Computational Physics Homework 8

Cristopher Cerda Puga

Runge-Kutta for higher-order ODEs.

We seek to extend the RK4 method to higher order differential equations. Let us consider the ODE:

$$x + x' + x'' + \cdots + x^{(n)} = y(t) \tag{1}$$

with $x(t_0) = x_0, x'(t_0) = x'_0, \ldots, x^{(n-1)}(t_0) = x_0^{(n-1)}$. We can use vector notation to write

$$\mathbf{x} = (x, x', x'', \ldots, x^{(n-1)})^\intercal = (x_1, x_2, \ldots, x_n)^\intercal,$$

so the ODE becomes

$$\mathbf{x}' = \begin{pmatrix} x'_1 \\ x'_2 \\ \vdots \\ x_{n-1} \\ x'_n \end{pmatrix} = \begin{pmatrix} x_2 \\ x_3 \\ \vdots \\ x_n \\ y(t) - x_n - x_{n-1} - \cdots - x_1 \end{pmatrix} = \mathbf{f}(\mathbf{x}, t) \tag{2}$$

and together with the initial conditions $\mathbf{x}(t_0) = (x_0, x'_0, \ldots, x_0^{(n-1)})^\intercal = \mathbf{x}_0$. The RK4 method becomes:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k_4}) \tag{3}$$

where

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(\mathbf{x}_n, t_n) \\ \mathbf{k}_2 &= \mathbf{f}(\mathbf{x}_n + \tfrac{h}{2}\mathbf{k}_1, t_n + \tfrac{h}{2}) \\ \mathbf{k}_3 &= \mathbf{f}(\mathbf{x}_n + \tfrac{h}{2}\mathbf{k}_2, t_n + \tfrac{h}{2}) \\ \mathbf{k}_4 &= \mathbf{f}(\mathbf{x}_n + h\mathbf{k}_3, t_n + h) \end{aligned}$$

Nonlinear pendulum

Runge-Kutta of Fourth order

Now, back to our problem. The nonlinear pendulum differential equations is

$$\ddot{\theta} = -\omega_0^2 \sin\theta \tag{4}$$

with $\omega_0 = \sqrt{\frac{g}{l}}$.

We can rewrite this nonlinear second order differential equation as a system of first order differential equations:

$$\dot{\theta} = \omega \tag{5}$$

$$\dot{\omega} = -\omega_0^2 \sin\theta \tag{6}$$

And looking forward the Runge-Kutta implementation, we can write this in vector notation as:

$$\begin{pmatrix} \dot{\theta} \\ \dot{\omega} \end{pmatrix} = \begin{pmatrix} \omega \\ -\omega_0^2 \sin\theta \end{pmatrix}$$

With this definition, now we can implement the method of Runge-Kutta of fourth order to solve this differential equation. First, we define our $\mathbf{f}(\mathbf{x}, t)$, where here $\mathbf{x} = (\theta, \omega)^\intercal$:

```
1 # Define the system of ODEs
2 def pendulum_system(t, x, g, L):
3     theta, omega = x
4     omega_prime = -(g / L) * np.sin(theta)
5     theta_prime = omega
6     return [theta_prime, omega_prime]
```

Then we set up the parameters, the maximum time number of steps and the step size

```
# Parameters
g = 9.8
L = 1.0
t0 = 0.0
theta0 = np.pi/2  # Initial angle
omega0 = 0.0  # Initial angular velocity

# Time settings
t_max = 10  # Maximum time
h = 0.01  # Step size
num_steps = int(t_max / h)
```

We then initialize three arrays: one for the time values, other for the θ values and another for the ω values. And then we set the initial conditions on the component of this vector:

```
# Initialize arrays
t_values = np.zeros(num_steps + 1)
theta_values = np.zeros(num_steps + 1)
omega_values = np.zeros(num_steps + 1)

# Initial conditions
t_values[0] = t0
theta_values[0] = theta0
omega_values[0] = omega0
```

Finally, we apply the Runge-Kutta method to each of ODE of our system:

```
def RungeKutta4(t, x, g, L):
# Solve the system using the Runge-Kutta method
    for i in range(num_steps):
        t = t_values[i]
        x = [theta_values[i], omega_values[i]]
        k1 = np.multiply(h, pendulum_system(t, x, g, L))
        k2 = np.multiply(h, pendulum_system(t + 0.5 * h, np.add(x, 0.5 * k1), g, L))
        k3 = np.multiply(h, pendulum_system(t + 0.5 * h, np.add(x, 0.5 * k2), g, L))
        k4 = np.multiply(h, pendulum_system(t + h, np.add(x, k3), g, L))
        x_new = np.add(x, (k1 + 2 * k2 + 2 * k3 + k4) / 6)
        t_values[i + 1] = t + h
        theta_values[i + 1] = x_new[0]
        omega_values[i + 1] = x_new[1]

    return t_values, theta_values,omega_values

t = []
theta = []
omega = []
t,theta,omega = RungeKutta4(t0, [theta0, omega0], g, L)
```

Now, we can plot the phase space, angle vs time and the energy of the system:

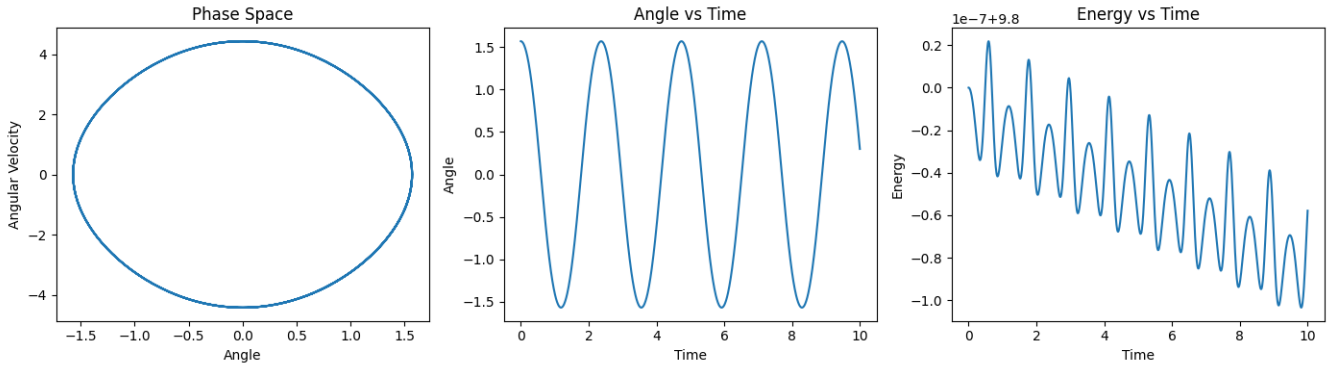

Figure 1: $\theta_0 = \pi/2, \omega_0 = 0.0$

We can see that even though nonlinear, the system carries some kind of periodic motion, as seen in the phase space plot. One can also notice that the energy of the system "change" over time, which does not mean that the system is not conservative, since in the phase space the particle follows a closed trajectory. One can see this in a more analytical way by returning to the original differential equation Eq.(4), which can be multiplied by a factor of $\dot{\theta}$

$$\dot{\theta}\ddot{\theta} + \omega_0^2 \sin\theta\dot{\theta} = 0, \tag{7}$$

which can be written as

$$\frac{\mathrm{d}}{\mathrm{d}t}\left[\frac{1}{2}\left(\frac{\mathrm{d}\theta}{\mathrm{d}t}\right)^2 - \omega_0^2\cos\theta\right] \tag{8}$$

Equation (8) corresponds to the conservation of the mechanical energy, as desired. Or if you get tricked by the plot of the energy, we can clearly see once we print the values of the energy that it is *almost* conserved over time:

```
        # Calculate and store the energy at each step
        energy_values[i + 1] = 0.5 * L * omega_values[i + 1]**2 + g * (1 - np.cos(↩
            theta_values[i + 1]))

    return energy_values

energy = []
energy = RungeKutta4(t0, [theta0, omega0], g, L)

print(energy)
>>> [9.8         9.8         9.8         ... 9.79999994 9.79999994 9.79999994]
```

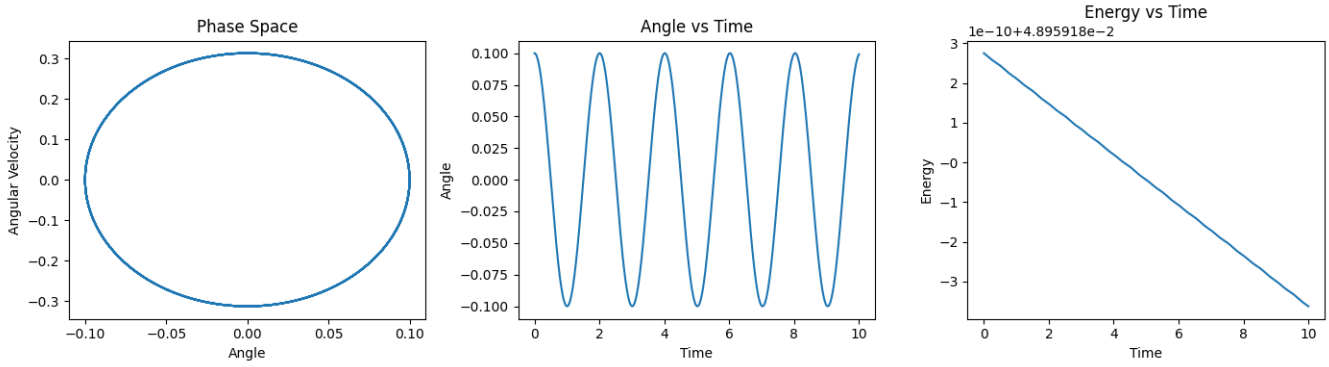

Figure 2: $\theta_0 = 0.1, \omega_0 = 0.0$

In this case the energy plot is better behaved, and the energy is also conserved.

Let us consider another limiting case: In this case, we can see the most interesting part in the first and

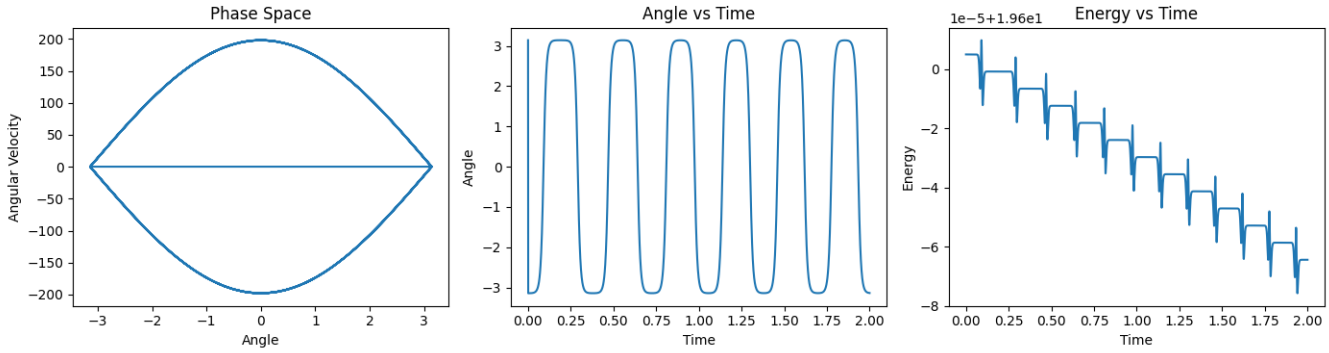

Figure 3: $\theta_0 = \pi, \omega_0 = 0.1$

second plot, it takes a slightly longer time to the pendulum to pass over $\theta = \pm\pi$. In the phase plane this translates into a rapidely decrease of angular speed as the pendulum approaches this values of the angle. In this case the energy is also conserved.

Runge-Kutta-Fehlberg

Now let us consider another kind of approach, one in which we utilize the knowledge we have of the error to our benefit. The summary of this is the next, we will use the fourth/fifth-order Runge Kutta method, developed by Fehlberg, in which the intermediate functions are as follows (in python):

```
1      k1 = f(t, y, g, L)
2      k2 = f(t + 0.25*h, y + 0.25*h*k1, g, L)
3      k3 = f(t + 3*h/8, y + 3*h*k1/32 + 9*h*k2/32, g, L)
4      k4 = f(t + 12*h/13, y + 1932*h*k1/2197 - 7200*h*k2/2197 + 7296*h*k3/2197, g, L)
5      k5 = f(t + h, y + 439*h*k1/216 - 8*h*k2 + 3680*h*k3/513 - 845*h*k4/4104, g, L)
6      k6 = f(t + 0.5*h, y - 8*h*k1/27 + 2*h*k2 - 3544*h*k3/2565 + 1859*h*k4/4104 - 11*↩
           h*k5/40, g, L)
```

With these definitions, the fourth order approximation is (again in python)

```
1      x_next = y + h*(25*k1/216 + 1408*k3/2565 + 2197*k4 /4104 - k5/5)
```

and the fifth order one

```
1      y_next = y + h*(16*k1/135 + 6656*k3/12825 + 28561*k4/56430 - 9*k5/50 + 2*k6/55)
```

We later evaluate the error with the difference of this two approximations

```
1       error = np.linalg.norm(y_new - x_new)
```

In this way, since it is a system of equations. And depending of the error, we redefine the step as

```
1      hnew = 0.9*h*(np.abs(h)*error/(x*(t0+h)-y*(t0+h)))**0.25
```

Putting all these definitions together and implementing it completely in python we end up with the next code: (see Appendix).
We can now explore the same initial conditions as in the previous method

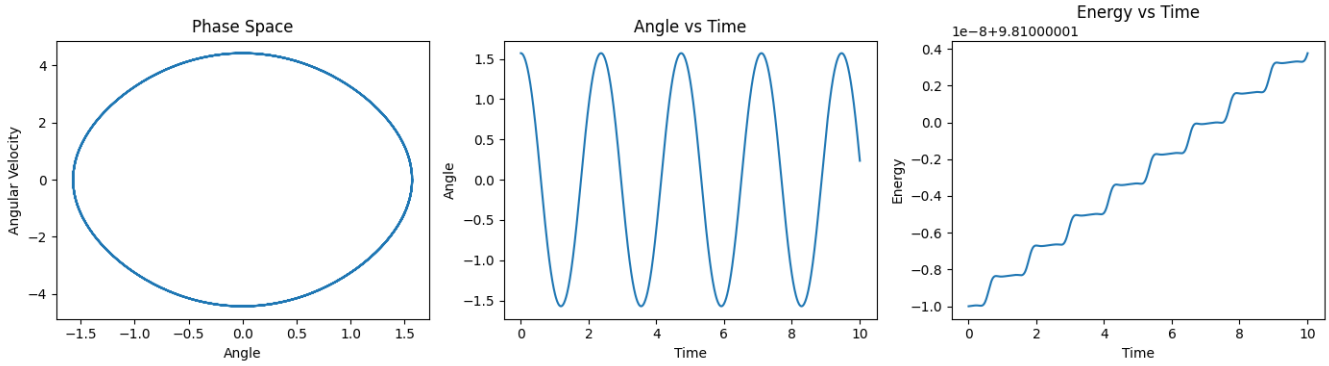

Figure 4: $\theta_0 = \pi/2, \omega_0 = 0.0$

We can see that there is a pretty big similarity with the RK4 method, the energy seems to increase, but in a very low order, so it is almost conserved. Also pretty similar results, with a slightly difference in the energy,

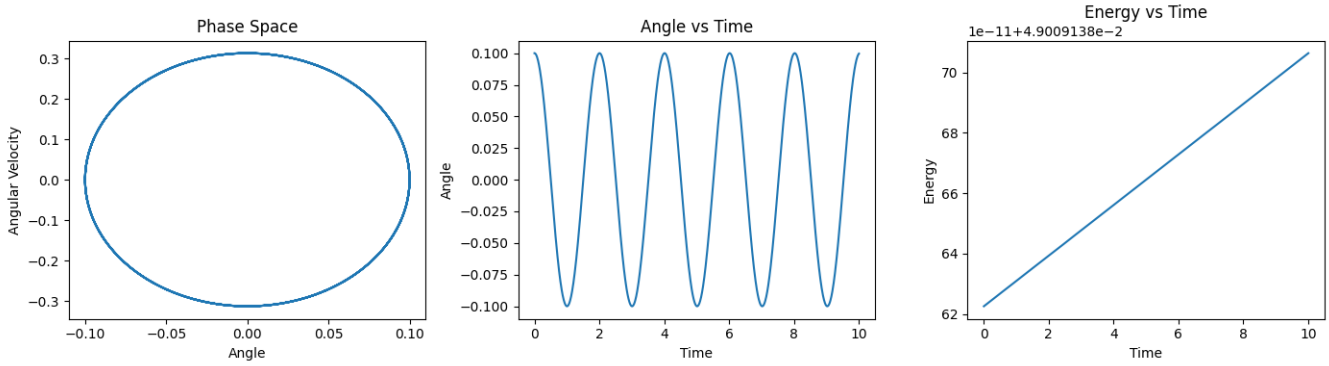

Figure 5: $\theta_0 = 0.1, \omega_0 = 0.0$

which is of course conserved. Now for the last one

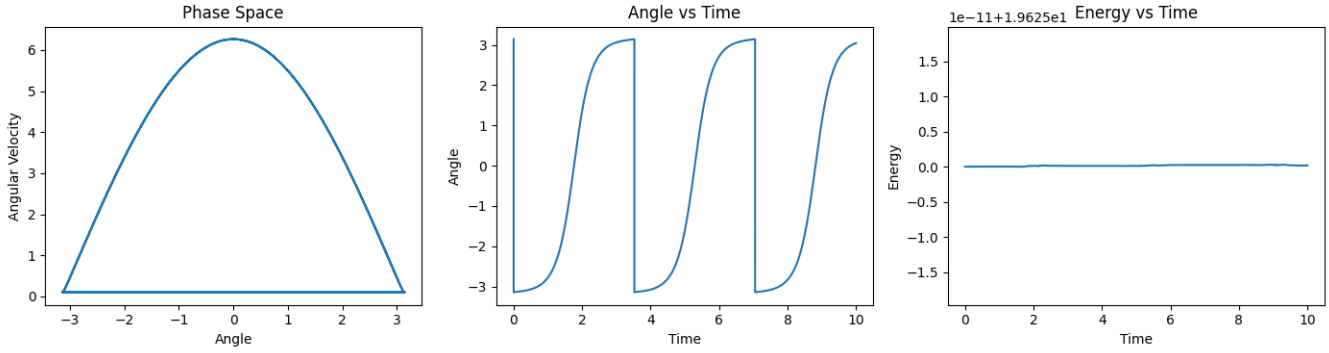

Figure 6: $\theta_0 = \pi, \omega_0 = 0.1$

In this case, for some reason that I don't understand yet, you can see that the phase plot is cut, and the angles have a discontinuous jump, though with the same behavior and energy conservation.

Euler-Crommer method

We need to do some modifications to this method in order to be able to use it in this problem. You can see it in the Appendix, it is just a mortification to apply it component-wise.

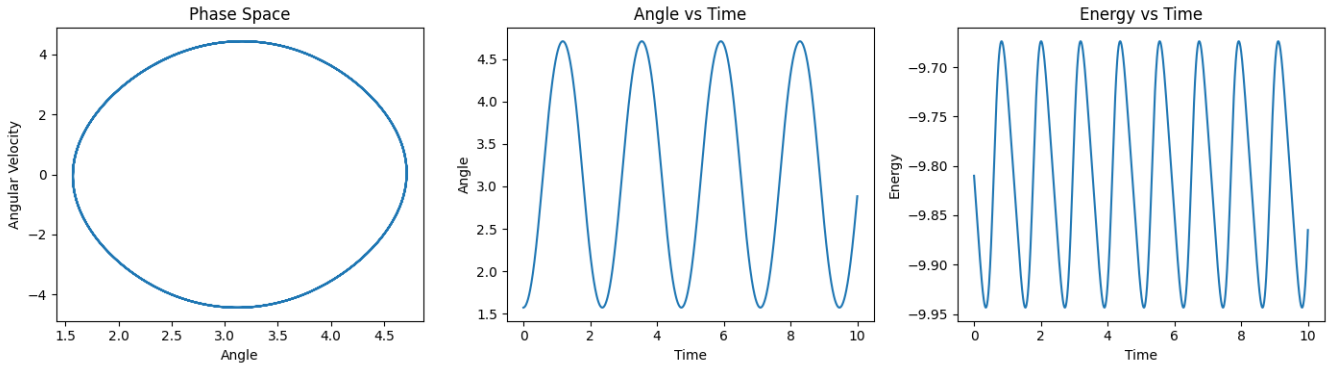

Figure 7: $\theta_0 = \pi/2, \omega_0 = 0.0$

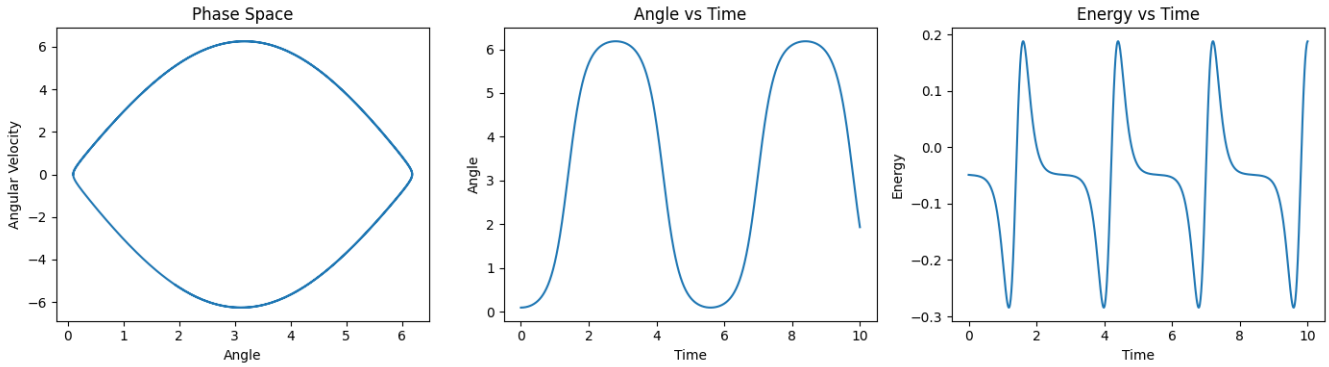

Figure 8: $\theta_0 = 0.1, \omega_0 = 0.0$

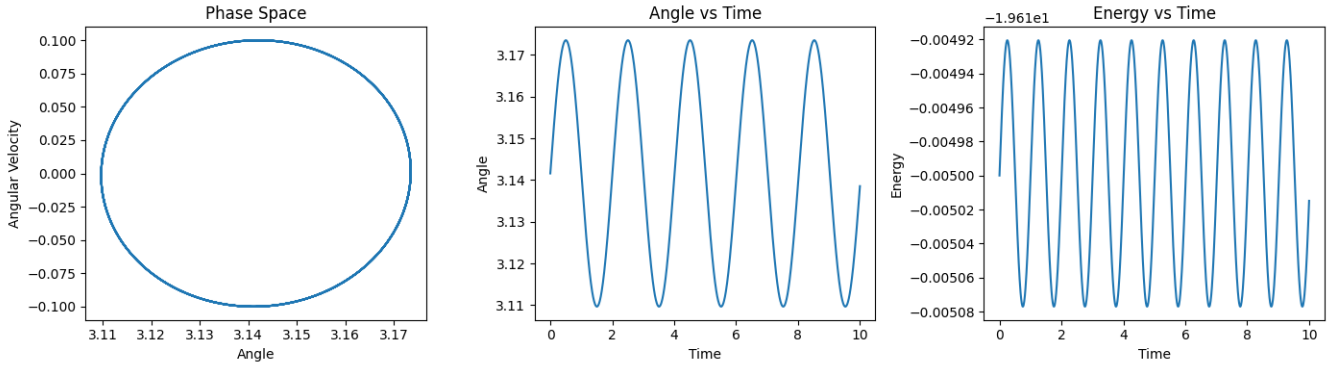

Figure 9: $\theta_0 = \pi, \omega_0 = 0.1$

Surprisingly, this method works as a somewhat good approximation of the behavior of the system.

Appendix

```
1  # RungeKutta4
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  # Define the system of ODEs
```

```
def pendulum_system(t, x, g, L):
    theta, omega = x
    theta_prime = omega
    omega_prime = -(g / L) * np.sin(theta)
    return [theta_prime, omega_prime]

# Parameters
g = 9.8
L = 1.0
t0 = 0.0
theta0 =  np.pi/2 # Initial angle
omega0 = 0.0  # Initial angular velocity
energy0 = 0.5 * L * omega0**2 + g * (1 - np.cos(theta0))


# Time settings
t_max = 10  # Maximum time
h = 0.01  # Step size
num_steps = int(t_max / h)

# Initialize arrays
t_values = np.zeros(num_steps + 1)
theta_values = np.zeros(num_steps + 1)
omega_values = np.zeros(num_steps + 1)

# Energy array to store the energy values
energy_values = np.zeros(num_steps + 1)
energy_values[0] = energy0
# Initial conditions
t_values[0] = t0
theta_values[0] = theta0
omega_values[0] = omega0

# Modify the angle to stay within [-pi, pi]
def enforce_periodicity(angle):
    return (angle + np.pi) % (2 * np.pi) - np.pi

def RungeKutta4(t, x, g, L):
# Solve the system using the Runge-Kutta method
    for i in range(num_steps):
        t = t_values[i]
        x = [enforce_periodicity(theta_values[i]), omega_values[i]]

        k1 = np.multiply(h, pendulum_system(t, x, g, L))
        k2 = np.multiply(h, pendulum_system(t + 0.5 * h, np.add(x, 0.5 * k1), g, L))
        k3 = np.multiply(h, pendulum_system(t + 0.5 * h, np.add(x, 0.5 * k2), g, L))
        k4 = np.multiply(h, pendulum_system(t + h, np.add(x, k3), g, L))
        x_new = np.add(x, (k1 + 2 * k2 + 2 * k3 + k4) / 6)
        t_values[i + 1] = t + h

        theta_values[i + 1] = x_new[0]
        omega_values[i + 1] = x_new[1]

        # Calculate and store the energy at each step
        energy_values[i + 1] = 0.5 * L * omega_values[i + 1]**2 + g * (1 - np.cos(↩
            theta_values[i + 1]))

    return t_values, theta_values,omega_values,energy_values

t = []
theta = []
```

```
omega = []
energy = []
t,theta,omega,energy = RungeKutta4(t0, [theta0, omega0], g, L)

print(energy)

# Create subplots in a single line
plt.figure(figsize=(14, 4))
plt.subplot(1, 3, 1)
plt.title('Phase Space')
plt.plot(theta_values, omega_values)
plt.xlabel('Angle')
plt.ylabel('Angular Velocity')

plt.subplot(1, 3, 2)
plt.title('Angle vs Time')
plt.plot(t_values, theta_values)
plt.xlabel('Time')
plt.ylabel('Angle')

plt.subplot(1, 3, 3)
plt.title('Energy vs Time')
plt.plot(t_values,energy)
plt.xlabel('Time')
plt.ylabel('Energy')

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

```
# runge_kutta_fehlberg
import numpy as np
import matplotlib.pyplot as plt

def pendulum_system(t, x, g, L):
    theta, omega = x
    theta_prime = omega
    omega_prime = -(g / L) * np.sin(theta)
    return np.array([theta_prime, omega_prime])


# Parameters
g = 9.81  # Acceleration due to gravity (m/s^2)
L = 1.0   # Length of the pendulum (m)
t0 = 0.0  # Initial time
theta0 =  np.pi/2 # Initial angle
omega0 = 0.0  # Initial angular velocity
energy0 = 0.5 * L * omega0**2 + g * (1 - np.cos(theta0))

# Time settings
t_max = 10.0  # Maximum time
h0 = 0.001  # Initial step size
t, y, x = t0, np.array([theta0, omega0]), np.array([theta0, omega0])

# Initialize arrays to store results
t_values = [t]
theta_values = [y[0]]
omega_values = [y[1]]
energy_values = [energy0]

def enforce_periodicity(x):
```

```
    x[0] = (x[0] + np.pi) % (2 * np.pi) - np.pi
    return x

def runge_kutta_fehlberg(f, t, y, h, g, L):
    k1 = f(t, y, g, L)
    k2 = f(t + 0.25*h, y + 0.25*h*k1, g, L)
    k3 = f(t + 3*h/8, y + 3*h*k1/32 + 9*h*k2/32, g, L)
    k4 = f(t + 12*h/13, y + 1932*h*k1/2197 - 7200*h*k2/2197 + 7296*h*k3/2197, g, L)
    k5 = f(t + h, y + 439*h*k1/216 - 8*h*k2 + 3680*h*k3/513 - 845*h*k4/4104, g, L)
    k6 = f(t + 0.5*h, y - 8*h*k1/27 + 2*h*k2 - 3544*h*k3/2565 + 1859*h*k4/4104 - 11*↩
        h*k5/40, g, L)

    x_next = y + h*(25*k1/216 + 1408*k3/2565 + 2197*k4 /4104 - k5/5)
    y_next = y + h*(16*k1/135 + 6656*k3/12825 + 28561*k4/56430 - 9*k5/50 + 2*k6/55)

    x_next = enforce_periodicity(x_next)
    y_next = enforce_periodicity(y_next)

    return t + h, y_next, x_next


while t < t_max:
    h = h0  # Initial step size

    # Attempt a step using RKF45
    t_new, y_new, x_new = runge_kutta_fehlberg(pendulum_system, t, y, h, g, L)

    # Calculate the estimated local error
    error = np.linalg.norm(y_new - x_new)

    # Adapt the step size based on the error
    if error < 1e-6:
        t = t_new
        y = y_new
        x = x_new
        t_values.append(t)
        theta_values.append(y[0])
        omega_values.append(y[1])

        energy_values = np.append(energy_values, 0.5 * L * omega_values[-1]**2 + g *↩
            (1 - np.cos(theta_values[-1])))

    #h0 = h*min(max(0.9 * (1e-6 / error) ** 0.2, 0.3), 2.0)
    hnew = 0.9*h*(np.abs(h)*error/(x*(t0+h)-y*(t0+h)))**0.25

# Convert lists to NumPy arrays
t_values = np.array(t_values)
theta_values = np.array(theta_values)
omega_values = np.array(omega_values)
energy = np.array(energy_values)

print(energy)


# Create subplots in a single line
plt.figure(figsize=(14, 4))
plt.subplot(1, 3, 1)
plt.title('Phase Space')
plt.plot(theta_values, omega_values)
plt.xlabel('Angle')
plt.ylabel('Angular Velocity')
```

```
91
92  plt.subplot(1, 3, 2)
93  plt.title('Angle vs Time')
94  plt.plot(t_values, theta_values)
95  plt.xlabel('Time')
96  plt.ylabel('Angle')
97
98  plt.subplot(1, 3, 3)
99  plt.title('Energy vs Time')
100 plt.plot(t_values, energy)
101 plt.xlabel('Time')
102 plt.ylabel('Energy')
103
104 plt.tight_layout()
105 plt.show()
```

```
1   # Cromer
2   import numpy as np
3   import matplotlib.pyplot as plt
4
5   # Define the differential equation for the nonlinear pendulum
6   def nonlinear_pendulum(t, theta, omega, g, L):
7       # Equations of motion
8       theta_dot = omega
9       omega_dot = g*np.sin(theta) / L
10      return theta_dot, omega_dot
11
12  # Set the initial conditions
13  t0 = 0.0
14  theta0 = np.pi/2
15  omega0 = 0.0
16
17  # Set the time step size and the number of iterations
18  dt = 0.01
19  num_iterations = 10**3
20
21  g = 9.81
22  L = 1.0
23
24  # Initialize arrays to store the solution
25  t_values = np.zeros(num_iterations + 1)
26  theta_values = np.zeros(num_iterations + 1)
27  omega_values = np.zeros(num_iterations + 1)
28  energy_values = np.zeros(num_iterations + 1)
29
30  # Perform the Euler-Cromer method
31  t_values[0] = t0
32  theta_values[0] = theta0
33  omega_values[0] = omega0
34  energy_values[0] = 0.5 * L * omega0**2 - g * L * (1 - np.cos(theta0))
35
36  for i in range(num_iterations):
37      t = t_values[i]
38      theta = theta_values[i]
39      omega = omega_values[i]
40      theta_dot, omega_dot = nonlinear_pendulum(t, theta, omega, g, L)
41
42      omega_new = omega + omega_dot * dt
43      theta_new = theta + omega_new * dt
44
```

```
    t_values[i + 1] = t + dt
    theta_values[i + 1] = theta_new
    omega_values[i + 1] = omega_new

    # Calculate the energy at each step
    energy_values[i + 1] = 0.5 * L * omega_new**2 - g * L * (1 - np.cos(theta_new))

# Create subplots for Phase Space, Angle vs Time, and Energy vs Time
plt.figure(figsize=(14, 4))

# Phase Space
plt.subplot(1, 3, 1)
plt.title('Phase Space')
plt.plot(theta_values, omega_values)
plt.xlabel('Angle')
plt.ylabel('Angular Velocity')

# Angle vs Time
plt.subplot(1, 3, 2)
plt.title('Angle vs Time')
plt.plot(t_values, theta_values)
plt.xlabel('Time')
plt.ylabel('Angle')

# Energy vs Time
plt.subplot(1, 3, 3)
plt.title('Energy vs Time')
plt.plot(t_values, energy_values)
plt.xlabel('Time')
plt.ylabel('Energy')

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