Nicolino Primavera
FEM for Fluid Flow and FSI Interactions
Assignment 3
11/27/24

Objective:

- Solving the damped harmonic oscillator equation using the generalized-α time integration method.
- A damped harmonic oscillator is governed by (m * (d^2u / dt^2)) + c (du / dt) + ku = 0
  - 2nd order ODE
  - m is the object's mass
  - c is the damping coefficient
  - k is the spring stiffness
  - u is the vector of displacement unknowns
  - γ (gamma) = c / cr is the damping ratio
  - cr = 2sqrt(mk) is the critical damping
  - the system behaves as an undamped (γ = 0), underdamped (0 < γ < 1), overdamped (γ > 1), or critically damped (γ = 1) system
  - ω_n = sqrt(k / m) denotes the natural frequency of the system
- Equation can be rewritten as (d^2u / dt^2) + 2 * γ * ω_n * (du / dt) + (ω_n^2) * u = 0
  - initial conditions u(0) = 1 , du/dt (0) = 1 , ω_n = pi
- Solve EQN 2 for all the damping regimes (γ = 0, 0.5, 1, 2) using the generalized-α time integration method
  - For each γ, choose at least four values of the spectral radius parameters (0 ≤ ρ_∞ ≤ 1) and compare your numerical solution against the analytical solution
  - For each ρ_∞, refine the time step size, and comment on the error convergence and the numerical solution behavior

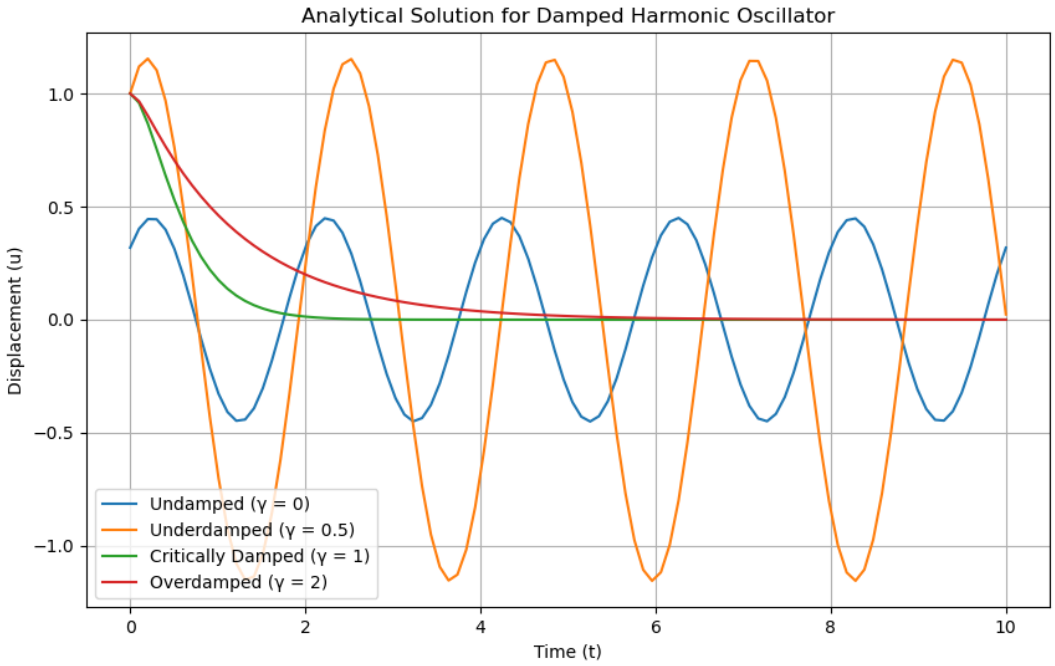
Methodology:

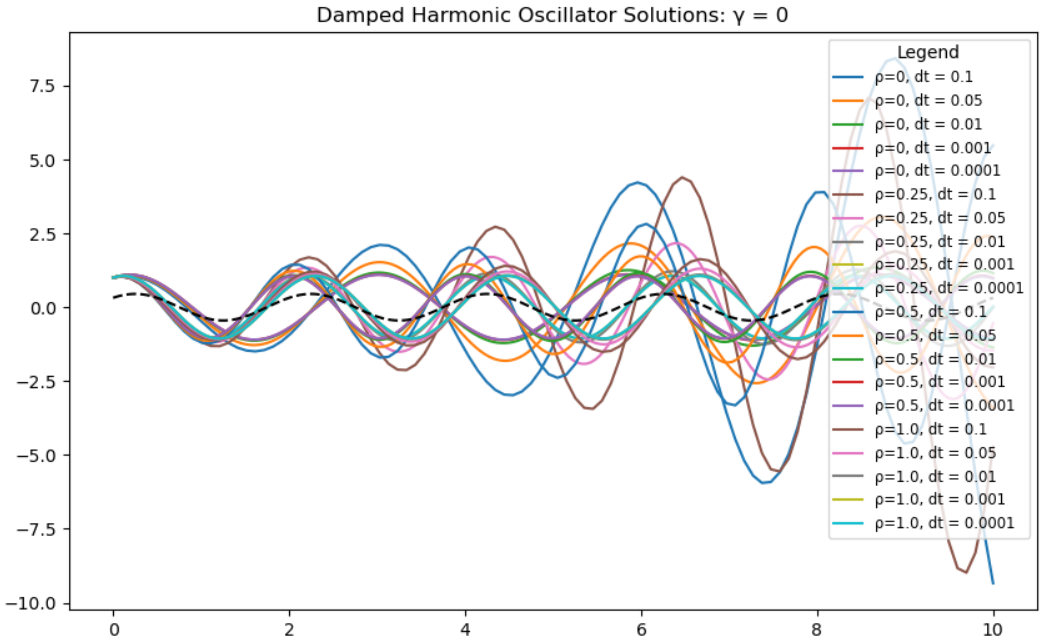
- Using α-time integration to solve a 2nd order ODE (damped harmonic oscillator equation)
- Algorithmic damping in the high frequency regime is desired
- Initialize parameters:
  - Initial displacement
  - Initial velocity
  - Natural frequency
  - Object mass
  - Spring stiffness
  - Critical damping
  - Gammas
  - Damping coefficient
  - Spectral radius parameters
  - Time steps
  - Time end
- Solve the analytical solution for the damped harmonic oscillator
  - For gamma = 0, undamped oscillation u(t) = e^-γ * sin(ω_nt)

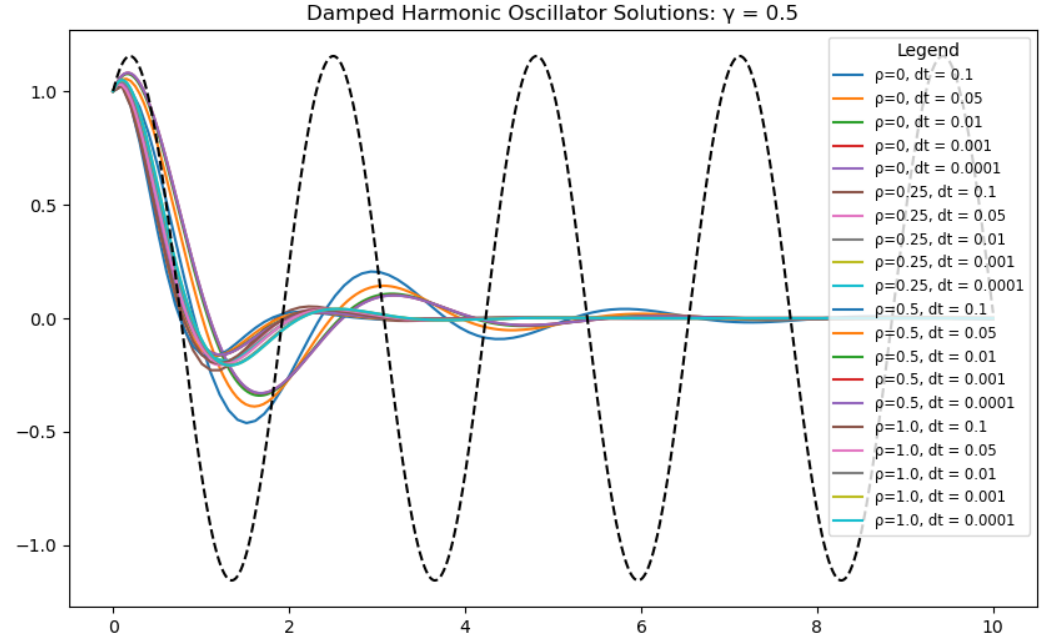
    - For gamma < 1, underdamped oscillation (complex roots)
    - For gamma = 1, critically damped (double real root)
    - For gamma > 1, overdamped (two distinct real roots)
- Apply the Generalized-α Time Integration Method for second-order ODEs
    - proposed for a 2nd order ODE: M(d^2x/dt^2)+C(dx/dt)+DX=F
    - used for solving structural dynamics problems
    - possesses numerical dissipation that can be controlled by the user
    - achieves high-frequency dissipation while minimizing unwanted low-frequency dissipation
    - controls high frequency oscillations
    - Main advantages:
        - implicit (unconditional stability)
        - 2nd order accurate ~ 0(Δt^2)
        - controls (damps) high frequency oscillations very effectively
    - Parameters:
        - Spectral radius parameter, ρ_∞, controls the high-frequency damping
        - For each γ, choose at least four values of the spectral radius parameters (0 ≤ ρ_∞ ≤ 1) and compare your numerical solution against the analytical solution
            - ex: ρ_∞ = 0,0.5,0.8,1.0
        - Formula for Beta(β) and Gamma(γ) based on ρ_∞ for numerical stability and accuracy:
            - β = (1 + ρ_∞)^2 / 4
            - γ = 0.5 + ρ_∞
            - stability parameters
- Solve and plot the numerical solution against the analytical solution
- Steps:
    - Compute total number of time steps based on time increment (dt) and simulation duration (t_end)
    - Initialize arrays to store displacement (u), velocity (v), and acceleration (a)
    - Apply initial conditions for displacement (u), velocity (v), and acceleration (a)
    - Define stability parameters based on spectral radius
        - Alpha_m # Mass
        - Alpha_f # Force
    - Define parameters that guarantee 2nd order accuracy
        - Beta # int param for displacement, high frequency dissipation is maximized
        - Gamma # int param for velocity
    - Compute the time integration loop
        - Displacement and velocity predictions
        - Solve for displacement, velocity, acceleration
- Plot each gamma with respective rho and time step values
- Error convergence:
    - Calculate the error E = ||u_numerical - u_analytical|| using the L2-norm for each time step size and ρ_∞
    - Plot E vs. time step size for each ρ_∞

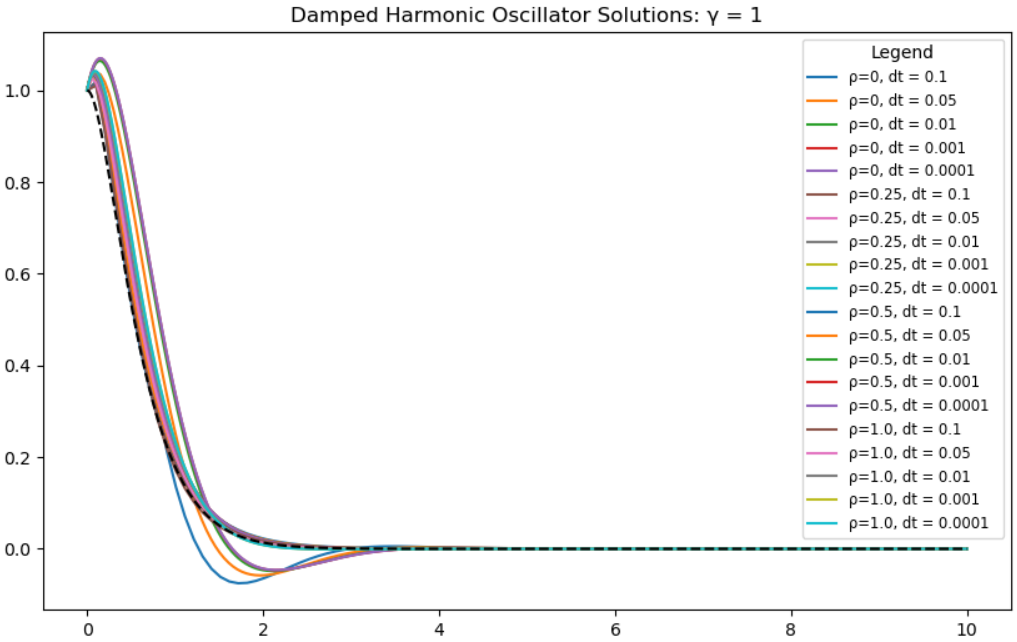
Results and Discussion:

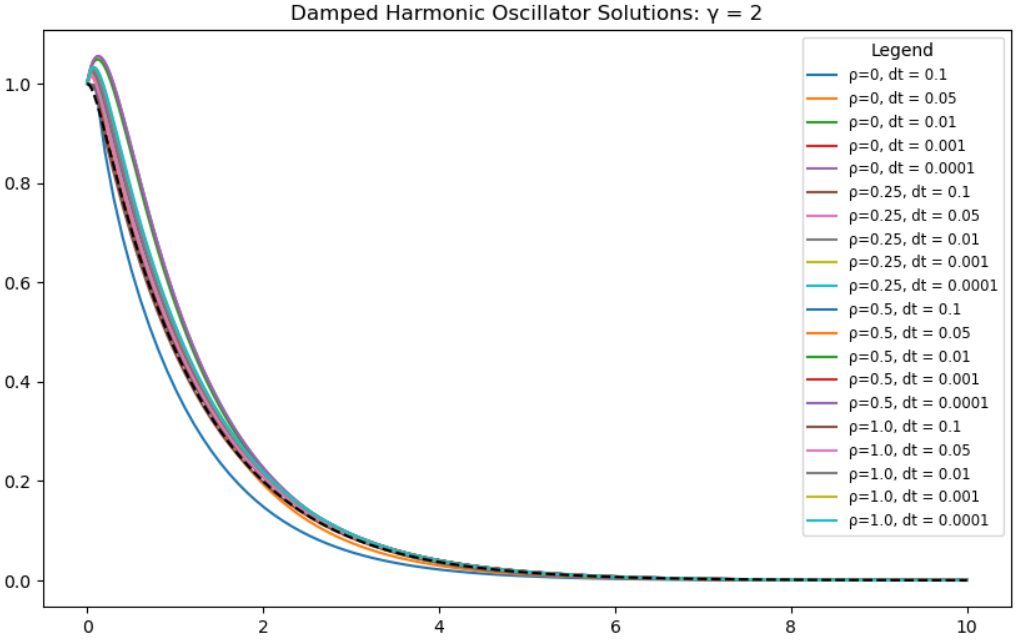
- My results for gamma=1 and gamma=2 were very accurate.
- My results for gamma=0 are slightly inaccurate. The shape of the curve was similar for the more refined time steps, but the amplitude of the wave is slightly bigger than that of the analytical solution.
- My results for gamma=1/2 are inaccurate. The curve starts out correct for about half a wavelength, but then it bottoms out instead of continuing the sine wave like curve.
    - Need to refine my code for gamma=1/2
- My error (L-2 norm) was around 2.4.
    - This needs refinement and improvement.

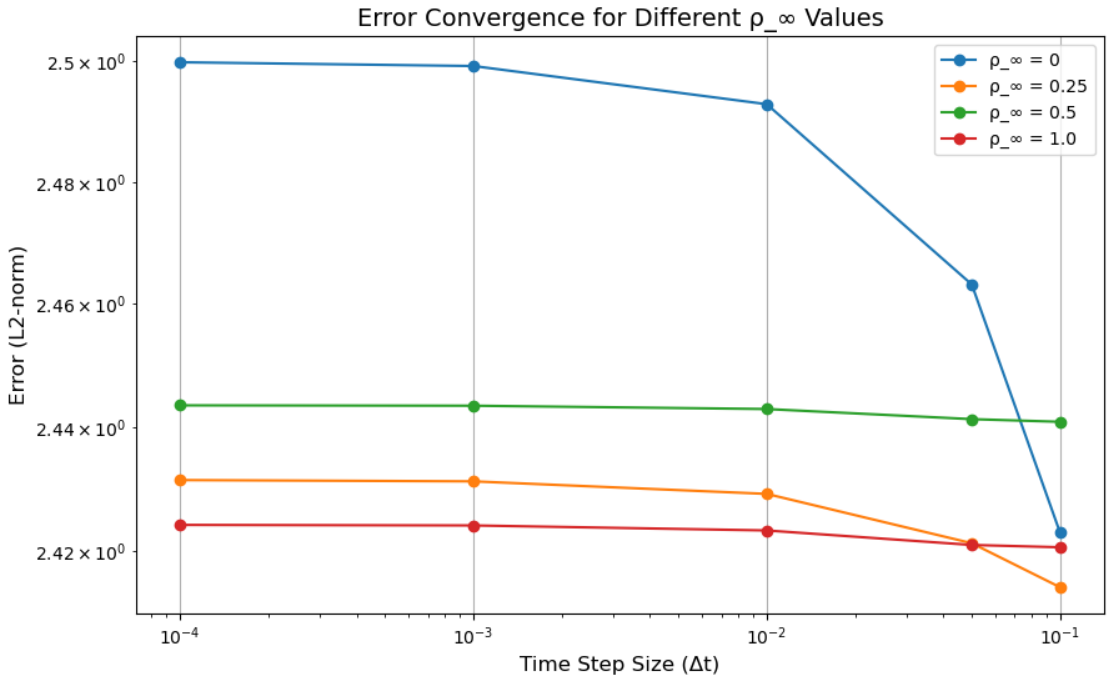



Damped Harmonic Oscillator Solutions: γ = 0
Legend
ρ=0, dt = 0.1
ρ=0, dt = 0.05
ρ=0, dt = 0.01
ρ=0, dt = 0.001
ρ=0, dt = 0.0001
ρ=0.25, dt = 0.1
ρ=0.25, dt = 0.05
ρ=0.25, dt = 0.01
ρ=0.25, dt = 0.001
ρ=0.25, dt = 0.0001
ρ=0.5, dt = 0.1
ρ=0.5, dt = 0.05
ρ=0.5, dt = 0.01
ρ=0.5, dt = 0.001
ρ=0.5, dt = 0.0001
ρ=1.0, dt = 0.1
ρ=1.0, dt = 0.05
ρ=1.0, dt = 0.01
ρ=1.0, dt = 0.001
ρ=1.0, dt = 0.0001
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
0
2
4
6
8
10


Damped Harmonic Oscillator Solutions: γ = 0.5
Legend
ρ=0, dt = 0.1
ρ=0, dt = 0.05
ρ=0, dt = 0.01
ρ=0, dt = 0.001
ρ=0, dt = 0.0001
ρ=0.25, dt = 0.1
ρ=0.25, dt = 0.05
ρ=0.25, dt = 0.01
ρ=0.25, dt = 0.001
ρ=0.25, dt = 0.0001
ρ=0.5, dt = 0.1
ρ=0.5, dt = 0.05
ρ=0.5, dt = 0.01
ρ=0.5, dt = 0.001
ρ=0.5, dt = 0.0001
ρ=1.0, dt = 0.1
ρ=1.0, dt = 0.05
ρ=1.0, dt = 0.01
ρ=1.0, dt = 0.001
ρ=1.0, dt = 0.0001
1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10


Damped Harmonic Oscillator Solutions: γ = 1
Legend
ρ=0, dt = 0.1
ρ=0, dt = 0.05
ρ=0, dt = 0.01
ρ=0, dt = 0.001
ρ=0, dt = 0.0001
ρ=0.25, dt = 0.1
ρ=0.25, dt = 0.05
ρ=0.25, dt = 0.01
ρ=0.25, dt = 0.001
ρ=0.25, dt = 0.0001
ρ=0.5, dt = 0.1
ρ=0.5, dt = 0.05
ρ=0.5, dt = 0.01
ρ=0.5, dt = 0.001
ρ=0.5, dt = 0.0001
ρ=1.0, dt = 0.1
ρ=1.0, dt = 0.05
ρ=1.0, dt = 0.01
ρ=1.0, dt = 0.001
ρ=1.0, dt = 0.0001
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10


Damped Harmonic Oscillator Solutions: γ = 2
Legend
ρ=0, dt = 0.1
ρ=0, dt = 0.05
ρ=0, dt = 0.01
ρ=0, dt = 0.001
ρ=0, dt = 0.0001
ρ=0.25, dt = 0.1
ρ=0.25, dt = 0.05
ρ=0.25, dt = 0.01
ρ=0.25, dt = 0.001
ρ=0.25, dt = 0.0001
ρ=0.5, dt = 0.1
ρ=0.5, dt = 0.05
ρ=0.5, dt = 0.01
ρ=0.5, dt = 0.001
ρ=0.5, dt = 0.0001
ρ=1.0, dt = 0.1
ρ=1.0, dt = 0.05
ρ=1.0, dt = 0.01
ρ=1.0, dt = 0.001
ρ=1.0, dt = 0.0001
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10



Conclusion:

- The more refined the time step size was, the closer the solution was to the analytical solution.
    - The numerical solution was more accurate for the more refined time steps.
- Solutions when gamma was greater than one looked very similar.
    - Gamma = 1 and Gamma = 2 graphs are almost the exact same
- Solutions when gamma was closer to zero looked more like a sine wave
    - Gamma = 0 and Gamma = 0.5
- The error (L-2 norm) went down as the time step size increased.

References:

Hughes, T. J. R. (2000). *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Dover Publications.

Chung, J. (1993). *A Time Integration Algorithm for Structural Dynamics with Improved Numerical Dissipation: The Generalized-α Method.* ASME.

Code

```
#!/usr/bin/env python3

import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os

print("\nStarting Program...\n")

"""
Nicolino Primavera
FEM for Fluid Flow and FSI Interactions
Assignment 3
11/27/24

A damped harmonic oscillator is governed by (m * (d^2u / dt^2)) + c (du / dt) + ku = 0          # EQN 1
   - m is the object's mass
   - c is the damping coefficient
   - k is the spring stiffness
   - u is the vector of displacement unknowns
   - γ (gamma) = c / cr is the damping ratio
   - cr = 2sqrt(mk) is the critical damping
   - the system behaves as an undamped (γ = 0), underdamped (0 < γ < 1), overdamped (γ > 1), or critically damped (γ = 1) system
   - ω_n = sqrt(k / m) denotes the natural frequency of the system

Equation can be rewritten as (d^2u / dt^2) + 2 * γ * ω_n * (du / dt) + (ω_n^2) * u = 0          # EQN 2
   - initial conditions u(0) = 1 , du/dt (0) = 1 , ω_n = pi

Solve EQN 2 for all the damping regimes (γ = 0, 0.5, 1, 2) using the generalized-α time integration method
   - For each γ, choose at least four values of the spectral radius parameters (0 ≤ ρ_∞ ≤ 1) and compare your numerical solution against the analytical solution
   - For each ρ_∞, refine the time step size, and comment on the error convergence and the numerical solution behavior
"""

# Initial conditions
u_0 = 1                 # initial displacement
dudt_0 = 1              # initial velocity
```

```
ω_n = math.pi              # natural frequency
m = 1                      # object mass
k = ω_n**2 * m             # spring stiffness
c_r = 2 * math.sqrt(m * k)  # critical damping

# Define gammas (γ) - for all the damping regimes
gammas = [0, 0.5, 1, 2]  # undamped, underdamped, critically damped, overdamped

# Damping coefficient
for gamma in gammas:
    c = 2*gamma*ω_n            # damping coefficient

# Define spectral radius parameters (ρ_∞)
rho_inf_values = [0, 0.25, 0.5, 1.0]    # spectral radius values

# Time parameters
time_steps = [0.1, 0.05, 0.01, 0.001, 0.0001]      # start with a coarse time step (0.1) and
gradually decrease it and observe convergence behavior
t_end = 10.0

# Analytical solution
def analytical_solution(gamma, t):
    """
    Analytical solution for the damped harmonic oscillator.

    (d^2u / dt^2) + 2 * γ * ω_n * (du / dt) + (ω_n^2) * u = 0   ,   set (du/dt) = r
    r^2 + 2*γ*ω_n*r + ω_n^2 = 0
        - For gamma = 0, undamped oscillation u(t) = e^-γ * sin(ω_nt)
        - For gamma < 1, underdamped oscillation (complex roots)
        - For gamma = 1, critically damped (double real root)
        - For gamma > 1, overdamped (two distinct real roots)
    """

    # Undamped case
    if gamma == 0:
        #return np.exp(-gamma * ω_n * t) * (np.cos(ω_n * t) + np.sin(ω_n * t))
        return (np.cos(ω_n * t) + np.sin(ω_n * t)) / ω_n

    # Underdamped case
    elif gamma < 1:
        ω_d = ω_n * np.sqrt(1 - gamma**2)  # Damped natural frequency
        #return np.exp(-gamma * ω_n * t) * (np.cos(ω_d * t) + np.sin(ω_d * t))
        return np.cos(ω_d * t) + (gamma * ω_n * np.sin(ω_d * t)) / ω_d

    # Critically damped case
    elif gamma == 1:
```

```
        #return np.exp(-ω_n * t) * (1 + ω_n * t)
        return (1 + ω_n * t) * np.exp(-ω_n * t)

    # Overdamped case
    else:  # gamma > 1
        λ1 = -ω_n * (gamma - np.sqrt(gamma**2 - 1))
        λ2 = -ω_n * (gamma + np.sqrt(gamma**2 - 1))
        C1, C2 = 1, 1   # constants based on initial conditions
        #return C1 * np.exp(λ1 * t) + C2 * np.exp(λ2 * t)
        return np.exp(λ1 * t) * (λ2 / (λ2 - λ1)) + np.exp(λ2 * t) * (-λ1 / (λ2 - λ1))

def solve_analytical_solution():
    """
    Solve and plot the analytical solution for all damping regimes.
    """
    t = np.linspace(0, 10, 100)  # Time from 0 to 10 seconds, 100 points

    # Analytical solutions for different damping regimes
    u_undamped = analytical_solution(gamma=0, t=t)
    u_underdamped = analytical_solution(gamma=0.5, t=t)
    u_critical = analytical_solution(gamma=1, t=t)
    u_overdamped = analytical_solution(gamma=2, t=t)

    # Error handling
    #print(f"\nAnalytical solution for the undamped case: \n {u_undamped}\n")
    #print(f"\nAnalytical solution for the underdamped case: \n {u_underdamped}\n")
    #print(f"Analytical solution for the critically damped case:\n {u_critical}\n")
    #print(f"Analytical solution for the overdamped case:\n {u_overdamped}\n")

    # Create folder to save plots
    save_folder = "/Users/nicolinoprimavera/Desktop/Columbia University/Finite Element
Method for Fluid Flow and Fluid-Structure Interactions/HW3/Plots"
    if not os.path.exists(save_folder):
        os.makedirs(save_folder)

    # Plot results
    plt.figure(figsize=(10, 6))
    plt.plot(t, u_undamped, label="Undamped (γ = 0)")
    plt.plot(t, u_underdamped, label="Underdamped (γ = 0.5)")
    plt.plot(t, u_critical, label="Critically Damped (γ = 1)")
    plt.plot(t, u_overdamped, label="Overdamped (γ = 2)")
    plt.title("Analytical Solution for Damped Harmonic Oscillator")
    plt.xlabel("Time (t)")
    plt.ylabel("Displacement (u)")
    plt.legend()
    plt.grid()
```

```
    # Save the plot
    filename = f"Analytical Solution for Damped Harmonic Oscillator.png"
    save_path = os.path.join(save_folder, filename)
    plt.savefig(save_path)
    print(f"\nGraph saved as {save_path}.\n")

    plt.show()
solve_analytical_solution()

# Generalized-alpha Time Integration Method
def generalized_alpha_time_integraton_method(gamma, rho_inf, dt, t_end):
    """
    Generalized-α Time Integration Method for second-order ODEs
        - proposed for a 2nd order ODE: M(d^2x/dt^2)+C(dx/dt)+DX=F
        - used for solving structural dynamics problems
        - possesses numerical dissipation that can be controlled by the user
        - achieves high-frequency dissipation while minimizing unwanted low-frequency
dissipation
        - controls high frequency oscillations

    Main Advantages:
        - implicit (unconditional stability)
        - 2nd order accurate ~ 0(Δt^2)
        - controls (damps) high frequency oscillations very effectively

    Parameters:
        - Spectral radius parameter, ρ_∞, controls the high-frequency damping
        - For each γ, choose at least four values of the spectral radius parameters (0 ≤ ρ_∞ ≤ 1) and
compare your numerical solution against the analytical solution
            - ex: ρ_∞ = 0,0.5,0.8,1.0
        - Formula for Beta(β) and Gamma(γ) based on ρ_∞ for numerical stability and accuracy:
            - β = (1 + ρ_∞)^2 / 4
            - γ = 0.5 + ρ_∞
            - stability parameters
    """

    # Compute the total number of time steps based on time increment (dt) and simulation
duration (t_end=10)
    num_steps = int(t_end / dt)
    #print(f"\nNumber of time steps:\n {num_steps}\n")

    # Initialize arrays to store displacement (u), velocity (v), and acceleration (a)
    u = np.zeros(num_steps)  # Displacement
    v = np.zeros(num_steps)  # Velocity
    a = np.zeros(num_steps)  # Acceleration
```

```
    # Initial conditions
    u[0] = u_0                                  # Initial displacement
    v[0] = dudt_0                               # Initial velocity
    a[0] = -2 * gamma * ω_n * v[0] - ω_n**2 * u[0]      # Initial acceleration from equation: a = -2γω_nv - ω_n^2u

    # Stability parameters based on spectral radius parameter (ρ_∞)
    α_m = (2 - rho_inf) / (1 + rho_inf)     # Mass matrix weighting factor for 2nd order systems
    α_f = 1 / (1 + rho_inf)                 # Force weighting factor

    # Guarantee 2nd Order Accuracy
    β = ((1 + α_f - α_m)**2)/4              # Integration parameter for displacement , high frequency dissipation is maximized
    γ = 0.5 - α_m + α_f                     # Integration parameter for velocity

    # Error handling
    #print(f"\nMass matrix wighting factor for 2nd order system: α_m = {α_m}\n")
    #print(f"\nForce weighting factor: α_f = {α_f}\n")
    #print(f"\nGamma(γ) = {γ}\n")

    # Time integration loop
    for n in range(num_steps - 1):
        # Predictor step for displacement and velocity
        y_u = u[n] + (dt * v[n]) + 0.5 * dt**2 * ((1 - 2 * β) * a[n] + 2 * β * a[n])   # Displacement (y_n+1) update equation
        y_v = v[n] + dt * ((1 - γ) * a[n] + γ * a[n])                                   # Velocity (ydot_n+1) update equation

        # Solve for acceleration at the next step using the residual equation: a[n+1] = (-2γω_nv_pred - ω_n^2u_pred) / (1 + 2γβω_n)
        a[n + 1] = (-2 * gamma * ω_n * y_v - ω_n**2 * y_u) / (1 + 2 * γ * β * ω_n)

        # Correct displacement using acceleration at the next time step
        u[n + 1] = y_u + β * dt**2 * a[n + 1]  # solve displacement

        # Correct velocity using acceleration at the next time step
        v[n + 1] = y_v + γ * dt * a[n + 1]  # solve velocity

    # Return time array and displacement solution
    return np.linspace(0, t_end, num_steps), u

# Main script
def solve():
    """
    Numerical Solution vs. Analytical Solution
```

```
    - compute the analytical solution for u(t)
    - plot u(t) for each gamma (γ) and spectral radius (ρ_∞)
    """

    # Create folder to save plots
    save_folder = "/Users/nicolinoprimavera/Desktop/Columbia University/Finite Element Method for Fluid Flow and Fluid-Structure Interactions/HW3/Plots"
    if not os.path.exists(save_folder):
        os.makedirs(save_folder)

    for gamma in gammas:
        print(f"Analyzing system for γ = {gamma}")

        # Preparing plot
        plt.figure(figsize=(10, 6))
        plt.title(f"Damped Harmonic Oscillator Solutions: γ = {gamma}")#, ρ_∞ = {rho_inf}")

        for rho_inf in rho_inf_values:
            for dt in time_steps:
                print(f"  - ρ_∞ = {rho_inf}, dt = {dt}")
                t, u_numerical = generalized_alpha_time_integraton_method(gamma, rho_inf, dt, t_end)


                # Plot - creates individual graphs for each gamma, spectral radius and time step and plots it against the analytical solution --> generates too many graphs
                #plt.figure()
                #plt.plot(t, u_numerical, label="Numerical Solution when time step = {dt}")
                #plt.plot(t, u_analytical, label="Analytical Solution", linestyle="dashed")
                #plt.title(f"Damped Harmonic Oscillator Solutions: γ = {gamma}, ρ_∞ = {rho_inf}") #, dt = {dt}")
                #plt.xlabel("Time (t)")
                #plt.ylabel("Displacement (u)")
                #plt.legend()
                #plt.grid()
                #plt.show()

                # Plot numerical solution for this time step (dt)
                plt.plot(t, u_numerical, label=f"ρ={rho_inf}, dt = {dt}")

                # Plot legend
                plt.legend(loc='upper right', fontsize='small', title='Legend')

        # Plot the analytical solution
        u_analytical = analytical_solution(gamma, t)
        plt.plot(t, u_analytical, label="Analytical Solution", linestyle="dashed", color="black")
```

```
        # Save the plot as a .png file with a unique name based on γ and ρ_∞
        filename = f"gamma_{gamma}_rho_inf_{rho_inf}.png"
        save_path = os.path.join(save_folder, filename)
        plt.savefig(save_path)
        print(f"\nGraph saved as {save_path}.\n")

        # Plot
        plt.xlabel("Time (t)")
        plt.ylabel("Displacement (u)")
        plt.grid(True)
        plt.tight_layout()

        # Display the plot with all dt solutions
        plt.show()
solve()

# Plot error convergence
def error_convergence():
    """
    Error Convergence
        - Calculate the error E = ||u_numerical - u_analytical|| using the L2-norm for each time step
size and ρ_∞
        - Plot E vs. time step size for each ρ_∞
    """

    # Create folder to save plots
    save_folder = "/Users/nicolinoprimavera/Desktop/Columbia University/Finite Element
Method for Fluid Flow and Fluid-Structure Interactions/HW3/Plots"
    if not os.path.exists(save_folder):
        os.makedirs(save_folder)

    errors = {}

    for rho_inf in rho_inf_values:
        errors[rho_inf] = []

        for dt in time_steps:
            # Numerical solution
            t_num, u_numerical = generalized_alpha_time_integraton_method(gamma=0.5,
rho_inf=rho_inf, dt=dt, t_end=t_end)

            # Analytical solution
            t_analytical = np.linspace(0, t_end, len(t_num))
            u_analytical = analytical_solution(gamma=0.5, t=t_analytical)
```

```
            # Compute the L2-norm error
            error = np.sqrt(np.sum((u_numerical - u_analytical) ** 2) * dt)
            errors[rho_inf].append(error)

    # Plot errors for each rho_inf
    plt.figure(figsize=(10, 6))
    for rho_inf, error_values in errors.items():
        plt.plot(time_steps, error_values, marker='o', label=f'ρ_∞ = {rho_inf}')

    plt.xlabel('Time Step Size (Δt)', fontsize=12)
    plt.ylabel('Error (L2-norm)', fontsize=12)
    plt.title('Error Convergence for Different ρ_∞ Values', fontsize=14)
    plt.legend()
    plt.grid()
    plt.xscale('log')  # Use log scale for better visualization
    plt.yscale('log')  # Use log scale for better visualization

    # Save the plot
    filename = f"Error Convergence for Damped Harmonic Oscillator.png"
    save_path = os.path.join(save_folder, filename)
    plt.savefig(save_path)
    print(f"\nGraph saved as {save_path}.\n")

    plt.show()
error_convergence()

print("\nProgram Finished.\n")
```