Detailed Explanation of the Python Code: Wave Equation Solver

Petar Bosnic

October, 2024

1 Introduction

This document explains the Python code for solving the 1D wave equation using various numerical methods, including scalar and vectorized implementations. The wave equation in its general form is:

$$u_{tt} = c^2 u_{xx} + f(x,t), \tag{1}$$

where $u(x,t)$ is the displacement at position x and time t, c is the wave propagation speed, and $f(x,t)$ is an optional source term. The solver implements this equation over a spatial domain $(0, L)$ and a time domain $(0, T)$.

2 Understanding Time and Spatial Stepping in the Solver

To better understand how the wave equation solver updates values over time and space, we can visualize the computation as a grid. Each point on the grid represents a value of $u(x,t)$, where x is a spatial point, and t is a time step. The solver uses finite difference methods to compute values at each time step based on the previous ones.

2.1 2D Grid for Time and Spatial Stepping

The grid below represents the spatial domain on the x-axis and the time domain on the t-axis. The wave equation is solved iteratively, meaning that values at the next time step depend on the current and previous time 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

Explanation:

- Each row represents a time step t_n.
- Each column represents a spatial point x_i.
- The value u_i^n represents the solution at spatial point x_i and time step t_n.
- The solver updates the values u_i^n using the finite difference scheme, based on previous time steps.

2.2 Finite Difference Update Rule

At each time step, the values u_i^n are updated using the values from previous time steps. For example, using the centered finite difference method, the update rule is:

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

Explanation of Terms:

- u_i^{n+1}: The value at the next time step.
- u_i^n: The value at the current time step.
- u_i^{n-1}: The value from the previous time step.
- C^2: The square of the Courant number, which controls the stability of the numerical scheme.
- u_{i-1}^n, u_{i+1}^n: The neighboring spatial points used for computing the spatial derivative.

2.3 Visualization of the Stepping Process

The update rule uses values from the current time step and the previous time step to compute the values at the next time step. Visually, the grid looks like this:

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

Explanation:

- The solver uses the values at u_i^n, u_i^{n-1}, and the neighboring spatial points u_{i-1}^n, u_{i+1}^n to compute the new value u_i^{n+1}.
- This process is repeated for all spatial points and for each time step until the entire grid is filled with the solution.

2.4 Conclusion

This section provides a visual representation of the time and spatial stepping used in the wave equation solver. The finite difference method iteratively updates the grid, and the Courant number ensures the stability of the solution. This grid visualization helps to understand how each point in the solution is computed based on previous values and neighboring points.

3 Code

3.1 Function Definition and Parameters

The Python function `solve_wave_equation` is responsible for solving the wave equation numerically. Below is the function definition, where each parameter is described:

```
1  def solve_wave_equation(I, V, f, c, L, dt, C, T, user_action=None,
       version='scalar', save_dir=None):
2      """
3      Unified solver: Solve u_tt = c^2 * u_xx + f on (0, L) x (0, T] using scalar or
           vectorized approaches.
4      """
```

Parameters:

- `I`: The initial displacement function $I(x)$ that specifies the wave's initial configuration.
- `V`: The initial velocity function $V(x)$ that defines the initial speed of the wave.
- `f`: The source term $f(x,t)$, which represents external forces acting on the system. This can be zero.
- `c`: The wave speed, which governs how fast the wave propagates through the domain.
- `L`: The length of the spatial domain $(0, L)$.
- `dt`: The time step size.
- `C`: The Courant number, defined as $C = \frac{cdt}{dx}$. This value controls the stability of the solver.
- `T`: The total time for the simulation.
- `user_action`: A user-defined function that can be applied at each time step (optional).
- `version`: Specifies whether the solver uses a scalar or vectorized implementation.
- `save_dir`: The directory to save results, such as images of the wave, during the simulation.

3.2 Grid Setup and Courant Number

The grid setup initializes the computational domain in both space and time. The Courant number C is squared for use later in the finite difference scheme.

```
            if save_dir is None:
            save_dir = os.getcwd() # Save images in the current working directory

            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

            print("Courant number is:", C)
```

Explanation:

- The number of time steps `Nt` is computed based on the total time T and the time step size dt.
- The spatial grid points `x` and the time grid points `t` are generated.
- The spatial step size `dx` is calculated using the Courant number C to ensure stability.
- `C2` is the square of the Courant number C, which will be used in the time-stepping loop.

3.3 Initialization of Solution Arrays

Here, the code initializes arrays to store the solution at the current time step (`u`), the previous time step (`u_1`), and two time steps ago (`u_2`).

```
u = np.zeros(Nx + 1)
u_1 = np.zeros(Nx + 1)
u_2 = np.zeros(Nx + 1)

t0 = time.time()

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

Explanation:

- `u`, `u_1`, and `u_2` are initialized as arrays of zeros, representing the wave at different time steps.
- `u_1` is filled with the initial displacement function $I(x)$, which represents the initial configuration of the wave.
- The time `t0` is recorded to calculate the CPU time later.

3.4 First Time Step Calculation

The first time step is special because it requires both the initial displacement and velocity to initialize the solution.

```
n = 0
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[n])
u[0] = 0
u[Nx] = 0
```

Explanation:

- The code computes the new solution at the first time step using a combination of the initial displacement `u_1`, velocity `V`, and the source term `f(x,t)`.
- A second-order accurate finite difference scheme is used.
- Boundary conditions are applied by setting `u[0]` and `u[Nx]` to zero (Dirichlet boundary conditions).

3.5 Time-Stepping Loop

The core of the numerical solver is the time-stepping loop, which advances the solution in time using the scalar or vectorized approach.

```
for n in range(1, Nt):
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])

elif version == 'vectorized':
f_a = f(x, t[n])
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]
```

Explanation:

- In the scalar version, each point in the domain is updated sequentially using the finite difference method.
- In the vectorized version, the entire domain is updated simultaneously using NumPy's slicing, which makes the code more efficient.
- The source term `f` is applied at each time step.

3.6 Boundary Conditions and User Action

Boundary conditions are enforced after each time step, and the optional `user_action` is called to perform actions such as saving images or logging results.

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

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

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

Explanation:

- Boundary conditions are applied at every time step to ensure the wave behavior is consistent with physical constraints.
- `user_action` allows flexibility for additional tasks such as saving the current state of the solution or visualizing the wave propagation.
- The arrays `u_2` and `u_1` are updated to prepare for the next time step.

3.7 Finalizing the Simulation and CPU Time

The simulation ends by calculating the CPU time, which helps in evaluating the performance of the solver.

```
            cpu_time = time.time() - t0
            print(f"CPU time (s) of {version} solver: {cpu_time:.5f}")
            return u, x, t, cpu_time
```

Explanation:

- The CPU time for the entire simulation is calculated and printed.
- The final wave solution `u`, the spatial grid `x`, the time grid `t`, and the CPU time are returned.

3.8 Conclusion

This Python function provides a unified approach to solving the 1D wave equation using either scalar or vectorized methods. The use of the Courant number ensures stability, and the code is flexible, allowing for different types of user-defined actions during the simulation.

4 Visualization of the Simulation

In addition to solving the wave equation, the code provides functionality to visualize the wave propagation by saving images at each time step and generating a GIF of the entire simulation. The following functions handle the visualization process.

4.1 Saving Wave Images

The `save_wave_image` function saves a snapshot of the wave at each time step. The Courant number C is included in the plot title for reference.

```
def save_wave_image(u, x, t, n, C, save_dir='wave_images'):
"""Save the wave at each time step as an image, with Courant number in the title."""
if not os.path.exists(save_dir):
os.makedirs(save_dir)

plt.figure(figsize=(8, 4))
plt.plot(x, u, label=f"t = {t[n]:.5f}")
plt.ylim(-0.01, 0.01)
plt.xlim(0, max(x))
plt.xlabel('x')
plt.ylabel('u(x,t)')

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

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

# Debugging message
print(f"Saving image: {filename}")

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

Explanation:

- The function creates a directory (`save_dir`) for storing the images, if it does not already exist.
- A plot of the wave at the current time step is generated, and the Courant number is displayed in the title.
- Each image is saved with a filename that includes the time step number, ensuring proper ordering.

4.2 Generating a GIF

The `generate_gif_from_images` function creates an animated GIF from the saved images, allowing you to visualize the entire wave propagation over time.

```
def generate_gif_from_images(image_folder='wave_images', gif_name='wave_animation.gif',
    duration=0.1):
"""Generate a GIF from the images stored in a folder."""
images = []
image_files = sorted([img for img in os.listdir(image_folder) if img.endswith(".png")])

if not image_files:
print("Warning: No images found in the specified folder to create a GIF.")
return

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

gif_path = os.path.join(image_folder, gif_name)
imageio.mimsave(gif_path, images, duration=duration)
print(f"GIF saved as {gif_path}")
```

Explanation:

- The function gathers all the PNG files in the specified folder and sorts them to ensure proper sequence.
- The images are combined into a GIF using `imageio.mimsave`, with a specified frame duration for smooth animation.
- The resulting GIF is saved in the folder, and its path is printed for reference.

4.3 Conclusion

This section of the code provides functionality to visually track the wave's propagation during the simulation. The ability to generate a GIF allows for easy visualization of the wave dynamics over time.

5 Main Program

This section provides the main program that sets up the wave equation simulation, calls the solver, and generates visualizations. The program is divided into logical steps for clarity.

5.1 Importing Required Functions

First, the necessary functions from the `wave_eq_solver` module are imported.

```
import numpy as np
from wave_eq_solver import solve_wave_equation, save_wave_image,
    generate_gif_from_images
```

Explanation:

- `solve_wave_equation`: Solves the wave equation.
- `save_wave_image`: Saves images of the wave propagation at each time step.
- `generate_gif_from_images`: Generates a GIF from saved wave images.

5.2 Defining Parameters

The parameters for the wave equation are defined based on the problem of a vibrating guitar string.

```
if __name__ == "__main__":

        # Parameters for the wave equation
        L = 0.75              # Length of the string (in meters)
        x0 = 0.8 * L          # Initial displacement position
        a = 0.005             # Maximum amplitude of the initial displacement (in meters)
        freq = 440            # Frequency of the wave (in Hertz)
        wavelength = 2 * L     # Wavelength of the wave
        c = freq * wavelength  # Wave speed
        omega = 2 * np.pi * freq # Angular frequency
        num_periods = 1        # Number of periods to simulate
        T = 2 * np.pi / omega * num_periods # Total simulation time (in seconds)
        C = 0.2               # Courant number
        dt = L / 50.0 / c     # Time step size (in seconds)
```

Explanation:

- `L`: The length of the string is set to 0.75 meters.

- `freq`: The frequency of the string is 440 Hz (standard A note).
- `c`: The wave speed is computed using the wave frequency and wavelength.
- `C`: The Courant number controls the stability of the simulation. It is set to 0.2 for this example.
- `T` and `dt`: The total simulation time is set to cover one period of oscillation, and `dt` is computed for proper resolution.

5.3 Defining Initial Conditions and Source Term

The initial displacement, velocity, and source term functions are defined. These are specific to the vibrating string problem.

```
        # Problem-specific functions for the guitar 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)
```

Explanation:

- `initial_displacement`: Models the initial pluck of the string, where the displacement is a triangular shape.
- `initial_velocity`: Sets the initial velocity to zero (the string starts at rest).
- `source_term`: No external forces are applied in this case, so the source term is set to zero.

5.4 Choosing the Solver

The user is prompted to choose between the scalar or vectorized solvers. This allows the user to control the performance and behavior of the simulation.

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

Explanation:

- The user can choose between three solver versions: `scalar`, `vectorized`, and `vectorized2`.
- The `scalar` version uses a simple loop, while the `vectorized` versions use efficient array operations for faster computations.

5.5 Solving the Wave Equation

The `solve_wave_equation` function is called to solve the wave equation based on the user's choice of solver.

```
        # Use the unified solver based on user choice
        if solver_choice in ["scalar", "vectorized", "vectorized2"]:
        solve_wave_equation(
        initial_displacement,
        initial_velocity,
        source_term,
        c,
        L,
        dt,
        C,
        T,
        user_action=lambda u, x, t, n, save_dir: save_wave_image(u, x, t, n, C, save_dir), #
            Pass C via lambda
        version=solver_choice,
        save_dir='wave_images'
        )
        else:
        print("Invalid choice. Please choose 'scalar', 'vectorized', or 'vectorized2'.")
```

Explanation:

- `solve_wave_equation`: This function solves the wave equation using the initial conditions, velocity, and source term defined earlier.
- `user_action`: A lambda function is passed to save images of the wave at each time step. The Courant number C is passed to the `save_wave_image` function.
- `version`: The user's choice of solver (`scalar`, `vectorized`, or `vectorized2`) determines which method is used to solve the equation.

5.6 Generating a GIF

After solving the wave equation and saving the images, a GIF is generated from the saved images to visualize the wave propagation over time.

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

Explanation:

- The function `generate_gif_from_images` creates an animated GIF of the wave propagation. The frame duration is set to 0.1 seconds.
- The images are taken from the `wave_images` folder, where they were saved during the simulation.

6 Conclusion

This section provides the main program for simulating the wave equation. It allows the user to select between different solvers, visualize the wave propagation as images, and generate an animated GIF to capture the entire process. The solver is flexible, allowing custom initial conditions, and the visualization helps track the evolution of the wave over time.