Numerical Study of Heat Conduction in a 2D Multi-Material Rod

Giada Alessi
Master in Thermal Engineering
Universitat Politècnica de Catalunya

February 18, 2025

Abstract

This report presents a numerical study of transient heat conduction in a two-dimensional rod composed of four different materials. The heat equation is solved using the implicit finite difference method (Backward Euler). The study involves defining boundary conditions, implementing numerical methods, and analyzing the temperature distribution over time. The results include numerical data at specified points and temperature contour maps at different time steps.

1 Introduction

This report focuses on solving the transient heat equation for a 2D rectangular rod composed of four different materials (Figure 1), subject to specific boundary conditions.

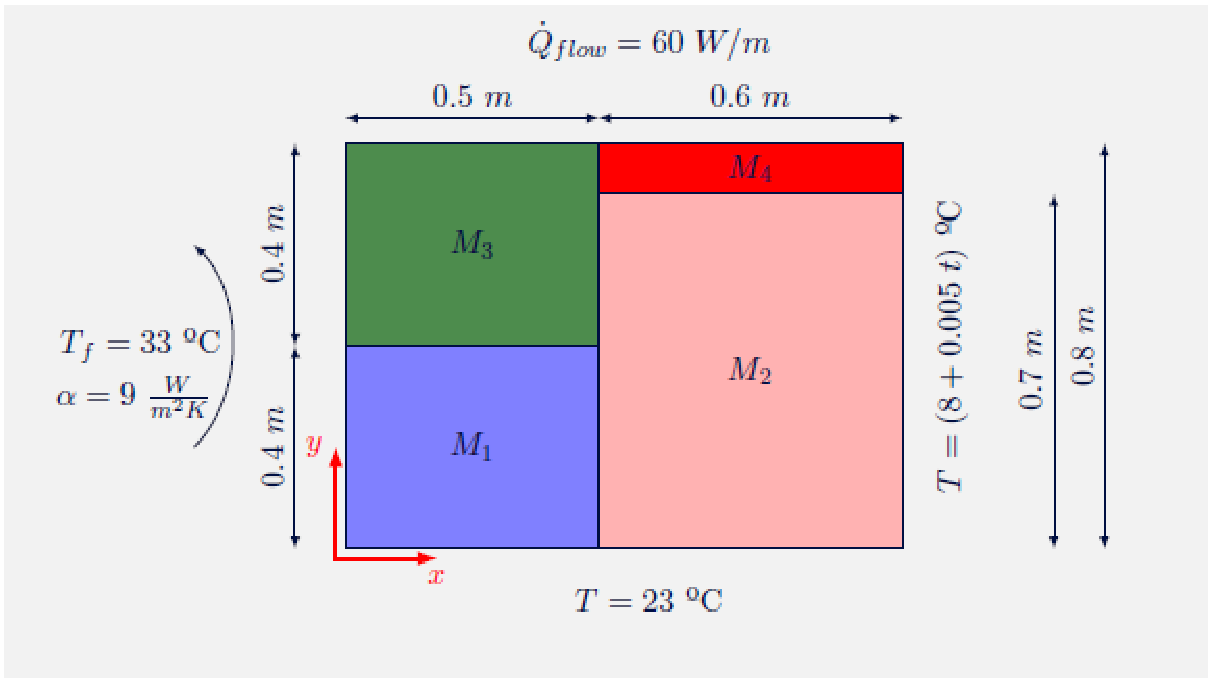

Figure 1: Very long rod composed of four different materials.

The materials considered exhibit the following properties:

Material	Density (ρ) [kg/m^3]	Heat Capacity (c_p) [J/kgK]	Conductivity (λ) [W/mK]
M1	1500	750	170
M2	1600	770	140
M3	1900	810	200
M4	2500	930	140

Table 1: Thermophysical properties of the materials used in the domain.

The study follows a numerical approach using the implicit Backward Euler method for time integration.

2 Theoretical Background

The governing equation for 2D transient heat conduction is:

$$\rho c_p \frac{\partial T}{\partial t} = \lambda \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) \tag{1}$$

Using the finite volume method, equation (1) is integrated for an arbitrary control volume, obtaining:

$$\rho c_p \frac{\partial T}{\partial t} \Delta \mathcal{V} = \sum_{\text{faces}} \lambda \frac{T_{nb} - T_P}{d_{nb}} \Delta S \tag{2}$$

In the 2D case, equation (2) can be rewritten as follows:

$$\rho c_p \frac{T_P^{n+1} - T_P^n}{\Delta t} \Delta \mathcal{V} = \lambda_e \frac{T_E - T_P}{d_{PE}} \Delta S_e + \lambda_w \frac{T_W - T_P}{d_{PW}} \Delta S_w + \lambda_n \frac{T_N - T_P}{d_{PN}} \Delta S_n + \lambda_s \frac{T_S - T_P}{d_{PS}} \Delta S_s \tag{3}$$

where ρ is the density, c_p is the specific heat capacity, λ is the thermal conductivity, and T is the temperature field.

The discretized heat conduction equation (3) must also be discretized in time. For the purpose of this exercise, it is suggested to use the most simple implicit method, a Backward Euler implicit method, meaning that the right-hand side of the equation is evaluated at the time step t^{n+1}.
The temperatures at the neighboring nodes are then defined as:

$$T_P \equiv T_P^{n+1} \quad T_E \equiv T_E^{n+1} \quad T_W \equiv T_W^{n+1} \quad T_N \equiv T_N^{n+1} \quad T_S \equiv T_S^{n+1} \tag{4}$$

Building on this formulation, the next section presents the numerical methodology used to solve the discretized heat conduction equation.

3 Numerical Methodology

The domain is discretized using a uniform mesh, with centered nodes, applying a finite difference approximation for the spatial derivatives.

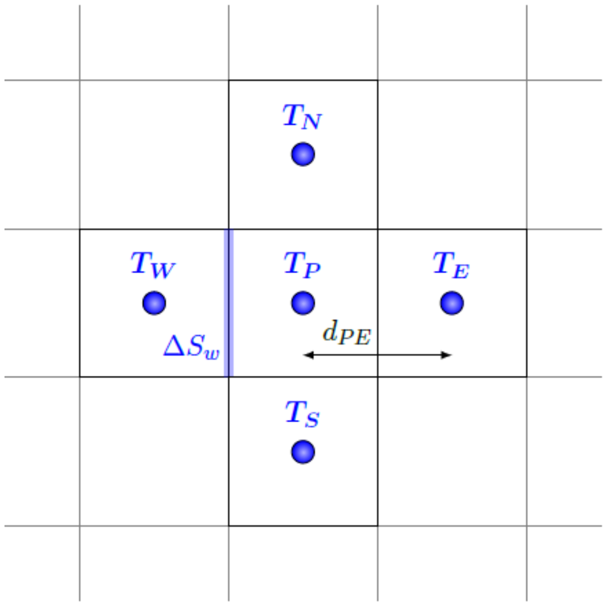

Figure 2: Centered nodes, uniform mesh with notation for non-boundary nodes.

Due to the constant meshing, the volume, surface areas, and distances between points can be expressed as:

$$\Delta \mathcal{V} = \Delta x \Delta y \Delta z, \tag{5}$$

$$\Delta S_e = \Delta S_w = \Delta y \Delta z, \tag{6}$$

$$\Delta S_n = \Delta S_s = \Delta x \Delta z, \tag{7}$$

$$d_{PE} = d_{PW} = \Delta x, \quad d_{PN} = d_{PS} = \Delta y. \tag{8}$$

This leads to the following simplified form of equation 3:

$$\rho c_p \frac{T_P^{n+1} - T_P^n}{\Delta t} = \lambda_e \frac{T_E - T_P}{\Delta x^2} + \lambda_w \frac{T_W - T_P}{\Delta x^2} + \lambda_n \frac{T_N - T_P}{\Delta y^2} + \lambda_s \frac{T_S - T_P}{\Delta y^2}. \tag{9}$$

The implicit Backward Euler method is employed to advance the temperature field in time. The discrete formulation at each grid point is:

$$a_p T_P = a_E T_E + a_W T_W + a_N T_N + a_S T_S + b \tag{10}$$

where a_p and $a_{E,W,N,S}$ are coefficients derived from material properties and discretization. For each internal node, the resulting coefficients are:

$$a_E = \frac{\lambda_e}{\Delta x^2} \quad a_W = \frac{\lambda_w}{\Delta x^2} \quad a_N = \frac{\lambda_n}{\Delta y^2} \quad a_S = \frac{\lambda_s}{\Delta y^2} \tag{11}$$

$$a_P = \rho c_p \frac{1}{\Delta t} + a_E + a_W + a_N + a_S \quad b = \rho c_p \frac{T_P^n}{\Delta t} \tag{12}$$

Since the thermal conductivity λ is associated with the face rather than the point, special attention must be given to the interfaces between different materials, which have varying thermal conductivities. In particular, at the interface, the harmonic mean is used, simplified due to the uniform mesh.

$$\lambda_f = \frac{2}{\frac{1}{\lambda_1} + \frac{1}{\lambda_2}} \tag{13}$$

Where the subscripts 1 and 2 indicate two different materials. The relation (13) was applied to every face, regardless of the material, to simplify the code. While this may slightly reduce efficiency, the computational cost remains negligible for this case.
By applying the boundary conditions from Figure 1, the governing equation (9) for the boundary nodes is modified as follows:

- Bottom (Dirichlet):

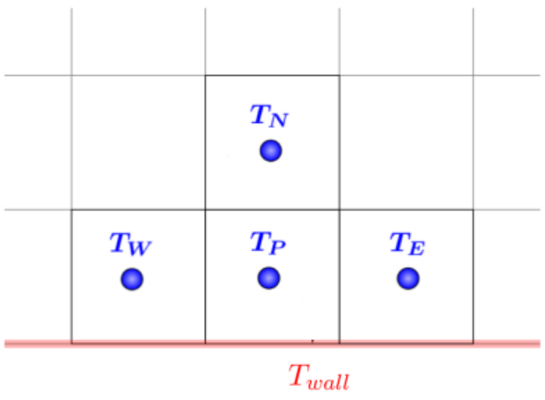

Figure 3: Dirichlet B.C. on bottom wall.

Isothermal wall at $T_{bottom} = 23°\text{C}$, resulting in:

$$\rho c_p \frac{T_P^{n+1} - T_P^n}{\Delta t} = \lambda_e \frac{T_E - T_P}{\Delta x^2} + \lambda_w \frac{T_W - T_P}{\Delta x^2} + \lambda_n \frac{T_N - T_P}{\Delta y^2} + \lambda_s \frac{T_{bottom} - T_P}{\Delta y^2} \tag{14}$$

where the coefficients (11) and (12) change as follows:

$$a_E = \frac{\lambda_e}{\Delta x^2} \quad a_W = \frac{\lambda_w}{\Delta x^2} \quad a_N = \frac{\lambda_n}{\Delta y^2} \qquad a_S = 0 \tag{15}$$

$$a_P = \rho c_p \frac{1}{\Delta t} + a_E + a_W + a_N + \frac{\lambda_s}{\Delta y^2} \quad b = \rho c_p \frac{T_P^n}{\Delta t} + T_{bottom} \frac{\lambda_s}{\Delta y^2} \tag{16}$$

- Top (Neumann):

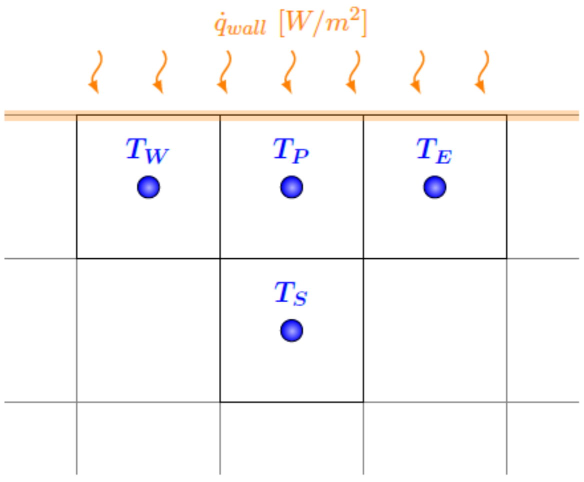

Figure 4: Neumann B.C. at the top wall.

Uniform heat flux of 60 W/m, resulting in:

$$\rho c_p \frac{T_P^{n+1} - T_P^n}{\Delta t} = \lambda_e \frac{T_E - T_P}{\Delta x^2} + \lambda_w \frac{T_W - T_P}{\Delta x^2} + \frac{\dot{Q}_{\text{wall}}}{\Delta y^2} + \lambda_s \frac{T_S - T_P}{\Delta y^2} \tag{17}$$

where the coefficients (11) and (12) change as follows:

$$a_E = \frac{\lambda_e}{\Delta x^2} \quad a_W = \frac{\lambda_w}{\Delta x^2} \quad a_N = 0 \quad a_S = \frac{\lambda_s}{\Delta y^2} \tag{18}$$

$$a_P = \rho c_p \frac{1}{\Delta t} + a_E + a_W + a_S \quad b = \rho c_p \frac{T_P^n}{\Delta t} + \frac{\dot{Q}_{\text{wall}}}{\Delta y} \tag{19}$$

- Left (Convection):

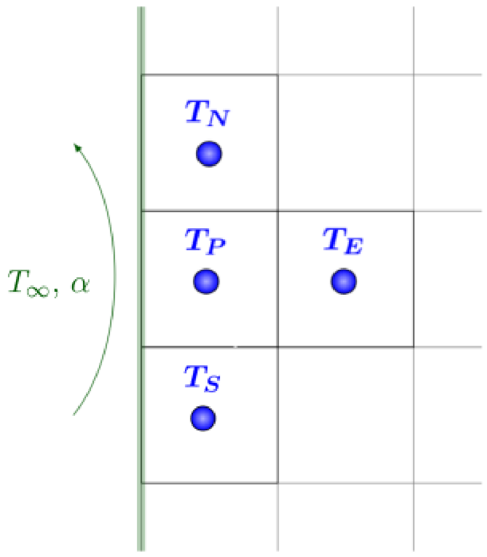

Figure 5: Convection B.C. at left wall.

Convective heat transfer with $T_f = 33$°C and $\alpha = 9$ W/m^{2}K. For simplicity, the temperature of the node P has been considered equal to the temperature of the wall, resulting in:

$$\rho c_p \frac{T_P^{n+1} - T_P^n}{\Delta t} = \lambda_e \frac{T_E - T_P}{\Delta x^2} + \alpha \frac{T_f - T_P}{\Delta x} + \lambda_n \frac{T_N - T_P}{\Delta y^2} + \lambda_s \frac{T_S - T_P}{\Delta y^2} \tag{20}$$

where T_f is the temperature of the external fluid and $T_P = T_{wall}$. The coefficients (11) and (12) change as follows:

$$a_E = \frac{\lambda_e}{\Delta x^2} \quad a_W = 0 \quad a_N = \frac{\lambda_n}{\Delta y^2} \quad a_S = \frac{\lambda_s}{\Delta y^2} \tag{21}$$

$$a_P = \rho c_p \frac{1}{\Delta t} + a_E + a_N + a_S + \frac{\alpha}{\Delta x} \quad b = \rho c_p \frac{T_P^n}{\Delta t} + \alpha \frac{T_f}{\Delta x} \tag{22}$$

- Right (Dirichlet):

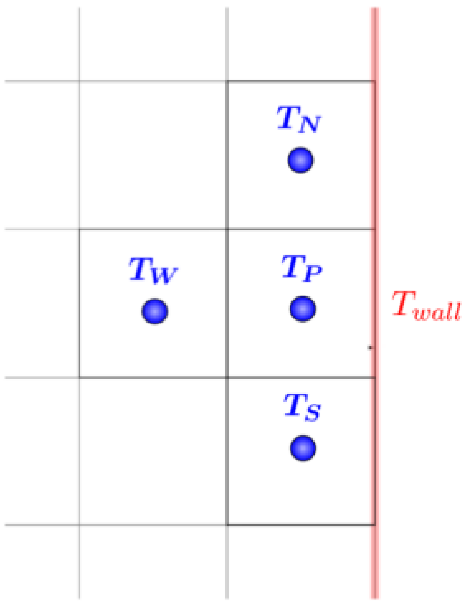

Figure 6: Dirichlet B.C. at right wall.

Linearly increasing temperature $T_{right} = 8 + 0.005t$, resulting in:

$$\rho c_p \frac{T_P^{n+1} - T_P^n}{\Delta t} = \lambda_e \frac{T_{right} - T_P}{\Delta x^2} + \lambda_w \frac{T_W - T_P}{\Delta x^2} + \lambda_n \frac{T_N - T_P}{\Delta y^2} + \lambda_s \frac{T_S - T_P}{\Delta y^2} \tag{23}$$

where the coefficients (11) and (12) change as follows:

$$a_E = 0 \quad a_W = \frac{\lambda_w}{\Delta x^2} \quad a_N = \frac{\lambda_n}{\Delta y^2} \qquad a_S = \frac{\lambda_s}{\Delta y^2} \tag{24}$$

$$a_P = \rho c_p \frac{1}{\Delta t} + a_W + a_N + a_S + \frac{\lambda_e}{\Delta x^2} \quad b = \rho c_p \frac{T_P^n}{\Delta t} + T_{right} \frac{\lambda_e}{\Delta x^2} \tag{25}$$

The initial condition is $T_0 = 8°\text{C}$ throughout the domain.

4 Algorithm Structure

The computational domain is divided into four distinct regions, each characterized by different thermal properties. The function `MaterialMatrix()` assigns a material index to each grid point based on its position. The classification of each point is determined using the blocked-off method, which states that a control volume is entirely assigned to a material if its central point falls within that material. This approximation eliminates the need to weight the contributions of density and heat capacity for control volumes located at material interfaces. Once the material indices are assigned, the initial temperature at each grid point is set to the specified starting value.
To numerically solve the heat conduction equation, several matrices are used:

- **Temperature matrices**: store the temperature field at different time steps.

- **Material matrix**: stores material indices, allowing retrieval of thermal properties such as thermal conductivity, density and heat capacity.
- **Coefficient matrices**: contain the discretized terms of the finite volume equation, such as heat conduction between neighboring nodes and source terms. Thermal conductivity coefficients are computed using the harmonic mean approach to ensure smooth property transitions between materials.

The **Gauss-Seidel** iterative method is used for numerical resolution. The temperature of each grid node is updated based on the temperatures of its neighbors. The iteration continues until the difference between two successive steps falls below a predefined tolerance, ensuring convergence. A residual is computed to track the maximum temperature change in the domain, and the algorithm stops when this value is sufficiently small.
Since the simulation is time-dependent, a time-stepping loop repeatedly executes the procedure over multiple time steps. At each iteration:

1. Boundary conditions are updated, considering convection, constant heat flux, and prescribed temperatures at the walls.
2. The Gauss-Seidel solver is executed to update the temperature field.
3. The results are saved to output files for visualization.
4. Execution time is monitored to evaluate the performance of the code.

At the end of the simulation, the program saves the temperature distribution and generates a script for visualization using Gnuplot.

5 Results

Temperature evolution is recorded at points (0.65, 0.56) and (0.74, 0.72), and temperature maps are generated for different time steps.
The following table provides temperature data at key points:

Time (s)	T_1 at (0.65, 0.56) [°C]	T_2 at (0.74, 0.72) [°C]
0	8.00	8.00
1000	12.21	11.00
2000	16.38	15.57
3000	19.62	19.34
4000	22.38	22.64
5000	24.90	25.71

Table 2: Temperature evolution at selected points.

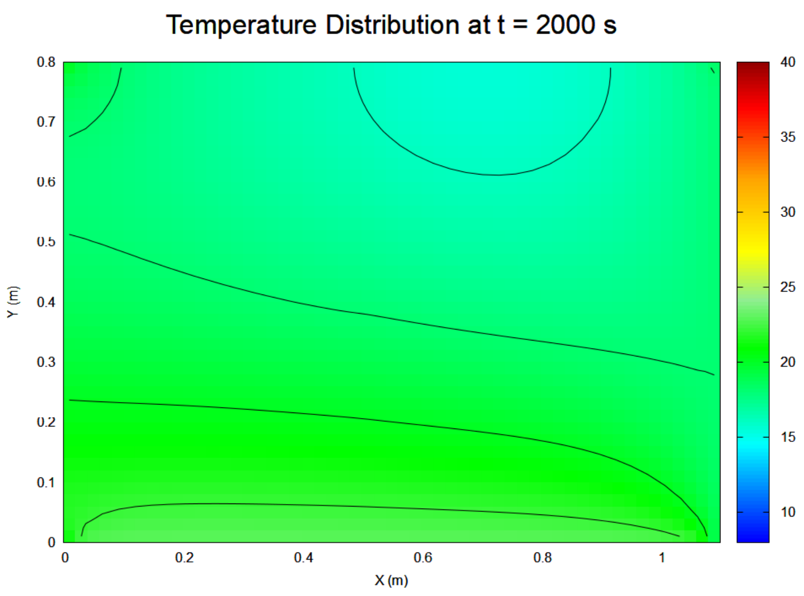

Figure 7: Temperature distribution at t = 2000 s.

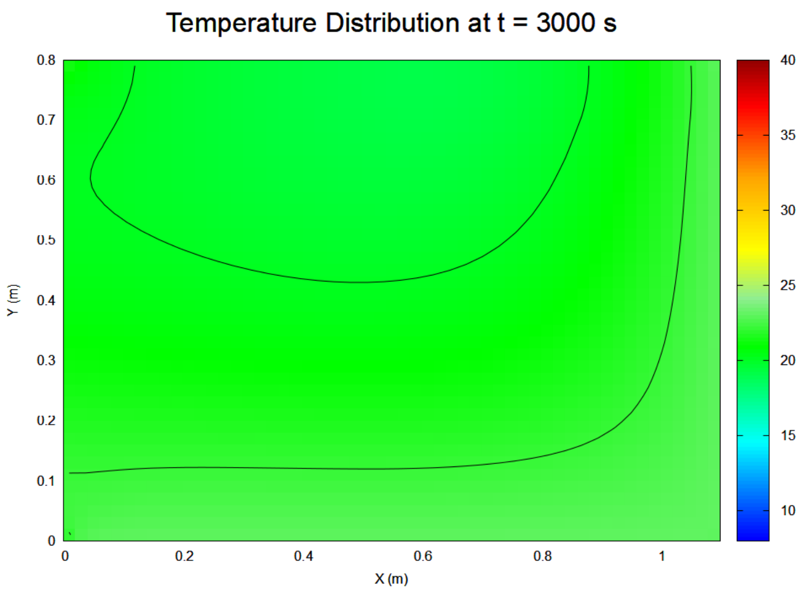

Figure 8: Temperature distribution at t = 3000 s.

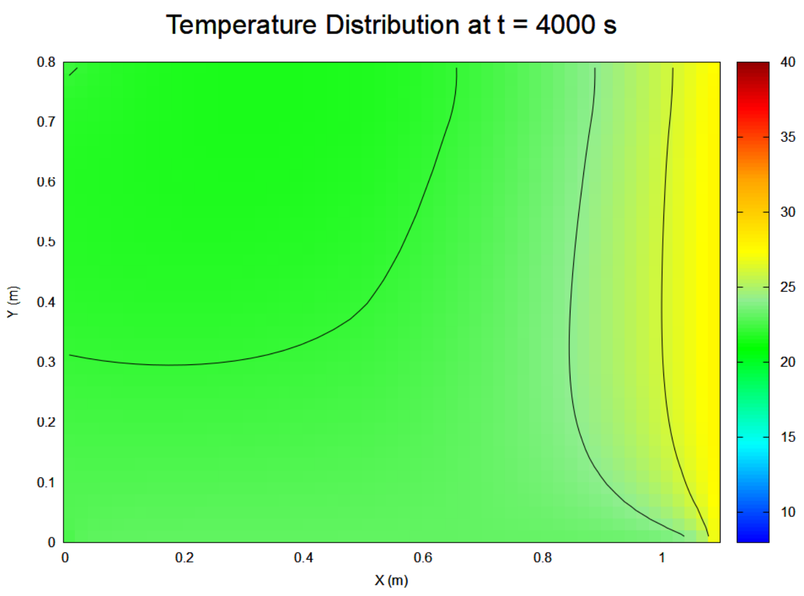

Figure 9: Temperature distribution at t = 4000 s.

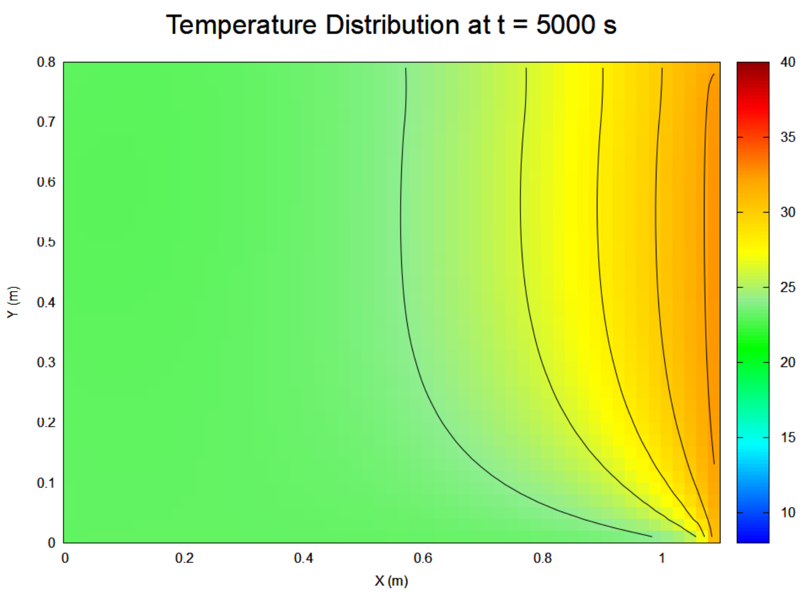

Figure 10: Temperature distribution at t = 5000 s.

6 Conclusion

The results obtained are well aligned with the expected ones, both for the target points and the temperature maps. The temperature evolution over time accurately reflects the applied boundary conditions and the behavior of the different materials.

Appendix: C++ Code

```
// 2-D Heat Conduction Problem in a Composite Rod
```

```
#include "Heat_Conduction_2D.h"
#include <iostream>
#include <fstream>
#include <iomanip>
#include <chrono>
#include <cmath>
#include <cstdlib>

using namespace std;
using namespace std::chrono;

// Geometry
double const xM1 = 0.5, yM1 = 0.4;
double const xM2 = 0.6, yM2 = 0.7;
double const xM3 = 0.5, yM3 = 0.4;
double const xM4 = 0.6, yM4 = 0.1;

// Target Points
double xP1 = 0.65;
double yP1 = 0.56;
double xP2 = 0.74;
double yP2 = 0.72;

// Boundary Conditions
double const Tf = 33 + 273.15, alpha = 9, Tb = 23 + 273.15, Qf = 60;

// Thermo-Physical Properties
double const rho1 = 1500, rho2 = 1600, rho3 = 1900, rho4 = 2500;
double const cp1 = 750, cp2 = 770, cp3 = 810, cp4 = 930;
double const lambda1 = 170, lambda2 = 140, lambda3 = 200, lambda4 = 140;

// Numerical Data
const int Nx = 55, Ny = 40;
double Dx = (0.5 + 0.6) / Nx, Dy = 0.8 / Ny;
double dt = 1.0;
double t_start = 0.0;
double Interval = 2000;
int maxIte = 1e6;
double maxRes = 1e-6;

// Vectors Definition
int Mat[Ny][Nx]; // Material matrix
double T[Ny][Nx], T1[Ny][Nx], Tg[Ny][Nx]; // Temperature matrix
double aP[Ny][Nx], aE[Ny][Nx], aW[Ny][Nx],
aN[Ny][Nx], aS[Ny][Nx], bP[Ny][Nx];  // Coefficients

// Function to Compute Harmonic Mean
double HarmonicMean(double lambdaA, double lambdaB) {
    return 2.0 / ((1 / lambdaA) + (1 / lambdaB));
}

// Function to Fill the Material Matrix and Initialize the Temperature
void MaterialMatrix() {
    int jM1 = static_cast<int>(round(xM1 / Dx));
    int iM3 = static_cast<int>(round(yM3 / Dy));
    int iM4 = static_cast<int>(round(yM4 / Dy));

    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx; j++) {
            if (i <= iM3 && j < jM1)
                Mat[i][j] = 3;  // M3
            else if (i > iM3 && j < jM1)
                Mat[i][j] = 1;  // M1
            else if (i < iM4 && j >= jM1)
                Mat[i][j] = 4;  // M4
            else
                Mat[i][j] = 2;  // M2

            cout << Mat[i][j] << " ";
        }
        cout << endl;
    }
}


// Function to Evaluate Coefficients
void InternalNodesCoefficients() {
    for (int i = 1; i < Ny-1; i++) {
        for (int j = 1; j < Nx-1; j++) {
            double lambda_mat[5] = {0, lambda1, lambda2, lambda3, lambda4};
            double rho_cp[5] = {0, rho1 * cp1, rho2 * cp2, rho3 * cp3,
                                rho4 * cp4};

            double rhocpP = rho_cp[Mat[i][j]];
            double lambdaP = lambda_mat[Mat[i][j]];

            double lambdaE = HarmonicMean(lambdaP, lambda_mat[Mat[i][j+1]]);
            double lambdaW = HarmonicMean(lambdaP, lambda_mat[Mat[i][j-1]]);
            double lambdaN = HarmonicMean(lambdaP, lambda_mat[Mat[i-1][j]]);
            double lambdaS = HarmonicMean(lambdaP, lambda_mat[Mat[i+1][j]]);

            aE[i][j] = lambdaE / (Dx * Dx);
            aW[i][j] = lambdaW / (Dx * Dx);
            aN[i][j] = lambdaN / (Dy * Dy);
            aS[i][j] = lambdaS / (Dy * Dy);
            aP[i][j] = rhocpP / dt + aE[i][j] + aW[i][j] + aN[i][j] + aS[i][j];
        }
    }
}

// Function for Boundary Conditions and Source Term
void BoundaryConditions(double t) {
    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx; j++) {
            double lambda_mat[5] = {0, lambda1, lambda2, lambda3, lambda4};
            double rho_cp[5] = {0, rho1 * cp1, rho2 * cp2, rho3 * cp3,
                                rho4 * cp4};

            double rhocpP = rho_cp[Mat[i][j]];
            double lambdaP = lambda_mat[Mat[i][j]];

            if (j == 0)
            // Convection at Left
                aE[i][j] = lambdaP / (Dx * Dx),
                aW[i][j] = 0,
                aN[i][j] = lambdaP / (Dy * Dy),
                aS[i][j] = lambdaP / (Dy * Dy),
                aP[i][j] = rhocpP / dt + aE[i][j] + aN[i][j] + aS[i][j]
                           + alpha / Dx,
                bP[i][j] = T[i][j] * rhocpP / dt + alpha * Tf / Dx;

            else if (j == Nx-1)
            // Variable T at Right
                aE[i][j] = 0,
                aW[i][j] = lambdaP / (Dx * Dx),
                aN[i][j] = lambdaP / (Dy * Dy),
                aS[i][j] = lambdaP / (Dy * Dy),
                aP[i][j] = rhocpP / dt + aW[i][j] + aN[i][j] + aS[i][j]
                           + lambdaP / (Dx * Dx),
                bP[i][j] = T[i][j] * rhocpP / dt
                           + (8 + 0.005 * t + 273.15) * lambdaP / (Dx * Dx);

            else if (i == 0)
            // Constant Flux at the Top
                aE[i][j] = lambdaP / (Dx * Dx),
               aW[i][j] = lambdaP / (Dx * Dx),
               aN[i][j] = 0,
               aS[i][j] = lambdaP / (Dy * Dy),
               aP[i][j] = rhocpP / dt + aE[i][j] + aW[i][j] + aS[i][j],
               bP[i][j] = T[i][j] * rhocpP / dt + Qf / Dy;

            else if (i == Ny-1)
            // Constant T at the Base
                aE[i][j] = lambdaP / (Dx * Dx),
                aW[i][j] = lambdaP / (Dx * Dx),
                aN[i][j] = lambdaP / (Dy * Dy),
                aS[i][j] = 0,
                aP[i][j] = rhocpP / dt + aE[i][j] + aW[i][j] + aN[i][j]
                           + lambdaP / (Dy * Dy),
                bP[i][j] = T[i][j] * rhocpP / dt + Tb * lambdaP / (Dy * Dy);

            else
            // Source Term for Internal Nodes
                bP[i][j] = T[i][j] * rhocpP / dt;
        }
    }
}

// Gauss-Seidel Solver
void GaussSeidelSolver() {
    // Time Loop
    double res = maxRes + 1;  // Condition to enter the loop
    int ite = 0;
    while (res > maxRes && ite < maxIte) {
        double maxDiff = 0.0;
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx; j++) {
                T1[i][j] = (aE[i][j] * T1[i][j+1] + aW[i][j] * T1[i][j-1] +
                            aN[i][j] * T1[i-1][j] + aS[i][j] * T1[i+1][j] +
                            bP[i][j]) / aP[i][j];
                double diff = fabs(T1[i][j] - Tg[i][j]);
                if (diff > maxDiff) {
                    maxDiff = diff;
                }
            }
        }
        res = maxDiff;
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx; j++) {
                Tg[i][j] = T1[i][j];
            }
        }
        ite++;
    }
}

int main() {
    // Initial Temperature at t = 0;
    double T0 = 8 + 273.15;

    // Start Timer
    auto start = high_resolution_clock::now();
```

```
// Find Indexes of Target Points
int jP1 = static_cast<int>(round(xP1 / Dx));
int iP1 = static_cast<int>(round((0.8 - yP1) / Dy));
int jP2 = static_cast<int>(round(xP2 / Dx));
int iP2 = static_cast<int>(round((0.8 - yP2) / Dy));

// Initial Map
for (auto& row : T) {
    for (auto& elem : row) {
        elem = T0;
    }
}

MaterialMatrix(); // Compute Matrix
InternalNodesCoefficients(); // Compute Internal Nodes

// File to Print Results
ofstream TargetPoints("Exercise2_Results.txt");
ofstream TemperatureMap("Exercise2_TemperatureMap_Data.txt");
ofstream TemperatureDistribution("Exercise2_TemperatureDistribution_Data.txt");

for (double t = 0.0; t <= Interval; t += dt) {

    BoundaryConditions(t); // Compute Boundary Nodes and Source Term
    GaussSeidelSolver(); // Evaluate Temperature Profile

    // Output results
    if (t == 0 or t == 1000 or t == 2000 or t == 3000 or
        t == 4000 or t == 5000) {
        TargetPoints << t << " " << T[iP1][jP1] - 273.15 << " "
                     << T[iP2][jP2] - 273.15<< endl;
    }

    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx; j++) {
            T[i][j] = T1[i][j];
        }
    }
}

// Temperature Map at t = Interval
for (int i = 0; i < Ny; i++) {
    for (int j = 0; j < Nx; j++) {
        TemperatureMap <<Dx/2 + j * Dx << " "
                       << (Ny - i - 1) * Dy + Dy/2<< " "
                       << T[i][j] - 273.15 << endl;
        TemperatureDistribution << T[i][j] - 273.15 << " ";
    }
    TemperatureDistribution << endl;
}

TargetPoints.close();
TemperatureMap.close();
TemperatureDistribution.close();

//Gnuplot
ofstream GnuPlot("TemperatureMap_Plot.plt");

GnuPlot << "set terminal pngcairo size 1000,700 enhanced font
           'Arial,12'\n";
GnuPlot << "set output 'TemperatureMap_Plot.png'\n";

GnuPlot << "set pm3d at b\n";
GnuPlot << "set dgrid3d 55,40 splines\n";
GnuPlot << "set cbrange [8:40]\n";
GnuPlot << "set palette defined ("
        << "0 'blue', "
        << "0.2 'cyan', "
        << "0.4 'green', "
        << "0.5 'light-green', "
        << "0.6 'yellow', "
        << "0.75 'orange', "
        << "0.9 'red', "
        << "1 'dark-red')\n";
GnuPlot << "set colorbox\n";

GnuPlot << "set xlabel 'X (m)'\n";
GnuPlot << "set ylabel 'Y (m)'\n";
GnuPlot << "set title 'Temperature Distribution at t = 2000 s' font
           'Arial,24'\n";
GnuPlot << "set xrange [0:1.1]\n";

GnuPlot << "set contour base\n";
GnuPlot << "set cntrparam levels incremental 10,2,40\n";
GnuPlot << "set cntrlabel start 5 interval 5 format '%2.0f°C' font
           'Arial,10'\n";
GnuPlot << "unset surface\n";
GnuPlot << "set view map\n";

GnuPlot << "set table 'contour_data.txt'\n";
GnuPlot << "splot 'Exercise2_TemperatureMap_Data.txt' using 1:2:3 with
            lines\n";
GnuPlot << "unset table\n";

GnuPlot << "unset pm3d\n";
```

```
    GnuPlot << "set multiplot\n";

    GnuPlot << "plot 'Exercise2_TemperatureMap_Data.txt' using 1:2:3 with
                image notitle\n";

    GnuPlot << "replot 'contour_data.txt' with lines lc rgb 'black' lw 1
                notitle\n";

    GnuPlot << "unset multiplot\n";

    GnuPlot.close();

    int result = system("TemperatureMap_Plot.plt");
    if (result != 0) {
        cerr << "Error during the execution of Gnuplot!" << endl;
        return 1;
    }

    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<seconds>(stop - start);  // Total Duration

    cout << "Transient simulation complete. Results saved to
             Exercise2_Results.txt" << endl;
    cout << "Total Execution Time: " << duration.count() << " seconds" << endl;

    return 0;
}
```