Part 2: Wave Equations

Solving Generalized Wave Equations with the Finite Difference Method

Petar Bosnic

University of South-Eastern Norway

November 13, 2024

Note

This document was developed as part of the PhD course *Numerical Solutions to Partial Differential Equations* under the guidance of Professors Svein Linge and Knut Vågsæther at the University of South-Eastern Norway (USN). Course material includes the textbooks:

- *Finite Difference Computing with PDEs: A Modern Software Approach* by Hans Petter Langtangen and Svein Linge (DOI: 10.1007/978-3-319-55456-3)
- *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction* by E. F. Toro (ISBN: 978-3-540-25202-3 978-3-540-49834-6)

Contents

1 Introduction

The wave equation is a fundamental partial differential equation (PDE) that describes a broad range of physical phenomena involving waves or vibrations. It is crucial in fields such as acoustics, electromagnetics, seismology, and fluid dynamics, providing a framework to model how waves propagate in different media. The wave equation models the behavior of oscillatory systems, capturing the essence of how disturbances travel through a medium over time and space.

The generalized form of the wave equation in one spatial dimension is expressed as:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} + f(x,t), \tag{1}$$

or, in a more compact notation:

$$u_{tt} = c^2 u_{xx} + f(x,t), \tag{2}$$

where:

- $u(x,t)$ represents the displacement of the wave at position x and time t.
- c is the wave propagation speed, which depends on the physical properties of the medium (e.g., tension and density for a vibrating string or compressibility and density for sound waves in air).
- $f(x,t)$ is an optional source term that introduces external forces, driving terms, or other energy sources that can affect the wave's behavior over time and space.

This PDE states that the acceleration of the displacement field, represented by the second derivative with respect to time, u_{tt}, is proportional to the spatial curvature of the displacement field, u_{xx}. In regions where the curvature is high, the acceleration is also high, causing the wave to propagate by transferring energy through the medium.

1.1 Finite Difference Approximation of the Wave Equation

The finite difference method transforms the continuous partial differential wave equation into a discrete approximation, allowing it to be solved numerically. To approximate the second derivatives in time and space, we begin by discretizing both the time and spatial domains into small steps. Let $x_i = i\Delta x$ denote discrete spatial points, and $t_n = n\Delta t$ denote discrete time points, where Δx is the spatial step size and Δt is the time step size.

1.1.1 Approximating the Temporal Derivative with Central Difference Scheme

In the wave equation, the second derivative with respect to time, u_{tt}, represents the acceleration of the wave at a point. Using the central difference method, we can approximate u_{tt} at a point (x_i, t_n) by the difference between values at adjacent time steps, as follows:

$$u_{tt} \approx \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2}, \tag{3}$$

where:

- u_i^{n+1} is the displacement at the next time step.
- u_i^n is the displacement at the current time step.
- u_i^{n-1} is the displacement at the previous time step.
- Δt is the time step size.

This approximation is derived by applying the Taylor expansion to $u(x_i, t_n \pm \Delta t)$, yielding an accurate estimate of the second derivative in time for small Δt.

1.1.2 Approximating the Spatial Derivative with Central Difference Scheme

Similarly, the second derivative with respect to space, u_{xx}, represents the curvature of the wave at a point, indicating how steeply the wave changes between neighboring spatial points. Using the central difference method, we approximate u_{xx} at (x_i, t_n) by:

$$u_{xx} \approx \frac{u_{i-1}^n - 2u_i^n + u_{i+1}^n}{\Delta x^2}, \tag{4}$$

where:

- u_{i-1}^n and u_{i+1}^n are the displacements at the neighboring spatial points to the left and right of x_i.
- u_i^n is the displacement at the current spatial point.
- Δx is the spatial step size.

This approximation is also derived by applying the Taylor expansion, making it accurate for small values of Δx.

1.1.3 Substituting into the Wave Equation

Now, we substitute these finite difference approximations into the original wave equation, given by:

$$u_{tt} = c^2 u_{xx} + f(x, t). \tag{5}$$

For simplicity, let's consider the case with no source term, $f(x, t) = 0$. Substituting the finite difference approximations for u_{tt} and u_{xx}, we get:

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = c^2 \frac{u_{i-1}^n - 2u_i^n + u_{i+1}^n}{\Delta x^2}. \tag{6}$$

To simplify, we multiply both sides by Δt^2, yielding:

$$u_i^{n+1} - 2u_i^n + u_i^{n-1} = \frac{c^2 \Delta t^2}{\Delta x^2} \left(u_{i-1}^n - 2u_i^n + u_{i+1}^n \right). \tag{7}$$

1.1.4 Introducing the Courant Number

We introduce the Courant number, C, which is a dimensionless parameter defined as:

$$C = \frac{c\Delta t}{\Delta x}. \tag{8}$$

The Courant number, C, encapsulates the relationship between the time step size, Δt, the spatial step size, Δx, and the wave propagation speed, c. This number is significant because it indicates how far information (or a "signal") can travel across the spatial grid in one time step. It acts as a stability condition for the numerical scheme.

By substituting C^2 into the finite difference expression derived from the wave equation, we obtain the following update rule:

$$u_i^{n+1} = 2u_i^n - u_i^{n-1} + C^2 \left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right). \tag{9}$$

This is an explicit finite difference scheme for the wave equation, where u_i^{n+1}, the displacement at the next time step, is computed based on values at the current time step, u_i^n, the previous time step, u_i^{n-1}, and neighboring spatial points u_{i-1}^n and u_{i+1}^n.

Why This Expression? The form of this update rule arises from the central difference approximations applied to the wave equation's second derivatives in time and space. By utilizing central differences, we ensure that the approximation maintains second-order accuracy in both time and space, meaning the error decreases quadratically as the step sizes decrease. This specific form also ensures that the solution depends only on known values from previous steps, making it computationally efficient and straightforward to implement.

Explicit Solution and Stability Considerations An "explicit" method means that the solution at the next time step, u_i^{n+1}, can be computed directly using known values at the current and previous time steps. This contrasts with "implicit" methods, where unknowns at the next time step would need to be solved simultaneously, typically involving more computational effort. Explicit methods, like the one derived here, are usually easier to implement but can be conditionally stable.

For the explicit finite difference scheme to remain stable, the Courant number C must satisfy a critical condition:

$$C \leq 1. \tag{10}$$

This condition is crucial because, if C exceeds this threshold, the numerical solution can become unstable, resulting in growing oscillations and inaccurate results. Physically, this stability criterion ensures that the numerical propagation of information through the grid does not exceed the actual wave propagation speed, thereby preserving the integrity of the solution. Meeting this condition requires choosing Δt and Δx carefully in relation to the wave speed c, which plays a central role in wave dynamics.

In summary, the Courant number C is vital in explicit finite difference schemes for the wave equation, governing both the stability and accuracy of the numerical solution.

2 Understanding Time and Spatial Stepping in the Solver

The finite difference method used in this solver divides the spatial and temporal domains into discrete steps, represented by a grid of points. Each point on this grid corresponds to a value of

$u(x,t)$ at a specific spatial position x and a specific time t. The solution advances through each time step, with values at the next time level determined by values at the current and previous time steps.

2.1 2D Grid for Time and Spatial Stepping

To better visualize this process, imagine a 2D grid where the x-axis represents spatial points and the t-axis represents time steps. The solver calculates values of $u(x,t)$ for each grid point (x_i, t_n) iteratively, meaning each point at a new time step depends on the values from previous steps.

Time Steps (t)	x_0	x_1	x_2	$\cdots$	x_N
t_0	u_0^0	u_1^0	u_2^0	$\cdots$	u_N^0
t_1	u_0^1	u_1^1	u_2^1	$\cdots$	u_N^1
t_2	u_0^2	u_1^2	u_2^2	$\cdots$	u_N^2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
t_n	u_0^n	u_1^n	u_2^n	$\cdots$	u_N^n

2.2 Finite Difference Update Rule

To compute the values at the next time step, the solver applies the following finite difference update rule:

$$u_i^{n+1} = 2u_i^n - u_i^{n-1} + C^2(u_{i-1}^n - 2u_i^n + u_{i+1}^n), \tag{11}$$

where:

- u_i^{n+1} is the value at the next time step for position x_i.
- u_i^n is the value at the current time step for position x_i.
- u_i^{n-1} is the value from the previous time step for position x_i.
- $C^2 = \left(\frac{c\Delta t}{\Delta x}\right)^2$ is the square of the Courant number, controlling stability in the scheme.
- u_{i-1}^n and u_{i+1}^n are values at neighboring spatial points, helping compute the spatial derivative.

Explanation of the Update Rule:

- This update rule uses the values from the current and previous time steps, along with neighboring spatial points, to compute u_i^{n+1}.
- The Courant number C is essential for the stability of the simulation. If C exceeds a certain threshold, the solution can become unstable, causing errors to accumulate.
- This formula is derived from a central difference approximation of the spatial and temporal derivatives in the wave equation.

2.3 Visualization of the Stepping Process

To illustrate how this update rule advances the solution in time, consider the following grid representation:

	u_i^{n-1}	
u_{i-1}^n	u_i^n	u_{i+1}^n
	u_i^{n+1}	

In this visualization:

- The solver uses values from two previous time steps (u_i^n and u_i^{n-1}) and neighboring spatial points (u_{i-1}^n and u_{i+1}^n) to calculate the value at u_i^{n+1}.
- This process continues iteratively for each spatial point, advancing the wave through time.

2.4 Calculating the First Time Step

To begin the finite difference solution process for the wave equation, we need to initialize the values of $u(x,t)$ at $t = 0$ and calculate the values at the first time step, $t = \Delta t$. The initial values and the values at the first time step are essential because the finite difference scheme relies on values from two preceding time steps to advance the solution.

Initial Condition: Assume that we have an initial displacement profile, $u(x,0)$, representing the state of the wave across the spatial domain at $t = 0$. We might also have an initial velocity profile, $u_t(x,0)$, which gives the rate of change of $u(x,t)$ at $t = 0$. These initial profiles are given by:

$$u(x,0) = f(x), \tag{12}$$

$$u_t(x,0) = g(x), \tag{13}$$

where $f(x)$ is the initial displacement function, and $g(x)$ is the initial velocity function.

To compute u_i^1, the displacement at each spatial point x_i at the first time step $t = \Delta t$, we use the Taylor expansion of $u(x,t)$ around $t = 0$. For small Δt, the displacement can be approximated as:

$$u_i^1 = u_i^0 + \Delta t\, g(x_i) + \frac{C^2}{2}\left(u_{i-1}^0 - 2u_i^0 + u_{i+1}^0\right), \tag{14}$$

where:

- $u_i^0 = u(x_i, 0)$ is the initial displacement at x_i,
- $g(x_i)$ is the initial velocity at x_i,
- C is the Courant number, defined as $C = \frac{c\Delta t}{\Delta x}$.

This expression provides the values of $u(x, \Delta t)$ across the spatial domain, setting up the solution to advance with the finite difference scheme. By combining the initial displacement and velocity profiles with the spatial finite difference term, this formula respects both the initial wave configuration and propagation behavior from the start.

3 Development of Generalized Solver for 1D Wave Equations in Python

The function `solve_wave_equation` is a Python implementation of a finite difference solver for the 1D wave equation, given by:

$$u_{tt} = c^2 u_{xx} + f(x,t), \tag{15}$$

3.1 Function Definition: `solve_wave_equation`

The function `solve_wave_equation` implements a finite difference solver for the 1D wave equation. The function signature is as follows:

```
def solve_wave_equation(I, V, f, c, L, dt, C, T, user_action=None):
```

Listing 1: Function Definition

This function takes the following parameters:

- `I(x)`: Initial displacement function, representing $u(x,0)$.
- `V(x)`: Initial velocity function, representing $u_t(x,0)$.
- `f(x, t)`: Source term in the wave equation, representing external forces as a function of space and time.
- `c`: Wave propagation speed.
- `L`: Length of the spatial domain.
- `dt`: Time step size.
- `C`: Courant number, defined as $C = \frac{c\Delta t}{\Delta x}$, which is critical for ensuring the stability of the numerical solution.
- `T`: Total simulation time.
- `user_action`: Optional callback function for custom actions at each time step, such as visualization or data collection.

The function returns the following outputs:

```
return u, x, t, cpu_time
```

Listing 2: Function Return

- `u`: Solution array containing the wave displacement values at the final time step.
- `x` and `t`: Arrays representing the spatial and temporal grid points, respectively.
- `cpu_time`: Total computation time for the simulation, useful for performance evaluation.

3.2 Discretization of Time and Space

The first step in setting up the finite difference solver is discretizing the time and spatial domains.

```
Nt = int(round(T / dt)) # Number of time steps
t = np.linspace(0, Nt * dt, Nt + 1) # Time mesh points
dx = dt * c / float(C) # Spatial step size based on Courant number
Nx = int(round(L / dx)) # Number of spatial points
x = np.linspace(0, L, Nx + 1) # Spatial mesh points
C2 = C ** 2 # Courant number squared (for convenience in the scheme)
```

Listing 3: Discretization of Time and Space

Here:

- `Nt` is the total number of time steps, computed as $\texttt{Nt} = \frac{T}{\Delta t}$.
- `Nx` is the number of spatial points, calculated using $\texttt{dx} = \frac{c\,\Delta t}{C}$ and $\texttt{Nx} = \frac{L}{\Delta x}$.
- The Courant number squared, `C2`, is precomputed as $C^2 = \left(\frac{c\Delta t}{\Delta x}\right)^2$ to simplify expressions in the finite difference scheme.

3.3 Initial Condition at t = 0

The initial displacement profile $u(x, 0) = I(x)$ is applied to initialize the array `u_1`, which represents the displacement at the first time level.

```
for i in range(Nx + 1):
    u_1[i] = I(x[i])
```

Listing 4: Applying Initial Condition at t

3.4 Calculating the First Time Step

For the first time step, $t = \Delta t$, a special formula based on the Taylor expansion of $u(x, t)$ around $t = 0$ is used. This initialization accounts for both the initial velocity $u_t(x, 0) = V(x)$ and the effect of the source term $f(x, t)$.

The displacement $u(x, \Delta t)$ for each spatial point x_i at the first time step can be computed as:

$$u_i^1 = u_i^0 + \Delta t\, V(x_i) + 0.5\, C^2 \left(u_{i-1}^0 - 2u_i^0 + u_{i+1}^0\right) + 0.5\, \Delta t^2\, f(x_i, 0), \tag{16}$$

where:

- $u_i^0 = u(x_i, 0)$ is the initial displacement at spatial point x_i.
- $V(x_i)$ is the initial velocity at x_i.
- C is the Courant number, $C = \frac{c\,\Delta t}{\Delta x}$.
- $f(x_i, 0)$ is the source term evaluated at $t = 0$ and spatial point x_i.

In the code, this calculation is implemented as follows:

```
                    for i in range(1, Nx):
                        u[i] = u_1[i] + dt * V(x[i]) + \
                        0.5 * C2 * (u_1[i - 1] - 2 * u_1[i] + u_1[i + 1]) + \
                        0.5 * dt ** 2 * f(x[i], t[0])
```

Listing 5: First Time Step Calculation

The boundary conditions are then applied at the edges of the spatial domain, setting $u_0 = 0$ and $u_N = 0$:

```
                    u[0] = 0 # Boundary condition at the left end
                    u[Nx] = 0 # Boundary condition at the right end
```

Listing 6: Boundary Conditions at First Time Step

These initial calculations set up the values at $t = 0$ and $t = \Delta t$, providing the necessary starting conditions for the finite difference scheme to continue calculating $u(x, t)$ at subsequent time steps.

3.5 Time-Stepping Loop for the Finite Difference Scheme

Once the initial conditions and the first time step are calculated, the solver proceeds with a time-stepping loop to advance the solution through each time level. This loop iterates from $n = 1$ to $n =$ `Nt`, where `Nt` is the total number of time steps. At each time step, the wave displacement $u(x, t)$ is updated according to the finite difference scheme.

```
                    # Time-stepping loop for n = 1 to Nt
                    for n in range(1, Nt):
                        # Update all inner points at time t[n+1]
                        for i in range(1, Nx):
                            u[i] = -u_2[i] + 2 * u_1[i] + \
                            C2 * (u_1[i - 1] - 2 * u_1[i] + u_1[i + 1]) + \
                            dt ** 2 * f(x[i], t[n])
```

Listing 7: Time-Stepping Loop

The update formula applied here is derived from the finite difference approximation of the wave equation. Specifically, the displacement u_i^{n+1} at each interior spatial point x_i is computed as:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2 \left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right) + \Delta t^2 f(x_i, t_n), \tag{17}$$

where:

- $-u_i^{n-1} + 2u_i^n$ represents the time-stepping component, incorporating values from two previous time steps.
- $C^2 \left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right)$ approximates the spatial second derivative.
- $\Delta t^2 f(x_i, t_n)$ accounts for the source term's influence at position x_i and time t_n.

3.5.1 Applying Boundary Conditions at Each Time Step

Boundary conditions are applied at each time step to maintain the required conditions at the edges of the spatial domain. In this case, we use Dirichlet boundary conditions by setting the displacements at the boundaries to zero:

```
u[0] = 0 # Boundary condition at the left end
u[Nx] = 0 # Boundary condition at the right end
```

Listing 8: Applying Boundary Conditions

This enforces that $u(0,t) = 0$ and $u(L,t) = 0$ for all t, which is consistent with fixed endpoints.

3.5.2 User Action for Visualization or Data Processing

The `user_action` parameter in `solve_wave_equation` is an optional argument that allows users to pass a custom callback function to be executed at each time step. This capability is useful for purposes such as:

- **Visualization**: Plotting the wave's progression in real time or saving snapshots of the solution at each time step.
- **Data Collection**: Storing or processing wave data at specific time steps for later analysis.
- **Monitoring**: Checking certain conditions during the simulation to potentially stop it early if a condition is met (e.g., a maximum amplitude threshold).

The `user_action` function, if provided, is called at each time step in the main time-stepping loop of the solver. Below is the code that implements the `user_action` callback:

```
if user_action is not None:
    if user_action(u, x, t, n + 1):
        break
```

Listing 9: User Action Callback

How the User Action Works 1. **Checking if `user_action` is Provided**: The code first checks if the `user_action` parameter is not `None`, meaning the user has provided a callback function to be executed. If `user_action` is `None`, the code skips this section entirely.

2. **Calling the User Action Function**: If `user_action` is provided, it is called with the following arguments:

- `u`: The current solution array, representing the displacement values at all spatial points at the current time step.
- `x`: The spatial grid array, which contains the spatial coordinates of each point.
- `t`: The time grid array, representing all time points in the simulation.
- `n + 1`: The index of the current time step, which is incremented by 1 to reflect the current time level in the simulation.

This function call gives the user access to the wave's current state at each time step, allowing them to perform custom operations, such as plotting the solution at each time step or saving data for later analysis.

3. **Conditional Early Stopping**: The `user_action` function may return a boolean value (e.g., `True` or `False`). If the function returns `True`, the simulation's main time-stepping loop will break early. This capability is useful for cases where the simulation should be stopped based on certain criteria, such as reaching a desired wave amplitude or detecting an anomaly in the solution.

Example Usage of `user_action` Here is a simple example of how the `user_action` function could be used for visualization:

```
import matplotlib.pyplot as plt

def visualize(u, x, t, n):
    plt.plot(x, u, label=f'Time step {n}')
    plt.xlabel('Position')
    plt.ylabel('Displacement')
    plt.title(f'Wave displacement at t = {t[n]:.2f}')
    plt.legend()
    plt.show()
    return False # Continue simulation
```

Listing 10: Example of `user_action` for Visualization

In this example, the `visualize` function plots the wave displacement at each time step. The function does not return `True`, so the simulation will continue to the next time step without breaking the loop.

Alternatively, the `user_action` function could be set up to collect data only at specific time intervals, stop the simulation when a certain condition is met, or save snapshots of the wave progression:

```
def stop_if_high_amplitude(u, x, t, n):
    max_amplitude = np.max(np.abs(u))
    if max_amplitude > threshold:
        print(f'Stopping simulation at time step {n}, max amplitude
            reached: {max_amplitude}')
        return True # Stop the simulation
    return False # Continue simulation
```

Listing 11: Example of Early Stopping with `user_action`

In this case, the simulation will stop early if the wave amplitude exceeds a given `threshold`.

Summary of `user_action` Benefits The `user_action` callback provides significant flexibility in the `solve_wave_equation` function. It enables users to:

- Visualize wave evolution in real time.
- Collect or process data during the simulation.
- Implement custom stopping criteria based on simulation progress.

This flexibility makes the solver adaptable to various applications, allowing users to gather insights during the simulation and modify its behavior without altering the core solver code.

Variable Switching for the Next Time Step After updating the solution array for the current time step, the variables are switched to prepare for the next step:

```
# Switch variables for the next time step
u_2[:] = u_1
u_1[:] = u
```

Listing 12: Switching Variables for the Next Time Step

Here:

- `u_2` (representing u^{n-1}) is updated to hold the values of `u_1` (representing u^n).
- `u_1` (representing u^n) is updated to hold the current values of `u` (representing u^{n+1}).

This process shifts the time levels forward, allowing the scheme to compute the next time step using the latest information.

3.5.3 CPU Time Calculation and Return

At the end of the simulation, the total CPU time taken is calculated and printed, then the function returns the final solution and grid points.

```
                    cpu_time = time.time() - t0 # Calculate CPU time
                    print(cpu_time)

                    return u, x, t, cpu_time
```

Listing 13: CPU Time Calculation and Return

Output:

- `u`: Solution array containing the wave displacement values at the final time step.
- `x` and `t`: Spatial and time grid points used in the simulation.
- `cpu_time`: Total CPU time required to complete the computation.

This complete time-stepping loop iteratively applies the finite difference scheme for each time step, calculates values at all spatial points, and updates the boundary conditions, resulting in a stable and accurate numerical solution to the wave equation.

3.6 Vectorization

In numerical computing, vectorization is a technique used to improve performance by replacing explicit loops with array operations that leverage optimized, low-level implementations. The goal of vectorization is to speed up computation by taking advantage of the highly efficient, parallel processing capabilities of modern CPUs and specialized libraries (such as `NumPy` in Python).

In the context of the `solve_wave_equation` function, we provide three versions of the solver:

- **Scalar (Loop-Based) Version:** Each element in the solution array is updated individually in a loop, allowing for straightforward and readable code but without the computational efficiency of vectorized operations.
- **Vectorized Version:** Array slicing is used to update the solution array in bulk. This removes the need for explicit loops, significantly improving the speed of computations for larger arrays.
- **Alternative Vectorized Version (Vectorized2):** A variation of the vectorized approach, this implementation uses a different array slicing method but achieves similar results.

3.6.1 Function Modifications to Support Multiple Versions

To allow the user to select the solver version, we introduce a `version` parameter in the function's input, which specifies whether the `'scalar'`, `'vectorized'`, or `'vectorized2'` implementation should be used. The updated function signature is as follows:

```
def solve_wave_equation(I, V, f, c, L, dt, C, T, user_action=None, version='
    scalar', save_dir=None):
```

Listing 14: Function Signature with Solver Version Parameter

The `version` parameter is passed as a string and defaults to `'scalar'`. Based on the value of `version`, the function executes the corresponding time-stepping loop.

3.6.2 Time-Stepping Loop: Scalar vs. Vectorized Implementations

The time-stepping loop in `solve_wave_equation` is responsible for advancing the solution at each time step, updating the wave displacement array `u` based on values from previous time levels. We differentiate between scalar and vectorized implementations as follows:

1. Scalar (Loop-Based) Implementation The scalar version uses a loop to update each element in `u` individually. This approach is straightforward and maintains clarity, with each element $u[i]$ computed as:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2\left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right) + \Delta t^2 f(x_i, t_n), \tag{18}$$

where:

- $-u_i^{n-1} + 2u_i^n$ represents the time-stepping update using values from the two previous time levels.
- $C^2\left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right)$ is the spatial second derivative in finite difference form.
- $\Delta t^2 f(x_i, t_n)$ accounts for the source term at position x_i and time t_n.

The scalar version in code is implemented as follows:

```
# Time-stepping loop
if version == 'scalar':
        for i in range(1, Nx):
                u[i] = -u_2[i] + 2 * u_1[i] + \
                            C2 * (u_1[i - 1] - 2 * u_1[i] + u_1[i + 1]) + \
                            dt ** 2 * f(x[i], t[n])
```

Listing 15: Scalar Loop Implementation

2. Vectorized Implementation The vectorized version uses array slicing to update multiple elements simultaneously, improving computational efficiency by avoiding explicit loops. This approach calculates the solution array `u` in a single line for all interior points using `NumPy` operations. In the vectorized implementation, the update for $u[1:-1]$ (representing all interior points) is:

$$u[1:-1] = -u_2[1:-1] + 2u_1[1:-1] + C^2(u_1[0:-2] - 2u_1[1:-1] + u_1[2:]) + \Delta t^2 f(x[1:-1], t_n), \tag{19}$$

where:

- $u_1[0:-2]$ corresponds to u^n_{i-1}, $u_1[1:-1]$ to u^n_i, and $u_1[2:]$ to u^n_{i+1}.
- `f(x, t[n])` is precomputed as an array `f_a` for all spatial points, enabling simultaneous computation across the array.

The vectorized version in code is as follows:

```
elif version == 'vectorized':
    f_a = f(x, t[n]) # Precompute source term as an array
    u[1:-1] = -u_2[1:-1] + 2 * u_1[1:-1] + \
              C2 * (u_1[0:-2] - 2 * u_1[1:-1] + u_1[2:]) + \
              dt ** 2 * f_a[1:-1]
```

Listing 16: Vectorized Implementation

3. Alternative Vectorized Implementation (Vectorized2) An alternative vectorized approach, `vectorized2`, uses a slightly different slicing syntax but achieves the same results. Instead of `u[1:-1]`, it uses `u[1:Nx]`, providing another way to handle the array indices. The formula remains the same, only the slicing syntax differs:

```
elif version == 'vectorized2':
    f_a = f(x, t[n]) # Precompute source term as an array
    u[1:Nx] = -u_2[1:Nx] + 2 * u_1[1:Nx] + \
              C2 * (u_1[0:Nx - 1] - 2 * u_1[1:Nx] + u_1[2:Nx + 1]) +
                  \
              dt ** 2 * f_a[1:Nx]
```

Listing 17: Alternative Vectorized Implementation

3.6.3 Boundary Conditions and Function Execution

In each version, boundary conditions are applied at each time step by setting $u[0] = 0$ and $u[\texttt{Nx}] = 0$, which ensures that the boundaries remain fixed.

After updating `u` for the current time step, we swap arrays so that `u_2` represents the previous time level, `u_1` represents the current level, and `u` is ready for the next iteration.

```
u[0] = 0
u[Nx] = 0

u_2[:] = u_1
u_1[:] = u
```

Listing 18: Boundary Conditions and Swapping Arrays

3.6.4 Benefits of Vectorization

Vectorization offers significant performance benefits, especially for large grids, by eliminating explicit loops and leveraging `NumPy`'s efficient array operations. For high-resolution simulations with many spatial points, vectorized versions can achieve substantial speed-ups over the scalar implementation, making them preferable for performance-critical applications.

3.7 Boundary Conditions

Boundary conditions are essential in solving the wave equation numerically, as they define how the wave behaves at the edges of the spatial domain, $x = 0$ and $x = L$. This code includes a flexible approach to boundary conditions, allowing the user to choose between Dirichlet, Neumann, and custom boundary conditions for each boundary. These conditions are specified via the `boundary` parameter in the `solve_wave_equation` function.

3.7.1 Function Modifications to Support Different Boundary Conditions

The `boundary` parameter is introduced in the function's input, allowing the user to select the type of boundary condition to apply:

```
def solve_wave_equation(I, V, f, c, L, dt, C, T, user_action=None, version='
    scalar', save_dir=None, boundary='Dirichlet'):
```

Listing 19: Function Signature with Boundary Condition Parameter

This parameter can take the following values:

- `'Dirichlet'`: Implements fixed boundary conditions.
- `'Neumann'`: Implements zero-gradient (free) boundary conditions.
- `'leftFree'`: Fixes the right boundary while applying a zero-gradient condition at the left.
- `'rightFree'`: Fixes the left boundary while applying a zero-gradient condition at the right.

Based on the chosen boundary type, the code applies the corresponding boundary conditions at the initial time step and updates them in each iteration of the time-stepping loop.

3.7.2 Implemented Boundary Conditions

1. Dirichlet Boundary Condition (Fixed Ends) The Dirichlet boundary condition sets the displacement $u(x, t)$ to zero at both ends of the domain, which models a wave on a string with fixed endpoints. Mathematically, this condition is expressed as:

$$u(0, t) = 0 \quad \text{and} \quad u(L, t) = 0.$$

In the code, this is implemented as follows:

```
if boundary == 'Dirichlet':
        u[0] = 0
        u[Nx] = 0
```

Listing 20: Dirichlet Boundary Condition

This condition is enforced at both the initial time step and every subsequent time step, ensuring that the wave remains fixed at the boundaries throughout the simulation.

2. Neumann Boundary Condition (Zero-Gradient/Free Ends) The Neumann boundary condition sets the gradient of $u(x,t)$ to zero at the boundaries, allowing the wave to be "free" at the endpoints. This is typically used to model waves in an open domain. Mathematically, this condition is:

$$\left.\frac{\partial u}{\partial x}\right|_{x=0} = 0 \quad \text{and} \quad \left.\frac{\partial u}{\partial x}\right|_{x=L} = 0.$$

In the discrete form, a zero-gradient condition at the boundaries is achieved by setting the values at the boundary points equal to their neighboring points:

```
                elif boundary == 'Neumann':
                    u[0] = u[1]      # Zero-gradient at left boundary
                    u[Nx] = u[Nx - 1] # Zero-gradient at right boundary
```

Listing 21: Neumann Boundary Condition

This approach effectively "mirrors" the boundary points, simulating a zero gradient at each end of the domain.

3. Mixed Boundary Condition (leftFree and rightFree) In addition to standard Dirichlet and Neumann conditions, the code also allows for mixed boundary conditions where one end is free (zero-gradient) while the other is fixed. These are useful for cases where one side of the domain represents a free edge while the other side is anchored.

- **leftFree**: Applies a zero-gradient condition on the left boundary and a fixed condition on the right boundary.

$$\left.\frac{\partial u}{\partial x}\right|_{x=0} = 0, \quad u(L,t) = 0. \tag{20}$$

  In the code, the `leftFree` condition is implemented by setting the left boundary equal to its neighboring point (zero-gradient), while fixing the right boundary to zero:

```
                elif boundary == 'leftFree':
                    u[0] = u[1]      # Zero-gradient at left boundary
                    u[Nx] = 0        # Fixed at right boundary
```

Listing 22: Left-Free Boundary Condition

- **rightFree**: Applies a fixed condition on the left boundary and a zero-gradient condition on the right boundary:

$$u(0,t) = 0, \quad \left.\frac{\partial u}{\partial x}\right|_{x=L} = 0. \tag{21}$$

  This boundary condition is implemented by fixing the left boundary to zero while setting the right boundary to equal its neighboring point, simulating a zero-gradient condition on the right edge:

```
                elif boundary == 'rightFree':
                    u[0] = 0         # Fixed at left boundary
                    u[Nx] = u[Nx - 1] # Zero-gradient at right boundary
```

Listing 23: Right-Free Boundary Condition

3.7.3 Applying Boundary Conditions During the Time-Stepping Loop

Each boundary condition is enforced at every time step to maintain the desired constraints throughout the simulation. The code applies the selected boundary conditions as follows:

```
            # Apply boundary conditions at each time step
            if boundary == 'Dirichlet':
                u[0] = 0
                u[Nx] = 0
            elif boundary == 'Neumann':
                u[0] = u[1]      # Zero-gradient at left boundary
                u[Nx] = u[Nx - 1] # Zero-gradient at right boundary
            elif boundary == 'leftFree':
                u[0] = u[1]      # Zero-gradient at left boundary
                u[Nx] = 0        # Fixed at right boundary
            elif boundary == 'rightFree':
                u[0] = 0         # Fixed at left boundary
                u[Nx] = u[Nx - 1] # Zero-gradient at right boundary
```

Listing 24: Applying Boundary Conditions at Each Time Step

3.7.4 Summary of Boundary Conditions

- **Dirichlet**: Fixes both boundaries, ensuring that the displacement is zero at $x = 0$ and $x = L$.
- **Neumann**: Enforces zero-gradient conditions on both boundaries, allowing for "free" boundaries.
- **leftFree**: Allows a free boundary on the left and a fixed boundary on the right.
- **rightFree**: Fixes the left boundary while allowing a free boundary on the right.

4 Development of Visualization Code for Post-Processing Purposes

Visualization is a crucial aspect of understanding and validating wave simulations, as it allows researchers and engineers to analyze wave propagation, reflections, and boundary interactions more intuitively. Effective visualization also aids in validating the accuracy of the numerical methods used, helping identify issues like numerical dispersion, boundary reflections, and stability problems.

The following visualization functions are designed to facilitate post-processing in the `solve_wave_equation` solver, enabling users to create both static and dynamic representations of the wave evolution over time. By capturing and analyzing visual outputs, users can assess how well the simulation aligns with theoretical expectations, providing valuable insights into the reliability of the solver under different conditions.

In this section, we introduce three key functions for visualization:

- `save_wave_image`: A function that saves static images of the wave at specified time steps. These images allow users to analyze the wave at individual points in time or assemble them into an animation.
- `generate_gif_from_images`: A function that compiles saved images into an animated GIF, providing a continuous view of the wave's progression across the spatial domain.
- `generate_html_animation`: A function that generates an interactive HTML animation of the wave, allowing users to view and manipulate the wave's progression in a browser.

These functions collectively enable users to capture, animate, and interact with the results of the wave simulation, enhancing interpretability of the data and supporting in-depth post-processing analyses to validate the accuracy and stability of the numerical solution.

4.1 Function `save_wave_image`

The `save_wave_image` function is designed to save a static image of the wave profile at a specified time step during the simulation. This function enables users to capture snapshots of the wave's behavior over time, which can be useful for visual inspection, validation, and for assembling into an animation.

Function Signature :

```
1                                   def save_wave_image(u, x, t, n, C, save_dir='wave_images', ymin=None,
                                        ymax=None):
```

Listing 25: Function Definition

Parameters:

- `u`: The solution array representing the wave displacement values at each spatial point at the current time step.
- `x`: The array of spatial points, providing the x-coordinates for plotting.

- `t`: The array of time points in the simulation, used to label the time at the current step.
- `n`: The index of the current time step, allowing access to both the current displacement values and time.
- `C`: The Courant number, which is displayed in the title to provide information on the stability condition of the simulation.
- `save_dir`: The directory path where images will be saved. Default is `'wave_images'`.
- `ymin, ymax`: Optional parameters specifying the limits for the y-axis. If these are not provided, the limits will be set dynamically based on the data.

Code Explanation The function proceeds through the following steps:

1. Directory Creation :

```
if not os.path.exists(save_dir):
    os.makedirs(save_dir)
```

This section checks if the specified directory for saving images, `save_dir`, exists. If it does not, the function creates it using `os.makedirs(save_dir)`. This ensures that images are stored in an organized manner without requiring the user to create the directory manually.

2. Setting Up the Plot :

```
plt.figure(figsize=(8, 4))
plt.plot(x, u, label=f"t = {t[n]:.5f}")
```

Here, a new figure is created with a fixed size of 8×4 inches to maintain consistency across images. The function then plots the wave profile, using `x` as the horizontal axis (spatial points) and `u` as the vertical axis (wave displacement). The label `t = t[n]:.5f` displays the current time with five decimal precision, which is useful for identifying the time step when analyzing the images.

3. Setting the Y-Axis Limits :

```
if ymin is not None and ymax is not None:
    plt.ylim(ymin, ymax)
else:
    plt.ylim(min(u), max(u)) # Dynamic ylim if no limits are provided
```

The y-axis limits can either be set by the user or dynamically adjusted based on the wave displacement values. If `ymin` and `ymax` are provided, they set fixed limits, making it easier to compare images. If they are not provided, the limits are set dynamically to fit the range of `u` at the current time step. This dynamic setting can be helpful when the wave amplitude varies significantly over time.

4. Setting the X-Axis Limits :

```
plt.xlim(0, max(x))
plt.xlabel('x')
plt.ylabel('u(x,t)')
```

The x-axis limits are set from 0 to the maximum value of `x`, covering the entire spatial domain. The x-axis is labeled `x`, representing the spatial variable, and the y-axis is labeled `u(x,t)`, representing wave displacement.

5. Adding the Title with Courant Number :

```
plt.title(f"Wave propagation at time t = {t[n]:.5f}, Courant number = {C
    }")
```

The plot title displays both the current time `t[n]` and the Courant number `C`. Showing the Courant number in the title provides information on the stability condition and numerical parameters, which is useful for documentation and analysis.

6. Enabling Grid and Adding Legend :

```
plt.legend()
plt.grid(True)
```

A grid is added to the plot for visual clarity, and a legend is enabled to label the time step, helping users distinguish between different time steps if multiple plots are displayed or overlaid.

7. Defining the Filename :

```
filename = os.path.join(save_dir, f'wave_step_{n:04d}.png')
```

The `filename` is defined to save the image in `save_dir`. The naming convention `wave_step_{n:04d}.png` ensures that files are sorted in chronological order, with zero-padded indices (e.g., `wave_step_0001.png`) for consistent file naming.

8. Debug Message for Confirmation :

```
print(f"Saving image: {filename}")
```

A print statement provides a debug message showing the filename of the saved image. This message is useful for tracking progress in long simulations or debugging if files are not saved correctly.

9. Saving and Closing the Plot :

```
plt.savefig(filename)
plt.close()
```

The plot is saved to the specified filename in `save_dir`, and `plt.close()` ensures that the plot is cleared from memory. Closing the plot is especially important in long simulations to prevent excessive memory usage from open figures.

Summary The `save_wave_image` function provides a systematic way to capture snapshots of the wave profile at each time step, saving them as PNG images in a specified directory. These images are useful for post-processing, enabling visual inspection of the wave's evolution or for assembling into animations to observe wave propagation across time.

4.2 Function `generate_gif_from_images`

The `generate_gif_from_images` function compiles a sequence of saved wave images into an animated GIF. This enables users to view the wave's progression continuously across time steps, providing an intuitive representation of wave propagation dynamics in a single, easily shareable file.

Function Signature :

```
def generate_gif_from_images(image_folder='wave_images', gif_name='
    wave_animation.gif', duration=0.1):
```

Listing 26: Function Definition

Parameters:

- `image_folder`: The directory containing the saved wave images (default is `'wave_images'`). Each image should correspond to a specific time step in the wave simulation.
- `gif_name`: The name of the output GIF file (default is `'wave_animation.gif'`).
- `duration`: The display duration of each frame in the GIF, in seconds (default is 0.1 seconds). This value controls the animation speed.

Code Explanation The function consists of the following main steps:

1. Initializing the List of Images :

```
images = []
```

The function begins by creating an empty list, `images`, which will hold the image data for each frame to be included in the GIF.

2. Retrieving and Sorting Image Files :

```
image_files = sorted([img for img in os.listdir(image_folder) if img.
    endswith(".png")])
```

The function retrieves a list of all files in `image_folder` that have the `.png` extension, which corresponds to the saved wave images. It sorts the filenames in chronological order to maintain the correct sequence in the animation. Sorting by filename ensures that frames are displayed in the correct order, assuming filenames follow a numerical or chronological naming convention.

3. Handling the Case of No Images Found :

```
if not image_files:
    print("Warning: No images found in the specified folder to create
        a GIF. Ensure that images are saved correctly during the
        wave equation simulation.")
return
```

If `image_files` is empty, indicating no images were found in the specified folder, the function prints a warning message and exits without creating a GIF. This check prevents errors if the function is called before images are generated, providing a clear message for troubleshooting.

4. Reading and Appending Images :

```
for filename in image_files:
    image_path = os.path.join(image_folder, filename)
    images.append(imageio.imread(image_path))
```

The function iterates over each filename in `image_files`, reads the image file using `imageio.imread`, and appends the image data to the `images` list. This approach ensures that all images are stored in memory as frames, ready for GIF creation.

5. Defining the Output GIF Path :

```
gif_path = os.path.join(image_folder, gif_name)
```

The output path for the GIF is set by combining the `image_folder` and `gif_name`. This keeps the GIF within the same folder as the images, making it easy to locate.

6. Saving the GIF :

```
imageio.mimsave(gif_path, images, duration=duration)
```

The `imageio.mimsave` function is used to create the GIF. It takes the list of images and the specified `duration` as input, saving the GIF file at the path specified by `gif_path`. The `duration` parameter controls the time each frame is displayed, allowing users to adjust the playback speed of the GIF to their preference.

7. Confirmation Message :

```
print(f"GIF saved as {gif_path}")
```

A confirmation message is printed to the console, showing the path where the GIF was saved. This message confirms successful completion and provides the user with the location of the output file.

Summary :

The `generate_gif_from_images` function automates the creation of a GIF from individual frames of wave simulation results, allowing users to observe wave propagation over time in a single file. This functionality is especially useful for presentations and analysis, as it encapsulates the temporal evolution of the wave in a convenient, visual format.

4.3 Function `generate_html_animation`

The `generate_html_animation` function creates an HTML-based animation that visually represents the wave's evolution over time. This type of interactive output is highly useful for dynamic visualization, allowing users to view the entire wave progression within a web browser.

Function Signature :

```
def generate_html_animation(x, results, save_dir, filename="movie.html",
      ymin=-0.005, ymax=0.005, fps=10):
```

Listing 27: Function Definition

Parameters:

- `x`: The array of spatial grid points, providing the x-coordinates for plotting.
- `results`: A list of tuples `(u, t)`, where each tuple contains the wave displacement array `u` and the time `t` at each time step. This list holds the full time-series data for the animation.
- `save_dir`: The directory where the HTML animation file will be saved.
- `filename`: The name of the output HTML file (default is `"movie.html"`).
- `ymin, ymax`: Optional parameters specifying the limits for the y-axis of the animation plot, controlling the vertical bounds for visual consistency.
- `fps`: Frames per second for the animation (default is 10), controlling the playback speed.

Code Explanation The function can be broken down into the following primary steps:

1. Setting Up the Plot :

```
fig, ax = plt.subplots()
ax.set_xlim(0, x[-1]) # Set x-axis based on spatial grid
ax.set_ylim(ymin, ymax) # Set y-axis based on provided limits
line, = ax.plot([], [], lw=2, color="blue")
```

The code initializes a new figure and axis using `plt.subplots()`. The x-axis limits are set from 0 to the maximum value in `x`, ensuring the plot covers the entire spatial domain. The y-axis limits are set according to `ymin` and `ymax` to maintain consistent bounds across frames. The wave data will be plotted as a blue line with a specified line width (`lw=2`), which is initially empty (no data) as the plot will be updated by the animation.

2. Initializing the Line Plot with `init` Function :

```
def init():
      line.set_data([], [])
      return line,
```

The `init` function sets the initial state of the line plot to be empty by calling `line.set_data([], [])`. This is necessary for the animation framework, as it prepares the plot for each frame update. Returning `line,` ensures compatibility with `FuncAnimation`, indicating which graphical elements will be updated.

3. Defining the `animate` Function to Update Frames :

```
def animate(n):
    u, current_time = results[n]
    line.set_data(x, u)
    ax.set_title(f"Wave Solution at t = {current_time:.3f}")
    return line,
```

The `animate` function is called for each frame in the animation. At each time step, it retrieves the displacement data `u` and time `current_time` from `results` at index `n`. The line plot is updated by setting its data to the current displacement `u` and spatial points `x`. The title is also updated to reflect the current time, providing context for each frame. Returning `line,` ensures that the plot is redrawn with the updated data for each frame.

4. Creating the Animation with `FuncAnimation` :

```
ani = FuncAnimation(fig, animate, init_func=init, frames=len(results),
    blit=True)
```

The `FuncAnimation` function from `matplotlib.animation` is used to create the animation. It takes the following arguments:

- `fig`: The figure object to animate.
- `animate`: The function that updates the plot data for each frame.
- `init_func=init`: The initialization function that prepares the line plot.
- `frames=len(results)`: The total number of frames, based on the length of `results`.
- `blit=True`: Improves performance by only redrawing parts that have changed.

This setup allows the wave solution to be animated smoothly over the time steps stored in `results`.

5. Defining the Output Path and Saving the Animation as HTML :

```
html_path = os.path.join(save_dir, filename)
ani.save(html_path, writer=HTMLWriter())
```

The output path for the HTML animation file is defined by combining `save_dir` and `filename`. The `ani.save` function is called to save the animation as an HTML file using the `HTMLWriter`. This generates a standalone HTML file that can be opened in any web browser to view the animation interactively.

6. Confirmation Message :

```
print(f"HTML animation saved as {html_path}")
```

A print statement confirms that the HTML animation has been successfully saved, displaying the path for easy reference. This message helps users verify that the animation file was created as expected.

Summary The `generate_html_animation` function creates a web-based animation of the wave's time evolution, saving it as an HTML file. This provides a highly accessible and interactive way to visualize the wave simulation in a browser, enabling users to examine the behavior of the wave over time with ease. This visualization is particularly useful for presentations, educational purposes, and in-depth analysis of wave behavior across the simulation.

5 Verification

Verification of the wave equation solver is essential to ensure that the numerical methods accurately capture the theoretical behavior of the wave. This section presents a verification test using an exact solution, which is compared against the numerical solution to validate the solver's correctness across different implementations (scalar, vectorized, and vectorized2).

5.1 Function `test_quadratic`

The `test_quadratic` function is a verification test for the wave equation solver. It tests the solver's accuracy by comparing its output to a known exact solution:

$$u(x,t) = x(L-x)(1+0.5\,t),$$

where L is the length of the spatial domain. This function evaluates the solver across different implementations, asserting that the numerical solution matches the exact solution within a specified tolerance.

5.1.1 Code Explanation

The function `test_quadratic` is structured as follows:

```
def test_quadratic():
```

Listing 28: Function Definition

Parameters and Exact Solution Setup :

```
# Parameters for the test
L = 2.5
c = 1.5
C = 0.75
Nx = 6 # Very coarse mesh for this exact test
dt = C * (L / Nx) / c
T = 18

# Exact solution for u(x, t) = x(L - x)(1 + 0.5*t)
u_exact = lambda x, t: x * (L - x) * (1 + 0.5 * t)
```

Here, we define the parameters for the test:

- $L = 2.5$: Length of the spatial domain.
- $c = 1.5$: Wave propagation speed.
- $C = 0.75$: Courant number, ensuring stability of the numerical scheme.
- $Nx = 6$: Number of spatial grid points (a coarse mesh is used to keep calculations simple).
- $dt = \frac{C \cdot L}{Nx \cdot c}$: Time step, computed to satisfy the Courant condition.
- $T = 18$: Total simulation time.

The exact solution $u(x,t) = x(L-x)(1+0.5\,t)$ is defined as a lambda function, allowing us to evaluate the exact displacement at any point x and time t for comparison with the numerical solution.

Initial and Boundary Conditions :

```
I = lambda x: u_exact(x, 0) # Initial displacement
V = lambda x: 0.5 * u_exact(x, 0) # Initial velocity
f = lambda x, t: np.zeros_like(x) + 2 * c ** 2 * (1 + 0.5 * t) # Source
    term
```

These definitions set up the initial and boundary conditions:

- `I(x)`: Initial displacement, derived from the exact solution $u(x,0)$.
- `V(x)`: Initial velocity, set to half of the initial displacement, $0.5\,u(x,0)$.
- `f(x, t)`: Source term, defined as $f(x,t) = 2c^2(1+0.5\,t)$, to ensure that the exact solution satisfies the wave equation with this source.

Mathematical Verification Condition The exact solution $u(x,t)$ is derived by substituting it into the wave equation:

$$u_{tt} = c^2 u_{xx} + f(x,t).$$

The source term $f(x,t) = 2c^2(1+0.5\,t)$ guarantees that the exact solution satisfies the wave equation, making it an appropriate reference for validating the numerical solution.

Accuracy Assertion Function The `assert_no_error` function checks the accuracy of the numerical solution by comparing it to the exact solution:

```
def assert_no_error(u, x, t, n):
    u_e = u_exact(x, t[n]) # Exact solution at time step n
    tol = 1E-13 # Tolerance for error comparison
    diff = np.abs(u - u_e).max() # Max difference between computed
        and exact
    print(diff)

    assert diff < tol, f"Difference {diff} exceeds tolerance at step
        {n}"
```

The `assert_no_error` function:

- Computes the exact solution $u_e = u(x, t[n])$ at each time step n.
- Sets a tolerance `tol` $= 1 \times 10^{-13}$ to define the acceptable error limit.
- Calculates the maximum difference `diff` between the exact and computed solutions at each time step. If `diff` exceeds `tol`, an assertion error is raised, signaling that the numerical solution deviates too much from the exact solution.

Running the Test on Different Solver Versions The test is run on all versions of the solver (scalar, vectorized, and vectorized2) to confirm that each implementation produces consistent and accurate results:

```
for version in ['scalar', 'vectorized', 'vectorized2']:
    print(f"Testing {version} solver...")

    try:
        # Run the solver with the assertion check as the user action
        solve_wave_equation(
            I=I,
            V=V,
            f=f,
            c=c,
            L=L,
            dt=dt,
            C=C,
            T=T,
            user_action=assert_no_error,
            version=version
            )
        print(f"{version} solver: Pass")
    except AssertionError as e:
        print(f"{version} solver: Fail - {e}")

print("All tests completed.")
```

The loop iterates over the available versions and calls the `solve_wave_equation` function with the `assert_no_error` function as the `user_action` callback. This setup allows the test to verify each solver implementation, outputting any deviations from the expected solution.

5.1.2 Test Results

```
Testing scalar solver...
CPU time (s) of scalar solver: 0.00199
scalar solver: Pass
Testing vectorized solver...
CPU time (s) of vectorized solver: 0.00103
vectorized solver: Pass
Testing vectorized2 solver...
CPU time (s) of vectorized2 solver: 0.00089
vectorized2 solver: Pass
All tests completed.
```

5.1.3 Summary of Verification Test

The `test_quadratic` function serves as a robust verification test for the wave equation solver, ensuring that all solver versions accurately reproduce the exact solution $u(x,t) = x(L-x)(1+0.5\,t)$ within a tight tolerance. By comparing each numerical solution to this known solution, the test confirms the correctness of the solver under ideal conditions. The success message indicates that

all versions produce consistent, accurate results, verifying the reliability of the solver's implementations.

5.2 Function `convergence_rates`

The `convergence_rates` function estimates the convergence rates of the wave equation solver. Convergence rates are essential for assessing the accuracy of a numerical method as the spatial and temporal discretization are refined. This function gradually reduces the time step by a factor of two over a specified number of mesh refinements and computes the error and corresponding convergence rate at each level.

5.2.1 Code Explanation

The function `convergence_rates` is structured as follows:

```
def convergence_rates(u_exact, I, V, f, c, L, dt0, num_meshes, C, T,
    solver_choice='scalar'):
```

Listing 29: Function Definition

Parameters:

- `u_exact`: The exact solution function for $u(x, t)$.
- `I, V`: Functions for the initial displacement and initial velocity.
- `f`: Source term function.
- `c`: Wave propagation speed.
- `L`: Length of the spatial domain.
- `dt0`: Initial time step size.
- `num_meshes`: Number of times the mesh will be refined (by halving the time step).
- `C`: Courant number, controlling stability.
- `T`: Total simulation time.
- `solver_choice`: The version of the solver to use (e.g., `'scalar'`, `'vectorized'`, etc.).

Error Computation Function To track and compute the error, the nested function `compute_error` is defined:

```
def compute_error(u, x, t, n):
    global error
    if n == 0:
        error = 0
    else:
        # Calculate max error for the current time step
        current_error = np.abs(u - u_exact(x, t[n])).max()
        error = max(error, current_error)
```

The `compute_error` function calculates the maximum error between the computed solution u and the exact solution u_{exact} at each time step. The function performs the following operations:

1. **Initialization at the Start of Simulation**: If `n = 0`, the function sets the global variable `error` to zero, initializing it for the current simulation. This ensures that any residual error from previous mesh levels is cleared.

2. **Computing the Maximum Absolute Error**: For each subsequent time step (`n` ¿ 0), the function calculates the error as the maximum absolute difference between the computed and exact solutions:

$$\text{current\_error} = \max\left(|u(x,t) - u_{\text{exact}}(x,t)|\right)$$

Here:

- $u(x,t)$: The numerical solution at spatial points x and time t.
- $u_{\text{exact}}(x,t)$: The exact analytical solution at spatial points x and time t.
- $\max\left(|u(x,t) - u_{\text{exact}}(x,t)|\right)$: The maximum absolute error at the current time step.

This computation captures the largest deviation between the computed and exact solutions across the spatial domain at each time step, ensuring that any local discrepancies are detected.

3. **Updating the Cumulative Maximum Error**: The `compute_error` function then updates the cumulative maximum error:

$$\text{error} = \max(\text{error}, \text{current\_error})$$

This operation ensures that `error` stores the largest observed error across all time steps for a given mesh refinement level. This cumulative error is crucial for convergence analysis, as it provides a measure of the worst-case deviation for each mesh level.

Purpose and Use of Global Variable `error`:

The use of `error` as a global variable allows it to be updated within `compute_error` and accessed outside the function scope, storing the final error for each mesh level. This setup is essential because it enables the main function to record and analyze error values across multiple mesh levels without resetting the error at each time step.

Setting Up Error and Step Size Lists To facilitate the calculation of convergence rates, the lists `E` and `h` are initialized to store the error and time step size values for each mesh level, as shown below:

```
# Initialize lists to store error and step size values
E = []
h = []
```

- `E`: Stores the cumulative maximum error for each mesh refinement level, as calculated by the `compute_error` function.
- `h`: Stores the time step size $h = \Delta t$ corresponding to each mesh level.

At each mesh refinement level, these lists are updated with:

$$E[i] = \max\left(|u(x,t) - u_{\text{exact}}(x,t)|\right),$$

$$h[i] = \Delta t,$$

where $E[i]$ represents the maximum observed error for the i-th refinement level, and $h[i]$ is the associated time step size. By storing these values across all mesh levels, we can analyze the convergence behavior of the solver by calculating the convergence rates, which quantify how quickly the error decreases as the time step size h is reduced.

These lists enable us to calculate convergence rates in subsequent steps and provide insights into the accuracy and stability of the numerical method across progressively finer meshes.

Mesh Refinement Loop The main loop runs the solver over a series of meshes with progressively smaller time steps:

```
# Starting time step and corresponding spatial step
dt = dt0
dx = dt * c / C # Based on Courant number and wave speed

for i in range(num_meshes):
    # Run the solver with the current resolution
    solve_wave_equation(I, V, f, c, L, dt, C, T, user_action=
        compute_error, version=solver_choice)

    # Store the computed error and step size
    E.append(error)
    h.append(dt)

    # Debugging output to trace errors and steps
    print(f"Mesh {i + 1}: dt = {dt:.5e}, error = {error:.5e}")

    # Halve the time step for the next iteration
    dt /= 2
    dx = dt * c / C
```

For each iteration, the function:

- Runs the solver with the current time step size, `dt`, and Courant number `C`, using `compute_error` as the `user_action` to calculate the error.
- Appends the maximum error (stored in `error`) and the time step size `dt` to lists `E` and `h`, respectively.
- Prints a debug message showing the current time step and error.
- Halves the time step `dt` for the next mesh refinement.

Convergence Rate Calculation After running the simulations for all mesh levels, the convergence rates are calculated as follows:

```
rates = []
for i in range(1, num_meshes):
    if E[i] > 0 and E[i - 1] > 0:
        rate = np.log(E[i] / E[i - 1]) / np.log(h[i] / h[i - 1])
    else:
        rate = float('inf') # Avoid division by zero
    rates.append(rate)
    print(f"Convergence rates {i + 1}: r = {rate:.5e}, error = {E[i
        ]:.5e}")
```

The convergence rate r for each successive mesh refinement level i is computed using the formula:

$$r = \frac{\log\left(\frac{E[i]}{E[i-1]}\right)}{\log\left(\frac{h[i]}{h[i-1]}\right)},$$

where:

- $E[i]$: The cumulative maximum error for the i-th mesh refinement level.
- $h[i]$: The time step size (or grid spacing) for the i-th mesh refinement level.

Mathematical Derivation of Convergence Rate Formula The convergence rate formula measures how the error $E[i]$ decreases relative to the step size $h[i]$ across consecutive mesh levels. This calculation assumes that the error behaves as:

$$E \propto h^r,$$

where r represents the order of accuracy. Given two consecutive errors $E[i]$ and $E[i-1]$ and corresponding step sizes $h[i]$ and $h[i-1]$, we have:

$$\frac{E[i]}{E[i-1]} \approx \left(\frac{h[i]}{h[i-1]}\right)^p.$$

Taking the logarithm of both sides, we obtain:

$$\log\left(\frac{E[i]}{E[i-1]}\right) \approx p \cdot \log\left(\frac{h[i]}{h[i-1]}\right).$$

We find that the convergence rate r is approximated by:

$$r \approx \frac{\log\left(\frac{E[i]}{E[i-1]}\right)}{\log\left(\frac{h[i]}{h[i-1]}\right)}.$$

This formulation provides an estimate of the convergence rate between two mesh levels. A rate close to $r = 2$ typically indicates second-order convergence, suggesting that the error decreases quadratically as the mesh is refined. This is expected for well-designed wave equation solvers that achieve second-order accuracy.

Implementation in Code:

- The code checks if both $E[i]$ and $E[i-1]$ are greater than zero to avoid division by zero or taking the logarithm of zero, which would result in undefined behavior.
- The calculated rate r is then appended to the list `rates` for storage.
- A print statement provides debugging information by outputting each computed convergence rate and error for the current mesh level, helping verify the solver's convergence behavior.

Returning and Interpreting the Convergence Rates :

```
return rates
```

The function returns the list of convergence rates, which provides insights into the solver's accuracy. Convergence rates close to 2 indicate that the solver exhibits second-order accuracy, meaning that the error approximately halves as the time step size is halved. This behavior is a key criterion for assessing the reliability and accuracy of the solver under mesh refinement.

These calculated convergence rates are particularly useful for verifying that the numerical scheme conforms to theoretical expectations. High and stable rates across multiple mesh levels indicate that the solver is performing as expected, accurately capturing the wave behavior as the spatial and temporal resolutions are refined.

5.2.2 Summary of Convergence Test

The `convergence_rates` function systematically reduces the time step size and calculates the corresponding convergence rates for the solver. By verifying that the rates align with theoretical expectations (e.g., close to 2 for a second-order accurate method), this test confirms the accuracy and robustness of the wave equation solver.

5.3 Function `test_convrate_sincos`

The `test_convrate_sincos` function evaluates the convergence rate of the wave equation solver using a known exact solution in the form of a sinusoidal wave, which can be accurately represented by the numerical scheme. This test is important for verifying that the solver achieves second-order accuracy in time and space, as theoretically expected for well-designed finite difference solvers.

5.3.1 Code Explanation

The `test_convrate_sincos` function is structured as follows:

```
def test_convrate_sincos():
```

Parameters Setup:

```
# Define parameters
n = m = 2
L = 1.0
```

Here, the parameters are set up as follows:

- n and m: Constants that define the frequency of the wave.
- $L = 1.0$: The length of the spatial domain.

Exact Solution: The exact solution $u_{\text{exact}}(x, t)$ is defined as a function of both space and time:

```
u_exact = lambda x, t: np.cos(m * np.pi / L * t) * np.sin(m * np.pi / L
    * x)
```

The exact solution is:

$$u_{\text{exact}}(x,t) = \cos\left(\frac{m\pi}{L}t\right)\sin\left(\frac{m\pi}{L}x\right).$$

This solution satisfies the wave equation with a fixed wave speed and without an external force. The oscillatory behavior of $\cos(t)$ in time and $\sin(x)$ in space makes this function well-suited for testing the solver's accuracy, as it provides a smooth solution with known derivatives.

Calling the Convergence Rate Calculation: The function then calls `convergence_rates` to calculate the convergence rate for this exact solution across several mesh refinements:

```
rates = convergence_rates(
        u_exact=u_exact,
        I=lambda x: u_exact(x, 0), # Initial displacement
        V=lambda x: 0,             # Initial velocity
        f=0,                       # Source term, no external force
        c=1,                       # Wave speed
        L=L,                       # Length of the domain
        dt0=0.1,                   # Initial time step size
        num_meshes=6,              # Number of grid refinements
        C=0.9,                     # Courant number
        T=1                        # Total simulation time
        )
```

The parameters passed to `convergence_rates` are:

- `u_exact`: The exact solution function defined above.
- `I=lambda x: u_exact(x, 0)`: Initial displacement, set to $u_{\text{exact}}(x,0)$.
- `V=lambda x: 0`: Initial velocity is set to zero, indicating no initial motion.
- `f=0`: The source term is zero, meaning there is no external forcing.
- `c=1`: Wave speed, chosen to match the properties of the exact solution.
- `L=L`: Spatial domain length.
- `dt0=0.1`: Initial time step size.
- `num_meshes=6`: The number of grid refinements to test.
- `C=0.9`: Courant number, slightly below 1 to ensure stability.
- `T=1`: Total simulation time.

The `convergence_rates` function refines the mesh six times, progressively halving the time step size, and calculates the convergence rates based on the errors for each refinement level. These rates provide an indication of the numerical solver's accuracy.

5.3.2 Output and Convergence Check

```
# Output the computed convergence rates, rounded for readability
print('Rates for sin(x) * cos(t) solution:', [round(r_, 2) for r_ in
    rates])
```

The computed convergence rates are printed for reference, rounded to two decimal places to enhance readability.

Assertion for Second-Order Convergence:

```
# Assert that the last computed rate is close to 2, indicating second-order accuracy
assert abs(rates[-1] - 2) < 0.002, f"Expected rate ~2, but got {rates[-1]}"
```

An assertion statement checks that the last computed rate is approximately 2, which would confirm that the solver achieves second-order accuracy. The test allows for a small tolerance (0.002) to account for numerical fluctuations. If the last rate deviates significantly from 2, the assertion fails, signaling that the solver may not be achieving the expected accuracy.

5.3.3 Convergence Results for Scalar Solver

The output below provides a detailed performance summary for the scalar solver. Each test iteration shows the CPU time required, the time step size `dt`, and the corresponding error values, demonstrating the accuracy improvements as the mesh resolution is refined. Additionally, convergence rates are calculated, confirming the solver's second-order accuracy.

```
CPU time (s) of scalar solver: 0.00100
Mesh 1: dt = 1.00000e-01, error = 1.89472e-02
CPU time (s) of scalar solver: 0.00208
Mesh 2: dt = 5.00000e-02, error = 4.58887e-03
CPU time (s) of scalar solver: 0.00607
Mesh 3: dt = 2.50000e-02, error = 1.16273e-03
CPU time (s) of scalar solver: 0.02331
Mesh 4: dt = 1.25000e-02, error = 2.90252e-04
CPU time (s) of scalar solver: 0.09178
Mesh 5: dt = 6.25000e-03, error = 7.25754e-05
CPU time (s) of scalar solver: 0.37134
Mesh 6: dt = 3.12500e-03, error = 1.81417e-05
Convergence rates step 2: r = 2.04577e+00, error = 4.58887e-03
Convergence rates step 3: r = 1.98062e+00, error = 1.16273e-03
Convergence rates step 4: r = 2.00214e+00, error = 2.90252e-04
Convergence rates step 5: r = 1.99975e+00, error = 7.25754e-05
Convergence rates step 6: r = 2.00017e+00, error = 1.81417e-05
Rates for sin(x) * cos(t) solution: [2.05, 1.98, 2.0, 2.0, 2.0]
```

5.3.4 Summary of the `test_convrate_sincos` Function

The `test_convrate_sincos` function validates that the wave equation solver achieves second-order accuracy when applied to a smooth, oscillatory solution. By comparing the numerical and exact solutions across progressively finer meshes and computing the convergence rates, this test confirms the solver's accuracy. A convergence rate close to 2 indicates that the solver meets theoretical expectations, providing confidence in the reliability of the numerical method.

6 GitHub Link to Solver and Exercise Codes and Animations

The developed code, along with the simulation results and animations of various experiments, is available on GitHub. These resources include visualizations of wave behavior under different physical setups and boundary conditions, illustrating the accuracy and stability of the implemented solver.

Solver Code Link:

Generalised Wave PDE Solver Code

Exercise Codes Link:

Exercise Codes and Setups

Animation Link:

Animations

7 Exercise: Simulating a Plucked Guitar String

In this exercise, we use the developed wave equation solver to simulate the behavior of a plucked guitar string. The initial conditions are designed to represent the shape and properties of a plucked string. The simulation aims to capture the wave propagation along the string, saving snapshots of the wave at each time step to produce visualizations and an HTML animation of the wave's evolution.

7.1 Setting Up the Problem

The parameters and initial conditions for the problem are set to reflect the physical characteristics of a plucked guitar string. Key parameters include the length of the domain L, the wave frequency, and the wave speed c, which is calculated based on the frequency and wavelength.

```
L = 0.75          # Length of the domain
x0 = 0.8 * L      # Position where the initial displacement changes slope
a = 0.005         # Amplitude of the initial displacement
freq = 440        # Frequency of the wave (e.g., for a sound wave)
wavelength = 2 * L # Wavelength of the wave, assumed to be twice the
    domain length
c = freq * wavelength # Wave speed, calculated as frequency times
    wavelength
omega = 2 * np.pi * freq # Angular frequency of the wave
num_periods = 1  # Number of wave periods to simulate
```

- $L = 0.75$: The length of the spatial domain, representing the length of the string.
- $x_0 = 0.8 \cdot L$: The location where the initial displacement changes slope, representing the point where the string is plucked.
- $a = 0.005$: Amplitude of the initial displacement, controlling how far the string is displaced.
- `freq` $= 440$: Frequency of the wave in Hz, which is typical for the note A4 in musical notation.
- `wavelength` $= 2 \cdot L$: Wavelength of the wave, assumed to be twice the domain length to reflect the fundamental frequency.
- $c =$ `freq` $\times$ `wavelength`: Wave speed, calculated as the product of frequency and wavelength.
- `omega` $= 2\pi \cdot$ `freq`: Angular frequency, calculated from the frequency.
- `num_periods` $= 1$: Number of wave periods to simulate, ensuring that the simulation captures one full oscillation of the string.

Simulation Time and Stability Conditions:

```
T = 2 * np.pi / omega * num_periods # Total simulation time for one
    period
C = 0.5           # Courant number, a stability factor for the numerical
    method
Nx = 25           # Number of spatial points along the domain
dx = L / Nx       # Spatial step size, calculated based on domain length
    and spatial points
```

```
dt = C * dx / c  # Time step size, adjusted based on the Courant number and spatial resolution
```

- T: Total simulation time, calculated as one period of oscillation based on the angular frequency.
- $C = 0.5$: Courant number, which controls the stability of the numerical method by ensuring the ratio of time step to spatial step meets stability criteria.
- $N_x = 25$: Number of spatial points along the domain.
- $dx = \frac{L}{N_x}$: Spatial step size.
- $dt = \frac{C \cdot dx}{c}$: Time step size, calculated to satisfy the Courant condition.

7.2 Defining Problem-Specific Functions

The initial displacement and velocity functions are defined to represent the plucking of the guitar string. The initial displacement is set to have a triangular shape, with a change in slope at the plucking point x_0. The initial velocity is zero, representing a stationary string.

```
def initial_displacement(x):
    """Initial condition for displacement."""
    return a * x / x0 if x < x0 else a / (L - x0) * (L - x)

def initial_velocity(x):
    """Initial condition for velocity (e.g., zero velocity)."""
    return np.zeros_like(x)

def source_term(x, t):
    """Optional source term f(x, t), here assumed zero."""
    return np.zeros_like(x)
```

- `initial_displacement(x)`: Defines a piecewise linear function for the initial displacement. If $x < x_0$, the displacement increases linearly; otherwise, it decreases linearly.
- `initial_velocity(x)`: Defines the initial velocity as zero, indicating no initial motion in the string.
- `source_term(x, t)`: Represents the source term $f(x, t)$, which is zero for this problem, implying no external force acting on the string.

7.3 Setting Up Directory for Saving Results and Defining Callback Functions

The function defines the directory to save images for visualization, and callback functions for saving results for the GIF and HTML animation:

```
save_dir = 'path/to/guitar_string_simulation'

# Array to store solution data for HTML animation
results = []

# Capture results for HTML animation
def capture_results(u, x, t, n):
    results.append((u.copy(), t[n]))

# Save wave image for GIF generation
def save_guitar_string_image(u, x, t, n, C, save_dir=save_dir, ymin=-0.005, ymax=0.005):
    save_wave_image(u, x, t, n, C, save_dir=save_dir, ymin=ymin, ymax=ymax)

# Combined user action that calls both capture_results and save_wave_image
def combined_user_action(u, x, t, n):
    capture_results(u, x, t, n)
    save_guitar_string_image(u, x, t, n, C)
```

- `capture_results`: Saves wave data for each time step in the `results` array, enabling it to be later used for HTML animation.
- `save_guitar_string_image`: Saves images of the wave at each time step to create a GIF, with y-axis limits set to -0.005 to 0.005 to capture the wave amplitude.
- `combined_user_action`: Combines the functions `capture_results` and `save_guitar_string_image` into a single callback function.

7.4 Solving the Wave Equation and Generating Output

The code prompts the user to select a solver version (scalar, vectorized, or vectorized2) and runs the simulation using the chosen version. Afterward, it generates a GIF and an HTML animation of the wave.

```
# Ask user to choose between solvers
solver_choice = input("Choose solver (scalar/vectorized/vectorized2): ").strip()

# Use the unified solver based on user choice
if solver_choice in ["scalar", "vectorized", "vectorized2"]:
    solve_wave_equation(
    I=initial_displacement,
    V=initial_velocity,
    f=source_term,
    c=c,
    L=L,
    dt=dt,
    C=C,
    T=T,
    user_action=combined_user_action,
    version=solver_choice,
    save_dir=save_dir,
```

```
                )
        else:
                print("Invalid choice. Please choose 'scalar', 'vectorized', or '
                    vectorized2'.")

        # Generate a GIF from the saved images
        generate_gif_from_images(image_folder=save_dir, gif_name='wave_animation
            .gif', duration=0.1)

        # Generate HTML animation from captured results
        x = np.linspace(0, L, int(L / dx) + 1)
        generate_html_animation(x, results, save_dir, ymin=-0.005, ymax=0.005,
            fps=10)
```

This code block:

- Runs the wave equation solver with the initial conditions, wave parameters, and user-specified solver version.
- Generates a GIF from the saved images using `generate_gif_from_images`, allowing for a sequential view of wave evolution.
- Generates an HTML animation using `generate_html_animation`, which can be viewed interactively in a web browser.

7.5 Summary of Exercise

This exercise simulates the behavior of a plucked guitar string using the 1D wave equation solver. The setup demonstrates how to configure initial conditions, select solver versions, and use post-processing functions to generate visualizations. By capturing the wave's evolution over time, this exercise provides a detailed view of wave dynamics, which can be used for both educational and analytical purposes.

7.6 Link to Code, Simulation Results and Experiments

The results of the guitar string simulation, including animations of the wave propagation for various Courant numbers, are available on GitHub. The animations illustrate the effect of different Courant numbers on wave stability and accuracy. You can view the results through the following link:

Code Link:

Simulation Results a Plucked Guitar String using developed solver

Animation Link:

Code for Simulation of a Plucked Guitar String using developed solver

7.6.1 Experiment: Wave Propagation on a Plucked String

As an additional experiment, this exercise can be expanded to replicate the classic demonstration of wave propagation on a plucked string, as shown on the Penn State Acoustics website:

> Theory and Experiment behind a Plucked Guitar String

In this demonstration, the initial conditions set up a triangular shape along the string, which corresponds to plucking the string at a specific point. The wave travels back and forth, reflecting at the ends, illustrating the standing wave patterns that form due to constructive and destructive interference of waves traveling in opposite directions.

To replicate this in our solver:

- Set the initial displacement as a triangular shape, with the peak at the plucking point (e.g., 80% of the string length).
- Set the initial velocity to zero, representing a stationary string before plucking.
- Run the simulation over multiple periods to observe the formation of standing wave patterns and analyze how harmonics contribute to the overall wave shape.

This experiment provides insight into wave interference, boundary reflections, and the formation of harmonic frequencies, contributing to a deeper understanding of wave dynamics in musical instruments like the guitar.

8 Exercise: Simulating Standing Waves

In this exercise, we use the developed wave equation solver to simulate standing waves on a string with fixed endpoints. Standing waves occur when a wave reflects back and forth in a confined space, producing stable patterns based on constructive and destructive interference. The simulation incorporates phase changes upon reflection, a characteristic behavior observed in standing waves.

8.1 Setting Up the Problem

We simulate a standing wave on a domain with fixed endpoints, where the wave speed, domain length, and initial displacement are set to produce a standing wave pattern. Fixed boundary conditions create a 180° phase change upon reflection, which contributes to the formation of nodes (points of no motion) and antinodes (points of maximum motion) along the wave.

```
L = 12            # Length of the domain
A = 1             # Amplitude of the standing wave
m = 1             # Mode number
c = 2             # Wave speed
C = 0.5           # Courant number
Nx = 50           # Number of spatial points
dx = L / Nx       # Spatial step size
dt = C * dx / c   # Time step size based on Courant condition
T = 10            # Total simulation time
save_dir = r'path/to/standing_wave_simulation'
```

- $L = 12$: Length of the domain.
- $A = 1$: Amplitude of the standing wave, controlling the initial displacement's maximum height.
- $m = 1$: Mode number, determining the number of half-wavelengths fitting into the domain.
- $c = 2$: Wave speed.
- $C = 0.5$: Courant number, meeting the stability requirement.
- $N_x = 50$: Number of spatial points.
- $dx = \frac{L}{N_x}$: Spatial step size.
- $dt = \frac{C \cdot dx}{c}$: Time step size, calculated based on the Courant condition.

8.2 Phase Change upon Reflection and Boundary Conditions

A wave on a string reflects with a 180° phase change at fixed endpoints, meaning that a crest reflecting from a fixed boundary becomes a trough and vice versa. This phase change is essential for forming standing waves, as it enables nodes (points of zero displacement) to remain stationary while antinodes oscillate with maximum amplitude.

When waves reflect from a free boundary, such as a lighter string meeting a heavier one, no phase change occurs, allowing continuous motion. In this simulation, we use Dirichlet (fixed) boundary conditions to enforce the phase change, contributing to the formation of stable standing wave patterns. The relationship between fixed and free reflections can be further observed when studying wave behavior at boundaries of differing mass densities.

8.3 Defining Initial Conditions for a Standing Wave

The initial displacement is defined as a sinusoidal shape that matches the expected standing wave pattern. The initial velocity is set to zero, implying the wave starts from rest.

```
1                    def initial_displacement(x):
2                        return A * np.sin(np.pi * m * x / L)
3
4                    def initial_velocity(x):
5                        return np.zeros_like(x)
6
7                    def source_term(x, t):
8                        return np.zeros_like(x)
```

- `initial_displacement(x)`: Defines the initial standing wave profile:

$$u(x, 0) = A \sin\left(\frac{\pi m x}{L}\right).$$

- `initial_velocity(x)`: Sets the initial velocity to zero.
- `source_term(x, t)`: No external forcing is applied, so $f(x, t) = 0$.

8.4 Capturing Simulation Data for Visualization

The simulation results are stored for animation purposes, enabling both GIF and HTML animations of the standing wave pattern.

```
results = []

def capture_results(u, x, t, n):
    results.append((u.copy(), t[n]))

def combined_user_action(u, x, t, n):
    capture_results(u, x, t, n)
    save_wave_image(u, x, t, n, C, save_dir=save_dir, ymin=-A, ymax=A)
```

- `capture_results`: Stores the wave solution at each time step for HTML animation.
- `combined_user_action`: Combines result capturing with saving wave images for the GIF.

8.5 Running the Simulation and Generating Output

The wave equation solver is run with Dirichlet boundary conditions to enforce fixed endpoints. Afterward, GIF and HTML animations are generated to visualize the wave pattern.

```
solve_wave_equation(
    I=initial_displacement,
    V=initial_velocity,
    f=source_term,
    c=c,
    L=L,
    dt=dt,
    C=C,
    T=T,
    user_action=combined_user_action,
    version='scalar',
    save_dir=save_dir,
    boundary='Dirichlet'
    )

generate_gif_from_images(image_folder=save_dir, gif_name='wave_animation.gif',
    duration=0.1)

x = np.linspace(0, L, Nx + 1)
generate_html_animation(x, results, save_dir, filename="wave_animation.html",
    ymin=-A, ymax=A, fps=10)
```

- The solver is configured with Dirichlet (fixed) boundary conditions, which ensure that the wave undergoes a 180° phase change upon reflection.
- `generate_gif_from_images` and `generate_html_animation` create visualizations of the standing wave.

8.6 Summary of the Standing Wave Simulation

This simulation illustrates the formation of standing waves in a string with fixed endpoints. The Dirichlet boundary conditions enforce a 180° phase change, essential for forming nodes and antinodes. The setup can be modified to explore the effects of different boundary conditions, such as Neumann (free) boundaries, which exhibit no phase change. Such variations are useful for understanding the principles behind sound production in stringed instruments and wave behavior across boundaries of differing densities.

9 Example: Standing Wave with Exact Solution Comparison

In this example, we solve a wave equation for a standing wave on a domain of length $L = 0.75$, with the wave mode $m = 1$ and wave speed $c = 2$. We use a Courant number $C = 0.5$ for stability, and set the number of spatial points $N_x = 25$. This setup provides a scenario where we can compare the numerical solution against an analytical, exact solution to assess the accuracy of the implemented solver.

9.1 Problem Setup and Parameters

The setup starts with defining key parameters, including the length of the domain, wave speed, and Courant number, which determines the stability of the simulation.

```
# Parameters for the Wave Equation
L = 0.75            # Length of the domain
A = 0.005           # Amplitude of the standing wave
m = 1               # Mode number of the standing wave
c = 2               # Wave speed
C = 0.5             # Courant number for stability
Nx = 25             # Number of spatial points
dx = L / Nx         # Spatial step size
dt = C * dx / c     # Time step size based on the Courant number
T = 2               # Total simulation time
```

9.2 Exact Solution for Standing Wave

For this problem, the exact solution to the wave equation for a standing wave is used to evaluate the accuracy of the numerical solver. In a standing wave scenario, each point in the domain oscillates in place without a net movement along the spatial direction. This behavior results from constructive and destructive interference of two traveling waves moving in opposite directions. The general form of the solution for a standing wave is derived from the separation of variables in the wave equation.

9.2.1 Mathematical Representation of the Exact Solution

For a wave equation in one dimension with fixed boundaries, the exact solution $u(x, t)$ for a standing wave can be expressed as:

$$u(x,t) = A \sin\left(\frac{m\pi x}{L}\right) \cos\left(\frac{m\pi ct}{L}\right),$$

where:

- A: Amplitude of the wave.
- m: Mode number, which determines the number of half-wavelengths that fit within the domain L.
- L: Length of the spatial domain.
- c: Wave speed, which affects the temporal frequency of the wave.
- x and t: Spatial and temporal coordinates, respectively.

9.2.2 Interpretation of the Solution Components

- The spatial term $\sin\left(\frac{m\pi x}{L}\right)$ describes the shape of the wave along the length of the domain at any given time. The mode number m determines the number of nodes (points of zero amplitude) along the length, with $m = 1$ representing the fundamental frequency and $m = 2, 3, \ldots$ representing higher harmonics.

- The temporal term $\cos\left(\frac{m\pi ct}{L}\right)$ governs the oscillation of each point in the domain, producing periodic oscillations without spatial translation. The frequency of oscillation is proportional to the wave speed c and the mode number m.

9.2.3 Code Implementation of the Exact Solution

The exact solution is implemented in the following Python function, allowing for comparison with the numerical solution at each time step. This function takes the spatial coordinates x and a specific time t and returns the wave amplitude $u(x, t)$ at those points.

```
1                    def exact_solution(x, t):
2                        """Exact solution of the wave equation for a standing wave."""
3                        return A * np.sin(np.pi * m * x / L) * np.cos(np.pi * m * c * t / L)
```

9.2.4 Purpose of the Exact Solution in Error Analysis

Using this exact solution, we calculate error norms, such as the L2 and maximum errors, by comparing the exact and numerical solutions. At each time step, these error metrics provide insight into the accuracy of the numerical method and highlight any discrepancies due to factors like discretization, boundary conditions, or numerical instability.

This exact solution serves as a benchmark, allowing us to verify that the numerical scheme converges toward the true solution as we refine the spatial and temporal discretizations.

9.3 Initial and Boundary Conditions

For this simulation, the initial displacement is given by the exact solution at $t = 0$, while the initial velocity is set to zero, indicating no initial movement.

```
                    def initial_displacement(x):
                        """Initial displacement of the wave, matching the exact solution."""
                        return A * np.sin(np.pi * m * x / L)

                    def initial_velocity(x):
                        """Initial velocity of the wave (set to zero for a standing wave)."""
                        return np.zeros_like(x)

                    def source_term(x, t):
                        """Source term (set to zero for this example)."""
                        return np.zeros_like(x)
```

9.4 Error Calculation

To evaluate the accuracy of the numerical solution, we calculate two important error norms at each time step: the L2 norm and the maximum error norm. These metrics provide a quantitative measure of how closely the numerical solution approximates the exact solution.

9.4.1 L2 Norm Error Calculation

The L2 norm (or root mean square error) provides a measure of the average difference between the numerical solution u_{num} and the exact solution u_{exact} across the spatial domain. Mathematically, it is defined as:

$$\text{L2 Error} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(u_{\text{num}}[i] - u_{\text{exact}}[i]\right)^2},$$

where:

- N is the number of spatial points.
- $u_{\text{num}}[i]$ and $u_{\text{exact}}[i]$ are the numerical and exact solutions, respectively, at spatial point i.

The L2 norm is useful for measuring the overall error distributed across the domain. A smaller L2 norm indicates that, on average, the numerical solution is close to the exact solution over the entire domain.

9.4.2 Maximum Error Norm Calculation

The maximum error norm (or L_∞ norm) provides a measure of the largest pointwise difference between the numerical and exact solutions across the domain. It is defined as:

$$\text{Max Error} = \max_i \left|u_{\text{num}}[i] - u_{\text{exact}}[i]\right|.$$

This metric captures the maximum deviation between the numerical and exact solutions at any point in the domain, providing a measure of the worst-case error.

9.4.3 Implementation of Error Calculation in Code

The error norms are calculated in the following Python function, which takes as input the numerical solution u_{num} and the exact solution u_{exact}. The function computes both the L2 and maximum errors and returns them.

```
def calculate_error(u_num, u_exact):
    """Calculates L2 and Max errors."""
    error_L2 = np.sqrt(np.mean((u_num - u_exact) ** 2)) # L2 norm error
    error_max = np.max(np.abs(u_num - u_exact))     # Max error

    return error_L2, error_max
```

9.4.4 Purpose of Error Calculation

By calculating both the L2 and maximum error norms at each time step, we gain insight into the accuracy and stability of the numerical scheme. These error norms help identify:

- **Global Accuracy (L2 Norm):** The L2 norm provides an overall measure of accuracy across the entire spatial domain.
- **Pointwise Deviation (Max Error):** The maximum error highlights the worst-case deviation, which is essential for ensuring that the numerical solution does not diverge significantly from the exact solution at any specific point.

Tracking these errors over time helps in diagnosing issues such as numerical instability, boundary inaccuracies, or discretization errors. Smaller errors indicate a closer alignment of the numerical solution with the exact solution, which is critical for validating the effectiveness of the chosen numerical method.

9.5 Visualization of Results with Error Comparison

To visualize the results, we save images showing both the numerical and exact solutions, displaying error norms in the plot title for each frame.

```
def save_wave_image_with_exact(u_num, x, t, n, C, save_dir='wave_images', ymin
    =-0.01, ymax=0.01):
    """Saves images with both numerical and exact solutions at each time
        step."""
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

    u_exact = exact_solution(x, t[n]) # Exact solution at this time step
    # Calculate error norms using the `calculate_error` function
    error_L2, error_max = calculate_error(u_num, u_exact)

    # Plot numerical and exact solutions
    plt.figure(figsize=(8, 4))
    plt.plot(x, u_num, label="Numerical Solution", color="blue")
    plt.plot(x, u_exact, label="Exact Solution", linestyle="--", color="red"
        )
    plt.ylim(ymin, ymax)
    plt.xlim(0, L)
```

```
                plt.xlabel('x')
                plt.ylabel('u(x,t)')

                # Add Courant number, time, and error norms to the plot title
                plt.title(f"Wave at t = {t[n]:.3f}, C = {C}, L2 Error = {error_L2:.5e},
                    Max Error = {error_max:.5e}")

                plt.legend()
                plt.grid(True)
                plt.savefig(os.path.join(save_dir, f'wave_step_{n:04d}.png'))
                plt.close()
```

9.6 Running the Solver and Generating the Animation

The solver function, `solve_wave_equation`, is called with a combined user action that captures the results for the HTML animation and saves images for GIF generation.

```
# Run the Solver
solve_wave_equation(
        I=initial_displacement,
        V=initial_velocity,
        f=source_term,
        c=c,
        L=L,
        dt=dt,
        C=C,
        T=T,
        user_action=combined_user_action,
        version="scalar", # Solver version: 'scalar', 'vectorized', or '
            vectorized2'
        boundary='Dirichlet'
        )
```

After the simulation completes, we generate a GIF and HTML animation to visualize the wave propagation, highlighting the differences between the numerical and exact solutions.

```
# Generate a GIF from the saved images
generate_gif_from_images(image_folder=save_dir, gif_name='
    wave_comparison_animation.gif', duration=0.1)
```

—

This section demonstrates the implementation of a wave equation solver with exact solution comparison, showcasing error analysis techniques and visualization of standing waves in a one-dimensional domain.

9.7 Link to Code, Simulation Results, and Theory/Experiments

Code Link:

Standing Wave Problem and Error Evaluation

Animation Link:

Animations Standing Wave Problem and Error Evaluationy

Theory Link:

Standing Sound Waves (Longitudinal Standing Waves)

10 Exercise: Simulating Gaussian Wave Propagation

In this exercise, we use the developed wave equation solver to simulate the propagation of a Gaussian wave packet along a 1D domain. Gaussian wave packets are commonly used to model localized wave disturbances and are useful in understanding wave dispersion, reflection, and stability in numerical simulations.

10.1 Setting Up the Problem

The Gaussian wave packet is initially centered at a specific point on the domain with a specified standard deviation, controlling the width of the packet. The wave packet propagates according to the 1D wave equation, without an external source term.

```
L = 10.0          # Length of the domain
c = 10            # Wave speed
sigma = 0.5       # Standard deviation for Gaussian initial condition
Nx = 50           # Number of spatial points for resolution
C = 1             # Courant number
T = 3             # Total simulation time
loc = 5           # Location of Gaussian peak
```

- $L = 10.0$: Length of the domain.
- $c = 10$: Wave speed, controlling the rate at which the wave propagates.
- `sigma` $= 0.5$: Standard deviation of the Gaussian, defining the packet's width.
- `Nx` $= 50$: Number of spatial points, controlling the spatial resolution.
- `C` $= 1$: Courant number, which governs stability and ensures accuracy in the numerical solution.
- `T` $= 3$: Total simulation time, which allows us to observe the full propagation of the wave packet.
- `loc` $= 5$: Initial location of the Gaussian peak, set to the center of the domain.

10.1.1 Derived Parameters

The spatial and temporal step sizes are calculated to satisfy the Courant condition, which is essential for stability.

```
dx = L / Nx       # Spatial step size
dt = C * dx / c   # Time step size
```

- $dx = \frac{L}{N_x}$: Spatial step size, calculated based on the length of the domain and the number of spatial points.
- $dt = \frac{C \cdot dx}{c}$: Time step size, adjusted to satisfy the Courant condition.

10.2 Defining Problem-Specific Functions

The initial displacement is defined as a Gaussian function centered at `loc`, with zero initial velocity and no external force (source term).

```
def initial_displacement(x):
    return (1 / np.sqrt(2 * np.pi * sigma)) * np.exp(-0.5 * ((x - loc) /
        sigma) ** 2)

def initial_velocity(x):
    return np.zeros_like(x)

def source_term(x, t):
    return np.zeros_like(x)
```

- `initial_displacement(x)`: Defines the initial Gaussian profile of the wave packet, with the peak centered at `loc` and width controlled by `sigma`. Mathematically, this is given by:

$$u(x,0) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\text{loc})^2}{2\sigma^2}\right).$$

- `initial_velocity(x)`: Defines the initial velocity as zero, meaning the wave starts from rest.
- `source_term(x, t)`: The source term $f(x,t) = 0$, indicating that there is no external forcing in the wave equation for this problem.

10.3 Setting Up Directory for Saving Results and Defining Callback Functions

The directory for saving images is specified, and callback functions are defined for capturing results for HTML animation and saving wave images for a GIF animation.

```
save_dir = 'path/to/gaussian_wave_images'

results = []

def capture_results(u, x, t, n):
    results.append((u.copy(), t[n]))

def save_wave_image(u, x, t, n, C, save_dir='wave_images', ymin=-0.6, ymax=0.7)
    :
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

plt.figure(figsize=(8, 4))
plt.plot(x, u, label=f"Numerical Solution (t = {t[n]:.3f})", color="blue")
plt.ylim(ymin, ymax)
plt.xlim(0, L)
plt.xlabel('x')
plt.ylabel('u(x,t)')
plt.title(f"Gaussian Wave Propagation at t = {t[n]:.3f}, Courant number = {C}")
plt.legend()
plt.grid(True)
filename = os.path.join(save_dir, f'gaussian_step_{n:04d}.png')
```

```
                    plt.savefig(filename)
                    plt.close()

                    def combined_user_action(u, x, t, n):
                        capture_results(u, x, t, n)
                        save_wave_image(u, x, t, n, C, save_dir=save_dir, ymin=-0.6, ymax=0.7)
```

- `capture_results`: Saves the wave data at each time step for use in an HTML animation.
- `save_wave_image`: Generates and saves images of the wave at each time step for later compilation into a GIF. The y-axis is set to capture the amplitude of the Gaussian wave packet.
- `combined_user_action`: Combines both functions above into a single callback function that both stores the wave data and saves images.

10.4 Running the Simulation and Generating Output

The wave equation solver is run with the specified parameters and initial conditions, using the combined user action to store and save results.

```
solve_wave_equation(
I=initial_displacement,
V=initial_velocity,
f=source_term,
c=c,
L=L,
dt=dt,
C=C,
T=T,
user_action=combined_user_action,
version='scalar', # Choose solver version: 'scalar', 'vectorized', or '
    vectorized2'
boundary='Neumann' # Use Dirichlet or Neumann boundaries
)

generate_gif_from_images(image_folder=save_dir, gif_name='
    gaussian_wave_animation.gif', duration=0.1)

x = np.linspace(0, L, Nx + 1)
generate_html_animation(x, results, save_dir, filename="gaussian_wave_animation
    .html")
```

- The wave equation solver runs with the Gaussian initial displacement and zero initial velocity, capturing the wave's evolution over time.
- A GIF is generated from the saved images using `generate_gif_from_images`, providing a visual animation of the wave packet's propagation.
- An HTML animation is created using `generate_html_animation`, allowing for an interactive visualization in a web browser.

10.5 Summary of the Gaussian Wave Packet Exercise

This exercise illustrates the propagation of a Gaussian wave packet in a one-dimensional domain. By analyzing the wave's evolution, we can observe the effects of reflection, wave speed, and Courant number on wave behavior. Gaussian wave packets are valuable for exploring localized wave phenomena and understanding the stability and accuracy of the numerical scheme.

10.6 Link to Code, Simulation Results, and Theory/Experiments

Code Link:

Gaussian Wave Problem and Wave Superposition

Animation Link:

Animations Gaussian Wave propagation and its superposition upon reflecting of BC - different BC examples

Theory Link:

Superposition of Waves)

11 Exercise: Simulating Wave Propagation with a Moving Left Boundary

This exercise simulates wave propagation in a one-dimensional domain, with a sinusoidal oscillation applied to the left boundary. This oscillating boundary condition acts as a source, initiating wave propagation through the domain. Such simulations can model scenarios where an external force or oscillation is applied at the edge of a medium.

11.1 Setting Up the Problem

The simulation is configured to apply a time-dependent sinusoidal displacement on the left boundary. By introducing an oscillating boundary, we can observe how waves propagate and reflect within the domain. The initial displacement and velocity are set to zero, so the wave motion is generated solely by the boundary condition.

```
L = 1.0           # Length of the domain
c = 1.0           # Wave speed
Nx = 50           # Number of spatial points for resolution
C = 0.5           # Courant number for stability
T = 2.0           # Total simulation time
dt = C * (L / Nx) / c # Time step size based on the wave speed and
    Courant number
save_dir = 'path/to/moving_wave' # Directory to save images
```

- $L = 1.0$: Length of the domain.
- $c = 1.0$: Wave speed, determining the propagation speed through the domain.

- $N_x = 50$: Number of spatial points, defining the spatial resolution.
- $C = 0.5$: Courant number for stability.
- $T = 2.0$: Total simulation time, chosen to capture multiple oscillations.
- $dt = \frac{C \cdot (L/N_x)}{c}$: Time step size, calculated to meet the Courant condition.

11.2 Defining the Moving Left Boundary Condition

The boundary condition at the left edge of the domain, $x = 0$, is designed to introduce a time-dependent oscillation. This oscillation can simulate various physical conditions, such as a wave entering the domain intermittently. Two versions of the boundary condition function, `U_0(t)`, are provided below, allowing for different types of oscillatory input.

11.2.1 Option 1: Specified Oscillation Intervals

In this version, the left boundary undergoes sinusoidal oscillations at specific intervals. The function `U_0(t)` returns an oscillatory value with amplitude 0.25 and frequency 6π during three specified time periods.

```
def U_0(t):
    return 0.25 * np.sin(6 * np.pi * t) if ((t < 1./6) or (0.5 +
        3./12 <= t <= 0.5 + 4./12) or (1.5 <= t <= 1.5 + 1./3)) else
         0
```

- `U_0(t)`: Produces sinusoidal oscillations with an amplitude of 0.25 and frequency 6π during the following time intervals:
  - $t < \frac{1}{6}$: Initial oscillation phase.
  - $0.5 + \frac{3}{12} \leq t \leq 0.5 + \frac{4}{12}$: Midway oscillation phase.
  - $1.5 \leq t \leq 1.5 + \frac{1}{3}$: Final oscillation phase.

  Outside these intervals, the left boundary remains stationary.

11.2.2 Option 2: Periodic Pulses with Pulse Duration

The second option for defining the boundary condition applies a periodic pulse at the left boundary, triggered at regular intervals. In this approach, a sinusoidal pulse is emitted every `pulse_period` seconds, active only for a duration of `pulse_duration` within each period.

```
def U_0(t):
    """
    Boundary condition at x=0, generating a sinusoidal pulse at
        regular intervals.
    Only active for ‘pulse_duration‘ within each ‘pulse_period‘.
    """
    # Periodicity of pulses at the left boundary
    pulse_period = 2.0
    # Duration of each pulse within the period
    pulse_duration = 0.15
```

```
                pulse_time = t % pulse_period
                return 0.25 * np.sin(6 * np.pi * pulse_time) if pulse_time <=
                    pulse_duration else 0
```

- `U_0(t)`: Produces a sinusoidal oscillation with amplitude 0.25 and frequency 6π during each pulse duration:
  - `pulse_period` defines the interval at which a pulse is triggered, set here to 2 seconds.
  - `pulse_duration` controls how long each pulse lasts within the interval, set here to 0.15 seconds.

  During each pulse duration, a sinusoidal wave is emitted at the boundary; otherwise, the boundary remains stationary.

Each boundary condition option provides a different wave input behavior, allowing for flexibility in simulating various wave interactions within the domain.

11.3 Defining Initial Conditions

The initial conditions are set to zero displacement and velocity, meaning that the wave propagation is entirely driven by the boundary condition.

```
def initial_displacement(x):
    return np.zeros_like(x)

def initial_velocity(x):
    return np.zeros_like(x)

def source_term(x, t):
    return np.zeros_like(x)
```

- `initial_displacement(x)`: Sets the initial displacement to zero.
- `initial_velocity(x)`: Sets the initial velocity to zero, implying no initial motion.
- `source_term(x, t)`: No external forcing is applied, so $f(x,t) = 0$.

11.4 Setting Up Data Capture and Visualization

The results are stored at each time step for animation purposes. The `combined_user_action` function calls both `capture_results` and `save_wave_image` to store data and images for visualization.

```
results = []

def capture_results(u, x, t, n):
    results.append((u.copy(), t[n]))

def save_wave_image(u, x, t, n, save_dir='wave_images', ymin=-0.6, ymax
    =0.7):
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

```

```
                plt.figure(figsize=(8, 4))
                plt.plot(x, u, label=f"Numerical Solution (t = {t[n]:.3f})", color="blue
                    ")
                plt.ylim(ymin, ymax)
                plt.xlim(0, L)
                plt.xlabel('x')
                plt.ylabel('u(x,t)')
                plt.title(f"Wave Propagation at t = {t[n]:.3f}, Courant number = {C}")
                plt.legend()
                plt.grid(True)
                filename = os.path.join(save_dir, f'wave_step_{n:04d}.png')
                plt.savefig(filename)
                plt.close()

                def combined_user_action(u, x, t, n):
                        capture_results(u, x, t, n)
                        save_wave_image(u, x, t, n, save_dir=save_dir, ymin=-0.6, ymax
                            =0.7)
```

11.5 Modifying the Solver for the Moving Boundary Condition

To incorporate an oscillating boundary condition at the left edge of the domain, a wrapper function, `solve_wave_equation_with_moving_left_boundary`, is introduced. This function modifies the standard wave equation solver by applying a specified time-dependent boundary condition to the left boundary. The primary role of this wrapper is to ensure that the oscillating condition is applied at each time step before the solver computes the wave field for that step.

11.5.1 Function Overview and Code Explanation

The function `solve_wave_equation_with_moving_left_boundary` sets up the solver parameters and wraps the standard solver's `user_action` function. The wrapper function handles three primary tasks:

1. Setting up the time and spatial grids based on the input parameters.
2. Defining a modified user action function, `wrapped_user_action`, to apply the moving boundary condition.
3. Calling the standard `solve_wave_equation` function with the wrapped user action function.

The following code shows the structure of this wrapper function:

```
def solve_wave_equation_with_moving_left_boundary(
I, V, f, c, L, dt, C, T, U_0, user_action=None, version='scalar',
    boundary='Dirichlet'
):
        # Calculate the number of time steps based on total time and time
            step size
        Nt = int(round(T / dt))

        # Create an array of time points from 0 to T, with Nt+1 elements
        t = np.linspace(0, Nt * dt, Nt + 1)

```

```
# Define spatial step size based on domain length and spatial resolution
dx = L / Nx

# Create an array of spatial points from 0 to L, with Nx+1 elements
x = np.linspace(0, L, Nx + 1)

# Define a modified user action function that applies the moving left boundary condition
def wrapped_user_action(u, x, t, n):
    # Set the left boundary value according to the oscillating boundary function U_0 at time step n
    u[0] = U_0(t[n])

    # If a user-defined action is provided, call it after applying the boundary condition
    if user_action:
        user_action(u, x, t, n)

# Call the wave equation solver, passing in the wrapped user action
solve_wave_equation(
    I=I,              # Initial displacement function
    V=V,              # Initial velocity function
    f=f,              # Source term function
    c=c,              # Wave speed
    L=L,              # Length of the spatial domain
    dt=dt,            # Time step size
    C=C,              # Courant number for stability
    T=T,              # Total simulation time
    user_action=wrapped_user_action, # Use the modified action to apply boundary condition
    version=version,  # Solver version ('scalar', 'vectorized', etc.)
    boundary=boundary # Boundary type ('Dirichlet', 'Neumann', etc.)
    )
```

11.5.2 Step-by-Step Explanation

1. Defining Time and Spatial Grids The wrapper function first computes the total number of time steps, `Nt`, based on the total simulation time, T, and the time step size, Δt:

$$\texttt{Nt} = \texttt{int(round(T / dt))}$$

Then, a time array `t` is created, which contains all the time points from $t = 0$ to $t = T$. The spatial grid `x` is similarly defined, creating points from $x = 0$ to $x = L$, with `Nx + 1` points based on the domain length and spatial resolution.

2. Applying the Oscillating Boundary Condition: The Wrapped User Action Function The core functionality for handling the moving boundary condition is implemented in the `wrapped_user_action` function. This function modifies the value at the left boundary, `u[0]`, ac-

cording to the oscillating condition given by the `U_0` function:

$$u[0] = U_0(t[n])$$

Here, `U_0(t[n])` evaluates the boundary function at the current time step n. This value is assigned to the left boundary of the solution array `u`, effectively setting the boundary condition at each time step before proceeding with the rest of the computations.

By defining `wrapped_user_action` this way, we ensure that the boundary condition is applied at every time step before any other user-defined actions (such as capturing or visualizing results) are executed. If an additional `user_action` function is provided as an argument, `wrapped_user_action` calls it after applying the boundary condition, allowing for data storage or visualization as required.

3. Calling the Standard Solver The wrapper then calls the standard `solve_wave_equation` function, passing the wrapped user action function, along with other input parameters:

```
solve_wave_equation(
I=I,              # Initial displacement function
V=V,              # Initial velocity function
f=f,              # Source term function
c=c,              # Wave speed
L=L,              # Length of the spatial domain
dt=dt,            # Time step size
C=C,              # Courant number for stability
T=T,              # Total simulation time
user_action=wrapped_user_action, # Use the modified action to apply
    boundary condition
version=version,  # Solver version ('scalar', 'vectorized', etc.)
boundary=boundary # Boundary type ('Dirichlet', 'Neumann', etc.)
)
```

This setup allows the `solve_wave_equation` function to handle the wave propagation while incorporating the oscillating boundary condition. By using `wrapped_user_action`, the solver consistently applies the specified left boundary condition during each time step.

Summary of Key Components

- `solve_wave_equation_with_moving_left_boundary`: A wrapper function that incorporates an oscillating left boundary condition by setting $u[0] = U_0(t[n])$ at each time step n.
- `wrapped_user_action`: A modified user action function that ensures the oscillating boundary condition is applied before any additional actions (e.g., data capture or visualization) are executed.

This approach allows for flexible handling of dynamic boundary conditions, making it useful for simulating scenarios such as oscillating boundaries or driven oscillations in physical systems.

11.6 Running the Simulation and Generating Output

The wave equation solver is run with the sinusoidal left boundary condition, and animations are generated to visualize the wave propagation.

```
solve_wave_equation_with_moving_left_boundary(
        I=initial_displacement,
        V=initial_velocity,
        f=source_term,
        c=c,
        L=L,
        dt=dt,
        C=C,
        T=T,
        U_0=U_0,
        user_action=combined_user_action,
        version='scalar',
        boundary= 'leftFree'
        )

generate_gif_from_images(image_folder=save_dir, gif_name='
    wave_with_moving_boundary.gif', duration=0.1)

x = np.linspace(0, L, Nx + 1)
generate_html_animation(x, results, save_dir, filename="
    wave_with_moving_boundary.html", ymin=-0.6, ymax=0.7)
```

- The solver executes with the moving boundary condition, simulating wave propagation initiated by the oscillating boundary.
- `generate_gif_from_images` creates a GIF of the wave propagation.
- `generate_html_animation` generates an HTML animation for interactive viewing.

11.7 Summary of the Moving Boundary Condition Exercise

This exercise demonstrates wave propagation with an oscillating boundary, simulating the effect of an external force applied at one end. This setup is useful for modeling waves in scenarios such as vibrating boundaries or driven oscillations, where the boundary condition itself acts as a wave source. The resulting visualizations show the influence of the moving boundary on wave generation and propagation throughout the domain.

11.8 Link to Code, Simulation Results, and Theory/Experiments

Code Link:

Moving Wave Problem

Animation Link:

Animations of Moving Wave Problem

Theory Link:

Reflection of Waves from Boundaries

12 Solver for the Wave Equation with Variable Wave Velocity

In many real-world scenarios, wave speed is not constant throughout the medium; it may vary due to changes in material properties or environmental conditions. The `solve_wave_equation_variable_velocity` function is designed to handle such cases, where the wave speed $c(x)$ changes spatially. This solver accounts for these variations in wave speed, represented as:

$$q(x) = c(x)^2,$$

which directly influences the wave propagation and reflections within the domain. The equation governing this model is:

$$u_{tt} = (q(x)u_x)_x + f(x,t),$$

where $u(x,t)$ is the displacement field and $f(x,t)$ is an optional source term that can represent external forces or disturbances.

This function implements finite difference methods to approximate the solution, allowing for both Dirichlet and Neumann boundary conditions. The inclusion of an optional `user_action` parameter provides flexibility to incorporate custom post-processing, such as visualization or data collection.

The solver's ability to handle spatially varying wave speeds and custom boundary conditions makes it highly adaptable for simulations.

12.1 Function Header and Code

```
def solve_wave_equation_variable_velocity(I, V, f, q, L, dt, C, T, user_action=
    None, version='scalar', save_dir=None, boundary='Dirichlet'):
"""
Solve u_tt = (q(x) * u_x)_x + f with variable wave velocity q(x) = c(x)^2.

Parameters:
- I: Initial displacement function.
- V: Initial velocity function.
- f: Source term function.
- q: Spatially varying wave speed squared (q(x) = c(x)^2).
- L: Domain length.
- dt: Time step size.
- C: Courant number.
- T: Total simulation time.
- user_action: Optional function for post-processing at each time step.
- version: Solver version ('scalar' or other optimized options).
- save_dir: Directory to save output files.
- boundary: Boundary condition type ('Dirichlet' or 'Neumann').

Returns:
- u, x, t arrays for the final solution and computational grid.
"""
```

12.1.1 Parameters

- `I`: Initial displacement function $I(x) = u(x, 0)$, which defines the starting shape of the wave at $t = 0$.
- `V`: Initial velocity function $V(x) = u_t(x, 0)$, which determines the initial rate of change of the wave at $t = 0$.
- `f`: Source term function $f(x, t)$, an external force applied to the wave at each point x and time t. Setting $f(x, t) = 0$ simulates a free wave with no external influence.
- `q`: Array representing the spatially varying wave speed squared $q(x) = c(x)^2$, where $c(x)$ is the local wave speed at each point in the domain. The variable wave speed introduces inhomogeneity in the medium, affecting wave propagation.
- `L`: Length of the spatial domain, defining the interval $x \in [0, L]$.
- `dt`: Time step size Δt, which, together with the spatial resolution, controls the accuracy and stability of the simulation.
- `C`: Courant number $C = \frac{c\Delta t}{\Delta x}$, a stability factor in numerical wave propagation. It is crucial to set $C \leq 1$ to ensure stability in explicit finite difference schemes.
- `T`: Total simulation time, determining the duration for which the solver computes the wave propagation.
- `user_action`: Optional callback function that executes additional actions at each time step. This is often used for visualization, data collection, or custom analysis. If not specified, the solver runs without additional output at each step.
- `version`: Specifies the solver version, with options such as `'scalar'` for basic implementations or other optimized options for enhanced performance.
- `save_dir`: Directory path for saving any output files, such as images or data logs, if the user action includes file-saving operations.
- `boundary`: Boundary condition type, with options `'Dirichlet'` (fixed boundary) or `'Neumann'` (zero-gradient boundary) for controlling wave behavior at the domain edges.

In the subsequent sections, we delve into each stage of the function, including initialization of solution arrays, application of boundary conditions, and the main time-stepping loop.

12.2 Initialization of the Solution Arrays

In this solver, three arrays are initialized to store the solution at different time levels:

$$u, \quad u_1, \quad \text{and} \quad u_2.$$

Each array has a length equal to the number of spatial points in the domain, ensuring consistent indexing and alignment with the spatial grid defined by q. The purpose of each array is as follows:

- u: Stores the solution at the **current time step**.

- u_1: Stores the solution at the **previous time step**.
- u_2: Stores the solution at **two steps prior**.

This setup enables the solver to implement a time-stepping scheme where the solution at each new time level depends on values from the previous two levels, providing stability and accuracy in the wave equation calculation.

12.2.1 Code Implementation for Initialization

The initialization of these arrays is implemented as follows:

```
# Initialize solution arrays
u = np.zeros(len(q))  # Solution array for the current time step
u_1 = np.zeros(len(q)) # Solution array for the previous time step
u_2 = np.zeros(len(q)) # Solution array for two time steps prior
```

Explanation of the Code

- `u = np.zeros(len(q))`: Creates an array `u` filled with zeros, with a length matching the number of spatial points defined by `q`. This array will hold the solution values at each spatial point for the current time step.
- `u_1 = np.zeros(len(q))`: Creates a second array `u_1` to store the solution at the previous time step, initialized to zero initially. This array will be updated after each time step to hold the most recent solution values, allowing it to serve as the previous time step in the time-stepping scheme.
- `u_2 = np.zeros(len(q))`: A third array `u_2` is created to store the solution two time steps prior. Like `u_1`, it is initialized with zeros and updated at each time step, ensuring that the solver has access to values from two previous time levels.

12.2.2 Role of Each Array in the Time-Stepping Scheme

In the finite difference method for wave equations, the solution at each spatial point and current time step, $u[i]$, depends on values from the previous two time steps (u_1 and u_2). This time-stepping scheme is represented mathematically as:

$$u[i] = 2u_1[i] - u_2[i] + \Delta t^2 \left(\frac{q_{i+1/2}(u_1[i+1] - u_1[i]) - q_{i-1/2}(u_1[i] - u_1[i-1])}{\Delta x^2} + f(x[i], t) \right).$$

After each time step, the arrays are updated:

$$u_2[:] = u_1, \quad u_1[:] = u.$$

This array switching ensures that u_1 and u_2 hold the values from the previous two time levels, preparing them for the next iteration.

12.2.3 Benefits of This Initialization

- **Efficiency**: By reusing these arrays at each time step, memory usage is minimized, as only three arrays are needed, regardless of the number of time steps.
- **Consistency**: Initializing each array to zero provides a stable starting point and avoids potential errors from undefined values, particularly if initial conditions do not set all elements explicitly.

In summary, the initialization of these solution arrays allows for efficient and accurate propagation of the wave equation over time, using a stable and consistent data structure.

12.3 Implementation of the First Time Step

The first time step in the solver is treated separately to incorporate the initial conditions for both displacement and velocity. Specifically, the initial conditions are given as:

$$u(x,0) = I(x) \quad \text{and} \quad u_t(x,0) = V(x),$$

where $I(x)$ defines the initial displacement profile of the wave, and $V(x)$ specifies the initial velocity at each spatial point x in the domain. The goal for the first time step is to calculate the updated displacement values $u[i]$ at each spatial point i based on these initial conditions.

12.3.1 Mathematical Update Formula

To update the displacement u for each spatial point i at the first time step, we use the following formula:

$$u_i = u_1[i] + \Delta t \cdot V(x_i) + \frac{\Delta t^2}{2}\left(\frac{q_{i+1/2}(u_1[i+1] - u_1[i])}{\Delta x^2} - \frac{q_{i-1/2}(u_1[i] - u_1[i-1])}{\Delta x^2}\right) + \frac{\Delta t^2}{2} f(x_i, t_0),$$

where: - Δt is the time step size, - $V(x_i)$ is the initial velocity at x_i, - $f(x_i, t_0)$ is the source term at the initial time $t = 0$, - $q_{i+1/2}$ and $q_{i-1/2}$ are averaged values of $q(x) = c(x)^2$ between neighboring points, defined as:

$$q_{i+1/2} = \frac{q[i] + q[i+1]}{2}, \quad q_{i-1/2} = \frac{q[i] + q[i-1]}{2}.$$

Averaging $q(x)$ between points i and $i+1$ reduces artificial amplification and improves numerical stability by smoothing transitions in wave speed across the grid.

12.3.2 Code Implementation for the First Time Step

```
                    # First time step using initial velocity V(x) and initial displacement I(x)
                    for i in range(1, len(q) - 1):
                        avg_q_right = (q[i] + q[i + 1]) / 2 # Average q between points i and i+1
                        avg_q_left = (q[i] + q[i - 1]) / 2 # Average q between points i and i-1
                        u[i] = (u_1[i] + dt * V(x[i]) +
                                0.5 * dt**2 * (avg_q_right * (u_1[i + 1] - u_1[i]) / dx**2 -
                                avg_q_left * (u_1[i] - u_1[i - 1]) / dx**2) +
                                0.5 * dt**2 * f(x[i], t[0]))
```

12.3.3 Explanation of the Code

1. **Averaging of q Values**: - `avg_q_right` and `avg_q_left` calculate the averaged values of $q(x)$ between spatial points i and $i+1$, and i and $i-1$, respectively. These averages, $q_{i+1/2}$ and $q_{i-1/2}$, are used to control the spatial derivatives in the update formula.

2. **Update Formula**: - The displacement $u[i]$ at each point i is updated based on the previous displacement $u_1[i]$, the initial velocity $V(x[i])$, and the source term $f(x[i], t[0])$. - The term `0.5 * dt**2` multiplies the spatial derivatives and source term to account for the first time step in a way that includes contributions from both displacement and velocity.

3. **Boundary Handling**: - The loop runs from `i = 1` to `i = len(q) - 2` to exclude the boundary points, where specific boundary conditions (Dirichlet or Neumann) are applied separately.

12.4 Time-Stepping Loop

The time-stepping loop is the core of the solver, iterating over each time step from $n = 1$ to N_t, where N_t is the total number of time steps. At each time step, the solver updates the wave displacement u across the spatial domain based on values from the previous two time steps. This process continues until the entire simulation duration T is covered.

12.4.1 Code Implementation of the Time-Stepping Loop

```
# Main time-stepping loop
for n in range(1, Nt):
    for i in range(1, len(q) - 1):
        # Averaged q values to reduce artificial amplification
        avg_q_right = (q[i] + q[i + 1]) / 2
        avg_q_left = (q[i] + q[i - 1]) / 2
        u[i] = (2 * u_1[i] - u_2[i] +
            dt**2 * (avg_q_right * (u_1[i + 1] - u_1[i]) -
            avg_q_left * (u_1[i] - u_1[i - 1])) / dx**2 +
            dt**2 * f(x[i], t[n]))
```

12.4.2 Boundary Conditions

The solver supports two types of boundary conditions: Dirichlet and Neumann, which affect the behavior of the wave at the edges of the domain.

- **Dirichlet Boundary Condition**: Sets the displacement to zero at both boundaries, simulating fixed ends. This condition is implemented as:

$$u[0] = 0 \quad \text{and} \quad u[N_x] = 0.$$

- **Neumann Boundary Condition**: Sets a zero-gradient (free) boundary condition, allowing the displacement to vary freely at the edges. This is implemented by setting:

$$u[0] = u[1] \quad \text{and} \quad u[N_x] = u[N_x - 1].$$

  This condition is suitable for open ends where the wave reflects with no phase change.

Code for Boundary Conditions :

```
                    # Apply boundary conditions
                    if boundary == 'Dirichlet':
                           u[0] = u[-1] = 0 # Fixed displacement at both boundaries
                    elif boundary == 'Neumann':
                           u[0] = u[1]      # Zero-gradient (free) at the left boundary
                           u[-1] = u[-2]    # Zero-gradient (free) at the right boundary
```

12.5 Updating Previous Solutions

In each iteration of the time-stepping loop, after computing $u[i]$ for all spatial points i, the solver updates the solution arrays to prepare for the next time step. This is achieved by shifting the values of the displacement arrays to represent the previous two time levels. The update rule is given by:

$$u_2[:] = u_1, \quad u_1[:] = u.$$

Code Implementation :

```
                    # Update previous solutions for the next time step
                    u_2[:] = u_1 # Shift the previous time step solution to u_2
                    u_1[:] = u   # Shift the current solution to u_1
```

Explanation of the Code 1. **Assignment of Arrays**: The command `u_2[:] = u_1` assigns the values of `u_1` (the previous time step solution) to `u_2`. Similarly, `u_1[:] = u` assigns the current solution in `u` to `u_1`. This step ensures that the arrays u_1 and u_2 are ready for use in the next iteration of the time-stepping loop.

12.6 Return Values

At the end of the simulation, the function returns:

- `u`: Final solution array for $u(x, T)$.
- `x` and `t`: Arrays of spatial and temporal grid points, providing the computational grid information.

This solver implementation allows flexibility for simulating waves with variable speeds across a spatial domain, capturing the effects of inhomogeneities in the medium.

13 Example: Wave Propagation with Variable Velocity and Periodic Boundary Pulses

This example demonstrates the simulation of wave propagation in a one-dimensional domain with variable wave speed. The simulation models a domain split into two regions with distinct wave velocities, separated by an impedance boundary. Additionally, a periodic boundary pulse is applied at the left end of the domain, mimicking a recurring wave input. The principles governing wave

reflection and transmission at impedance boundaries can be explored in depth through interactive demonstrations, such as those available at Penn State's Acoustics website. For a detailed overview, refer to website*.

13.1 Problem Setup

The domain length $L = 1.0$ is divided into 100 spatial points for adequate resolution, resulting in a spatial step size of $\Delta x = \frac{L}{N_x}$. The simulation runs for a total time $T = 1$, which allows observation of multiple wave reflections across the domain.

Wave Speeds and Medium Properties The domain is split into two regions with different wave speeds and densities:

- **Medium 1**: From $x = 0$ to $x = L/3$, the wave speed is $c_1 = 1$ with a density $\rho_1 = 1.0$.
- **Medium 2**: From $x = L/3$ to $x = L$, the wave speed is $c_2 = 0.5$ and the density is $\rho_2 = 4.0$.

These values create an impedance difference, resulting in partial reflection and transmission of the wave at the boundary. Reflection and transmission coefficients are calculated based on the impedance mismatch between the two media:

$$\text{Reflection Coefficient} = \frac{Z_1 - Z_2}{Z_1 + Z_2}, \quad \text{Transmission Coefficient} = \frac{2Z_1}{Z_1 + Z_2},$$

where $Z_1 = \rho_1 c_1$ and $Z_2 = \rho_2 c_2$.

Boundary Pulse At the left boundary, a sinusoidal pulse is applied periodically with a duration of 0.16 seconds and a period of 2.0 seconds. This pulse acts as a recurring wave source, simulating continuous energy input into the domain.

13.2 Code Implementation

This section breaks down the implementation of the variable velocity wave equation solver, focusing on key components such as the initial and boundary conditions, variable wave speed, and the solution and visualization setup.

13.3 Problem Setup and Parameters

This simulation models wave propagation across a heterogeneous one-dimensional domain, where two distinct materials create an impedance boundary. The purpose of this setup is to observe wave behavior, including reflection and transmission, as it encounters a boundary with different material properties. The following parameters define the spatial resolution, material properties, and wave behavior:

```
# ==============================
# Problem Setup and Parameters
# ==============================
# Length of the domain and the number of spatial points
```

```
L = 1.0                    # Total length of the domain
Nx = 100                    # Number of spatial points for resolution

# Courant number, which is essential for stability in wave equations.
# Here, we set C = 1 for the maximum stability within homogeneous regions.
C = 1

# Total simulation time, which will determine how long we observe the wave motion.
T = 1

# Periodicity of pulses at the left boundary; we emit a pulse every ‘pulse_period‘
    seconds.
pulse_period = 2.0

# Duration of each pulse; each pulse lasts ‘pulse_duration‘ seconds.
pulse_duration = 0.16
```

13.3.1 Domain and Spatial Resolution

The length of the domain $L = 1.0$ is divided into $N_x = 100$ spatial points. This resolution ensures that the wave behavior is well-captured across the domain. The spatial step size Δx is calculated as:

$$\Delta x = \frac{L}{N_x}.$$

13.3.2 Courant Number and Stability

The Courant number C is set to 1, which is the maximum stability condition for homogeneous wave propagation. This value is crucial in wave simulations to ensure numerical stability and accurate wave behavior. The time step size Δt is then determined by:

$$\Delta t = C \frac{\Delta x}{\max(c_1, c_2)},$$

where c_1 and c_2 represent the wave speeds in each material.

13.3.3 Simulation Time and Boundary Pulse Parameters

The total simulation time $T = 1$ allows sufficient observation of the wave motion, including reflections at the impedance boundary. A periodic pulse is introduced at the left boundary:

- **Pulse Period** (`pulse_period`): Each pulse is emitted every 2 seconds, simulating a recurring energy input.
- **Pulse Duration** (`pulse_duration`): Each pulse lasts 0.16 seconds, creating a short, sharp wave that travels through the domain.

13.3.4 Directory for Saving Animation Frames

The generated frames of the simulation are saved in the directory specified by `save_dir`. This setup allows for easy post-processing into animations.

```
# Directory to save generated frames for the animation
save_dir = r'path/to/moving_variable_medium_wave_simulation'

# Ensure that the directory for saving images exists
if not os.path.exists(save_dir):
       os.makedirs(save_dir)
```

13.3.5 Material Properties and Wave Speeds

The domain is split into two media, each with different density and wave speed values:

- **Medium 1**: Density $\rho_1 = 1.0$ and wave speed $c_1 = 1$.
- **Medium 2**: Density $\rho_2 = 4.0$ and wave speed $c_2 = 0.5$, making it eight times denser and with a slower wave speed.

This density and wave speed variation creates an impedance difference at the boundary, leading to partial reflection and transmission of waves. The impedance Z for each medium is calculated as:

$$Z_1 = \rho_1 c_1, \quad Z_2 = \rho_2 c_2.$$

13.3.6 Reflection and Transmission Coefficients

The reflection and transmission coefficients quantify the portion of the wave energy that is reflected or transmitted at the impedance boundary:

$$\text{Reflection Coefficient} = \frac{Z_1 - Z_2}{Z_1 + Z_2}, \quad \text{Transmission Coefficient} = \frac{2Z_1}{Z_1 + Z_2}.$$

These coefficients are calculated based on the impedance mismatch between the two media, ensuring that the simulation accurately reflects wave interactions at the boundary.

```
# ================================
# Material Properties and Wave Speeds
# ================================
# Density and wave speed for each medium
rho1 = 1.0 # Density in medium 1
rho2 = 4.0 # Density in medium 2, 4 times denser than medium 1
c1 = 1     # Wave speed in medium 1
c2 = 0.5   # Wave speed in medium 2, slower in denser medium

# Derived values
dx = L / Nx                # Spatial step size based on the number of points
dt = C * dx / max(c1, c2)  # Time step size derived from the Courant number for
     stability
Z1 = rho1 * c1             # Impedance in medium 1
Z2 = rho2 * c2             # Impedance in medium 2

# Reflection and Transmission coefficients based on impedance mismatch
reflection_coeff = (Z1 - Z2) / (Z1 + Z2)
transmission_coeff = (2 * Z1) / (Z1 + Z2)
```

13.3.7 Initial and Boundary Conditions

The initial displacement $I(x)$ and initial velocity $V(x)$ are set to zero, simulating a stationary start where no wave motion exists initially. The source term $f(x,t)$ is also set to zero, meaning there is no external force within the domain. The left boundary is given a periodic pulse defined by the function $U_0(t)$, which generates a sinusoidal pulse every 2 seconds, mimicking a recurring wave input into the domain.

```
# Initial displacement and velocity (both set to zero for a stationary initial state)
def initial_displacement(x):
    return np.zeros_like(x)

def initial_velocity(x):
    return np.zeros_like(x)

# Define a periodic pulse at the left boundary that triggers every `pulse_period`
def U_0(t):
    """
    Boundary condition at x=0, generating a sinusoidal pulse at regular intervals.
    Only active for `pulse_duration` within each `pulse_period`.
    """
    pulse_time = t % pulse_period
    return 0.25 * np.sin(6 * np.pi * pulse_time) if pulse_time <= pulse_duration else 0
```

Here, the function `U_0(t)` generates a sinusoidal wave at $x = 0$ every `pulse_period` seconds, but only for a limited `pulse_duration`. This ensures periodic energy input without continuous oscillation, allowing for controlled wave pulses into the domain.

13.3.8 Variable Wave Speed Across the Domain

The function `variable_wave_speed` defines the squared wave speed $q(x) = c(x)^2$ across the domain. This variable wave speed reflects the physical setup, where wave speed changes at $x = L/3$, representing a boundary with impedance mismatch.

```
def variable_wave_speed(x):
    """
    Function to define q(x) = c(x)^2 across the domain.
    The wave speed changes at x = L/3, representing an impedance discontinuity.
    """
    midpoint = L / 3
    return np.where(x < midpoint, c1**2, c2**2)
```

In this implementation, the wave speed squared $q(x)$ is set to c_1^2 for the first $\frac{L}{3}$ of the domain and c_2^2 for the remainder. This change in $q(x)$ causes partial reflection and transmission of the wave at the boundary, demonstrating the effect of an impedance discontinuity.

13.3.9 Solution and Visualization Setup

To capture and visualize the wave's behavior, each time step's solution is saved both for generating a GIF and for creating an HTML animation. This allows the observation of wave reflections,

transmissions, and the impact of the periodic pulse at the boundary.

```
# Array to store solution data for HTML animation
results = []

# Capture results at each time step for HTML animation
def capture_results(u, x, t, n):
       results.append((u.copy(), t[n]))

# Combined user action to capture results and save images for GIF creation
def combined_user_action(u, x, t, n):
       capture_results(u, x, t, n) # Capture data for HTML animation
       save_wave_image(u, x, t, n, C, save_dir=save_dir, ymin=-0.6, ymax=0.7) #
            Save image for GIF
```

The `capture_results` function appends each time step's solution u and time $t[n]$ to the `results` array, allowing for HTML animation generation after the simulation completes. The `combined_user_action` function combines this with image saving for GIF creation, providing a comprehensive visualization of the simulation.

13.3.10 Setting Up the Spatial Grid and Variable Wave Speed

To model wave propagation through a medium with varying wave speed, we define a spatial grid and compute $q(x) = c(x)^2$, which represents the square of the wave speed. This step ensures that the solver can accommodate spatially varying properties, allowing for more realistic simulations where waves encounter different media with distinct physical properties.

Spatial Grid and Wave Speed Function The spatial domain is divided into $N_x + 1$ points across a length L, creating a grid array x with coordinates evenly spaced by $\Delta x = L/N_x$:

$$x = \texttt{np.linspace(0, L, Nx + 1)}.$$

The variable wave speed $q(x)$ is computed using the `variable_wave_speed` function, which applies different wave speeds in distinct regions, simulating an impedance boundary within the domain.

```
# Spatial grid across the domain and calculating q(x) based on wave speed
x = np.linspace(0, L, Nx + 1)
q = variable_wave_speed(x)
```

13.3.11 Solver Wrapper for Variable Velocity

The `solve_wave_equation_with_variable_velocity` function is a wrapper designed to handle the wave equation with variable wave speed $q(x) = c(x)^2$ and to apply a periodic boundary condition $U_0(t)$ at the left boundary. This setup is essential for simulating wave propagation in a heterogeneous medium where the wave speed changes across the spatial domain.

Function Parameters :

```
# ================================
# Solver Wrapper for Variable Velocity
# ================================

```

```
def solve_wave_equation_with_variable_velocity(I, V, f, q, L, dt, C, T, U_0,
    user_action=None, version='scalar', boundary='Dirichlet'):
"""
Wrapper to solve the wave equation with variable wave velocity `q(x)` and a moving
    boundary condition `U_0`.

Parameters:
- I: Initial displacement function
- V: Initial velocity function
- f: Source term function
- q: Spatially varying wave speed squared (q(x) = c(x)^2)
- L: Length of the domain
- dt: Time step size
- C: Courant number
- T: Total simulation time
- U_0: Left boundary pulse function
- user_action: Action to take at each time step (e.g., save images)
"""
```

The function accepts several parameters:

- `I`, `V`, and `f`: These are the initial displacement, initial velocity, and source term functions, respectively.
- `q`: An array representing the squared wave speed $q(x) = c(x)^2$ across the spatial domain.
- `L`, `dt`, `C`, and `T`: These parameters define the domain length L, time step size Δt, Courant number C, and total simulation time T, each critical for maintaining stability and controlling the simulation duration.
- `U_0`: A function that defines the left boundary condition, applying a pulse at periodic intervals.
- `user_action`: An optional action executed at each time step, allowing for tasks such as saving or visualizing the current solution.

Setting Up the Time and Spatial Grid The function first calculates the time grid and spatial grid required for solving the wave equation:

- `Nt` is the number of time steps, calculated by dividing the total simulation time T by the time step Δt.
- `t` is the time grid, created with evenly spaced points from $t = 0$ to $t = T$.
- `dx` is the spatial step, calculated as $\Delta x = L/N_x$.
- `x` is the spatial grid, a sequence of points from $x = 0$ to $x = L$.

```

    Nt = int(round(T / dt)) # Number of time steps based on T and dt
    t = np.linspace(0, Nt * dt, Nt + 1) # Time grid
    dx = L / Nx             # Reconfirming spatial step
    x = np.linspace(0, L, Nx + 1) # Spatial grid
```

Boundary Condition with Wrapped User Action The function defines a nested function, `wrapped_user_action`, to apply the left boundary condition at each time step. Specifically, $u[0] = U_0(t[n])$ is set at the beginning of each time step to simulate an oscillating pulse at the left boundary. This wrapped action applies the boundary condition before calling the main `user_action` function, if provided.

```
                    def wrapped_user_action(u, x, t, n):
                        u[0] = U_0(t[n]) # Apply the boundary condition at each time step
                        if user_action:
                            user_action(u, x, t, n)
```

Executing the Variable Velocity Solver The last part of the wrapper function calls `solve_wave_equation_variable` passing all the required parameters, including the boundary-modified `wrapped_user_action` to manage the oscillating boundary condition. The solver then runs over each time step and spatial point to compute the wave propagation across a domain with variable wave speed.

```
                    solve_wave_equation_variable_velocity(
                        I=I, V=V, f=f, q=q, L=L, dt=dt, C=C, T=T,
                        user_action=wrapped_user_action,
                        version=version,
                        boundary=boundary
                        )
```

13.3.12 Execution of the Solver with Variable Velocity

Finally, we execute the solver with variable velocity and the defined boundary conditions. The `solve_wave_equation_variable_velocity` function is wrapped in a custom function `solve_wave_equation_with_variab` to integrate the periodic boundary pulse at the left boundary.

```
# Execute the solver with variable velocity and periodic pulse at the boundary
solve_wave_equation_with_variable_velocity(
        I=initial_displacement,
        V=initial_velocity,
        f=source_term,
        q=q,
        L=L,
        dt=dt,
        C=C,
        T=T,
        U_0=U_0,
        user_action=combined_user_action,
        version='scalar',
        boundary='Dirichlet'
        )
```

In this call, `solve_wave_equation_with_variable_velocity` runs the simulation, applying the periodic pulse $U_0(t)$ at the left boundary and using the combined user action to store results and save images. The resulting animation captures how the wave propagates through the domain, reflects at the impedance boundary, and responds to the periodic pulses.

13.4 Link to Code, Simulation Results, and Theory/Experiments

Code Link:

Moving Wave with Heterogenious String - Variable Velocity Problem

Animation Link:

Animations of Moving Wave - Variable Velocity Problem

Theory Link:

Reflection from an impedance discontinuity