Solve Second-Order Differential Linear Equation

Ahmad Ahmad

November 2025

1 Introduction

Formulating a mathematical model of a real-world problem either through intuitive reasoning about the phenomenon or from a physical law based on evidence from experiments. The mathematical model often takes the form of a differential equation, that is, an equation that contains an unknown function and some of its derivatives. This is not surprising because in a real-world problem we often notice that changes occur and we want to predict future behavior on the basis of how current values change.

A second-order linear differential equation has the form :

$$p(y)x^{''} + Q(y)x^{'} + R(y)x = G(y) \tag{1}$$

if $G(y) \neq 0$,Equation (1) is **nonhomogeneous**.
The following is the solutions of the using both analytical and numerical methods for this equation with coefficients :

$$9.38x^{''} - 8.02x^{'} - 4.56x = 5.04 \tag{2}$$

The equation (2) is **nonhomogeneous and linear**

2 Analytical method

To solve second-order nonhomogeneous linear differential equations with constant coefficients, that is, equations of the form :

$$ax^{''} + bx^{'} + cx = d \tag{3}$$

where a, b, c and d are constants. The related homogeneous equation :

$$ax^{''} + bx^{'} + cx = 0 \tag{4}$$

is called the complementary equation and plays an important role in the solution of the original nonhomogeneous equation .
The general solution of the nonhomogeneous differential equation (1) can be written as :

$$x(y) = x_g(y) + x_p(y) \tag{5}$$

where x_p is a particular solution of Equation (3) and X_g is the general solution of the complementary Equation (4). The general solution for equation (3) is :

$$x_g(y) = c_1x_1(y) + c_2x_2(y) \tag{6}$$

We are looking for a function x such that a constant times its second derivative x'' plus another constant times x' plus a third constant times x is equal to 0.
We know that the exponential function $x = e^{ry}$ (where r is a constant) has the property that its derivative

is a constant multiple of itself: $x' = re^{ry}$. Furthermore, $x'' = r^2e^{ry}$. If we substitute these expressions into Equation (4), we see that $x = e^{ry}$ is a solution if

$$(ar^2 + br + c)e^{rt} = 0 \tag{7}$$

But e^{ry} is never 0. Thus $x = e^{ry}$ is a solution of Equation 5 if $r$is a root of the equation

$$(ar^2 + br + c) = 0 \tag{8}$$

Equation (8) is called the auxiliary equation (or characteristic equation) of the differential equation $ax^{''} + bx^{'} + cx = 0$. Notice that it is an algebraic equation that is obtained from the differential equation by replacing $x^{''}$ by$r2$, $x^{'}$ by r, and x by 1. Sometimes the roots $r1$ and $r2$ of the auxiliary equation can be found by factoring. In other cases they are found by using the quadratic formula:

$$r1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \tag{9}$$

$$r2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{10}$$

and we have 3 cases for roots :

1. Case 1 : $b^2 - 4ac > 0$
   In this case the roots $r1$ and $r2$ of the auxiliary equation are real and distinct, so $x_1 = e^{r1y}$ and $x_2 = e^{r2y}$ are two linearly independent solutions of equation (4).and the general solution is :

   $$x_g(y) = c_1e^{r_1y} + c_2e^{r_2y} \tag{11}$$

2. Case 2 : $b^2 - 4ac = 0$
   In this case $r_1 = r_2$; that is, the roots of the auxiliary equation are real and equal. Let's denote by r the common value of r_1 and r_2. Then, from Equations (9),(10), we have :

   $$r = \frac{-b}{2a} \tag{12}$$

   If the auxiliary equation (8) has only one real root r, then the general solution of equation (4) is :

   $$x_g(y) = c_1e^{ry} + c_2ye^{ry} \tag{13}$$

3. Case 3 : $b^2 - 4ac < 0$
   In this case the rootsr_1 and r_2 of the auxiliary equation are complex numbers.

   $$r_1 = \alpha + i\beta \qquad r_2 = \alpha - i\beta \tag{14}$$

   where is α and β is real number :

   $$\alpha = -b/2a \qquad \beta = \frac{\sqrt{4ac - b^2}}{2a} \tag{15}$$

   In this case the roots $r1$ and $r2$ of the auxiliary equation are complex number and the general solution is :

   $$x_g = e^{\alpha y}(c_1 \cos\beta y + c_2 sin\beta y) \tag{16}$$

An initial-value problem for the second-order Equation 1 or 2 consists of finding a solution y of the differential equation that also satisfies initial conditions of the form :

$$x(y0) = y0 \qquad x^{'}(y0) = y1 \tag{17}$$

And for particular solution we substitute into the equation (2). we can substitute both the particular and general solutions into equation (5). From the initial conditions, we can calculate c_1 and c_2.

Listing 1: Analytical method

```
def solve_second_order_differetial_equation (a,b,c,y,t,y_p):
 delta=b**2-(4*a*c)
 if delta >0 :
  r1= (-b+np.sqrt(delta))/(2*a)
  r2= (-b-np.sqrt(delta))/(2*a)
  x    = lambda t, C1, C2: C1*np.exp(r1*t) + C2*np.exp(r2*t)
  x_dot= lambda t, C1, C2: C1*r1*np.exp(r1*t) + C2*r2*np.exp(r2*t)
 elif delta==0 :
  r=-b/(2*a)
  x     = lambda t, C1, C2: (C1 + C2*t)*np.exp(r*t)
  x_dot = lambda t, C1, C2: C2*np.exp(r*t) + r*(C1 + C2*t)*np.exp(r*t)
 else :
    alpha = -b/(2*a)
    beta  = math.sqrt(-delta)/(2*a)
    x     = lambda t, C1, C2: np.exp(alpha*t)*(C1*np.cos(beta*t) + C2*np.sin(beta*t))
    x_dot = lambda t, C1, C2: np.exp(alpha*t)*((C2*beta + C1*alpha)*np.sin(beta*t) + (C2*
        alpha - C1*beta)*np.cos(beta*t))
#constant c1 & c2
 A = np.array([[x(0,1,0), x(0,0,1)],[x_dot(0,1,0), x_dot(0,0,1)]])
 B = np.array([y[0] - y_p, y[1]])
 C1, C2 = np.linalg.solve(A, B)
 #final solution
 t_vals=np.linspace(0,t,1001)
 x_total     = x(t_vals, C1, C2) + y_p
 x_dot_total = x_dot(t_vals, C1, C2)
 return (x_total,x_dot_total,t_vals)
```

3 Explicit Eular method (Forward Euler)

Euler's method consists of repeatedly evaluating equation (18), using the result of each step to execute the next step. In this way we obtain a sequence of values $y0, y1, y2, ..., yn, ...$ that approximate the values of the solution at the points $t0, t1, t2, ..., tn, ...$ In the Forward Euler method, the new value is computed using the slope at the current point. That is, the estimate depends only on the information available at the beginning of the step, making it a straightforward and easy-to-implement method. However, it can become unstable if the time step is too large.

$$x_{n+1} = x_n + hf_n \qquad n = 0, 1, 2 \ldots \tag{18}$$

Listing 2: Explicit Euler integration method

```
    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
```

4 Implicit Euler method (Backward Euler)

On the other hand, in the Backward Euler method, the new value is computed using the slope at the next point (which is not yet known). This makes the method implicit, meaning that it requires solving an equation at each step. Although slightly more complex to implement, it is much more stable, especially for problems involving stiff systems.

$$x_{n+1} = x_n + hf_n(t_{n+1}, x_{n+1}) \qquad n = 0, 1, 2 \dots \tag{19}$$

Listing 3: Implicit Euler integration method

```
    """
␣␣␣␣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
```

5 Rung-Kutta method

The Euler and improved Euler methods belong to what is now called the Runge-Kutta2 class of methods.This method is now called the classic fourth-order four-stage Runge-Kutta method, but it is often referred to simply as the Runge-Kutta method, and we will follow this practice for brevity. This method has a local truncation error that is proportional to h^5. Thus it is two orders of magnitude more accurate than the improved Euler method and three orders of magnitude better than the Euler method. It is relatively simple to use and is sufficiently accurate to handle many problems efficiently.

steps fo calculate Rung-kutta :

$$\begin{aligned}
k_1 &= f(t, x) \\
k_2 &= f(t + \frac{h}{2}, x + \frac{h}{2} * k1) \\
k_3 &= f(t + \frac{h}{2}, x + \frac{h}{2} * k2) \\
k_4 &= f(t + \frac{h}{2}, x + \frac{h}{2} * k3) \\
x &= x + \frac{h}{6} * (k_1 + 2 * k_2 + 2 * k_3 + k4) \\
t &= t + h
\end{aligned}$$

Listing 4: Implicit Euler integration method

```
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
```

6 Results and Dicsuss :

using analytical techniques such as integration or series expansions. Usually, the emphasis was on finding an exact expression for the solution. Unfortunately, there are many important problems in engineering and science, especially nonlinear ones, to which these methods either do not apply or are very complicated to use.

Table 1: Comparison of Numerical and analytical Methods for Solving ODE

Method	Accuracy	Stability	Computational Complexity
Forward Euler	High	Limited	Very simple
Backward Euler	Medium	Excellent	Moderate
Runge-Kutta	High	Very good	Moderate
Analytical Solution	Exact	Always stable	Very high

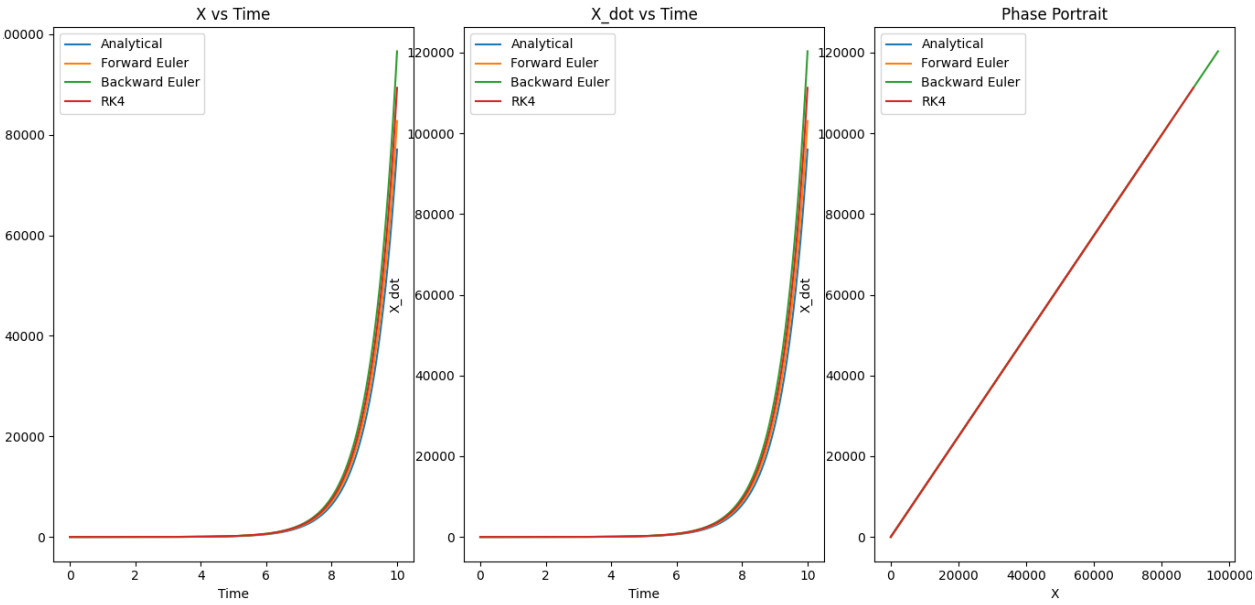

Figure 1: Results with step h=0.01

Figure 1 illustrate the numerical solutions compared with the analytical solution:
Forward Euler: Provides the closest approximation to the analytical solution for the current step size.
Runge-Kutta RK4: Very accurate, but slightly less aligned with the analytical solution compared to Forward Euler in this example.
Backward Euler: Shows more deviation from the analytical solution, though it remains stable.

Figure 2 shows that all numerical methods coincide closely with each other as the number of samples increases; however, a noticeable deviation from the analytical solution is still observed.

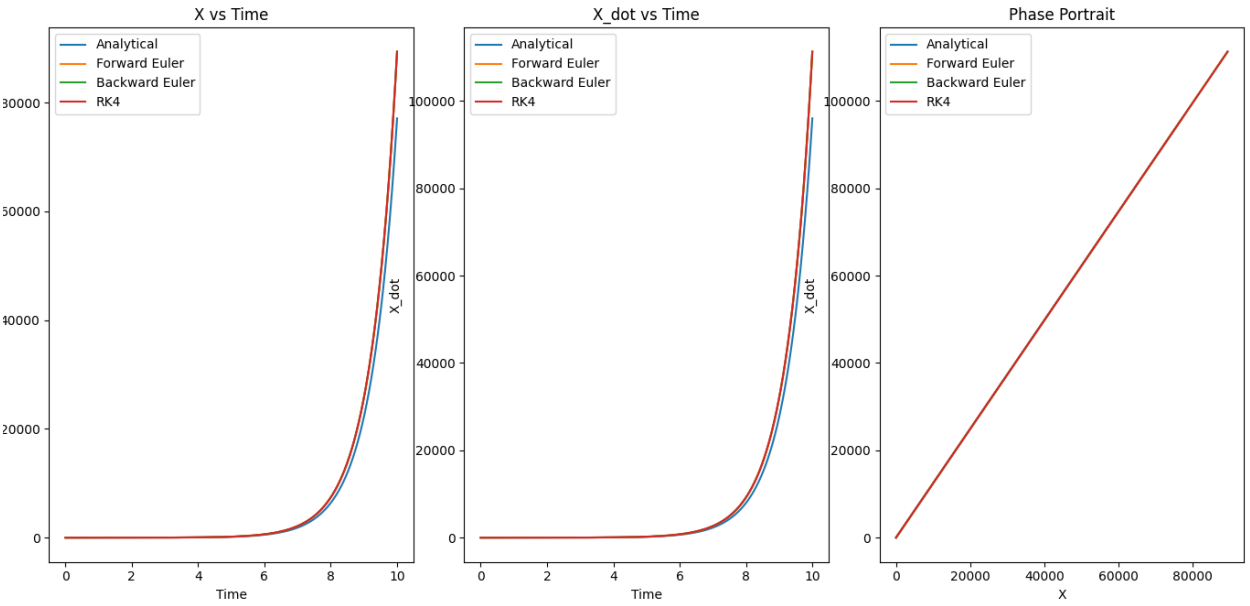

Figure 2: Results with step h=0.00001

Figure 3 shows a large step size, resulting in a noticeable divergence of the numerical solutions from the analytical solution. In this case, the Runge-Kutta method provides the closest approximation to the analytical solution, whereas both the Forward and Backward Euler methods exhibit significant errors and deviations.

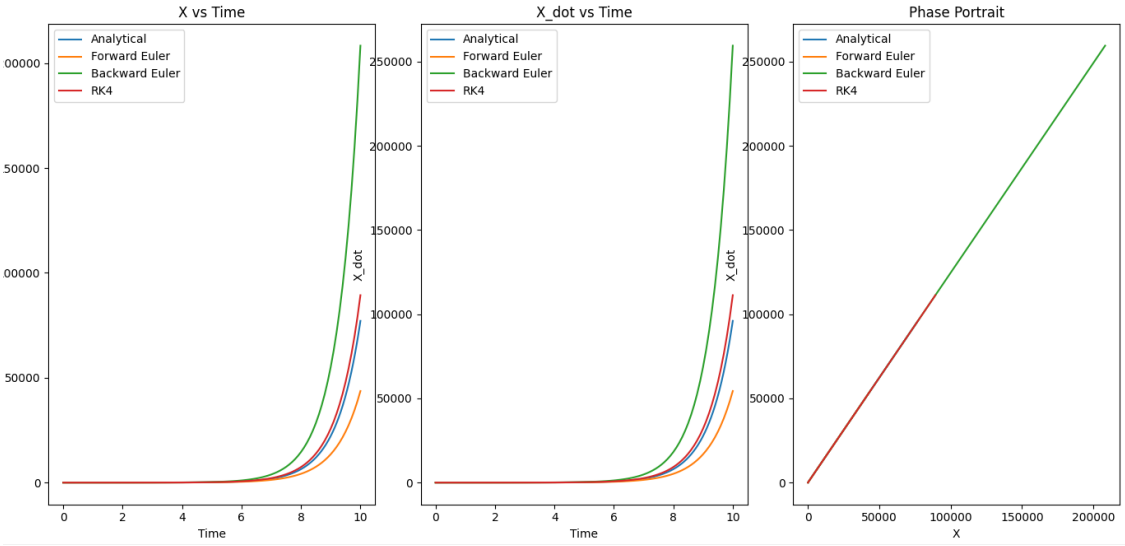

Figure 3: Results with step h=0.1

References

[1] A. James Stewart, *CALCULUS 8th*, 2015.

[2] b. BOYCE DIPRIMA MEADE, *Elementary Differential Equations and Boundary Value Problems 11th*, 2017.

A Program Code Appendix

A.1 Numerical and Analytical Methods for Second-Order Differential Equation

```
1
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import cmath
5
6  def second_order_differetial_equation (y):
7   #a*x_ddot+b*x_dot+c*x==d
8   #x_ddot+(b/a)*x_dot+(c/a)*x==(d/a)
9   x=y[0]
10  x_dot=y[1]
11  x_ddot= -(b/a)*x_dot-(c/a)*x+(d/a)
12  return np.array([x_dot,x_ddot])
13
14 def solve_second_order_differetial_equation (a,b,c,y,t,y_p):
15  delta=b**2-(4*a*c)
16
17  if delta >0 :
18   r1= (-b+np.sqrt(delta))/(2*a)
19   r2= (-b-np.sqrt(delta))/(2*a)
20   x    = lambda t, C1, C2: C1*np.exp(r1*t) + C2*np.exp(r2*t)
21   x_dot= lambda t, C1, C2: C1*r1*np.exp(r1*t) + C2*r2*np.exp(r2*t)
22
23  elif delta==0 :
24   r=-b/(2*a)
25   x     = lambda t, C1, C2: (C1 + C2*t)*np.exp(r*t)
26   x_dot = lambda t, C1, C2: C2*np.exp(r*t) + r*(C1 + C2*t)*np.exp(r*t)
27
28  else :
29     alpha = -b/(2*a)
30     beta  = cmath.sqrt(-delta)/(2*a)
31     x     = lambda t, C1, C2: np.exp(alpha*t)*(C1*np.cos(beta*t) + C2*np.sin(beta*t))
32     x_dot = lambda t, C1, C2: np.exp(alpha*t)*((C2*beta + C1*alpha)*np.sin(beta*t) + (C2*
           alpha - C1*beta)*np.cos(beta*t))
33
34
35 # constant c1 & c2
36  A = np.array([[x(0,1,0), x(0,0,1)],[x_dot(0,1,0), x_dot(0,0,1)]])
37  B = np.array([y[0] - y_p, y[1]])
38  C1, C2 = np.linalg.solve(A, B)
39
40  #final solution
41  t_vals=np.linspace(0,t,1001)
42  x_total     = x(t_vals, C1, C2) + y_p
43  x_dot_total = x_dot(t_vals, C1, C2)
44
45  return (x_total,x_dot_total,t_vals)
46
47 def pendulum_dynamics(x):
48     """
49 ␣␣␣␣Pendulum␣dynamics:␣d    /dt  ␣=␣-(g/l)␣*␣sin(  )
50 ␣␣␣␣State␣vector␣x␣=␣[  ,␣d  /dt]
51 ␣␣␣␣"""
52     l = 1.0
53     g = 9.81
54
55     theta = x[0]
56     theta_dot = x[1]
57
```

```
    theta_ddot = -(g/l) * np.sin(theta)

    return np.array([theta_dot, theta_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

    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):
        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


# Test all integrators
x0 = np.array([0.1, 0.1])  # Initial state: [x,x_dot]
Tf = 10.0
h = 0.00001
#PARAMTERS FOR EQUATION

a=  9.38
b= -8.02
c= -4.56
d=  5.04
y_p=c/d
```

```
126
127
128 #Analytical method
129 x,x_dot,t = solve_second_order_differetial_equation(a,b,c,x0,Tf,y_p)
130
131 # Forward Euler
132 x_fe, t_fe = forward_euler(second_order_differetial_equation, x0, Tf, h)
133
134 # Backward Euler
135 x_be, t_be = backward_euler(second_order_differetial_equation, x0, Tf, h)
136
137 # Runge-Kutta 4
138 x_rk4, t_rk4 = runge_kutta4(second_order_differetial_equation, x0, Tf, h)
139
140
141 # Plot results
142 plt.figure(figsize=(24, 8))
143 # fig  for X solution
144 plt.subplot(1, 3, 1)
145 plt.plot(t, x, label='Analytical')
146 plt.plot(t_fe, x_fe[0, :], label='Forward␣Euler')
147 plt.plot(t_be, x_be[0, :], label='Backward␣Euler')
148 plt.plot(t_rk4, x_rk4[0, :], label='RK4')
149 plt.xlabel('Time')
150 plt.ylabel('X')
151 plt.legend()
152 plt.title('X␣vs␣Time')
153 # fig for X_dot
154 plt.subplot(1, 3, 2)
155 plt.plot(t, x_dot, label='Analytical')
156 plt.plot(t_fe, x_fe[1, :], label='Forward␣Euler')
157 plt.plot(t_be, x_be[1, :], label='Backward␣Euler')
158 plt.plot(t_rk4, x_rk4[1, :], label='RK4')
159 plt.xlabel('Time')
160 plt.ylabel('X_dot')
161 plt.legend()
162 plt.title('X_dot␣vs␣Time')
163
164 plt.subplot(1, 3, 3)
165 plt.plot(x, x_dot, label='Analytical')
166 plt.plot(x_fe[0, :], x_fe[1, :], label='Forward␣Euler')
167 plt.plot(x_be[0, :], x_be[1, :], label='Backward␣Euler')
168 plt.plot(x_rk4[0, :], x_rk4[1, :], label='RK4')
169 plt.xlabel('X')
170 plt.ylabel('X_dot')
171 plt.legend()
172 plt.title('Phase␣Portrait')
173
174 plt.tight_layout()
175 plt.show()
```