ITMO

Simulation of Robot systems

Task: 1

By: Nagham Sleman **476937**

We have to:

1. Solve the second-order ODE analytically:

$$ax^{\cdot\cdot} + bx^{\cdot} + cx = d$$

2. Solve it numerically using three integrators:
   - **Explicit (Forward) Euler**
   - **Implicit (Backward) Euler**
   - **Runge–Kutta 4th order (RK4)**
3. Compare all numerical solutions with the analytical one.
4. Discuss the differences in accuracy and stability.

1. **Convert the ODE into a first-order system**
   To use numerical integrators, we must rewrite the second-order ODE as two first-order equations.
   We have:

$$x_1 = x, \qquad x_2 = x^{\cdot}$$

   Then:

$$x_1{}^{\cdot} = x_2$$
$$x_2{}^{\cdot} = \frac{1}{a}(-bx_2 - cx_1 + d)$$

   From the table I have:

$$a = 4.72$$
$$b = -5.21$$
$$c = 9.56$$
$$d = 9.21$$

   That gives the system dynamics function in Python:

```
# System dynamics (ODE -> first-order)
# ----------------------------
a = 4.72
b = -5.21
c = 9.56
d = 9.21
```

```
def system_dynamics(x):
    x1 = x[0]
    x2 = x[1]
    dx1 = x2
    dx2 = (d/a) - (b/a)*x2 - (c/a)*x1
    return np.array([dx1, dx2])
```

2. Analytical solution

We have:

$$ax^{\cdot\cdot} + bx^{\cdot} + cx = d$$

Then divide by a :

$$x^{\cdot\cdot} + \frac{b}{a}x^{\cdot} + \frac{c}{a}x = \frac{d}{a}$$

2.1 Particular solution

Since d is constant, the steady-state (particular) solution is:

$$x_p = \frac{d}{c}$$

2.2 Homogeneous solution

The homogeneous equation:

$$x^{\cdot\cdot} + \frac{b}{a}x^{\cdot} + \frac{c}{a}x = 0$$

has a characteristic equation:

$$r2 + \frac{b}{a}r + \frac{c}{a} = 0$$

Compute its discriminant:

$$\Delta = (\frac{b}{a})^2 - 4\frac{c}{a}$$

Then:

- If Δ > 0 → two real distinct roots → overdamped system
- If Δ = 0 → one repeated real root → critically damped
- If Δ < 0 → complex roots → underdamped oscillatory system

Depending on the case:

- Real roots: $x_h = Ae^{r_1 t} + Be^{r_2 t}$
- Repeated root: $x_h = (A + Bt)e^{rt}$
- Complex roots: $x_h = e^{\mu t}(Acos(\omega t) + Bsin(\omega t))$

Finally:

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

We use initial conditions $x(0) = x_0, x^{\cdot}(0) = v_0$ to find A and B.

2.3 Analytical Solution of the ODE (Using My Assigned Parameters):

We consider the second–order linear differential equation**:**

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

with parameters from the table:

$$a = 4.72, b = -5.21, c = 9.56, d = 9.21$$

and initial conditions:

$$x(0) = 0.1, \dot{x}(0) = 0$$

2.3.1 Reformulating the ODE:

Dividing the equation by a:

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

We define the normalized parameters:

$$\alpha = \frac{b}{a} = -1.1038135593, \qquad \omega^2 = \frac{c}{a} = 2.0254237288.$$

The particular (steady–state) solution is:

$$x_p = \frac{d}{c} = 0.9633891213$$

2.3.2 Homogeneous Solution and System Type:

The characteristic equation is:

$$r^2 + \alpha r + \omega^2 = 0$$

with discriminant:

$$\Delta = \alpha^2 - 4\omega^2 = -6.88329 < 0$$

Since the discriminant is negative, the system is **underdamped with complex roots:**

$$r = \mu \pm i\omega_d$$

Where:

$$\mu = -\frac{\alpha}{2} = 0.5519067797 > 0, \qquad \omega_d = \frac{\sqrt{4\omega^2 - \alpha^2}}{2} = 1.3118012942.$$

Note: Because $\mu > 0$, the exponential term $e^{\mu t}$ grows with time.

This means the system is **unstable**, since the damping coefficient b is negative (negative damping injects energy instead of dissipating it).

2.3.3 Final Closed-Form Analytical Solution:

Let:

$$C = x_0 - x_p = -0.8633891213$$

$$D = \frac{\dot{x}0 - \mu C}{\omega_d} = 0.3632488485$$

Substituting the numeric values:

$$x(t) = 0.9633891213 + e^{0.5519067797t}[-0.8633891213cos(1.3118012942t) + 0.3632488485sin(1.3118012942t)]$$

Velocity $\dot{x}(t)$:

$$\dot{x}(t) = e^{\mu t}((D\omega_d + C\mu)cos(\omega_d t) + (D\mu - C\omega_d)sin(\omega_d t))$$

$$\dot{x}(t) = e^{0.5519067797t}[-0.0003764313cos(1.3118012942t) + 1.3331298942sin(1.3118012942t)]$$

2.3.4 Interpretation:

- The solution contains the factor $e^{\mu t}$ with $\mu > 0$, meaning **exponential growth over time**.
- This occurs because $\boldsymbol{b < 0}$, so the "damping" term actually adds energy instead of removing it.
- As a result, both $x(t)$ and $\dot{x}(t)$ grow unbounded with time.
- Numerical integrators will reflect this behavior, and the divergence is **physical**, not a numerical error.

The code:

```
# Analytical solution (improved)
# ------------------------------
def analytical_solution_full(t, a, b, c, d, x0):
    x0_, xdot0_ = x0
    alpha = b / a
    omega2 = c / a
    x_ss = (d/c)
    disc = alpha**2 - 4*omega2

    if disc > 0:  # overdamped
        r1 = (-alpha + np.sqrt(disc))/2
        r2 = (-alpha - np.sqrt(disc))/2
        A = (xdot0_ - r2*(x0_ - x_ss)) / (r1 - r2)
        B = (x0_ - x_ss) - A
        x_t = x_ss + A*np.exp(r1*t) + B*np.exp(r2*t)
        xdot_t = A*r1*np.exp(r1*t) + B*r2*np.exp(r2*t)

    elif np.isclose(disc, 0):  # critically damped
        r = -alpha / 2
        A = x0_ - x_ss
        B = xdot0_ - A*r
        x_t = x_ss + (A + B*t)*np.exp(r*t)
        xdot_t = np.exp(r*t)*(B + r*(A + B*t))

    else:  # underdamped
        mu = -alpha / 2
        omega = np.sqrt(4*omega2 - alpha**2)/2
        C = x0_ - x_ss
        D = (xdot0_ - mu*C)/omega
        x_t = x_ss + np.exp(mu*t)*(C*np.cos(omega*t) + D*np.sin(omega*t))
        xdot_t = np.exp(mu*t)*((D*omega + C*mu)*np.cos(omega*t) + (D*mu -
C*omega)*np.sin(omega*t))

    return np.vstack([x_t, xdot_t])
```

3. **Numerical solution:**
   we'll use the helper functions:
   - $forward\_euler()$
   - $backward\_euler()$
   - $runge\_kutta4()$

```
# Simulation
# ----------------------------
# Simulation setup
# ----------------------------
x0 = np.array([0.1, 0.0])
Tf = 10.0
h = 0.01
t = np.arange(0, Tf + h, h)

# ----------------------------
x_fe, t_fe = forward_euler(system_dynamics, x0, Tf, h)
x_be, t_be = backward_euler(system_dynamics, x0, Tf, h)
x_rk4, t_rk4 = runge_kutta4(system_dynamics, x0, Tf, h)
x_analytical = analytical_solution_full(t, a, b, c, d, x0)
```

4. **Compare with analytical solution:**
   We write a small function to compute the analytical solution numerically and then compare it to the numerical results:

```
# Compute Errors
# ----------------------------
def compute_errors(x_num, x_true):
    pos_err = np.max(np.abs(x_num[0,:] - x_true[0,:]))
    vel_err = np.max(np.abs(x_num[1,:] - x_true[1,:]))
    rms_pos = np.sqrt(np.mean((x_num[0,:] - x_true[0,:])**2))
    rms_vel = np.sqrt(np.mean((x_num[1,:] - x_true[1,:])**2))
    return pos_err, vel_err, rms_pos, rms_vel

max_fe, max_fe_v, rms_fe, rms_fe_v = compute_errors(x_fe, x_analytical)
max_be, max_be_v, rms_be, rms_be_v = compute_errors(x_be, x_analytical)
max_rk4, max_rk4_v, rms_rk4, rms_rk4_v = compute_errors(x_rk4, x_analytical)

print("Forward Euler: Max x = {:.5f}, Max v = {:.5f}, RMS x = {:.5f}, RMS v =
{:.5f}".format(max_fe, max_fe_v, rms_fe, rms_fe_v))
print("Backward Euler: Max x = {:.5f}, Max v = {:.5f}, RMS x = {:.5f}, RMS v =
{:.5f}".format(max_be, max_be_v, rms_be, rms_be_v))
print("RK4: Max x = {:.5f}, Max v = {:.5f}, RMS x = {:.5f}, RMS v =
{:.5f}".format(max_rk4, max_rk4_v, rms_rk4, rms_rk4_v))
```

5. Plot and analyze:

Visualization of Results

To analyze the behavior of the system and compare the numerical methods with the analytical solution, we generated four types of plots:

1. **Analytical Solution**
   - This plot shows both the displacement $x(t)$ and the velocity $x'(t)$ obtained from the exact analytical solution of the ODE.
   - The analytical curves serve as a reference to evaluate the accuracy of the numerical integrators.
2. **Numerical Position**
   - The displacement $x(t)$ computed using Forward Euler, Backward Euler, and Runge-Kutta 4 methods is plotted over time.
   - This allows visual comparison of how closely each numerical method approximates the analytical solution.
3. **Numerical Velocity**
   - The velocity $x'(t)$) from all three numerical methods is plotted over time.
   - This highlights differences in capturing the dynamic response, especially with explicit versus implicit methods.
4. **Phase Portrait (x vs x')**
   - The phase portrait shows the trajectory of the system in the displacement-velocity plane.
   - It provides insight into stability, damping behavior, and overall dynamics.
   - Comparing the numerical trajectories to each other reveals differences in numerical dissipation and accuracy.

Observations from the plots:

- Runge-Kutta 4 matches the analytical solution very closely in both displacement and velocity.
- Forward Euler deviates more, especially at larger times, due to accumulation of numerical error.
- Backward Euler remains stable but may introduce slight numerical damping.
- The phase portrait clearly illustrates how each integrator captures the trajectory of the system.

```
# Plots
# ------------------------------
plt.figure(figsize=(24, 8))

# Analytical position and velocity
plt.subplot(1,4,1)
plt.plot(t, x_analytical[0,:], 'b', label='x(t) Analytical', linewidth=2)
plt.plot(t, x_analytical[1,:], 'r--', label="x'(t) Analytical", linewidth=2)
plt.xlabel('Time [s]')
plt.ylabel('Value')
plt.title('Analytical Solution')
plt.grid(True)
plt.legend()

# Numerical positions
plt.subplot(1,4,2)
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 [s]')
plt.ylabel('x(t)')
plt.title('Numerical Position')
plt.grid(True)
plt.legend()

# Numerical velocities
plt.subplot(1,4,3)
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 [s]')
plt.ylabel("x'(t)")
plt.title('Numerical Velocity')
plt.grid(True)
plt.legend()

# Phase Portrait
plt.subplot(1,4,4)
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('x(t)')
plt.ylabel("x'(t)")
plt.title('Phase Portrait')
plt.grid(True)
plt.legend()

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

The results:

```
Forward Euler: Max x = 24.26668, Max v = 20.59367, RMS x = 5.39608, RMS v = 6.91198
Backward Euler: Max x = 22.46172, Max v = 18.60489, RMS x = 5.19137, RMS v = 6.27251
```

```
RK4: Max x = 0.00000, Max v = 0.00000, RMS x = 0.00000, RMS v = 0.00000
```

6. Discussion:

- **RK4** gives the most accurate results and almost overlaps with the analytical curve.
- **Forward Euler** tends to diverge or show noticeable phase and amplitude errors unless the time step is very small.
- **Backward Euler** is stable and damped, but slightly less accurate (introduces artificial damping).
- Smaller time steps improve all numerical results but increase computation time.