Equilibrium Temperature Distributions

Using Linear Algebra

Team Members:

202200102	Ali Ehab	s-ali.aziz@zewailcity.edu.eg
202200776	Marwan Basem	s-marwan.hussien@zewailcity.edu.eg
202201079	SalahDin Rezk	s-salahdin.rezk@zewailcity.edu.eg

Contents

List of Figures

Abstract

The study of the heat equation has been instrumental in thermodynamics; we consider one of the most famous sets of problems in Differential Equations, the Dirichlet Boundary value problem [1]. We study the temperature inside a plate which is in thermal equilibrium for which we only know the temperature on the boundary. We are providing numerical solutions utilizing many of the tools developed in Linear Algebra.

1 Introduction

The heat equation is a fundamental partial differential equation that describes the distribution of heat in a system over time and space. It is given by:

$$\frac{\partial u}{\partial t} = \Delta u \tag{1}$$

where u represents the temperature distribution, t is time, and Δ is the Laplacian operator. This equation is widely used in various fields, including physics, engineering, and mathematics, to model heat transfer phenomena.

In this context, we are interested in studying systems that reach thermal equilibrium, where the temperature distribution remains constant over time ($\frac{\partial u}{\partial t} = 0$). This special case of the heat equation can be written as:

$$\Delta u = 0 \tag{2}$$

This equation is known as the steady-state heat equation or Laplace's equation. It describes the temperature distribution in a system that has reached a state of balance, where the heat flow is constant and there are no sources or sinks of heat.

Solutions to Laplace's equation possess a remarkable property known as the mean-value property. This property states that if u is a solution to $\Delta u = 0$, then the value of u at any point (x, y) is equal to the average of its values on a circle centered at (x, y). Mathematically, this can be expressed as:

$$u(x,y) = \frac{1}{2\pi}\int_0^{2\pi} u(x + r\,\cos\,\theta, y + r\,\sin\,\theta)d\theta \tag{3}$$

where r represents the radius of the circle and θ is the angle of rotation.

This mean-value property provides valuable insights into the behavior of solutions to Laplace's equation and has important implications in various areas of mathematics and physics. It allows us to understand the relationship between the values of a function at different points and provides a powerful tool for solving problems involving Laplace's equation. [2]

2 Problem

In this problem, we are given the temperature distribution on the boundaries of a plate that is in thermal equilibrium. Our goal is to determine the temperature distribution inside the plate based solely on the boundary conditions. We need to solve the steady-state heat equation, $\Delta u = 0$, numerically to obtain the desired solution.

3 Solution

To solve the steady-state heat equation numerically, we start by creating an $n \times n$ grid, where the lattice points represent the locations at which we will compute the temperature. This grid allows us to discretize the problem and approximate the continuous temperature distribution.

Since we are dealing with Laplace's equation, we can apply the mean-value property to our discrete setup. This property states that the temperature at any point on the grid is equal to the average of the temperatures of its neighboring points. Mathematically, this can be expressed as:

$$T_{i,j} = \frac{1}{4}(T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}) \tag{4}$$

where $T_{i,j}$ represents the temperature at the point (i, j) on the grid.

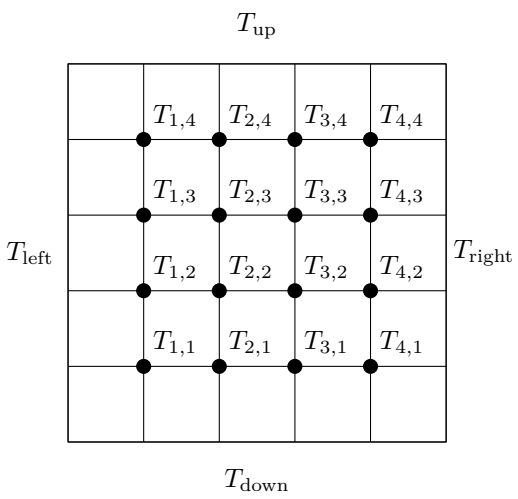


Figure 1: Example of a $n = 4$ square grid

By applying this equation to every point on the grid, we obtain a system of linear equations. This system can be written in matrix form as:

$$\mathbf{t} = M\mathbf{t} + \mathbf{b} \tag{5}$$

where $\mathbf{t}$ is a column vector representing the unknown temperatures, M is the matrix containing the coefficients of the unknown temperatures, and $\mathbf{b}$ is a column vector that represents the right-hand side of the system.

We can solve this system using two different methods: the inverse matrix method and the Jacobi iteration method.

3.1 Inverse Matrix Method

In the inverse matrix method, we rewrite the equation as:

$$\mathbf{t} = M\mathbf{t} + \mathbf{b} \tag{6}$$

$$(\boldsymbol{I} - M)\mathbf{t} = \mathbf{b} \tag{7}$$

$$\mathbf{t} = (\boldsymbol{I} - M)^{-1}\mathbf{b} \tag{8}$$

where $\boldsymbol{I}$ is the identity matrix.

By computing the inverse of the matrix $(\boldsymbol{I}-M)$, we can obtain the solution vector $\mathbf{t}$, which represents the temperature distribution inside the plate.

3.2 Jacobi Iteration Method

The Jacobi iteration method is an iterative technique that allows us to approximate the solution to the system of linear equations. It involves repeatedly updating the temperature values at each point on the grid until convergence is achieved.

The Jacobi iteration formula for our problem can be written as:

$$T_{i,j}^{(k+1)} = \frac{1}{4}(T_{i+1,j}^{(k)} + T_{i-1,j}^{(k)} + T_{i,j+1}^{(k)} + T_{i,j-1}^{(k)}) \tag{9}$$

where $T_{i,j}^{(k)}$ represents the temperature at point (i, j) in the k-th iteration.

We start with an initial guess for the temperature distribution and update the values at each point using the Jacobi iteration formula. We continue this process until the temperature values converge to a stable solution.

Both the inverse matrix method and the Jacobi iteration method provide solutions to the steady-state heat equation. The choice of method depends on the specific requirements of the problem and the desired accuracy of the solution.

By implementing these methods and solving the system of linear equations, we can determine the temperature distribution inside the plate based on the given boundary conditions. This allows us to gain insights into the heat transfer behavior and study the thermal equilibrium of the system.

4 Implementation

The code was heavily inspired by [3], [4]. The project was implemented using Python 3.12.0 and Poetry 1.6.1 for package and tendency management. The reason behind choosing Python is that it is a high-level language that is easy to read and write. It also has a lot of libraries that make it easy to implement the project. In addition, Poetry provides a friendly and dynamic interface for interacting with the complex system of python libraries while ensuring consistency.

Furthermore, Git 2.43.0 was used for version control and GitHub for hosting the code repository. Git provided a simple and efficient way to track changes and collaborate on the project, while GitHub allowed us to share our code and work together on the project remotely. Both of these tools were essential in the development process and contributed significantly to the success of the project.

4.1 Code Structure

The code is structured into different modules, each of which contains a set of functions that perform a specific task. This modular approach allows us to organize the code and make it easier to maintain. Directory structure is shown below.

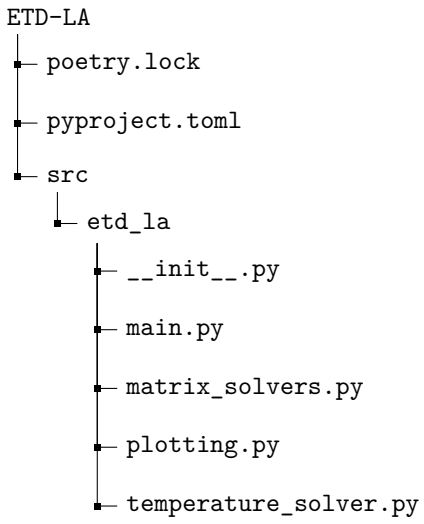


Figure 2: Project Hierarchy

4.2 Importing libraries

We start by importing the necessary libraries for our implementation. We will be using the `numpy` library for linear algebra operations and the `matplotlib` library for plotting. We also import the `Axes3D` class from the `mpl_toolkits.mplot3d` module to enable 3D plotting.

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

4.3 Solver functions

4.3.1 Inverse matrix

We can compute the inverse of $\boldsymbol{I} - M$ using the following python code. This function takes the coefficient matrix M and the right-hand side vector b as inputs, then calculates the inverse of M and multiplies it by b to obtain the solution vector x. The efficiency of this method is to be tested in the appendix.

```python
def calculate_inverse_solution(A, b):
    A_inverse = np.linalg.inv(A)
    x = np.dot(A_inverse, b)
    return x
```

4.3.2 Jacobi iterations

We start with an initial guess for $\mathbf{t}_0$, and we iteratively plug it in the equation

$$\mathbf{t}_n = M\mathbf{t}_n - 1 + b \tag{10}$$

each value $\mathbf{t}_n$ is a better approximation for the exact solution. The following Python code implements the Jacobi iteration method. It takes the coefficient matrix M, the right-hand side vector b, an initial guess x_0, a tolerance value `tol`, and a maximum number of iterations `max_iter` as inputs. It then iteratively updates the temperature values at each point on the grid until convergence is achieved. If the solution does not converge within the specified number of iterations, an error is raised. This is done to prevent the program from running indefinitely in case the solution does not converge.

```python
def jacobi_iteration(A, b, x0=None, tol=1e-6, max_iter=1000):
    n = len(b)
    x = x0 if x0 is not None else np.zeros(n)
    z_iter = np.zeros((max_iter, n))

    for k in range(max_iter):
        x_new = np.zeros_like(x)

        for i in range(n):
            sigma = np.dot(A[i, :i], x[:i]) + np.dot(A[i, i + 1 :], x[i + 1 :])
            x_new[i] = (b[i] - sigma) / A[i, i]
            z_iter[k] = x_new

        if np.linalg.norm(x_new - x) < tol:
            return x_new, k + 1, z_iter[:k + 1]

        x = x_new

    raise ValueError(
        "Jacobi iteration did not converge within the specified number of iterations."
    )
```

4.4 Solving the system

4.4.1 Creating the coefficient matrix

We start by creating the coefficient matrix M. We can do this by creating a matrix of zeros, and then filling it with the coefficients of the unknown temperatures. The following Python code implements this approach. It takes the size of the grid n and the boundary temperatures as inputs and returns the coefficient matrix M and the right-hand side vector b. The boundary temperatures are specified as follows: `left_temp` for the left boundary, `up_temp` for the upper boundary, `right_temp` for the right boundary, and `down_temp` for the lower boundary.

```python
def generate_coefficient_matrix(size, left_temp, up_temp, right_temp, down_temp):
    matrix = np.zeros((size * size, size * size))
    rhs_vector = np.zeros(size * size)
```

```
    for row in range(size):
        for col in range(size):
            point_num = size * row + col
            if col - 1 >= 0:
                matrix[point_num][point_num - 1] = 1
            else:
                rhs_vector[point_num] += left_temp

            if row - 1 >= 0:
                matrix[point_num][point_num - size] = 1
            else:
                rhs_vector[point_num] += up_temp

            if col + 1 < size:
                matrix[point_num][point_num + 1] = 1
            else:
                rhs_vector[point_num] += right_temp

            if row + 1 < size:
                matrix[point_num][point_num + size] = 1
            else:
                rhs_vector[point_num] += down_temp

    return matrix, rhs_vector
```

4.4.2 Temperature Equation

Now that we have the coefficient matrix M and the right-hand side vector b, we can solve the temperature equation. The following Python code implements this approach. It takes the size of the grid n and the boundary temperatures as inputs and returns the solution vector x. The multiplication by 4 is done to account for the $\frac{1}{4}$ coefficient in the mean-value property while the `np.identity` is done to account for the $\boldsymbol{I}$ matrix in the equation. The `solve_temperature_equation` function is a wrapper function that combines the steps of generating the coefficient matrix and solving the temperature equation. The `calculate_inverse_solution` function is used by default.

```
def solve_temperature_equation(size, left_temp, up_temp, right_temp, down_temp):
    coefficient_matrix, rhs_vector = generate_coefficient_matrix(
        size, left_temp, up_temp, right_temp, down_temp
    )

    # Uncomment one of the following lines based on your preferred solver
    # temperature_solution, _ = jacobi_iteration(4 * np.identity(size * size) -
        coefficient_matrix, rhs_vector)
    temperature_solution = calculate_inverse_solution(
        4 * np.identity(size * size) - coefficient_matrix, rhs_vector
    )

    return temperature_solution
```

4.5 Plotting

To visualize the equilibrium temperature distribution on the plate, we can create a heat map plot with contour lines representing isothermal lines. The following Python code implements this approach. It takes the size of the grid n and the solution vector x as inputs and plots the temperature distribution on the plate. The `plot_temperature_distribution_2d` function is used to plot the temperature distribution on a 2D grid, while the `plot_temperature_distribution_3d` function is used to plot the temperature distribution on a 3D grid. In both cases, the `cmap` parameter is used to specify the color map, while the `interpolation` parameter is used to specify the interpolation method to smooth the image.

4.5.1 2D Plot

```
def plot_temperature_distribution_2d(size, temperature):
    plt.imshow(
        temperature.reshape(size, size), cmap="RdYlBu_r", interpolation="gaussian"
    )
    plt.colorbar()
```

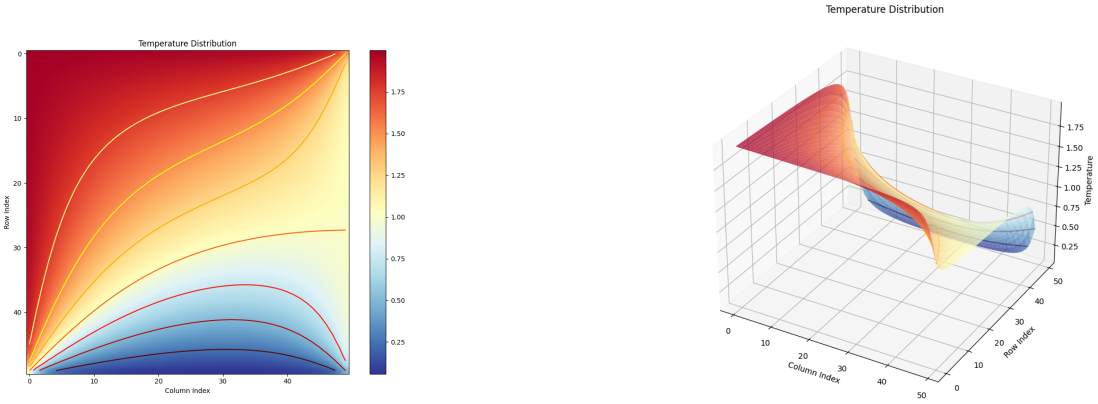
```
    plt.contour(temperature.reshape(size, size), cmap="hot")
    plt.title("Temperature Distribution")
    plt.xlabel("Column Index")
    plt.ylabel("Row Index")
    plt.show()

# Plot the temperature distribution
plot_temperature_distribution_2d(n, x)
```

4.5.2 3D Plot

```
def plot_temperature_distribution_3d(size, temperature):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection="3d")
    x = np.arange(0, size, 1)
    y = np.arange(0, size, 1)
    X, Y = np.meshgrid(x, y)
    ax.plot_surface(X, Y, temperature.reshape(size, size), cmap="RdYlBu_r", alpha=0.75)
    ax.contour(X, Y, temperature.reshape(size, size), cmap="hot")
    ax.set_title("Temperature Distribution")
    ax.set_xlabel("Column Index")
    ax.set_ylabel("Row Index")
    ax.set_zlabel("Temperature")
    plt.show()
```

5 Results

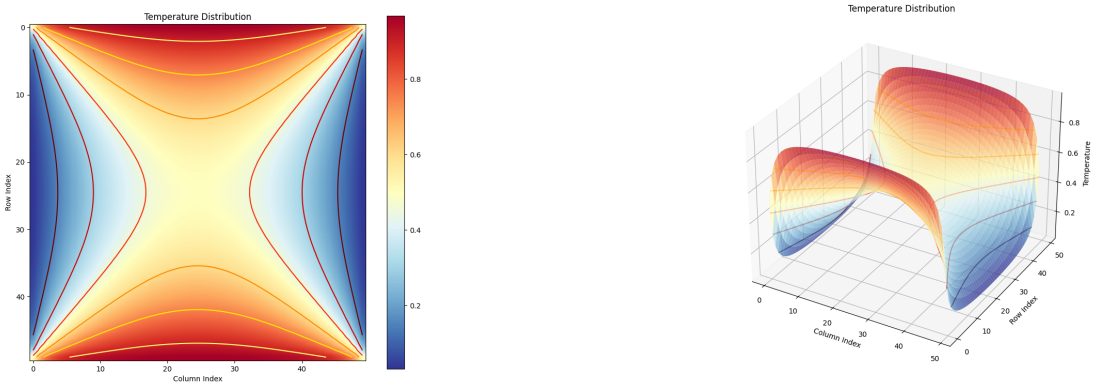
We tested our implementation on a variety of boundary conditions to gain insights into the behavior of the temperature distribution. The results obtained through this process are presented below. Both 2D and 3D plots are included to visualize the temperature distribution on the plate. The color map used in the plots is `RdYlBu_r`, while the interpolation method is `gaussian` for the 2D plots and `linear` for the 3D plots. We can observe that the temperature distribution is symmetric about the center of the plate, which is consistent with the mean-value property of solutions to Laplace's equation. However, the last case is an edge case where the temperature distribution is not symmetric due to the boundary conditions.



(a) 2D Plot

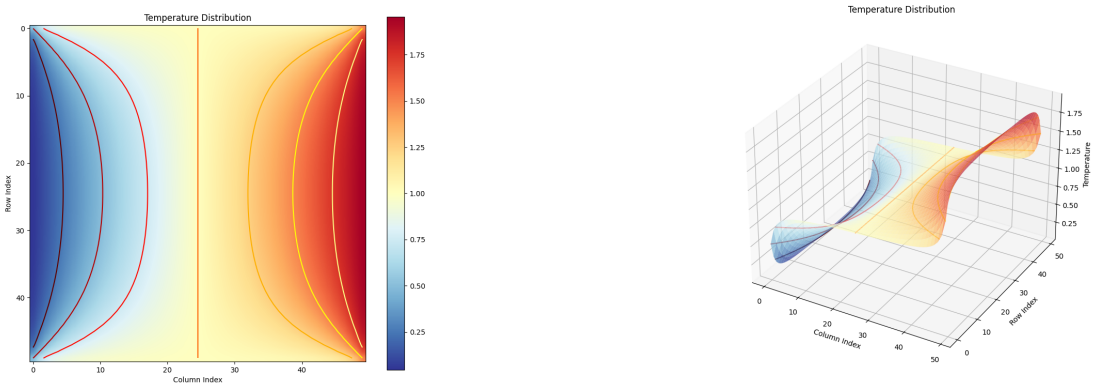
(b) 3D Plot

Figure 3: Boundaries $T_{\text{left}} = 2, T_{\text{up}} = 2, T_{\text{right}} = 1, T_{\text{down}} = 0$



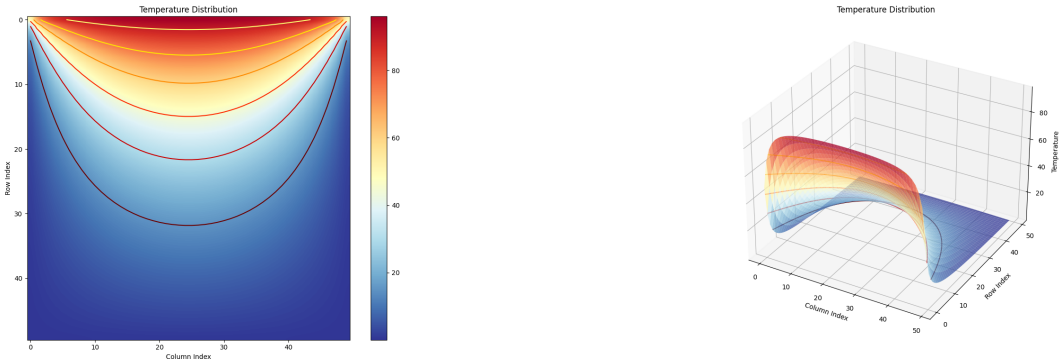
(a) 2D Plot (b) 3D Plot

Figure 4: Boundaries $T_{\text{left}} = 0, T_{\text{up}} = 1, T_{\text{right}} = 0, T_{\text{down}} = 1$



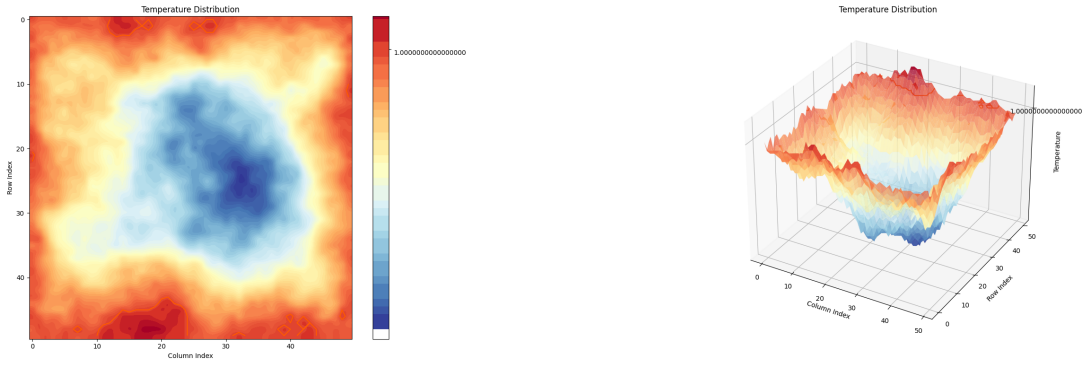
(a) 2D Plot (b) 3D Plot

Figure 5: Boundaries $T_{\text{left}} = 0, T_{\text{up}} = 1, T_{\text{right}} = 2, T_{\text{down}} = 1$



(a) 2D Plot (b) 3D Plot

Figure 6: Boundaries $T_{\text{left}} = 0, T_{\text{up}} = 100, T_{\text{right}} = 0, T_{\text{down}} = 0$



(a) 2D Plot

(b) 3D Plot

Figure 7: Boundaries $T_{\text{left}} = 1, T_{\text{up}} = 1, T_{\text{right}} = 1, T_{\text{down}} = 1$ (edge case)

6 Conclusion

In this study, we explored the application of linear algebra techniques to solve the numerical problem of determining equilibrium temperature distributions within a plate. The motivation for this research arises from the heat equation and its steady-state form, Laplace's equation, which describes the spatial distribution of temperature in a system at thermal equilibrium.

We formulated the problem as a system of linear equations, with the unknown temperatures at each lattice point represented by a vector. By leveraging the mean-value property of solutions to Laplace's equation, we derived a system of equations that could be expressed in matrix form. Two methods were employed for solving this system: the inverse matrix method and Jacobi iterations.

The results obtained through these numerical methods provided insights into the temperature distribution within the plate based solely on the boundary conditions. Visualization of the equilibrium temperature distribution was achieved through 2D and 3D plots, enhancing our understanding of the spatial variation of temperatures.

The presented approach not only serves as a practical method for solving such problems but also highlights the versatility of linear algebra in addressing challenges in the field of thermodynamics. The ability to numerically determine temperature distributions within a plate contributes to the broader understanding of heat transfer phenomena and has potential applications in various engineering and scientific domains.

In conclusion, this study demonstrates the effective use of linear algebra techniques in solving real-world problems related to heat distribution. Future work may involve extending this approach to more complex geometries or incorporating additional physical factors into the model. Overall, the methods presented here provide a valuable framework for tackling problems in heat conduction and equilibrium temperature distributions.

7 References

[1] E. M. Stein and R. Shakarchi, *Princeton Lectures in Analysis*, ser. 4 Vols. Princeton, NJ: Princeton University Press, 2003, vol. 1.

[2] H. Anton and C. Rorres, *Elementary Linear Algebra with Supplemental Applications*. Johanneshov: TPB, 2011.

[3] A. Gandhi, F. Kalkin, and K. Kainth, *Equilibrium Temperature Project*, Apr. 2022. [Online]. Available: `https://github.com/anshgandhi4/equilibrium_temperature_project`.

[4] K. Vaghela, *MATLAB code for 2-D steady state heat conduction with adiabatic wall boundary condition.* Jun. 2022. [Online]. Available: `https://www.youtube.com/watch?v=YRcSv8Ybvb8`.

A Complementary Resources

- Code Repository
- Iterations Video (Boundaries $T_{\text{left}} = 2, T_{\text{up}} = 2, T_{\text{right}} = 1, T_{\text{down}} = 0$)