Numerical Analysis Project 3 Fernando Thaureaux: 6236069

Writing the Code

-Language: Python

-Input:

- h - the step size, which determines the time intervals at which we find approximate solutions, smaller h improves accuracy but increases computational complexity (in theory)
- t - time points inside the time interval
- t_end - the final time when the solution is approximated (t = 2)
- initial condition (y0) - defining y(0) = 0, the starting value for the diff eq.
- Derivative Func (dydt) - defining y'(t) = t + y, the differential equation that represents the slope of the solution curve for approximations
- Exact Solution - defining y(t) = $e^t - t - 1$, so we can calculate the true values at t for comparison with approximations
- Tol - the tolerance for error between successive approximations (10^-6) to determine the stopping point of the program
- max_iter - max number of iterations to run if the error tolerance is never achieved

-Output:

The results are printed in a table that displays the step size (h), time (t), predictor-correct value, exact solution, and percent error. The percentage error is calculated as |(Approx - Exact)/ Exact| * 100.

The method is applied twice because we are looking for the results considering two step sizes (h = 0.25 and h = 0.5) to show how reducing h improves accuracy.

Choice of Parameters

We choose parameters to balance accuracy, computational complexity, and demonstrate convergence. We choose step size h = 0.25 as the smaller step size, which in theory would increase computational complexity (making it slower) but provide a more accurate result. Meanwhile step size h = 0.5 is a larger step size, which should decrease computation (making it faster) but at the cost of lower accuracy. A high tolerance at 10^-6 provides room for the code to run until we have a high precision result. The maximum iterations prevent an infinite loop. We use the initial conditions defined by the IVP and set the end time at t = 2 because

the problem specifically asks for the approximations at y(2) or when t = 2. The code generates intermediate values t = 0.5, 1, 1.5, 2, to achieve the result.

Results and Analysis

Results for Euler-Trapezoidal Predictor-Corrector Method at y(2):

Step size (h)	t	Predictor-Corrector	Exact Solution	Error (%)
0.50	2.0	4.716042	4.389056	7.450028
0.25	2.0	4.467153	4.389056	1.779361

As expected, the smaller the step size h, the closer to the exact result the predictor-corrector method is. It has a much lower percentage error. Comparing the results with Problem 1 (part a), we see that the Euler-Trapezoidal predictor-corrector method is much more accurate than the basic Euler method, and surprisingly is actually **slightly** more accurate than the Modified Euler, Heun, and Midpoint methods also. This method has a second order accuracy, so it being more accurate than Euler is predictable, and it is possible for it to be more accurate than the Modified and Midpoint methods, but having more accuracy than Heun's is unexpected, although Heun seems to behave the same way as the Modified and Midpoint methods for this function anyway. It might be because of the nature of the predictor-corrector method, and the way it uses both an explicit and implicit method of solving for the function, using iterations to find convergence. This is obviously only for the step size h = 0.5, where the values are only marginally different (this method has 7.45% error and the other high-order methods have 9.48%). Clearly for a smaller step size h = 0.25 it is much more accurate for methods of comparable orders (error being 1.78%).

Code

Euler-Trapezoidal Predictor-Corrector:

```
import numpy as np
import pandas as pd
# Define the derivative function and exact solution
def dydt(t, y):
    return t + y
def exact_solution(t):
    return np.exp(t) - t - 1
# Euler-Trapezoidal Predictor-Corrector Method
def predictor_corrector_method(f, t_values, y0, h, tol=1e-6, max_iter=10):
```

```
    y_values = [y0]
    for i in range(len(t_values) - 1):
        t_n = t_values[i]
        y_n = y_values[-1]
        # Predictor Step (Euler's Method)
        y_pred = y_n + h * f(t_n, y_n)
        # Corrector Step (Trapezoidal Rule)
        y_corr = y_pred  # Initialize corrector
        for _ in range(max_iter):
            y_new = y_n + (h / 2) * (f(t_n, y_n) + f(t_n + h, y_corr))
            if abs(y_new - y_corr) < tol:  # Convergence check
                break
            y_corr = y_new
        y_values.append(y_corr)
    return y_values
# Function to run and display results for two step sizes
def run_and_display(h_values, t_end, y0):
    results = []
    # Run for each step size
    for h in h_values:
        t_values = np.arange(0, t_end + h, h)
        y_exact = exact_solution(t_values)
        y_pc = predictor_corrector_method(dydt, t_values, y0, h)
        # Append the results at t = 2
        idx_2 = np.where(t_values == 2.0)[0][0]
        results.append({
            "Step size (h)": h,
            "t": t_values[idx_2],
            "Predictor-Corrector": y_pc[idx_2],
            "Exact Solution": y_exact[idx_2],
            "Error (%)": abs((y_pc[idx_2] - y_exact[idx_2]) / y_exact[idx_2]) * 100
        })
  # Display results in tabular form
    df = pd.DataFrame(results)
    print("\nResults for Euler-Trapezoidal Predictor-Corrector Method at y(2):")
    print(df.to_string(index=False))
# Define parameters
y0 = 0  # Initial condition
t_end = 2  # Target time
h_values = [0.5, 0.25]  # Step sizes to ensure convergence
# Run and display results
run_and_display(h_values, t_end, y0)
```