ENGR 313/3202
Engineering Analysis and Computation I
Spring 2025

Submitted to:
Dr. Mohamed Rashed Helmy

Submitted by:
Ismail Adel — 900221909
Omar Badawy — 900212151
Karim Elmasry — 900222714
Omar Mohammed — 900221999

List of Contribution

Name	Tasks
Ismail Adel	• Gauss Seidel function with Relaxation • Testing function (Extra) • Abstract • Background Review Section
Omar Badawy	• Main script code • Steady State function • Inverse function • Difficulties Section • Conclusion • Flowchart
Karim ElMasry	• Main Script Code • Steady State function • Testing function (Extra) • Gauss-Seidel Background • Inverse Function Background
Omar Mohammed	• Gauss Elimination function • Plotting function • Function Testing Section • Code Manual Section • Gauss-Elimination Background • Flowchart

Abstract

The purpose of this program is to create a MATLAB program that calculates the temperature of nodes inside a square 2D plate at steady state, subject to boundary conditions at the four sides. The code is designed to be modular, with different functions including iterative techniques, steady-state equations, and plotting. First, the user is asked to input many initial values; the temperature at each of the four sides, the dimensions of the plate, the number of nodes to be solved for, the technique to be used for solving the matrix (Gaussian Elimination, Gauss-Seidel, or Inverse), and in the case of Gauss-Seidel the stopping criteria. Then, the steady state function uses the plate dimensions and chosen number of nodes to construct a matrix. This matrix is fed into the user-selected matrix-solving function to be solved within a certain margin of error (also user-selected). Finally, the outputs are displayed, once in a table with the x,y, and steady-state temperature of each node, and second in a heat map.

Contents

List of Figures

Listings

1 Background

Steady-state heat transfer in 2D plates is a critical challenge in engineering, especially for aerospace components like airplane parts that endure extreme temperature changes and harsh operating conditions. The project involves predicting temperature distribution across a plate governed by partial differential equations where analytical solutions are infeasible, necessitating numerical methods. Key challenges include handling diverse boundary conditions—fixed temperatures, insulated edges, or convective cooling—and discretizing the plate into a grid of nodes to solve the system. A modular computational solution is essential to separate tasks like grid generation, boundary application, matrix-based equation solving, and visualization, ensuring adaptability for varied aerospace designs such as turbine blades or composite materials.

2 Numerical Methods

2.1 Gauss Elimination

Gauss elimination is a systematic method of solving linear algebraic equations. It involves transforming a given system into an augmented matrix, then into an *upper triangular matrix*, hence solving it with back substitution. A given system is represented in the following form:

$$[A]\{x\} = [B]$$

Where $[A]$ is an $n \times n$ coefficient matrix, $\{x\}$ is a column vector of variables, and $[B]$ is the right-hand-side column vector. The augmented matrix could be better visualised as follows:

$$\left[\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} & b_2 \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} & b_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} & b_n \end{array}\right]$$

A crucial step in gauss elimination is *pivoting*, which involves swapping rows such as the greatest element becomes *pivot element*. Pivoting is important not only because it prevents dividing zero in the following step, but also because it ensures we divide by the greatest values, which yields higher numerical accuracy. Then, forward elimination is performed, where we make all entries below the pivot element (diagonal) in each column zero by subtracting appropriate multiples of the pivot row from the rows below.

$$R_i \Rightarrow R_i - kR_j$$
$$k = \frac{a_{ij}}{a_{jj}}$$

Where R_i is the pivot row, R_j is the row we are eliminating, a_{jj} is the pivot element, and a_{jj} is the element we want to eliminate. Pivoting is repeated before eliminating a new column. An upper triangular matrix is hence resulted:

$$\left[\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & b_1 \\ 0 & a'_{22} & a'_{23} & \cdots & a'_{2n} & b'_2 \\ 0 & 0 & a''_{33} & \cdots & a''_{3n} & b''_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn}^{(n-1)} & b_n^{(n-1)} \end{array}\right]$$

Notice that by the end of forward elimination, each row R_n would have been recalculated $n-1$ times. Starting from the last row, we solve for $\{x\}$ via back substitution. For a system whose x vector consists of three elements, the equations for back substitution are shown bellow. The solutions are *exact* **(Rashed 2025)**.

$$x_3 = \frac{c''_3}{a''_{33}}$$
$$x_2 = \frac{c'_2 - a'_{23}x_3}{a'_{22}}$$
$$x_1 = \frac{c_1 - a_{12}x_2 - a_{13}x_3}{a_{11}}$$

2.2 Gauss-Seidel

Gauss Seidel is an iterative technique used to solve systems of linear equations. For a system of equations whose unknown variables are x_1, x_2, and x_3, Gauss-Seidel derives an equation with respect to these variables, as shown below.

$$x_1 = \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3)$$
$$x_2 = \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3)$$
$$x_3 = \frac{1}{a_{33}}(b_3 - a_{31}x_1 - a_{32}x_2)$$

For the first iteration, initial guesses for x_1, x_2, and x_3 are used. These guesses are substituted into the system of equations. Each time an equation is solved, the value of the corresponding variable must be updated immediately so it can be used in the subsequent equations. This is essential because the Gauss-Seidel method relies on the most recent values.

If the system converges and the coefficient matrix is diagonally dominant, the accuracy of the method improves with each iteration. A **diagonally dominant matrix** is one in which each diagonal element is greater in absolute value than the sum of the absolute values of the other elements in the same row.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \quad \text{where} \quad \begin{cases} |a_{11}| \geq |a_{12}| + |a_{13}| \\ |a_{22}| \geq |a_{21}| + |a_{23}| \\ |a_{33}| \geq |a_{31}| + |a_{32}| \end{cases}$$

If the system is not diagonally dominant, the results are usually more inaccurate and unpredictable.

The Gauss-Seidel method is extensively used in engineering fields where large systems of linear equations arise naturally from physical models

2.3 Inverse Method

To solve a system of linear equations using the Matrix Inverse method, the inverse of matrix A must first be determined. Once the inverse is obtained, it is multiplied by the system as shown below.

$$A^{-1}(A \cdot x) = A^{-1}b$$

Then, using the property of inverses, multiplying a matrix A by its inverse A^{-1} yields the identity matrix I. Alternatively, by definition:

$$I \cdot x = A^{-1}b$$

$$A \cdot A^{-1} = A^{-1} \cdot A = I$$

where I is the identity matrix.

A matrix multiplied by an identity matrix remains the same. This results in the final solution of B multiplied by the inverse matrix. The solution B is an *exact* solution.

3 Functions Testing

3.1 Gauss Elimination

```
function X = gaussElim(A, B)

% Inputs: Matrices A & B
% Output: solution vector X

  [n, m] = size(A);

  % Augment the matrix A with B
  Ab = [A, B];

  % Forward elimination with partial pivoting
  for k = 1:n-1
      % Find the row with the largest absolute value in the current column
      [~, maxRow] = max(abs(Ab(k:n, k)));
      maxRow = maxRow + k - 1;  % Correct the index

      % Swap the current row with the row with the largest absolute value
      if maxRow ~= k
          Ab([k, maxRow], :) = Ab([maxRow, k], :);
      end

      % Eliminate the entries below the pivot
      for i = k+1:n
          factor = Ab(i, k) / Ab(k, k);
          Ab(i, k:n+1) = Ab(i, k:n+1) - factor * Ab(k, k:n+1);
      end
  end

  % Back substitution
  X = zeros(n, 1);
  X(n) = Ab(n, n+1) / Ab(n, n);
  for i = n-1:-1:1
      X(i) = (Ab(i, n+1) - Ab(i, i+1:n) * X(i+1:n)) / Ab(i, i);
  end
end
```

Listing 1: Gauss Elimination Function

For testing the function, `function X = gaussElim (A , B)` and its corresponding `end` are commented, and the following A and B matrices are added to the beginning of the script:

```
A = [-8   1   8   5  -3;
      1   0  -3   3  -1;
      4   6  -2   3   1;
      7   9  -1   7  -9;
     -1   3   4   0   1];

B = [1;
     2;
     3;
     4;
     5];
```

The script is run, and when `Ab` and `X` are typed in the command window, their corresponding upper-triangular matrix and solution vector are displayed. All values are indeed correct.

```
>> gaussElim
>> Ab

Ab =

   -8.0000    1.0000    8.0000    5.0000   -3.0000    1.0000
         0    9.8750    6.0000   11.3750  -11.6250    4.8750
         0         0   -2.0759    3.4810   -1.2278    2.0633
         0         0         0   -5.2561    8.3049   -1.6463
         0         0         0         0    1.1183    5.2761

>> X

X =

   11.1369
   -8.5145
    9.2407
    7.7676
    4.7178
```

3.2 Inverse Method Function

```
function X = inverseMethod(A, B)
% Check if A is square and invertible
    if size(A,1) ~= size(A,2)
        error('Matrix A must be square.');
    end

    if det(A) == 0
        error('Matrix A is SINGULAR and cannot be inverted.');
    end
    % Compute solution using matrix inverse
    X = inv(A) * B;
end
```

Listing 2: Steady State Equation Function

The Inverse Method Function is identically tested, and it produces the exact same solution

3.3 Gauss-Seidel

```
function [X, relErr, avgErr, iterations] = gaussSeidel(A, B, tol, maxIter)

%tol = tolerence
%maxIter = maximum allowed no. of iterations

    n = length(B);
    X = zeros(n, 1);
    relErr = zeros(n, 1);

    % Check for zero diagonal elements
    if any(diag(A) == 0)
        error('Matrix has zero(s) on the diagonal. Gauss-Seidel cannot proceed.');
    end

    % Improved diagonal dominance check
    isDiagDominant = all(abs(diag(A)) > sum(abs(A), 2) - abs(diag(A)));
    if isDiagDominant
        lamda = 1.2; % over-relaxation
        fprintf('Matrix is diagonally dominant: using over-relaxation (w = %.2f)\n', lamda);
    else
        lamda = 0.8; % under-relaxation
        fprintf('Matrix is not diagonally dominant: using under-relaxation (w = %.2f)\n',
    lamda);
    end

    for iterations = 1:maxIter
        X_old = X;
        for i = 1:n
            sum1 = A(i, 1:i-1) * X(1:i-1);
            sum2 = A(i, i+1:n) * X_old(i+1:n);
            x_new = (B(i) - sum1 - sum2) / A(i, i);
            X(i) = lamda * x_new + (1 - lamda) * X_old(i);
        end

        % Relative error check
        relErr = abs((X - X_old) ./ (X + eps));
        avgErr = mean(relErr) * 100;

        if all(relErr < tol)
            break;
        end
    end
end
```

Listing 3: Gauss-Seidel Function

For testing the function, `function [X , relErr , avgErr , iterations ] = gaussSeidel (A , B , tol , maxIter )` and its corresponding `end` are commented, and the following variables are added to the beginning of the script:

```
A = [  3 -0.1 -0.2;
     0.1    7 -0.3;
     0.3 -0.2   10];

B = [7.85;
    -19.3;
     71.4];

tol = 0.0001;
maxIter = 1000;
```

The script is run, and when `X` is typed in the command window, its solution vector is displayed. All values are indeed correct.

```
>> gaussSeidel
Matrix is diagonally dominant: using over-relaxation (w = 1.20)
>> X

X =

    3.0000
   -2.5000
    7.0000
```

Note that if the diagonal of matrix A have any zeros, the function would fail to solve for X, nevertheless such scenario does not occur will running the main code.

3.4 Steady State Equation

```
function [PointDB, A, B] = steadyStateEq(topT, bottomT, leftT, rightT, L, N)

%   PointDB  - A database with rows: [point index, x coordinate, y coordinate]
%   A        - Coefficient matrix of the system (sparse matrix)
%   B        - Right-hand side vector of the system

    % Calculate total number of points Nodes^2
    Points=N^2;

    % Grid spacing
    dx = L / (N - 1);

    % Initialize outputs
    A = sparse(Points, Points);  % Use sparse for efficiency
    B = zeros(Points, 1);
    PointDB = zeros(Points, 3);       % [index, x, y]

    % Loop over each grid point
    for j = 1:N  % y-direction
        for i = 1:N  % x-direction
            idx = (j-1)*N + i;  % Point number in 1D
            x = (i-1) * dx;
            y = (j-1) * dx; %dx since its same as dy since its square

            % Store point data
            PointDB(idx, :) = [idx, x, y];

            % Check if the point is on the boundary
            if i == 1
                % Left boundary
                A(idx, idx) = 1;
                B(idx) = leftT;
            elseif i == N
                % Right boundary
                A(idx, idx) = 1;
                B(idx) = rightT;
                elseif j == 1
                % Bottom boundary
                A(idx, idx) = 1;
                B(idx) = bottomT;
            elseif j == N
                % Top boundary
                A(idx, idx) = 1;
                B(idx) = topT;
            else
                % Interior point: apply 5-point Laplacian stencil
                A(idx, idx) = -4;
                A(idx, idx - 1) = 1;     % left neighbor
                A(idx, idx + 1) = 1;     % right neighbor
                A(idx, idx - N) = 1;     % bottom neighbor
       A(idx, idx + N) = 1;     % top neighbor
     % B(idx) remains zero
            end
        end
    end
end
```

Listing 4: Steady State Equation Function

The Steady State Equation Function is tested on the following figure:

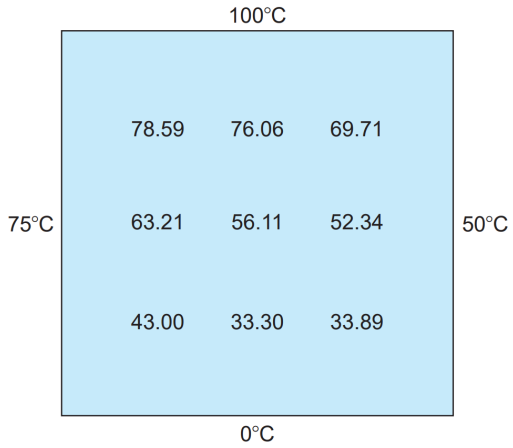


Figure 1: Testing figure — Adapted from [**Chapra 2010**], *Figure 29.5*.

The function is tested within the main script. All boundary conditions are input as below. More about using the main code in the **Code Manual** section.

Plate Dimensions:

- Length = 0.04 m × 0.04 m
- Number of nodes in the x direction = y direction = 5

Boundary Conditions:

- Temperature (east side) = 75 °C
- Temperature (west side) = 50 °C
- Temperature (north side) = 100 °C
- Temperature (south side) = 0 °C

The code is run, and the following temperature distribution is printed. All values indeed match the ones in the testing figure.

```
Temperature distribution at each point:
Index    X (m)    Y (m)    Temperature (C)
1    0.00    0.00    75.00
2    0.01    0.00    0.00
3    0.02    0.00    0.00
4    0.03    0.00    0.00
5    0.04    0.00    50.00
6    0.00    0.01    75.00
7    0.01    0.01    42.86
8    0.02    0.01    33.26
9    0.03    0.01    33.93
10    0.04    0.01    50.00
11    0.00    0.02    75.00
12    0.01    0.02    63.17
13    0.02    0.02    56.25
14    0.03    0.02    52.46
15    0.04    0.02    50.00
16    0.00    0.03    75.00
17    0.01    0.03    78.57
18    0.02    0.03    76.12
19    0.03    0.03    69.64
20    0.04    0.03    50.00
21    0.00    0.04    75.00
22    0.01    0.04    100.00
23    0.02    0.04    100.00
24    0.03    0.04    100.00
25    0.04    0.04    50.00
```

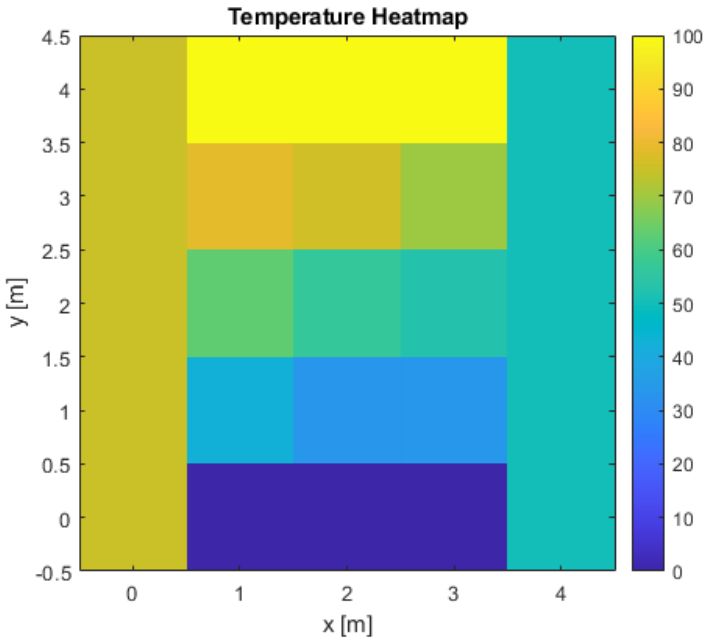


Figure 2: Steady State Function Test — Heat Map

4 Code Manual

Upon running the code, the user is asked whether he/she wants to *Test a Function* or to *Solve the Main Problem*, for which the user would type either '1' or '2' in the command window. If the user types in '1,' he/she is asked to choose a function. The function is tested in a manner similar to what is in the **Functions Testing** section. The user is asked whether to *restart the program* or *exit* after his/her desired operation is done.

```
Choose what you would like to do:
1: Test a Function
2: Solve the Main Problem
Enter the choice number: 1

Choose which function you would like to test:
1: Gauss Elimination
2: Gauss-Seidel
3: Inverse Method
4: SS eqn
5: Plotting
Enter function number: 1
Test for Gauss Elimination:

ans =

11.1369
-8.5145
9.2407
7.7676
4.7178
Would you like to:
1: Test another function
2: Continue to the main problem
Enter your choice: 2

Steady-State Heat Distribution Solver
Enter temperature North Side (C):
```

If the user chooses to solve the main problem, he/she is prompted for the following:

1. Temperature on the north side of the plate (°C).
2. Temperature on the south side of the plate (°C).
3. Temperature on the west side of the plate (°C).
4. Temperature on the east side of the plate (°C).
5. Length of the plate (m).
6. Desired number of nodes.
7. Desired method of solving the system of algebraic linear equations.

The user can enter any value for each of the prompts, except for the last one, where the user will have to enter either "1" for *Gauss Elimination*, "2" for *Gauss-Seidel*, or "3" for *Inverse Method*. Shortly, all x and y and their corresponding temperatures are printed and the heat map plotted.

Case Study

The data below is used to test the code. The code is tested three times — one for each solving method.

Plate Dimensions:

- Length = 1 m × 1 m
- Number of nodes in the x direction = y direction = 26

Boundary Conditions:

- Temperature (east side) = 150 °C
- Temperature (west side) = 300 °C
- Temperature (north side) = 50 °C
- Temperature (south side) = 100 °C

```
>> mainscript
Steady-State Heat Distribution Solver
Enter temperature North Side (C):
50
Enter temperature South Side (C):
100
Enter temperature West Side (C):
300
Enter temperature East Side (C):
150
Enter the length of one side of the plate (m):
1
Enter the number of nodes on x-direction (one side square grid):
26
Choose a method to solve the system:
1: Gauss Elimination
2: Gauss-Seidel
3: Inverse Method
Enter the method number:
1
Solving using Gauss Elimination...
Temperature distribution at each point:
Index   X (m)   Y (m)   Temperature (C)
1   0.00   0.00   300.00
2   0.04   0.00   100.00
```

Listing 5: Prompts in Command Window — Gauss Elimination

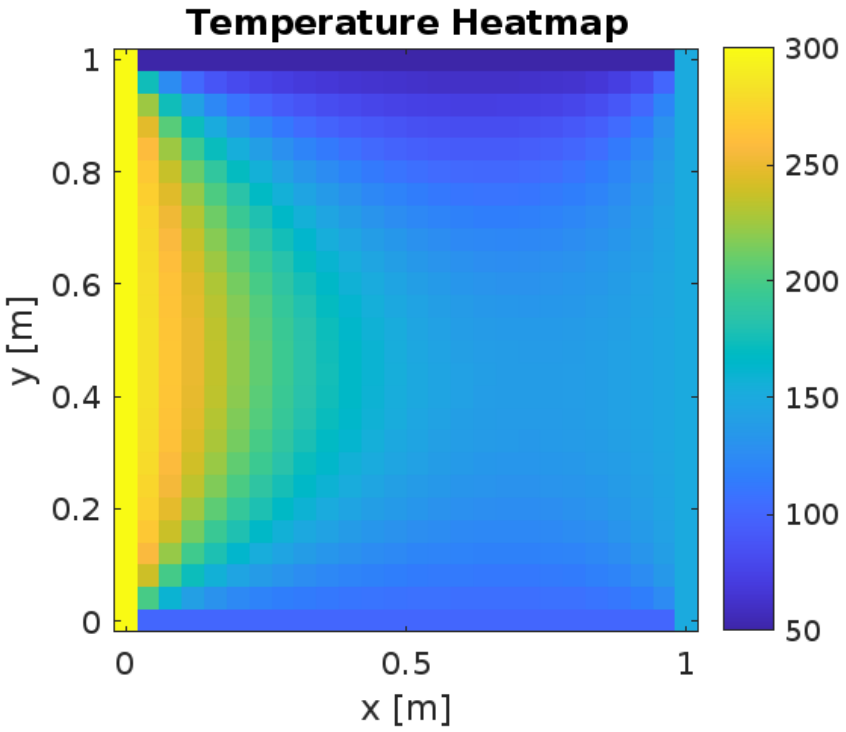


Figure 3: Gauss Elimination Heat Map

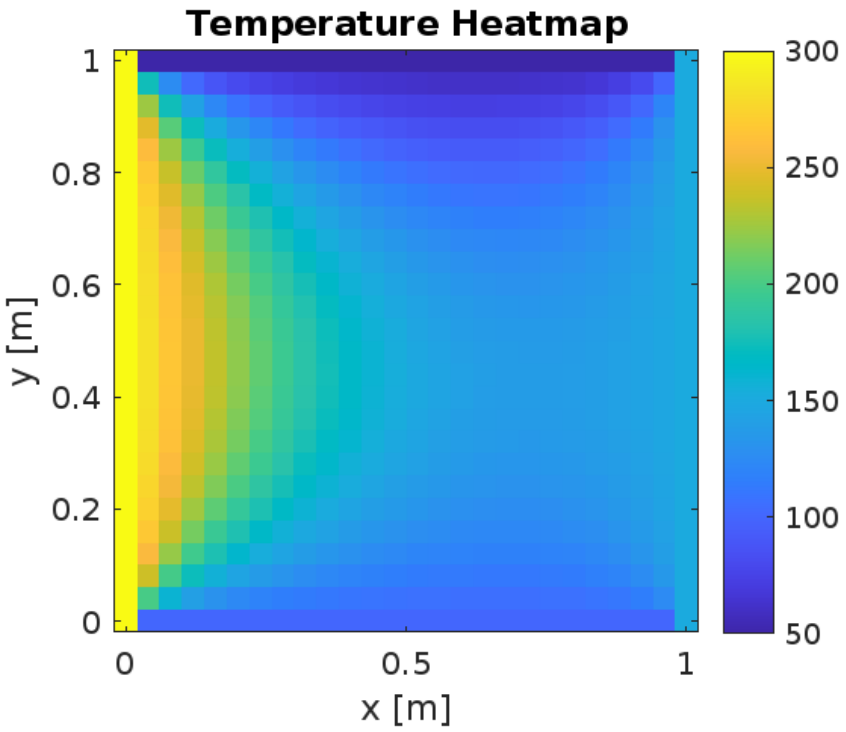


Figure 4: Inverse function Heat Map

If the user chooses *Gauss-Seidel* for solving the system, the user will be further prompted the following:

1. Acceptable relative error (%).
2. Maximum number of iterations.

The main code is run again using *Gauss-Seidel* with an acceptable relative error of 0.0001% and a maximum number of iterations of 1000.

```
>> mains
Steady-State Heat Distribution Solver
Enter temperature North Side (C):
50
Enter temperature South Side (C):
100
Enter temperature West Side (C):
300
Enter temperature East Side (C):
150
Enter the length of one side of the plate (m):
1
Enter the number of nodes on x-direction (one side square grid):
26
Choose a method to solve the system:
1: Gauss Elimination
2: Gauss-Seidel
3: Inverse Method
Enter the method number:
2
Enter the acceptable relative error (%):
0.0001
Enter the maximum number of iterations:
1000
Solving using Gauss-Seidel...
Matrix is not diagonally dominant: using under-relaxation (w = 0.80)
Converged in 491 iterations with average error 0.00%.

Temperature distribution at each point:
Index   X (m)   Y (m)   Temperature (C)
1   0.00   0.00   300.00
2   0.04   0.00   100.00
3   0.08   0.00   100.00
```

Listing 6: Prompts in Command Window — Gauss-Seidel

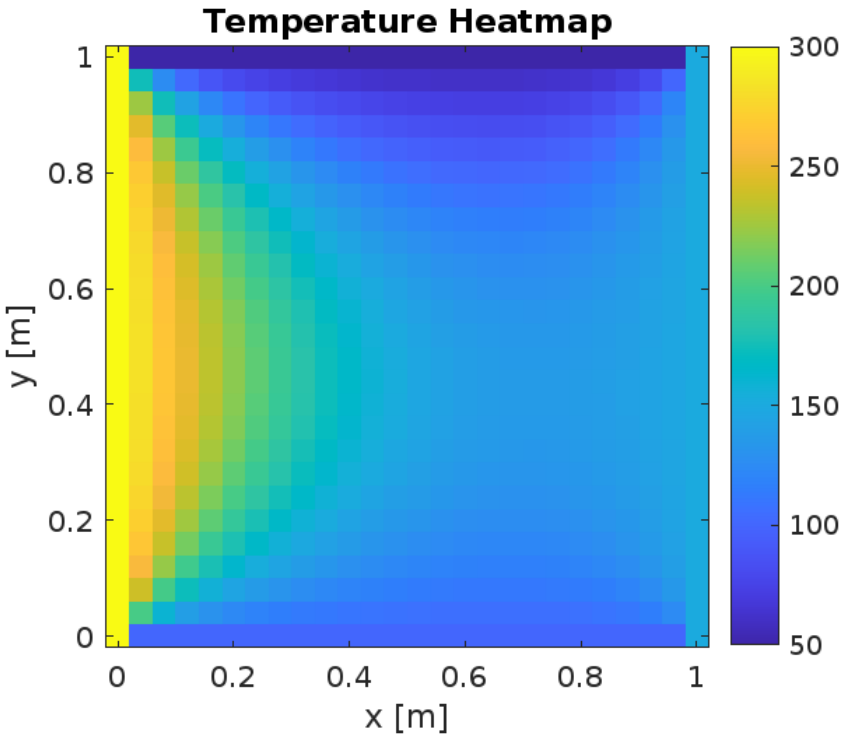
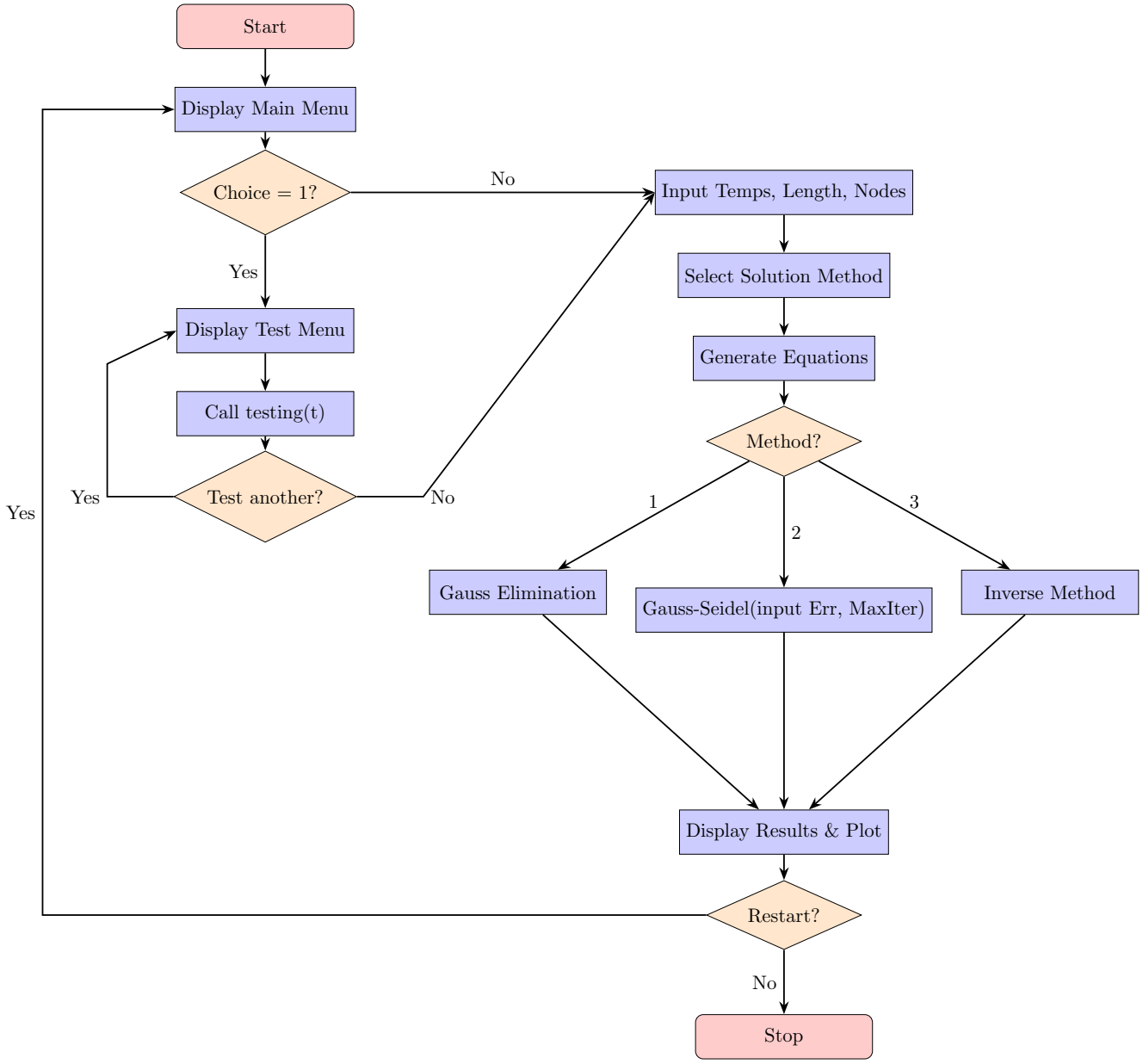


Figure 5: Gauss-Seidel Heat Map

5 Program Flowchart



6 Difficulties

Throughout the project, we faced common challenges, especially as some of us are still unfamiliar with how to gain the maximum power that MATLAB provides. Learning how to call external functions, test outputs, and debugging errors a lot took time and effort. Integrating different modules was also tricky due to varied inputs and outputs. However, with all the struggles, we overcame these issues by consulting documentation, using MATLAB's debugging tools, and breaking problems into smaller parts. Guidance from our professor and peers further helped us refine the logic and improve our code. Overall, these experiences significantly deepened our understanding and skills in MATLAB.

7 Conclusion

In conclusion, the MATLAB program developed for this project effectively calculates and visualizes the temperature distribution of a square 2D plate under specified boundary conditions. By implementing three numerical methods, Gauss Elimination, Gauss-Seidel, and the Inverse Matrix method, the program shows flexibility and adaptability for analyzing heat transfer problems. At the core of the program, the Steady State function plays a vital role as it processes user inputs, constructing a grid for the plate, and formulating the steady state heat equations using the Laplacian finite difference method. These equations are then efficiently solved using the user-chosen numerical method, with the resulting temperature data visualized through a heat map.

The modular design of the program not only promotes clarity and good maintenance but also provides a solid foundation for future enhancements. Potential improvements include expanding the program's capabilities to handle more complex geometries such as irregular boundaries or internal features. Moreover, optimizing the efficiency of the numerical methods, through increased use of MATLAB's built-in functions, could potentially significantly reduce computation time, especially for fine grids with a high number of nodes. Overall, this initial version demonstrates strong promise as a practical tool for engineering analysis for thermal systems.

References

[1] Chapra, S. C., & Canale, R. P. (2010). *Numerical Methods for Engineers* (6th ed.). McGraw-Hill.

[2] Rashed, M. (2025). Linear algebraic equations [Lecture slide — 10]. Engineering Analysis & Computation I (ENGR 3202), The American University in Cairo.