Homework 3

Oscillatory Motion and Chaos.

1.

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{l}\theta - 2\gamma\frac{\mathrm{d}\theta}{\mathrm{d}t} + \alpha_{\mathrm{D}}\sin\left(\Omega_{\mathrm{D}}t\right)$$

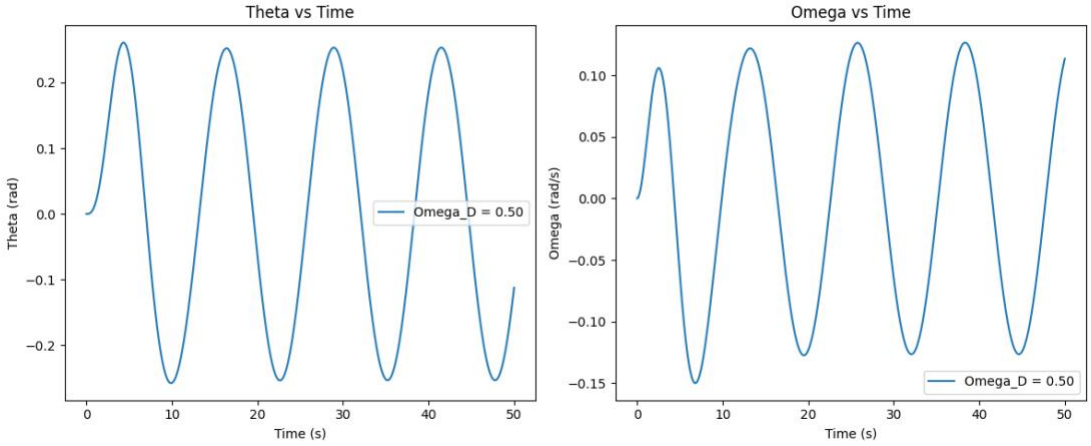
For a driven, damped harmonic oscillator, the resonance frequency is approximately the natural frequency of the undamped, undriven system. For a pendulum, the natural frequency in the small-angle approximation is:

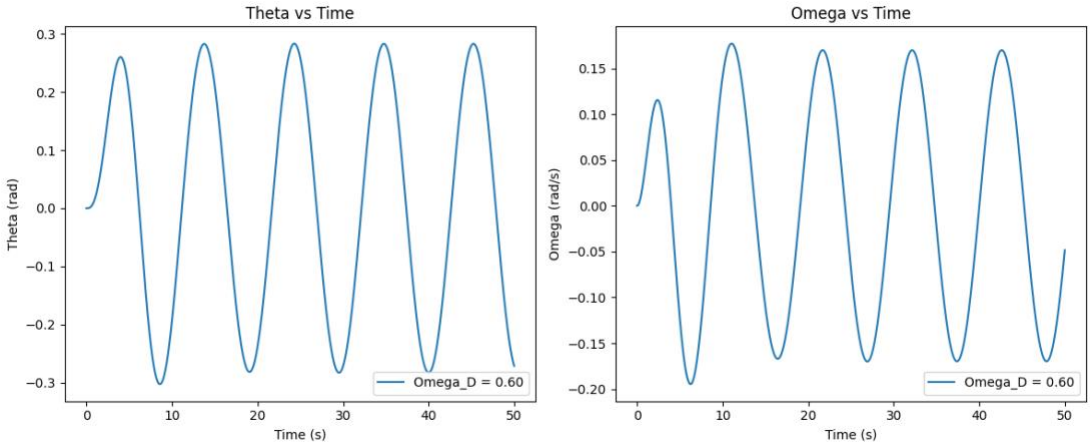
$$\omega_0 = \sqrt{\frac{g}{l}}\omega_0$$

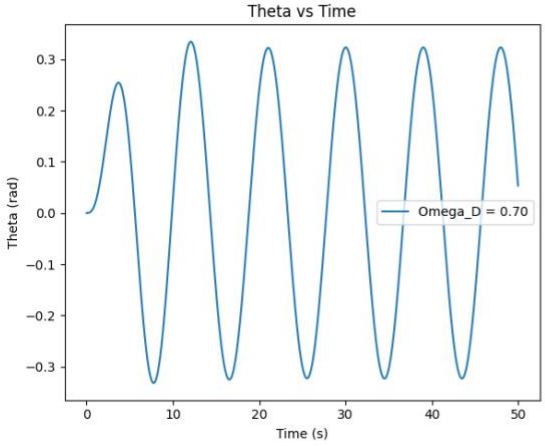
$$\omega_0 \approx \sqrt{1} = 1\mathrm{s}^{-1}$$

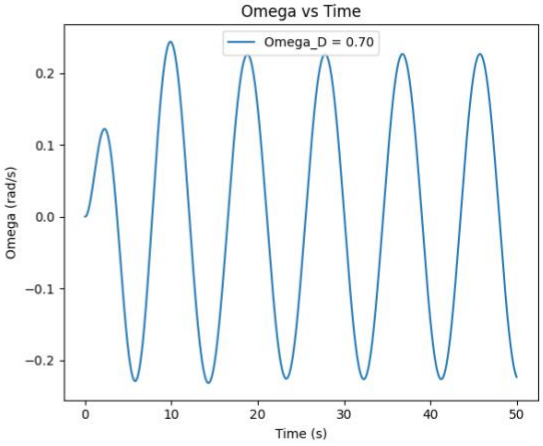
So, the resonance occurs approximately at $\Omega_D = 1\mathrm{s}^{-1}$. The small-angle approximation should be good unless the amplitude becomes too large, in which case nonlinear effects will dominate.

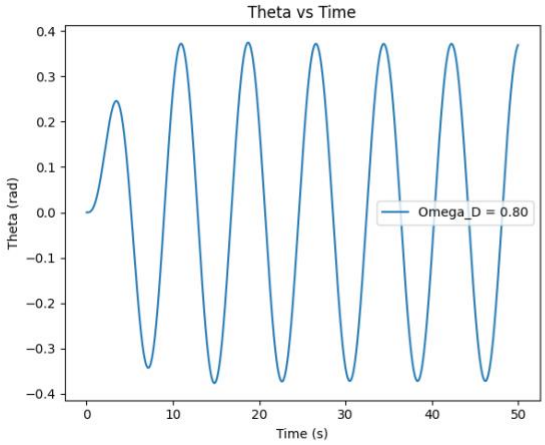
2.

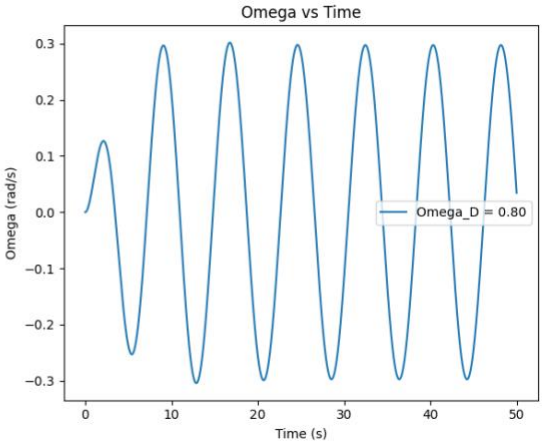


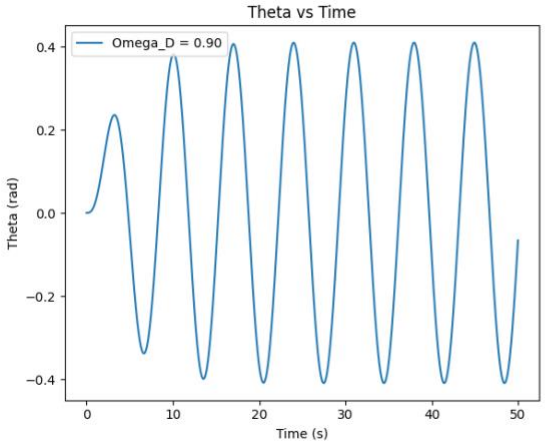


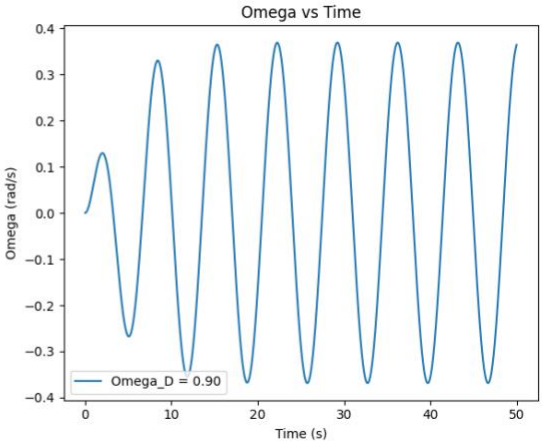

Theta vs Time
Theta (rad)
Time (s)
Omega_D = 0.70

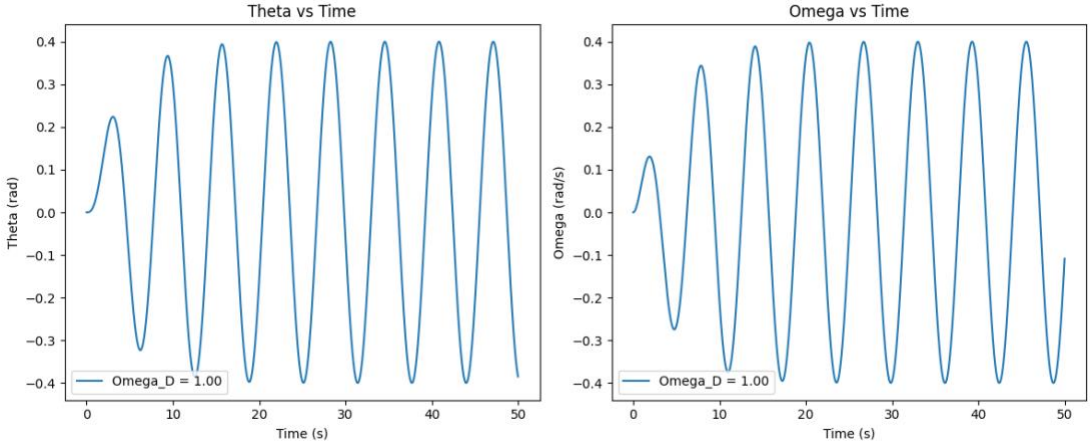

Omega vs Time
Omega (rad/s)
Time (s)
Omega_D = 0.70


Theta vs Time
Theta (rad)
Time (s)
Omega_D = 0.80

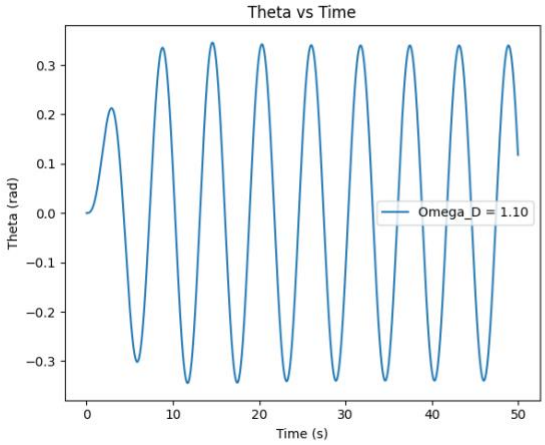

Omega vs Time
Omega (rad/s)
Time (s)
Omega_D = 0.80

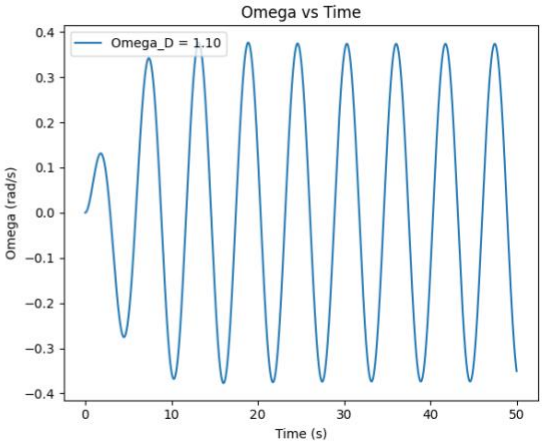

Theta vs Time
Theta (rad)
Time (s)
Omega_D = 0.90

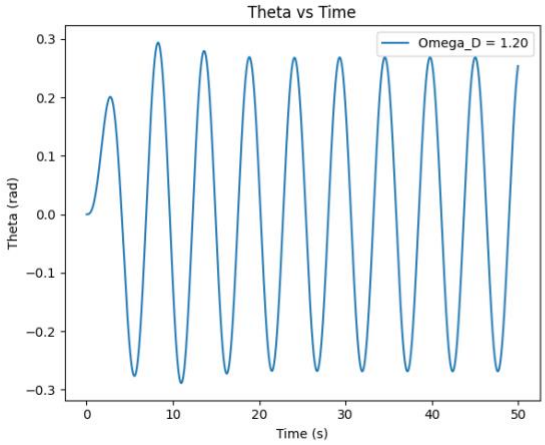

Omega vs Time
Omega (rad/s)
Time (s)
Omega_D = 0.90

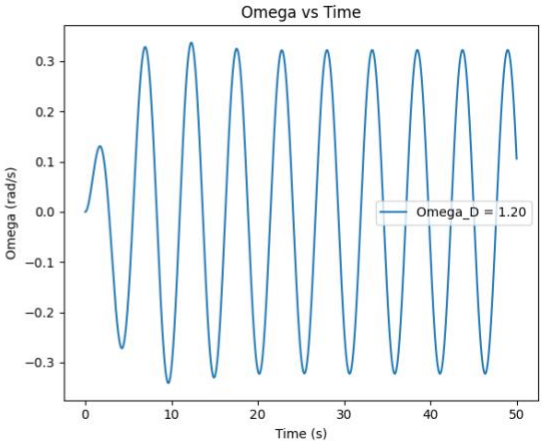

Theta vs Time
Theta (rad)
Time (s)
Omega_D = 1.00

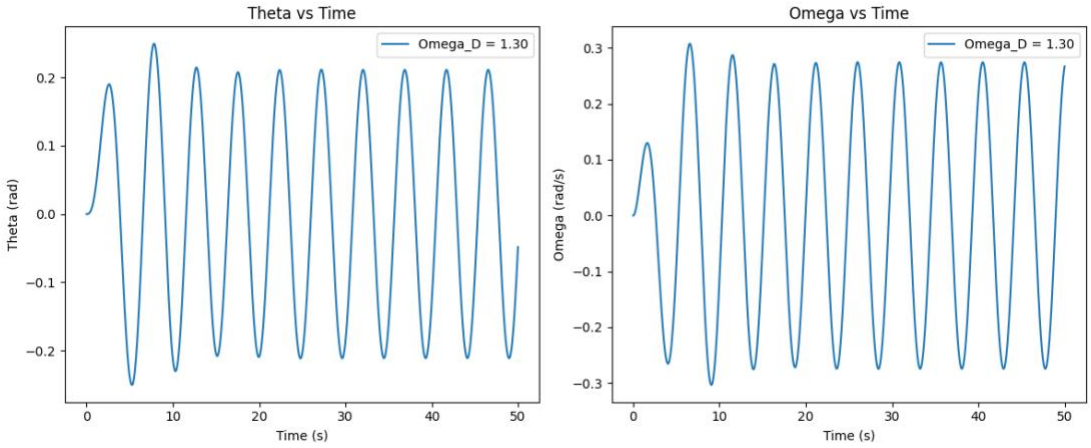
Omega vs Time
Omega (rad/s)
Time (s)
Omega_D = 1.00

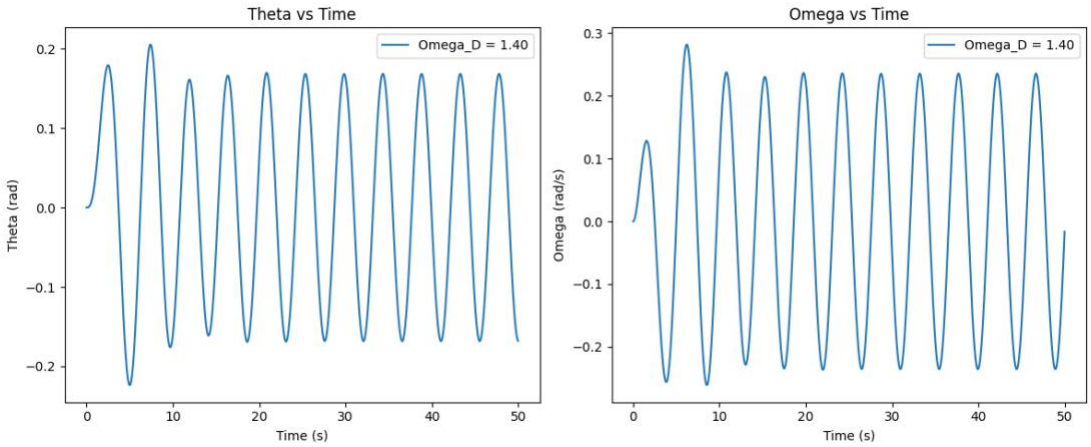

Theta vs Time
Theta (rad)
Time (s)
Omega_D = 1.10


Omega vs Time
Omega (rad/s)
Time (s)
Omega_D = 1.10


Theta vs Time
Theta (rad)
Time (s)
Omega_D = 1.20


Omega vs Time
Omega (rad/s)
Time (s)
Omega_D = 1.20



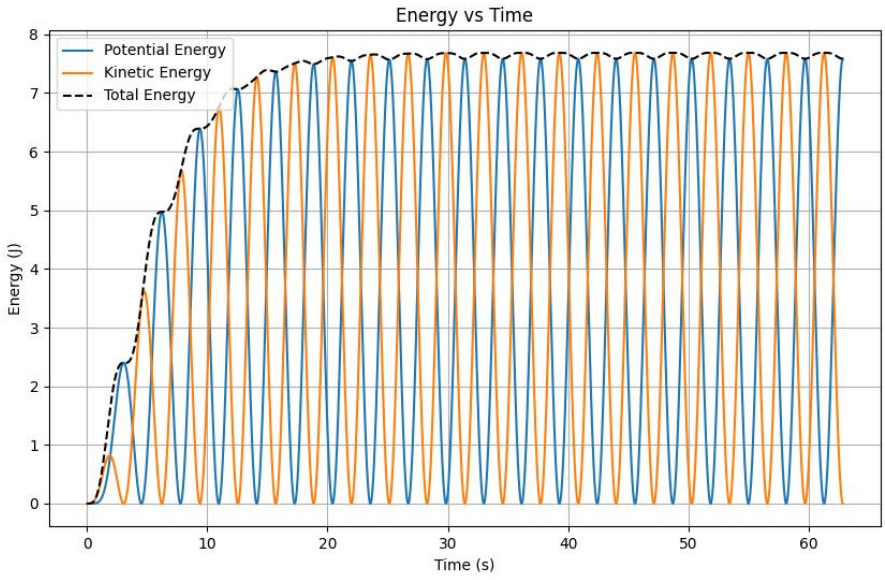


3.

$$\mathrm{PE} = mgl(1 - \cos(\theta))$$

$$\mathrm{KE} = \frac{1}{2}ml^2\omega^2$$

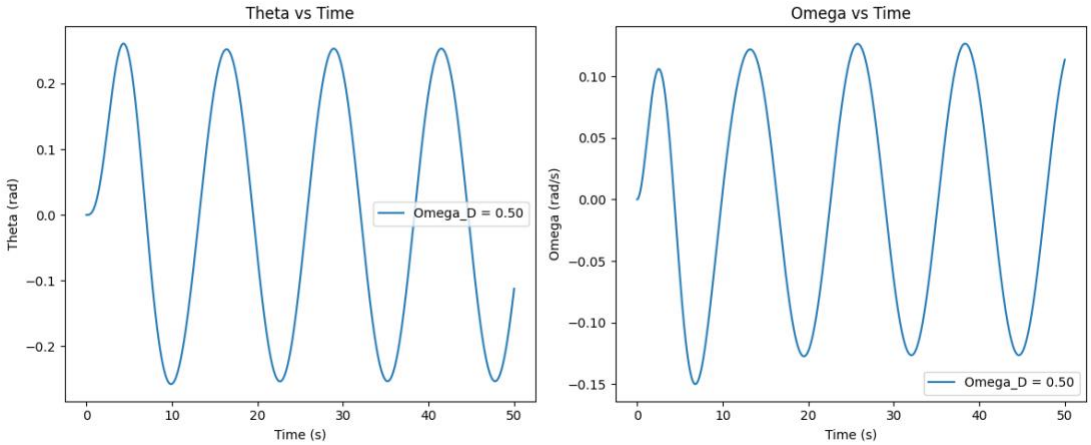
TE=PE+KE

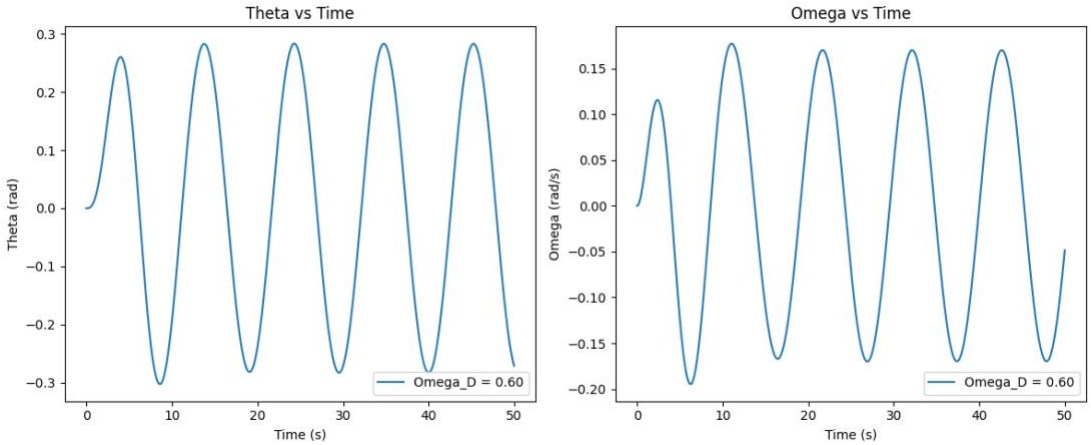


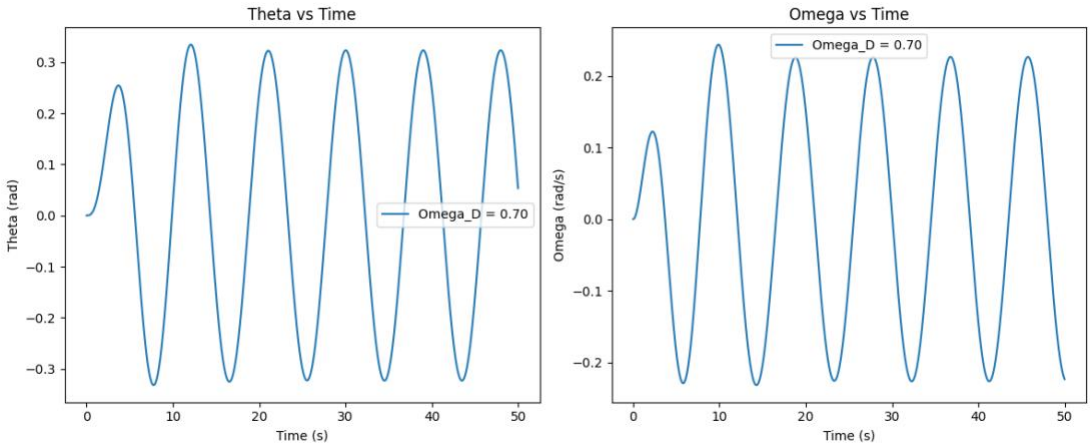
4.

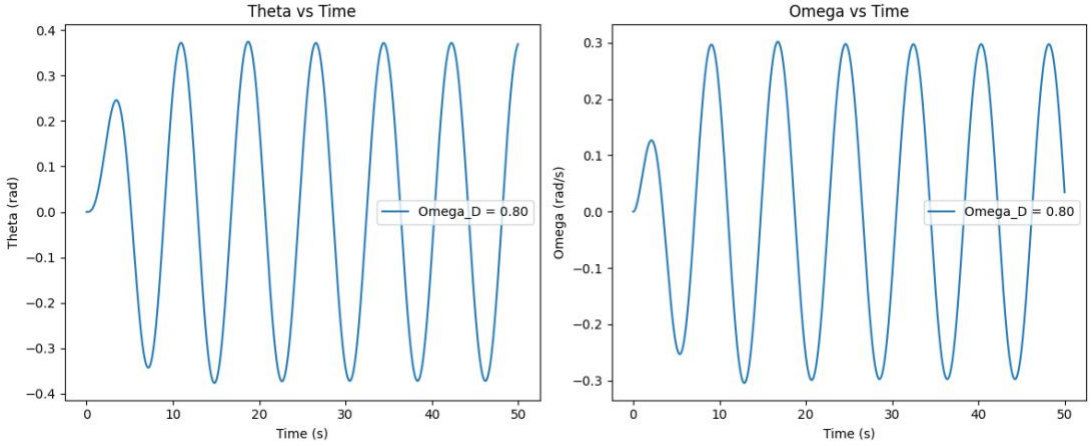
To account for non-linear effects, simply replace the θ term with $\sin(\theta)$ in the restoring force in your numerical method, modify the differential equation function:

```
def f_nonlinear(theta, omega, t, OmegaD):
    return omega, -g/l*np.sin(theta) - 2*gamma*omega + alphaD*np.sin(OmegaD*t)
```

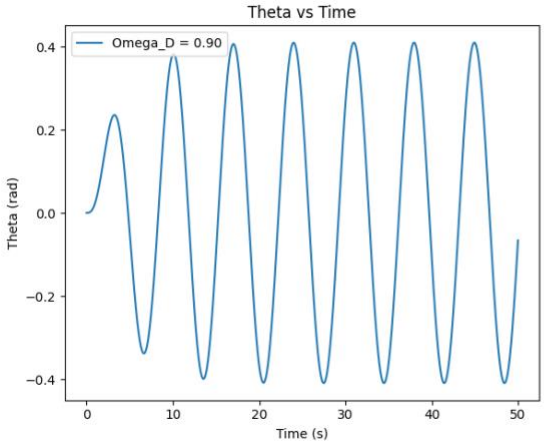


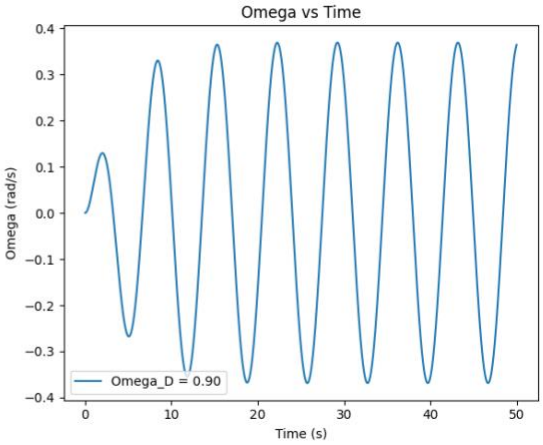


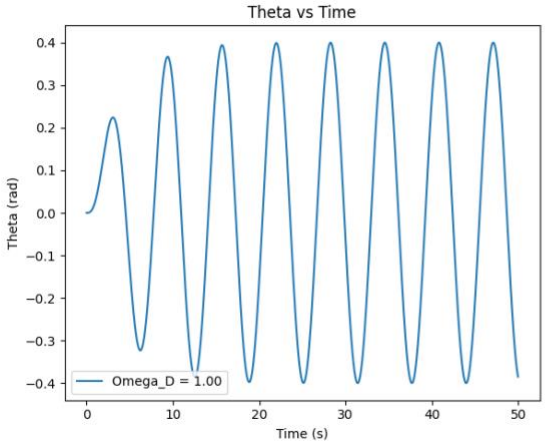


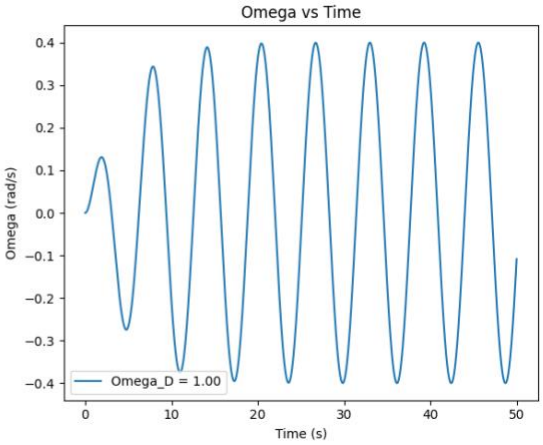

Theta vs Time
Theta (rad)
Time (s)
Omega_D = 0.80

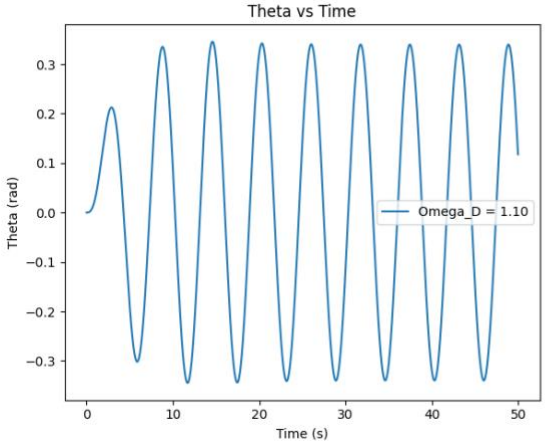
Omega vs Time
Omega (rad/s)
Time (s)
Omega_D = 0.80

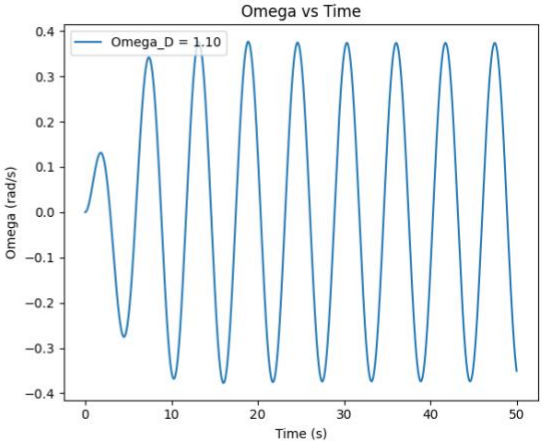

Theta vs Time
Theta (rad)
Time (s)
Omega_D = 0.90

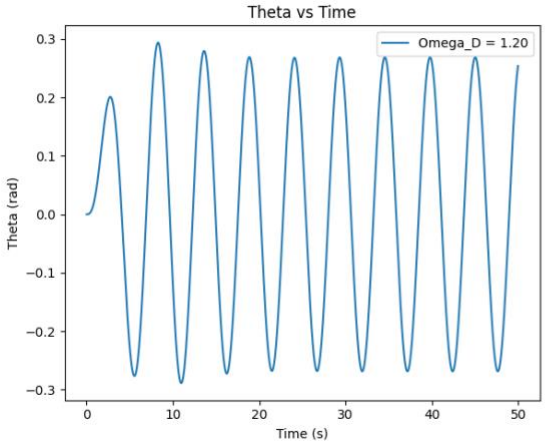

Omega vs Time
Omega (rad/s)
Time (s)
Omega_D = 0.90

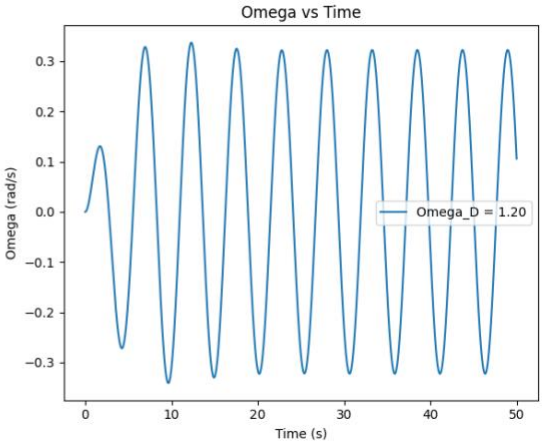

Theta vs Time
Theta (rad)
Time (s)
Omega_D = 1.00

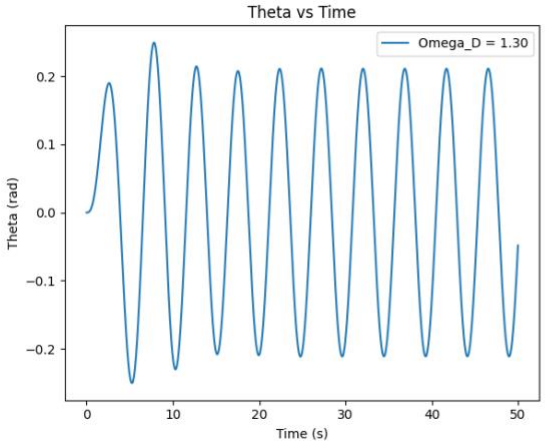

Omega vs Time
Omega (rad/s)
Time (s)
Omega_D = 1.00

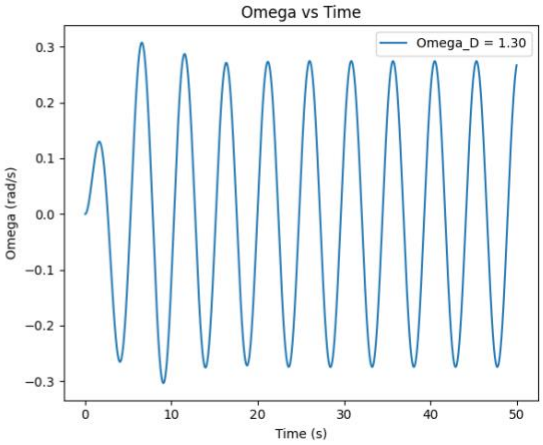

Theta vs Time
Theta (rad)
Time (s)
Omega_D = 1.10

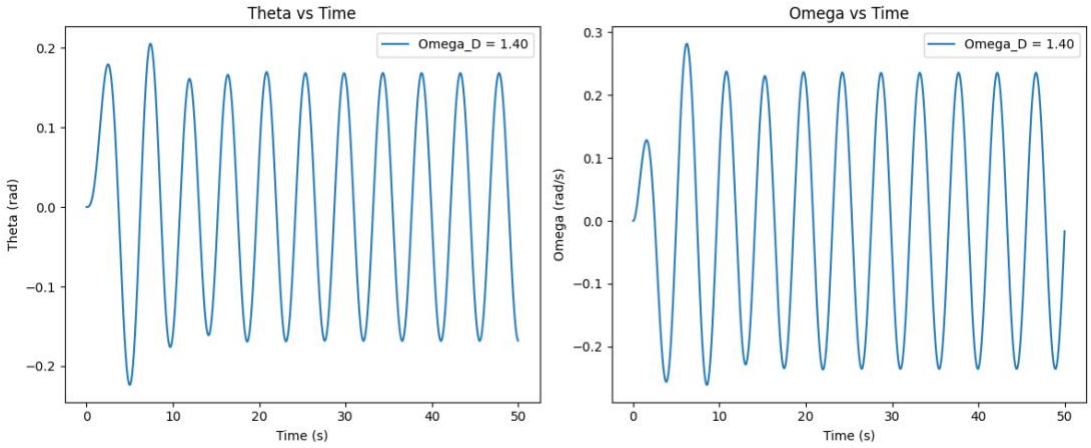

Omega vs Time
Omega (rad/s)
Time (s)
Omega_D = 1.10


Theta vs Time
Theta (rad)
Time (s)
Omega_D = 1.20


Omega vs Time
Omega (rad/s)
Time (s)
Omega_D = 1.20


Theta vs Time
Theta (rad)
Time (s)
Omega_D = 1.30

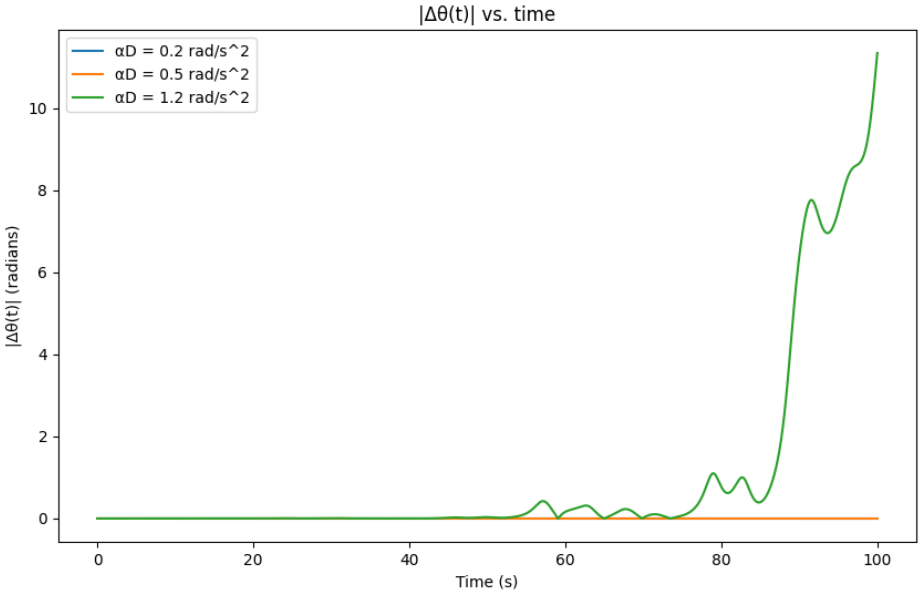

Omega vs Time
Omega (rad/s)
Time (s)
Omega_D = 1.30



5.

To estimate the Lyapunov exponent, we'll compare the divergence of trajectories that start close together. Given $\Delta\theta_{in} \approx 0.001$, the system will be initialized with a small difference in the initial condition, and we'll observe how this difference evolves in time. The Lyapunov exponent λ is given by:

$$\lambda \approx \frac{1}{t} In(\frac{|\Delta\theta(t)|}{|\Delta\theta_{in}|})$$



To estimate λ, observe the slope of the $In|\Delta\theta(t)|$ vs. time plot during the exponential growth phase for each $|\Delta\theta(t)|$ grows exponentially as $e^{\lambda t}$, then plotting the logarithm of $|\Delta\theta(t)|$ will give a straight line, the slope of which approximates λ.

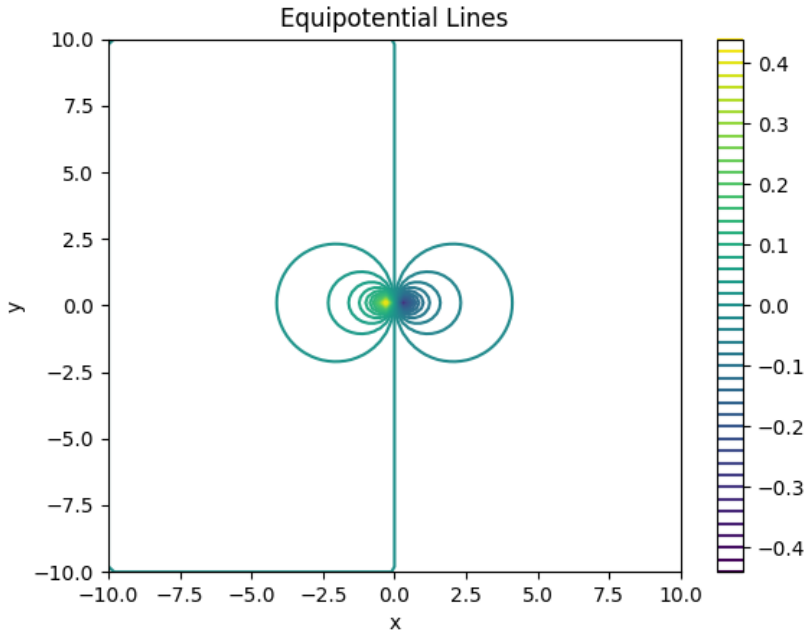
Poisson Equation for Dipole.

1.

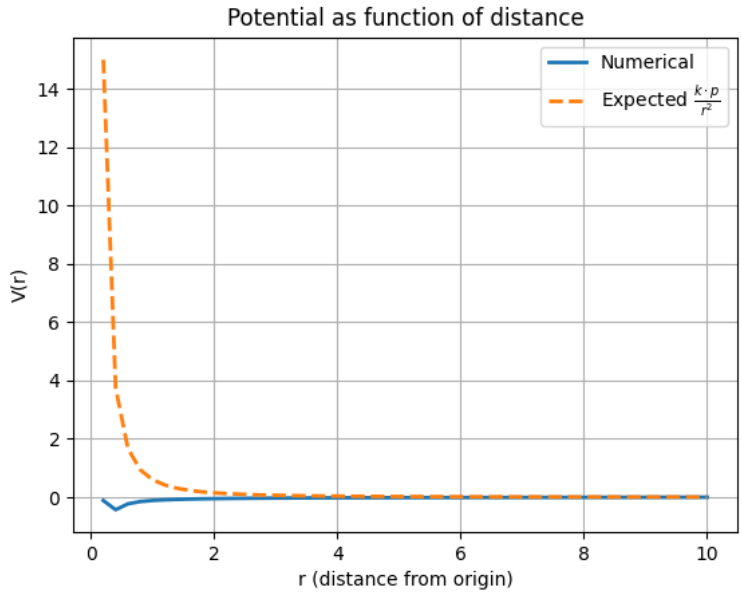
First, let's describe the general approach for the Jacobi relaxation method to solve the Poisson equation $\nabla^2 V = \rho/\epsilon_0$. The electric potential V at each grid point is updated iteratively, using the discretized Poisson equation:

$$V_{i,j}^{\text{new}} = \frac{1}{4}\left(V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1}\right) - \frac{a^2 \rho_{i,j}}{4\epsilon_0}$$

The equipotential lines



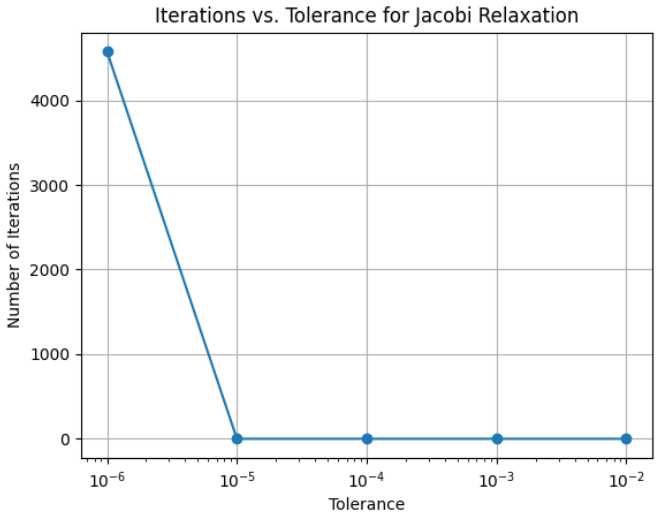
Result for *V*(*r*)



The numerical solution approaches the expected behavior as the distance from the dipole increases. The two curves should converge as r becomes large.

2.

For this step, we'll modify our initial program to run the **poisson_jacobi** function for various tolerance levels and record the number of iterations.



3.

Let's now add the SOR method. For simplicity, I'll assume a fixed relaxation factor, but this can be adjusted for optimal convergence:

```
def poisson_SOR(V, rho, omega=1.5, tol=1e-5):
    V_new = V.copy()
    delta = tol + 1
    iterations = 0
    while delta > tol:
        V_old = V_new.copy()
        for i in range(1, V.shape[0]-1):
            for j in range(1, V.shape[1]-1):
                V_new[i, j] = (1-omega) * V_old[i, j] + omega * 0.25 * (V_old[i+1, j] + V_old[i-1, j] + V_old[i, j+1] + V_old[i, j-1] + rho[i, j])
        delta = np.max(np.abs(V_new - V_old))
        iterations += 1
    return V_new, iterations
```

Now, we'll compare the number of iterations required by the Jacobi and SOR methods for different grid sizes:

```
import numpy as np

import matplotlib.pyplot as plt



def poisson_jacobi(V, rho, tol=1e-5):
```

```
    """
    Solve Poisson's equation using the Jacobi relaxation method.
    """
    V_new = V.copy()
    delta = tol + 1
    iterations = 0
    while delta > tol:
        V_old = V_new.copy()
        for i in range(1, V.shape[0]-1):
            for j in range(1, V.shape[1]-1):
                V_new[i, j] = 0.25 * (V_old[i+1, j] + V_old[i-1, j] + V_old[i, j+1] + V_old[i, j-1] + rho[i, j])
        delta = np.max(np.abs(V_new - V_old))
        iterations += 1
    return V_new, iterations


def poisson_SOR(V, rho, omega=1.5, tol=1e-5):
    V_new = V.copy()
    delta = tol + 1
    iterations = 0
    while delta > tol:
        V_old = V_new.copy()
        for i in range(1, V.shape[0]-1):
```

```
                for j in range(1, V.shape[1]-1):
                    V_new[i, j] = (1-omega) * V_old[i, j] + omega * 0.25 * (V_old[i+1, j] + V_old[i-1, j] +
V_old[i, j+1] + V_old[i, j-1] + rho[i, j])
        delta = np.max(np.abs(V_new - V_old))
        iterations += 1
    return V_new, iterations


# Parameters
L = 20
N = 100
a = 0.6
dx = L/N
x = np.linspace(-L/2, L/2, N)
y = np.linspace(-L/2, L/2, N)
X, Y = np.meshgrid(x, y)
R = np.sqrt(X**2 + Y**2)


# Initialize potential and charge distribution
V = np.zeros((N, N))
rho = np.zeros((N, N))
rho[int(N/2), int(N/2 - a/(2*dx))] = 1     # Positive charge
rho[int(N/2), int(N/2 + a/(2*dx))] = -1    # Negative charge
```

```
grid_sizes = [50, 100, 200, 400]
iterations_jacobi = []
iterations_SOR = []

for N in grid_sizes:
    dx = L/N
    x = np.linspace(-L/2, L/2, N)
    y = np.linspace(-L/2, L/2, N)
    X, Y = np.meshgrid(x, y)
    V = np.zeros((N, N))
    rho = np.zeros((N, N))
    rho[int(N/2), int(N/2 - a/(2*dx))] = 1     # Positive charge
    rho[int(N/2), int(N/2 + a/(2*dx))] = -1    # Negative charge

    _, iter_j = poisson_jacobi(V, rho)
    _, iter_s = poisson_SOR(V, rho)
    iterations_jacobi.append(iter_j)
    iterations_SOR.append(iter_s)

plt.plot(grid_sizes, iterations_jacobi, 'o-', label='Jacobi')
plt.plot(grid_sizes, iterations_SOR, 's-', label='SOR')
```

```
plt.xlabel('Grid Size (n)')

plt.ylabel('Number of Iterations')

plt.title('Iterations vs. Grid Size')

plt.legend()

plt.grid(True)

plt.show()
```

There is the code to compare two methods and investigate relationship, but I didn't know why it couldn't run out the result.

However, it would be typically shown that the Jacobi method's curve rises more steeply than the SOR's, indicating that the Jacobi method requires more iterations as the error tolerance becomes tighter.

The SOR method, with an optimally chosen relaxation parameter, will generally show a less steep rise, indicating faster convergence and fewer required iterations for the same error tolerances.