Lab Manual for Numerical Analysis

Lab No. 10

NUMERICAL SOLUTION OF ODES - EULER'S METHOD, IMPROVEMENT OF EULER'S METHOD

Software Engineering Department
Bahria University (Karachi Campus)

BAHRIA UNIVERSITY KARACHI CAMPUS
Department of Software Engineering

NUMERICAL ANALYSIS

LAB EXPERIMENT # 10

Numerical Solution of ODEs - Euler's method, improvement of Euler's method

OBJECTIVE:

The objective of this lab is to introduce **Euler's method** for solving **ODEs** numerically, emphasizing limitations and exploring modifications to enhance solution accuracy.

Introduction

ODEs stand for **Ordinary Differential Equations**, which are pivotal mathematical expressions involving functions of a singular variable and their derivatives. They establish profound connections between a function and its derivatives within a single independent variable. These equations serve as the backbone for depicting the evolving behaviors of diverse dynamic systems across time, encompassing realms such as physical phenomena, biological processes, economic models, and intricate engineering systems. They provide a fundamental framework for modeling and understanding how these systems evolve and change over time, playing a vital role in various scientific and applied disciplines.

In engineering, ODEs are crucial for modeling and understanding dynamic systems such as electrical circuits, mechanical systems, chemical reactions, control systems, and more. They enable engineers to predict and analyze system behavior, design optimal controls, simulate complex processes, and predict system responses to different inputs or perturbations.

Numerical methods play a crucial role in solving ODEs, particularly when deriving analytical solutions becomes arduous or impractical. These techniques encompass a diverse array of approaches essential for approximating solutions to differential equations that might pose challenges or lack feasible analytical solutions. Some prevalent numerical methods employed for this purpose include: Euler's Method, Improved Euler's Method, Runge-Kutta Methods, Finite Difference Methods, Boundary Value Methods, and Shooting Methods etc. These numerical approaches help engineers obtain approximate solutions to ODEs, enabling them to analyze and predict the behavior of systems that are otherwise challenging to solve analytically.

In this lab, we will explore and implement Euler's methods for the numerical approximation of ordinary differential equations (ODEs), examining its principles and practical applications in solving dynamic systems.

Implementation of Euler's Method

This section will cover Euler's method, elucidating its principles and demonstrating its Python implementation for numerical solutions of ordinary differential equations (ODEs).

a. Euler's method

Euler's method is a straightforward numerical technique used to approximate the solutions of ordinary differential equations (ODEs). It works by iteratively estimating the next point of a function's graph based on its derivative at the current point. The method uses the derivative to predict the function's behavior over small intervals.

The formula for Euler's method is:

$$y_{n+1} = y_n + h \cdot f(t_n, y_n)$$

Where:

- y_{n+1} is the next approximation of the solution.
- y_n is the current approximation of the solution.
- h is the step size (the interval at which the solution is approximated).
- $f(t_n, y_n)$ represents the derivative of the function at the point (t_n, y_n)

Implementation in Python

- The following code provides a function **euler_method** to solve the ordinary differential equation x **+ y + xy** at **y (0.1)** with initial condition **y (0) = 1** and step size **h = 0.025**.

```
# Defining a differential equation dy/dx = (x + y + xy)
def func( x, y ):
  return (x + y + x * y)

# Function for euler formula
def euler_method( t0, y, h, t ):
  # Iterating till the point at which we need approximation
  while t0 < t:
    y += h * func(t0, y)
    t0 += h

  # Printing approximation
  print("Approximate solution at t = ", t, " is ", "%.4f"% y)

# Initial Values
t0 = 0
y0 = 1
h = 0.025

t = 0.1 # Value of x at which we need approximation
euler_method(t0, y0, h, t)
```

b. Predictor - corrector method: the improved Euler method

The Predictor-Corrector method, also recognized as the Modified Euler method, addresses limitations present in the basic Euler method, which primarily performs well with linear functions but exhibits truncation errors with non-linear ones. To mitigate this, the Modified Euler method employs an arithmetic mean of slopes over the interval (t_n, t_{n+1}) instead of a single point, enhancing accuracy.

Within the Predictor-Corrector method, each step involves two stages: firstly, the anticipated value of y_{n+1} is computed using Euler's method. Subsequently, slopes at the points (t_n, y_n) and (t_{n+1}, y_{t+1}) are determined, and their arithmetic average is added to y_n to calculate the adjusted value of y_{n+1}.

By utilizing the average slope, this method substantially reduces errors. Moreover, it allows for iterative correction, progressively diminishing errors at each step, thereby refining the value of y and improving overall accuracy.

The **predictor-corrector method** is implement in two steps:

Predictor Step: The Predictor step typically employs a simple method like Euler's method to predict the next value of the solution:

$$p_y = y_n + h \cdot f(t_n, y_n)$$

Where:

- p_y is the predicted value at y_{n+1}
- y_n is the current approximation of the solution.
- h is the step size (the interval at which the solution is approximated).
- $f(t_n, y_n)$ represents the derivative of the function at the point (t_n, y_n)

Corrector Step: The Corrector step adjusts the prediction by using a weighted average of the predicted and actual derivatives to refine the solution:

$$y_{n+1} = y_n + \frac{h}{2}\left(f(t_n, y_n) + f(t_{n+1}, p_y)\right)$$

Where:

- y_{n+1} is the refined approximation of the solution.
- y_n is the previous approximation of the solution.
- h is the step size.
- $f(t_n, y_n)$ represents the derivative at the point (t_n, y_n).
- p_y is the predicted value obtained in the prediction step.

Implementation in Python

- The following code provides a function **predictor_corrector_method** (which is the improved version of euler's method) to solve the ordinary differential equation **$y - 2x^2 + 1$** at **y (1)** with initial condition **y (0) = 0.5** and step size **h = 0.2**.

```
def f(x, y):
    return y - 2 * x * x + 1;

def predictor_corrector_method(t0, tn, y0, h):
    # Iterating until the final time tn is reached
    while (t0 < tn):
        # Calculating the next time step
        t1 = t0 + h

        # Predictor step to calculate the predicted value of y at t0
        p_y = y0 + h * f(t0, y0)

        # Corrector step to refine the value of y at t1
        c_y = y0 + (h / 2) * (f(t0, y0) + f(t1, p_y))

        # Updating the t and y value for the next iteration
        t0 = t1
        y0 = p_y

    return y0

#initial values
t0 = 0 #initial value of t
y0 = 0.5 #initial value of y
h = 0.2 #step size

tn = 1  # Value of x at which we need approximation

# Printing the final value of y at tn using Predictor-Corrector
method
print("The final value of y at t =", tn, "is:",
predictor_corrector_method(t0, tn, y0, h))
```

Lab Tasks

1. Write a Python program that utilize Euler's method for IVP: $y' = 2 - e^{-4t} - 2y$ with y (0) = 1, step size of h=0.1 to find approximate values of the solution at t = 0.1, 0.2, 0.3, 0.4, and 0.5.

   (Output: 0.9, 0.852967995, 0.837441500, 0.839833779, 0.851677371)

2. Write a Python program that utilize improved Euler's method for IVP: $\boldsymbol{y' = -\frac{1}{2}e^{\frac{t}{2}}\sin(5t) + 5e^{\frac{t}{2}}\cos(5t) + y}$ with y (0) = 0, to find approximate values of the solution at t = 1, t = 2, t = 3, t = 4, and t = 5. Use h = 0.1, h = 0.05, h = 0.01, h = 0.005 and h = 0.001 for the approximations.

   Output:

	approximations				
t	***h=0.1***	***h=0.05***	***h=0.01***	***h=0.005***	***h=0.001***
t = 1	-0.75594882	-1.26511659	-1.51580084	-1.54826581	-1.57443054
t = 2	0.65270348	-0.34326837	-1.23906961	-1.41821032	-1.46664450
t = 3	7.30208959	4.69013206	3.29452477	3.10610361	2.95299750
t = 4	15.5612836	13.02600082	8.09409787	7.42609891	6.88285949
t = 5	30.2683985	15.91132648	2.18799672	0.30780976	-1.28498314