МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»
(Университет ИТМО)
Международный научно-образовательный центр
Физики наноструктур

Отчет лаб 1

по дисциплине:

Simulation of Robotic Systems

Студент:

Группа № R4134c *Буй Динь Кхай Нгуен*

Преподаватель: *Ракшин Егор Александрович*

Санкт-Петербург 2025

TABLE OF CONTENTS

1. ANALYTICAL SOLUTION CALCULATION

The given ODE is a **second-order linear ODE with constant coefficients**:

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

With the provided coefficients:

$$-4.42\,\ddot{x} + 9.25\,\dot{x} - 2.03\,x = 9.65$$

and the chosen initial conditions:

$$x(0) = 1.0,\ \ \dot{x}(0) = 0.0$$

The total analytical solution $x(t)$is the sum of the **homogeneous solution** $x_h(t)$and the **particular solution** x_p:

$$x(t) = x_h(t) + x_p$$

1.1 Particular Solution (x_p)

Since the forcing term d is a constant ($d = 9.65$) and $c \neq 0$, we assume the particular solution is also constant, $x_p = A$.

Substituting into the ODE ($\ddot{x}_p = 0$, $\dot{x}_p = 0$) gives:

$$c\,A = d \Rightarrow A = \frac{d}{c}$$

$$x_p = \frac{9.65}{-2.03} \approx -4.7537$$

Thus, the particular solution is:

$$x_p = -4.7537$$

1.2 Homogeneous Solution ($x_h(t)$)

The homogeneous equation is:

$$-4.42\,\ddot{x} + 9.25\,\dot{x} - 2.03\,x = 0$$

The corresponding characteristic equation is:

$$ar^2 + br + c = 0 \Rightarrow -4.42r^2 + 9.25r - 2.03 = 0$$

Solving for the roots r using the quadratic formula:

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Compute the discriminant:

$$\Delta = 9.25^2 - 4(-4.42)(-2.03) = 85.5625 - 35.8904 = 49.6721$$

The roots are real and distinct:

$$r_1 = \frac{-9.25 + \sqrt{49.6721}}{2(-4.42)} = 0.2491, \qquad r_2 = \frac{-9.25 - \sqrt{49.6721}}{2(-4.42)} = 1.8448$$

Hence, the homogeneous solution is:

$$x_h(t) = C_1 e^{0.2491t} + C_2 e^{1.8448t}$$

1.3 General Solution

$$x(t) = C_1 e^{0.2491t} + C_2 e^{1.8448t} - 4.7537$$

1.4. Applying Initial Conditions

Given the initial conditions $x(0) = 1.0$ and $\dot{x}(0) = 0.0$:

Condition 1:

$$x(0) = C_1 + C_2 - 4.7537 = 1.0$$

$$C_1 + C_2 = 5.7537 \text{ (Equation A)}$$

Condition 2:

Differentiate $x(t)$to find $\dot{x}(t)$:

$$\dot{x}(t) = 0.2491\, C_1 e^{0.2491t} + 1.8448\, C_2 e^{1.8448t}$$

At $t = 0$:

$$0.2491\, C_1 + 1.8448\, C_2 = 0.0 \text{ (Equation B)}$$

1.5. Solving for C_1 and C_2

From (A): $C_1 = 5.7537 - C_2$.

Substitute into (B):

$$0.2491(5.7537 - C_2) + 1.8448C_2 = 0.0$$

$$1.4332 - 0.2491C_2 + 1.8448C_2 = 0$$

$$1.5957C_2 = -1.4332 \Rightarrow C_2 = -0.8981$$

$$C_1 = 5.7537 - (-0.8981) = 6.6518$$

1.6. Final Analytical Solution

$$x(t) = 6.6518\, e^{0.2491t} - 0.8981\, e^{1.8448t} - 4.7537$$

2. NUMERICAL INTEGRATION METHODS

The given second-order linear ordinary differential equation (ODE) is:

$$-4.42\,\ddot{x} + 9.25\,\dot{x} - 2.03\,x = 9.65$$

To solve this ODE using numerical integration methods (Euler and Runge–Kutta), it must first be converted into an equivalent system of **first-order** ODEs — the standard **state-space form**.

2.1 Definition of State Variables

We define the state vector **Y(t)** using the primary variable $x(t)$and its first derivative $\dot{x}(t)$:

$$y_1 = x$$

$$y_2 = \dot{x}$$

This immediately gives the first equation of the system:

$$\dot{y_1} = y_2$$

2.2 Expressing the Second Derivative ($\ddot{x}$)

From the original ODE, solve for the second derivative $\ddot{x}$:

$$-4.42\,\ddot{x} = 9.65 - 9.25\,\dot{x} + 2.03\,x$$

$$\ddot{x} = \frac{2.03}{4.42}x - \frac{9.25}{4.42}\dot{x} - \frac{9.65}{4.42}$$

2.3. Final First-Order System

By substituting the state variables $y_1 = x$and $y_2 = \dot{x}$, we obtain:

$$\dot{y_1} = y_2$$

$$\dot{y_2} = \frac{2.03\,y_1 - 9.25\,y_2 - 9.65}{4.42}$$

or, in vector form:

$$\dot{\mathbf{Y}} = \begin{pmatrix} \dot{y_1} \\ \dot{y_2} \end{pmatrix} = \begin{pmatrix} y_2 \\ \frac{2.03\,y_1 - 9.25\,y_2 - 9.65}{4.42} \end{pmatrix}$$

This represents the **state-space system**:

$$\dot{\mathbf{Y}} = f(t, \mathbf{Y})$$

which can be directly implemented in **Python** and passed to numerical integration functions (Forward Euler, Backward Euler, and RK4) in the **integrators.ipynb** file.

3. APPENDIX

Below is the code implemented in python to calculate $\boldsymbol{x}(\boldsymbol{t})$ in both analytical and numerical integration methods for comparison:

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

def my_ode_dynamics(x):
    """
    Dynamics for a*x'' + b*x' + c*x = d
    -4.42*x'' + 9.25*x' - 2.03*x = 9.65
    State vector x = [x, x_dot]
    """
    a = -4.42
    b = 9.25
    c = -2.03
    d = 9.65

    y1 = x[0]  # x
    y2 = x[1]  # x_dot

    y1_dot = y2  # x_dot
    y2_dot = (d - b * y2 - c * y1) / a  # x''

    return np.array([y1_dot, y2_dot])

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):
```

```python
        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

# ANALYTICAL SOLUTION
def analytical_solution(t):
    """
    x(t) = C1*e^(r1*t) + C2*e^(r2*t) + xp
    C1=6.6518, C2=-0.8981 (for x(0)=1, x'(0)=0)
    """
    r1 = 0.2491
    r2 = 1.8448
    C1 = 6.6518
    C2 = -0.8981
    xp = -4.7537
    return C1 * np.exp(r1 * t) + C2 * np.exp(r2 * t) + xp

# PRINT COMPARISON TABLE
def print_comparison_results(t_values, t_full, x_an, x_fe, x_be, x_rk4):
    # Setup tables
    data_values = []
    data_errors = []
```

```python
    for t in t_values:
        # find the closest index in t_full
        index = np.argmin(np.abs(t_full - t))

        val_an = x_an[index]
        val_fe = x_fe[0, index]
        val_be = x_be[0, index]
        val_rk4 = x_rk4[0, index]

        # Compute the absolute error
        error_fe = np.abs(val_fe - val_an)
        error_be = np.abs(val_be - val_an)
        error_rk4 = np.abs(val_rk4 - val_an)

        # Save data
        data_values.append((t, val_an, val_fe, val_be, val_rk4))
        data_errors.append((t, error_fe, error_be, error_rk4))

    # Values comparison
    print("\n" + "="*80)
    print("Values Comparison X(T)")
    print("="*80)
    print(f"{'Time (t)':<15}{'Analytical':<15}{'Forward Euler':<15}{'Backward Euler':<15}{'Runge-Kutta 4':<20}")
    print("-"*80)
    for t, an, fe, be, rk4 in data_values:
        print(f"{t:<15.2f}{an:<15.4f}{fe:<15.4f}{be:<15.4f}{rk4:<20.4f}")
    print("="*80)

    # Error comparison
    print("\n" + "="*80)
    print("Absolute Error Comparison |X_NUM - X_AN|")
    print("="*80)
    print(f"{'Time (t)':<15}{'Error FE':<15}{'Error BE':<15}{'Error RK4':<15}")
    print("-"*80)
    for t, err_fe, err_be, err_rk4 in data_errors:
        print(f"{t:<15.2f}{err_fe:<15.2e}{err_be:<15.2e}{err_rk4:<15.2e}")
    print("="*80)

# SET UP AND RUN SIMULATIONS
x0_ode = np.array([1.0, 0.0])  # [x(0), x'(0)]
Tf = 3.0                        # Simulation time
h = 0.01                        # Time step

# Run the integrators
x_fe, t_full = forward_euler(my_ode_dynamics, x0_ode, Tf, h)
```

```python
x_be, t_full = backward_euler(my_ode_dynamics, x0_ode, Tf, h)
x_rk4, t_full = runge_kutta4(my_ode_dynamics, x0_ode, Tf, h)

# Compute the analytical solution at the time points
x_an = analytical_solution(t_full)

# Time points for comparison
t_comparison = [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]

# PRINT COMPARISON TABLE
print_comparison_results(t_comparison, t_full, x_an, x_fe, x_be, x_rk4)


# PLOT COMPARISON
plt.figure(figsize=(10, 6))

plt.plot(t_full, x_an, 'k-', label='Analytical Solution', linewidth=3,
alpha=0.7)
plt.plot(t_full, x_fe[0, :], 'r--', label='Forward Euler')
plt.plot(t_full, x_be[0, :], 'b-.', label='Backward Euler')
plt.plot(t_full, x_rk4[0, :], 'g:', label='Runge-Kutta 4')

plt.xlabel('Time (t)')
plt.ylabel('Position x(t)')
plt.title(f'Comparison of ODE Integrators (h={h})')
plt.legend()
plt.grid(True)
plt.show()
```

4. RESULTS

4.1 Comparison table

Values Comparison X(T)

Time (t)	Analytical	Forward Euler	Backward Euler	Runge-Kutta 4
0.00	1.0000	1.0000	1.0000	1.0000
0.50	0.5214	0.5394	0.5033	0.5216
1.00	-1.9024	-1.8078	-1.9957	-1.8998
1.50	-9.3808	-9.0168	-9.7363	-9.3674
2.00	-29.7564	-28.5251	-30.9538	-29.7037
2.50	-82.7799	-78.8978	-86.5548	-82.5976
3.00	-218.1588	-206.4415	-229.5775	-217.5721

Table 1: Comparison of $x(t)$ Values (step size h=0.01)

Absolute Error Comparison |X_NUM - X_AN|

Time (t)	Error FE	Error BE	Error RK4
0.00	8.88e-16	8.88e-16	8.88e-16
0.50	1.80e-02	1.81e-02	2.51e-04
1.00	9.46e-02	9.33e-02	2.65e-03
1.50	3.64e-01	3.55e-01	1.34e-02
2.00	1.23e+00	1.20e+00	5.26e-02
2.50	3.88e+00	3.77e+00	1.82e-01
3.00	1.17e+01	1.14e+01	5.87e-01

Table 2: Comparison of Absolute Error (step size h=0.01)

4.2 Graphic comparison

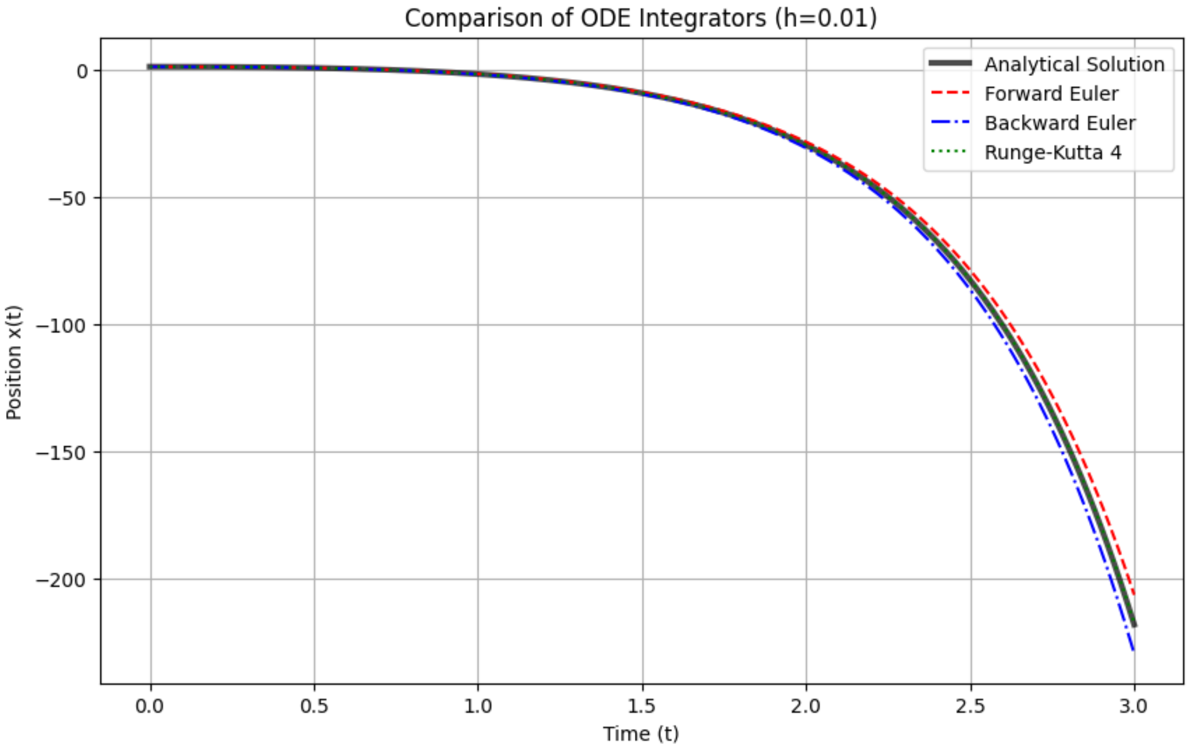

Figure 1: Comparison of ODE Integrators (step size h=0.01)

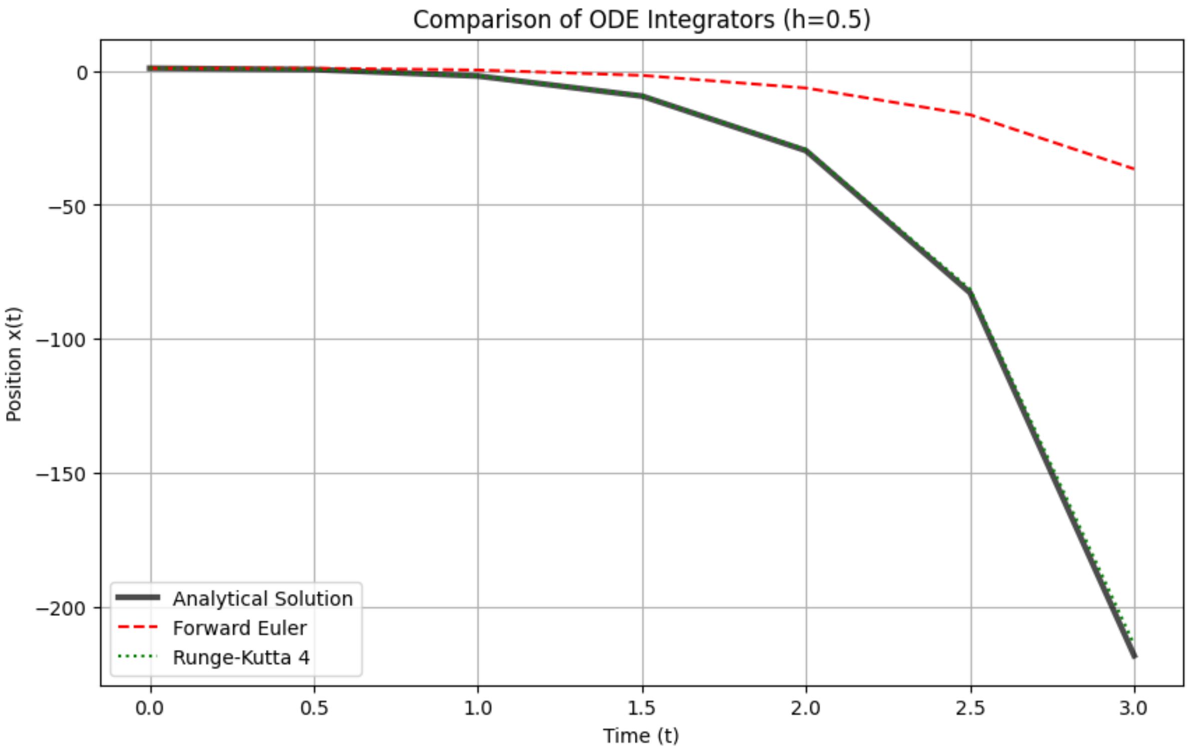

Figure 2: Comparison of ODE Integrators (step size h=0.5)

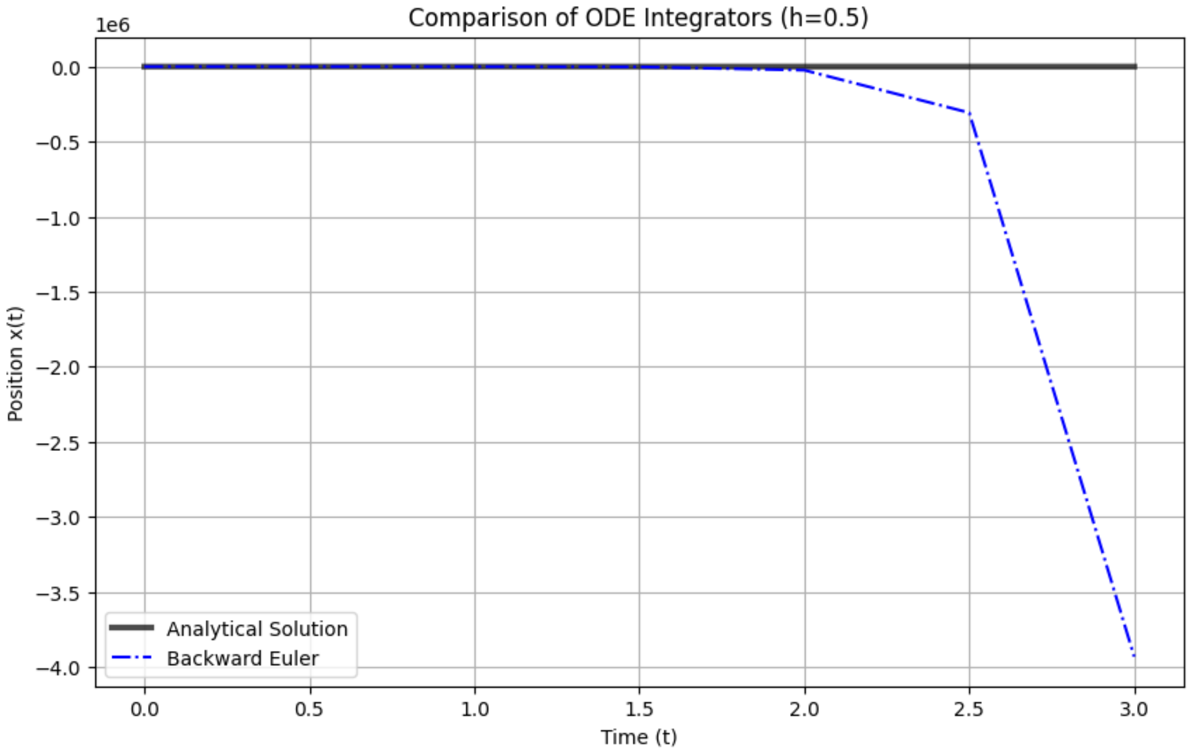

Figure 3: Comparison of ODE Integrators (step size h=0.5)

5. DISCUSSION AND CONCLUSION

The purpose of this exercise was to compare the accuracy of three numerical integrators—Explicit Euler (FE), Implicit Euler (BE), and Runge-Kutta — against the analytical solution for the unstable second-order ODE

5.1 Accuracy Comparison

The computational results clearly demonstrate that the **Runge–Kutta 4 (RK4)** method delivers superior accuracy compared to the tested first-order methods.

- **Runge–Kutta 4:**
  The RK4 curve aligns most closely with the analytical solution over the entire simulation interval ($T_f = 3.0$).
  At $t = 3.00$, the RK4 result (≈ -217.57) exhibited the smallest deviation from the analytical value (-218.16).

  For large step size (h =0.5), RK4 still outperformed all other numerical methods, maintaining high accuracy ($O(h^4)$).

- **Euler Methods**:
  Both the Forward Euler (**FE**) and Backward Euler (**BE**) methods accumulated noticeably larger errors as time progressed.
  In the comparison table, the **FE** method's absolute error at $t = 3.00$ reached approximately 11.71, illustrating the limitation of first-order schemes for rapidly growing solutions.

  - **Forward Euler (FE)**: Tended to **underestimate** the magnitude of the negative position; for example, $x_{FE} = -206.44$ at $t = 3.00$. The FE trajectory consistently appeared **above** the analytical curve.

  - **Backward Euler (BE)**: Tended to **overestimate** the magnitude of the negative position; for example, $x_{BE} = -229.58$ at $t = 3.00$. The BE trajectory consistently appeared **below** the analytical curve.

  - For large step size (h = 0.5), **FE** and **BE** both suffered catastrophic accuracy loss. Their low-order accuracy ($O(h)$) was insufficient to control the accumulation of error in the exponentially growing system.

5.2 Stability Analysis

The given ODE system is **inherently unstable** due to its positive characteristic roots ($r_1 \approx 0.2491$, $r_2 \approx 1.8448$), which cause the solution to grow exponentially over time.

During the test with a small step size ($h = 0.01$) and later a large step size ($h = 0.5$), all three numerical integrators remained **numerically stable** for the simulated duration ($T_f = 3.0$), exhibiting no signs of divergence or numerical "blow-up."

However, the stability demonstrated by **FE** and **BE** at step size ($h = 0.5$) was misleading, as it was decoupled from practical accuracy. The RK4 method alone proved its **stability and high accuracy** simultaneously.

5.3. Conclusion

Based on quantitative error analysis and graphical comparisons, it show that in comparison between these numerical methods, the 4th-order Runge–Kutta method is clearly the most efficient, offering the highest accuracy and reliable performance across both small and large step sizes in this unstable ODE system.