ИТМО

ИМИТАЦИОННОЕ МОДЕЛИРОВАНИЕ РОБОТОТЕХНИЧЕСКИХ СИСТЕМ

ОТЧЕТ

ПО ПРАКТИЧЕСКОЙ РАБОТЕ №1

Работу выполнил:

Сосенский Е.К.

Группа:

R4136c

Преподаватель:

Ракшин Е. А.

Санкт-Петербург

2025

Цель

Целью данной работы является решение ОДЕ используя явный метод Эйлера, неявный метод Эйлера и метод Рунге-Кутты. Также, в ходе решения с помощью программы, необходимо вывести получившиеся графики, а затем сравнить их с полученными в ходе построения Simulink модели для сравнения и дополнительной проверки.

Решение

Из задания возьмем указанное ОДЕ:

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

Затем возьмем из прикрепленной к заданию таблицы значения для своего варианта:

$$a = -9.3$$

$$b = -7.94$$

$$c = -8.83$$

$$d = -1.05$$

Приведем к актуальному виду и запишем в код:

```
theta_ddot = (d-b*theta_dot-c*theta)/a
```

Далее, после интегрирования наших значений и уравнения, перейдем к построению графиков:

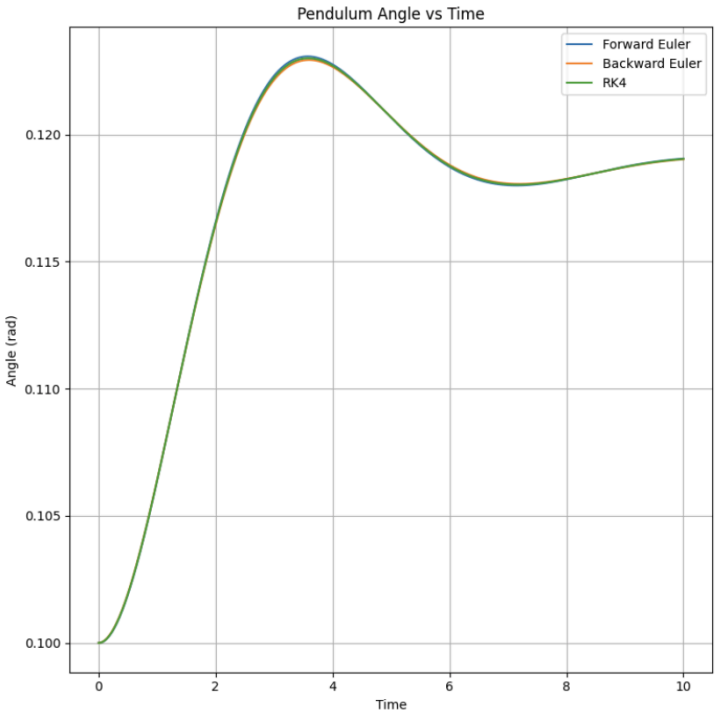

Рисунок 1 - График положения

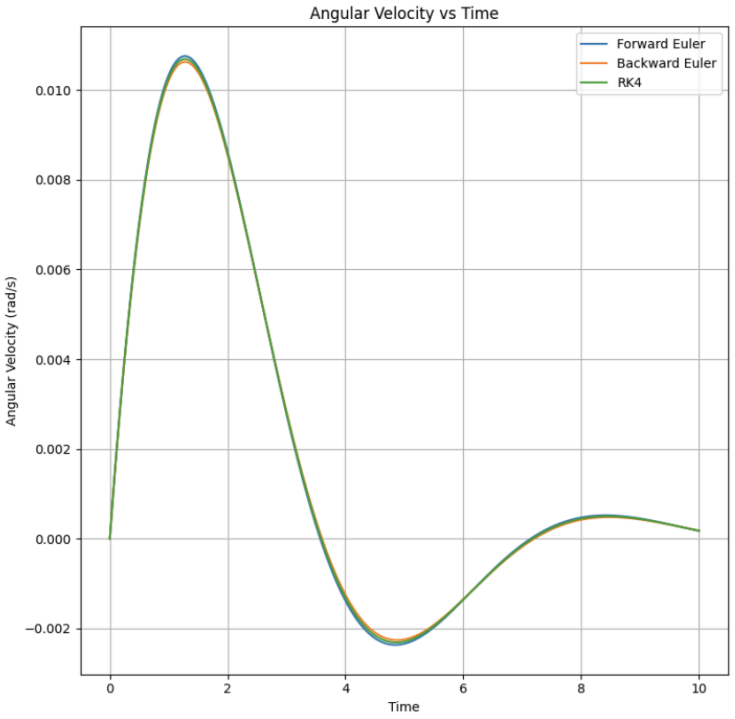

Рисунок 2 - График скорости

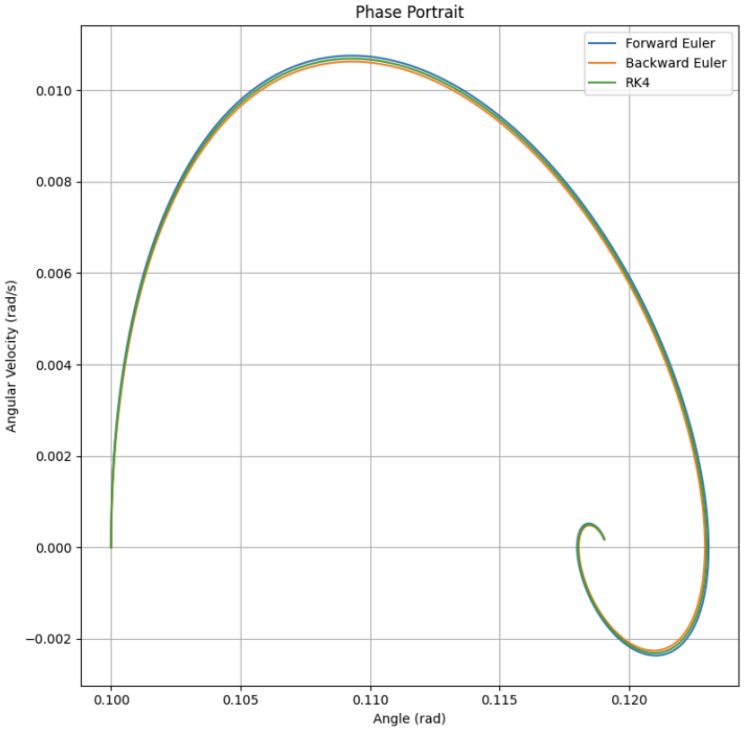

Рисунок 3 - График фазы

Теперь, когда графики построены, необходимо осуществить дополнительную проверку. Для этого построим схему в симулинке для решения данной задачи:

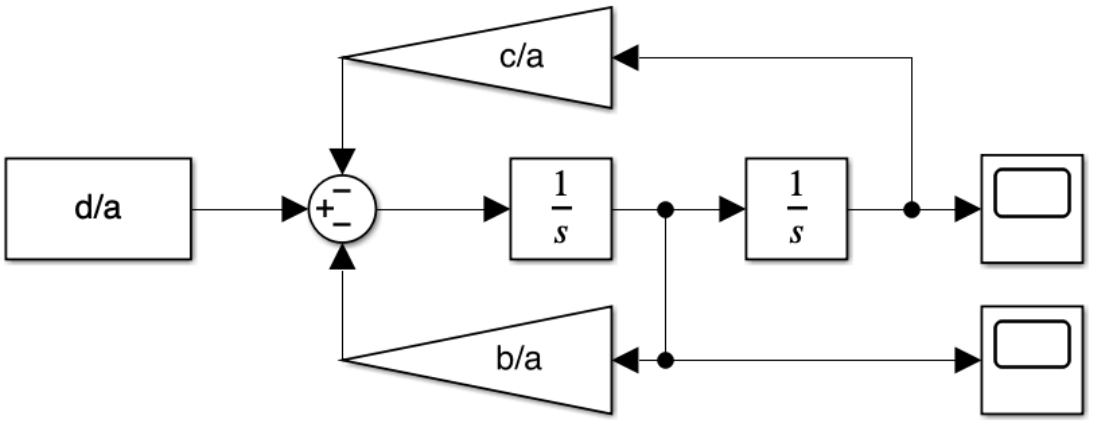

Рисунок 4 - Схема Simulink для решения ОДЕ

Запустив схему, получаем следующие графики:

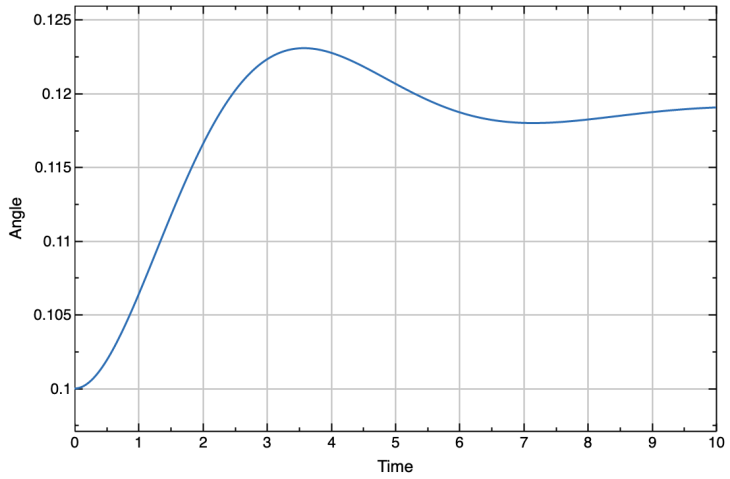

Рисунок 5 - График положения в Simulink

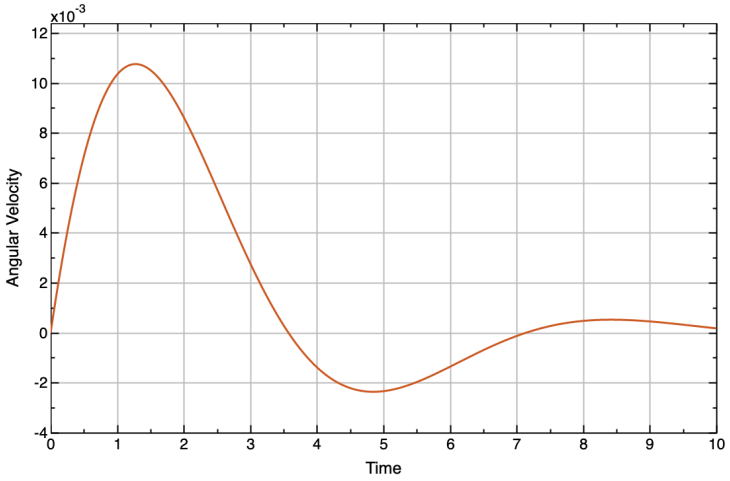

Рисунок 6 - График скорости в Simulink

Вывод

В ходе решения лабораторной работы были рассмотрены три метода решения ОДУ. Сравнение показало, что методы Эйлера являются менее точными по сравнению с методом Рунге-Кутты, так как имеют большую ошибку. По графикам, построенным с помощью среды Simulink, можно проверить достоверность программного решения, так как получившиеся графики совпадают с первоначальными. График фазы в Simulink не построен, так как он показывает зависимость скорости от положения, а значит при совпадении первых двух, он также будет совпадать.

Приложение А

Код Python для решения:

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


def pendulum_dynamics(x):
    """
    Pendulum dynamics: d²θ/dt² = -(g/l) * sin(θ)
    State vector x = [θ, dθ/dt]
    """
    a=-9.3
    b=-7.94
    c=-8.83
    d=-1.05


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


    theta_ddot = (d-b*theta_dot-c*theta)/a


    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

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.grid(True)
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.grid(True)
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.grid(True)
    plt.title('Phase Portrait')


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