2166659	919162.81	0.1967182
0.8	925145.68	925146.02	0.3443415	925146.0	0.3246179
1.0	930931.64	930932.11	0.4694697	930932.09	0.449864
1.2	936524.21	936524.8	0.5918485	936524.79	0.5723817
1.4	941927.25	941927.96	0.7114087	941927.94	0.6921083
1.6	947144.75	947145.58	0.8280998	947145.56	0.8089934
1.8	952180.81	952181.76	0.9418827	952181.74	0.9229974
2.0	957039.64	957040.7	1.0527296	957040.68	1.0340913

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
900000	-	900000	-	900000	-
906646.46	55.151928	906594.57	3.2641656	906594.69	3.3804002
913083.5	104.56211	912982.1	3.1648682	912982.34	3.4042113
919314.16	151.54809	919165.67	3.059337	919166.04	3.4272622
925341.76	196.08556	925148.63	2.9489124	925149.13	3.4495652
931169.8	238.16325	930934.48	2.8348572	930935.11	3.4711335
936801.99	277.78201	936526.93	2.718354	936527.7	3.491981
942242.2	314.95385	941929.85	2.6005014	941930.76	3.5121219
947494.45	349.70102	947147.23	2.4823136	947148.28	3.5315713
952562.87	382.05503	952183.18	2.3647193	952184.36	3.5503445
957451.7	412.05576	957041.89	2.2485624	957043.21	3.568457

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
900000	-	909810.81	9810.8127
906591.31	-	903320.04	3271.2682
912979.07	0.1322536	903323.44	9655.4967
919162.88	0.2620676	909814.29	9348.3215
925146.07	0.3893416	916099.63	9046.0439
930932.15	0.5139887	922182.66	8748.9775
936524.85	0.6359347	928066.81	8457.4
941928.01	0.755118	933755.7	8171.554
947145.62	0.8714887	939253.1	7891.6493
952181.8	0.9850079	944562.95	7617.8638
957040.74	1.0956471	949689.3	7350.3456

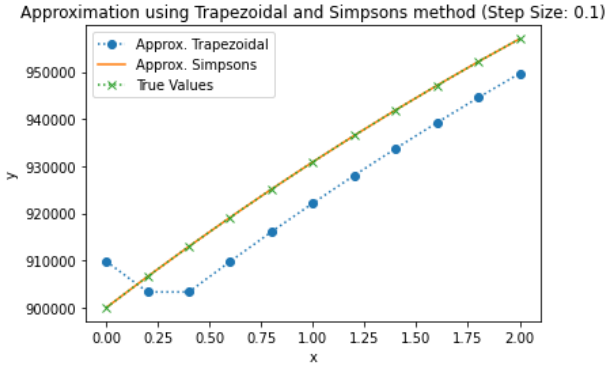

Fig. 5: The image shows the difference between the approximation of Simpsons method, Trapezoidal method and the true values in **Test Problem 5**, with step size = 0.1

Example 6*Consider the following nonlinear IVP:*

$$\frac{d}{dx}y(x) = e^{-xy(x)}, \quad 0 \le x \le 2, \quad y(0) = 1 \tag{17}$$

Eq. (17) *has no known exact solution.*

Table 10: Results of numerical methods for the ODE (15), h = 0.2

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	900000	-	900000	-
0.2	906591.31	906591.31	-	906591.31	-
0.4	912978.93	912978.93	-	912978.93	-
0.6	919162.62	919162.62	-	919162.62	-
0.8	925145.68	925145.86	0.1848023	925145.81	0.1342841
1.0	930931.64	930931.96	0.3174546	930931.91	0.2650864
1.2	936524.21	936524.67	0.4561679	936524.61	0.3932737
1.4	941927.25	941927.84	0.5866541	941927.77	0.518584
1.6	947144.75	947145.46	0.7148113	947145.39	0.6409652
1.8	952180.81	952181.65	0.8390015	952181.57	0.760374
2.0	957039.64	957040.6	0.9597044	957040.52	0.8767778

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
900000	-	900000	-	900000	-
906699.18	107.86986	906594.23	2.9173973	906594.69	3.3802997
913186.52	207.58383	912981.38	2.4496998	912982.34	3.404013
919465.03	302.41558	919164.57	1.9584087	919166.04	3.426969
925537.99	392.31158	925147.13	1.4489204	925149.13	3.4491803
931408.89	477.24513	930932.57	0.9263224	930935.11	3.4706603
937081.43	557.21444	936524.61	0.3953779	936527.7	3.4914228
942559.49	632.24073	941927.11	0.1394854	941930.76	3.5114825
947847.12	702.36631	947144.08	0.6741815	947148.28	3.5308543
952948.47	767.65267	952179.61	1.2049686	952184.36	3.5495537
957867.82	828.17852	957037.92	1.7284472	957043.21	3.5675963

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
900000	-	919162.62	19162.615
906591.31	-	912978.93	6387.6254
912978.93	-	906591.31	6387.6254
919162.62	-	900000	19162.615
925145.94	0.268175	906594.92	18550.755
930931.9	0.2598024	912982.82	17948.823
936524.74	0.5255493	919166.78	17357.433
941927.76	0.508329	925150.13	16777.123
947145.52	0.7715473	930936.38	16208.365
952181.56	0.7451025	936529.25	15651.563
957040.65	1.0057882	941932.58	15107.06

Table 11: Results of numerical methods for the ODE (17), h = 0.1

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	1	-	1	-
0.2	1.17917	1.17917	-	1.17917	-
0.4	1.31675	1.316816	6.6e-05	1.3167459	4.1e-06
0.6	1.41784	1.4180072	0.0001672	1.4178284	1.16e-05
0.8	1.49027	1.4904862	0.0002162	1.4902613	8.7e-06
1.0	1.54153	1.5417496	0.0002196	1.5415206	9.4e-06
1.2	1.57763	1.577834	0.000204	1.5776168	1.32e-05
1.4	1.60302	1.6032174	0.0001974	1.6030149	5.1e-06
1.6	1.62091	1.6210975	0.0001875	1.6209078	2.2e-06
1.8	1.63355	1.6337197	0.0001697	1.63354	1e-05
2.0	1.64248	1.6426515	0.0001715	1.6424791	9e-07

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
1	-	1	-	1	-
1.1895834	0.0104134	1.1793194	0.0001494	1.1791713	1.3e-06
1.3367606	0.0200106	1.3166839	6.61e-05	1.316746	4e-06
1.4451191	0.0272791	1.4174465	0.0003935	1.4178349	5.1e-06
1.5224474	0.0321774	1.4895834	0.0006866	1.4902701	1e-07
1.5767689	0.0352389	1.5406173	0.0009127	1.5415293	7e-07
1.6146862	0.0370562	1.5765576	0.0010724	1.5776249	5.1e-06
1.6411201	0.0381001	1.6018528	0.0011672	1.6030221	2.1e-06
1.659572	0.038662	1.6196798	0.0012302	1.6209143	4.3e-06
1.6724819	0.0389319	1.6322704	0.0012796	1.633546	4e-06
1.681538	0.039058	1.6411836	0.0012964	1.6424848	4.8e-06

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
1	-	1.25296	0.25296
1.17917	-	1.09487	0.0843
1.3167427	7.3e-06	1.0948149	0.2219351
1.4178306	9.4e-06	1.2529558	0.1648842
1.4902655	4.5e-06	1.3715285	0.1187415
1.5415247	5.3e-06	1.4574453	0.0840847
1.5776204	9.6e-06	1.5185642	0.0590658
1.6030177	2.3e-06	1.5616791	0.0413409
1.62091	-	1.5920131	0.0288969
1.6335417	8.3e-06	1.6133627	0.0201873
1.6424805	5e-07	1.6284156	0.0140644

Table 12: Results of numerical methods for the ODE (17), h = 0.2

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	1	-	1	-
0.2	1.17917	1.17917	-	1.17917	-
0.4	1.31675	1.31675	-	1.31675	-
0.6	1.41784	1.41784	-	1.41784	-
0.8	1.49027	1.4918027	0.0015327	1.4902099	6.01e-05
1.0	1.54153	1.5436036	0.0020736	1.541454	7.6e-05
1.2	1.57763	1.5798489	0.0022189	1.5775549	7.51e-05
1.4	1.60302	1.6051092	0.0020892	1.6029629	5.71e-05
1.6	1.62091	1.6228338	0.0019238	1.620865	4.5e-05
1.8	1.63355	1.6353145	0.0017645	1.6335045	4.55e-05
2.0	1.64248	1.6441398	0.0016598	1.6424488	3.12e-05

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
1	-	1	-	1	-
1.2	0.02083	1.18	0.00083	1.1791696	4e-07
1.3573256	0.0405756	1.3168223	7.23e-05	1.3167429	7.1e-06
1.4735342	0.0556942	1.4165866	0.0012534	1.4178313	8.7e-06
1.5561497	0.0658797	1.4877713	0.0024987	1.4902664	3.6e-06
1.6137425	0.0722125	1.5380719	0.0034581	1.5415257	4.3e-06
1.6535707	0.0759407	1.5735007	0.0041293	1.5776214	8.6e-06
1.6810664	0.0780464	1.598457	0.004563	1.6030188	1.2e-06
1.7000738	0.0791638	1.6160653	0.0048447	1.6209111	1.1e-06
1.7132472	0.0796972	1.6285171	0.0050329	1.6335429	7.1e-06
1.7224037	0.0799237	1.637343	0.005137	1.6424817	1.7e-06

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
1	-	1.41784	0.41784
1.17917	-	1.31675	0.13758
1.31675	-	1.17917	0.13758
1.41784	-	1	0.41784
1.4902539	1.61e-05	1.178994	0.311276
1.5415316	1.6e-06	1.3170362	0.2244938
1.5776115	1.85e-05	1.418772	0.158858
1.6030288	8.8e-06	1.4917778	0.1112422
1.6209043	5.7e-06	1.54346	0.07745
1.6335555	5.5e-06	1.5798437	0.0537063
1.6424765	3.5e-06	1.6054286	0.0370514

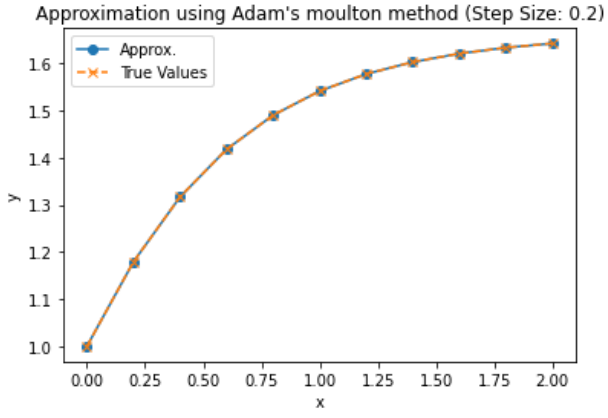

Fig. 6: The image shows the difference between the approximation of Adam's Moulton method and the true values in **Test Problem 6**, with step size = 0.2

Example 7*Consider the following nonlinear IVP:*

$$\frac{d}{dx}y(x) = x\cos(y(x)), \quad 0 \le x \le 2, \quad y(0) = 1 \tag{18}$$

with the exact smooth solution

$$y(x) = 2\arctan\left(\tanh\left(0.25x^2 + 0.613095585441758\right)\right) \tag{19}$$

Table 13: Results of numerical methods for the ODE (18), h = 0.1

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	1	-	1	-
0.2	1.0107154	1.0107154	-	1.0107154	-
0.4	1.041789	1.0417866	2.5e-06	1.0417893	3e-07
0.6	1.0901204	1.090109	1.13e-05	1.0901215	1.1e-06
0.8	1.150958	1.150933	2.5e-05	1.1509603	2.3e-06
1.0	1.218562	1.2185221	3.99e-05	1.2185654	3.4e-06
1.2	1.2870809	1.2870311	4.98e-05	1.2870848	3.9e-06
1.4	1.3514416	1.3513917	4.99e-05	1.3514452	3.7e-06
1.6	1.4080001	1.4079602	3.99e-05	1.4080028	2.7e-06
1.8	1.4547965	1.4547723	2.41e-05	1.4547979	1.5e-06
2.0	1.4914211	1.4914127	8.4e-06	1.4914215	3e-07

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
1	-	1	-	1	-
1.005403	0.0053124	1.0107492	3.38e-05	1.0107154	-
1.0319181	0.009871	1.0419402	0.0001511	1.041789	-
1.0772175	0.0129029	1.0904476	0.0003272	1.0901203	-
1.1370434	0.0139146	1.1514779	0.0005199	1.150958	-
1.2057242	0.0128378	1.2192398	0.0006779	1.2185619	1e-07
1.2770051	0.0100758	1.2878383	0.0007574	1.2870808	1e-07
1.3450551	0.0063865	1.3521802	0.0007386	1.3514414	2e-07
1.4053686	0.0026315	1.4086323	0.0006322	1.4079997	4e-07
1.4552839	0.0004874	1.4552677	0.0004712	1.4547959	6e-07
1.4940086	0.0025875	1.4917174	0.0002963	1.4914203	8e-07

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
1	-	1.0238567	0.0238567
1.0107154	-	1.0026958	0.0080196
1.0417892	1e-07	1.0026902	0.0390989
1.0901207	3e-07	1.0238076	0.0663128
1.1509586	6e-07	1.0639352	0.0870228
1.2185628	8e-07	1.1191003	0.0994617
1.2870818	9e-07	1.1839818	0.1030991
1.3514423	8e-07	1.2526912	0.0987504
1.4080006	5e-07	1.3196787	0.0883214
1.4547967	3e-07	1.380527	0.0742694
1.4914212	1e-07	1.4324158	0.0590053

Table 14: Results of numerical methods for the ODE (18), h = 0.2

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	1	-	1	-
0.2	1.0107154	1.0107154	-	1.0107154	-
0.4	1.041789	1.041789	-	1.041789	-
0.6	1.0901204	1.0901204	-	1.0901204	-
0.8	1.150958	1.1507821	0.0001759	1.1509762	1.82e-05
1.0	1.218562	1.218172	0.00039	1.2185999	3.79e-05
1.2	1.2870809	1.2864717	0.0006092	1.2871327	5.18e-05
1.4	1.3514416	1.3507259	0.0007156	1.3514953	5.38e-05
1.6	1.4080001	1.4073204	0.0006797	1.4080435	4.34e-05
1.8	1.4547965	1.4542938	0.0005027	1.4548223	2.59e-05
2.0	1.4914211	1.4911484	0.0002728	1.4914292	8e-06

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
1	-	1	-	1	-
1.0	0.0107154	1.010806	9.06e-05	1.0107154	-
1.0216121	0.0201769	1.0423167	0.0005276	1.0417889	2e-07
1.0633714	0.0267489	1.0913454	0.001225	1.0901201	3e-07
1.1216828	0.0292752	1.1529795	0.0020215	1.1509576	4e-07
1.1911495	0.0274124	1.2212703	0.0027084	1.2185613	7e-07
1.265268	0.0218129	1.290175	0.0030941	1.2870793	1.6e-06
1.3374593	0.0139823	1.3545184	0.0030769	1.3514381	3.4e-06
1.4022024	0.0057977	1.410681	0.0026809	1.4079938	6.3e-06
1.4558973	0.0011008	1.4568294	0.0020329	1.4547866	9.9e-06
1.49717	0.0057489	1.4927219	0.0013007	1.4914075	1.36e-05

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
1	-	1.0901204	0.0901204
1.0107154	-	1.041789	0.0310736
1.041789	-	1.0107154	0.0310736
1.0901204	-	1	0.0901204
1.1509666	8.6e-06	1.0106266	0.1403314
1.2185704	8.4e-06	1.0414519	0.1771101
1.2870952	1.43e-05	1.08943	0.1976508
1.3514497	8.1e-06	1.1498961	0.2015455
1.4080097	9.6e-06	1.2172092	0.1907909
1.4547963	1e-07	1.2855975	0.169199
1.4914241	3e-06	1.3500178	0.1414033

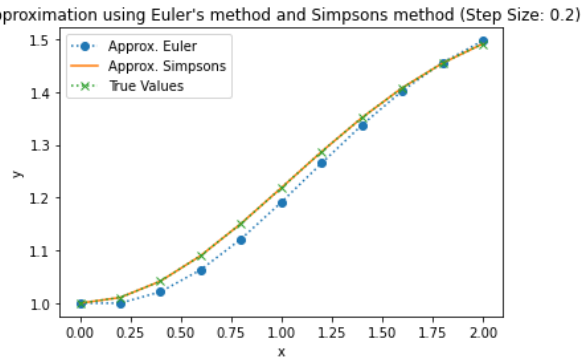

Fig. 7: The image shows the difference between the approximation of Simpsons method, Euler's method and the true values in **Test Problem 7**, with step size = 0.2

Example 8*Consider the following nonlinear IVP:*

$$\frac{d}{dx}y(x) = x^2 + y^2(x), \quad 0 \le x \le 2, \quad y(0) = 0 \tag{20}$$

Eq. (20) *has no known exact solution.*

Table 15: Results of numerical methods for the ODE (20), h = 0.1

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	-	-	-	-
0.2	0.0026669	0.0026669	-	0.0026669	-
0.4	0.0213595	0.0213537	5.8e-06	0.0213603	8e-07
0.6	0.072448	0.0724061	4.19e-05	0.0724529	4.9e-06
0.8	0.17408	0.1739369	0.0001431	0.1740961	1.61e-05
1.0	0.350232	0.3498199	0.0004121	0.3502769	4.49e-05
1.2	0.641077	0.6398395	0.0012375	0.6412186	0.0001416
1.4	1.13311	1.1285565	0.0045535	1.1336934	0.0005834
1.6	2.07642	2.0516962	0.0247238	2.0803417	0.0039217
1.8	4.68814	4.3996518	0.2884882	4.7741735	0.0860335
2.0	317.746	17.9352968	299.810703	213.3343793	304.4116207

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
-	-	-	-	-	-
0.001	0.0016669	0.002	0.0006669	0.0026669	-
0.0140026	0.0073569	0.0200148	0.0013447	0.0213594	1e-07
0.0551123	0.0173357	0.0703485	0.0020995	0.0724481	1e-07
0.1412518	0.0328282	0.170948	0.003132	0.174081	1e-06
0.2925421	0.0576899	0.3452675	0.0049645	0.3502337	1.7e-06
0.5381883	0.1028887	0.6319867	0.0090903	0.6410818	4.8e-06
0.9307269	0.2023831	1.112143	0.020967	1.1331276	1.76e-05
1.5855744	0.4908456	2.0066725	0.0697475	2.0764608	4.08e-05
2.8200352	1.8681048	4.2151357	0.4730043	4.686651	0.001489
5.8520996	311.893900	416.8345715	300.911428	571.5789954	246.1670046

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
-	-	0.0090036	0.0090036
0.0026669	-	0.0003334	0.0023336
0.0213598	3e-07	0.0005	0.0208595
0.0724496	1.6e-06	0.0095054	0.0629426
0.1740851	5.1e-06	0.0426495	0.1314305
0.3502454	1.34e-05	0.1169648	0.2332672
0.6411211	4.41e-05	0.2529344	0.3881426
1.1333095	0.0001995	0.4811599	0.6519501
2.0779374	0.0015174	0.8605047	1.2159153
4.7292627	0.0411227	1.5404261	3.1477139
14.9841617	302.761838	33.1062705	314.6397295

Table 16: Results of numerical methods for the ODE (20), h = 0.2

x	y_i	Adam's Bashforth	Error Adam's Bashforth	Adam's Moulton	Error Adam's Moulton
0	0	-	-	-	-
0.2	0.0026669	0.0026669	-	0.0026669	-
0.4	0.0213595	0.0213595	-	0.0213595	-
0.6	0.072448	0.072448	-	0.072448	-
0.8	0.17408	0.1732982	0.0007818	0.1742028	0.0001228
1.0	0.350232	0.3472892	0.0029428	0.3506828	0.0004508
1.2	0.641077	0.6320536	0.0090234	0.6425673	0.0014903
1.4	1.13311	1.1033871	0.0297229	1.139044	0.005934
1.6	2.07642	1.9505766	0.1258434	2.1141924	0.0377724
1.8	4.68814	3.7886314	0.8995086	6.6660013	1.9778613
2.0	317.746	9.5648443	308.18115576.6660013		311.0799987

Euler's method	Error Euler's method	Taylor Series	Error Taylor series	Runge Kutta method	Error Runge Kutta method
-	-	-	-	-	-
-	0.0026669	-	0.0026669	0.0026669	1e-07
0.008	0.0133595	0.016	0.0053595	0.0213601	6e-07
0.0400128	0.0324352	0.0641538	0.0082942	0.0724512	3.2e-06
0.112333	0.061747	0.1619113	0.0121687	0.1740902	1.02e-05
0.2428567	0.1073753	0.331469	0.018763	0.3502575	2.55e-05
0.4546526	0.1864244	0.6081589	0.0329181	0.6411433	6.63e-05
0.7839944	0.3491156	1.0621576	0.0709524	1.13329	0.00018
1.2989239	0.7774961	1.8669987	0.2094213	2.0766884	0.0002684
2.1483645	2.5397755	3.5916268	1.0965132	4.67089	0.01725
3.7194586	314.02654	149.2103065	308.535693	539.2230645	278.5229355

Simpson's Method	Simpson's Method error	Trapezoidal Rule method	Trapezoidal Rule method error
-	-	0.072448	0.072448
0.0026669	-	0.0213595	0.0186926
0.0213595	-	0.0026669	0.0186926
0.072448	-	-	0.072448
0.1741447	6.47e-05	0.0040016	0.1700784
0.3504038	0.0001718	0.0240611	0.3261709
0.6416932	0.0006162	0.0767074	0.5643696
1.1357185	0.0026085	0.1805558	0.9525542
2.0950572	0.0186372	0.3608361	1.7155839
5.280246	0.592106	0.6616321	4.0265079
7.4965581	310.24944	191.1860884	316.5599116

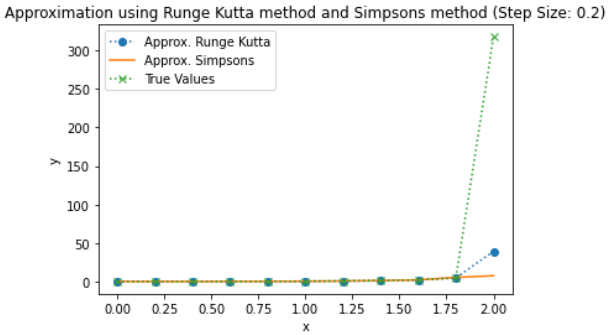

Fig. 8: The image shows the difference between the approximation of Runge Kutta method, Simpsons method and the true values in **Test Problem 8**, with step size = 0.2

4 Discussion

We covered eight different examples for different purposes, three linear equations and five nonlinear equations, six examples that have known exact smooth solutions, and two examples that have no known exact/closed-form solution, we compared different finite difference numerical methods for solving these examples for mesh size 0.1 and 0.2.

From the results reported in **Table 1** and **Table 2** we can conclude that Runge Kutta method is the best for **Test Problem 1**.

From the results reported in **Table 3** and **Table 4** we can conclude that Runge Kutta method is the best for **Test Problem 2**.

From the results reported in **Table 5** and **Table 6** we can conclude that Runge Kutta method is the best for **Test Problem 3**.

From the results reported in **Table 7** and **Table 8** we see that Runge Kutta, Simpson's Adam's Bashforth and Adam's Moulton performed well with**Test Problem 4**, which is similar to the following equations:

$$\frac{dy}{dx} = x(1-0.0005x) \tag{21}$$

$$\frac{dy}{dx} = -10x((1-7x) \tag{22}$$

From the results reported in **Table 9** and **Table 10** we can conclude that Simpson's method is the best for **Test Problem 5**.

From the results reported in **Table 11** and **Table 12** we can conclude that Adam's Moulton method is the best for **Test Problem 6**.

From the results reported in **Table 13** and **Table 14** we can conclude that Simpson's method is the best for **Test Problem 7**.

From the results reported in **Table 15** and **Table 16** we can conclude that none of the numerical methods used in this paper is suitable for this type of DE **Test Problem 8** and similar ones.

5 Conclusion

In this review study, we provided a comprehensive survey to classical finite difference techniques employed in the numerical resolution of first-order initial value problems. These techniques encompass the Adams-Bashforth Method, Adams-Moulton Method, Euler's Method,

Taylor Method, Fourth Order Runge-Kutta Method, Simpson's Method, and the Trapezoidal Rule Method. Additionally, we have included dynamic Python code implementations for each of these methods. Our findings are visually represented through tables and figures. Our survey study helps researchers in numerical analysis who develop new techniques to solve ODEs to apply their algorithms with classical FDM by directly using the offered dynamic codes.

Availability of data and material

The authors did not use any scientific data during this research.

Conflict of interest

The authors declare that they have no conflict of interest.

Funding

The authors received no financial support for the research.

Authors' contributions

All authors contributed equally work.

Acknowledgement

The authors would like to thank the anonymous reviewers for carefully reading the article and also for their constructive and valuable comments, which have improved the paper in its present form.

References

[1] Richard L. Burden, J. Douglas Faires, Annette M. Burden. *Numerical analysis*. Cengage Learning, 2015.
[2] Anthony Ralston, Philip Rabinowitz. *A first course in numerical analysis*. Courier Corporation, 2001.
[3] Leon Lapidus, John H. Seinfeld. *Numerical solution of ordinary differential equations*. Academic Press, 1971.