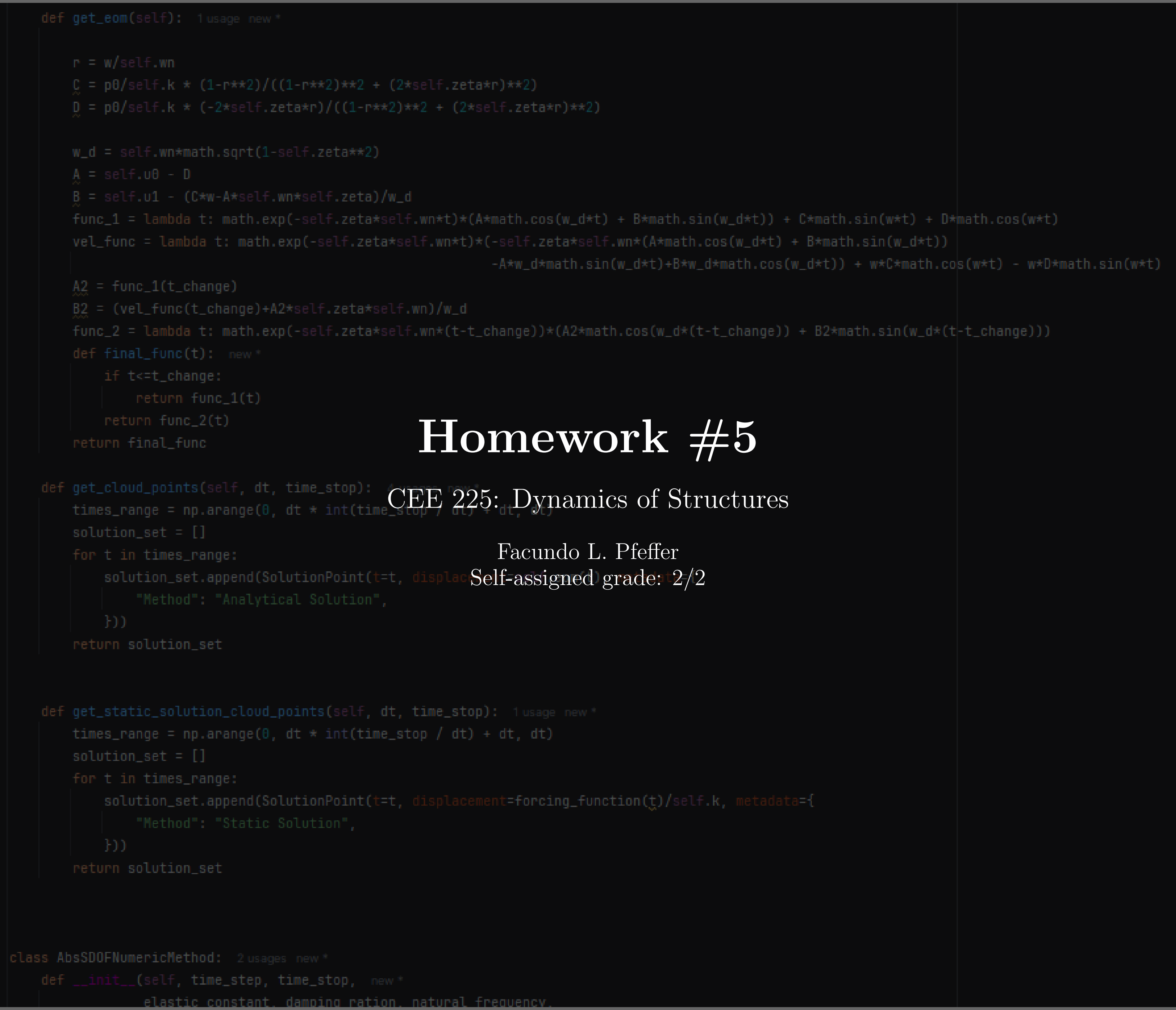

```
    def get_eom(self):  1 usage  new *

        r = w/self.wn
        C = p0/self.k * (1-r**2)/((1-r**2)**2 + (2*self.zeta*r)**2)
        D = p0/self.k * (-2*self.zeta*r)/((1-r**2)**2 + (2*self.zeta*r)**2)

        w_d = self.wn*math.sqrt(1-self.zeta**2)
        A = self.u0 - D
        B = self.u1 - (C*w-A*self.wn*self.zeta)/w_d
        func_1 = lambda t: math.exp(-self.zeta*self.wn*t)*(A*math.cos(w_d*t) + B*math.sin(w_d*t)) + C*math.sin(w*t) + D*math.cos(w*t)
        vel_func = lambda t: math.exp(-self.zeta*self.wn*t)*(-self.zeta*self.wn*(A*math.cos(w_d*t) + B*math.sin(w_d*t))
                                                     -A*w_d*math.sin(w_d*t)+B*w_d*math.cos(w_d*t)) + w*C*math.cos(w*t) - w*D*math.sin(w*t)
        A2 = func_1(t_change)
        B2 = (vel_func(t_change)+A2*self.zeta*self.wn)/w_d
        func_2 = lambda t: math.exp(-self.zeta*self.wn*(t-t_change))*(A2*math.cos(w_d*(t-t_change)) + B2*math.sin(w_d*(t-t_change)))
        def final_func(t):  new *
            if t<=t_change:
                return func_1(t)
            return func_2(t)
        return final_func

    def get_cloud_points(self, dt, time_stop):  [illegible]
        times_range = np.arange(0, dt * int(time_stop / dt) + dt, dt)
        solution_set = []
        for t in times_range:
            solution_set.append(SolutionPoint(t=t, displac[illegible]
                "Method": "Analytical Solution",
            }))
        return solution_set


    def get_static_solution_cloud_points(self, dt, time_stop):  1 usage  new *
        times_range = np.arange(0, dt * int(time_stop / dt) + dt, dt)
        solution_set = []
        for t in times_range:
            solution_set.append(SolutionPoint(t=t, displacement=forcing_function(t)/self.k, metadata={
                "Method": "Static Solution",
            }))
        return solution_set



class AbsSDOFNumericMethod:  2 usages  new *
    def __init__(self, time_step, time_stop,  new *
                 elastic_constant, damping_ration, natural_frequency,
```

Homework #5

CEE 225: Dynamics of Structures

Facundo L. Pfeffer

Self-assigned grade: 2/2

Self-Grading Declaration

Based on the guidelines provided, **I self-assign a grade of 2/2 for this homework**. I have made a legitimate effort to complete 100% of the problems.

Repository Access

All Python source codes, generated figures, and HTML visualizations developed for this study are publicly available in the following repository:

GitHub Repository: `https://github.com/Facundo-Pfeffer/UCBerkeley-SEMM-MS-Codebook/tree/85b93e5589ace430a7fa40341f686eb45bbd9d2f/CEE225%20%20-%20Dynamics/HW5%20CE%20220%20-%20Facundo%20Pfeffer`

The repository includes:

- The complete implementation of the numerical integration algorithms (*Central Difference Method* and *Average Acceleration Method*).
- Analytical and numerical comparison plots in both PNG and interactive HTML format.
- Supporting documentation and input files for reproducibility.

AI Usage Statement

During the development of this assignment, artificial intelligence was primarily employed to enhance the quality of writing within the LaTeX environment and to refine the implemented code by improving its documentation and overall professionalism, without intervening in the core development process. From a programming standpoint, AI assistance represented approximately 15% of the total work. The main model utilized was GPT-5.0[1].

1 Problem Statement

Assignment 1

Problem Statement

An SDOF system has the following properties:

$$k = 5 \text{ kips/in},\quad T_n = 1 \text{ sec},\quad \zeta = 0.05$$

The following forcing function is applied:

$$p(t) = \begin{cases} 8\sin\left(\dfrac{\pi t}{0.4}\right) \text{ kips}, & 0 \le t \le 1.2 \text{ sec}, \\ 0, & t > 1.2 \text{ sec}. \end{cases}$$

1. Determine the analytical (exact) solution of the equation of motion of the system using any method.
2. Determine the response $u(t)$ of this system using the *central difference method*, implemented in a computer program in a language of your choice, using $\Delta t = 0.1$ sec. Provide your solution as a table of $u(t)$ values.
3. Determine the response $u(t)$ of this system using the *constant average acceleration method*, implemented in a computer program in a language of your choice, using $\Delta t = 0.1$ sec. Provide your solution as a table of $u(t)$ values.
4. (a) Plot the solutions obtained in (1) and (2), along with the static solution $u_s(t) = \frac{p(t)}{k}$ and the analytical solution of part (1). Compare all four and comment on the comparison.
   (b) Now solve again using both the central difference method and the constant acceleration method using three different levels of damping: $\zeta = 0.01$, $\zeta = 0.1$, and $\zeta = 0.25$. Plot all three solutions together for each method. Comment on how the methods and the damping affect the peak response.
   (c) Now solve again using both the central difference method and the constant acceleration method using three different time steps: $\Delta t = 0.05$, $\Delta t = 0.2$, and $\Delta t = 0.35$ seconds. Carry out your solution to 4 seconds. Plot all three solutions together for each method. Comment on what happens to both solutions and why.

1.1 Exact Solution

Including viscous damping, the differential equation that governs the motion of a single-degree-of-freedom (SDF) system under harmonic loading is:

$$m\ddot{u} + c\dot{u} + ku = p_0 \sin \omega t \tag{1.1}$$

The equation above is solved subject to the initial conditions

$$u(0) = 0 \text{ in}, \qquad \dot{u}(0) = 0 \text{ in/s}.$$

The particular (steady-state) solution can be written as [2, 3]

$$u_p(t) = C \sin \omega t + D \cos \omega t \tag{1.2}$$

with coefficients

$$C = \frac{p_0}{k} \frac{1 - (\omega/\omega_n)^2}{[1 - (\omega/\omega_n)^2]^2 + [2\zeta(\omega/\omega_n)]^2},$$
$$D = \frac{p_0}{k} \frac{-2\zeta(\omega/\omega_n)}{[1 - (\omega/\omega_n)^2]^2 + [2\zeta(\omega/\omega_n)]^2}.$$

The complementary (homogeneous) solution of **Eq. (1.1)** represents the free vibration response:

$$u_c(t) = e^{-\zeta\omega_n t}\big(A \cos \omega_D t + B \sin \omega_D t\big), \tag{1.3}$$

where $\omega_D = \omega_n \sqrt{1 - \zeta^2}$ is the damped natural frequency.
Hence, the complete solution of **Eq. (1.1)** is

$$u(t) = \underbrace{e^{-\zeta\omega_n t}\big(A \cos \omega_D t + B \sin \omega_D t\big)}_{\text{transient}} + \underbrace{C \sin \omega t + D \cos \omega t}_{\text{steady-state}}. \tag{1.4}$$

To determine A and B, apply the initial conditions at $t = 0$:

$$u(0) = u_c(0) + u_p(0) = A + D = 0 \;\;\Rightarrow\;\; A = -D.$$

The velocity condition uses

$$\dot{u}_c(t) = e^{-\zeta\omega_n t}\left[-\zeta\omega_n\big(A \cos \omega_D t + B \sin \omega_D t\big) - A\omega_D \sin \omega_D t + B\omega_D \cos \omega_D t\right],$$
$$\dot{u}_p(t) = \omega C \cos \omega t - \omega D \sin \omega t.$$

Evaluating at $t = 0$ (i.e., $\cos 0 = 1$, $\sin 0 = 0$) gives

$$\dot{u}(0) = -\zeta\omega_n A + \omega_D B + \omega C = 0 \;\;\Rightarrow\;\; B = \frac{\zeta\omega_n A - \omega C}{\omega_D}.$$

Substituting $A = -D$ yields the final constants

$$A = -D, \qquad B = \frac{-\zeta\omega_n D - \omega C}{\omega_D}.$$

Remark. The A-term in $\dot{u}(0)$ does not vanish in general because the derivative of the exponential factor contributes $-\zeta\omega_n A$ at $t = 0$. It only disappears in the undamped case $\zeta = 0$.

1.2 Piecewise Forcing and Two–Law Analytical Solution

Let t_c denote the time at which the harmonic load ceases (here, $t_c = 1.2$ s). The forcing is

$$p(t) = \begin{cases} p_0 \sin(\omega t), & 0 \le t \le t_c, \\ 0, & t > t_c, \end{cases} \tag{1.5}$$

so the equation of motion **Eq. (1.1)** is integrated in two phases:

Phase I (forced, $0 \le t \le t_c$). With the steady–state form $u_p(t) = C\sin\omega t + D\cos\omega t$ (coefficients C, D as above) and the complementary part $u_c(t) = e^{-\zeta\omega_n t}\big(A\cos\omega_D t + B\sin\omega_D t\big)$ (where $\omega_D = \omega_n\sqrt{1-\zeta^2}$), the total response is

$$u_1(t) = e^{-\zeta\omega_n t}\big(A\cos\omega_D t + B\sin\omega_D t\big) + C\sin\omega t + D\cos\omega t.$$

For general initial conditions $u(0) = u_0$, $\dot u(0) = u_1$ (units suppressed), application of the conditions at $t = 0$ gives

$$A = u_0 - D, \qquad B = \frac{u_1 + \zeta\omega_n A - \omega C}{\omega_D}.$$

(In our case $u_0 = \dot u_0 = 0 \Rightarrow A = -D,\ B = (\zeta\omega_n A - \omega C)/\omega_D$.)

Phase II (free vibration, $t > t_c$). For $t > t_c$ the load is zero, and the motion is free vibration referenced to t_c:

$$u_2(t) = e^{-\zeta\omega_n (t-t_c)}\Big(A_2\cos\big[\omega_D(t-t_c)\big] + B_2\sin\big[\omega_D(t-t_c)\big]\Big).$$

Continuity of displacement and velocity at t_c determines the new constants. Let

$$u_c^{(1)} = u_1(t_c), \qquad \dot u_c^{(1)} = \dot u_1(t_c),$$

then evaluating u_2 and $\dot u_2$ at $t = t_c$ yields

$$A_2 = u_c^{(1)}, \qquad B_2 = \frac{\dot u_c^{(1)} + \zeta\omega_n A_2}{\omega_D}.$$

This guarantees u and $\dot u$ are C^0 and C^0-continuous, respectively, across the switching time.

Python implementation (sketch). The above two laws are encoded in the analysis routine as follows. First, the piecewise load:

Python 3.13 Code

```python
@lru_cache(512)
def forcing_function(t: float):
    if t < 0:
        raise ValueError("t must be >= 0")
    elif t <= t_change:
        return p0*math.sin(w*t)
    return 0.0
```

The analytical solution is constructed phase-by-phase. In Phase I we compute $r = \omega/\omega_n$, the steady-state coefficients C, D, and the transient constants A, B from the initial conditions; then we evaluate the state at t_c and start Phase II with free vibration referenced to t_c:

Python 3.13 Code

```python
    def get_eom(self):
        r = w/self.wn
        C = p0/self.k * (1-r**2)/((1-r**2)**2 + (2*self.zeta*r)**2)
        D = p0/self.k * (-2*self.zeta*r)/((1-r**2)**2 + (2*self.zeta*r)**2)

        w_d = self.wn*math.sqrt(1-self.zeta**2)
        A = self.u0 - D
        B = self.u1 - (C*w-A*self.wn*self.zeta)/w_d
        func_1 = lambda t: math.exp(-self.zeta*self.wn*t)*(A*math.cos(w_d*t) +
            B*math.sin(w_d*t)) + C*math.sin(w*t) + D*math.cos(w*t)
        vel_func = lambda t: math.exp(-self.zeta*self.wn*t)*(-self.zeta*self.wn
            *(A*math.cos(w_d*t) + B*math.sin(w_d*t))
                                                   -A*w_d*math.sin(
                                                       w_d*t)+B*w_d*
                                                       math.cos(w_d*t)
                                                       ) + w*C*math.
                                                       cos(w*t) - w*D*
                                                       math.sin(w*t)
        A2 = func_1(t_change)
        B2 = (vel_func(t_change)+A2*self.zeta*self.wn)/w_d
        func_2 = lambda t: math.exp(-self.zeta*self.wn*(t-t_change))*(A2*math.
            cos(w_d*(t-t_change)) + B2*math.sin(w_d*(t-t_change)))
        def final_func(t):
            if t<=t_change:
                return func_1(t)
            return func_2(t)
        return final_func
```

Notes. (i) The comment on the line computing B clarifies the *general* expression for nonzero initial velocity. (ii) The switch is performed using the exact continuity relations $A_2 = u(t_c)$ and $B_2 = \big(\dot{u}(t_c) + \zeta\omega_n u(t_c)\big)/\omega_D$, which follow directly from the free-vibration form at $\tau = t - t_c = 0$.

2 Damping Ratio $\zeta = 0.05$

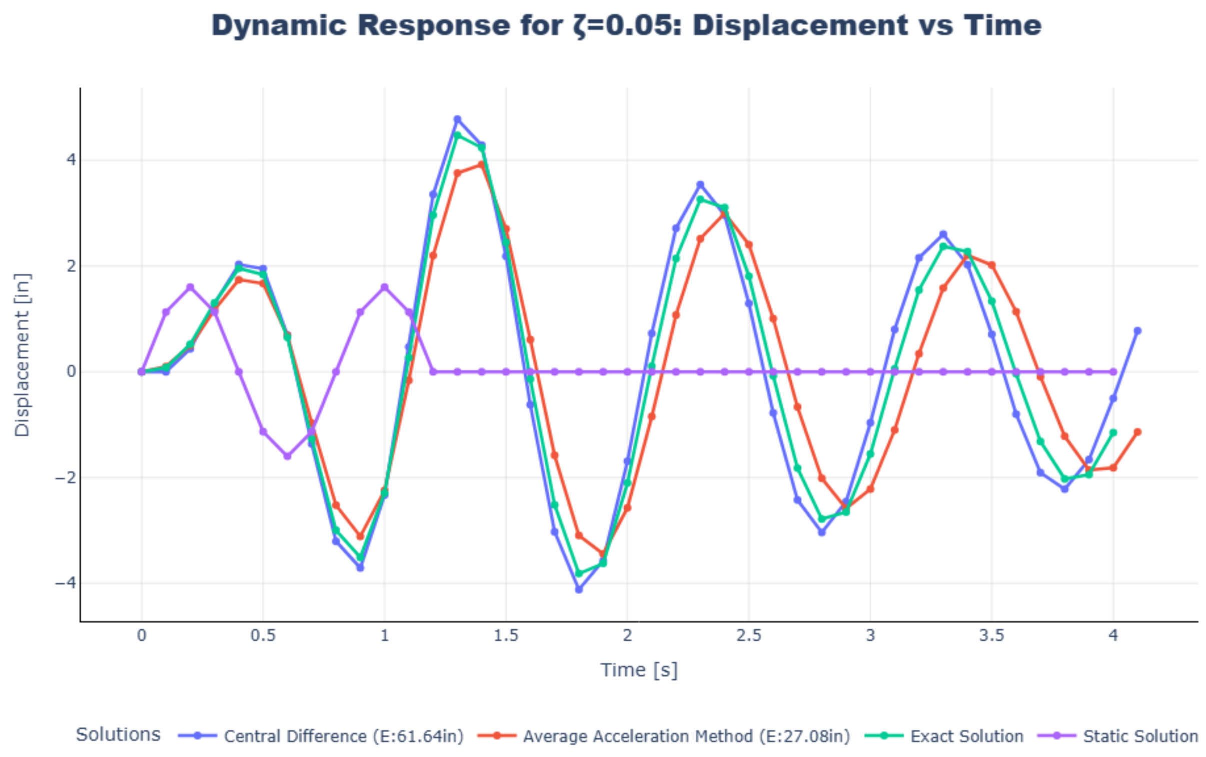

Figure 1: Dynamic response of the system for $\zeta = 0.05$: comparison between the Central Difference Method, Average Acceleration Method, and the Exact and Static Solutions.

The obtained responses for a damping ratio of $\zeta = 0.05$ are shown in **Fig. 1**. Both the Central Difference and Average Acceleration methods capture the oscillatory behavior of the system with satisfactory accuracy when compared to the exact solution. However, minor discrepancies are observed in the displacement amplitude, especially near the response peaks. These deviations are quantified through the error E, defined as the absolute difference between the computed and exact displacements at each discrete time step. The Central Difference Method yields a larger accumulated error ($E = 61.64\,\text{in}$) than the Average Acceleration Method ($E = 27.08\,\text{in}$), confirming that the latter provides a more stable and accurate time integration scheme for this damping ratio.

3 Damping Ratio Influence on Dynamic Response

The dynamic responses for different damping ratios are summarized in Figures **Fig. 2**–**Fig. 4**. Given that the natural period of the system is $T = 1\,\text{s}$, it can be observed that all responses complete approximately four cycles within the analyzed time window. This allows a clear visualization of how the damping ratio ζ influences both the amplitude reduction per cycle and the accuracy of the numerical integration schemes.

For a very small damping ratio ($\zeta = 0.01$), the response exhibits large oscillations with minimal amplitude decay over successive cycles. In this case, the Central Difference Method accumulates the largest error ($E = 93.52\,\text{in}$), while the Average Acceleration Method produces a smaller yet noticeable deviation ($E = 36.57\,\text{in}$). The persistence of large oscillations due to the near-undamped behavior makes the explicit scheme more prone to numerical instability and phase lag accumulation.

When the damping ratio increases to $\zeta = 0.10$, the energy dissipation becomes more evident: the displacement peaks gradually decrease with each cycle. Both numerical methods capture this decaying trend more accurately, although the Average Acceleration Method maintains superior precision ($E = 20.33\,\text{in}$) compared to the Central Difference Method ($E = 40.02\,\text{in}$). The moderate damping improves numerical stability by attenuating the high-frequency oscillations that typically amplify integration errors.

For the higher damping ratio of $\zeta = 0.25$, the oscillations are rapidly suppressed within approximately two cycles of the natural period. The dynamic response converges swiftly toward equilibrium, and both methods yield excellent agreement with the exact solution. The accumulated errors are substantially reduced ($E = 17.06\,\text{in}$ for the Central Difference Method and $E = 11.62\,\text{in}$ for the Average Acceleration Method). In this regime, the effects of numerical dispersion and phase errors become negligible due to the strong energy dissipation.

Overall, the results confirm that the damping ratio has a decisive influence on the system's dynamic response. Higher damping ratios reduce both the maximum displacement and the number of significant oscillations within the period range of interest. In addition, the Average Acceleration Method consistently provides more accurate predictions for all damping levels, while the Central Difference Method is more sensitive to low damping, where its explicit formulation tends to magnify small integration errors over multiple vibration cycles.

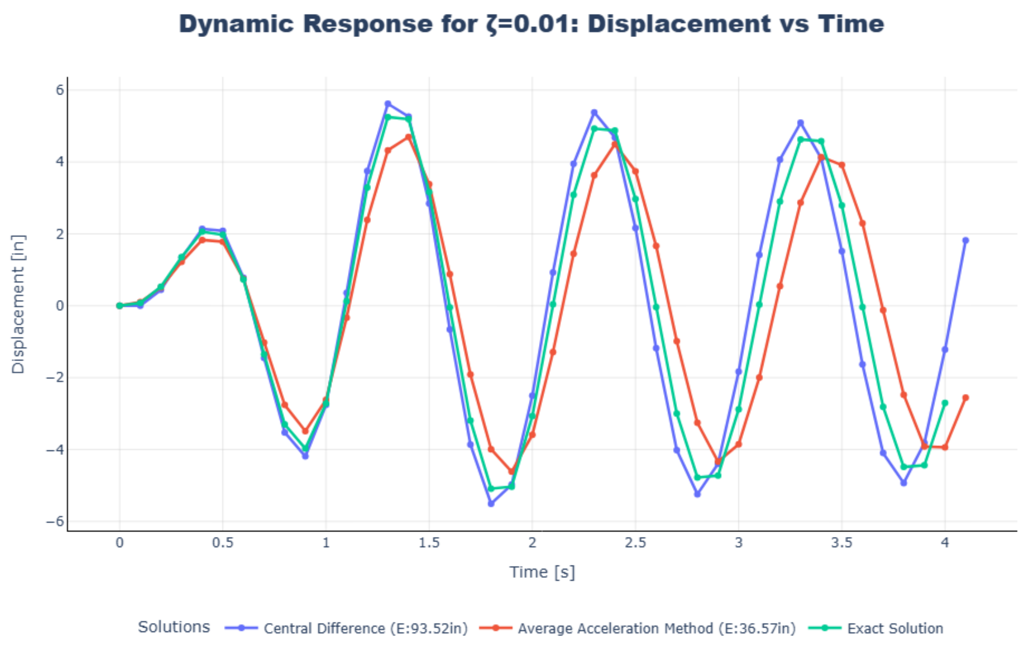

Figure 2: Dynamic response for $\zeta = 0.01$: comparison between the Central Difference Method, Average Acceleration Method, and the Exact Solution.

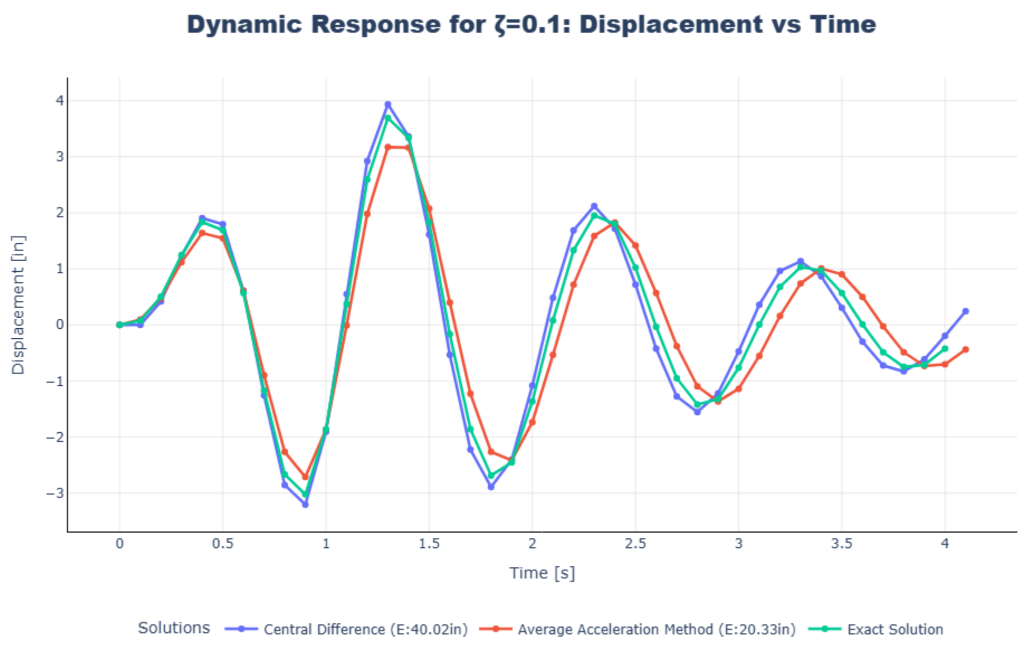

Figure 3: Dynamic response for $\zeta = 0.10$: comparison between the Central Difference Method, Average Acceleration Method, and the Exact Solution.

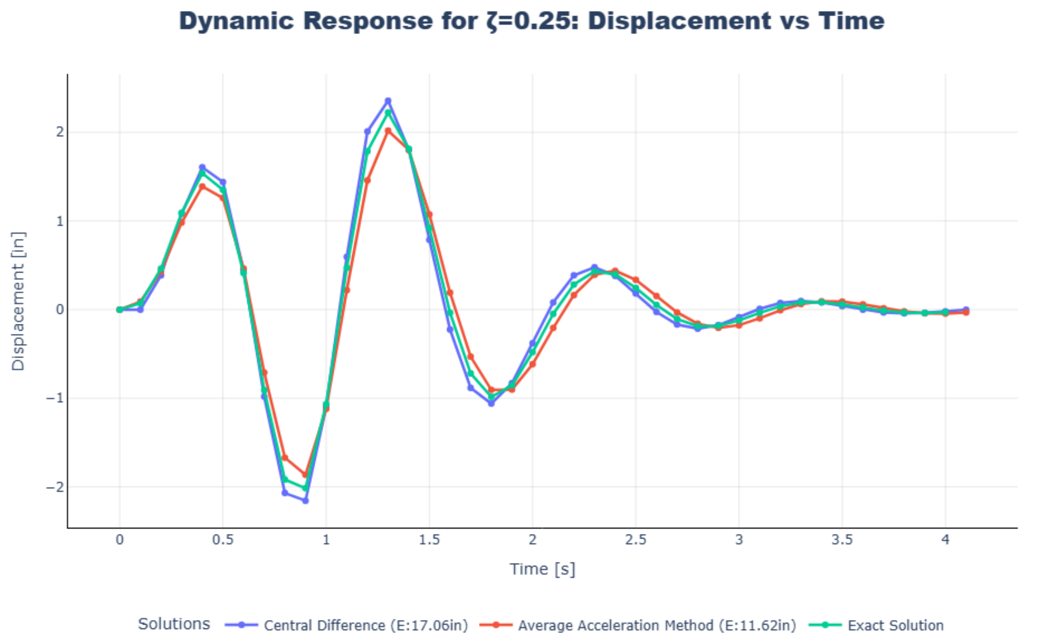

Figure 4: Dynamic response for $\zeta = 0.25$: comparison between the Central Difference Method, Average Acceleration Method, and the Exact Solution.

4 Influence of Time Step on Numerical Stability and Accuracy

The influence of the integration time step Δt on the computed dynamic response is illustrated in Figures **Fig. 5**–**Fig. 7**. The analyses were performed for a damping ratio of $\zeta = 0.05$ and a natural period of $T_n = 1\,\mathrm{s}$.

The results demonstrate that the choice of time step Δt plays a crucial role in both the accuracy and stability of the numerical integration schemes.

For the *Average Acceleration Method* (**Fig. 5**), the computed response remains stable for all tested time steps ($\Delta t = 0.05\,\mathrm{s}$, $0.2\,\mathrm{s}$, and $0.35\,\mathrm{s}$). Although small deviations in amplitude are observed for coarser time steps, the method exhibits unconditional stability, a well-known characteristic of this implicit integration scheme. The associated errors ($E = 37.13\,\mathrm{in}$, $24.03\,\mathrm{in}$, and $22.16\,\mathrm{in}$, respectively) indicate that increasing Δt does not cause divergence, but it reduces temporal resolution, slightly distorting the response shape.

In contrast, the *Central Difference Method* shows a strong dependence on the selected time step. As shown in **Fig. 6**, when $\Delta t = 0.35\,\mathrm{s}$, the solution becomes numerically unstable, leading to non-physical divergence with extremely large displacements ($E = 34{,}944.82\,\mathrm{in}$). This instability arises because the chosen step size exceeds the critical stability limit:

$$\Delta t_{\mathrm{cr}} = \frac{T_n}{\pi} \approx 0.318\,\mathrm{s}.$$

For $\Delta t > \Delta t_{\mathrm{cr}}$, the amplification matrix eigenvalues surpass unity in magnitude, causing exponential error growth over time.

When the time step is reduced below this limit (**Fig. 7**), the Central Difference Method produces stable and accurate results for $\Delta t = 0.05\,\mathrm{s}$ and $\Delta t = 0.2\,\mathrm{s}$. In these cases, the displacement time histories align closely with the exact solution, although a small phase shift is noticeable. The accumulated errors ($E = 54.86\,\mathrm{in}$ and $73.06\,\mathrm{in}$, respectively) confirm that finer time steps improve both amplitude and phase accuracy.

Overall, the comparison highlights that:

- The **Average Acceleration Method** is unconditionally stable, tolerating large Δt values without divergence, though accuracy benefits from smaller steps.
- The **Central Difference Method** is conditionally stable and requires $\Delta t < T_n/\pi$ to maintain physically meaningful results.
- Excessive time steps in explicit schemes amplify numerical errors rapidly, whereas implicit schemes remain robust at the expense of minor accuracy loss.

These findings underscore the practical importance of selecting an appropriate time increment in dynamic analysis: while implicit formulations guarantee convergence, explicit methods demand stricter control of Δt relative to the system's natural period to ensure stability.

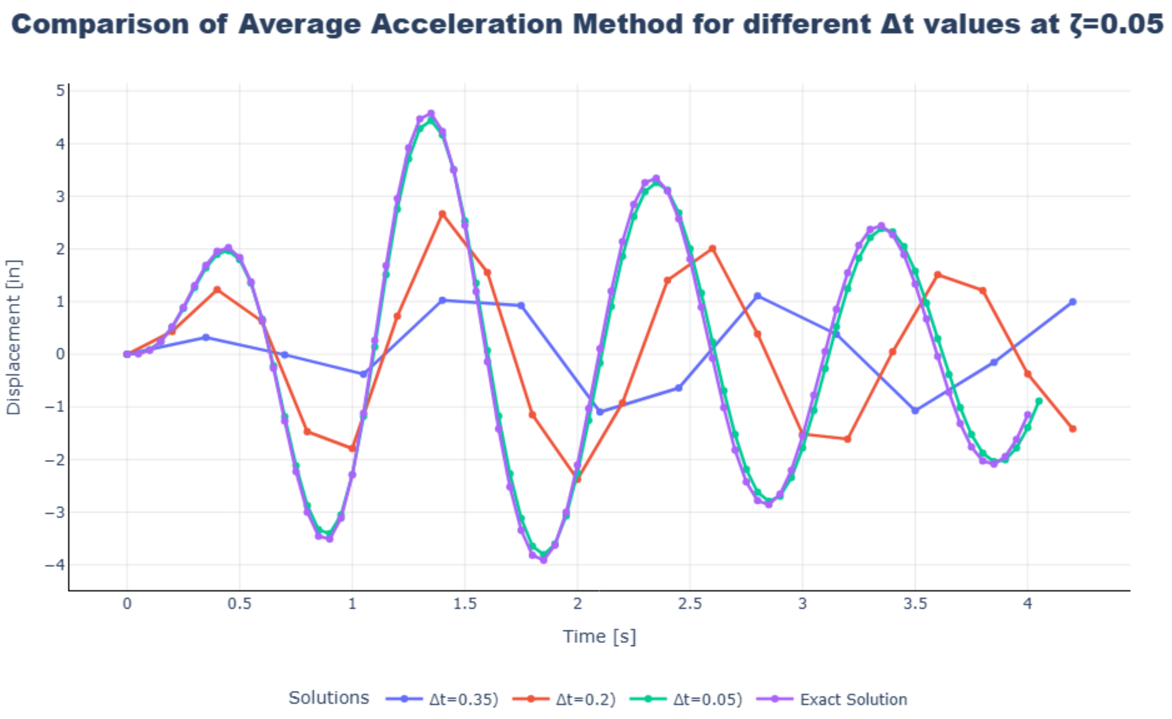

Figure 5: Comparison of the Average Acceleration Method for different Δt values at $\zeta = 0.05$.

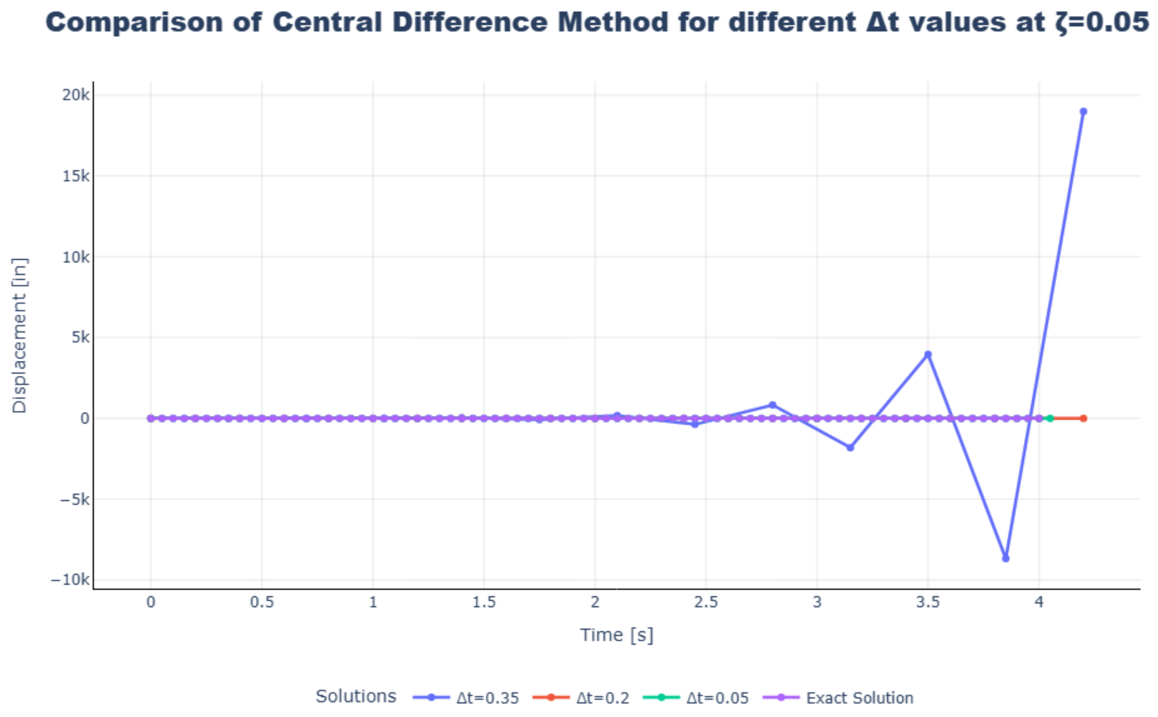

Figure 6: Comparison of the Central Difference Method for different Δt values at $\zeta = 0.05$ — unstable case.

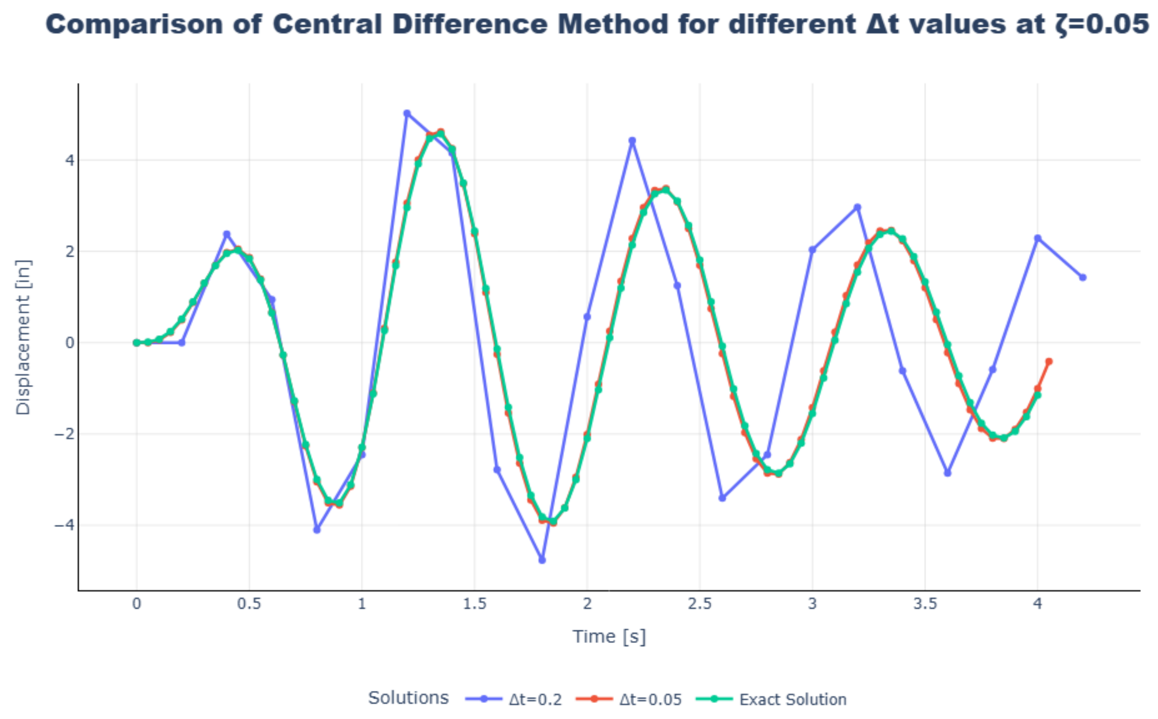

Figure 7: Comparison of the Central Difference Method for different Δt values at $\zeta = 0.05$ — stable case.

A Python Implementation of the Numerical Solution

The following appendix presents the complete Python code developed to compute and visualize the dynamic response of a single degree of freedom (SDOF) system subjected to harmonic excitation.

GitHub Repository: https://github.com/Facundo-Pfeffer/UCBerkeley-SEMM-MS-Codebook/tree/85b93e5589ace430a7fa40341f686eb45bbd9d2f/CEE225%20%20-%20Dynamics/HW5%20CE%20220%20-%20Facundo%20Pfeffer

The implementation compares the *Central Difference Method* and the *Average Acceleration Method* against the analytical (exact) solution. Each section of the code is briefly described to clarify its role in the overall analysis.

The libraries used for the development of this assignment are plotly[4] and numpy[5]

A.1 Main Script and Forcing Function Definition

Python 3.13 Code

```python
import numpy as np
import math
from functools import lru_cache
from plotly_generator import plot_displacement_vs_time

wn = math.pi*2 # Natural frequency [rad/s]
zeta = 0.05 # Damping ratio
k = 5  # Elastic constant [kips/in]

p0= 8
w = math.pi/0.4
t_change = 1.2

@lru_cache(512)
def forcing_function(t: float):
    if t < 0:
        raise ValueError("Error: the analyzed time should always be t>0")
    elif t<=t_change:
        return p0*math.sin(t*w)
    return 0
```

This section imports required libraries and defines the forcing function, a harmonic load $p(t) = p_0 \sin(\omega t)$ acting until $t_{\text{change}} = 1.2\,\text{s}$. The decorator `@lru_cache(512)` improves performance by caching previously computed values.

A.2 Class SolutionPoint

Python 3.13 Code

```python
class SolutionPoint:
    """Single solution point to be plotted."""
    def __init__(self, t, displacement, velocity=None, acceleration=None,
        metadata:dict = None):
        metadata = metadata or {}
        self.t = t
        self.u = displacement
        self.v = velocity
        self.a = acceleration
        self.metadata = metadata
```

The `SolutionPoint` class stores the time step data: displacement, velocity, acceleration, and metadata used for plotting and comparison with other methods.

A.3 Class SDOFHarmonicVibration

Python 3.13 Code

```python
class SDOFHarmonicVibration:
    """Harmonic vibration of a single degree of freedom until t_change, then
        free vibration."""
    def __init__(self,
                 elastic_constant, damping_ration, natural_frequency,
                 initial_displacement=None, initial_velocity=None,
                     initial_acceleration=None,
                 forcing_function=forcing_function, kwargs):
        ...
```

This class represents the analytical (exact) solution of the SDOF oscillator. It computes the displacement response both under harmonic excitation (for $t \leq t_{\text{change}}$) and during the free vibration phase that follows.

- `populate_sdof_constants`: derives m and c from k, ζ, and ω_n.
- `populate_initial_conditions`: ensures consistency among $u(0)$, $\dot{u}(0)$, and $\ddot{u}(0)$.
- `get_eom`: defines the analytical displacement and velocity functions using the standard damped harmonic solution.
- `get_cloud_points`: generates the displacement–time dataset for plotting.

A.4 Abstract Class `AbsSDOFNumericMethod`

Python 3.13 Code

```python
class AbsSDOFNumericMethod:
    def __init__(self, time_step, time_stop,
                 elastic_constant, damping_ration, natural_frequency,
                 initial_displacement=None, initial_velocity=None,
                    initial_acceleration=None,
                 forcing_function=forcing_function,
                 exact_solution=None):
        ...
```

This abstract class defines shared operations for both numerical methods, including:

- Calculation of system parameters (m, c, k).
- Initialization of displacement, velocity, and acceleration.
- Error evaluation by comparing each computed displacement with the exact analytical solution.

A.5 Central Difference Method

Python 3.13 Code

```python
class CentralDifferenceMethod(AbsSDOFNumericMethod):
    def __init__(self, kwargs):
        super().__init__(kwargs)
        self.k_hat, self.a, self.b = self.populate_method_constants()
        ...
```

This class implements the explicit Central Difference Method. It calculates successive displacements u_{i+1} using:

$$u_{i+1} = \frac{p_i - a\,u_{i-1} - b\,u_i}{\hat{k}},$$

where $\hat{k}$, a, and b depend on m, c, and Δt. The method is conditionally stable and diverges if $\Delta t > T_n/\pi$.

A.6 Average Acceleration Method

Python 3.13 Code

```python
class AverageAccelerationMethod(AbsSDOFNumericMethod):
    def __init__(self, kwargs):
        super().__init__(kwargs)
        self.a1, self.a2, self.a3, self.k_hat = self.populate_method_constants
            ()
        ...
```

This class implements the Average Acceleration Method, an implicit unconditionally stable time integration algorithm. It computes displacement, velocity, and acceleration iteratively using a constant average acceleration assumption over each step:

$$\begin{aligned} u_{i+1} &= \frac{\hat{p}_{i+1}}{\hat{k}}, \\ \dot{u}_{i+1} &= \frac{2}{\Delta t}(u_{i+1} - u_i) - \dot{u}_i, \\ \ddot{u}_{i+1} &= \frac{4}{\Delta t^2}(u_{i+1} - u_i) - \frac{4}{\Delta t}\dot{u}_i - \ddot{u}_i. \end{aligned}$$

Although larger Δt values reduce accuracy, they do not compromise stability.

A.7 Main Execution and Plotting

Python 3.13 Code

```python
if __name__ == "__main__":
    problem_a_params = dict(
        time_step=0.1,
        time_stop=4,
        elastic_constant=k,
        damping_ration=zeta,
        natural_frequency=wn,
        initial_displacement=0,
        initial_velocity=0,
    )
    ...
```

The main script executes three experiments:

1. **Problem A:** Comparison of both methods for $\zeta = 0.05$.
2. **Problem B:** Study of the damping effect for $\zeta = 0.01$, 0.10, and 0.25.
3. **Problem C:** Investigation of the influence of time step Δt on numerical stability.

The function `plot_displacement_vs_time()` generates interactive Plotly figures illustrating each response, enabling direct comparison of numerical and exact solutions.

A.8 Summary of the Implementation

The code demonstrates the effect of damping and time discretization on the dynamic response of an SDOF oscillator. The explicit Central Difference Method requires $\Delta t < T_n/\pi$ to remain stable, while the implicit Average Acceleration Method remains stable for any Δt. The total absolute error accumulated throughout the simulation quantifies the accuracy of each method relative to the analytical solution.

B References

[1] OpenAI, "Introducing gpt-5," Aug. 2025.

[2] A. K. Chopra, *Dynamics of Structures: Theory and Applications to Earthquake Engineering.* Boston: Pearson Education, 4th, global edition ed., 2014.

[3] M. DeJong, F. Filippou, S. Govindjee, A. D. Kiureghian, K. Mosalam, J. Moehle, and E. Opabola, "Semm graduate program primer: 2025," SEMM Reports Series UCB/SEMM-2025/04, University of California, Berkeley, July 2025.

[4] P. T. Inc., "Plotly: Collaborative data science." https://plotly.com/python/, 2015. Accessed: 2025-10-04.

[5] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, 2020.