NATIONAL RESEARCH UNIVERSITY OF INFORMATION TECHNOLOGY, MECHANICS AND OPTICS

FACULTY OF CONTROL SYSTEMS AND ROBOTICS

LABORATORY TASK N°1

COMPLETED BY:	Valverde Ramírez, Edgar David
ISU:	520193
DISCIPLINE:	Simulation of Robotic Systems
INSTRUCTOR:	PhD. Ivan Borisov

1. **Data:**
   The objective of this task is to analyze the behavior of a dynamic system governed by a linear second-order ordinary differential equation (ODE) with constant coefficients. The specific system parameters provided for the simulation are listed below:

a	b	c	d
-4.36	0.08	-6.39	1.25

The governing differential equation is defined as:

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

2. **Analytical Solution:**
   a. Homogeneous Solution
   To determine the natural response of the system, we first solve the associated homogeneous equation. Starting from the characteristic equation:

$$ar^2 + br + c = 0$$

Using the quadratic formula, the roots of the polynomial were calculated. The discriminant is negative, resulting in a pair of complex conjugate roots:

$$r_1 = 9.174 \times 10^{-3} + 1.2106i, \qquad r_2 = 9.174 \times 10^{-3} - 1.2106i$$

Because the roots are complex, the homogeneous solution describes an oscillatory motion with an exponential envelope. The positive real part (9.174×10^{-3}) indicates that the system is unstable and the amplitude will grow over time. The general form is:

$$x_h = e^{9.174\times10^{-3}t}(C_1\cos(1.2106t) + C_2\sin(1.2106t))$$

   b. Particular Solution
   Since the non-homogeneous term \$d\$ is a constant, we use the method of undetermined coefficients to propose a constant particular solution, $x_p = A$. Substituting this into the original ODE (where derivatives of a constant are zero) yields:

$$cA = d$$

$$A = \frac{d}{c} = -0.1956 = x_p$$

c. General Solution

The complete general solution is the superposition of the homogeneous and particular solutions:

$$x = x_h + x_p$$

$$x = e^{9.174\times10^{-3}t}(C_1\cos(1.2106t) + C_2\sin(1.2106t)) - 0.1956$$

d. Solution with Initial Conditions

Using the initial conditions $x(0) = 0$ and $\dot{x}(0) = 0$.

$$x(0) = 0 = C_1 - 0.1956$$

$$C_1 = 0.1956$$

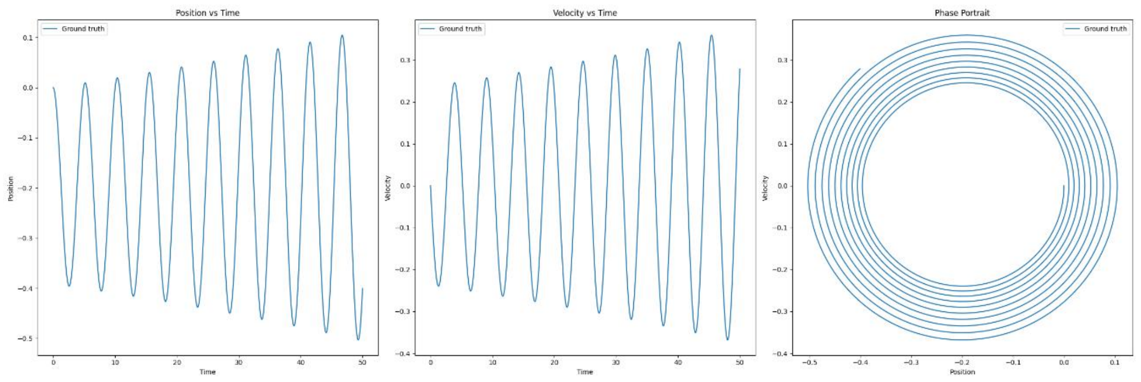
$$\dot{x}(0) = 0 = 9.174\times10^{-3}C_1 + 1.2106\,C_2$$

$$C_2 = -\frac{9.174\times10^{-3}C_1}{1.2106} = -1.4823\times10^{-3}$$

Substituting these constants back into the general equation yields the final specific analytical solution.

$$x = e^{9.174\times10^{-3}t}(0.1956\cos(1.2106t) - 1.4823\times10^{-3}\sin(1.2106t)) - 0.1956$$

The corresponding graph of the solution is shown below:



3. **Numerical Solution:**

   To enable numerical simulation, the second-order ODE is transformed into a state-space representation consisting of a system of two first-order ODEs.
   Let the state variables be defined as position $x_1 = x$ and velocity $x_2 = \dot{x}$.
   The system can then be expressed in matrix form $\dot{X} = AX + B$:

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -c/a & -b/a \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ d/a \end{bmatrix}$$

$$F(X) = \dot{X} = AX + B$$

   a. Explicit Euler

   This first-order method estimates the next state based on the gradient at the current time step. It is computationally simple but conditionally stable. The iteration is given by:

$$X_{i+1} = X_i + hF(X_i)$$
$$X_{i+1} = X_i + h(AX_i + B)$$

   b. Implicit Euler

   To improve stability, the Implicit Euler method evaluates the gradient at the future (unknown) time step. This requires solving a linear system at each iteration involving the matrix inversion of $(I - hA)$:

$$X_{i+1} = X_i + hF(X_{i+1})$$
$$X_{i+1} = X_i + h(AX_{i+1} + B)$$
$$(I - hA)X_{i+1} = X_i + hB$$
$$X_{i+1} = (I - hA)^{-1}X_i + h(I - hA)^{-1}B$$

   This method is generally more robust for stiff equations but computationally more intensive per step.

   c. Runge-Kutta 4 (RK4)

   The fourth-order Runge-Kutta method provides a high-accuracy approximation by taking a weighted average of four slopes calculated within the time step. This method achieves the accuracy of a Taylor series expansion up to the fourth term without calculating higher derivatives:

$$X_{i+1} = X_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h$$

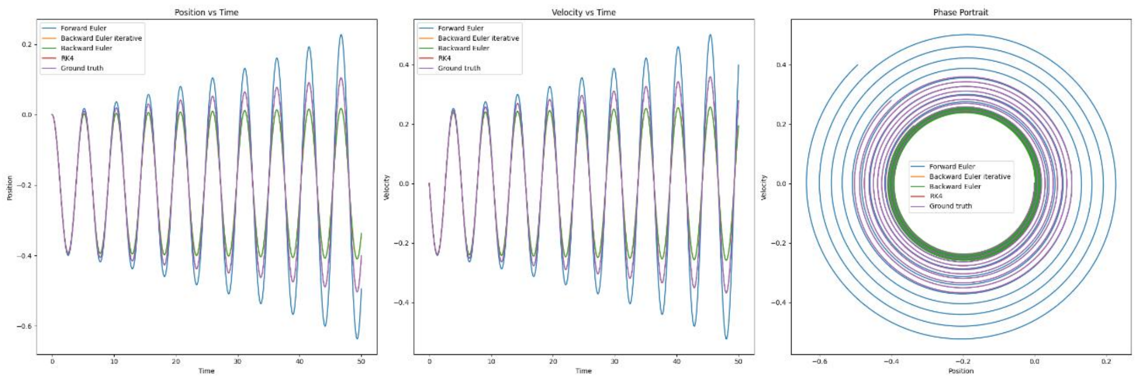
   Where

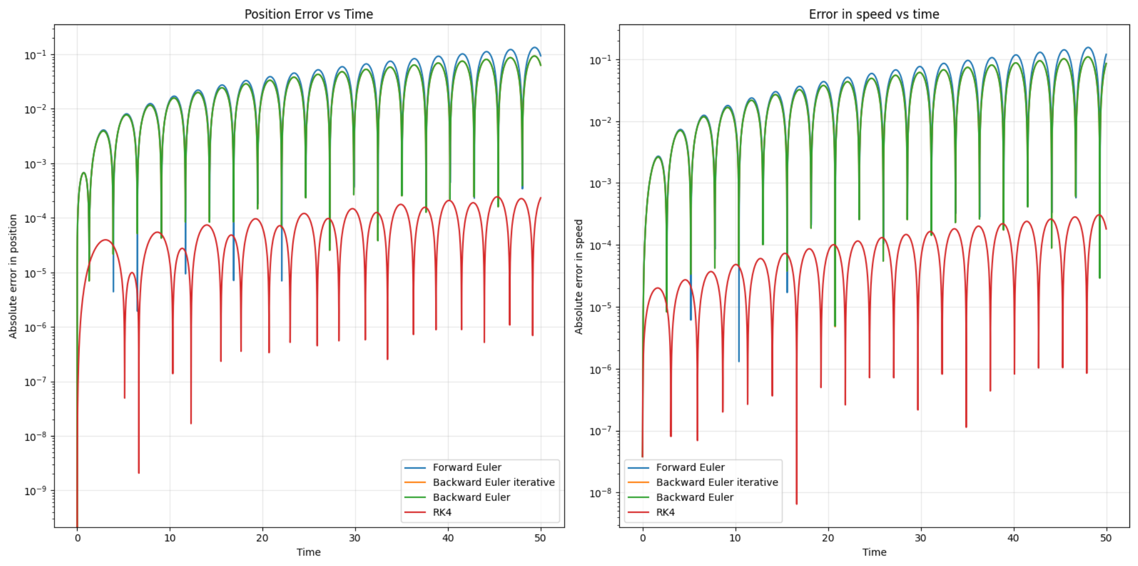
$$k_1 = F(X_i)$$
$$k_2 = F\left(X_i + \frac{1}{2}k_1 h\right)$$
$$k_3 = F\left(X_i + \frac{1}{2}k_2 h\right)$$
$$k_4 = F(X_i + k_3 h)$$

   and $F(\mathbf{X}) = A\mathbf{X} + \mathbf{B}$.

4. Results:

The simulation was conducted over a time interval from t=0 to t=50 seconds, divided into 5000 discrete time steps. The numerical results were validated by computing the absolute error relative to the analytical solution derived in Section 2.





5. Conclusions

- **Accuracy Comparison:** All implemented numerical methods successfully approximated the general behavior of the analytical solution; however, the precision varied significantly. The Runge-Kutta 4 (RK4) method proved to be the most accurate, maintaining errors on the order of 10^{-4}. In contrast, both Euler methods exhibited errors on the order of 10^{-1}, is three orders of magnitude higher.
- The **Explicit Euler** method showed a tendency to accumulate error by overestimating the solution energy, leading to a larger spiral than the true trajectory.
- The **Implicit Euler** method demonstrated better numerical stability properties but tended to dampen the system, underestimating the true solution amplitude.
- The results highlight the trade-off between computational simplicity and numerical accuracy, with RK4 being the most reliable on the group.

6. Code

```
def analytical_solution(t):
    A = 0.1956
    B = -1.4823e-3
    omega = 1.2106
    k = 9.174e-3

    exp_kt = np.exp(k * t)
    cos_omega_t = np.cos(omega * t)
    sin_omega_t = np.sin(omega * t)

    x = exp_kt * (A * cos_omega_t + B * sin_omega_t) - A
    dx = (k * exp_kt * (A * cos_omega_t + B * sin_omega_t) +
          exp_kt * (-A * omega * sin_omega_t + B * omega *
cos_omega_t))

    return np.array([x, dx])
```

```
def backward_euler_2(fun, x0, Tf, h):
    x_hist = np.zeros((len(x0), len(t)))
    x_hist[:, 0] = x0
    a = -4.36
    b = 0.08
    c = -6.39
    d = 1.25
    ca = c/a
    ba = b/a
    da = d/a
    A = np.array([[0, 1],
                  [-ca, -ba]])
    B = np.array([0, da])
    K = np.eye(2)-A*h
    Kinv = np.linalg.inv(K)

    for k in range(len(t) - 1):
        x_hist[:, k + 1] = Kinv@(x_hist[:, k]+h*B)

    return x_hist
```