

# Mass spring damper system Pendulum

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.

Let us examine the formulation of a mathematical model for the pendulum illustrated in the figure.

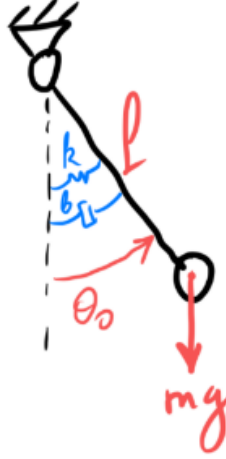


Figure 1: Pendulum schema

to write the motion equations of this system we begin deriving Lagrangian for the system:

$$\mathcal{L}(\mathcal{X}, \mathcal{X}') = \mathcal{K}(\mathcal{X}') - \mathcal{P}(\mathcal{X}, \mathcal{X}') \quad (1)$$

kinetic Energy in this system :

$$K = 0.5I\theta'^2 \quad (2)$$

Potential Energy in this system :

$$P = mgl \cos \theta + 0.5k\theta^2 \quad (3)$$

and by substituting into Equation (1) :

$$\mathcal{L} = 0.5I\theta'^2 - mgl \cos \theta - 0.5k\theta^2 \quad (4)$$

The Lagrangian equations for mechanical system :

$$\frac{d}{dt} \left( \frac{\partial L}{\partial \theta'} \right) - \frac{\partial L}{\partial \theta} = Q \quad (5)$$

where  $Q$  is the external force acting on the system , which is a damping in this case :  $Q = -b\theta'$   
The partial derivative with respect  $\theta$  :

$$\frac{\partial L}{\partial \theta} = mgl \sin \theta - k\theta \quad (6)$$

The partial derivative with respect  $\theta'$  :

$$\frac{\partial L}{\partial \theta'} = I\theta' \quad (7)$$

the full time derivative :

$$\frac{d}{dt} \left( \frac{\partial L}{\partial \theta'} \right) = I\theta'' \quad (8)$$

Finally the equations of motion for spring damper pendulum system :

$$I\theta'' - mgl \sin(\theta) + k\theta = -b\theta' \quad (9)$$

and we can rewrite equation (9) in the next form :

$$-b\theta' - I\theta'' + mgl \sin(\theta) - k\theta = 0 \quad (10)$$

and this equation description of the motion of system and it is non linear

## 2 Analytical method

To solve second-order nonhomogeneous non linear differential equations with constant coefficients, that is, equations of the form in (10). and as we see the equation is non linear because we have  $\sin$  in the gravity term . and in this case we can not solve it by analytical method . In particular case when  $\theta$  is small  $\sin(\theta) \simeq \theta$  the equation (10) become :

$$-I\theta'' - b\theta' + (mgl - k)\theta = 0 \quad (11)$$

and the equation become linear :

$$ax'' + bx' + cx = d \quad (12)$$

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

$$ax'' + bx' + cx = 0 \quad (13)$$

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 :

$$x(y) = x_g(y) + x_p(y) \quad (14)$$

Listing 1: Analytical method

```

1 def solve_second_order_differetial_equation (a,b,c,y,t):
2     delta=b**2-(4*a*c)
3
4     if delta >0 :
5         r1= (-b+np.sqrt(delta))/(2*a)
6         r2= (-b-np.sqrt(delta))/(2*a)
7         x      = lambda t, C1, C2: C1*np.exp(r1*t) + C2*np.exp(r2*t)
8         x_dot= lambda t, C1, C2: C1*r1*np.exp(r1*t) + C2*r2*np.exp(r2*t)
9     elif delta==0 :
10        r=-b/(2*a)
11        x      = lambda t, C1, C2: (C1 + C2*t)*np.exp(r*t)
12        x_dot = lambda t, C1, C2: C2*np.exp(r*t) + r*(C1 + C2*t)*np.exp(r*t)
13    else :
14        alpha = -b/(2*a)

```

```

15     beta = cmath.sqrt(-delta)/(2*a)
16     x     = lambda t, C1, C2: np.exp(alpha*t)*(C1*np.cos(beta*t) + C2*np.sin(beta*t))
17     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))
18 # constant c1 & c2
19 A = np.array([[x(0,1,0), x(0,0,1)], [x_dot(0,1,0), x_dot(0,0,1)]])
20 B = np.array([y[0], y[1]])
21 C1, C2 = np.linalg.solve(A, B)
22 #final solution
23 t_vals=np.linspace(0,t,1001)
24 x_total      = x(t_vals, C1, C2)
25 x_dot_total  = x_dot(t_vals, C1, C2)
26 return (x_total,x_dot_total,t_vals)

```

### 3 Explicit Euler method (Forward Euler)

Euler's method consists of repeatedly evaluating equation (15), using the result of each step to execute the next step. In this way we obtain a sequence of values  $y_0, y_1, y_2, \dots, y_n, \dots$  that approximate the values of the solution at the points  $t_0, t_1, t_2, \dots, t_n, \dots$ . 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.

$$x_{n+1} = x_n + hf_n \quad n = 0, 1, 2 \dots \quad (15)$$

Listing 2: Explicit Euler integration method

```

1     def forward_euler(fun, x0, Tf, h):
2         """
3         Explicit Euler integration method
4         """
5         t = np.arange(0, Tf + h, h)
6         x_hist = np.zeros((len(x0), len(t)))
7         x_hist[:, 0] = x0
8
9         for k in range(len(t) - 1):
10             x_hist[:, k + 1] = x_hist[:, k] + h * fun(x_hist[:, k])
11
12         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.

$$x_{n+1} = x_n + hf_n(t_{n+1}, x_{n+1}) \quad n = 0, 1, 2 \dots \quad (16)$$

Listing 3: Implicit Euler integration method

```

1     """
2     Implicit Euler integration method using fixed-point iteration
3     """
4     t = np.arange(0, Tf + h, h)
5     x_hist = np.zeros((len(x0), len(t)))
6     x_hist[:, 0] = x0
7
8     for k in range(len(t) - 1):

```

```

9         x_hist[:, k + 1] = x_hist[:, k] # Initial guess
10
11     for i in range(max_iter):
12         x_next = x_hist[:, k] + h * fun(x_hist[:, k + 1])
13         error = np.linalg.norm(x_next - x_hist[:, k + 1])
14         x_hist[:, k + 1] = x_next
15
16         if error < tol:
17             break
18
19     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 :

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

Listing 4: Implicit Euler integration method

```

1 def runge_kutta4(fun, x0, Tf, h):
2     """
3     4th order Runge-Kutta integration method
4     """
5     t = np.arange(0, Tf + h, h)
6     x_hist = np.zeros((len(x0), len(t)))
7     x_hist[:, 0] = x0
8
9     for k in range(len(t) - 1):
10         k1 = fun(x_hist[:, k])
11         k2 = fun(x_hist[:, k] + 0.5 * h * k1)
12         k3 = fun(x_hist[:, k] + 0.5 * h * k2)
13         k4 = fun(x_hist[:, k] + h * k3)
14
15         x_hist[:, k + 1] = x_hist[:, k] + (h / 6.0) * (k1 + 2*k2 + 2*k3 + k4)
16
17     return x_hist, t

```

## 6 Results and Dicsuss :

In Figure (2), we obtain the results for solving the nonlinear equation by replacing  $\sin(\theta$  with  $\theta$  for small angles , and it is solved like a homogeneous equation. In Figure (3), we have the correct numerical solution of

the equation; in this case, the analytical solution is very complicated. We can observe the effect of damping in the solutions, where energy is gradually lost until the pendulum reaches the vertical pose and comes to a complete stop.

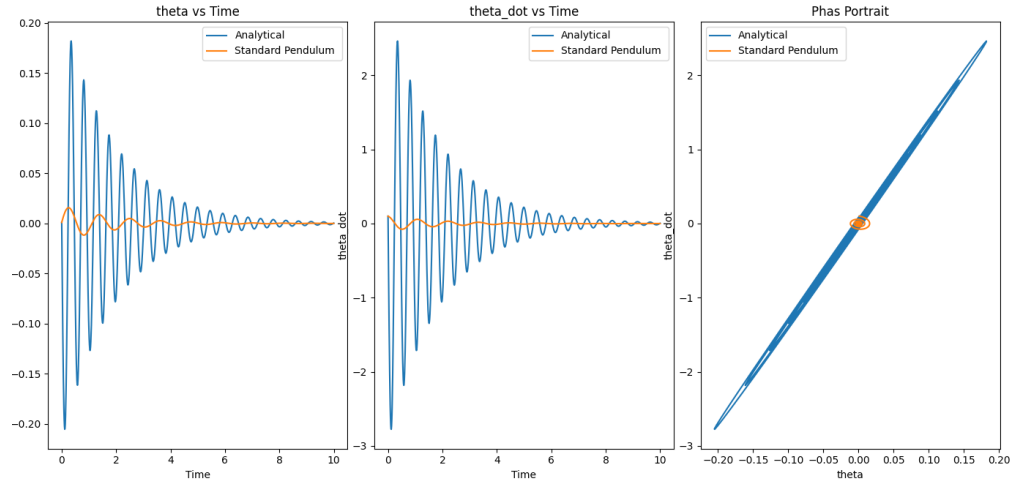


Figure 2: Analytical method

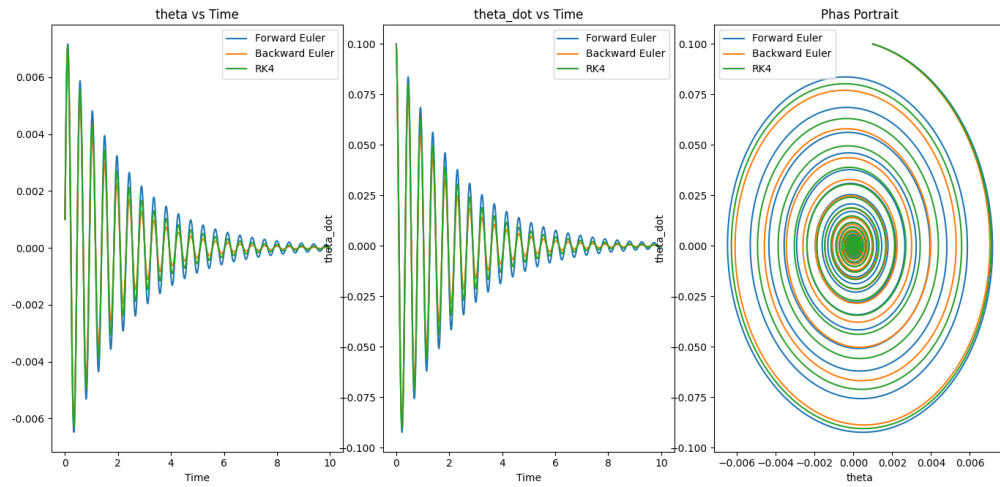


Figure 3: Numerical methods

## References

- [1] A. James Stewart, *CALCULUS 8th*, 2015.
- [2] b. BOYCE DIPRIMA MEADE, *Elementary Differential Equations and Boundary Value Problems 11th*, 2017.
- [3] c. Ivan.B ,Simulation of Robotic System course, ITMO University ,2025.

# A Program Code Appendix

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

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import cmath
4
5 def second_order_differetial_equation_pendulum (y):
6     #a*x_ddot+b*x_dot+c*x==d
7     #x_ddot+(b/a)*x_dot+(c/a)*x==(d/a)
8     theta=y[0]
9     theta_dot=y[1]
10    theta_dd=(-b/I)*theta_dot-(k/I)*theta+(m*g*l/I)*np.sin(theta)
11    return np.array([theta_dot,theta_dd])
12
13 def solve_second_order_differetial_equation (a,b,c,y,t):
14     delta=b**2-(4*a*c)
15
16     if delta >0 :
17         r1= (-b+np.sqrt(delta))/(2*a)
18         r2= (-b-np.sqrt(delta))/(2*a)
19         x      = lambda t, C1, C2: C1*np.exp(r1*t) + C2*np.exp(r2*t)
20         x_dot= lambda t, C1, C2: C1*r1*np.exp(r1*t) + C2*r2*np.exp(r2*t)
21
22     elif delta==0 :
23         r=-b/(2*a)
24         x      = lambda t, C1, C2: (C1 + C2*t)*np.exp(r*t)
25         x_dot = lambda t, C1, C2: C2*np.exp(r*t) + r*(C1 + C2*t)*np.exp(r*t)
26
27     else :
28         alpha = -b/(2*a)
29         beta  = cmath.sqrt(-delta)/(2*a)
30         x      = lambda t, C1, C2: np.exp(alpha*t)*(C1*np.cos(beta*t) + C2*np.sin(beta*t))
31         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))
32
33
34 # constant c1 & c2
35 A = np.array([[x(0,1,0), x(0,0,1)],[x_dot(0,1,0), x_dot(0,0,1)])]
36 B = np.array([y[0] , y[1]])
37 C1, C2 = np.linalg.solve(A, B)
38
39 #final solution
40 t_vals=np.linspace(0,t,1001)
41 x_total      = x(t_vals, C1, C2)
42 x_dot_total = x_dot(t_vals, C1, C2)
43
44 return (x_total,x_dot_total,t_vals)
45
46 def pendulum_dynamics(x):
47     """
48     Pendulum dynamics: d /dt u=-(g/l)*sin( )
49     State vector x=[ , d /dt]
50     """
51     l = 1.0
52     g = 9.81
53
54     theta = x[0]
55     theta_dot = x[1]
56
57     theta_ddot = -(g/l) * np.sin(theta)
58
59     return np.array([theta_dot, theta_ddot])
60
61 def forward_euler(fun, x0, Tf, h):
62     """
```

```

63  """Explicit Euler integration method
64  """
65  t = np.arange(0, Tf + h, h)
66  x_hist = np.zeros((len(x0), len(t)))
67  x_hist[:, 0] = x0
68
69  for k in range(len(t) - 1):
70      x_hist[:, k + 1] = x_hist[:, k] + h * fun(x_hist[:, k])
71
72  return x_hist, t
73
74  def backward_euler(fun, x0, Tf, h, tol=1e-8, max_iter=100):
75      """
76      Implicit Euler integration method using fixed-point iteration
77      """
78      t = np.arange(0, Tf + h, h)
79      x_hist = np.zeros((len(x0), len(t)))
80      x_hist[:, 0] = x0
81
82      for k in range(len(t) - 1):
83          x_hist[:, k + 1] = x_hist[:, k] # Initial guess
84
85          for i in range(max_iter):
86              x_next = x_hist[:, k] + h * fun(x_hist[:, k + 1])
87              error = np.linalg.norm(x_next - x_hist[:, k + 1])
88              x_hist[:, k + 1] = x_next
89
90              if error < tol:
91                  break
92
93          return x_hist, t
94
95  def runge_kutta4(fun, x0, Tf, h):
96      """
97      4th order Runge-Kutta integration method
98      """
99      t = np.arange(0, Tf + h, h)
100     x_hist = np.zeros((len(x0), len(t)))
101     x_hist[:, 0] = x0
102
103     for k in range(len(t) - 1):
104         k1 = fun(x_hist[:, k])
105         k2 = fun(x_hist[:, k] + 0.5 * h * k1)
106         k3 = fun(x_hist[:, k] + 0.5 * h * k2)
107         k4 = fun(x_hist[:, k] + h * k3)
108
109         x_hist[:, k + 1] = x_hist[:, k] + (h / 6.0) * (k1 + 2*k2 + 2*k3 + k4)
110
111     return x_hist, t
112
113  def standard_pendulum_equation(y):
114     l=0.31 #m
115     c=0.03 #Nms/rad
116     k=6.2 #Nm/rad
117     m=0.3 #kg
118     g=9.81
119     I=m*l*l
120
121
122     theta0 = y[0]
123     theta_dot0 = y[1]
124
125
126     k = m * g * l
127     omega_n = np.sqrt(k / I)
128     zeta = c / (2 * np.sqrt(I * k))
129     omega_d = omega_n * np.sqrt(1 - zeta**2)
130

```

```

131
132     t = np.linspace(0, 10, 1000)
133     A = theta0
134     B = (theta_dot0 + zeta * omega_n * theta0) / omega_d
135
136
137     theta = np.exp(-zeta * omega_n * t) * (A * np.cos(omega_d * t) + B * np.sin(omega_d * t)
138         )
139     theta_dot = np.gradient(theta, t)
140
141
142     return(theta_dot, theta, t)
143
144 # Test all integrators
145 #x0 = np.array([1.350926799, 0.1]) # Initial state: [theta0 ,theta0 dot]
146 x0 = np.array([0.001, 0.1])
147 Tf = 10.0
148 h = 0.001
149 #PARAMTERS FOR EQUATION
150 theta0=1.350926799 #rad
151 l=0.31 #m
152 b=0.03 #Nms/rad
153 k=6.2 #Nm/rad
154 m=0.3 #kg
155 g=9.81
156 I=m*l*l
157 a=I
158 c=(-m*g*l)+k
159
160 #Analytical method
161 x,x_dot,t = solve_second_order_differetial_equation(a,b,c,x0,Tf)
162 #standard module
163 theta_dot,theta,tt=standard_pendulum_equation(x0)
164 # Forward Euler
165 x_fe, t_fe = forward_euler(second_order_differetial_equation_pendulum, x0, Tf, h)
166
167 # Backward Euler
168 x_be, t_be = backward_euler(second_order_differetial_equation_pendulum, x0, Tf, h)
169
170 # Runge-Kutta 4
171 x_rk4, t_rk4 = runge_kutta4(second_order_differetial_equation_pendulum, x0, Tf, h)
172
173 # Plot results
174 plt.figure(figsize=(24, 8))
175 plt.clf()
176 # fig for X solution
177 plt.subplot(1, 3, 1)
178 plt.plot(t, x, label='Analytical')
179 plt.plot(tt, theta, label='Standard_Pendulum')
180 plt.plot(t_fe, x_fe[0, :], label='Forward_Euler')
181 plt.plot(t_be, x_be[0, :], label='Backward_Euler')
182 plt.plot(t_rk4, x_rk4[0, :], label='RK4')
183 plt.xlabel('Time')
184 plt.ylabel('theta')
185 plt.legend()
186 plt.title('theta vs Time')
187 # fig for X_dot
188 plt.subplot(1, 3, 2)
189 plt.plot(t, x_dot, label='Analytical')
190 plt.plot(tt, theta_dot, label='Standard_Pendulum')
191 plt.plot(t_fe, x_fe[1, :], label='Forward_Euler')
192 plt.plot(t_be, x_be[1, :], label='Backward_Euler')
193 plt.plot(t_rk4, x_rk4[1, :], label='RK4')
194 plt.xlabel('Time')
195 plt.ylabel('theta_dot')
196 plt.legend()
197 plt.title('theta_dot vs Time')

```

```

198
199 plt.subplot(1, 3, 3)
200 plt.plot(x, x_dot, label='Analytical')
201 plt.plot(theta, theta_dot, label='Standard Pendulum')
202 plt.plot(x_fe[0, :], x_fe[1, :], label='Forward Euler')
203 plt.plot(x_be[0, :], x_be[1, :], label='Backward Euler')
204 plt.plot(x_rk4[0, :], x_rk4[1, :], label='RK4')
205 plt.xlabel('theta')
206 plt.ylabel('theta_dot')
207 plt.legend()
208 plt.title('Phas Portrait')
209 plt.tight_layout()
210 plt.show()

```