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

Listings

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. [1]

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. [1]

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:

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

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

$$u_{i,j,k+1} = 2(1 - \lambda_x^2 - \lambda_y^2)u_{i,j,k} + \lambda_x^2(u_{i+1,j,k} + u_{i-1,j,k}) + \lambda_y^2(u_{i,j+1,k} + u_{i,j-1,k}) - u_{i,j,k-1} \tag{3}$$

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:

$$\lambda_x^2 + \lambda_y^2 \leq 1 \tag{4}$$

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 given by Equation 3. [2]

2.2 Mesh Grid

The mesh refers to the grid that discretizes the spatial and temporal domains into a finite number of points. In this project, the spatial domain is divided into a uniform grid with N_x points in the x-direction and N_y points in the y-direction. The spacing between the points in each direction is given by Δx and Δy, respectively. [2]

$$\Delta x = \frac{L_x}{N_x - 1}, \quad \Delta y = \frac{L_y}{N_y - 1}$$

where L_x and L_y are the lengths of the domain in the x and y directions. The grid points are denoted by (x_i, y_j) where

$$x_i = i\Delta x, \quad y_j = j\Delta y, \quad i = 0, 1, \ldots, N_x - 1, \quad j = 0, 1, \ldots, N_y - 1$$

The mesh provides the framework for applying the finite difference method to approximate the derivatives in the wave equation.

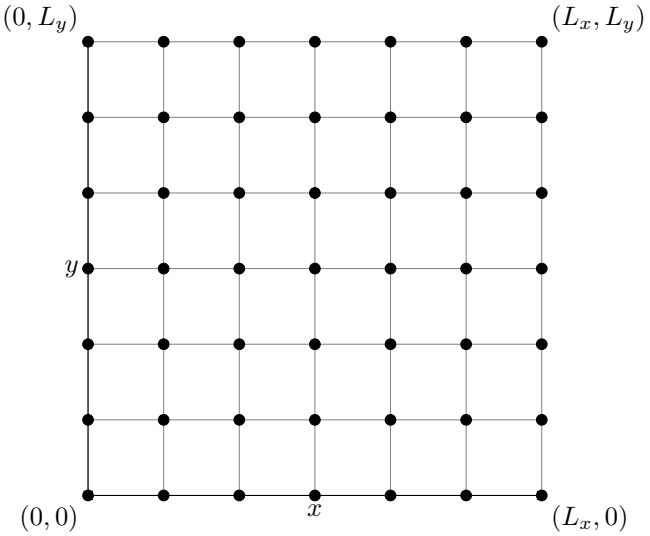


Figure 1: Mesh for discretizing the spatial domain.

The temporal domain is discretized into a grid with N_t points, where the spacing between points is given by Δt:

$$\Delta t = \frac{T}{N_t - 1}$$

where T is the total time for the simulation. The grid points in the temporal domain are denoted by t_k where

$$t_k = k\Delta t, \quad k = 0, 1, \ldots, N_t - 1$$

The temporal mesh ensures that the wave equation is solved at discrete time steps, allowing for the observation of wave propagation over time.

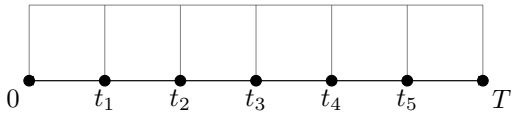


Figure 2: Mesh for discretizing the temporal domain.

Figure 3 illustrates the combined mesh for both spatial and temporal domains. The spatial domain, displayed on the left, is discretized into a grid with points in the x and y directions. The temporal domain, shown on the right, is discretized along the time axis and raised by one unit on the y-axis for clarity. Dashed lines connect each temporal grid point to the corresponding spatial grid, indicating how the solution progresses over time. This combined visualization helps to understand the relationship between the spatial and temporal discretizations in the finite difference method.

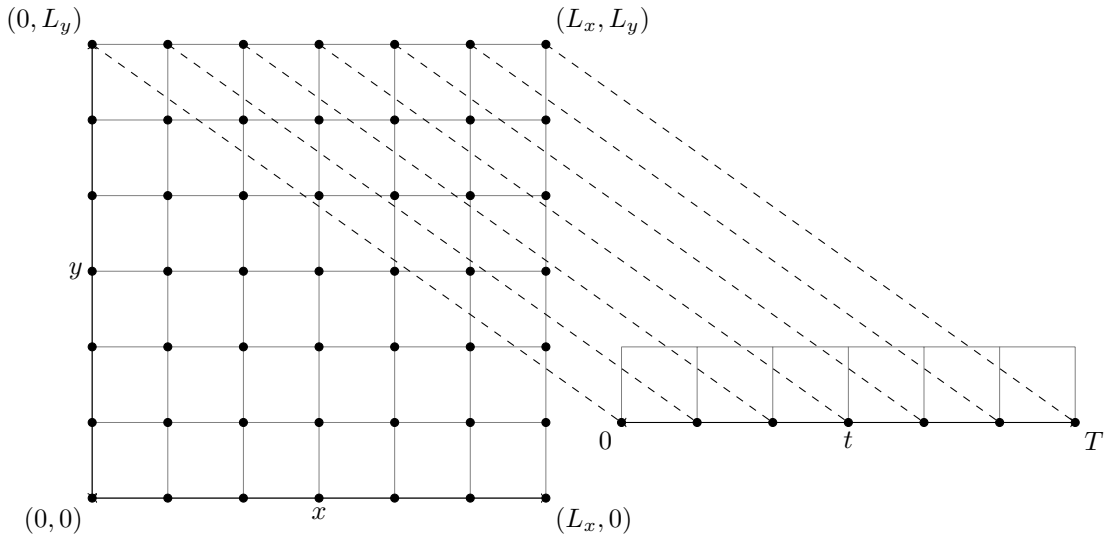


Figure 3: Mesh for discretizing both spatial and temporal domains.

2.3 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]

2.4 Implementation in Python

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

The first step in the implementation is importing the necessary libraries. We use `matplotlib` for plotting and `numpy` for numerical computations. The `cm` module from `matplotlib` is used to define the colormap for the plots.

Listing 1: Libraries

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
```

Then, we define the initial conditions functions $f(x, y)$ and $g(x, y)$ as shown below. These functions represent the initial displacement and velocity of the membrane. In this example, we use sine waves as the initial conditions. However, more complex functions can be used depending on the application.

Listing 2: Initial Conditions Functions

```
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)
```

Next, we need a Laplacian function to compute the Laplacian of the wave function $u(x, y, t)$. The Laplacian operator is defined as the sum of the second partial derivatives of u with respect to the spatial coordinates. However, in this implementation, we use central difference quotients to approximate the Laplacian. Note that this implementation could be optimized by vectorizing the computation.

Listing 3: Laplacian Function

```
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)
```

After that, we define the parameters of the problem, such as the wave speed c, the spatial and temporal steps Δx, Δy, and Δt, and the number of grid points in each direction. These parameters determine the resolution of the numerical solution and should be chosen carefully to balance accuracy and computational cost. The following function is used to initialize the parameters. `rect` is the domain of the problem, `hs` is the number of grid points in each direction, and `BC` is the boundary condition array.

Listing 4: Initialize Conditions

```
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
```

Now, we need to enforce the boundary conditions. The following function sets the boundary values of the wave function $u(x, y, t)$ to zero. This is done by setting the values of the first and last rows and columns of the grid to zero.

Listing 5: Apply Boundary Conditions

```
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]
```

Set up the plot for visualizing the wave simulation. Matplotlib's 3D plotting capabilities are used to create a surface plot of the wave function $u(x, y, t)$ at a specific time step. The following function configures the plot with appropriate labels and titles.

Listing 6: Configure Plot

```
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
```

We plot the boundary lines to visualize the domain of the problem. The following function plots the boundary lines of the rectangular domain with the specified boundary conditions.

Listing 7: Plot Boundary Lines

```
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)
```

Then we need to update the displacement matrix `Z_0` based on the finite difference approximation of the wave equation.

Listing 8: Update Wave Equation

```
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
```

Create a temporary matrix to store intermediate values during the simulation. This matrix is used to update the wave function at each time step. Initializes a temporary matrix `Z_temp` with boundary conditions, which is then used for updating the wave equation.

Listing 9: Create Temporary Matrix

```
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
```

Update the temporary matrix for each time step using the values previous time steps: an iterative solution of the wave equation.

Listing 10: Update Temporary Matrix

```
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
```

Finally, we can run the simulation by iterating over the time steps and updating the wave function at each step. The function initializes the conditions, applies boundary conditions, and sets up the plot. It calculates the spatial step `dx` and temporal step `dt` based on the grid size and wave speed. The function iteratively updates the wave equation and visualizes the wave propagation over time.

Listing 11: Simulation Function

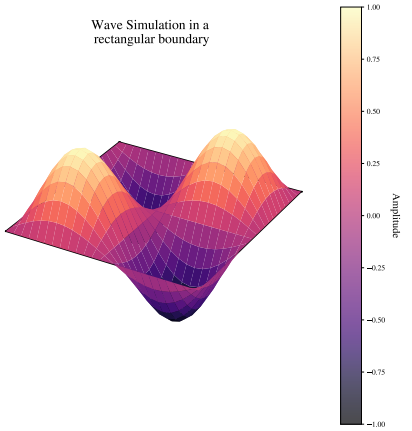
```
def simulation_wave(rect, hs, BC, c, frames):
    X, Y, Z_init, Z_0, Z_dot_init = initialize_conditions(rect, hs, BC)
    apply_boundary_conditions(Z_init, BC)

    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)
```

3 Results

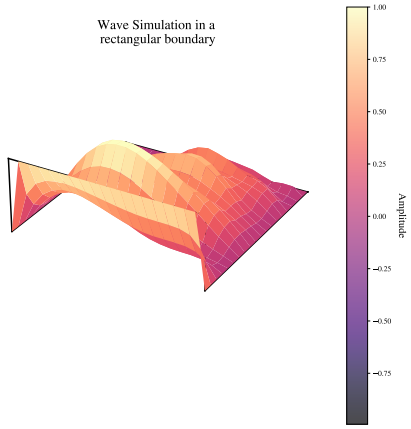
The numerical solution for the two-dimensional wave equation using the finite difference method is shown in Figure 4. The initial condition is a product of sine waves, and the boundary conditions are fixed at zero. The wave speed is set to $c = 3 \times 10^8$, and the domain is a square with side length $L = 1$. The simulation is run for 100 time steps with a spatial resolution of 25×25 grid points.

The surface plot in Figure 4 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. The wave speed, boundary conditions, and initial conditions can be adjusted to explore different scenarios.

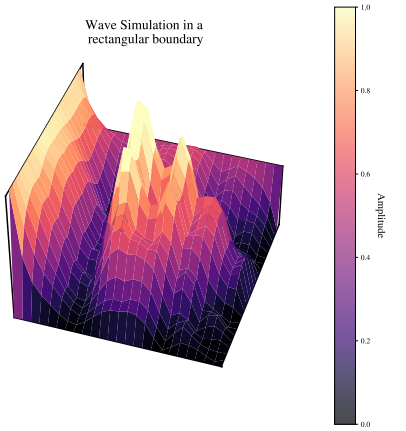
A notable shortcoming of this implementation is the time complexity of the algorithms used is exponential, which makes it computationally expensive for large grids. Future work could focus on optimizing the code for better performance. Additionally, some improvements could be made to the visualization of the wave propagation, such as adding interactive controls to adjust the parameters of the simulation. Lastly, there are few edge cases that need to be handled, such as the stability of the solution and the convergence of the numerical method.



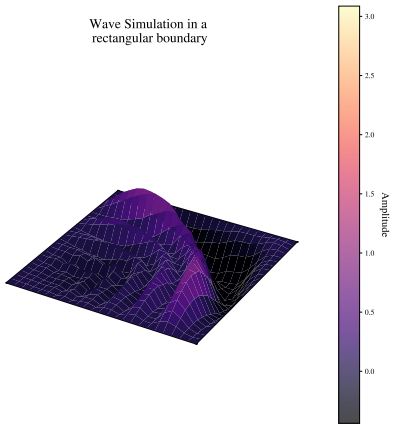
(a) BC: $u(0, y, t) = u(L, y, t) = u(x, 0, t) = u(x, L, t) = 0$
IC: $u(x, y, 0) = \sin(\pi x) \sin(\pi y)$



(b) BC: $u(0, y, t) = 1, u(L, y, t) = u(x, 0, t) = u(x, L, t) = 0$
IC: $u(x, y, 0) = \sin(\pi x) \sin(\pi y)$



(c) BC: $u(0, y, t) = 1, u(L, y, t) = u(x, 0, t) = u(x, L, t) = 0$
IC: $u(x, y, 0) = xy(1 - x)(1 - y)$



(d) BC: $u(0, y, t) = u(L, y, t) = u(x, 0, t) = u(x, L, t) = 0$
IC: $u(x, y, 0) = xy(1 - x)(1 - y)$

Figure 4: Numerical solution of the two-dimensional wave equation at a specific time step.

4 Conclusion

The finite difference method (FDM) proves to be an effective and robust numerical technique for solving partial differential equations (PDEs), specifically the two-dimensional wave equation. By discretizing the spatial and temporal domains, FDM enables the approximation of solutions with high accuracy, making it a valuable tool for understanding complex wave phenomena in various fields such as physics, engineering, and computational science.

In this project, we have successfully applied the finite difference method to solve the two-dimensional wave equation, demonstrating its utility through comprehensive implementation and visualization in Python. The study is structured into three significant parts: an introduction to the wave equation and the finite difference method, detailed implementation steps, and a thorough presentation of results and visualizations.

The numerical solutions obtained using the finite difference method align closely with the theoretical expectations of wave behavior. The stability condition, given by the Courant-Friedrichs-Lewy (CFL) condition, was crucial in ensuring the reliability of the simulations. Adhering to the CFL condition prevented numerical instabilities, allowing for consistent and accurate wave propagation results.

The project explored various initial and boundary conditions, showcasing the method's flexibility in handling different physical scenarios. The choice of initial conditions, such as sine waves and polynomial functions, demonstrated how the initial shape of the wave can influence its subsequent evolution. Fixed boundary conditions were applied to model realistic physical constraints, and the resulting wave behavior was visualized effectively.

The Python implementation leveraged libraries such as NumPy and Matplotlib to perform numerical computations and create visualizations. The step-by-step breakdown of the code, including the discretization of the wave equation, the application of boundary conditions, and the update of wave propagation, provided a clear framework for implementing FDM for similar problems. The visualizations, including 3D surface plots, helped in intuitively understanding the wave dynamics.

While the finite difference method is computationally intensive, especially for large grids and long simulation times, it remains a practical approach for many real-world applications. The project's implementation highlighted the trade-offs between computational cost and accuracy, suggesting areas for potential optimization, such as parallel processing or adaptive grid techniques.

Future work could involve the application of more complex boundary conditions, such as absorbing or periodic boundaries, to simulate a wider range of physical phenomena. This would enhance the method's applicability to real-world scenarios where waves interact with varying mediums and interfaces.

References

[1] D. Zill, *Differential Equations with Boundary-value Problems*. Cengage Learning, 2017, ISBN: 9781337559881. [Online]. Available: https://books.google.com.eg/books?id=apFMAQAACAAJ.

[2] R. Burden and J. Faires, *Numerical Analysis* (Mathematics Series). PWS-Kent Publishing Company, 1993, ISBN: 9780534932213. [Online]. Available: https://books.google.com.eg/books?id=AWLuAQAACAAJ.

A Code

Listing 12: Python code for solving the two-dimensional wave equation using the 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.zeros_like(X)


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


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]


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)
```