Numerical Solutions to the Wave Partial Differential Equation

Using the Finite Difference Method

Team Name: **Numerical Engineers**
Team Members:
SalahDin Rezk
Karim Elbahrwy
Hassan Rashwan

Contents

List of Figures

Abstract

This project focuses on solving the two-dimensional wave equation—a second-order partial differential equation—using the finite difference method, a numerical approach that discretizes the domain into a grid and approximates derivatives using finite differences. Implemented in Python, the project showcases the method through visualization of the wave propagation over a two-dimensional domain. The study is divided into three main parts: an introduction to the wave equation and finite difference method, the implementation details, and the presentation of results and visualizations.

1 Introduction

1.1 Wave Equation

The wave equation is a second-order partial differential equation that describes the behavior of waves in a medium. It is given by

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \tag{1}$$

where u is the wave function, c is the wave speed, and ∇^2 is the Laplacian operator, defined as the sum of the second partial derivatives of u with respect to the spatial coordinates.

For a two-dimensional wave equation, it simplifies to

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{2}$$

This equation models waves on a membrane. While analytical solutions exist for simple boundary conditions, numerical methods such as the finite difference method are used for more complex scenarios.

1.2 Finite Difference Method

The finite difference method is a numerical technique for solving partial differential equations by discretizing the domain into a grid and approximating derivatives using finite differences. For the two-dimensional wave equation, the domain is discretized into a grid with spatial steps Δx and Δy, and temporal step Δt.

Using central difference quotients, we approximate the second partial derivatives:

$$\begin{aligned}
\frac{\partial^2 u}{\partial x^2} &\approx \frac{1}{\Delta x^2}[u(x+\Delta x, y, t) - 2u(x, y, t) + u(x-\Delta x, y, t)] \\
\frac{\partial^2 u}{\partial y^2} &\approx \frac{1}{\Delta y^2}[u(x, y+\Delta y, t) - 2u(x, y, t) + u(x, y-\Delta y, t)] \\
\frac{\partial^2 u}{\partial t^2} &\approx \frac{1}{\Delta t^2}[u(x, y, t+\Delta t) - 2u(x, y, t) + u(x, y, t-\Delta t)]
\end{aligned}$$

Substituting these into the wave equation gives us:

$$\frac{c^2}{\Delta x^2}[u(x+\Delta x, y, t) - 2u(x, y, t) + u(x-\Delta x, y, t)] + \frac{c^2}{\Delta y^2}[u(x, y+\Delta y, t) - 2u(x, y, t) + u(x, y-\Delta y, t)] = \frac{1}{\Delta t^2}[u(x, y, t+\Delta t) - 2u(x$$

Solving for $u(x, y, t+\Delta t)$, we get:

$$u_{i,j,k+1} = 2(1 - \lambda_x^2 - \lambda_y^2)u_{i,j,k} + \lambda_x^2(u_{i+1,j,k} + u_{i-1,j,k}) + \lambda_y^2(u_{i,j+1,k} + u_{i,j-1,k}) - u_{i,j,k-1}$$

where $\lambda_x = \frac{c\Delta t}{\Delta x}$ and $\lambda_y = \frac{c\Delta t}{\Delta y}$.

1.3 Stability and Convergence

The stability condition for the finite difference method applied to the wave equation is given by the Courant-Friedrichs-Lewy (CFL) condition:

$$\lambda_x^2 + \lambda_y^2 \leq 1$$

Ensuring this condition prevents numerical instabilities and guarantees that the solution remains stable. Convergence implies that as Δx, Δy, and Δt approach zero, the numerical solution approaches the exact solution of the wave equation.

2 Methodology

2.1 Discretization of the Wave Equation

The spatial domain is discretized into a grid with points (x_i, y_j) where $x_i = i\Delta x$ and $y_j = j\Delta y$, and the temporal domain into points $t_k = k\Delta t$. The discretized wave equation for a two-dimensional domain is:

$$u_{i,j,k+1} = 2(1 - \lambda_x^2 - \lambda_y^2)u_{i,j,k} + \lambda_x^2(u_{i+1,j,k} + u_{i-1,j,k}) + \lambda_y^2(u_{i,j+1,k} + u_{i,j-1,k}) - u_{i,j,k-1}$$

2.2 Initial and Boundary Conditions

For a membrane with fixed boundaries, the boundary conditions are:

$$u(0, y, t) = u(L, y, t) = u(x, 0, t) = u(x, L, t) = 0 \quad \text{for} \quad t > 0$$

The initial conditions are:

$$u(x, y, 0) = f(x, y) \quad \text{and} \quad \frac{\partial u}{\partial t}(x, y, 0) = g(x, y)$$

where $f(x, y)$ and $g(x, y)$ represent the initial displacement and velocity of the membrane.

2.3 Implementation in Python

To solve the two-dimensional wave equation numerically, we implement the finite difference method using Python. Below is a sample implementation.

Listing 1: Python code for solving the two-dimensional wave equation using finite difference method

```python
# Libraries
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm


def f(X, Y):  # u(x,y,0) = f(x,y,0)
    return np.sin(np.pi * X) * np.sin(np.pi * Y)


def g(X, Y):  # u_t(x,y,0) = g(x,y,0)
    return np.zeros(X.shape)


def laplacian(arr, row, col, dx):  # Laplacian in terms of FDM
    return (
        arr[row + 1, col]
        + arr[row - 1, col]
        + arr[row, col + 1]
        + arr[row, col - 1]
        - 4 * arr[row, col]
    ) / (dx**2)


def initialize_conditions(rect, hs, BC):
    X, Y = np.meshgrid(
        np.linspace(rect[0], rect[1], hs[0]), np.linspace(rect[2], rect[3], hs[1])
    )
    Z_init = f(X, Y)
    Z_0 = f(X, Y)
    Z_dot_init = g(X, Y)
    return X, Y, Z_init, Z_0, Z_dot_init


def apply_boundary_conditions(Z, BC):
    Z[0] = np.ones(len(Z[0])) * BC[0]
    Z[-1] = np.ones(len(Z[-1])) * BC[1]
    Z[:, 0] = np.ones(len(Z[:, 0])) * BC[2]
    Z[:, -1] = np.ones(len(Z[:, -1])) * BC[3]

```

```python

def configure_plot(rect, zmin, zmax):
    fig = plt.figure()
    ax = plt.axes(projection="3d")
    ax.axes.set_xlim3d(rect[0], rect[1])
    ax.axes.set_ylim3d(rect[2], rect[3])
    ax.axes.set_zlim3d(zmin, zmax)
    plt.rcParams["mathtext.fontset"] = "stix"
    plt.rcParams["font.family"] = "STIXGeneral"

    ax.set_title(
        "Wave Simulation in a \nrectangular boundary",
        fontsize=18,
        fontname="STIXGeneral",
    )

    return fig, ax


def plot_boundary_lines(ax, rect, BC):
    lines = [
        ([rect[0], rect[1]], [rect[2], rect[2]], [BC[0], BC[0]]),
        ([rect[1], rect[1]], [rect[2], rect[3]], [BC[3], BC[3]]),
        ([rect[0], rect[0]], [rect[2], rect[3]], [BC[2], BC[2]]),
        ([rect[0], rect[1]], [rect[3], rect[3]], [BC[1], BC[1]]),
        ([rect[0], rect[0]], [rect[2], rect[2]], [BC[0], BC[2]]),
        ([rect[1], rect[1]], [rect[2], rect[2]], [BC[0], BC[3]]),
        ([rect[0], rect[0]], [rect[3], rect[3]], [BC[1], BC[2]]),
        ([rect[1], rect[1]], [rect[3], rect[3]], [BC[1], BC[3]]),
    ]
    for x, y, z in lines:
        ax.plot(x, y, z, color="black", linewidth=2)


def update_wave_equation(Z_0, Z_dot_init, Z_init, laplacian, hs, dx, dt, c):
    for row in range(1, hs[0] - 1):
        for col in range(1, hs[1] - 1):
            Z_0[row, col] = (
                Z_0[row, col]
                - 2 * laplacian(Z_dot_init, row, col, dx)
                + 0.5 * c**2 * dt**2 * laplacian(Z_0, row, col, dx)
            )
    return Z_0


def create_temp_matrix(hs, BC):
    Z_temp = np.zeros((hs[0], hs[1]))
    Z_temp[0] = np.ones(len(Z_temp[0])) * BC[0]
    Z_temp[-1] = np.ones(len(Z_temp[-1])) * BC[1]
    Z_temp[:, 0] = np.ones(len(Z_temp[:, 0])) * BC[2]
    Z_temp[:, -1] = np.ones(len(Z_temp[:, -1])) * BC[3]
    return Z_temp


def update_temp_matrix(Z_temp, zs, iteration, laplacian, hs, dx, dt, c):
    for row in range(1, hs[0] - 1):
        for col in range(1, hs[1] - 1):
            Z_temp[row, col] += (
                2 * zs[iteration - 1][row, col]
                - zs[iteration - 2][row, col]
                + c**2 * dt**2 * laplacian(zs[iteration - 1], row, col, dx)
            )
    return Z_temp


def simulation_wave(rect, hs, BC, c, frames):
    X, Y, Z_init, Z_0, Z_dot_init = initialize_conditions(rect, hs, BC)
    apply_boundary_conditions(Z_init, BC)

    zmax = max(Z_init.max(), BC[0], BC[1], BC[2], BC[3])
    zmin = min(Z_init.min(), BC[0], BC[1], BC[2], BC[3])

    fig, ax = configure_plot(rect, zmin, zmax)
```

```
    plot_boundary_lines(ax, rect, BC)

    dx = (rect[1] - rect[0]) / (hs[0] - 1)
    dt = dx / (10 * c)

    zs = []
    Z_0 = update_wave_equation(Z_0, Z_dot_init, Z_init, laplacian, hs, dx, dt, c)
    zs.extend([Z_0, Z_init])

    surf = ax.plot_surface(X, Y, Z_init, alpha=0.7, cmap="magma", vmin=zmin, vmax=zmax)
    cbar = fig.colorbar(surf)
    cbar.ax.set_ylabel("Amplitude", rotation=270, fontsize=14, labelpad=20)
    ax.set_axis_off()

    for iteration in range(2, frames):
        surf.remove()
        Z_temp = create_temp_matrix(hs, BC)
        Z_temp = update_temp_matrix(Z_temp, zs, iteration, laplacian, hs, dx, dt, c)
        zs.append(Z_temp)

        surf = ax.plot_surface(
            X, Y, zs[iteration - 2], alpha=1, cmap="magma", vmin=zmin, vmax=zmax
        )
        plt.pause(dt)


# Running the simulation
simulation_wave(
    rect=[-1, 1, -1, 1], hs=[20, 20], BC=[0, 0, 0, 0], c=3_000_000, frames=1000
)
```

3 Results and Discussion

3.1 Two-Dimensional Wave Equation

The numerical solution for the two-dimensional wave equation using the finite difference method is shown in Figure 1. The initial condition is a product of sine waves, and the boundary conditions are fixed at zero.

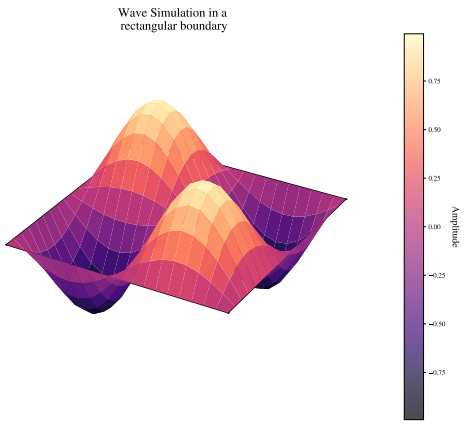

Figure 1: Numerical solution of the two-dimensional wave equation at a specific time step.

The surface plot in Figure 1 shows the wave propagation in a two-dimensional domain. The numerical solution captures the wave spreading outwards from the center, demonstrating the effectiveness of the finite difference method for solving two-dimensional problems.

4 Conclusion

The finite difference method is a powerful numerical technique for solving partial differential equations, such as the wave equation. By discretizing the spatial and temporal domains, we can approximate the solution with high accuracy. This project demonstrates the application of the finite difference method to the two-dimensional wave equation, providing insights into wave propagation phenomena. Future work could explore more complex initial and boundary conditions, as well as the implementation of other numerical methods such as finite element or spectral methods.