Numerical Analysis Final Project Report

Shumokh Naji Abdullah (S21107192) - Hadeel Balahmar (S20106481)

Computer Science Department, Effat University

Title:

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

Abstract:

This report presents the numerical solutions of initial value problems (IVPs) using three different methods: Euler's method, Taylor's method, and the Runge-Kutta method. Each method is described theoretically, followed by the corresponding Python code implementation. A numerical example is provided to demonstrate the application of these methods, and the results are visualized through a plot. By comparing the solutions obtained from the different methods, we can observe the accuracy and efficiency of each approach.

Theoretical Background:

Euler's Method:

Euler's method is a simple numerical method used to approximate solutions to first-order ordinary differential equations (ODEs) based on a given initial condition. It is based on the idea of approximating the derivative using the slope of a tangent line at each point. The method proceeds by taking small steps along the curve and updating the value of the function based on the local slope. The algorithm is as follows:

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

```python
import math
import matplotlib.pyplot as plt

def euler_method(f, x0, y0, h, n):
    """
    Solves an initial value problem using Euler's method.

    Parameters:
    - f: The function defining the differential equation dy/dx = f(x, y).
    - x0: The initial value of x.
    - y0: The initial value of y at x = x0.
    - h: The step size.
    - n: The number of steps to take.

    Returns:
    - x: A list of x-values.
    - y: A list of corresponding y-values.
    """
    x = [x0]
    y = [y0]

    for i in range(n):
        xi = x[i]
        yi = y[i]
        xi_plus_1 = xi + h
        f_func = lambda x, y: eval(f)  # Create dynamic function using lambda
        yi_plus_1 = yi + h * f_func(xi, yi)
        x.append(xi_plus_1)
        y.append(yi_plus_1)

    return x, y

try:
    # User input for the function f
    function_input = input("Enter the function f(x, y): ")
    f = function_input.replace('^', '**').replace('e', 'math.e').replace('sin',
'math.sin').replace('cos', 'math.cos').replace('exp', 'math.exp')

    # User input for other parameters
    x0 = float(input("Enter the initial value of x: "))
    y0 = float(input("Enter the initial value of y at x = x0: "))
    h = float(input("Enter the step size: "))
    n = int(input("Enter the number of steps: "))

    x_values, y_values = euler_method(f, x0, y0, h, n)

    # Print the results
    for i in range(len(x_values)):
        print(f"x = {x_values[i]}, y = {y_values[i]}")

    # Plotting the points on a graph
    plt.plot(x_values, y_values, '-o')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title("Euler's Method")
    plt.show()

except:
    print("Invalid input. Please make sure you have entered the correct values.")
```

Taylor's Method:

Taylor's method is an extension of Euler's method that provides higher-order accuracy by considering additional terms in the Taylor series expansion of the function. By incorporating these higher-order derivatives, the method improves the approximation accuracy and reduces the truncation error. The algorithm is as follows:

$$y_{i+1} = y_i + hf(x_i, y_i) + (h^\wedge 2/2)f'(x_i, y_i)$$

```
import math
import matplotlib.pyplot as plt

def taylor_method(f, x0, y0, h, n):
    x = [x0]
    y = [y0]

    for i in range(n):
        xi = x[i]
        yi = y[i]
        xi_plus_1 = xi + h

        f_func = lambda x, y: eval(f)
        dy_dx = lambda x, y: (f_func(x + 0.0001, y) - f_func(x, y)) / 0.0001

        yi_plus_1 = yi + h * f_func(xi, yi) + (h ** 2) / 2 * dy_dx(xi, yi)
        x.append(xi_plus_1)
        y.append(yi_plus_1)

    return x, y

try:
    # User input for the function f
    function_input = input("Enter the function f(x, y): ")
    f = function_input.replace('^', '**').replace('e', 'math.e').replace('sin', 
'math.sin').replace('cos', 'math.cos').replace('exp', 'math.exp')

    # User input for other parameters
    x0 = float(input("Enter the initial value of x: "))
    y0 = float(input("Enter the initial value of y at x = x0: "))
    h = float(input("Enter the step size: "))
    n = int(input("Enter the number of steps: "))

    x, y = taylor_method(f, x0, y0, h, n)
    # Print the results
    for i in range(len(x)):
        print(f"x = {x[i]}, y = {y[i]}")

    # Plotting the points on a graph
    plt.plot(x, y, '-o')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title("Taylor Method")

    plt.show()

except:
    print("Invalid input. Please make sure you have entered the correct values.")
```

Runge-Kutta Method:

The Runge-Kutta method is a widely used numerical method for solving initial value problems. It achieves higher accuracy by calculating intermediate values of the function at different points within each step. The most common form is the fourth-order Runge-Kutta method, which approximates the solution by considering the weighted average of four function evaluations at different points. The algorithm is as follows:

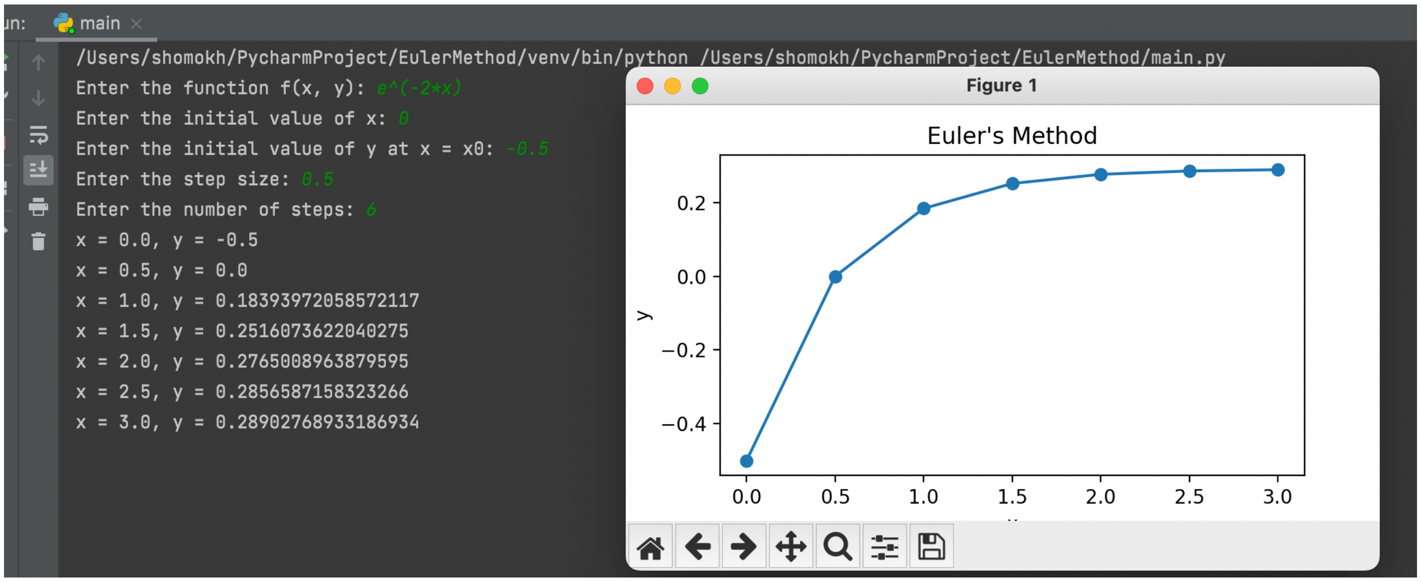
$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right), \\
k_3 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right), \\
k_4 &= f(t_n + h, y_n + hk_3).
\end{aligned}$$

$$\begin{aligned}
y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \\
t_{n+1} &= t_n + h
\end{aligned}$$

```
import numpy as np
import matplotlib.pyplot as plt

try:
    # User input for the function f(t, y)
    function_input = input("Enter the function f(t, y): ")
    # Define the function based on user input
    def f_user(t, y):
        return eval(function_input.replace('^', '**').replace('e',
'np.e').replace('sin', 'np.sin').replace('cos', 'np.cos'))

    # User input for the initial condition and step size
    t0 = float(input("Enter the initial t value: "))
    y0 = float(input("Enter the initial y value: "))
    h = float(input("Enter the step size (h): "))
    # Define the number of steps to take
    n = round((1 - t0) / h)

    # Initialize arrays to store the solution
    t = np.zeros(n+1)
    y = np.zeros(n+1)
    t[0] = t0
    y[0] = y0

    # Implement the RK4 method
    for i in range(n):
        k1 = h * f_user(t[i], y[i])
        k2 = h * f_user(t[i] + h/2, y[i] + k1/2)
        k3 = h * f_user(t[i] + h/2, y[i] + k2/2)
        k4 = h * f_user(t[i] + h, y[i] + k3)
        y[i+1] = y[i] + 1/6 * (k1 + 2*k2 + 2*k3 + k4)
        t[i+1] = t[i] + h

    # Print the results in a table
    print(" i    |   t    |   y   ")
    print("-----------------------")
    for i in range(n+1):
        print(f"{i:2d}    |  {t[i]:.1f}  |  {y[i]:.6f}")

    # Plot the solution
    plt.plot(t, y, '-o')
    plt.xlabel('t')
    plt.ylabel('y')
    plt.title('Numerical Solution using RK4')
    plt.show()

except:
    print("Invalid input. Please make sure you have entered the correct values.")
```

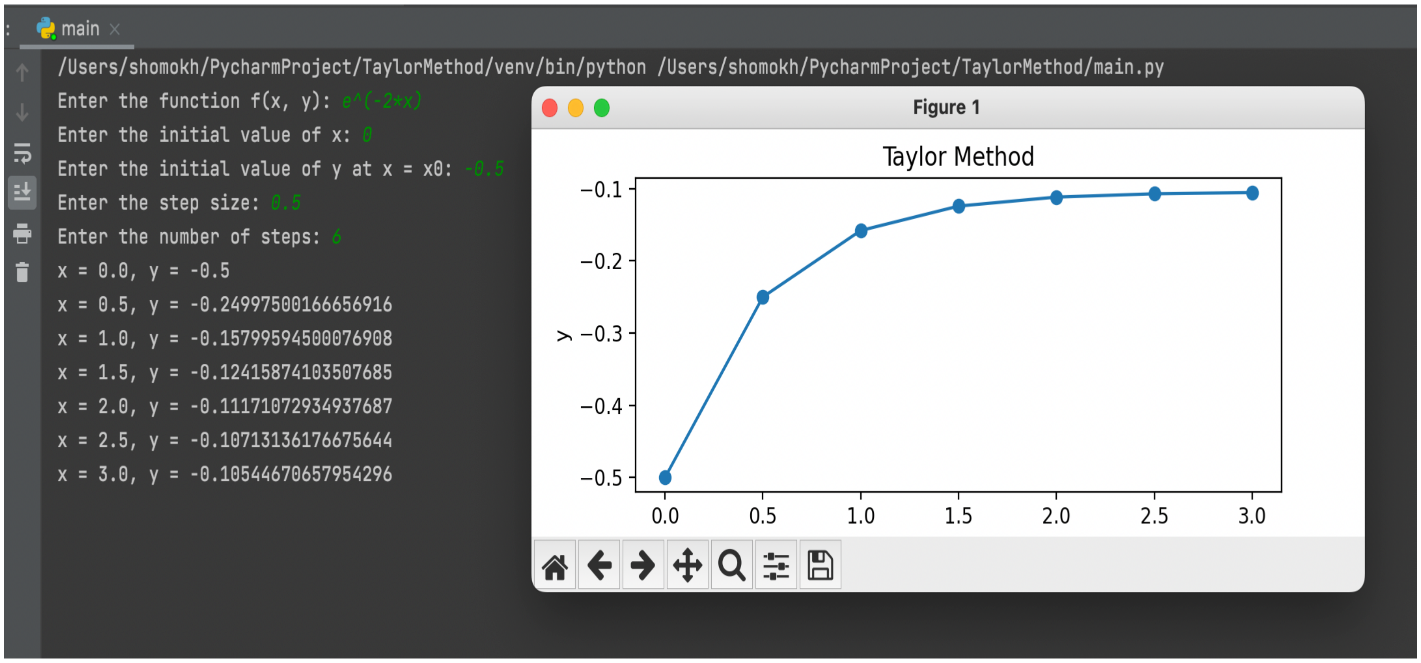
Execution:

Euler method



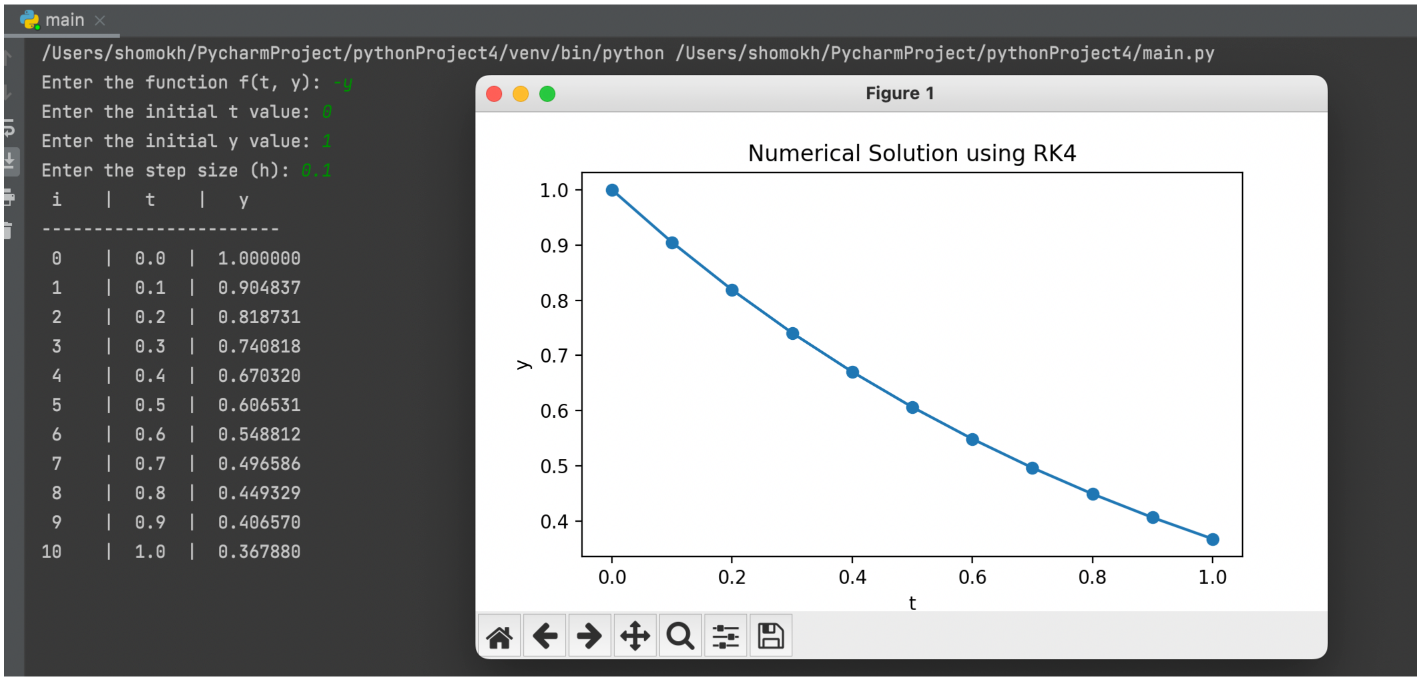
```
/Users/shomokh/PycharmProject/EulerMethod/venv/bin/python /Users/shomokh/PycharmProject/EulerMethod/main.py
Enter the function f(x, y): e^(-2*x)
Enter the initial value of x: 0
Enter the initial value of y at x = x0: -0.5
Enter the step size: 0.5
Enter the number of steps: 6
x = 0.0, y = -0.5
x = 0.5, y = 0.0
x = 1.0, y = 0.18393972058572117
x = 1.5, y = 0.2516073622040275
x = 2.0, y = 0.2765008963879595
x = 2.5, y = 0.2856587158323266
x = 3.0, y = 0.28902768933186934
```

Taylor method



```
/Users/shomokh/PycharmProject/TaylorMethod/venv/bin/python /Users/shomokh/PycharmProject/TaylorMethod/main.py
Enter the function f(x, y): e^(-2*x)
Enter the initial value of x: 0
Enter the initial value of y at x = x0: -0.5
Enter the step size: 0.5
Enter the number of steps: 6
x = 0.0, y = -0.5
x = 0.5, y = -0.24997500166656916
x = 1.0, y = -0.15799594500076908
x = 1.5, y = -0.12415874103507685
x = 2.0, y = -0.11171072934937687
x = 2.5, y = -0.10713136176675644
x = 3.0, y = -0.10544670657954296
```

Runge-kutta method



```
/Users/shomokh/PycharmProject/pythonProject4/venv/bin/python /Users/shomokh/PycharmProject/pythonProject4/main.py
Enter the function f(t, y): -y
Enter the initial t value: 0
Enter the initial y value: 1
Enter the step size (h): 0.1
 i    |   t    |   y
-----------------------
 0    |  0.0  |  1.000000
 1    |  0.1  |  0.904837
 2    |  0.2  |  0.818731
 3    |  0.3  |  0.740818
 4    |  0.4  |  0.670320
 5    |  0.5  |  0.606531
 6    |  0.6  |  0.548812
 7    |  0.7  |  0.496586
 8    |  0.8  |  0.449329
 9    |  0.9  |  0.406570
10    |  1.0  |  0.367880
```

Conclusion:

In this report, we have discussed and implemented three numerical methods for solving initial value problems: Euler's method, Taylor's method, and the Runge-Kutta method. Through a numerical example, we compared the solutions obtained from these methods. The results indicate that the Runge-Kutta method provides the most accurate approximation, followed by Taylor's method and Euler's method. Overall, these methods provide valuable tools for approximating solutions to differential equations in various scientific and engineering applications.