МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»
(Университет ИТМО)

Факультет систем управления и робототехники

Практическая работа №1
по дисциплине
«Имитационное моделирование робототехнических систем»

ID 506527

Студен:
Группа R4136c *Воронцов К.В.*

Преподаватель:
Ассистент *Е.А. Ракшин*

Санкт-Петербург 2025 г.

ЦЕЛЬ РАБОТЫ.

- Проанализировать код и файла Integrators.ipunb;
- Решить в аналитическом виде ОДУ

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

Где *a, b, c, d* – коэффициенты из таблицы;

- Решить ОДУ численными методами используя функции из Integrators.ipunb;
- Сравнить результаты методов с аналитическим решением.

МЕТОДЫ ЧИСЛЕННОГО ИНТЕГРИРОВАНИЯ.

В файле реализованы функции 3х методов численного интегрирования, а именно явный метод Эйлера, неявный метод Эйлера и метод Рунге-Кутты.

Явный метод Эйлера — из известной начальной точки двигается вперёд по направлению касательной, вычисленной в этой точке. Метод позволяет явно рассчитать следующее значение, используя только информацию с предыдущего шага. Метод обладает первым порядком точности, что означает накопление значительной ошибки.

Неявный метод Эйлера для определения нового значения использует производную в будущей точке. Поскольку искомая величина присутствует в обеих частях уравнения, для его нахождения требуется решать нелинейное уравнение на каждом шаге.

Метод Рунге-Кутты 4-го порядка (RK4) – явный многошаговый метод, который для повышения точности вычисляет значение производной не в одной, а в четырёх промежуточных точках. Эти "пробные шаги" (k_1, k_2, k_3, k_4) затем комбинируются по специальной формуле с весами, давая усреднённое, более точное значение наклона.

АНАЛИТИЧЕСКОЕ РЕШЕНИЕ ОДУ ВТОРОГО ПОРЯДКА.

Требуется аналитическим методом решить ОДУ вида:

$$a \cdot \ddot{x} + b \cdot \dot{x} + c \cdot x = d. \quad (1)$$

Согласно таблице – a = –7.8, b = –1.07, c = 2.13, d = –7.63.

Решение будет производиться в общем виде. Для начала правая часть ОДУ приравнивается к нулю:

$$a \cdot \ddot{x} + b \cdot \dot{x} + c \cdot x = 0. \quad (2)$$

Далее составляется характеристический многочлен:

$$a \cdot \lambda^2 + b \cdot \lambda + c = 0;$$
$$\lambda_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \quad (3)$$
$$\lambda_1 = 0.458;$$
$$\lambda_2 = -0.595$$

Два действительных корня, общее решение ОДУ:

$$x = C_1 e^{\alpha t} + C_2 e^{\beta t}. \quad (4)$$

Где C_1 и C_2 – постоянные величины, $\alpha = 0.458$, $\beta = -0.595$.

Нахождение частного решения ОДУ производится согласно методу неопределенных коэффициентов:

$$\tilde{x} = A, \dot{\tilde{x}} = 0, \ddot{\tilde{x}} = 0,$$

$$a \cdot 0 + b \cdot 0 + cA = d.$$

$$\tilde{x} = A = \frac{d}{c}.$$

Тогда общее решение неоднородного ДУ:

$$x = C_1 e^{\alpha t} + C_2 e^{\beta t} + \frac{d}{c};$$

Характеристическое уравнение равно:

$$x = C_1 e^{0.458t} + C_2 e^{-0.595t} - 3.58; \quad (5)$$

Для определения C_1 и C_2 нужно определить начальные условия. Пусть $x(0) = 0.1, \dot{x}(0) = 0$. Тогда:

$$x(0) = C_1 + C_2 - 3.58 = 0.1;$$

Найдем производную

$$\dot{x}(t) = 0.458 * C_1 e^{0.458t} + -0.595 * C_2 e^{-0.595t}$$

При t = 0:

$$\dot{x}(0) = 0.458 * C_1 - 0.595 * C_2;$$

Решая систему из двух уравнений, получим:

$$C_1 = 2.080, C_2 = 1.601$$

Итоговое уравнение будет представлять собой следующее:

$$x(t) = 2.080e^{0.458t} + 1.601e^{-0.595t} - 3.58; \quad (7)$$

Как можно видеть система не устойчива, первое слагаемое экспоненциально растет, второе слагаемое затухает.

Также дальнейшее решение можно подтвердить и через симуляцию в программе Matlab Simulink.

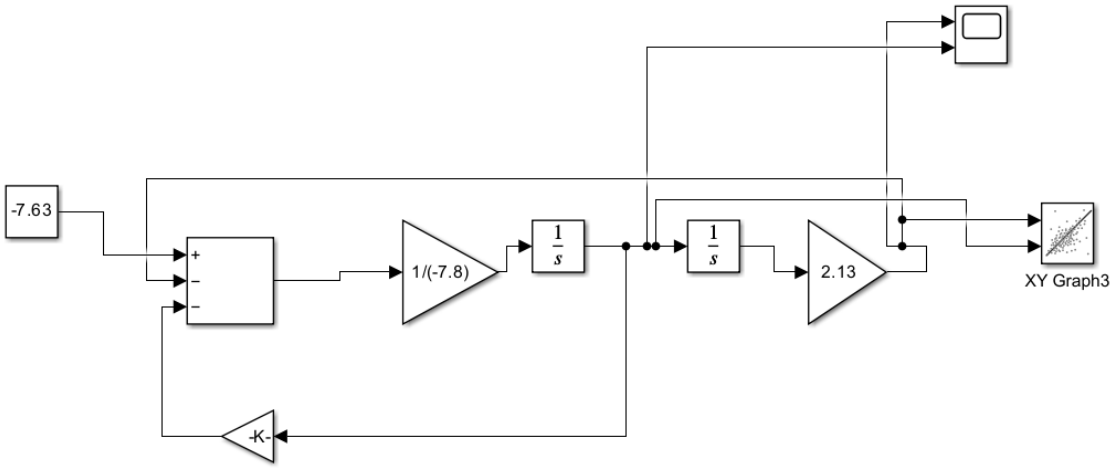

Рисунок 1 – Схема модели

Собираем данную схему и можем получить переходный процесс и фазовый портрет.

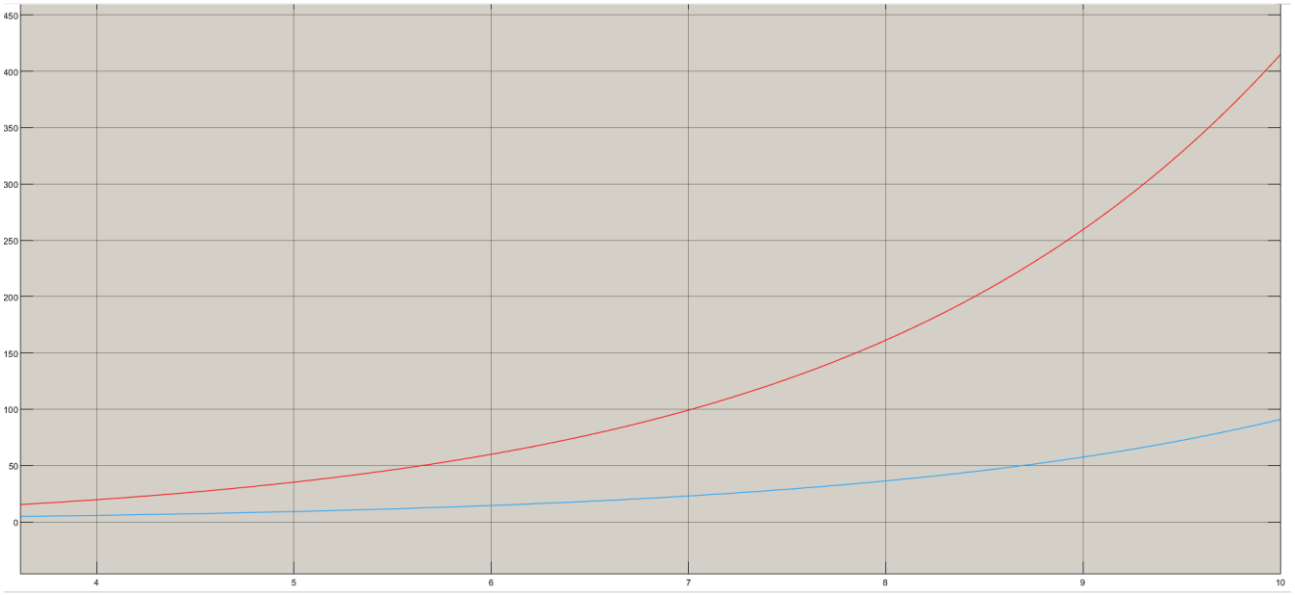

Рисунок 2 – Переходный процесс (синяя график – скорость), (красный график – положение)

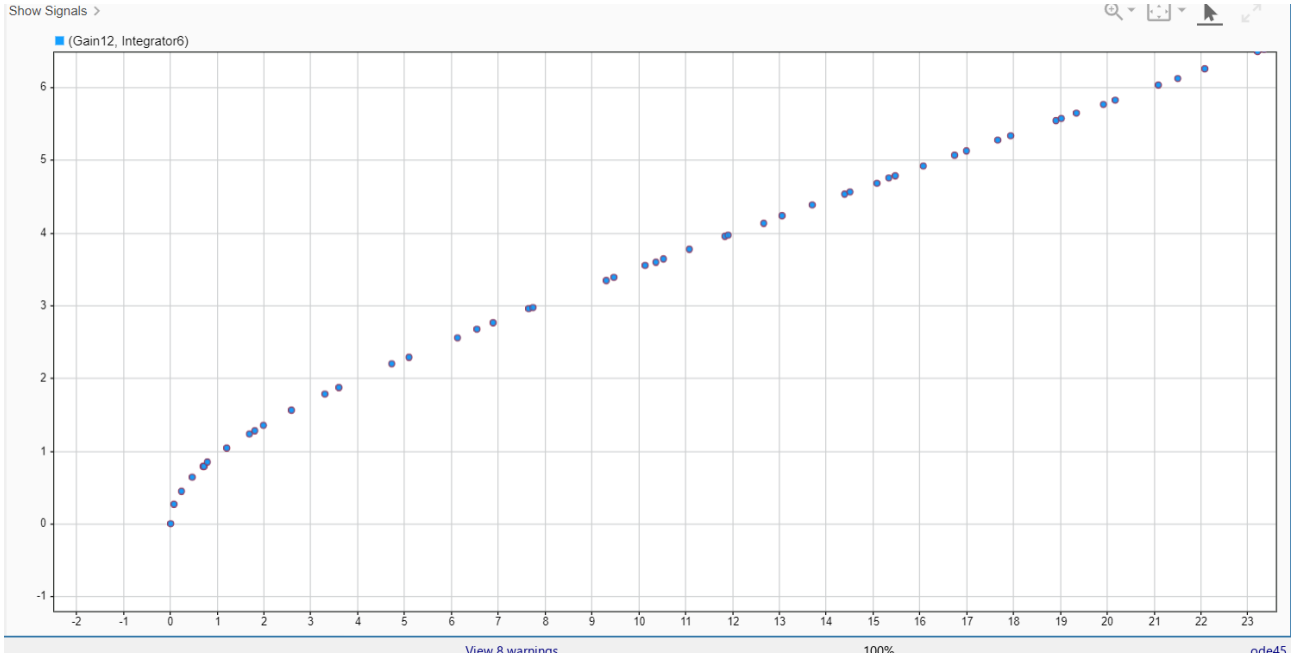

Рисунок 3 – Фазовый портрет (неустойчивая система)

РЕШЕНИЕ ОДУ ВТОРОГО ПОРЯДКА ЧИСЛЕННЫМИ МЕТОДАМИ.

Функция *dynamics* из файла Integrators.ipunb была изменена для решения ОДУ из формулы 1.

```python

import numpy as np
import matplotlib.pyplot as plt

def dynamics(X):

    a = -7.8
    b = -1.07
    c = 2.13
    d = -7.63

    x1 = X[0]  # текущее положение
    x2 = X[1]  # текущая скорость

    dx1dt = x2  # производная положения = скорость
    dx2dt = (d - b * x2 - c * x1) / a

    return np.array([dx1dt, dx2dt])

```

Рисунок 4 – функция *dynamics*

В целом все остальные функции остались без изменения.

```python
    """
    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)
```

```python
# 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()
```

Рисунок 5 – зависимость положения от времени

Ниже приведены зависимости, которые были получены построением через библиотеку matplotlib.

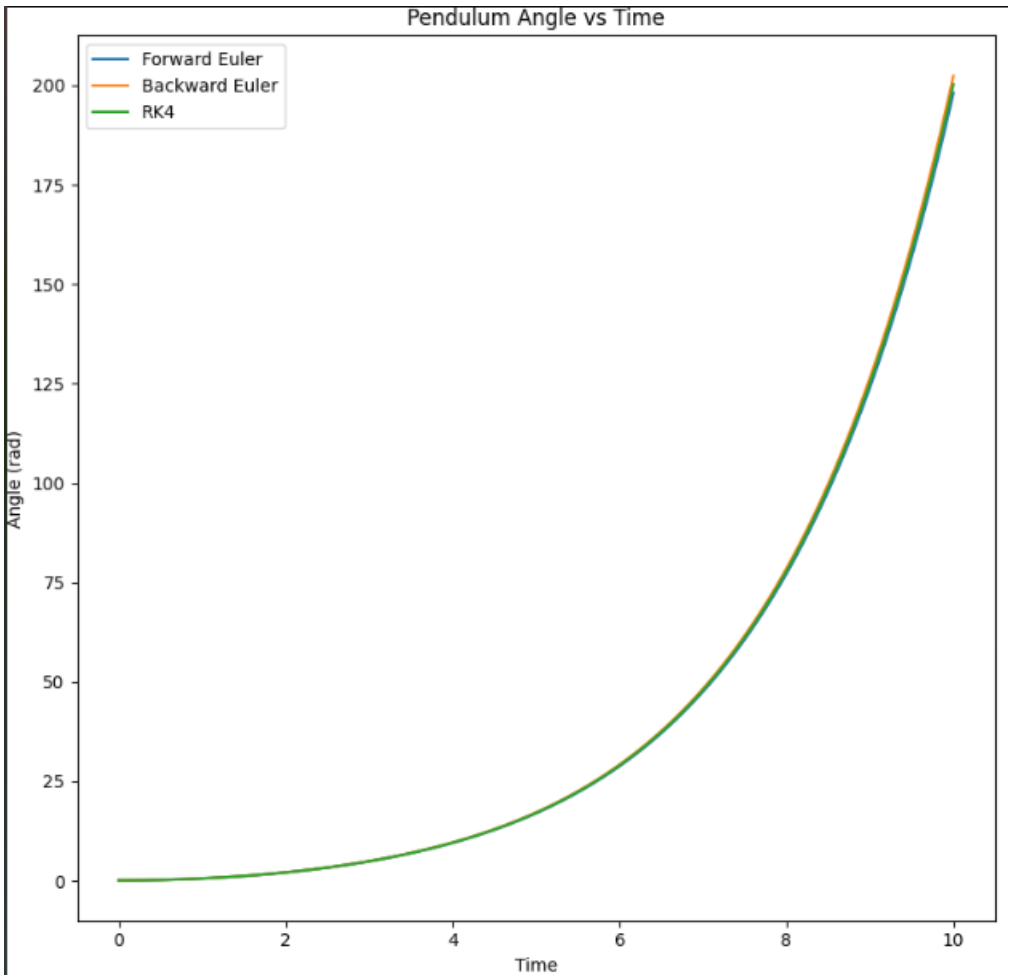

Рисунок 6 – зависимость положения от времени

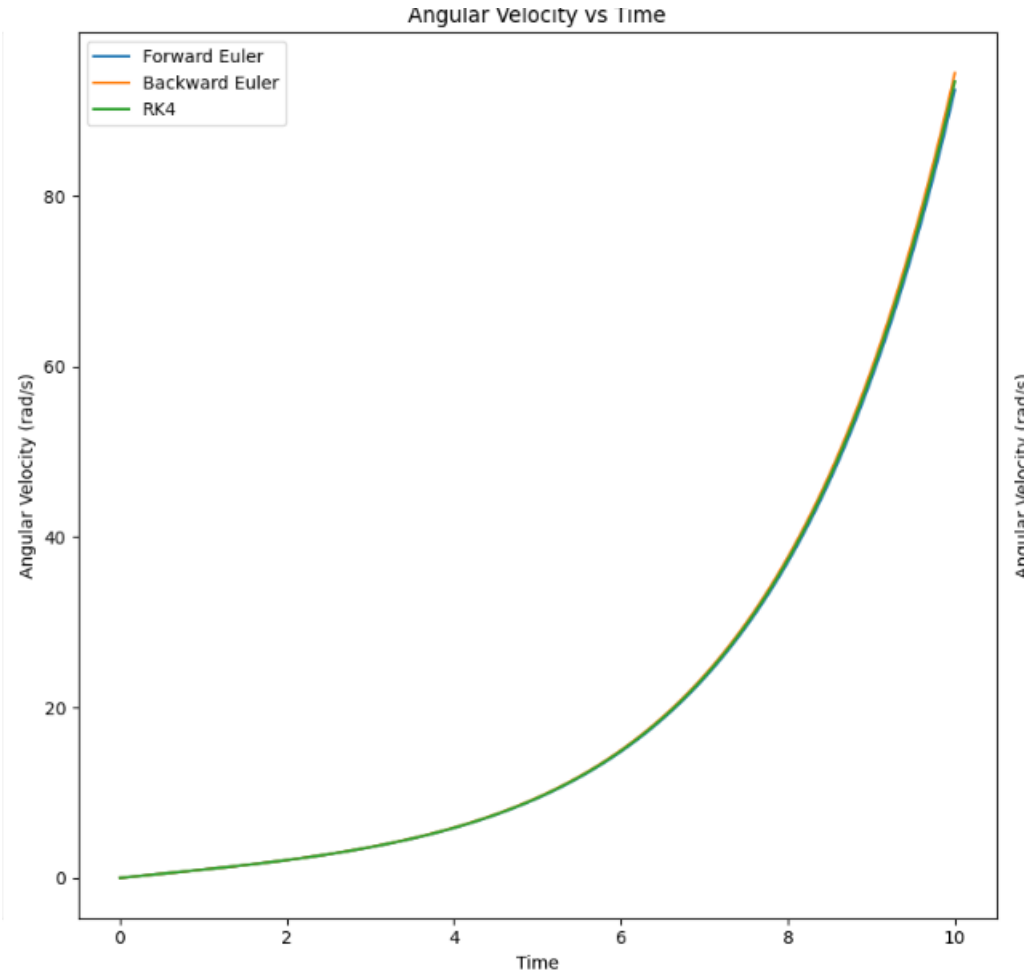

Рисунок 7 – зависимость скорости от времени

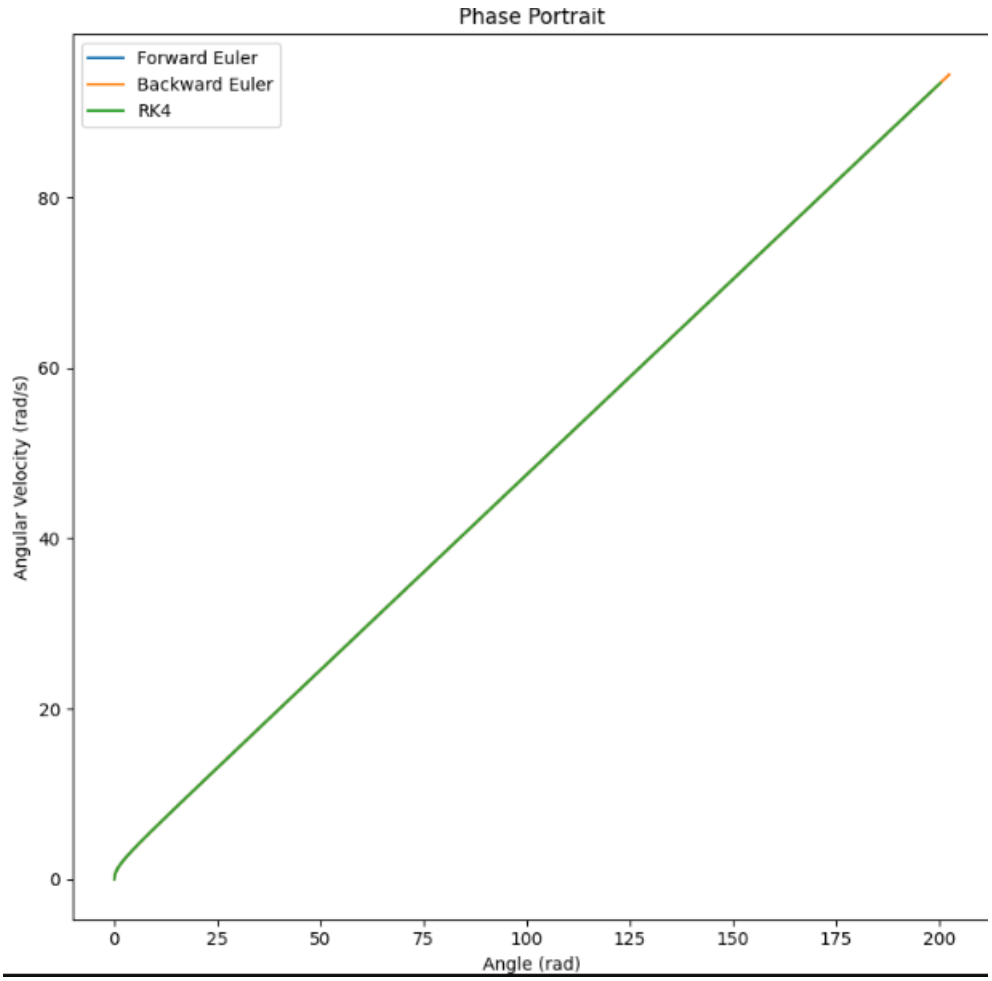

Рисунок 8 – Фазовый портрет (система неустойчива)

Как было ранее показано моделированием в Simulink фазовые портреты (соответствует неустойчивому узлу (поскольку корни действительные)) и графики переходных процессов совпадают соответственно моделирование проведено верно.

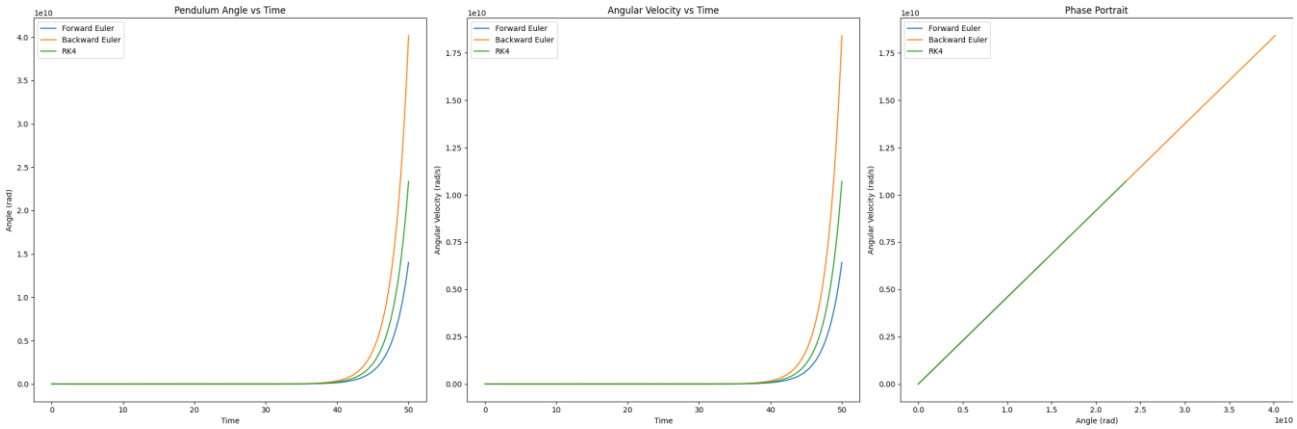

Рисунок 9 – Увеличенный шаг до 0.1

Для исследования также можно увеличить шаг h = 0.1 и результаты будут в явном виде говорить о точности каждого из методов, например по данному рисунку можно видеть что метод Рунге – Кутты дает средние результаты.

Напишем код для проверки аналитического решения. Разделим на две функции *analytical_solution(t) и analytical_velocity(t).*

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

def analytical_solution(t):

    C1 = 2.080
    C2 = 1.601
    r1 = 0.45845
    r2 = -0.59565
    x_eq = -3.582
    return C1 * np.exp(r1 * t) + C2 * np.exp(r2 * t) + x_eq

def analytical_velocity(t):
    C1 = 2.080
    C2 = 1.601
    r1 = 0.45845
    r2 = -0.59565
    return C1 * r1 * np.exp(r1 * t) + C2 * r2 * np.exp(r2 * t)

t = np.linspace(0, 10, 1000)


x_anal = analytical_solution(t)
v_anal = analytical_velocity(t)

plt.figure(figsize=(18, 5))
```

```
27
28  plt.subplot(1, 3, 1)
29  plt.plot(t, x_anal, 'k-', linewidth=2, label='Analytical')
30  plt.xlabel('Time')
31  plt.ylabel('x(t)')
32  plt.title('Angle vs Time')
33  plt.legend()
34  plt.grid(True)
35
36  plt.subplot(1, 3, 2)
37  plt.plot(t, v_anal, 'k-', linewidth=2, label='Analytical')
38  plt.xlabel('Time')
39  plt.ylabel("x'(t)")
40  plt.title('Angular Velocity vs Time')
41  plt.legend()
42  plt.grid(True)
43
44
45  plt.subplot(1, 3, 3)
46  plt.plot(x_anal, v_anal, 'k-', linewidth=2, label='Analytical')
47  plt.xlabel('x(t)')
48  plt.ylabel("x'(t)")
49  plt.title('Phase Portrait')
50  plt.legend()
51  plt.grid(True)
52
53  plt.tight_layout()
54  plt.show()
55
```

Рисунок 10 – Код для аналитического решения

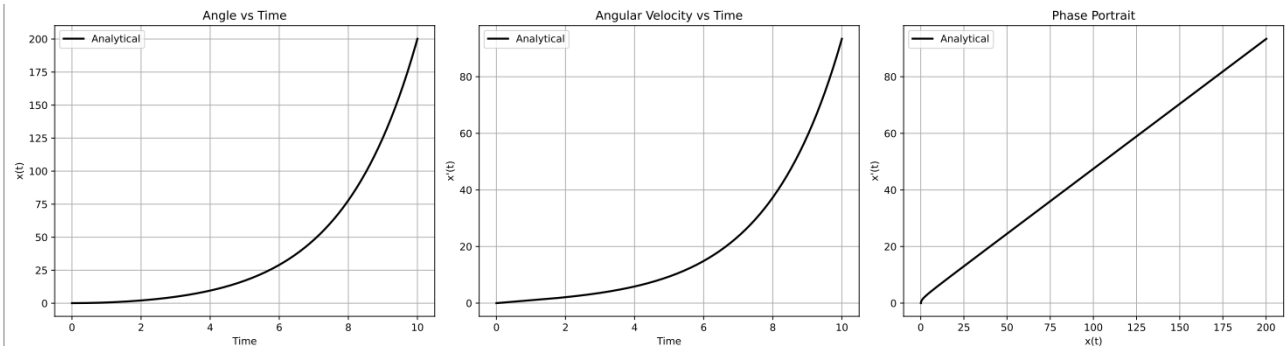

Рисунок 11 – График аналитического решения

Метод Рунге-Кутты 4-го порядка показал наилучшую точность и плавность траектории.

Явный метод Эйлера дал близкий результат, но менее точный (особенно при больших временах).

Неявный метод Эйлера, несмотря на свою устойчивость, в данном случае не проявил преимуществ, так как система неустойчива, и искусственная диссипация не влияет на общий рост.

ВЫВОД.

Согласно полученным данным и визуальной оценкой, можно сделать вывод о том, что при малом шаге интегрирования каждый из способов численного интегрирования дает приблизительно точные результаты, но с увеличением шага растет расхождение. Из всех приведенных методов численного интегрирования метод Рунге – Кутты является точным, но более затратным для расчетов.