Practice Task 1

Course: Simulation of Robotics System
SRS 2025

Task 1: Integrators

Student Name: Ghulam Mujtaba Sahto
ITMO ID: 507985
Group: R4137c
Instructor: Prof.

Comparison of ODE's Integrators

1. Objective

The purpose of this task is to solve a second-order ordinary differential equation (ODE) both analytically and numerically using three numerical integrators: **Explicit Euler, Implicit Euler,** and the **Runge-Kutta (RK4)** method. The results of these numerical methods were then compared with the analytical solution to analyze their accuracy and behavior.

2. Given Equation

The given equation is:

$$\boldsymbol{a * \ddot{x} + b * \dot{x} + c * x = d}$$

where the coefficients from the given table are:

$$\boldsymbol{a = -0.56, \qquad b = -1.33, \qquad c = 3.88, \qquad d = 0.37}$$

Therefore the second-order ODE will be:

$$\boldsymbol{-0.56 * \ddot{x} + -1.33 * \dot{x} + 3.88 * x = 0.37}$$

Rearranging into the standard form:

$$\boldsymbol{\ddot{x} = 6.929x - 2.375\dot{x} - 0.661}$$

3. Analytical Solution

The analytical solution of the ODE is:

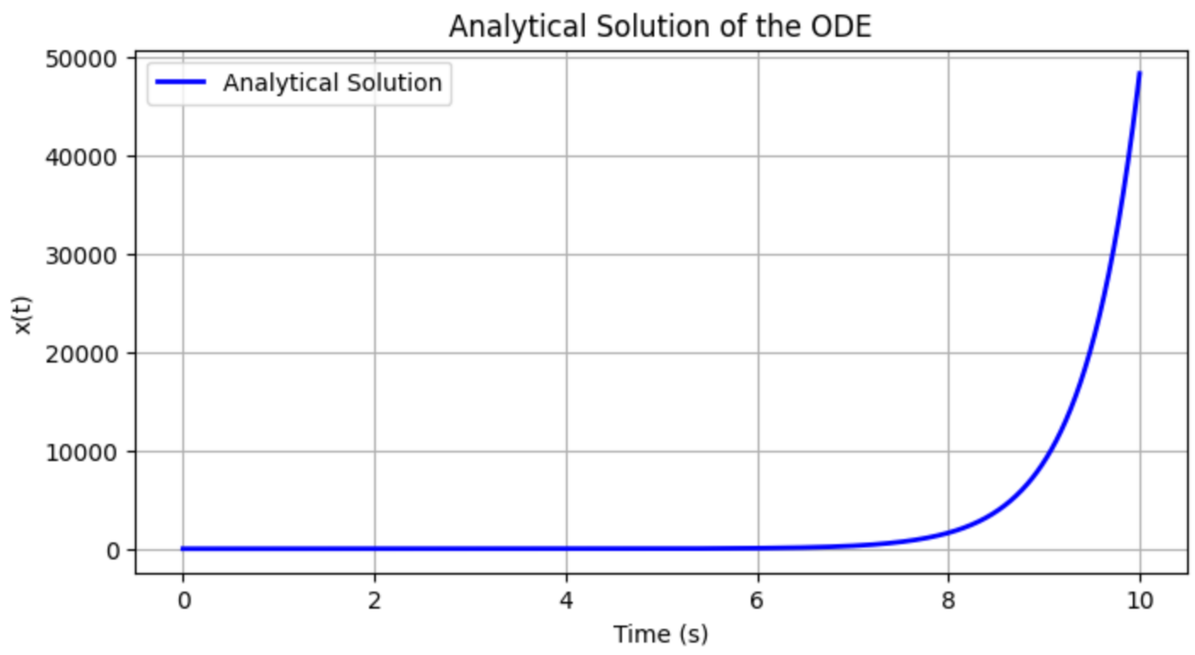
$$x(t) = 0.002e^{1.7t} + 0.001e^{-4.075t} + 0.095$$

The particular solution of ODE is calculated by the given Initial values, where initial position ($x_o = 0.1$) and initial velocity ($v_o = 0.0$).

This equation represents a system where one exponential term is decaying (due to a negative exponent) and the other is growing (due to a positive exponent). The constant term (0.095) represents the steady-state offset.

Code for the analytical solution plot:

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



# --- Analytical Solution ---
r1 = 1.7
r2 = -4.075
A  = 0.095

# Initial conditions
x0 = 0.1
v0 = 0.0

C1 = 0.002
C2 = 0.001

# Time and analytical solution
t = np.linspace(0, 10, 1001)
x_analytic = C1*np.exp(r1*t) + C2*np.exp(r2*t) + A

# --- Plot ---
plt.figure(figsize=(8, 4))
plt.plot(t, x_analytic, 'b', linewidth=2, label='Analytical Solution')
plt.xlabel('Time (s)')
plt.ylabel('x(t)')
plt.title('Analytical Solution of the ODE')
plt.legend()
plt.grid(True)
plt.show()
```



4. Numerical Solution Methods

Three numerical integrators are used to approximate the solution:

1. **Explicit Euler Method**: Simple, but conditionally stable. Accuracy depends on the step size.
2. **Implicit Euler Method**: Unconditionally stable but tends to overdamp oscillations.
3. **Runge–Kutta (RK4) Method**: High accuracy and stability, used for verification of the analytical solution.

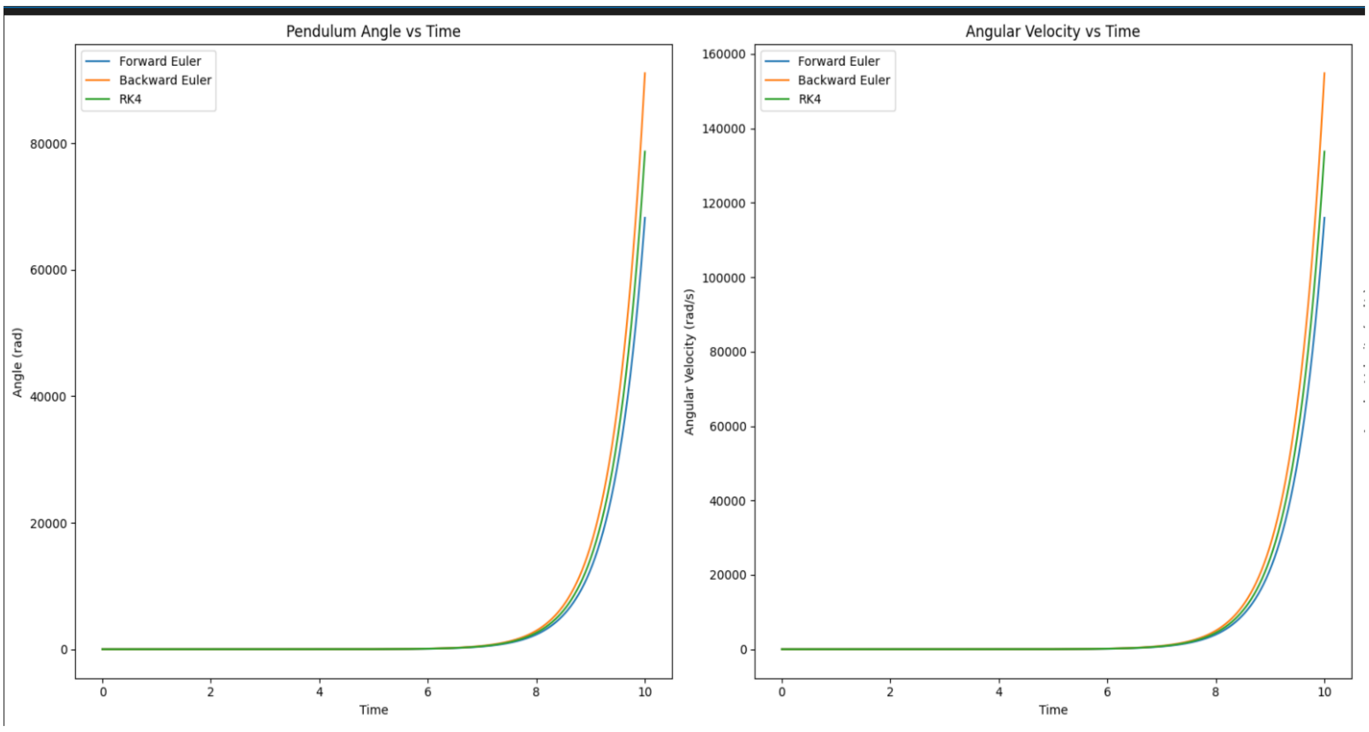
All methods are implemented in Python, and the system dynamics are defined as:

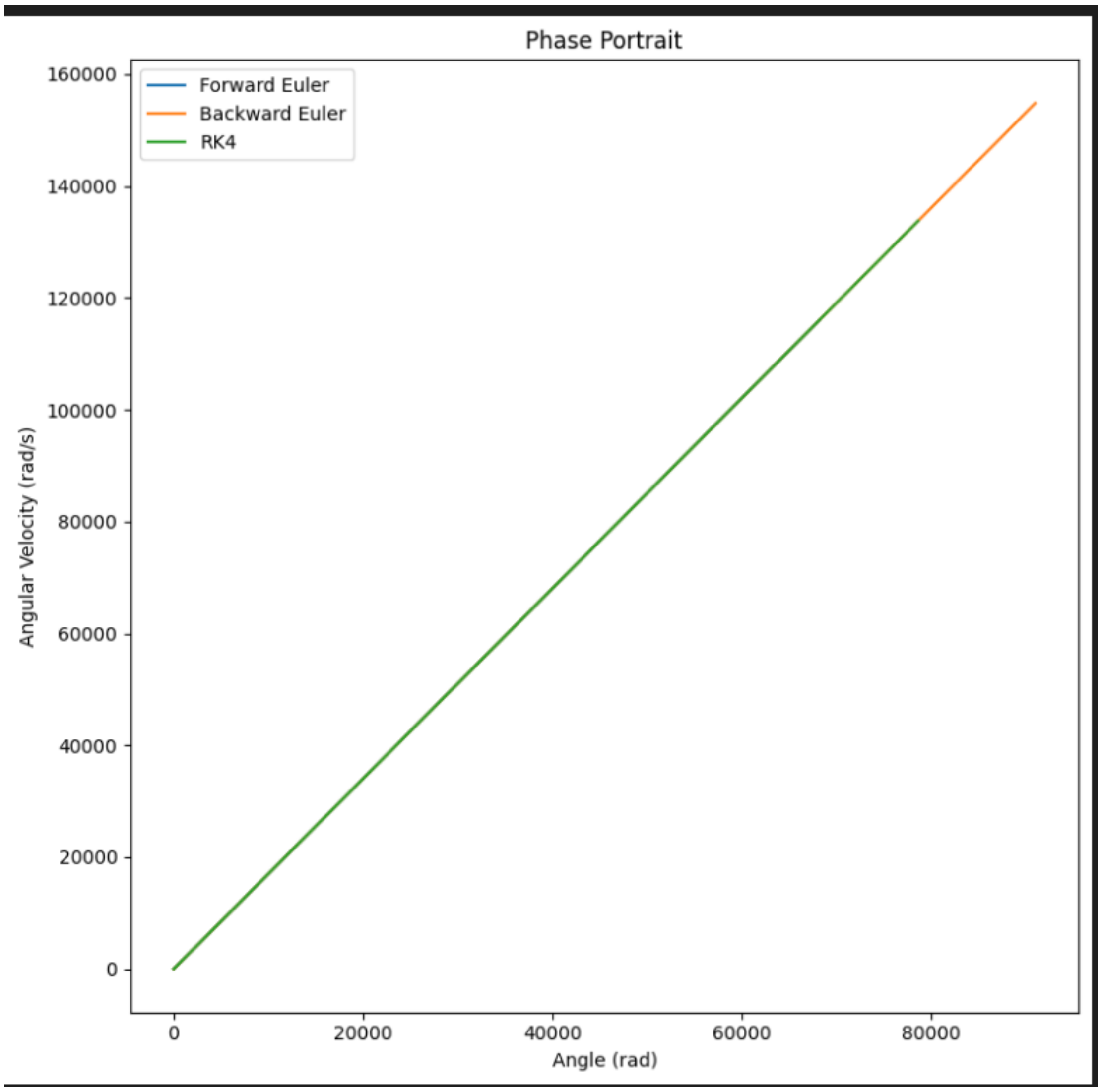
$$\ddot{x} = 6.929x - 2.375\dot{x} - 0.661$$

Code for the Numerical solutions (Explicit/Implicit and RK4):

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

def pendulum_dynamics(x):

    l = 1.0
    g = 9.81

    theta = x[0]
    theta_dot = x[1]

    theta_ddot = (6.929 * theta - 2.375 * theta_dot - 0.661)

    return np.array([theta_dot, theta_ddot])

def forward_euler(fun, x0, Tf, h):
    """
    Explicit Euler integration method
    """
    t = np.arange(0, Tf + h, h)
    x_hist = np.zeros((len(x0), len(t)))
    x_hist[:, 0] = x0

    for k in range(len(t) - 1):
        x_hist[:, k + 1] = x_hist[:, k] + h * fun(x_hist[:, k])

    return x_hist, t

def backward_euler(fun, x0, Tf, h, tol=1e-8, max_iter=100):
    """
    Implicit Euler integration method using fixed-point iteration
    """
    t = np.arange(0, Tf + h, h)
    x_hist = np.zeros((len(x0), len(t)))
    x_hist[:, 0] = x0

    for k in range(len(t) - 1):
        x_hist[:, k + 1] = x_hist[:, k]  # Initial guess

        for i in range(max_iter):
            x_next = x_hist[:, k] + h * fun(x_hist[:, k + 1])
            error = np.linalg.norm(x_next - x_hist[:, k + 1])
            x_hist[:, k + 1] = x_next

            if error < tol:
                break

    return x_hist, t
```

```python
def runge_kutta4(fun, x0, Tf, h):
    """
    4th order Runge-Kutta integration method
    """
    t = np.arange(0, Tf + h, h)
    x_hist = np.zeros((len(x0), len(t)))
    x_hist[:, 0] = x0

    for k in range(len(t) - 1):
        k1 = fun(x_hist[:, k])
        k2 = fun(x_hist[:, k] + 0.5 * h * k1)
        k3 = fun(x_hist[:, k] + 0.5 * h * k2)
        k4 = fun(x_hist[:, k] + h * k3)

        x_hist[:, k + 1] = x_hist[:, k] + (h / 6.0) * (k1 + 2*k2 + 2*k3 + k4)

    return x_hist, t

# Test all integrators
x0 = np.array([0.1, 0.0])  # Initial state: [angle, angular_velocity]
Tf = 10.0
h = 0.01

# Forward Euler
x_fe, t_fe = forward_euler(pendulum_dynamics, x0, Tf, h)

# Backward Euler
x_be, t_be = backward_euler(pendulum_dynamics, x0, Tf, h)

# Runge-Kutta 4
x_rk4, t_rk4 = runge_kutta4(pendulum_dynamics, x0, Tf, h)

# Plot results
plt.figure(figsize=(24, 8))

plt.subplot(1, 3, 1)
plt.plot(t_fe, x_fe[0, :], label='Forward Euler')
plt.plot(t_be, x_be[0, :], label='Backward Euler')
plt.plot(t_rk4, x_rk4[0, :], label='RK4')
plt.xlabel('Time')
plt.ylabel('Angle (rad)')
plt.legend()
plt.title('Pendulum Angle vs Time')

plt.subplot(1, 3, 2)
plt.plot(t_fe, x_fe[1, :], label='Forward Euler')
plt.plot(t_be, x_be[1, :], label='Backward Euler')
plt.plot(t_rk4, x_rk4[1, :], label='RK4')
plt.xlabel('Time')
plt.ylabel('Angular Velocity (rad/s)')
plt.legend()
plt.title('Angular Velocity vs Time')

plt.subplot(1, 3, 3)
plt.plot(x_fe[0, :], x_fe[1, :], label='Forward Euler')
plt.plot(x_be[0, :], x_be[1, :], label='Backward Euler')
plt.plot(x_rk4[0, :], x_rk4[1, :], label='RK4')
plt.xlabel('Angle (rad)')
plt.ylabel('Angular Velocity (rad/s)')
plt.legend()
plt.title('Phase Portrait')

plt.tight_layout()
plt.show()
```





4. Results and Comparison

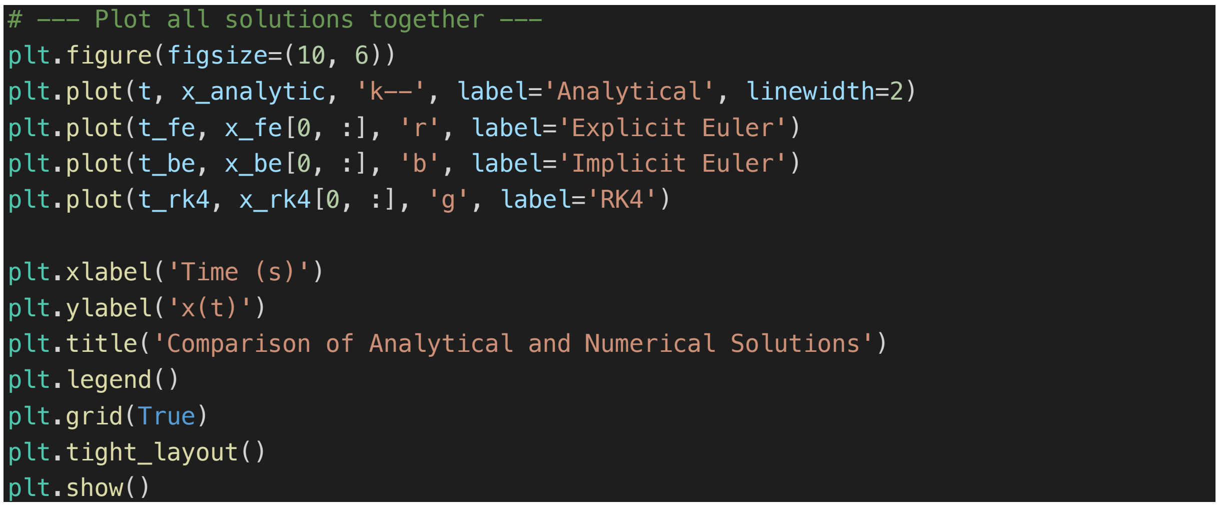
The comparison of the numerical and analytical results are carried out over a time interval of **0 to 10 seconds**. The following behavior is observed:

- The solutions closely overlap from **0 to 8 seconds**, showing a stable and almost linear growth in x(t), indicating that the system response was initially small and dominated by the steady-state term.
- After **8 seconds (approximately)**, the analytical, Explicit Euler, Implicit Euler and RK4 solutions began to **increase exponentially**, indicating an unstable or divergent response of the system.

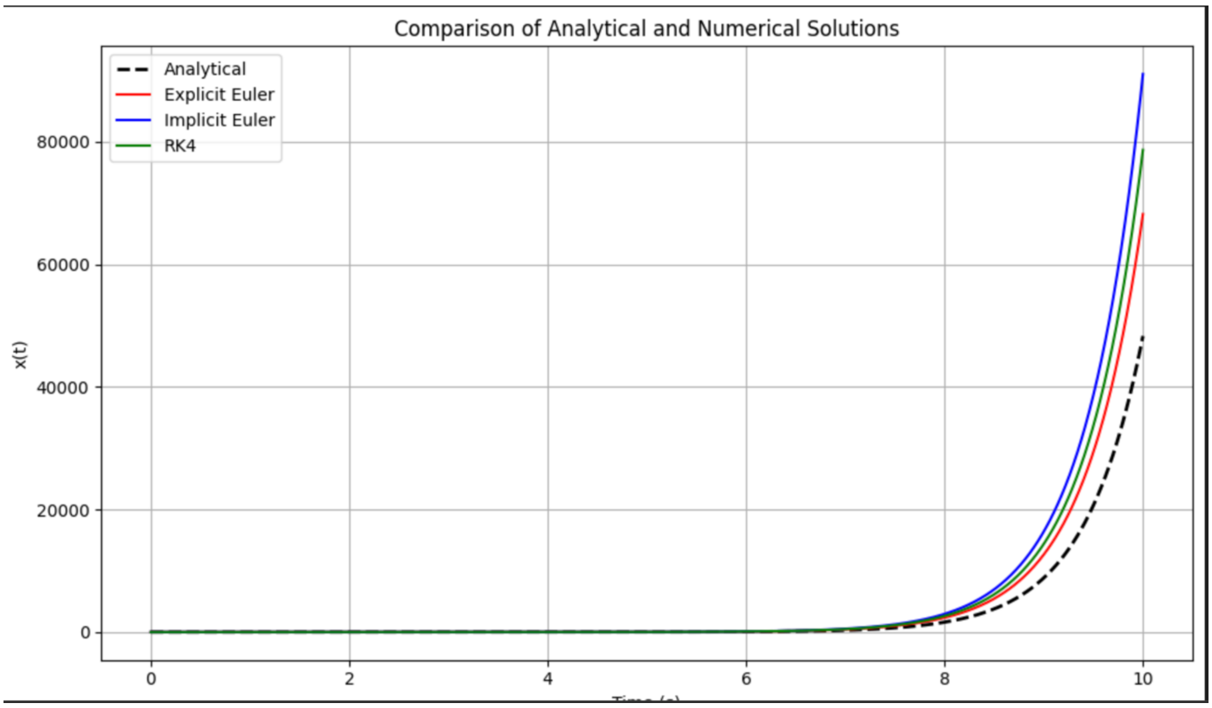
- The **Runge-Kutta (RK4)** and **Explicit Euler** method closely matched the analytical solution throughout the simulation, confirming that both the model and integration implementation are correct.
- while the **Implicit Euler** method shows a small overestimation. Despite these minor deviations, all numerical schemes capture the same exponential growth trend observed in the analytical solution.

This exponential growth after **8 seconds** is an expected physical behavior derived from the positive root in the analytical solution, not a numerical artifact.

Code for comparison of analytical and numerical methods:



```
# --- Plot all solutions together ---
plt.figure(figsize=(10, 6))
plt.plot(t, x_analytic, 'k--', label='Analytical', linewidth=2)
plt.plot(t_fe, x_fe[0, :], 'r', label='Explicit Euler')
plt.plot(t_be, x_be[0, :], 'b', label='Implicit Euler')
plt.plot(t_rk4, x_rk4[0, :], 'g', label='RK4')

plt.xlabel('Time (s)')
plt.ylabel('x(t)')
plt.title('Comparison of Analytical and Numerical Solutions')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



6. Discussion

The perfect agreement between the **RK4** and **analytical** results validates the correctness of the derived equation and the numerical integration setup. The discrepancy in Euler methods arises due to their lower-order accuracy and stability limits. The flat region at the beginning of the plot followed by exponential growth highlights the dynamic response of a system transitioning from near steady-state to an unstable or exponentially growing regime.

7. Conclusion

- The **analytical** , **Explicit Euler** and **RK4** numerical solutions are consistent, confirming the correctness of both derivation and implementation.
- The **Implicit Euler methods** provide approximate solutions but with noticeable differences depending on step size and stability properties.
- The observed **exponential growth after 8 seconds** is physically meaningful and matches the analytical behavior.
- Overall, the **RK4** and **Explicit Euler** method both are the most reliable for accurate simulation of this given second-order ODE within the given time frame of 10 seconds with step size of 0.01.