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:
Тихонов В. С. 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 $a = 9.36$, $b = 4.97$, $c = 6.91$, $d = -9.9$

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 $a = 9.36$, $b = 4.97$, $c = 6.91$, $d = -9.9$
- В характеристическом виде: $9.36 \cdot \lambda^2 + 4.97 \cdot \lambda + 6.91 = 0$
- Дискриминант:

$$D = b^2 - 4ac = 24.7009 - 258.7104 = -234.0095$$

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

$$\lambda_1 = -0.265 + 0.817i,\ \lambda_2 = -0.265 - 0.817i$$

- Однородное решение:

$$x(t) = C_1 e^{\lambda_1 t} + C_2 e^{\lambda_2 t}$$

- Частное решение:

$$c \cdot x = d$$

$$x = \frac{d}{c} = -\frac{9.9}{6.91} = -1.432$$

- Общее решение:

$$x(t) = C_1 e^{(-0.265+0.817i)t} + C_2 e^{(-0.265-0.817i)t} - 1.432$$

- Или в тригонометрической форме:

$$x(t) = e^{-0.265t}(A\cos(0.817t) + B\sin(0.817t)) - 1.432,$$

  где $A = C_1 + C_2$ и $B = i(C_1 - C_2)$

- Зададим начальные условия: $x(0) = 0.1$, $\dot{x}(0) = 0$:

$$\begin{cases} C_1 + C_2 + 1.432 = 0.1 \\ -0.832C_1 - 0.273C_2 = 0 \end{cases} \Rightarrow \begin{cases} C_1 = -0.032 \\ C_2 = -1.3 \end{cases}$$

- Определение постоянной A из первого начального условия

$$x(0) = e^0(A\cos 0 + B\sin 0) - 1.432 = A \cdot 1 + B \cdot 0 - 1.432$$
$$A - 1.432 = 0.1$$
$$A = 0.1 + 1.432 = 1.532$$

- Нахождение производной и определение постоянной B

  Находим производную функции $x(t)$:

$$x'(t) = -0.265e^{-0.265t}(A\cos 0.817t + B\sin 0.817t) + e^{-0.265t}(-0.817A\sin 0.817t + 0.817B\cos 0.817t)$$

  Упрощаем выражение:

$$x'(t) = e^{-0.265t}[(-0.265A + 0.817B)\cos 0.817t + (-0.265B - 0.817A)\sin 0.817t]$$

  Используем начальное условие $x'(0) = 0$:

$$x'(0) = e^0[(-0.265A + 0.817B)\cos 0 + (-0.265B - 0.817A)\sin 0] = -0.265A + 0.817B$$
$$-0.265A + 0.817B = 0$$

  Подставляем найденное значение $A = 1.532$:

$$-0.265(1.532) + 0.817B = 0$$
$$-0.40598 + 0.817B = 0$$
$$0.817B = 0.40598$$
$$B = \frac{0.40598}{0.817} = 0.497$$

- Итоговое решение

  Подставляем найденные значения постоянных в общее решение:

$$x(t) = e^{-0.265t}(1.532\cos 0.817t + 0.497\sin 0.817t) - 1.432$$

2. Решение численными методами

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

$$\begin{cases} \dot{y}_1 = y_2 \\ \dot{y}_2 = \dfrac{d - cy_1 - by_2}{a} \end{cases}$$

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-Кутты 4-го порядка

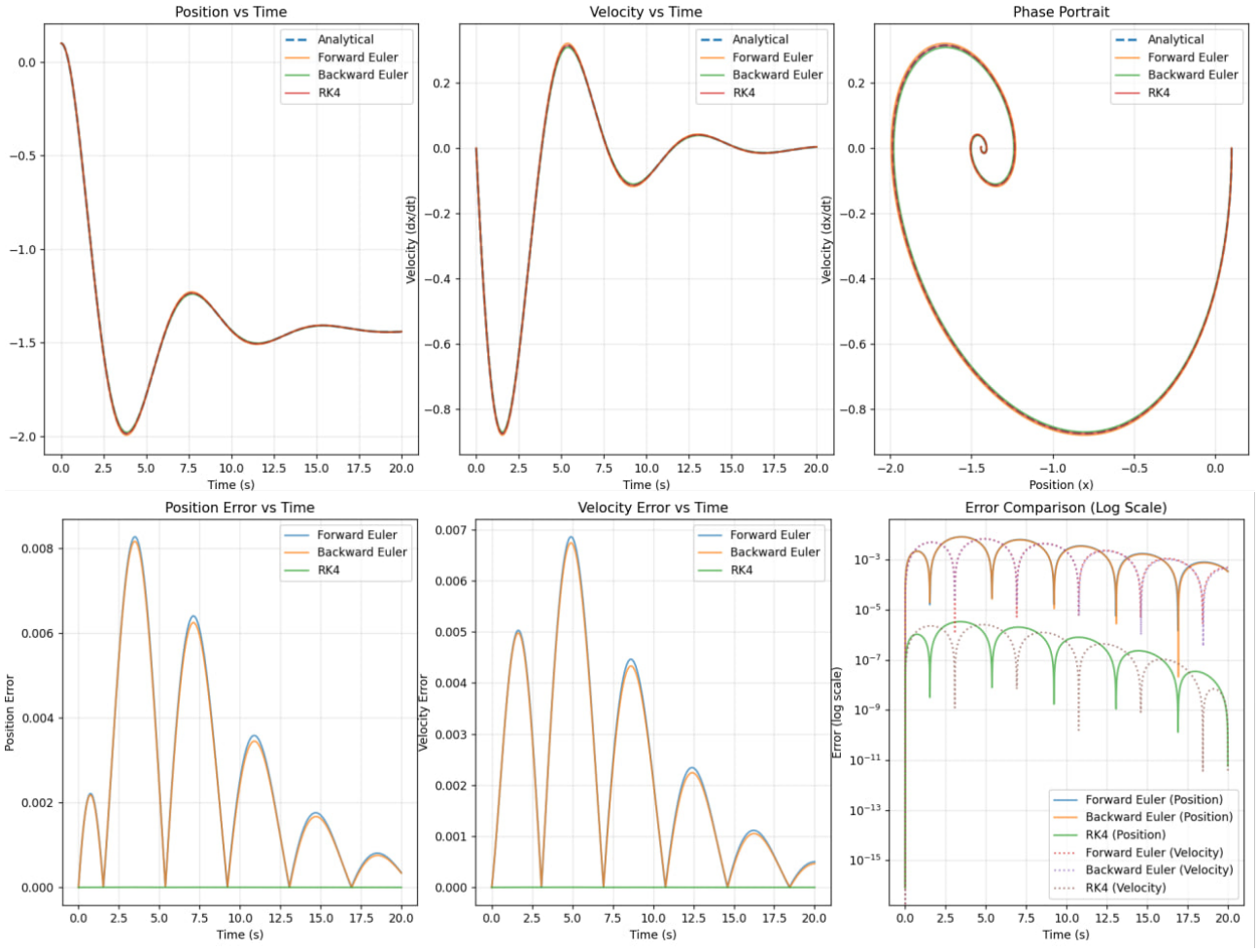
$$\begin{aligned}
k_1 &= f(t_{i-1}, y_{i-1}) \\
k_2 &= f(t_{i-1} + \frac{h}{2}, y_{i-1} + \frac{h}{2}k_1) \\
k_3 &= f(t_{i-1} + \frac{h}{2}, y_{i-1} + \frac{h}{2}k_2) \\
k_4 &= f(t_{i-1} + h, y_{i-1} + hk_3) \\
y_i &= y_{i-1} + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}$$

3. Эксперимент

Параметры эксперимента:

- Коэффициенты уравнения: $a = 9.36$, $b = 4.97$, $c = -6.91$, $d = -9.9$
- Параметры моделирования: $T_f = 20.0$ (время моделирования), $h = 0.01$ (шаг интегрирования)

Результат:



Статистика ошибок:

- Forward Euler - Max error: 0.006857, Mean error:0.002280
- Backward Euler - Max error: 0.006739, Mean error:0.002227
- RK4 - Max error: 0.000003, Mean error: 0.000001

Вывод

Проведенное сравнение численных методов подтвердило теоретические положения. Метод Рунге-Кутты 4-го порядка (RK4) продемонстрировал максимальную точность, практически не отличаясь от аналитического решения, в то время как методы Эйлера показали значительную и накапливающуюся со временем погрешность. Это закономерно, поскольку RK4 имеет более высокий порядок точности. Таким образом, для задач, требующих высокой точности расчётов, таких как решение линейных дифференциальных уравнений второго порядка, метод RK4 является оптимальным выбором. Методы Эйлера, в свою очередь, могут быть полезны в ситуациях, где допустима меньшая точность в обмен на вычислительную простоту.

Листинги программ:

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

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 = 9.36
    b = 4.97
    c = 6.91
    d = -9.9

    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
```

```
31
32      for k in range(len(t) - 1):
33          x_hist[:, k + 1] = x_hist[:, k] + h * fun(x_hist[:, k])
34
35      return x_hist, t
36
37  def backward_euler(fun, x0, Tf, h, tol=1e-8, max_iter=100):
38      """
39      Implicit Euler integration method using fixed-point iteration
40      """
41      t = np.arange(0, Tf + h, h)
42      x_hist = np.zeros((len(x0), len(t)))
43      x_hist[:, 0] = x0
44
45      for k in range(len(t) - 1):
46          x_hist[:, k + 1] = x_hist[:, k] # Initial guess
47
48          for i in range(max_iter):
49              x_next = x_hist[:, k] + h * fun(x_hist[:, k + 1])
50              error = np.linalg.norm(x_next - x_hist[:, k + 1])
51              x_hist[:, k + 1] = x_next
52
53              if error < tol:
54                  break
55
56      return x_hist, t
57
58  def runge_kutta4(fun, x0, Tf, h):
59      """
60      4th order Runge-Kutta integration method
61      """
62      t = np.arange(0, Tf + h, h)
63      x_hist = np.zeros((len(x0), len(t)))
64      x_hist[:, 0] = x0
65
66      for k in range(len(t) - 1):
67          k1 = fun(x_hist[:, k])
68          k2 = fun(x_hist[:, k] + 0.5 * h * k1)
69          k3 = fun(x_hist[:, k] + 0.5 * h * k2)
70          k4 = fun(x_hist[:, k] + h * k3)
71
72          x_hist[:, k + 1] = x_hist[:, k] + (h / 6.0) * (k1 + 2*k2 +
               2*k3 + k4)
73
74      return x_hist, t
75
76  def analytical_solution(t, x0, x0_dot):
77      """
78      Analytical solution for a*x_ddot + b*x_dot + c*x = d
79      """
80      a = 9.36
81      b = 4.97
82      c = 6.91
```

```
    d = -9.9

    # 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 = 9.36
b = 4.97
c = 6.91
d = -9.9

```

```
# 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 = 20.0
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}")
```

```python

# 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()

# Plotting and error calculation code continues...
```

Листинг 1: Код программы на Python