Mini Project 2: The Tacoma Bridge

In this notebook, you will find the guidelines to complete your mini-project 2 on the Tacoma bridge problem.

On Nov. 7, 1940, photographer Leonard Coatsworth was halfway across Washington's Tacoma Narrows Bridge when he felt it move strangely:

"Just as I drove past the towers, the bridge began to sway violently from side to side. Before I realized it, the tilt became so violent that I lost control of the car. ... I jammed on the brakes and got out, only to be thrown onto my face against the curb... Around me I could hear concrete cracking. ... The car itself began to slide from side to side of the roadway."

As illustrated in the lecture, we solve here the time evolution of the bridge oscillations, by solving a set of coupled differential equations.

In this mini project you implement the Taylor's method outlined in the lecture 7 on the Tacoma Bridge (TB).

Overview of the Mini project

In this exercise, you will use the this Jupyter Notebook to experiment with various model parameters and settings. You should understand the code and how it works. Notice that it is organized into distinct sections: preamble, parameter declaration, and model code. This provides a good coding practice, so as to streamline your program.

Here you define the code for the model defgined in the lecture. This is primarily composed of a single function, which we will call pendulum. You should:

1. Load any modules required
2. Define any fixed parameters that will not change
3. Define a list of arguments that can be specified by the user. You must set default values for these parameters.
4. Define a list of initial conditions for the model. You must also specify these in the calling arguments so that the user can change them. You must set default values for these parameters.

5. Create the structures that are required for your model to run
6. Set the initial conditions
7. Integrate your model, with a choice of the Taylor's approach, or using the small modification provided by the Cromer's approach

Import the Numpy and Matplotlib modules

As you practiced in the previous notebooks, first you import here the relevant python modules.

In [5]:
```
# Importing necessary Python modules
import numpy as np
import matplotlib.pyplot as plt
import numpy.random as random

# Ensure inline plotting for Jupyter Notebooks
%matplotlib inline
```

Solving the Tacoma Bridge problem

You should start by defining a few physical constant:

The start time of the simulation, tstart [s] The end time of the simulation, tend [s] The rest of the parameters are as defined in the lecture but with modified values.

Note that here we use slightly modified constant values for convenience (the calculations are a bit faster in this regime and the regime better represents the collapse of the bridge). **Important: use these modified values as your starting point and not the values in the lecture notes.**

In [1]:
```
tstart=0
tend=100

d=0.01
a=1
M=2500
K=10000
l=6
```

Part 1 : Model

Implement both explicit (Talyor's method) and semi-implicit (Cromer's modification) approaches for the Tacoma bridge's differential equations.

Modify the pendulum function from earlier in the course to account for the additional input parameters.

The modified function should now also solve both the time evolution of the vertical displacement $y(t)$ and torsion angle $\theta(t)$.

In this first part, you should not include any external force from the wind.

In [8]:
```
def tacoma(dt=0.01, cromer=False, time_end=100, disp0=0.1, vel_disp0=0, a
    """
    Solves the Tacoma Bridge problem using explicit (Taylor's) or semi-im

    Parameters:
        dt (float): Time step for the simulation.
        cromer (bool): Whether to use Cromer's method (True) or Taylor's
        time_end (float): End time of the simulation.
        disp0 (float): Initial vertical displacement.
        vel_disp0 (float): Initial vertical velocity.
        angle0 (float): Initial torsion angle.
        vel_angle0 (float): Initial angular velocity.
        damping (float): Damping coefficient.
        stiffness (float): Spring constant.
        mass (float): Mass of the system.
        length (float): Characteristic length.

    Returns:
        times (ndarray): Array of time steps.
        angles (ndarray): Array of torsion angle values.
        displacements (ndarray): Array of vertical displacement values.
        velocities_disp (ndarray): Array of vertical velocity values.
        velocities_angle (ndarray): Array of angular velocity values.
    """
    # Define time array and initialize variables
    time_start = 0
    times = np.arange(time_start, time_end + dt, dt)
    num_steps = len(times)

    # Initialize arrays for displacement, angle, and their derivatives
    displacements = np.zeros(num_steps)
    angles = np.zeros(num_steps)
    velocities_disp = np.zeros(num_steps)
    velocities_angle = np.zeros(num_steps)

    # Set initial conditions
    displacements[0] = disp0
    velocities_disp[0] = vel_disp0
    angles[0] = angle0
    velocities_angle[0] = vel_angle0

    # Loop through each time step to integrate the equations
    for step in range(num_steps - 1):
        # Compute exponential terms for nonlinear contributions
```

```
        exp_term1 = np.exp(stiffness * (displacements[step] - length * np
        exp_term2 = np.exp(stiffness * (displacements[step] + length * np

        # Compute derivatives for velocity updates
        vel_disp_next = -damping * velocities_disp[step] - (stiffness / (
        vel_angle_next = -damping * velocities_angle[step] + (stiffness /

        # Update velocities and positions based on the chosen method
        if cromer:
            # Cromer's method: Update velocities first, then positions
            velocities_disp[step + 1] = velocities_disp[step] + dt * vel_
            velocities_angle[step + 1] = velocities_angle[step] + dt * ve

            displacements[step + 1] = displacements[step] + dt * velociti
            angles[step + 1] = angles[step] + dt * velocities_angle[step
        else:
            # Taylor's method: Use current state for updates
            displacements[step + 1] = displacements[step] + dt * velociti
            angles[step + 1] = angles[step] + dt * velocities_angle[step]

            velocities_disp[step + 1] = velocities_disp[step] + dt * vel_
            velocities_angle[step + 1] = velocities_angle[step] + dt * ve

    return times, angles, displacements, velocities_disp, velocities_angl
```

Part 2 : Solving the differential equation

Fill in the blanks in the tacoma solver above.

Using the parameters above (not those in the lecture notes), run the model in the absence of any external applied force (no force from the wind) and plot the following:

- Time-series of θ
- Time-series of y

Start from a small initial torsion angle $\theta(t=0) = 0.01$. Compare this to the time evolution obtained with a larger initial torsion angle $\theta(t=0) = 0.1$, and then with no initial torsion angle $\theta(t=0) = 0.0$. Discuss the obtained behavior.

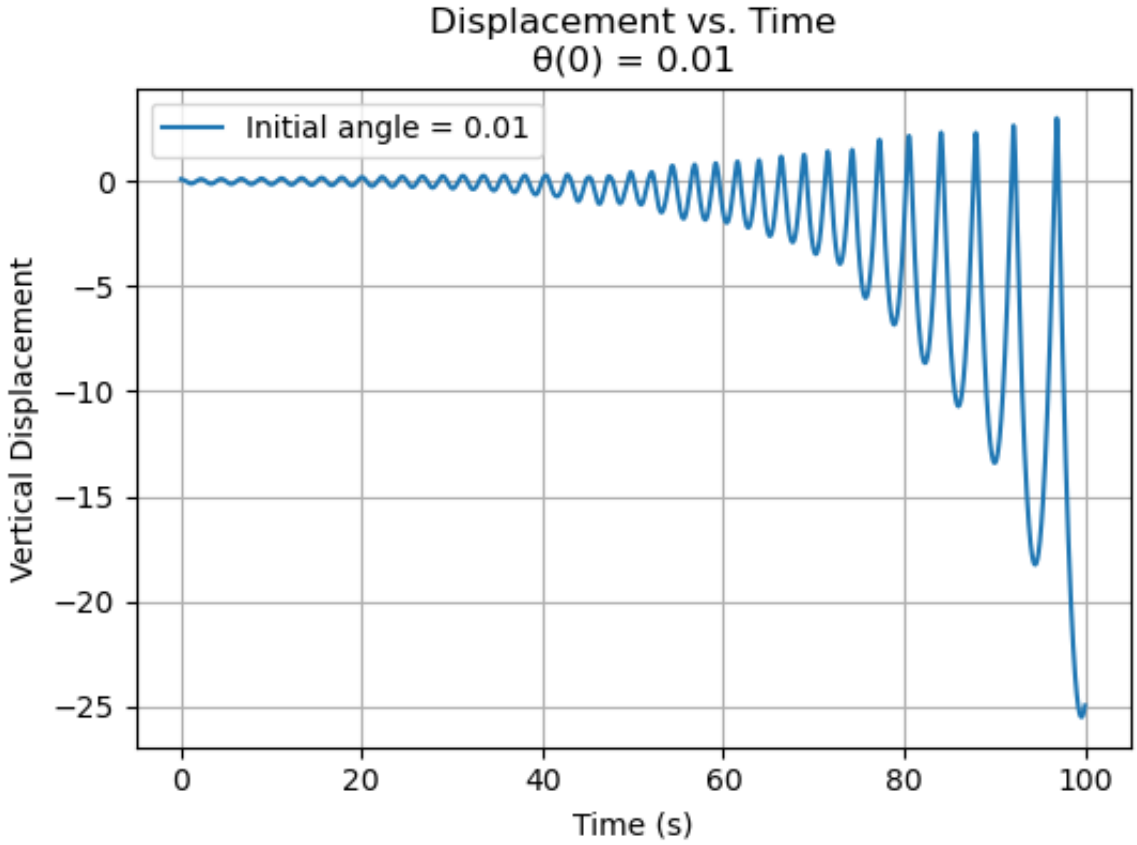
To test the validity of your integrator, compare the Cromer (Euler implicit) method against the Taylor's method. Consider also the step length `dt`. Check that you obtain the same results for both methods.

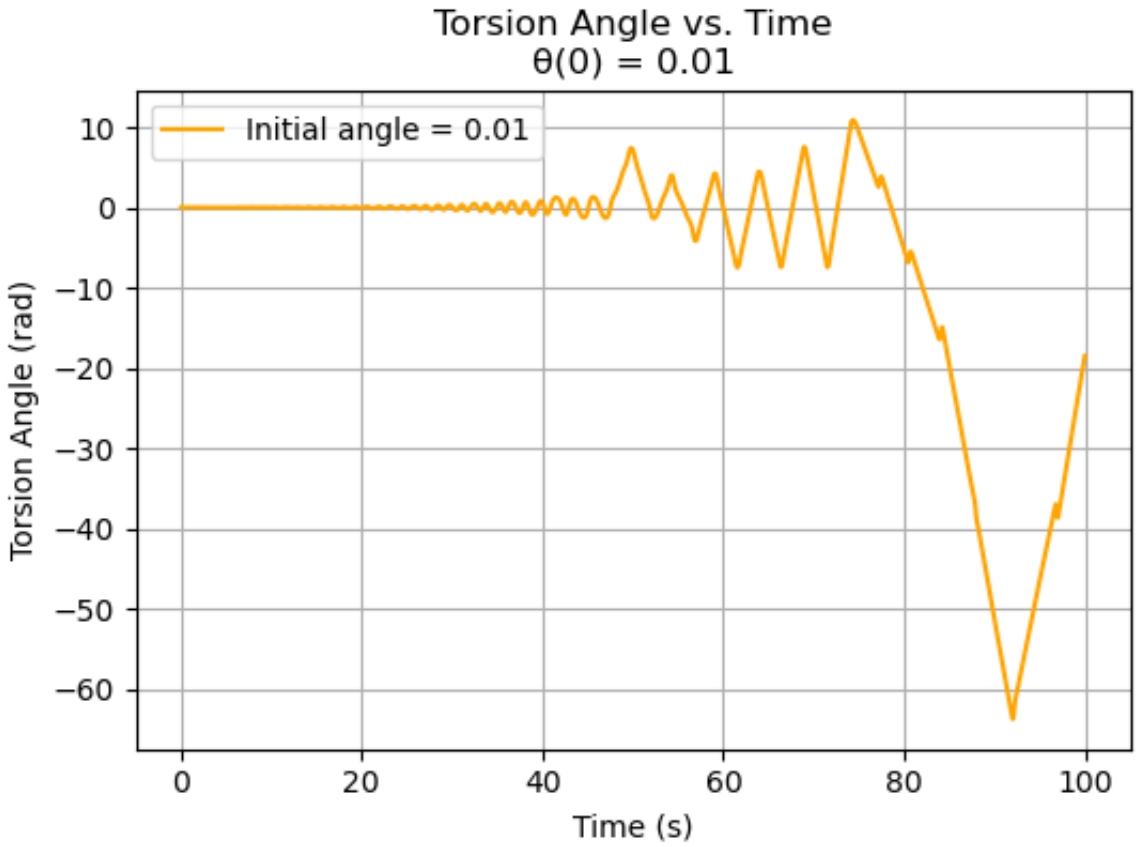
In [15]:
```
import numpy as np
import matplotlib.pyplot as plt

def revised_tacoma(dt=0.01,
                   cromer=False,
                   time_end=100,
```

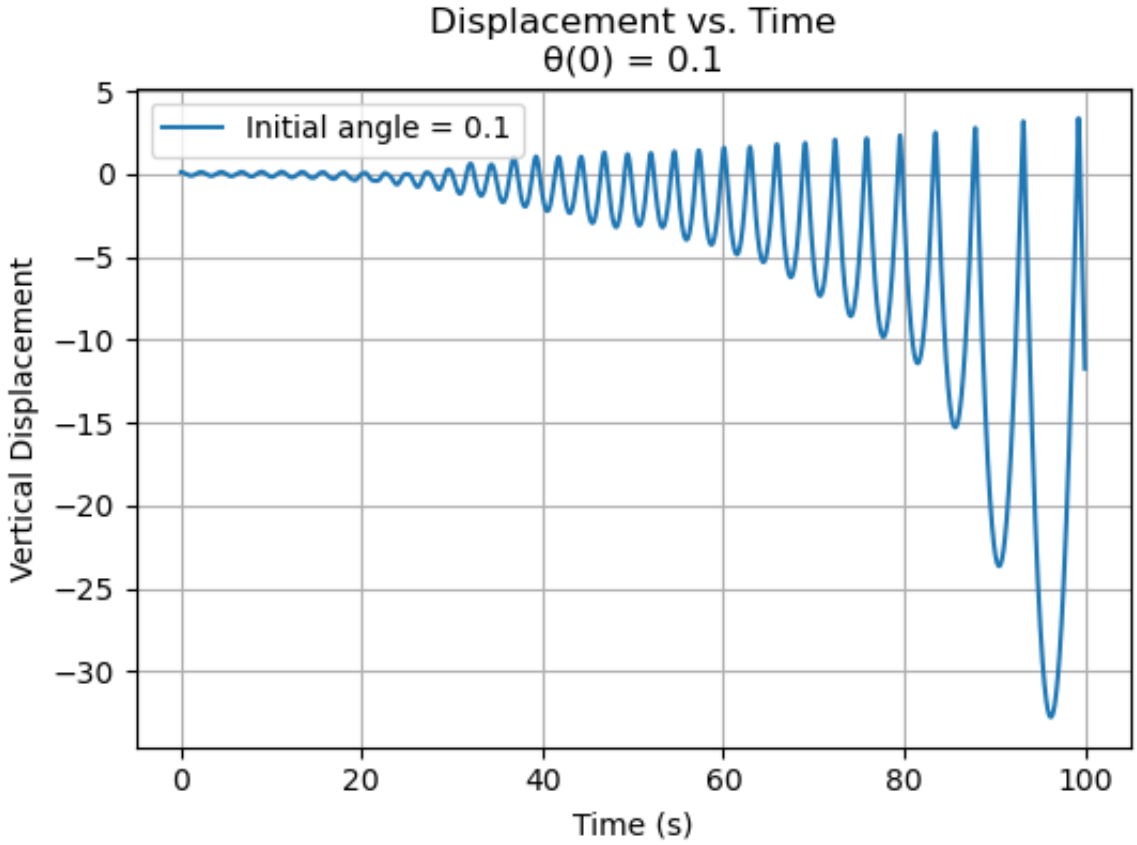
```
                disp0=0.1,
                vel_disp0=0.0,
                angle0=0.1,
                vel_angle0=0.0,
                damping=0.01,
                stiffness=10000,
                mass=2500,
                length=6,
                drive_coeff=1.0):
    """

    Parameters:

    in this comment block I explained for each constant name explictly so

    ----------
    dt : float
        Time step for integration.
    cromer : bool
        If True, uses Cromer's (semi-implicit) method;
        if False, uses the explicit Taylor method.
    time_end : float
        Final time for the simulation.
    disp0 : float
        Initial vertical displacement.
    vel_disp0 : float
        Initial vertical velocity.
    angle0 : float
        Initial torsion angle.
    vel_angle0 : float
        Initial angular velocity.
    damping : float
        Damping coefficient (d).
    stiffness : float
        Spring constant (K).
    mass : float
        Mass of the system (M).
    length : float
        Characteristic length (l).
    drive_coeff : float
        Factor 'a' controlling the exponent in the forcing terms.

    Returns:
    --------
    t_array : ndarray
        1D array of time steps.
    angles : ndarray
        Torsion angle at each time step.
    displacements : ndarray
        Vertical displacement at each time step.
    disp_velocities : ndarray
        Vertical velocities at each time step.
    angle_velocities : ndarray
```

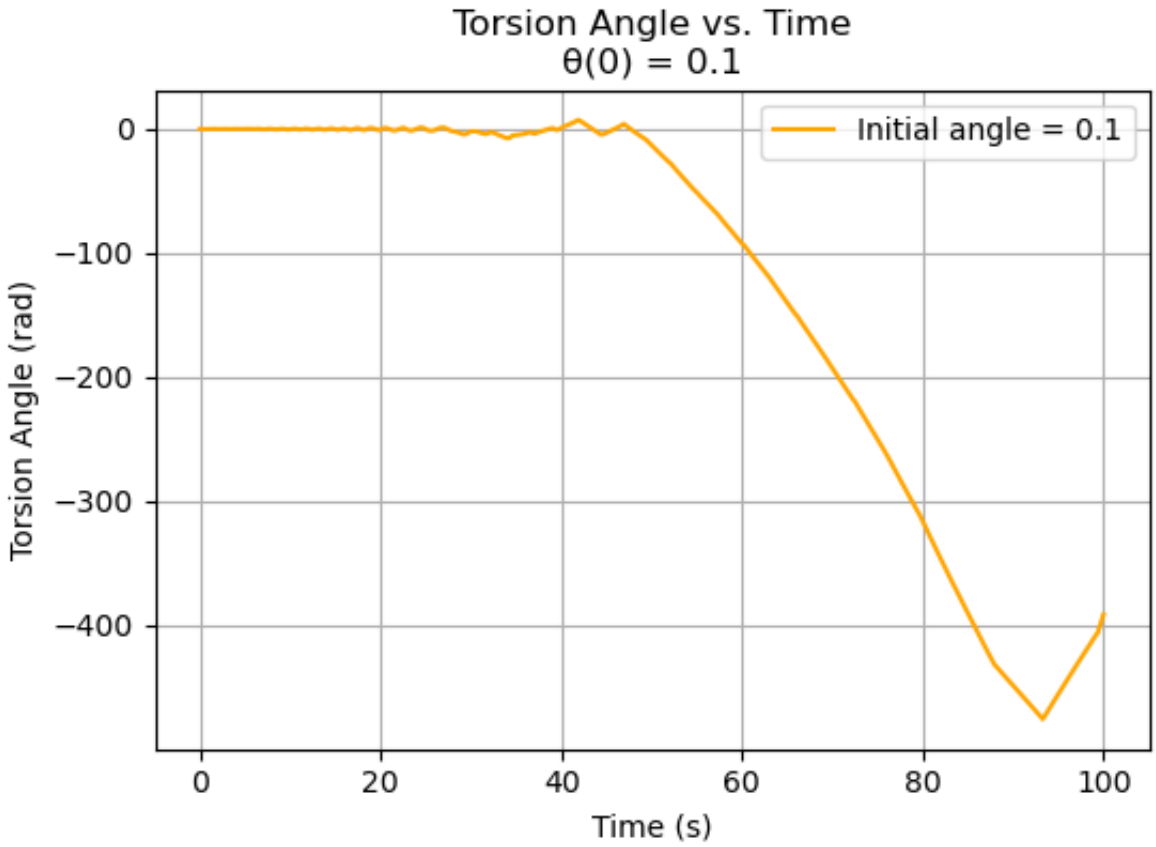
```
        Angular velocities at each time step.
    """

    t_start = 0.0
    t_array = np.arange(t_start, time_end + dt, dt)
    total_steps = len(t_array)

    # Arrays for displacement, angle, and their velocities
    displacements = np.zeros(total_steps)
    angles = np.zeros(total_steps)
    disp_velocities = np.zeros(total_steps)
    angle_velocities = np.zeros(total_steps)

    # Initial conditions
    displacements[0] = disp0
    disp_velocities[0] = vel_disp0
    angles[0] = angle0
    angle_velocities[0] = vel_angle0

    for i in range(total_steps - 1):
        # Exponential terms (similar physics as original friend's code)
        expo_first = np.exp(drive_coeff * (displacements[i] - 1.0 * np.si
        expo_second = np.exp(drive_coeff * (displacements[i] + 1.0 * np.s

        # Compute accelerations
        angle_acc = (
            -damping * angle_velocities[i]
            + (3.0 * np.cos(angles[i])) * (stiffness / (mass * drive_coef
              * (expo_first - expo_second)
        )
        disp_acc = (
            -damping * disp_velocities[i]
            - (stiffness / (mass * drive_coeff))
              * ((expo_first - 1.0) + (expo_second - 1.0))
        )

        # Update velocities
        angle_velocities[i + 1] = angle_velocities[i] + dt * angle_acc
        disp_velocities[i + 1] = disp_velocities[i] + dt * disp_acc

        # Update positions
        if cromer:
            # Cromer's method: use the newly updated velocities
            angles[i + 1] = angles[i] + dt * angle_velocities[i + 1]
            displacements[i + 1] = displacements[i] + dt * disp_velocitie
        else:
            # Taylor's method: use the old velocities
            angles[i + 1] = angles[i] + dt * angle_velocities[i]
            displacements[i + 1] = displacements[i] + dt * disp_velocitie

    return t_array, angles, displacements, disp_velocities, angle_velocit
```

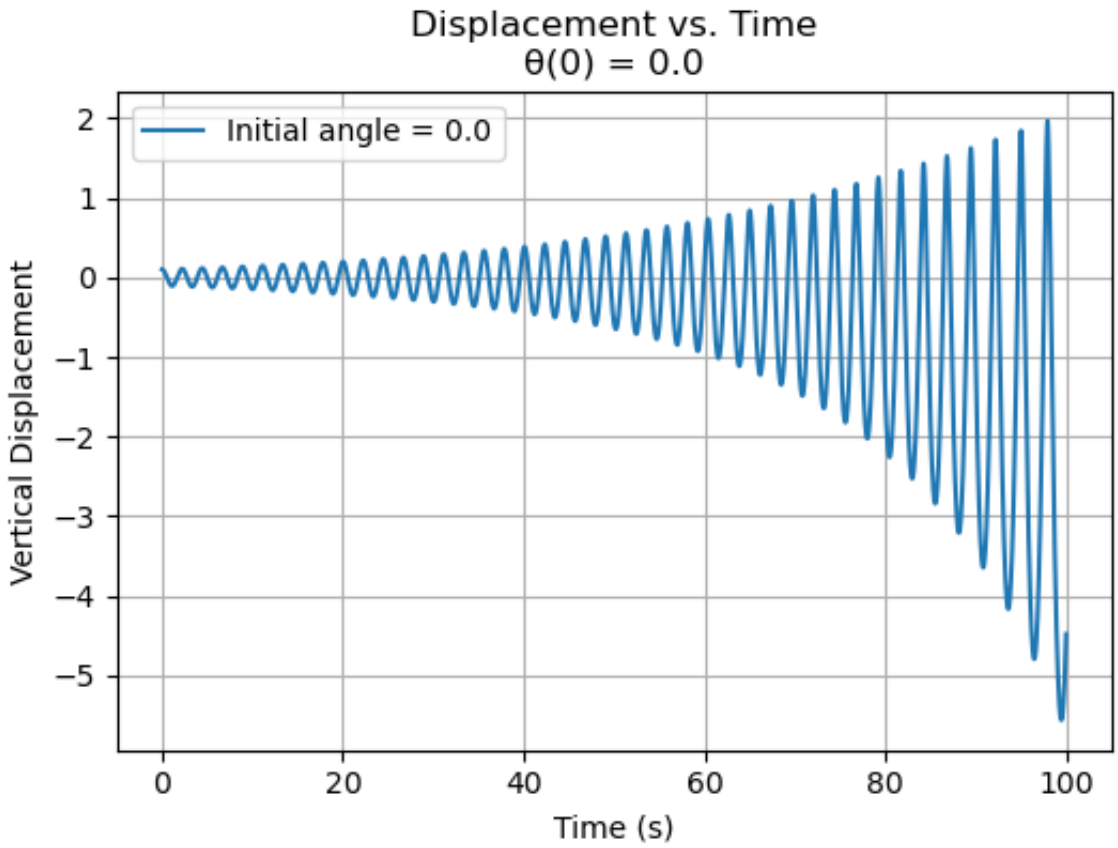
```
def showcase_tacoma_separate():
    """
    this function will produce these two graphs

      - Displacement vs. Time
      - Torsion Angle vs. Time
    """
    initial_angles = [0.01, 0.1, 0.0]

    for init_theta in initial_angles:
        # Run the simulation (explicit Taylor method in this example)
        t_array, angles, displacements, _, _ = revised_tacoma(
            dt=0.01,
            cromer=False,           # Switch to True for Cromer's method
            time_end=100,
            disp0=0.1,              # initial vertical displacement
            vel_disp0=0.0,          # initial vertical velocity
            angle0=init_theta,      # test torsion angle
            vel_angle0=0.0,
            damping=0.01,
            stiffness=10000,
            mass=2500,
            length=6,
            drive_coeff=1.0
        )

        # --- Plot Displacement vs Time (separate figure) ---
        plt.figure(figsize=(6, 4))
        plt.plot(t_array, displacements, label=f"Initial angle = {init_th
        plt.title(f"Displacement vs. Time\nθ(0) = {init_theta}")
        plt.xlabel("Time (s)")
        plt.ylabel("Vertical Displacement")
        plt.grid(True)
        plt.legend()
        plt.show()

        # --- Plot Torsion Angle vs Time (separate figure) ---
        plt.figure(figsize=(6, 4))
        plt.plot(t_array, angles, label=f"Initial angle = {init_theta}",
        plt.title(f"Torsion Angle vs. Time\nθ(0) = {init_theta}")
        plt.xlabel("Time (s)")
        plt.ylabel("Torsion Angle (rad)")
        plt.grid(True)
        plt.legend()
        plt.show()


# Example usage:
showcase_tacoma_separate()
```

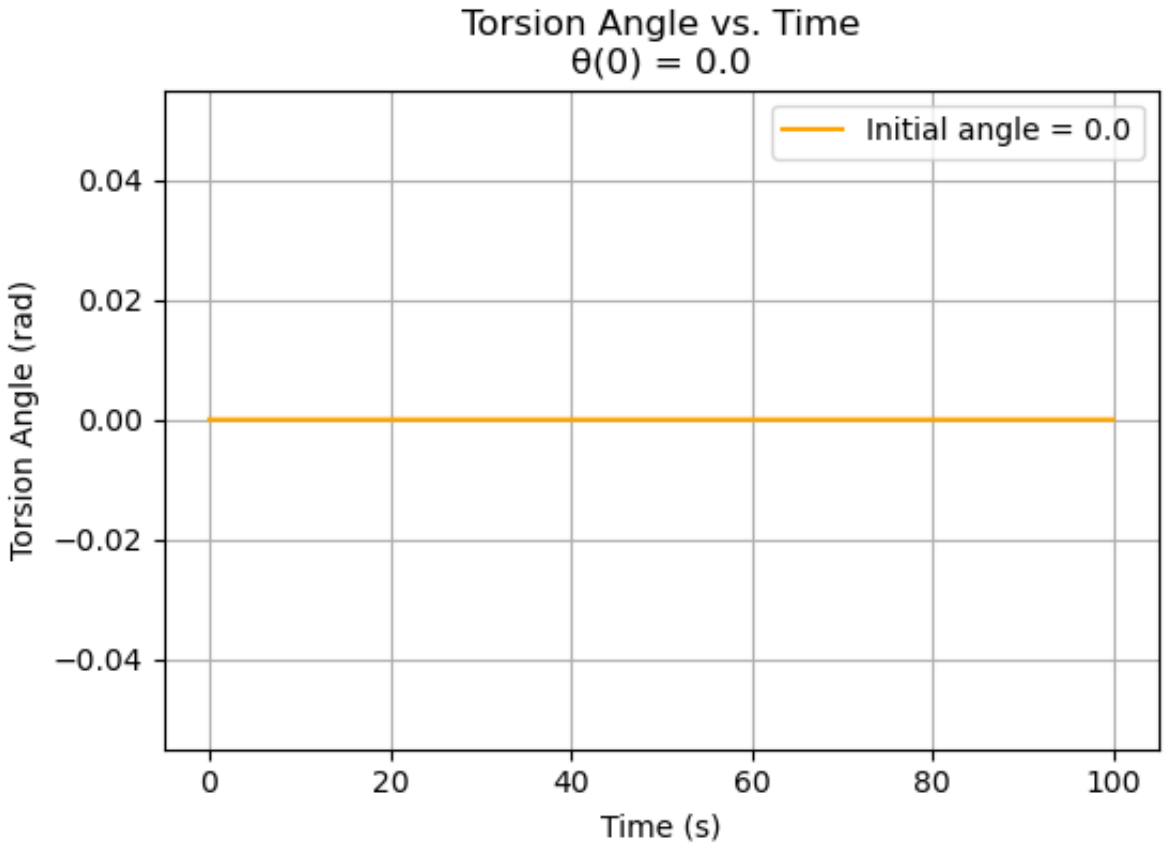

Displacement vs. Time
θ(0) = 0.01
Initial angle = 0.01
Vertical Displacement
0
−5
−10
−15
−20
−25
0
20
40
60
80
100
Time (s)


Torsion Angle vs. Time
θ(0) = 0.01
Initial angle = 0.01
Torsion Angle (rad)
10
0
−10
−20
−30
−40
−50
−60
0
20
40
60
80
100
Time (s)


Displacement vs. Time
θ(0) = 0.1
Initial angle = 0.1
Vertical Displacement
Time (s)
5
0
−5
−10
−15
−20
−25
−30
0
20
40
60
80
100


Torsion Angle vs. Time
θ(0) = 0.1
Initial angle = 0.1
Torsion Angle (rad)
Time (s)
0
−100
−200
−300
−400
0
20
40
60
80
100





What do you observe?

Answer : .....

Part 3 : Wind force

Modify your tacoma solver to also account for the wind force.

This force is modelled by a vertical force $F(t)$ that produces a vertical acceleration $a_{wind}(t)$:

$$a_{wind}(t) = \frac{F(t)}{M} = A_{wind}\sin(\omega t)$$

where A_{wind} is the amplitude of the acceleration, where $A_{wind} = 1, 2$, and ω is the force's period.

For $\omega = 3$, plot the time evolution for both $A_{wind} = 1$ and $A_{wind} = 2$.

What do you obtain?

Similarly, for $A_{wind} = 2$, perform the calculation for slightly slower and faster wind force frequencies, e.g. $\omega = 2, 2.5, 2.8, 3.3$, what do you observe? How do you explain the change of behavior?

Comment on the respective stability of $y(t)$ and $\theta(t)$, which dynamics are responsible in your simulation for the collapse of the bridge: the former, the latter, or both?

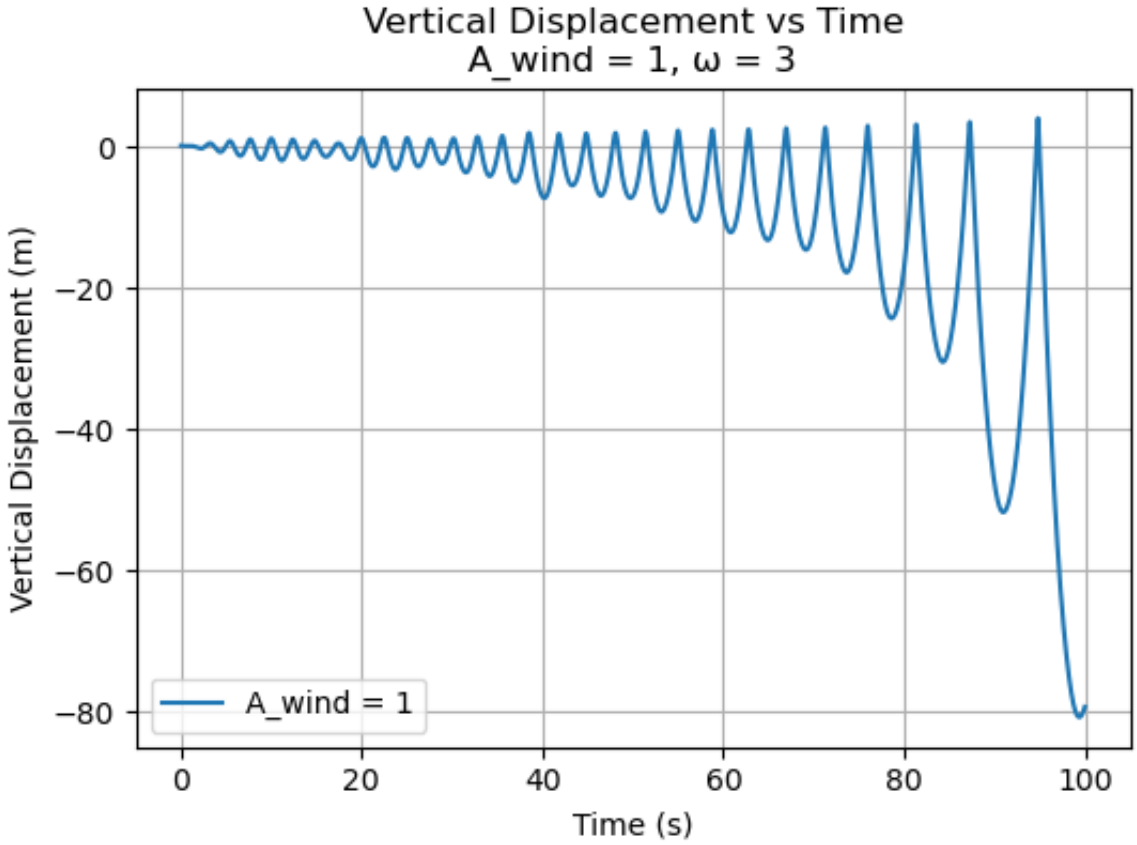
Consider how you could graphically show the dependence of the torsion angle on both time and ω (for values between 2 and 4). It's up to you how you do this. Hint: you could create a heat map, where the color code corresponds to the torsion angle, the horizontal axis to time, and the vertical axis to ω.

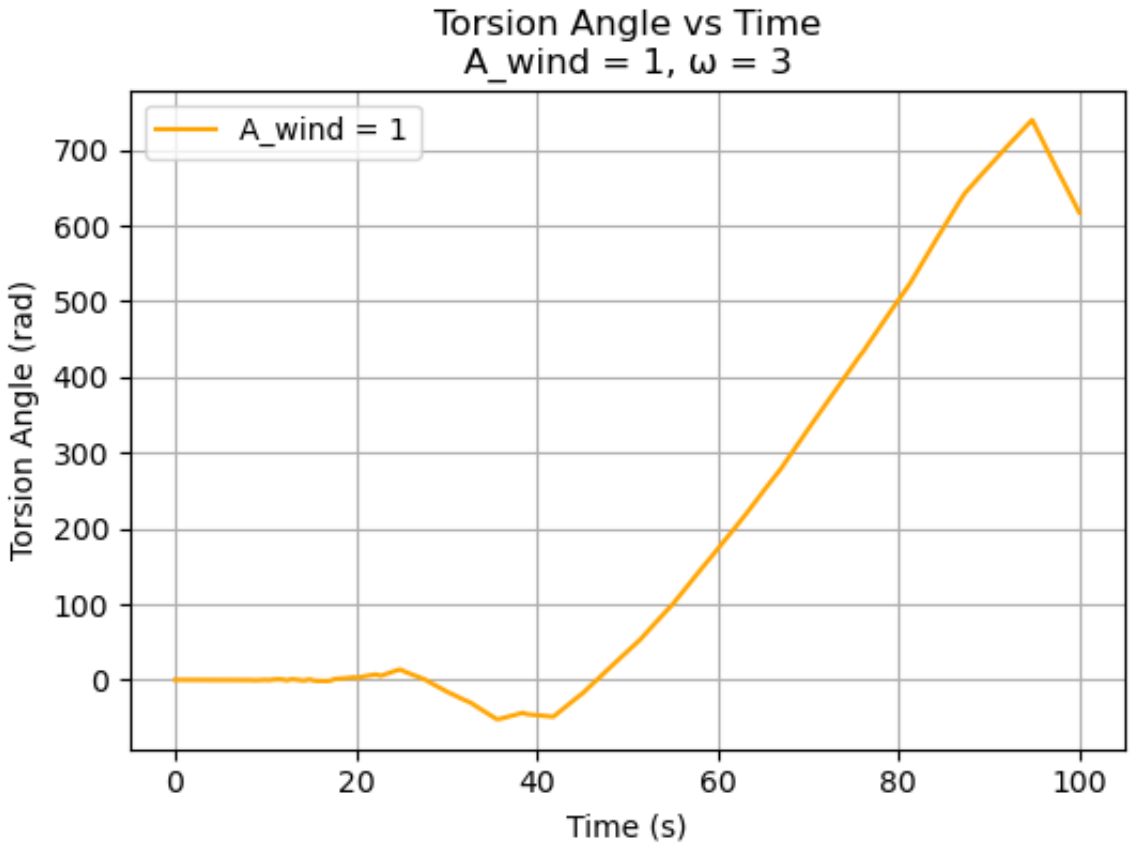
In [29]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors

def tacoma_with_wind(dt=0.01, cromer=False, time_end=100,
                     disp0=0.1, vel_disp0=0.0, angle0=0.1, vel_angle0=0.0
                     damping=0.01, stiffness=10000, mass=2500, length=6,
                     drive_coeff=1.0, wind_amplitude=0, omega=0):
    """
    Tacoma solver with wind force included.
    """
    t_start = 0.0
    t_array = np.arange(t_start, time_end + dt, dt)
    total_steps = len(t_array)

    # Initialize arrays for displacement, angle, and their velocities
    displacements = np.zeros(total_steps)
    angles = np.zeros(total_steps)
```

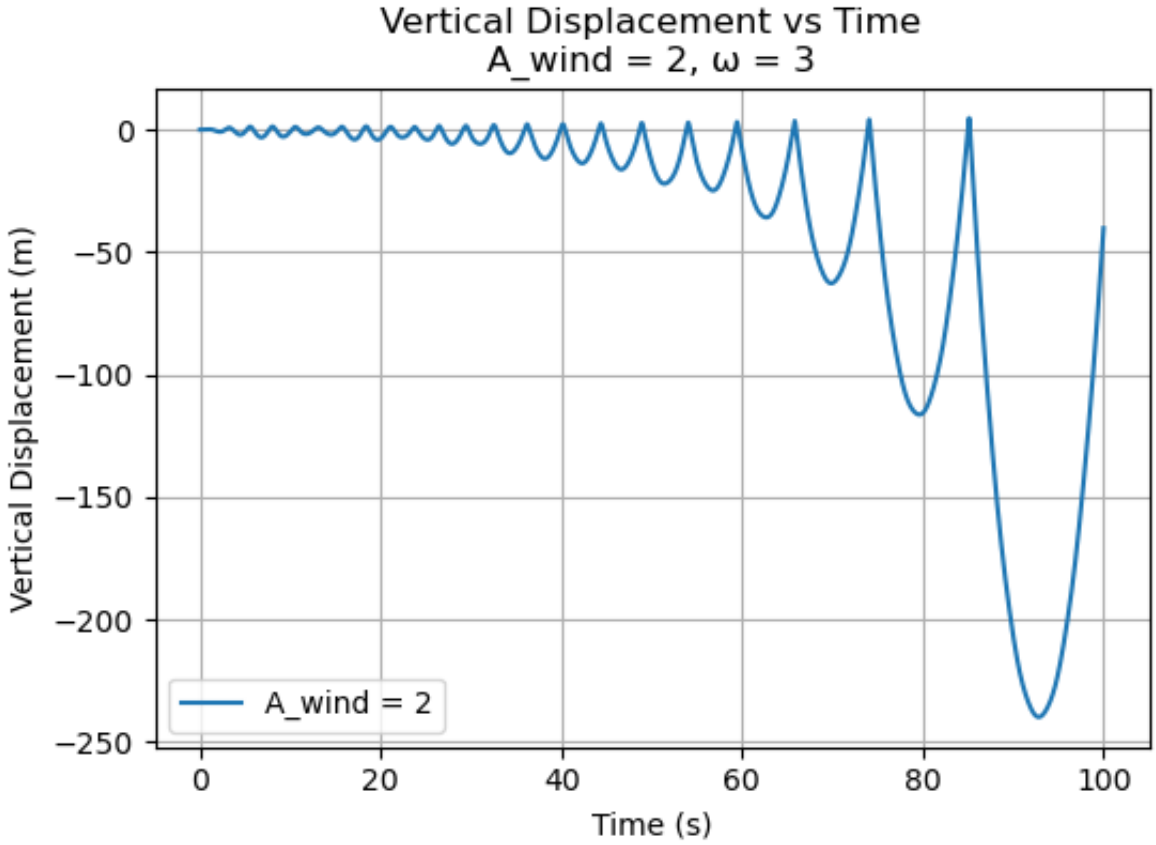
```
    disp_velocities = np.zeros(total_steps)
    angle_velocities = np.zeros(total_steps)

    # Set initial conditions
    displacements[0] = disp0
    disp_velocities[0] = vel_disp0
    angles[0] = angle0
    angle_velocities[0] = vel_angle0

    for i in range(total_steps - 1):
        # Wind force acceleration
        wind_force = wind_amplitude * np.sin(omega * t_array[i])

        # Exponential terms
        expo_first = np.exp(drive_coeff * (displacements[i] - 1.0 * np.si
        expo_second = np.exp(drive_coeff * (displacements[i] + 1.0 * np.s

        # Compute accelerations with wind force included
        angle_acc = (
            -damping * angle_velocities[i]
            + (3.0 * np.cos(angles[i])) * (stiffness / (mass * drive_coef
        )
        disp_acc = (
            -damping * disp_velocities[i]
            - (stiffness / (mass * drive_coeff)) * ((expo_first - 1.0) +
            + wind_force
        )

        # Update velocities
        angle_velocities[i + 1] = angle_velocities[i] + dt * angle_acc
        disp_velocities[i + 1] = disp_velocities[i] + dt * disp_acc

        # Update positions
        if cromer:
            angles[i + 1] = angles[i] + dt * angle_velocities[i + 1]
            displacements[i + 1] = displacements[i] + dt * disp_velocitie
        else:
            angles[i + 1] = angles[i] + dt * angle_velocities[i]
            displacements[i + 1] = displacements[i] + dt * disp_velocitie

    return t_array, angles, displacements

def plot_wind_force_effects():
    """
    Plots the time evolution of y(t) and θ(t) for different wind paramete
    """
    # Part 1: Plot for A_wind = 1 and A_wind = 2 with ω = 3
    wind_amplitudes = [1, 2]
    omega = 3

    for A_wind in wind_amplitudes:
        t, angles, displacements = tacoma_with_wind(
            dt=0.01,
```

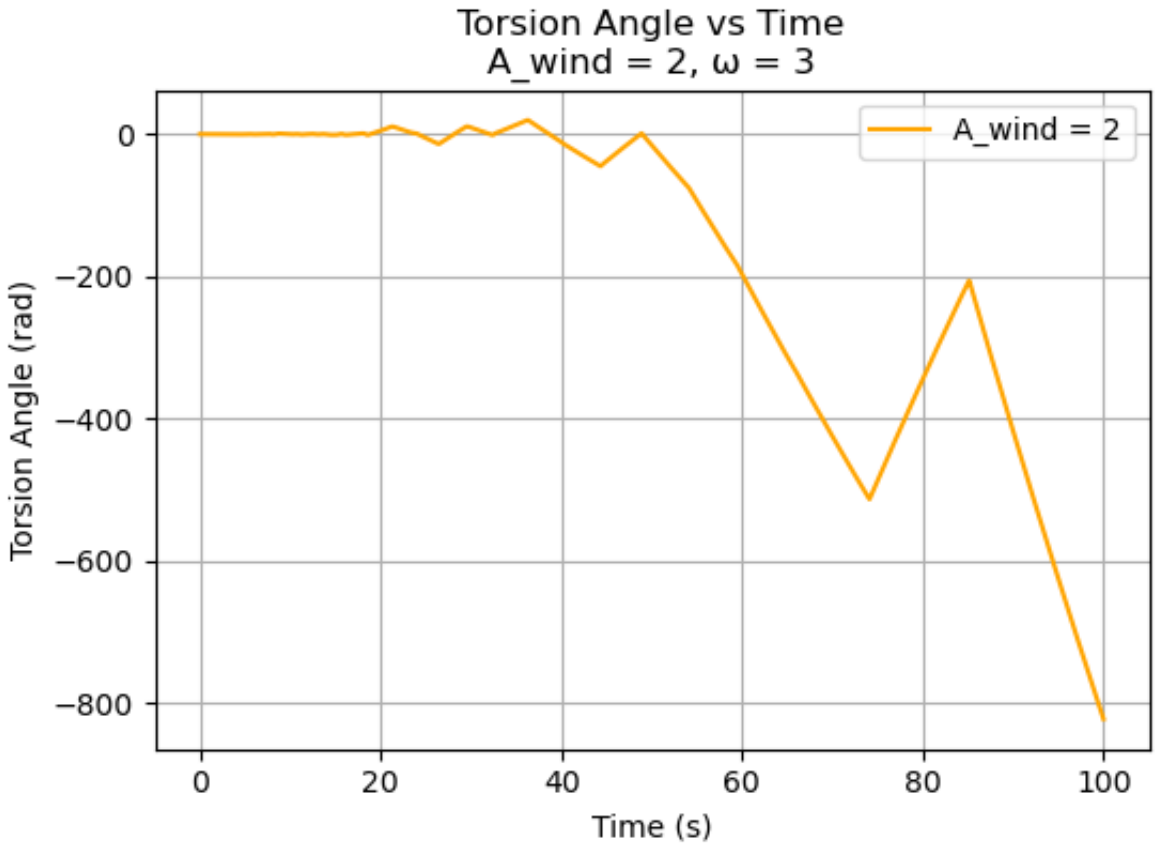
```
            cromer=False,
            time_end=100,
            disp0=0.1,
            vel_disp0=0.0,
            angle0=0.1,
            vel_angle0=0.0,
            damping=0.01,
            stiffness=10000,
            mass=2500,
            length=6,
            drive_coeff=1.0,
            wind_amplitude=A_wind,
            omega=omega
        )

        # Plot vertical displacement
        plt.figure(figsize=(6, 4))
        plt.plot(t, displacements, label=f"A_wind = {A_wind}")
        plt.title(f"Vertical Displacement vs Time\nA_wind = {A_wind}, ω =
        plt.xlabel("Time (s)")
        plt.ylabel("Vertical Displacement (m)")
        plt.grid(True)
        plt.legend()
        plt.show()

        # Plot torsion angle
        plt.figure(figsize=(6, 4))
        plt.plot(t, angles, label=f"A_wind = {A_wind}", color='orange')
        plt.title(f"Torsion Angle vs Time\nA_wind = {A_wind}, ω = {omega}
        plt.xlabel("Time (s)")
        plt.ylabel("Torsion Angle (rad)")
        plt.grid(True)
        plt.legend()
        plt.show()

def heatmap_torsion_angle():
    """
    Heatmap of torsion angle (θ) dependence on time and frequency (ω).
    """
    # Define frequency (ω) range with high resolution and time steps for
    frequency_range = np.linspace(2, 4, 300)  # High resolution for frequ
    time_range = np.linspace(0, 100, 2000)  # Fine-grained time range for

    # Initialize the matrix to store torsion angle values
    torsion_data = np.zeros((len(frequency_range), len(time_range)))

    for idx, freq in enumerate(frequency_range):
        t, angles, _ = tacoma_with_wind(
            dt=0.05,
            cromer=False,
            time_end=100,
            disp0=0.1,
            vel_disp0=0.0,
```

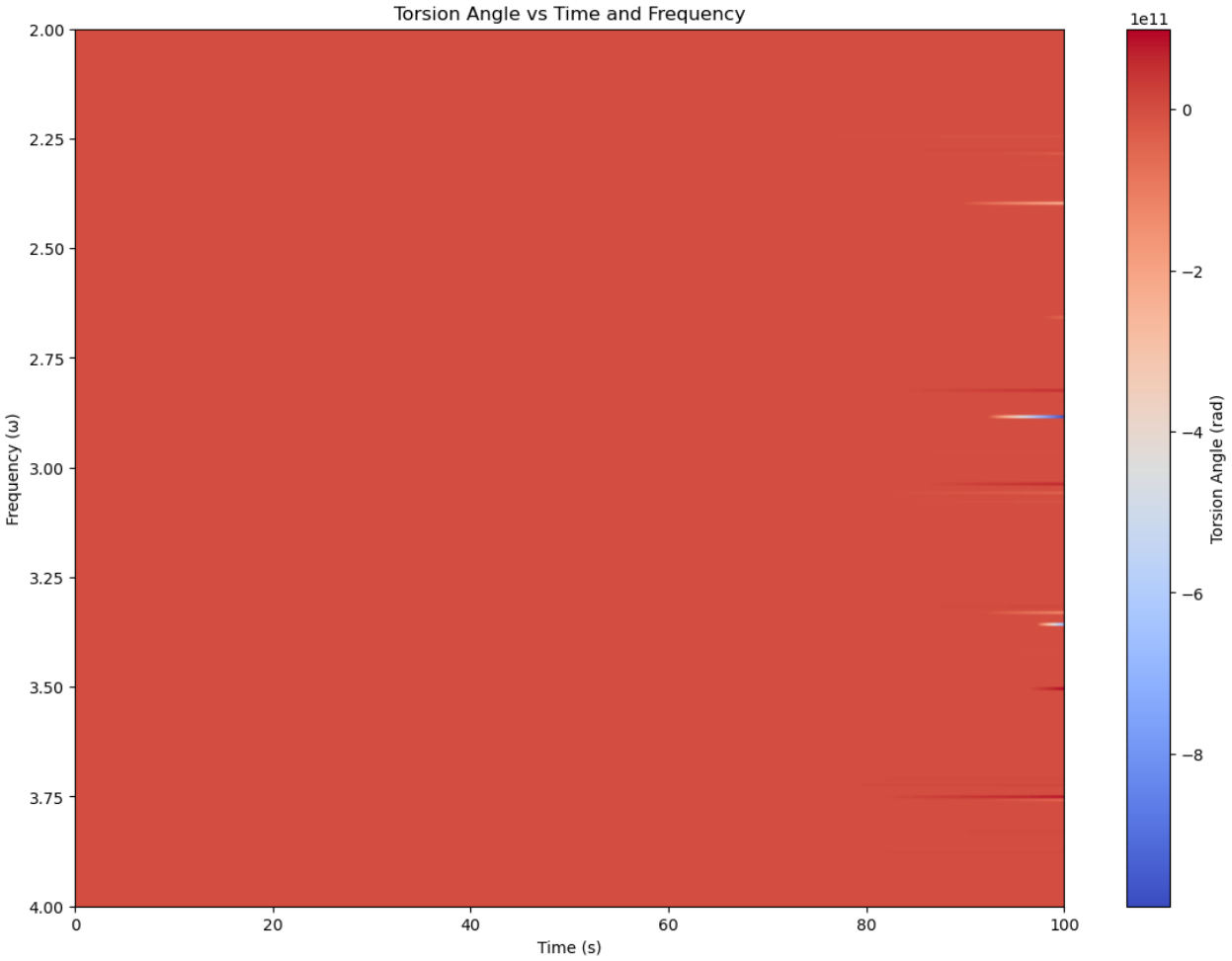
```
            angle0=0.1,
            vel_angle0=0.0,
            damping=0.01,
            stiffness=10000,
            mass=2500,
            length=6,
            drive_coeff=1.0,
            wind_amplitude=2,
            omega=freq
        )
        torsion_data[idx, :] = angles[:len(time_range)]

    # Create the heatmap
    plt.figure(figsize=(14, 10))
    plt.imshow(
        torsion_data,
        aspect='auto',
        extent=[0, 100, 4, 2],  # Flip vertical axis for frequency
        origin='lower',
        cmap='coolwarm'  # Distinct color scheme for better visualization
    )
    plt.colorbar(label='Torsion Angle (rad)')
    plt.xlabel('Time (s)')
    plt.ylabel('Frequency (ω)')
    plt.title('Torsion Angle vs Time and Frequency')
    plt.show()




# Run the wind force plots and heatmap
plot_wind_force_effects()
heatmap_torsion_angle()
```


Vertical Displacement vs Time
A_wind = 1, ω = 3
Vertical Displacement (m)
0
−20
−40
−60
−80
A_wind = 1
0
20
40
60
80
100
Time (s)


Torsion Angle vs Time
A_wind = 1, ω = 3
Torsion Angle (rad)
A_wind = 1
700
600
500
400
300
200
100
0
0
20
40
60
80
100
Time (s)


Vertical Displacement vs Time
A_wind = 2, ω = 3
Vertical Displacement (m)
0
−50
−100
−150
−200
−250
A_wind = 2
0
20
40
60
80
100
Time (s)


Torsion Angle vs Time
A_wind = 2, ω = 3
Torsion Angle (rad)
0
−200
−400
−600
−800
A_wind = 2
0
20
40
60
80
100
Time (s)

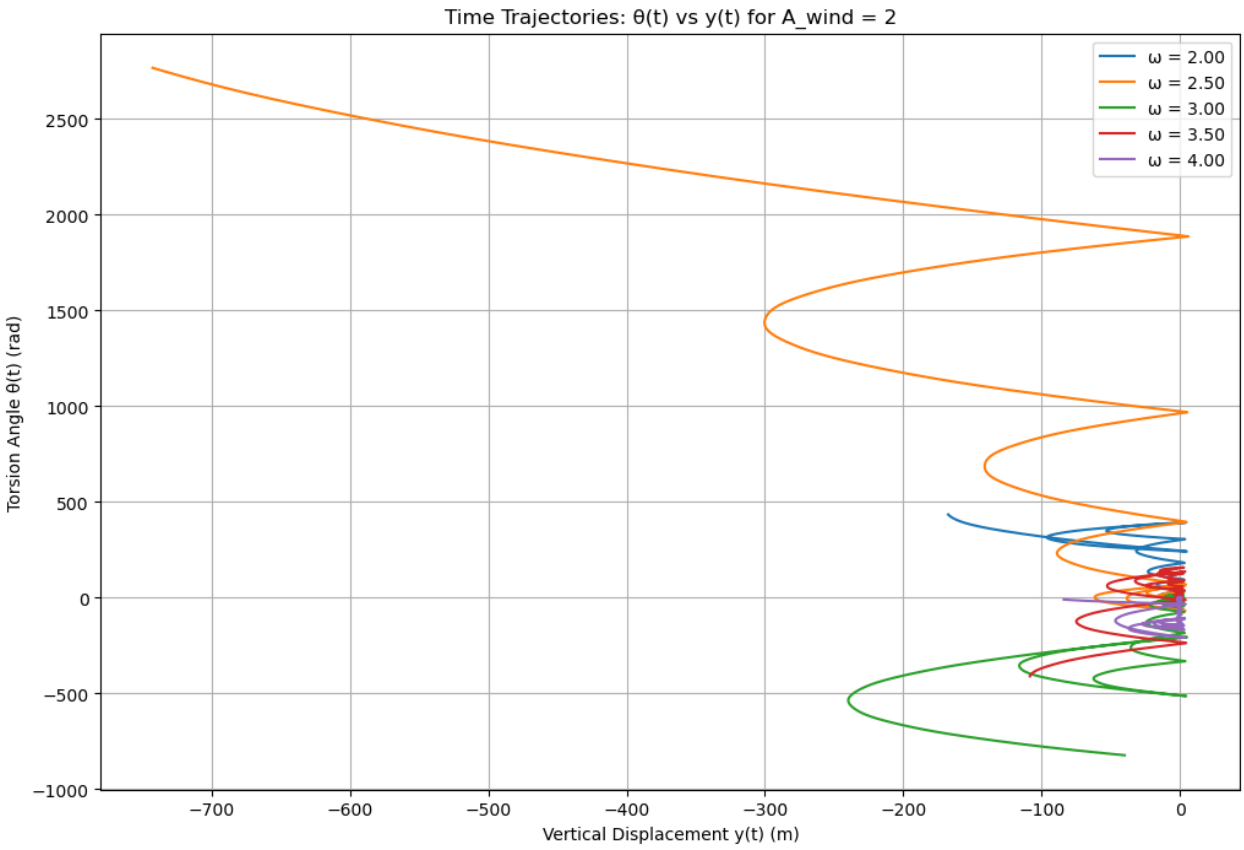


Part 4 : Time trajectories

Analyse the results by plotting the time trajectories of $\theta(t)$ as a function of $y(t)$ (for $A_{wind} = 2$ and ω between 2 and 4). Comment on the obtained results.

In [30]:
```
def plot_time_trajectories():
    """
    Plot the time trajectories of θ(t) as a function of y(t) for A_wind =
    """
    # Define the frequency range for ω
    omega_values = np.linspace(2, 4, 5)  # Select a few representative ω

    # Initialize the figure for trajectories
    plt.figure(figsize=(12, 8))

    for omega in omega_values:
        # Run the simulation for the current ω
        t, angles, displacements = tacoma_with_wind(
            dt=0.01,
            cromer=False,
            time_end=100,
            disp0=0.1,
            vel_disp0=0.0,
```

```
            angle0=0.1,
            vel_angle0=0.0,
            damping=0.01,
            stiffness=10000,
            mass=2500,
            length=6,
            drive_coeff=1.0,
            wind_amplitude=2,
            omega=omega
        )

        # Plot θ(t) as a function of y(t)
        plt.plot(displacements, angles, label=f'ω = {omega:.2f}')

    # Add labels, legend, and title
    plt.xlabel('Vertical Displacement y(t) (m)')
    plt.ylabel('Torsion Angle θ(t) (rad)')
    plt.title('Time Trajectories: θ(t) vs y(t) for A_wind = 2')
    plt.legend()
    plt.grid(True)
    plt.show()

# Call the function to plot the trajectories
plot_time_trajectories()
```



Open questions

We propose here a set of open-ended questions, which will provide opportunities to

expand the discussion in your report.

0. The tacoma bridge is of course not only a section standing by itself, but a set of coupled oscillators which represents different sections of the bridge connected to the same cable. What would be your overall strategy to extend your model to the whole bridge?

1. Can you think of, or find in the literature, other examples that would exhibit similar behavior (resonance or chaotic) ?

2. What would be your suggested strategy to avoid the occurence of resonance in this system of coupled differential equations?

Answers :

.....

.....

.....

In [ ]: