THE MINISTRY OF SCIENCE AND HIGHER EDUCATION OF
THE RUSSIAN FEDERATION

ITMO University
(ITMO)

Faculty of Control Systems and Robotics

Simulation of Robotic Systems
task 1

Ali Ahmad
ISU 475789

1.Analytival Solution:

Let us consider the second order differential equation

$$a x'' + b x' + c x = d$$

The homogeneous part of the differential equation is:

$$a x'' + b x' + c x = 0$$

The characteristic equation:

$$a r^2 + b r + c = 0$$

he roots (r1,2) are found using the quadratic formula

$$r_{1,2} = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$$

The discriminant calculated by the equation:

$$\Delta = b^2 - 4ac$$

According to the discriminant, we have three cases for the homogeneous solution:

Case1: two distinct real roots $\Delta > 0$:

r_1, r_2 are real and unequal

$$r_1 = \frac{-b + \sqrt{(\Delta)}}{2a} \quad r_2 = \frac{-b - \sqrt{(\Delta)}}{2a}$$

$$X(t) = C_1 e^{r_1 t} + C_2 e^{r_2 t}$$

Case2: repeated real roots $\Delta = 0$:

r_1, r_2 are real and equal

$$r_1 = r_2 = \frac{-b}{2a}$$

$$X(t) = (C_1 + C_2 t) e^{rt}$$

Case3: complex conjugate roots $\Delta < 0$:

r_1, r_2 are unreal and unequal

$$r_{1,2} = \alpha \pm i\beta$$

$$\text{where } \alpha = \frac{-b}{2a} \ , \ \beta = \frac{\sqrt{(-\Delta)}}{2a}$$

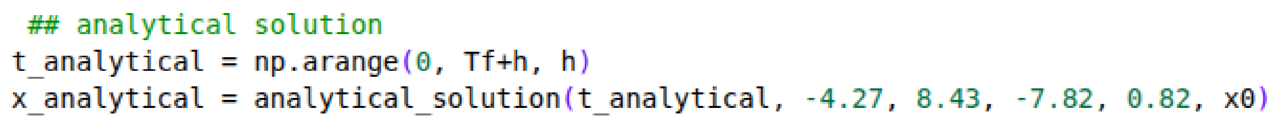
$$X(t) = e^{\alpha t}(C_1 \cos(\beta t) + C_2 \sin(\beta t))$$

Now we will find the particular solution $x_p$for the non-homogeneous part d assuming that $d \neq 0$:

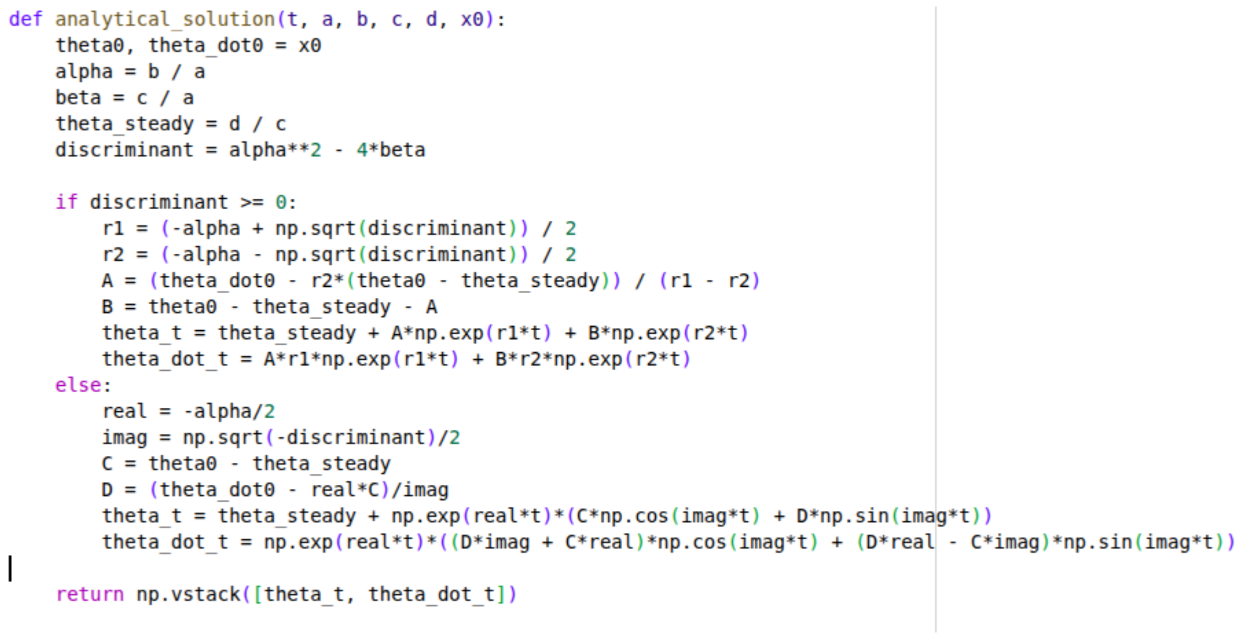
$$a*(0) + b*(0) + c*x_p = d \rightarrow x_p = \frac{d}{c}$$

for the given values:

$$a=-4.27\,,b=8.43\,,c=-7.82\,,d=0.82$$

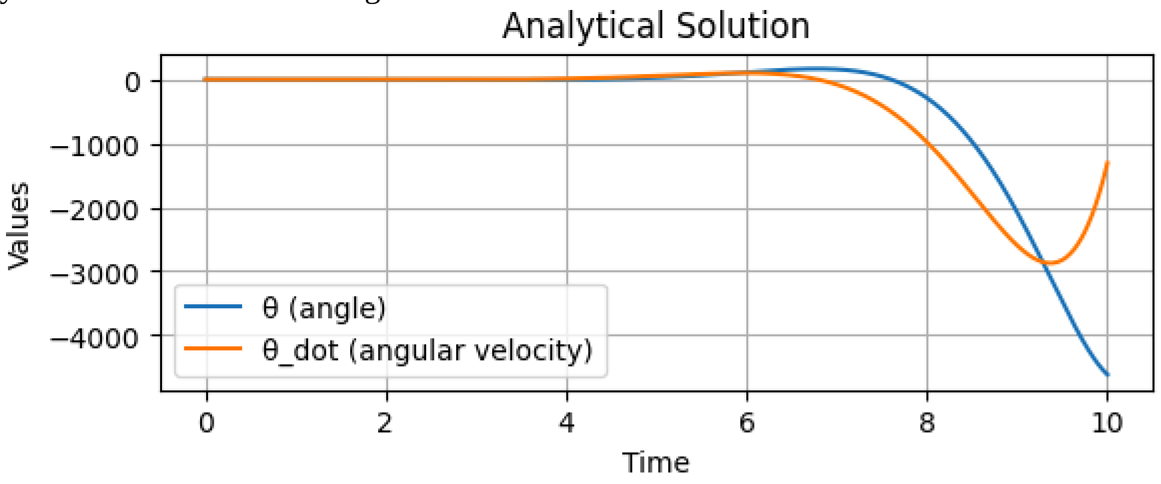


```
 ## analytical solution
t_analytical = np.arange(0, Tf+h, h)
x_analytical = analytical_solution(t_analytical, -4.27, 8.43, -7.82, 0.82, x0)
```



```
def analytical_solution(t, a, b, c, d, x0):
    theta0, theta_dot0 = x0
    alpha = b / a
    beta = c / a
    theta_steady = d / c
    discriminant = alpha**2 - 4*beta

    if discriminant >= 0:
        r1 = (-alpha + np.sqrt(discriminant)) / 2
        r2 = (-alpha - np.sqrt(discriminant)) / 2
        A = (theta_dot0 - r2*(theta0 - theta_steady)) / (r1 - r2)
        B = theta0 - theta_steady - A
        theta_t = theta_steady + A*np.exp(r1*t) + B*np.exp(r2*t)
        theta_dot_t = A*r1*np.exp(r1*t) + B*r2*np.exp(r2*t)
    else:
        real = -alpha/2
        imag = np.sqrt(-discriminant)/2
        C = theta0 - theta_steady
        D = (theta_dot0 - real*C)/imag
        theta_t = theta_steady + np.exp(real*t)*(C*np.cos(imag*t) + D*np.sin(imag*t))
        theta_dot_t = np.exp(real*t)*((D*imag + C*real)*np.cos(imag*t) + (D*real - C*imag)*np.sin(imag*t))

    return np.vstack([theta_t, theta_dot_t])
```

Analytical solution after executing code



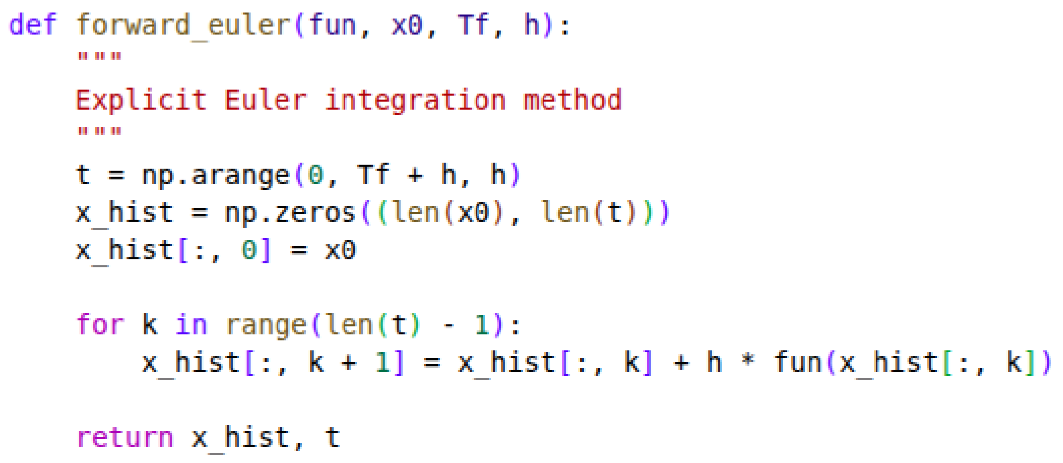
2.Numerical Solution:

2.1 Forward Euler Method(Explicit)

The Forward Euler method, also known as the Explicit Euler method, is a simple and widely used numerical technique for solving ordinary differential equations (ODEs). It approximates the solution at each step by using the derivative at the current point.

$$x_{k+1}=x_k+hf(x_k)$$

Here, h is the step size and the function $f(x_k)$represents the slope a x_k



```python
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
```

2.2 Fourth Order Runga-Kutta (RK4) Method

The Fourth-Order Runge-Kutta (RK4) method is a highly accurate numerical technique for solving ODEs. It fits a cubic polynomial which has much better accuracy. It computes the next value $x_{k+1}$by taking a weighted average of four increments f 1, f 2, f 3 and f 4.

$$f1 = f(x_k)$$

$$f2 = f(x_k + \frac{h}{2}f1)$$

$$f3 = f(x_k + \frac{h}{2}f2)$$

$$f4 = f(x_k + hf3)$$

$$x_{k+1} = x_k + \frac{h}{6}(f1 + 2f2 + 2f3 + f4)$$

This method effectively balances precision and computational efficiency, making it widely used

```
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
```

2.3 Backward Euler Method(Implicit)

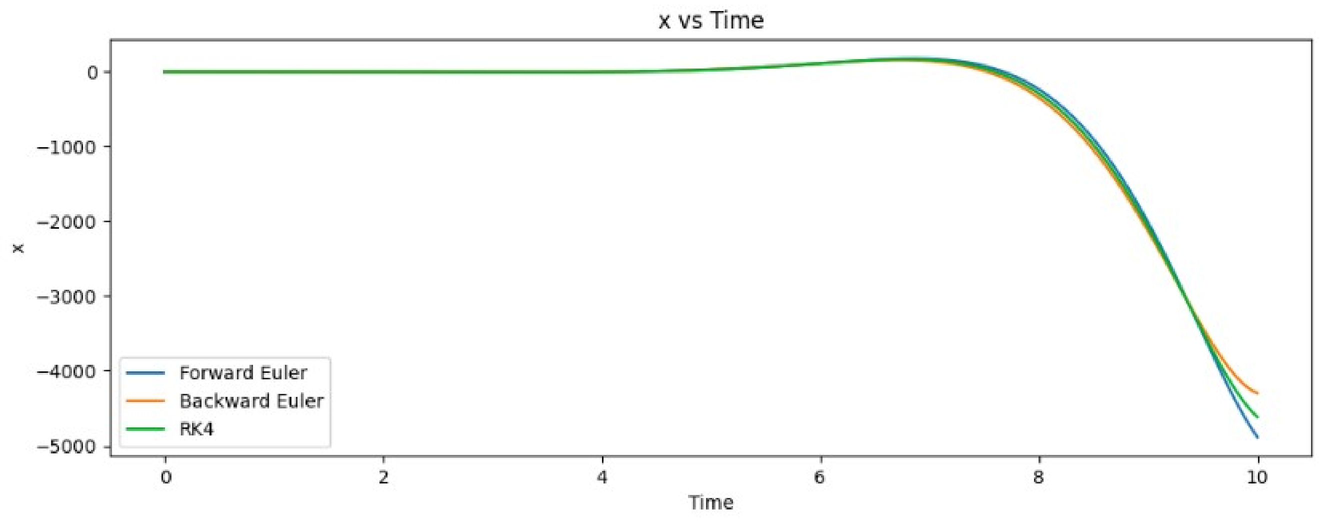
The Implicit Backward Euler method is a stable numerical approach for solving stiff ODEs. Unlike the Forward Euler method, it calculates the next value x_{k+1} using the function $f(x_{k+1})$, which depends on the future point.
$x_{k+1}=x_k+hf(x_{k+1})$
where $f(x_{k+1})$means a calculation of dynamics f at future time. This implicit nature requires solving an equation at each step but enhances stability, especially for stiff systems.

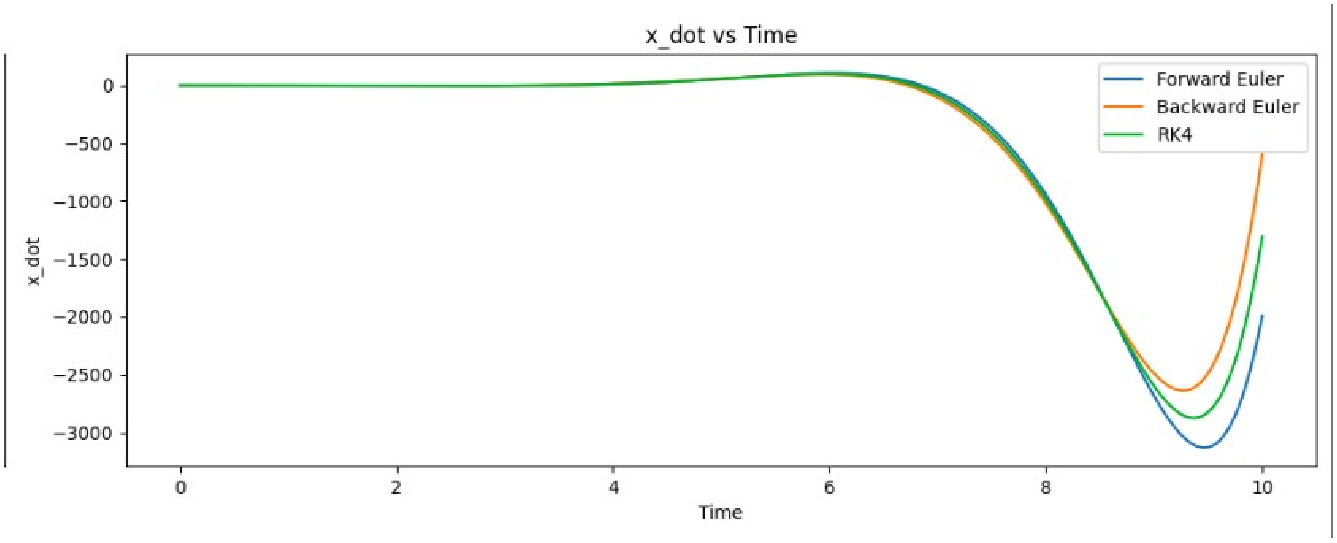
```
def backward_euler(fun, x0, Tf, h, tol=1e-8, max_iter=100):
    """
    Implicit Euler integration method using fixed-point iteration
    """
    t = np.arange(0, Tf + h, h)
    x_hist = np.zeros((len(x0), len(t)))
    x_hist[:, 0] = x0

    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
```

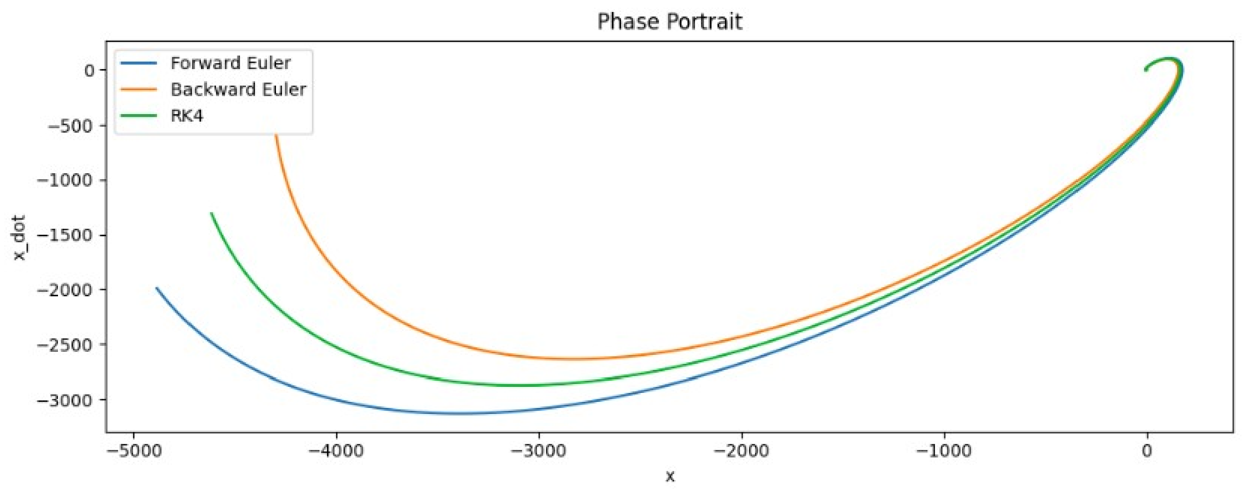
3.Results

3.1 Simulation Results

After executing our code, we get:



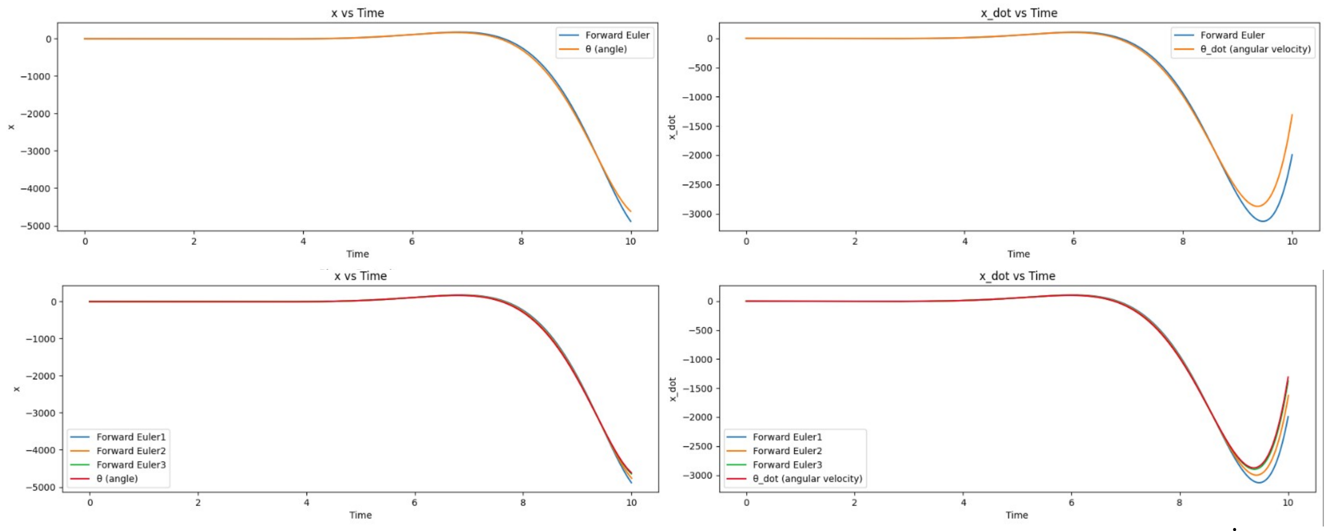




3.2 Simulation Results and Descriptive Analysis

Initially (from to $t=0$ to $t\simeq 8$), all three methods are in close agreement, showing that the system's dynamics were slow and easy to predict. **<u>However,</u>** a significant divergence occurs after $t\simeq 8$. At this point, the system enters a phase of rapid change, requiring the methods to make much larger and more complex predictions. By the end of the simulation, the trajectories have widely separated, resulting in three distinct final states

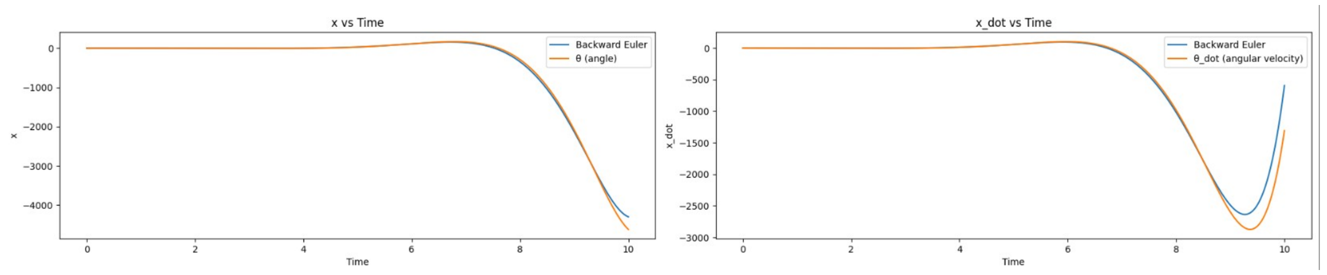
3.2.1 Forward Euler



- **Result:** The blue Forward Euler line shows a dramatic divergence, especially in the $(\dot{x})$ plot, where it severely undershoots the reference orange line.
- **Explanation:** Forward Euler is a simple, explicit method that is only **conditionally stable**. when the system becomes hard and stiff, the fixed time step used in the simulation is likely too large for the method to handle the sudden changes. This causes the method to become unstable, leading to a massive, accumulating error that drives the predicted trajectory far away from the true path.

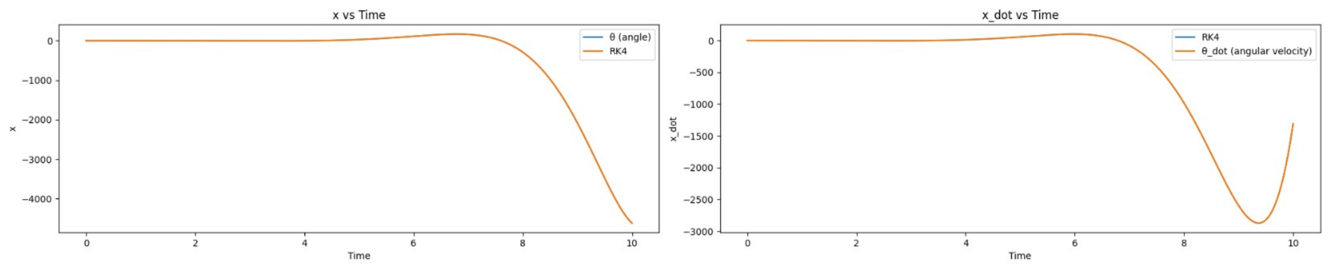
  The second image shows how the accuracy of the Forward Euler method changes as the time step (h) is reduced. while using $h=0.001$ yields a much more accurate result than$h=0.01$, it comes with a high computational cost. The $h=0.001$ simulation requires **ten times** as many calculations as the $h=0.01$ simulation, which is a key consideration when choosing a step size for real-world applications.

3.2.2 Backward Euler



- **Result:** The Backward Euler line also deviates from the orange reference, but its final state is noticeably less extreme than the Forward Euler result.
- **Explanation:** Backward Euler is an implicit method known for its strong stability. However, because it is still a low-precision method, the accumulated error is significant, causing it to deviate from the true path. It sacrifices accuracy for stability.

3.2.3 RK4



- **Result:** RK4 method is the closest to the true solution. It is the most accurate method because it takes multiple "peek-ahead" checks within each time step to calculate a highly precise average direction.

3.2.4 Phase Portrait Analysis

1. **Forward Euler :** Starts further out (near x≈−5000), meaning it has a larger initial error and is less stable. as its trajectory moves further away from the other two methods at the start (on the left side of the graph). In numerical analysis, this signifies instability, which causse the solution to diverge or overestimate/underestimate the true path more significantly.
2. **Backward Euler :** This method is generally known to be more stable than Forward Euler. Its trajectory sits between the Forward Euler and the RK4 methods, suggesting it provides a better approximation than Forward Euler but is not as accurate as RK4.The Backward Euler solution has the highest magnitude on the $\dot{x}$ *axis* and the shortest path on the x axis (x≈−4200,4300). shorter initial path suggests it successfully controlled the error and remained bounded

3. **RK4 :** it is the most accurate of the three methods shown for a given step size. Its trajectory is consistently closer to the likely true path of the system. It follows the overall shape of the system's dynamics .