МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Университет ИТМО

дисциплина

«Имитационное моделирование робототехнических систем»

Отчет по лабораторной работе № 1

Выполнил:
Смирнов И. Д. R4136c

Проверил:
Ракшин Е. А.

Санкт-Петербург

2025

Задание

1) Решить аналитически ОДУ в виде:

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

где $a = -8.77$ $b = -9.69$ $c = -1.99$ $d = -2.85$

2) Решить ОДУ с помощью трех интеграторов: явного/ неявного методов Эйлера, Рунге-Кутты.

3) Провести эксперимент

4) Сравнить результаты этих методов с аналитическим решением.

Ход работы

1.Аналитическое решение

Исходное уравнение

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

где $a = -8.77$ $b = -9.69$ $c = -1.99$ $d = -2.85$

В характеристическом виде

$-8.77 \cdot \lambda^2 - 9.69 \cdot \lambda - 1.99 = 0$

Дискриминант

$D = b^2 - 4ac = 93.8961 - 69.8092 = 24.0869$

$\lambda_{1,2} = \frac{-b \pm \sqrt{D}}{2a}$

$\lambda_1 = -0,832 \qquad \lambda_2 = -0,273$

Однородное решение

$x(t) = C_1 e^{-0,832t} + C_2 e^{-0,273t}$

Частное решение

$c\bar{x} = d$

$\bar{x} = \frac{d}{c} = \frac{-2,85}{-1,99} = 1.432$

Общее решение

$$x(t) = C_1 e^{-0,832t} + C_2 e^{-0,273t} + 1,432$$

Зададим начальные условия $x(0) = 0,1$; $\dot{x}(0) = 0$

$$\begin{matrix} C_1 + C_2 + 1,432 = 0,1 \\ -0,832\,C_1 - 0,273C_2 + 1, \end{matrix} \Rightarrow \begin{matrix} C_1 = -0,032 \\ C_2 = -1,3 \end{matrix}$$

Итоговое решение:

$$x(t) = -0,032e^{-0,832t} - 1,3e^{-0,273t} + 1,432$$

---

2. Решение численными методами

Зададим $y_1 = x,\ y_2 = \dot{x}$. тогда

$$\dot{y_1} = y_2, \quad \dot{y_2} = \frac{d - cy_1 - by_2}{a}$$

2.1. Явный метод эйлера

$$y_i = y_{i-1} + hf(t_{i-1},\ y_{i-1})$$

2.2 Неявный метод эйлера

$$y_i = y_{i-1} + hf(t_i,\ y_i)$$

2.3 Метод Рунге-Кутта

$$k_1 = f(t_{i-1}, y_{i-1})$$

$$k_2 = f(t_{i-1} + \frac{h}{2}, y_{i-1} + \frac{h}{2}k_1)$$

$$k_3 = f(t_{i-1} + \frac{h}{2}, y_{i-1} + \frac{h}{2}k_2)$$

$$k_4 = f(t_{i-1} + h, y_{i-1} + hk_3)$$

$$y_i = y_{i-1} + \frac{h}{6}(k_! + 2k_2 + 2k_3 + k_4)$$

---

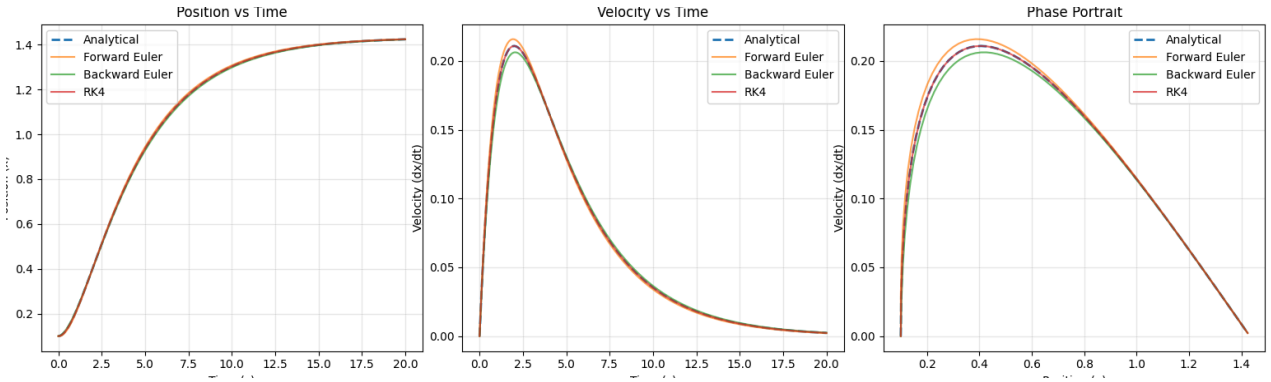
3. Эксперимент:

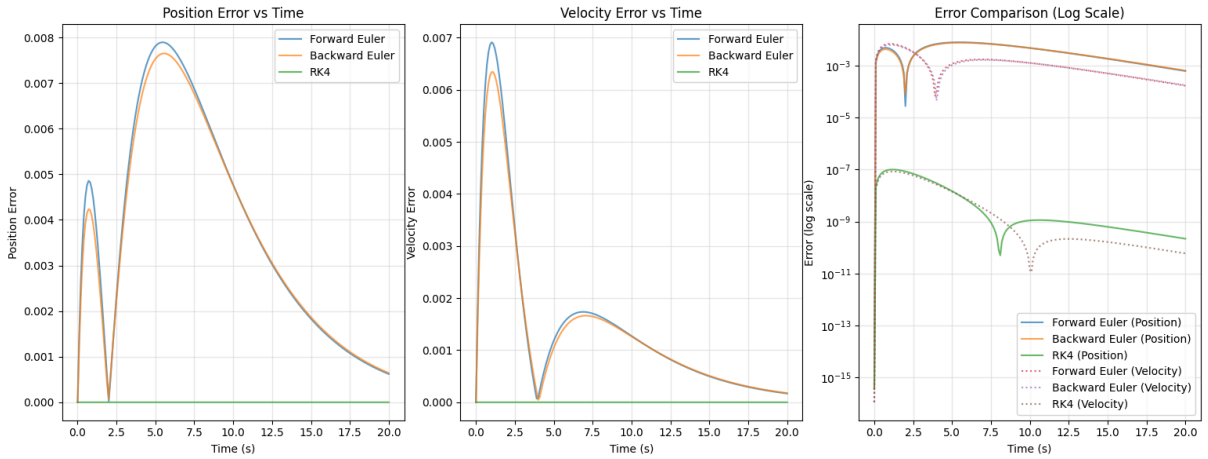
Параметры эксперимента:

```
#коэффициенты уравнения
a = -8.77
b = -9.69
c = -1.99
d = -2.85

# параметры моделирования
Tf = 20.0 # время моделирования
h = 0.1 # шаг интегрирования
```

Результат:





Статистика ошибок:

Forward Euler - Max error: 0.006910, Mean error: 0.001466
Backward Euler - Max error: 0.006340, Mean error: 0.001403
RK4 - Max error: 0.000000, Mean error: 0.000000

Вывод

RK4 показал исключительную точность, ошибки отсутствуют. В то время как явный и неявный методы Эйлера имеют сопоставимые ошибки, которые значительно выше, чем у RK4. Это соответствует теории, что это методы первого порядка точности.

RK4 настолько точно аппроксимирует аналитическое решение, что численная погрешность в пределах машинной точности. Явный и неявный методы Эйлера отклоняются от аналитического решения, что видно по накоплению ошибки со временем.

Для решения линейного дифференциального уравнения второго порядка с высокой точностью предпочтительно использовать метод RK4. Явный и неявный методы Эйлера подходят для более простых задач или при необходимости минимизировать вычислительные затраты, но с учетом пониженной точности. Полученные результаты подтверждают важность выбора численного метода в зависимости от требований к точности и вычислительным ресурсам.

Листинги программ:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy.optimize import йлерafsolve

def oscillator_dynamics(x):
    """
    Dynamics for oscillator: a*x_ddot + b*x_dot + c*x = d
    Rewritten as: x_ddot = (d - b*x_dot - c*x) / a
    State vector x = [x, x_dot]
    """
    a = -8.77
    b = -9.69
    c = -1.99
    d = -2.85

    x_val = x[0]
    x_dot = x[1]
```

```
    x_ddot = (d - b*x_dot - c*x_val) / a

    return np.array([x_dot, x_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

def analytical_solution(t, x0, x0_dot):
    """
    Analytical solution for a*x_ddot + b*x_dot + c*x = d
    """
    a = -8.77
    b = -9.69
```

```
    c = -1.99
    d = -2.85

    # Calculate the steady-state solution
    x_ss = d / c  # The equilibrium position

    # Characteristic equation: a*r^2 + b*r + c = 0
    discriminant = b**2 - 4*a*c

    if discriminant > 0:
        # Two distinct real roots
        r1 = (-b + np.sqrt(discriminant)) / (2*a)
        r2 = (-b - np.sqrt(discriminant)) / (2*a)

        # Solve for constants using initial conditions
        A = (x0_dot - r2*(x0 - x_ss)) / (r1 - r2)
        B = (r1*(x0 - x_ss) - x0_dot) / (r1 - r2)

        x = x_ss + A*np.exp(r1*t) + B*np.exp(r2*t)
        x_dot = A*r1*np.exp(r1*t) + B*r2*np.exp(r2*t)

    elif discriminant == 0:
        # One repeated real root
        r = -b / (2*a)

        # Solve for constants using initial conditions
        A = x0 - x_ss
        B = x0_dot - r*A

        x = x_ss + (A + B*t) * np.exp(r*t)
        x_dot = (B + r*(A + B*t)) * np.exp(r*t)

    else:
        # Complex roots
        alpha = -b / (2*a)
        beta = np.sqrt(-discriminant) / (2*a)

        # Solve for constants using initial conditions
        A = x0 - x_ss
        B = (x0_dot - alpha*A) / beta

        x = x_ss + np.exp(alpha*t) * (A*np.cos(beta*t) + B*np.sin(beta*t))
        x_dot = np.exp(alpha*t) * ((alpha*A + beta*B)*np.cos(beta*t) + (alpha*B -
beta*A)*np.sin(beta*t))

    return x, x_dot

# Parameters
a = -8.77
b = -9.69
c = -1.99
d = -2.85

# Initial conditions
x0_analytical = 0.1  # Initial position
x0_dot_analytical = 0.0  # Initial velocity
x0 = np.array([x0_analytical, x0_dot_analytical])

# Time parameters
Tf = 10.0
h = 0.01

# Solve using different methods
```

```
x_fe, t_fe = forward_euler(oscillator_dynamics, x0, Tf, h)
x_be, t_be = backward_euler(oscillator_dynamics, x0, Tf, h)
x_rk4, t_rk4 = runge_kutta4(oscillator_dynamics, x0, Tf, h)

# Analytical solution
t_analytical = np.linspace(0, Tf, len(t_fe))
x_analytical, x_dot_analytical = analytical_solution(t_analytical, x0_analytical, x0_dot_analytical)

# Plotting
plt.figure(figsize=(24, 8))

# Plot 1: Position vs Time
plt.subplot(1, 3, 1)
plt.plot(t_analytical, x_analytical, label='Analytical', linewidth=2, linestyle='--')
plt.plot(t_fe, x_fe[0, :], label='Forward Euler', alpha=0.7)
plt.plot(t_be, x_be[0, :], label='Backward Euler', alpha=0.7)
plt.plot(t_rk4, x_rk4[0, :], label='RK4', alpha=0.7)
plt.xlabel('Time (s)')
plt.ylabel('Position (x)')
plt.legend()
plt.title('Position vs Time')
plt.grid(True, alpha=0.3)

# Plot 2: Velocity vs Time
plt.subplot(1, 3, 2)
plt.plot(t_analytical, x_dot_analytical, label='Analytical', linewidth=2, linestyle='--')
plt.plot(t_fe, x_fe[1, :], label='Forward Euler', alpha=0.7)
plt.plot(t_be, x_be[1, :], label='Backward Euler', alpha=0.7)
plt.plot(t_rk4, x_rk4[1, :], label='RK4', alpha=0.7)
plt.xlabel('Time (s)')
plt.ylabel('Velocity (dx/dt)')
plt.legend()
plt.title('Velocity vs Time')
plt.grid(True, alpha=0.3)

# Plot 3: Phase Portrait
plt.subplot(1, 3, 3)
plt.plot(x_analytical, x_dot_analytical, label='Analytical', linewidth=2, linestyle='--')
plt.plot(x_fe[0, :], x_fe[1, :], label='Forward Euler', alpha=0.7)
plt.plot(x_be[0, :], x_be[1, :], label='Backward Euler', alpha=0.7)
plt.plot(x_rk4[0, :], x_rk4[1, :], label='RK4', alpha=0.7)
plt.xlabel('Position (x)')
plt.ylabel('Velocity (dx/dt)')
plt.legend()
plt.title('Phase Portrait')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Calculate errors
# Interpolate numerical solutions to match analytical time grid
def interpolate_solution(t_num, x_num, t_target):
    """Interpolate numerical solution to match target time grid"""
    x_interp = np.interp(t_target, t_num, x_num)
    return x_interp

# Interpolate all numerical solutions to analytical time grid
x_fe_interp = interpolate_solution(t_fe, x_fe[0, :], t_analytical)
x_be_interp = interpolate_solution(t_be, x_be[0, :], t_analytical)
x_rk4_interp = interpolate_solution(t_rk4, x_rk4[0, :], t_analytical)

x_fe_dot_interp = interpolate_solution(t_fe, x_fe[1, :], t_analytical)
```

```
x_be_dot_interp = interpolate_solution(t_be, x_be[1, :], t_analytical)
x_rk4_dot_interp = interpolate_solution(t_rk4, x_rk4[1, :], t_analytical)

# Calculate errors
error_pos_fe = np.abs(x_fe_interp - x_analytical)
error_pos_be = np.abs(x_be_interp - x_analytical)
error_pos_rk4 = np.abs(x_rk4_interp - x_analytical)

error_vel_fe = np.abs(x_fe_dot_interp - x_dot_analytical)
error_vel_be = np.abs(x_be_dot_interp - x_dot_analytical)
error_vel_rk4 = np.abs(x_rk4_dot_interp - x_dot_analytical)

# Print error statistics
print("Error Statistics (Position):")
print(f"Forward Euler - Max error: {np.max(error_pos_fe):.6f}, Mean error:
{np.mean(error_pos_fe):.6f}")
print(f"Backward Euler - Max error: {np.max(error_pos_be):.6f}, Mean error:
{np.mean(error_pos_be):.6f}")
print(f"RK4 - Max error: {np.max(error_pos_rk4):.6f}, Mean error: {np.mean(error_pos_rk4):.6f}")

print("\nError Statistics (Velocity):")
print(f"Forward Euler - Max error: {np.max(error_vel_fe):.6f}, Mean error:
{np.mean(error_vel_fe):.6f}")
print(f"Backward Euler - Max error: {np.max(error_vel_be):.6f}, Mean error:
{np.mean(error_vel_be):.6f}")
print(f"RK4 - Max error: {np.max(error_vel_rk4):.6f}, Mean error: {np.mean(error_vel_rk4):.6f}")

# Plot errors
plt.figure(figsize=(18, 6))

plt.subplot(1, 3, 1)
plt.plot(t_analytical, error_pos_fe, label='Forward Euler', alpha=0.7)
plt.plot(t_analytical, error_pos_be, label='Backward Euler', alpha=0.7)
plt.plot(t_analytical, error_pos_rk4, label='RK4', alpha=0.7)
plt.xlabel('Time (s)')
plt.ylabel('Position Error')
plt.legend()
plt.title('Position Error vs Time')
plt.grid(True, alpha=0.3)

plt.subplot(1, 3, 2)
plt.plot(t_analytical, error_vel_fe, label='Forward Euler', alpha=0.7)
plt.plot(t_analytical, error_vel_be, label='Backward Euler', alpha=0.7)
plt.plot(t_analytical, error_vel_rk4, label='RK4', alpha=0.7)
plt.xlabel('Time (s)')
plt.ylabel('Velocity Error')
plt.legend()
plt.title('Velocity Error vs Time')
plt.grid(True, alpha=0.3)

plt.subplot(1, 3, 3)
plt.semilogy(t_analytical, error_pos_fe, label='Forward Euler (Position)', alpha=0.7)
plt.semilogy(t_analytical, error_pos_be, label='Backward Euler (Position)', alpha=0.7)
plt.semilogy(t_analytical, error_pos_rk4, label='RK4 (Position)', alpha=0.7)
plt.semilogy(t_analytical, error_vel_fe, label='Forward Euler (Velocity)', alpha=0.7, linestyle=':')
plt.semilogy(t_analytical, error_vel_be, label='Backward Euler (Velocity)', alpha=0.7,
linestyle=':')
plt.semilogy(t_analytical, error_vel_rk4, label='RK4 (Velocity)', alpha=0.7, linestyle=':')
plt.xlabel('Time (s)')
plt.ylabel('Error (log scale)')
plt.legend()
plt.title('Error Comparison (Log Scale)')
plt.grid(True, alpha=0.3)
```

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