2D Pure Diffusion CFD Code

Finite Volume Solution of the 2D Diffusion Equation

António Reis

January 2025

Contents

List of Figures

1 Introduction

The present work presents a study of the energy conservation equation for an incompressible fluid under steady-state conditions. The modified governing differential equation (MGDE) was derived using the Method of Manufactured Solutions. The objective was to ensure that the analytical solution of the MGDE was consistent with the numerically obtained solution. To achieve this, the Finite Volume Method was applied to the MGDE, discretizing the differential equations over all cells of the defined computational mesh. A computational code was developed to solve the problem for varying numbers of cells in the x and y directions. The implemented code was tested using three different solvers: a direct method, the iterative Gauss-Seidel method, and the Gauss-Seidel method with OverRelaxation factor ω. The methods were compared in terms of accuracy relative to the analytical solution, computational time, memory usage, and convergence behavior as the mesh was refined. The results demonstrated that, with mesh refinement, the numerical solution converges toward the analytical solution, and also allowed for the evaluation of the efficiency and feasibility of each numerical approach.

2 Modified Governing Differential Equation (MGDE)

2.1 Energy Conservation Equation

The governing equation for the problem under study is the energy conservation equation. Since the case involves a steady-state regime and an incompressible fluid, there are no transient (time-dependent), advective, or source terms. As a result, the governing differential equation reduces to Eq.(2.1),

$$\nabla \cdot (k(x, y)\nabla T) = 0 \tag{2.1}$$

where the thermal conductivity, which may vary spatially in x and y, is given by Eq.(2.2):

$$k(x, y) = k_{\text{ref}} \left(k_x \cos\left(\frac{2\,\pi\,x}{L}\right) + k_y \sin\left(\frac{\pi\,y}{H}\right)\right) \tag{2.2}$$

which is shown to depend on the parameters k_i, the domain dimensions L and H, and the temperature T.

2.2 Method of Manufactured Solutions

The first objective of the project was the derivation of the Modified Governing Differential Equation (MGDE).

To achieve this, the Method of Manufactured Solutions was applied to ensure that the solution of the governing equation matched the prescribed analytical solution. The Method of Manufactured Solutions was carried out through the following steps:

1. An analytical solution was choosen, corresponding to Eq.(2.3)

$$T(x,y) = 50\,\cos\left(\frac{2\,\pi\,y}{H} + \frac{4\,\pi\,x}{L}\right) + 200 \tag{2.3}$$

2. The analytical solution was substituted into the governing equation of the problem, corresponding to Eq.(2.1), resulting in a new source term, given by Eq.(2.4).

$$\begin{aligned} F = &-\frac{400\,k_{\mathrm{ref}}\,k_x\,\pi^2\,\sin\left(\frac{2\pi x}{L}\right)\,\sin\left(\frac{2\pi y}{H}+\frac{4\pi x}{L}\right)}{L^2} \\ &+\frac{800\,k_{\mathrm{ref}}\,\pi^2\,\cos\left(\frac{2\pi y}{H}+\frac{4\pi x}{L}\right)\,\left(k_x\,\cos\left(\frac{2\pi x}{L}\right)+k_y\,\sin\left(\frac{\pi y}{H}\right)\right)}{L^2} \\ &+\frac{100\,k_{\mathrm{ref}}\,k_y\,\pi^2\,\cos\left(\frac{\pi y}{H}\right)\,\sin\left(\frac{2\pi y}{H}+\frac{4\pi x}{L}\right)}{H^2} \\ &+\frac{200\,k_{\mathrm{ref}}\,\pi^2\,\cos\left(\frac{2\pi y}{H}+\frac{4\pi x}{L}\right)\,\left(k_x\,\cos\left(\frac{2\pi x}{L}\right)+k_y\,\sin\left(\frac{\pi y}{H}\right)\right)}{H^2} \end{aligned} \tag{2.4}$$

3. The MGDE was obtained by adding the newly computed source term, Eq.(2.4), to Eq.(2.1). The MGDE is presented in Eq.(2.5).

$$\begin{aligned} \underline{\nabla}\cdot(k(x,y)\underline{\nabla}T) = &\frac{400\,k_{\mathrm{ref}}\,k_x\,\pi^2\,\sin\left(\frac{2\pi x}{L}\right)\,\sin\left(\frac{2\pi y}{H}+\frac{4\pi x}{L}\right)}{L^2} \\ &-\frac{800\,k_{\mathrm{ref}}\,\pi^2\,\cos\left(\frac{2\pi y}{H}+\frac{4\pi x}{L}\right)\,\left(k_x\,\cos\left(\frac{2\pi x}{L}\right)+k_y\,\sin\left(\frac{\pi y}{H}\right)\right)}{L^2} \\ &-\frac{100\,k_{\mathrm{ref}}\,k_y\,\pi^2\,\cos\left(\frac{\pi y}{H}\right)\,\sin\left(\frac{2\pi y}{H}+\frac{4\pi x}{L}\right)}{H^2} \\ &-\frac{200\,k_{\mathrm{ref}}\,\pi^2\,\cos\left(\frac{2\pi y}{H}+\frac{4\pi x}{L}\right)\,\left(k_x\,\cos\left(\frac{2\pi x}{L}\right)+k_y\,\sin\left(\frac{\pi y}{H}\right)\right)}{H^2} \end{aligned} \tag{2.5}$$

To compute the source term in Eq.(2.4), the *Symbolic Math Toolbox* of MATLAB was used to carry out the required symbolic algebra operations. The corresponding code can be found in AppendixA.

Also of critical importance for the development of the project was the determination of the partial derivatives of T, which are presented in Eqs.(2.6) and (2.7). These derivatives were likewise obtained using the MATLAB code provided in Appendix A.

$$\frac{\partial T}{\partial x} = -\frac{200\,\pi\,\sin\left(\frac{2\,\pi\,y}{H} + \frac{4\,\pi\,x}{L}\right)}{L} \tag{2.6}$$

$$\frac{\partial T}{\partial y} = -\frac{100\,\pi\,\sin\left(\frac{2\,\pi\,y}{H} + \frac{4\,\pi\,x}{L}\right)}{H} \tag{2.7}$$

3 Application of the Finite Volume Method to the MGDE

To obtain the numerical solution of the problem under study, the Finite Volume Method (FVM) was employed. This method is widely used in the simulation of fluid flow and heat transfer. It is based on the discretization of the domain into small control volumes, over which the conservation equations are applied in their integral form, ensuring both local and global conservation of physical quantities. Its application allows for the treatment of various boundary conditions, as is the case in the present study.

3.1 Derivation and Discretization of the MGDE for the General Case

The obtained MGDE, Eq.(2.5), was rewritten in the following form in order to simplify its application within the FVM framework:

$$\underline{\nabla} \cdot (k(x,y)\underline{\nabla} T) + F = 0 \tag{3.1}$$

As a first step in applying the method, Eq.(3.1) was integrated over the control volume, yielding:

$$\iiint_V \underline{\nabla} \cdot (k(x,y)\underline{\nabla} T)\, dV + \iiint_V F\, dV = 0 \tag{3.2}$$

Then, by applying the Gauss Divergence Theorem, the volume integral was converted into a surface integral.

$$\int_A \vec{n} \cdot (k(x,y)\underline{\nabla} T)\, dA + F\Delta V = 0 \tag{3.3}$$

which can then be approximated as a summation of the fluxes through the faces f of the control volume.

$$\sum_f (\vec{n} \cdot (k(x,y)\underline{\nabla} T)_f A_f) + F\Delta V = 0 \tag{3.4}$$

This summation was expanded over the four faces of the control volume: East (E), West (W), North (N), and South (S):

$$\left(k_E \frac{\partial T}{\partial x} A_E\right)_E - \left(k_W \frac{\partial T}{\partial x} A_W\right)_W + \left(k_N \frac{\partial T}{\partial y} A_N\right)_N - \left(k_S \frac{\partial T}{\partial y} A_S\right)_S + F\Delta V = 0 \tag{3.5}$$

The partial derivatives were subsequently discretized using central finite differences, yielding:

$$k_E \frac{T_E - T_P}{\delta x}\delta y - k_W \frac{T_P - T_W}{\delta x}\delta y + k_N \frac{T_N - T_P}{\delta y}\delta x - k_S \frac{T_P - T_S}{\delta y}\delta x + F\delta x\delta y = 0 \tag{3.6}$$

Finally, the equation for the general case was obtained in its canonical form, applicable to all interior cells of the domain,

$$\begin{aligned}\left(k_E \frac{\delta y}{\delta x} + k_W \frac{\delta y}{\delta x} + k_N \frac{\delta x}{\delta y} + k_S \frac{\delta x}{\delta y}\right) T_P = \\ = k_E \frac{\delta y}{\delta x} T_E + k_W \frac{\delta y}{\delta x} T_W + k_N \frac{\delta x}{\delta y} T_N + k_S \frac{\delta x}{\delta y} T_S + F\delta x\delta y\end{aligned} \tag{3.7}$$

and the following coefficients were extracted for this same set of cells.

$$a_E = k_E \frac{\delta y}{\delta x}, \quad a_W = k_W \frac{\delta y}{\delta x}, \quad a_N = k_N \frac{\delta x}{\delta y}, \quad a_S = k_S \frac{\delta x}{\delta y}, \quad S_P = 0, \quad S_U = F\delta x\delta y \tag{3.8}$$

3.2 Boundary Conditions

Eq.(3.7) represents the general case and is applied to all interior cells of the domain. However, it was necessary to determine the corresponding equations for the boundary cells (Figure1 in Appendix E illustrates the domain considered and the different boundaries). To this end, different boundary conditions—Prescribed Flux (Neumann) or Prescribed Temperature

(Dirichlet)—were imposed, and the corresponding equations were derived.

3.2.1 Southern Boundary (B) - Dirichlet boundary condition

For the southern boundary B, the boundary condition is a prescribed temperature, also known as a Dirichlet condition.

The Finite Volume Method was applied, and T_S was no longer considered; instead, the exact temperature value at the boundary, T_{wall}, was used, such that:

$$T_{wall} = T_{analitica}(x, 0) \tag{3.9}$$

The distance considered for the discretization of the derivative was no longer δy (which corresponds to the distance between the centers of cells P and S), but instead $\frac{\delta y}{2}$, which represents the distance from the center of cell P to the southern boundary. This led to Eq.(3.10), which results from the imposition of this boundary condition, and to the coefficients in (3.11).

$$\begin{aligned}&\left(k_E\frac{\delta y}{\delta x} + k_W\frac{\delta y}{\delta x} + k_N\frac{\delta x}{\delta y} + k_S\frac{2\delta x}{\delta y}\right)T_P = \\ &= k_E\frac{\delta y}{\delta x}T_E + k_W\frac{\delta y}{\delta x}T_W + k_N\frac{\delta x}{\delta y}T_N + 2k_S T_{\text{anal}}(x,0)\frac{\delta x}{\delta y} + F\delta x\delta y\end{aligned} \tag{3.10}$$

Southern Boundary Parameters:

$$\begin{aligned}a_P &= k_E\frac{\delta y}{\delta x} + k_W\frac{\delta y}{\delta x} + k_N\frac{\delta x}{\delta y} + k_S\frac{2\delta x}{\delta y} \\ a_E &= k_E\frac{\delta y}{\delta x}, \quad a_W = k_W\frac{\delta y}{\delta x}, \quad a_N = k_N\frac{\delta x}{\delta y}, \quad a_S = 0 \\ S_U &= F\delta x\delta y + 2k_S T_{\text{anal}}(x,0)\frac{\delta x}{\delta y} \\ S_P &= -2k_S\frac{\delta x}{\delta y}\end{aligned} \tag{3.11}$$

3.2.2 Western Boundary (A) – Prescribed Temperature

The western boundary A also has a prescribed temperature boundary condition, and the corresponding equation for this boundary was obtained analogously to Section 3.2.1, where:

$$T_W = T_{analitica}(0, y) \tag{3.12}$$

and the distance considered between cell P and the western boundary is given by $\frac{\delta x}{2}$.

Finally, the following was obtained:

$$\left(k_E\frac{\delta y}{\delta x}+2k_W\frac{\delta y}{\delta x}+k_N\frac{\delta x}{\delta y}+k_S\frac{\delta x}{\delta y}\right)T_P = \\ = k_E\frac{\delta y}{\delta x}T_E+k_N\frac{\delta x}{\delta y}T_N+k_S\frac{\delta x}{\delta y}T_S+2k_W T_{\text{anal}}(0,y)\frac{\delta y}{\delta x}+F\delta x\delta y \tag{3.13}$$

Western Boundary Parameters:

$$a_P = k_E\frac{\delta y}{\delta x}+2k_W\frac{\delta y}{\delta x}+k_N\frac{\delta x}{\delta y}+k_S\frac{\delta x}{\delta y} \\ a_E = k_E\frac{\delta y}{\delta x}, \quad a_W = 0, \quad a_N = k_N\frac{\delta}{\delta y}, \quad a_S = k_S\frac{\delta x}{\delta y} \\ S_U = F\delta x\delta y + 2k_W\cdot T_{\text{analítica}}(0,y)\cdot\frac{\delta y}{\delta x} \\ S_P = -2k_W\cdot\frac{\delta y}{\delta x} \tag{3.14}$$

3.2.3 Eastern Boundary (C) – Neumann Condition

At the eastern boundary C, a prescribed flux condition was applied, also known as a Neumann boundary condition. This condition imposes the rate of change of T in the direction normal to the eastern boundary.

It was substituted into Eq.(3.5),

$$\left(\frac{\partial T}{\partial x}\right)_E = \frac{\partial T}{\partial x}(L,y) \tag{3.15}$$

where $\frac{\partial T}{\partial x}$ is given by Eq.(2.6), and the following expression was obtained:

$$k_E\frac{\partial T}{\partial x}(L,y)\delta y - k_W\frac{T_P-T_W}{\delta x}\delta y + k_N\frac{T_N-T_P}{\delta y}\delta x - k_S\frac{T_P-T_S}{\delta y}\delta x + F\delta x\delta y = 0 \tag{3.16}$$

The equation was then rearranged:

$$\left(k_W\frac{\delta y}{\delta x}+k_N\frac{\delta x}{\delta y}+k_S\frac{\delta x}{\delta y}\right)T_p = \\ = k_W\frac{\delta y}{\delta x}T_W+k_N\frac{\delta x}{\delta y}T_N+k_S\frac{\delta x}{\delta y}T_S+F\delta x\delta y+k_E\frac{\partial T}{\partial x}(L,y)\delta y \tag{3.17}$$

Eastern Boundary Parameters:

$$
\begin{aligned}
&a_P = k_W\frac{\delta y}{\delta x} + k_N\frac{\delta x}{\delta y} + k_S\frac{\delta x}{\delta y} \\
&a_E = 0, \quad a_W = k_W\frac{\delta y}{\delta x}, \quad a_N = k_N\frac{\delta x}{\delta y}, \quad a_S = k_S\frac{\delta x}{\delta y} \\
&S_U = F\delta x\delta y + k_E\frac{\partial T}{\partial x}(L,y)\delta y, \quad S_P = 0
\end{aligned} \tag{3.18}
$$

3.2.4 Northern Boundary (D) – Neumann Condition

As with the eastern boundary, the northern boundary D is subject to a prescribed flux condition, or Neumann boundary condition.

Analogously to Section 3.2.3, a substitution was made in Eq. 3.5,

$$
\left(\frac{\partial T}{\partial y}\right)_N = \frac{\partial T}{\partial y}(x,H) \tag{3.19}
$$

where $\frac{\partial T}{\partial y}$ is given by Eq. 2.7, resulting in:

$$
\begin{aligned}
&\left(k_E\frac{\delta y}{\delta x} + k_W\frac{\delta y}{\delta x} + k_S\frac{\delta x}{\delta y}\right)T_p = \\
&= k_E\frac{\delta y}{\delta x}T_E + k_W\frac{\delta y}{\delta x}T_W + k_S\frac{\delta x}{\delta y}T_S + F\delta x\delta y + k_N\frac{\partial T_{\text{analítica}}}{\partial y}(x,H)\delta x
\end{aligned} \tag{3.20}
$$

Northern Boundary Parameters:

$$
\begin{aligned}
&a_P = k_E\frac{\delta y}{\delta x} + k_W\frac{\delta y}{\delta x} + k_S\frac{\delta x}{\delta y} \\
&a_E = k_E\frac{\delta y}{\delta x}, \quad a_W = k_W\frac{\delta y}{\delta x}, \quad a_S = k_S\frac{\delta x}{\delta y}, \quad a_N = 0 \\
&S_U = F\delta x\delta y + k_N\frac{\partial T_{\text{analítica}}}{\partial y}(x,H)\delta x, \quad S_P = 0
\end{aligned} \tag{3.21}
$$

4 Numerical Resolution

4.1 Programs Developed in Python

Based on the discretized algebraic equations for all computational cells, obtained in Sections 3.1 and 3.2, three `Python` programs were developed to perform the numerical resolution. These programs can be found in Appendices B, C, and D.

All three codes solve the problem for any number of cells in the x and y directions and generate four plots:

1. A plot showing the values obtained for each cell by solving the linear system $AT = b$, where A is the coefficient matrix, T is the temperature vector, and b is the source term vector;
2. A plot showing the values computed for each cell using the analytical solution, Eq. 2.3;
3. An **Absolute Error** plot, where the absolute error between the numerical and analytical solutions is computed for each cell;
4. A **Relative Error** plot, where the relative error between the numerical and analytical solutions is computed for each cell.

In addition, all three programs report the execution time, the memory usage of each simulation, and the average absolute and relative errors. These additional features enabled a more objective comparison of the numerical results.

4.2 Solvers

The three programs share the same general structure, objectives, and capabilities described in Section 4.1; however, they differ in the type of solver used to solve the linear system of equations $AT = b$.

Program Mk.1, available in Appendix B, uses a direct solver; Program Mk.2, available in Appendix C, employs an iterative solver, specifically the Gauss-Seidel method; finally, Program Mk.3 uses an iterative Gauss-Seidel solver with *Successive Over Relaxation* (SOR).

Table 4.1 presents the main characteristics of each of these algorithms. However, a detailed explanation of their operation goes beyond the scope of this work. Further information on the methods used can be found in [2] and [3].

Table 4.1: Comparison Between Numerical Methods

Method	Advantages	Disadvantages	Equation
Direct	High accuracy; simple implementation using `np.linalg.solve`.	High memory usage; low scalability; inefficient for large matrices.	–
Gauss-Seidel	Simple; low memory usage; effective for sparse matrices.	May be slow; convergence not guaranteed.	$T_i = -\frac{1}{a_{ii}} \sum_{j \neq i} a_{ij} T_j + \frac{b_i}{a_{ii}}$
SOR	Faster than Gauss-Seidel; lightweight iterations; stable with a good ω.	Choice of ω is non-trivial; convergence not guaranteed.	$T_i^{(k+1)} = (1 - \omega) T_i^{(k)} + \omega \left(-\frac{1}{a_{ii}} \sum_{j \neq i} a_{ij} T_j + \frac{b_i}{a_{ii}} \right)$

5 Results Analysis

Three custom-developed codes were used to demonstrate the convergence of the numerical solution toward the analytical solution as the mesh is refined. Execution time, memory consumption, and both absolute and average relative errors were also measured for each case.

Considering that $L = 2H$ (Anexo E), all implemented meshes have twice as many cells in the x-direction as in the y-direction. The program was executed for five progressively refined meshes with dimensions $20 \times 10, \quad 40 \times 20, \quad 80 \times 40, \quad 160 \times 80, \quad 320 \times 160$, where each refinement results in a fourfold increase in the total number of cells.

The result plots for each mesh are available in Anexo F, and Table 5.1 presents the comparison of the numerical methods.

Table 5.1: Comparison between numerical methods for different meshes

Mesh	Method	Time	Memory (MB)	Mean Absolute Error (K)	Mean Relative Error
20 × 10	Direct	0.21 s	101.5	0.177	0.001
	Gauss-Seidel	1.68 s	101.3	0.177	0.001
	SOR	0.49 s	101.5	0.177	0.001
40 × 20	Direct	0.17 s	160.77	0.022	0.0001
	Gauss-Seidel	15.19 s	153.05	0.022	0.0001
	SOR	2.93 s	153.05	0.022	0.0001
80 × 40	Direct	1.42 s	267.22	0.0027	1.5e-5
	Gauss-Seidel	137.8 s	181.31	0.0027	1.5e-5
	SOR	31.01 s	181.21	0.0027	1.5e-5
160 × 80	Direct	58.67 s	1536.69	0.0003	1.9e-6
	Gauss-Seidel	59 min	292.05	0.0003	2e-6
	SOR	12 min	255.22	0.0004	2e-6
320 × 160	Direct	N/A	N/A	N/A	N/A
	Gauss-Seidel	N/A	N/A	N/A	N/A
	SOR	2h 21 min	710.11	5.9e-5	3e-7

6 Final Remarks

Table 5.1 shows that, for the three methods tested (Direct, Gauss-Seidel, and SOR), the numerical solution converges to the analytical solution as the mesh is refined. This conclusion is supported by the progressive reduction in both absolute and average relative errors with the increasing number of cells.

Regarding execution time, it is observed that execution time consistently increases for all methods as the mesh is refined. The Direct method is the fastest (possibly because it

relies on a more optimized function that uses precompiled Python libraries), followed by the iterative methods: Gauss-Seidel with relaxation (SOR), and Gauss-Seidel without relaxation.

Memory usage increases for all three methods as the mesh is refined. The Direct method consumes the most memory; notably, for a 320×160 mesh, this method fails to run due to insufficient memory. The iterative methods consume less memory than the Direct method, and this difference becomes increasingly significant as the number of cells grows. This highlights that iterative methods become especially advantageous for highly refined meshes with sparse coefficient matrices A.

The 80×40 mesh proves to be a highly efficient solution, with extremely short simulation times and already low error levels on the order of 10^{-5}. It offers a good balance between speed and accuracy.

The 160×80 mesh presents a reasonable compromise between accuracy and computational efficiency, achieving errors on the order of 10^{-6} while maintaining manageable time and memory requirements. Further refinements yield only marginal improvements in error but incur significantly higher computational costs.

References

[1] Miguel Nóbrega. Projeto 1 – métodos computacionais avançados para aplicações aeroespaciais, 2024. Slides da disciplina MCAAA – 2º semestre, Universidade do Minho.

[2] Arnold Reusken. On the convergence of basic iterative methods for convection-diffusion problems. *Numerische Mathematik*, 53(6):641–654, 1988.

[3] Gilbert Strang. Lecture 18: Iterative methods: Jacobi, gauss-seidel, sor. `https://dspace.mit.edu/bitstream/handle/1721.1/75282/18-335j-fall-2006/contents/lecture-notes/lec18.pdf`, 2006. Lecture notes, MIT OpenCourseWare, Course 18.335J: Introduction to Numerical Methods.

A MATLAB Code

```
    syms x y L H k_ref k_x k_y

% Analytical Solution for T

T = 50*cos((4*x/L)*pi + (2*y/H)*pi) + 200;
latex_T = latex(T);

% K(x,y)

k = k_ref * (k_x*cos(pi*2*x/L) + k_y*sin(pi*y/H));
latex_k = latex(k)

% Gradient of T

grad_T = gradient(T, [x,y]);
latex_gradT = latex(grad_T);

% Thermal Flux

fluxo = k * grad_T;

% Divergence of Thermal Flux

Fonte = divergence(fluxo, [x,y]);
latex_Fonte = latex(Fonte);
disp(latex_Fonte);

% Partial Derivatives

dT_dx = diff(T, x);
dT_dy = diff(T, y);

```

```
latex_dT_dx = latex(dT_dx);
latex_dT_dy = latex(dT_dy);

disp(dT_dx);
disp(dT_dy);

disp(latex_dT_dx);
disp(latex_dT_dy);
```

Listing 1: MATLAB Code

B Python Mk.I – Direct Solver

```
# ## Import Libraries

# %%
import numpy as np
import math

# %% [markdown]
# ## Problem Data

# %%

# Computational Domain
L = 2
H = 1

ncx = 160
ncy = 80
ncell = ncx * ncy

dx = L/ncx
dy = H/ncy

# Cell Center
def cellCenter(i):
  xcell=dx/2+(i%ncx)*dx
  ycell=dy/2+int(i/ncx)*dy
  return xcell,ycell

# k(x,y)

k_ref = 0.15
k_x = 1
```

```
k_y = 0

def k_xy(k_ref,k_x,k_y,x,y):
  return k_ref * (k_x * np.cos(np.pi*(2*x/L)) + k_y * np.sin(np.
     pi * (y/H)))

# Source Term

def Fonte(k_ref,k_x,k_y,x,y):
  return (400*k_ref*k_x*np.pi**2*np.sin((2*np.pi*x)/L)*np.sin((2*
     np.pi*y)/H + (4*np.pi*x)/L))/L**2 - (800*k_ref*np.pi**2*np.
     cos((2*np.pi*y)/H + (4*np.pi*x)/L)*(k_x*np.cos((2*np.pi*x)/L
     ) + k_y*np.sin((np.pi*y)/H)))/L**2 - (100*k_ref*k_y*np.pi
     **2*np.cos((np.pi*y)/H)*np.sin((2*np.pi*y)/H + (4*np.pi*x)/L
     ))/H**2 - (200*k_ref*np.pi**2*np.cos((2*np.pi*y)/H + (4*np.
     pi*x)/L)*(k_x*np.cos((2*np.pi*x)/L) + k_y*np.sin((np.pi*y)/H
     )))/H**2

# Partial Derivatives, dT_dx e dT_dy

def dT_dx(x,y,L,H):
  return -(200*np.pi*np.sin((2*np.pi*y)/H + (4*np.pi*x)/L))/L
def dT_dy(x,y,L,H):
  return -(100*np.pi*np.sin((2*np.pi*y)/H + (4*np.pi*x)/L))/H




# %% [markdown]
# ## Analytical Solution

# %%
```

```
58
59
60 # Function to compute the Analytical Solution, T(x,y)
61
62 def T_analitica(x,y):
63   return 50 * np.cos((np.pi*4*x/L) + (np.pi*2*y/H)) + 200
64
65
66 # %% [markdown]
67 # ## Numerical Resolution and Errors
68
69 # %%
70 import numpy as np
71 from numpy import linalg as LA
72
73 #initialize arrays
74 A = np.zeros((ncell,ncell))
75 PHI = np.zeros(ncell)
76 AW = np.zeros(ncell)
77 AE = np.zeros(ncell)
78 AN = np.zeros(ncell)
79 AS = np.zeros(ncell)
80 SU = np.zeros(ncell)
81 SP = np.zeros(ncell)
82 Error_abs = np.zeros(ncell)
83 Error_rel = np.zeros(ncell)
84
85 #Set method coefficients
86 for i in range(0, ncell):
87
88     xcell,ycell=cellCenter(i)
89     Termo_fonte = - Fonte(k_ref,k_x,k_y,xcell,ycell)
90     FI_dT_dx = dT_dx(xcell+dx/2,ycell,L,H)
```

```
91      FI_dT_dy = dT_dy(xcell,ycell+dy/2,L,H)
92      TI_0Y = T_analitica(xcell -dx/2,ycell)
93      TI_X0 = T_analitica(xcell,ycell-dy/2)
94
95      k_w = k_xy(k_ref,k_x,k_y,xcell-dx/2,ycell)
96      k_e = k_xy(k_ref,k_x,k_y,xcell+dx/2,ycell)
97      k_n = k_xy(k_ref,k_x,k_y,xcell,ycell+dy/2)
98      k_s = k_xy(k_ref,k_x,k_y,xcell,ycell-dy/2)
99
100     AW[i]=k_w * dy/dx
101     AE[i]=k_e * dy/dx
102     AS[i]=k_s * dx/dy
103     AN[i]=k_n * dx/dy
104     SU[i]=Termo_fonte * dx * dy
105     SP[i]=0
106
107     if i<ncx: ## First Row, South Boundary, Imposed Temperature,
            Dirichlet Condition
108         AS[i] = 0
109         SU[i] += 2 * k_s * TI_X0 * dx/dy
110         SP[i] += -2 * k_s * dx/dy
111
112     if i % ncx == 0 : ## First Column, West Boundary, Imposed
            Temperature, Dirichlet Condition
113         AW[i] = 0
114         SU[i] += 2 * k_w * TI_0Y * dy/dx
115         SP[i] += -2 * k_w * dy/dx
116
117     if (i+1) % ncx == 0 : ## Last column, East Boundary, Imposed
            Flux, Neumann Condition
118         AE[i] = 0
119         SU[i] += k_e * FI_dT_dx * dy
120         SP[i] += 0
```

```
121
122     if i >= (ncell-ncx): ## Last Row, North Boundary, Imposed
            Flux, Neumann Condition
123         AN[i] = 0
124         SU[i] += k_n * FI_dT_dy * dx
125         SP[i] += 0
126
127 # Prepare the coefficients matrix [A]
128 for i in range(0, ncell):
129     icw=i-1
130     ice=i+1
131     icn=i+ncx
132     ics=i-ncx
133     if i>=ncx: ##not in first Row
134       A[i,ics]=-AS[i]
135     if i % ncx != 0 : ##not in first column
136       A[i,icw]=-AW[i]
137     if (i+1) % ncx != 0 : ##Not in Last column
138       A[i,ice]=-AE[i]
139     if i < (ncell-ncx): ##Not in Last Row
140       A[i,icn]=-AN[i]
141
142     A[i,i]=AW[i]+AE[i]+AN[i]+AS[i]-SP[i] #The Diagonal (
            Coefficient ap)
143
144
145 # Direct method for solving linear systems using LU decomposition
    .
146 # Computes the exact solution (up to numerical precision) in a
    finite number of steps, unlike iterative methods (Gauss-Seidel
     and SOR) that approximate the solution progressively.
147
148 PHI = np.linalg.solve(A, SU)
```

```
149
150 # Calculate the errors
151 for i in range(0, ncell):
152     xcell,ycell = cellCenter(i)
153     Error_abs[i] = abs((PHI[i]-T_analitica(xcell,ycell)))
154     Error_rel[i] = abs((PHI[i]-T_analitica(xcell,ycell))/
          T_analitica(xcell,ycell))
155
156 print("Average Absolute Error:",LA.norm(Error_abs)/ncell)
157 print("Average Relative Error:",LA.norm(Error_rel)/ncell)
158
159 # %% [markdown]
160 # ## Compute the Analytical Solution for All Cells
161
162 # %%
163 #Create new variable to store the Analytical solution
164 PHI_ana = np.zeros(ncell)
165
166 # Calculate Analytical solution values
167 for i in range(0, ncell):
168     xcell,ycell = cellCenter(i)
169     PHI_ana[i]=T_analitica(xcell,ycell)
170
171
172
173 # %% [markdown]
174 # ## Plot the Graph
175
176 # %%
177 def vec2Grid(var, nx, ny):
178     z = np.zeros((ny, nx))  # (Rows, Columns)
179     for i in range(len(var)):
180         z[int(i / nx), i % nx] = var[i]
```

```
181     return z
182
183 # %%
184 import matplotlib
185 import matplotlib.pyplot as plt
186 from scipy.interpolate import griddata
187
188 y, x = np.mgrid[slice(0, H + dy, dy),
189                 slice(0, L + dx, dx)]
190
191 #print(x
192 #print(y)
193 figScl=10
194 fig, axs = plt.subplots(2, 2,figsize=(figScl*L/H,figScl*H/L))
195
196 ax = axs[0, 0]
197 z = vec2Grid(PHI,ncx,ncy)
198 #print(z)
199 Var_min, Var_max = np.abs(PHI).min(), np.abs(PHI).max()
200 c = ax.pcolor(x, y, z, cmap='jet', vmin=Var_min, vmax=Var_max)
201 ax.set_title('Solucao Numerica')
202 fig.colorbar(c, ax=ax)
203
204 ax = axs[0, 1]
205 z = vec2Grid(PHI_ana,ncx,ncy)
206 #print(z)
207 Var_min, Var_max = np.abs(PHI_ana).min(), np.abs(PHI_ana).max()
208 c = ax.pcolor(x, y, z, cmap='jet', vmin=Var_min, vmax=Var_max)
209 ax.set_title('Solucao Analitica')
210 fig.colorbar(c, ax=ax)
211
212 ax = axs[1, 0]
213 z = vec2Grid(Error_abs,ncx,ncy)
```

```
214 Var_min, Var_max = np.abs(Error_abs).min(), np.abs(Error_abs).max
    ()
215 c = ax.pcolor(x, y, z, cmap='jet', vmin=Var_min, vmax=Var_max)
216 ax.set_title('Erro Absoluto')
217 fig.colorbar(c, ax=ax)
218
219 ax = axs[1, 1]
220 z = vec2Grid(Error_rel,ncx,ncy)
221 Var_min, Var_max = np.abs(Error_rel).min(), np.abs(Error_rel).max
    ()
222 c = ax.pcolor(x, y, z, cmap='jet', vmin=Var_min, vmax=Var_max)
223 ax.set_title('Erro Relativo')
224 fig.colorbar(c, ax=ax)
225
226 fig.tight_layout()
227
228 plt.show()
```

Listing 2: Python Mk.1, with direct solver

C Python Mk.II – Gauss-Seidel Iterative Solver

```
# # **Two-Dimensional Pure Diffusion - MARK II - Gauss-Seidel
   Iterative Method**
#
# ---
#
#

# %%


# %% [markdown]
# ## Import Libraries

# %%
import numpy as np
import math
import time
!pip install memory_profiler
!pip install scipy
from memory_profiler import memory_usage

# %% [markdown]
# ## Problem Data

# %%

# Computational Domain
L = 2
H = 1

ncx = 80
ncy = 40
```

```
32 ncell = ncx * ncy
33
34 dx = L/ncx
35 dy = H/ncy
36
37 # Cell Center
38 def cellCenter(i):
39   xcell=dx/2+(i%ncx)*dx
40   ycell=dy/2+int(i/ncx)*dy
41   return xcell,ycell
42
43 # k(x,y) Equation
44
45 k_ref = 0.15
46 k_x = 1
47 k_y = 0
48
49 def k_xy(k_ref,k_x,k_y,x,y):
50   return k_ref * (k_x * np.cos(np.pi*(2*x/L)) + k_y * np.sin(np.
       pi * (y/H)))
51
52 # Source Term
53
54 def Fonte(k_ref,k_x,k_y,x,y):
55   return (400*k_ref*k_x*np.pi**2*np.sin((2*np.pi*x)/L)*np.sin((2*
       np.pi*y)/H + (4*np.pi*x)/L))/L**2 - (800*k_ref*np.pi**2*np.
       cos((2*np.pi*y)/H + (4*np.pi*x)/L)*(k_x*np.cos((2*np.pi*x)/L
       ) + k_y*np.sin((np.pi*y)/H)))/L**2 - (100*k_ref*k_y*np.pi
       **2*np.cos((np.pi*y)/H)*np.sin((2*np.pi*y)/H + (4*np.pi*x)/L
       ))/H**2 - (200*k_ref*np.pi**2*np.cos((2*np.pi*y)/H + (4*np.
       pi*x)/L)*(k_x*np.cos((2*np.pi*x)/L) + k_y*np.sin((np.pi*y)/H
       )))/H**2
56
```

```
# Partial Derivatives, dT_dx e dT_dy

def dT_dx(x,y,L,H):
  return -(200*np.pi*np.sin((2*np.pi*y)/H + (4*np.pi*x)/L))/L
def dT_dy(x,y,L,H):
  return -(100*np.pi*np.sin((2*np.pi*y)/H + (4*np.pi*x)/L))/H




# %% [markdown]
# ## Analytical Solution

# %%


# Function to compute the analytical solution T(x,y)

def T_analitica(x,y):
  return 50 * np.cos((np.pi*4*x/L) + (np.pi*2*y/H)) + 200


# %% [markdown]
# ## Numerical Simulation and Erros

# %%
import numpy as np
from numpy import linalg as LA

#initialize arrays
A = np.zeros((ncell,ncell))
PHI = np.zeros(ncell)
```

```
AW = np.zeros(ncell)
AE = np.zeros(ncell)
AN = np.zeros(ncell)
AS = np.zeros(ncell)
SU = np.zeros(ncell)
SP = np.zeros(ncell)
Error_abs = np.zeros(ncell)
Error_rel = np.zeros(ncell)

#Set method coefficients
for i in range(0, ncell):

    xcell,ycell=cellCenter(i)
    Termo_fonte = - Fonte(k_ref,k_x,k_y,xcell,ycell)
    FI_dT_dx = dT_dx(xcell+dx/2,ycell,L,H)
    FI_dT_dy = dT_dy(xcell,ycell+dy/2,L,H)
    TI_0Y = T_analitica(xcell -dx/2,ycell)
    TI_X0 = T_analitica(xcell,ycell-dy/2)

    k_w = k_xy(k_ref,k_x,k_y,xcell-dx/2,ycell)
    k_e = k_xy(k_ref,k_x,k_y,xcell+dx/2,ycell)
    k_n = k_xy(k_ref,k_x,k_y,xcell,ycell+dy/2)
    k_s = k_xy(k_ref,k_x,k_y,xcell,ycell-dy/2)

    AW[i]=k_w * dy/dx
    AE[i]=k_e * dy/dx
    AS[i]=k_s * dx/dy
    AN[i]=k_n * dx/dy
    SU[i]=Termo_fonte * dx * dy
    SP[i]=0

    if i<ncx: ## First Row, South Boundary, Imposed Temperature,
       Dirichlet Condition
```

```
122             AS[i] = 0
123             SU[i] += 2 * k_s * TI_X0 * dx/dy
124             SP[i] += -2 * k_s * dx/dy
125 
126         if i % ncx == 0 : ## First Column, West Boundary, Imposed
                Temperature, Dirichlet Condition
127             AW[i] = 0
128             SU[i] += 2 * k_w * TI_0Y * dy/dx
129             SP[i] += -2 * k_w * dy/dx
130 
131         if (i+1) % ncx == 0 : ##Last column, East Boundary, Imposed
                Flux, Neumann Condition
132             AE[i] = 0
133             SU[i] += k_e * FI_dT_dx * dy
134             SP[i] += 0
135 
136         if i >= (ncell-ncx): ##Last Row, North Boundary, Imposed Flux
                , Neumann Condition
137             AN[i] = 0
138             SU[i] += k_n * FI_dT_dy * dx
139             SP[i] += 0
140 
141 # Prepare the coefficients matrix [A]
142 for i in range(0, ncell):
143     icw=i-1
144     ice=i+1
145     icn=i+ncx
146     ics=i-ncx
147     if i>=ncx: ##Not in first Row
148       A[i,ics]=-AS[i]
149     if i % ncx != 0 : ##Not in first column
150       A[i,icw]=-AW[i]
151     if (i+1) % ncx != 0 : ##Not in Last column
```

```
152         A[i,ice]=-AE[i]
153       if i < (ncell-ncx): ##Not in Last Row
154         A[i,icn]=-AN[i]
155
156       # A[i,i]=AW[i]+AE[i]+AN[i]+AS[i]-SP[i] #The Diagonal (
            Coefficient ap)
157
158 #Implementation of the Gauss-Seidel Iterative Method for
    Numerical Resolution
159
160 def gauss_seidel(PHI, AW, AE, AN, AS, SP, SU, ncx, ncy, max_iter
    =60000, tol=1e-8):
161     ncell = ncx * ncy
162     for it in range(max_iter):
163         PHI_old = PHI.copy()
164         for i in range(ncell):
165             ap = AW[i] + AE[i] + AN[i] + AS[i] - SP[i]
166             phi_w = PHI[i - 1] if i % ncx != 0 else 0
167             phi_e = PHI[i + 1] if (i + 1) % ncx != 0 else 0
168             phi_s = PHI[i - ncx] if i >= ncx else 0
169             phi_n = PHI[i + ncx] if i < ncell - ncx else 0
170
171             PHI[i] = (AW[i] * phi_w + AE[i] * phi_e + AS[i] *
                phi_s + AN[i] * phi_n + SU[i]) / ap
172
173         # Convergence criterion (Mean Relative Error)
174         error = np.linalg.norm(PHI - PHI_old) / np.linalg.norm(
            PHI)
175         if error < tol:
176             print(f"Gauss-Seidel convergiu em {it+1} itera  es
                com erro = {error:.2e}")
177             break
178     else:
```

```
179         print("Gauss-Seidel n o convergiu no n mero m ximo de
                itera   es.")
180     return PHI
181
182 # Numerical Resolution With Gauss-Seidel Method and Time
       Measurement
183 def run_gauss_seidel():
184     PHI = np.zeros(ncell)
185     PHI = gauss_seidel(PHI, AW, AE, AN, AS, SP, SU, ncx, ncy)
186     return PHI
187
188 start_time = time.time()
189 mem_usage, PHI = memory_usage(run_gauss_seidel, retval=True,
       max_usage=True)
190 end_time = time.time()
191
192 print(f"Tempo de execu   o: {end_time - start_time:.4f} segundos
       ")
193 print(f"Mem ria m xima usada: {mem_usage:.2f} MB")
194
195 # Calculate the errors
196 for i in range(0, ncell):
197     xcell,ycell = cellCenter(i)
198     Error_abs[i] = abs((PHI[i]-T_analitica(xcell,ycell)))
199     Error_rel[i] = abs((PHI[i]-T_analitica(xcell,ycell))/
          T_analitica(xcell,ycell))
200
201 print("Average Absolute Error:",LA.norm(Error_abs)/ncell)
202 print("Average Relative Error:",LA.norm(Error_rel)/ncell)
203
204 # %% [markdown]
205 # ## Calculation of the Analytical Solution for all cells
206
```

```
# %%
#Create new variable to store the Analytical solution
PHI_ana = np.zeros(ncell)

# Calculate Analytical solution values
for i in range(0, ncell):
    xcell,ycell = cellCenter(i)
    PHI_ana[i]=T_analitica(xcell,ycell)



# %% [markdown]
# ## Plot the Graph

# %%
def vec2Grid(var, nx, ny):
    z = np.zeros((ny, nx))  # (linhas, colunas)
    for i in range(len(var)):
        z[int(i / nx), i % nx] = var[i]
    return z

# %%
import matplotlib
import matplotlib.pyplot as plt
from scipy.interpolate import griddata

y, x = np.mgrid[slice(0, H + dy, dy),
                slice(0, L + dx, dx)]

#print(x
#print(y)
figScl=10
fig, axs = plt.subplots(2, 2,figsize=(figScl*L/H,figScl*H/L))
```

```
240
241 ax = axs[0, 0]
242 z = vec2Grid(PHI,ncx,ncy)
243 #print(z)
244 Var_min, Var_max = np.abs(PHI).min(), np.abs(PHI).max()
245 c = ax.pcolor(x, y, z, cmap='jet', vmin=Var_min, vmax=Var_max)
246 ax.set_title('Numerical Solution')
247 fig.colorbar(c, ax=ax)
248
249 ax = axs[0, 1]
250 z = vec2Grid(PHI_ana,ncx,ncy)
251 #print(z)
252 Var_min, Var_max = np.abs(PHI_ana).min(), np.abs(PHI_ana).max()
253 c = ax.pcolor(x, y, z, cmap='jet', vmin=Var_min, vmax=Var_max)
254 ax.set_title('Analytical Solution')
255 fig.colorbar(c, ax=ax)
256
257 ax = axs[1, 0]
258 z = vec2Grid(Error_abs,ncx,ncy)
259 Var_min, Var_max = np.abs(Error_abs).min(), np.abs(Error_abs).max
    ()
260 c = ax.pcolor(x, y, z, cmap='jet', vmin=Var_min, vmax=Var_max)
261 ax.set_title('Absolute Error')
262 fig.colorbar(c, ax=ax)
263
264 ax = axs[1, 1]
265 z = vec2Grid(Error_rel,ncx,ncy)
266 Var_min, Var_max = np.abs(Error_rel).min(), np.abs(Error_rel).max
    ()
267 c = ax.pcolor(x, y, z, cmap='jet', vmin=Var_min, vmax=Var_max)
268 ax.set_title('Relative Error')
269 fig.colorbar(c, ax=ax)
270
```

```
271 fig.tight_layout()
272
273 plt.show()
```

Listing 3: Python Mk.2 code, with Gauss-Seidel Iterative Solver

D Python Mk.III – Gauss-Seidel SOR Iterative Solver

```
# # **Two-Dimensional Pure Diffusion - MARK III - Gauss-Seidel With Successive Over-Relaxation (SOR)**
#
# ---
#
#

# %% [markdown]
# ## Import Libraries

# %%
import numpy as np
import math
import time
!pip install memory_profiler
from memory_profiler import memory_usage

# %% [markdown]
# ## Problem Data

# %%

# Computational Domain
L = 2
H = 1

ncx = 20
ncy = 10
ncell = ncx * ncy

omega_otimo = 2 / (1 + np.sin(np.pi / ncx))
print(f"Omega  timo      {omega_otimo:.4f}")
```

```
32
33 dx = L/ncx
34 dy = H/ncy
35
36 # Cell Center
37 def cellCenter(i):
38   xcell=dx/2+(i%ncx)*dx
39   ycell=dy/2+int(i/ncx)*dy
40   return xcell,ycell
41
42 # k(x,y) Equation
43
44 k_ref = 0.15
45 k_x = 1
46 k_y = 0
47
48 def k_xy(k_ref,k_x,k_y,x,y):
49   return k_ref * (k_x * np.cos(np.pi*(2*x/L)) + k_y * np.sin(np.
       pi * (y/H)))
50
51 # Source Term
52
53 def Fonte(k_ref,k_x,k_y,x,y):
54   return (400*k_ref*k_x*np.pi**2*np.sin((2*np.pi*x)/L)*np.sin((2*
       np.pi*y)/H + (4*np.pi*x)/L))/L**2 - (800*k_ref*np.pi**2*np.
       cos((2*np.pi*y)/H + (4*np.pi*x)/L)*(k_x*np.cos((2*np.pi*x)/L
       ) + k_y*np.sin((np.pi*y)/H)))/L**2 - (100*k_ref*k_y*np.pi
       **2*np.cos((np.pi*y)/H)*np.sin((2*np.pi*y)/H + (4*np.pi*x)/L
       ))/H**2 - (200*k_ref*np.pi**2*np.cos((2*np.pi*y)/H + (4*np.
       pi*x)/L)*(k_x*np.cos((2*np.pi*x)/L) + k_y*np.sin((np.pi*y)/H
       )))/H**2
55
56 # Partial Derivatives dT_dx e dT_dy
```

```
57
58 def dT_dx(x,y,L,H):
59   return -(200*np.pi*np.sin((2*np.pi*y)/H + (4*np.pi*x)/L))/L
60 def dT_dy(x,y,L,H):
61   return -(100*np.pi*np.sin((2*np.pi*y)/H + (4*np.pi*x)/L))/H
62
63
64
65
66
67 # %% [markdown]
68 # ## Analytical Solution
69
70 # %%
71
72
73 # Analytical Solution Function, T(x,y)
74
75 def T_analitica(x,y):
76   return 50 * np.cos((np.pi*4*x/L) + (np.pi*2*y/H)) + 200
77
78
79 # %% [markdown]
80 # ## Numerical Resolution and Errors
81
82 # %%
83 import numpy as np
84 from numpy import linalg as LA
85
86 #initialize arrays
87 A = np.zeros((ncell,ncell))
88 PHI = np.zeros(ncell)
89 AW = np.zeros(ncell)
```

```
AE = np.zeros(ncell)
AN = np.zeros(ncell)
AS = np.zeros(ncell)
SU = np.zeros(ncell)
SP = np.zeros(ncell)
Error_abs = np.zeros(ncell)
Error_rel = np.zeros(ncell)

#Set method coefficients
for i in range(0, ncell):

    xcell,ycell=cellCenter(i)
    Termo_fonte = - Fonte(k_ref,k_x,k_y,xcell,ycell)
    FI_dT_dx = dT_dx(xcell+dx/2,ycell,L,H)
    FI_dT_dy = dT_dy(xcell,ycell+dy/2,L,H)
    TI_0Y = T_analitica(xcell -dx/2,ycell)
    TI_X0 = T_analitica(xcell,ycell-dy/2)

    k_w = k_xy(k_ref,k_x,k_y,xcell-dx/2,ycell)
    k_e = k_xy(k_ref,k_x,k_y,xcell+dx/2,ycell)
    k_n = k_xy(k_ref,k_x,k_y,xcell,ycell+dy/2)
    k_s = k_xy(k_ref,k_x,k_y,xcell,ycell-dy/2)

    AW[i]=k_w * dy/dx
    AE[i]=k_e * dy/dx
    AS[i]=k_s * dx/dy
    AN[i]=k_n * dx/dy
    SU[i]=Termo_fonte * dx * dy
    SP[i]=0

    if i<ncx: ## First Row, South Boundary, Imposed Temperature,
       Dirichlet Condition
        AS[i] = 0
```

```
122             SU[i] += 2 * k_s * TI_X0 * dx/dy
123             SP[i] += -2 * k_s * dx/dy
124 
125         if i % ncx == 0 : ## First Column, West Boundary, Imposed
            Temperature, Dirichlet Condition
126             AW[i] = 0
127             SU[i] += 2 * k_w * TI_0Y * dy/dx
128             SP[i] += -2 * k_w * dy/dx
129 
130         if (i+1) % ncx == 0 : ## Last column, East Boundary, Imposed
            Flux, Neumann Condition
131             AE[i] = 0
132             SU[i] += k_e * FI_dT_dx * dy
133             SP[i] += 0
134 
135         if i >= (ncell-ncx): ## Last Row, North Boundary, Imposed
            Flux, Neumann Condition
136             AN[i] = 0
137             SU[i] += k_n * FI_dT_dy * dx
138             SP[i] += 0
139 
140 # Implementation of the Gauss-Seidel SOR Method
141 
142 def gauss_seidel_sor(PHI, AW, AE, AN, AS, SP, SU, ncx, ncy, omega
    =1.96, max_iter=60000, tol=1e-8):
143     ncell = ncx * ncy
144     for it in range(max_iter):
145         PHI_old = PHI.copy()
146         for i in range(ncell):
147             ap = AW[i] + AE[i] + AN[i] + AS[i] - SP[i]
148             phi_w = PHI[i - 1] if i % ncx != 0 else 0
149             phi_e = PHI[i + 1] if (i + 1) % ncx != 0 else 0
150             phi_s = PHI[i - ncx] if i >= ncx else 0
```

```
151             phi_n = PHI[i + ncx] if i < ncell - ncx else 0
152
153             phi_gs = (AW[i] * phi_w + AE[i] * phi_e + AS[i] *
                    phi_s + AN[i] * phi_n + SU[i]) / ap
154             PHI[i] = (1 - omega) * PHI[i] + omega * phi_gs
155
156         # Convergence Critterion
157         error = np.linalg.norm(PHI - PHI_old) / np.linalg.norm(
                PHI)
158         if error < tol:
159             print(f"SOR convergiu em {it+1} itera  es com erro
                    = {error:.2e}")
160             break
161     else:
162         print("SOR n o convergiu no n mero m ximo de
                itera  es.")
163     return PHI
164
165 #Numerical Resolution with SOR and Time Count
166 def run_sor():
167     PHI = np.zeros(ncell)
168     PHI = gauss_seidel_sor(PHI, AW, AE, AN, AS, SP, SU, ncx, ncy,
            omega=1.7, tol=1e-8)
169     return PHI
170
171 start_time = time.time()
172 mem_usage, PHI = memory_usage(run_sor, retval=True, max_usage=
    True)
173 end_time = time.time()
174
175 print(f"Tempo de execu  o: {end_time - start_time:.4f} segundos
    ")
176 print(f"Mem ria  m xima usada: {mem_usage:.2f} MB")
```

```
177
178 # Calculate the errors
179 for i in range(0, ncell):
180     xcell,ycell = cellCenter(i)
181     Error_abs[i] = abs((PHI[i]-T_analitica(xcell,ycell)))
182     Error_rel[i] = abs((PHI[i]-T_analitica(xcell,ycell))/
          T_analitica(xcell,ycell))
183
184 print("Average Absolute Error:",LA.norm(Error_abs)/ncell)
185 print("Average Relative Error:",LA.norm(Error_rel)/ncell)
186
187 # %% [markdown]
188 # ## Analytical Solution for all Cells
189
190 # %%
191 #Create new variable to store the Analytical solution
192 PHI_ana = np.zeros(ncell)
193
194 # Calculate Analytical solution values
195 for i in range(0, ncell):
196     xcell,ycell = cellCenter(i)
197     PHI_ana[i]=T_analitica(xcell,ycell)
198
199
200
201 # %% [markdown]
202 # ## Plot the Graph
203
204 # %%
205 def vec2Grid(var, nx, ny):
206     z = np.zeros((ny, nx))  # (rows, columns)
207     for i in range(len(var)):
208         z[int(i / nx), i % nx] = var[i]
```

```
209     return z
210
211 # %%
212 import matplotlib
213 import matplotlib.pyplot as plt
214 from scipy.interpolate import griddata
215
216 y, x = np.mgrid[slice(0, H + dy, dy),
217                 slice(0, L + dx, dx)]
218
219 #print(x
220 #print(y)
221 figScl=10
222 fig, axs = plt.subplots(2, 2,figsize=(figScl*L/H,figScl*H/L))
223
224 ax = axs[0, 0]
225 z = vec2Grid(PHI,ncx,ncy)
226 #print(z)
227 Var_min, Var_max = np.abs(PHI).min(), np.abs(PHI).max()
228 c = ax.pcolor(x, y, z, cmap='jet', vmin=Var_min, vmax=Var_max)
229 ax.set_title('Solucao Numerica')
230 fig.colorbar(c, ax=ax)
231
232 ax = axs[0, 1]
233 z = vec2Grid(PHI_ana,ncx,ncy)
234 #print(z)
235 Var_min, Var_max = np.abs(PHI_ana).min(), np.abs(PHI_ana).max()
236 c = ax.pcolor(x, y, z, cmap='jet', vmin=Var_min, vmax=Var_max)
237 ax.set_title('Solucao Analitica')
238 fig.colorbar(c, ax=ax)
239
240 ax = axs[1, 0]
241 z = vec2Grid(Error_abs,ncx,ncy)
```

```
242 Var_min, Var_max = np.abs(Error_abs).min(), np.abs(Error_abs).max
    ()
243 c = ax.pcolor(x, y, z, cmap='jet', vmin=Var_min, vmax=Var_max)
244 ax.set_title('Erro Absoluto')
245 fig.colorbar(c, ax=ax)
246
247 ax = axs[1, 1]
248 z = vec2Grid(Error_rel,ncx,ncy)
249 Var_min, Var_max = np.abs(Error_rel).min(), np.abs(Error_rel).max
    ()
250 c = ax.pcolor(x, y, z, cmap='jet', vmin=Var_min, vmax=Var_max)
251 ax.set_title('Erro Relativo')
252 fig.colorbar(c, ax=ax)
253
254 fig.tight_layout()
255
256 plt.show()
```

Listing 4: Python Mk.3 Code, with iterative solver Gauss-Seidel SOR

E Computational Domain and Boundary Conditions

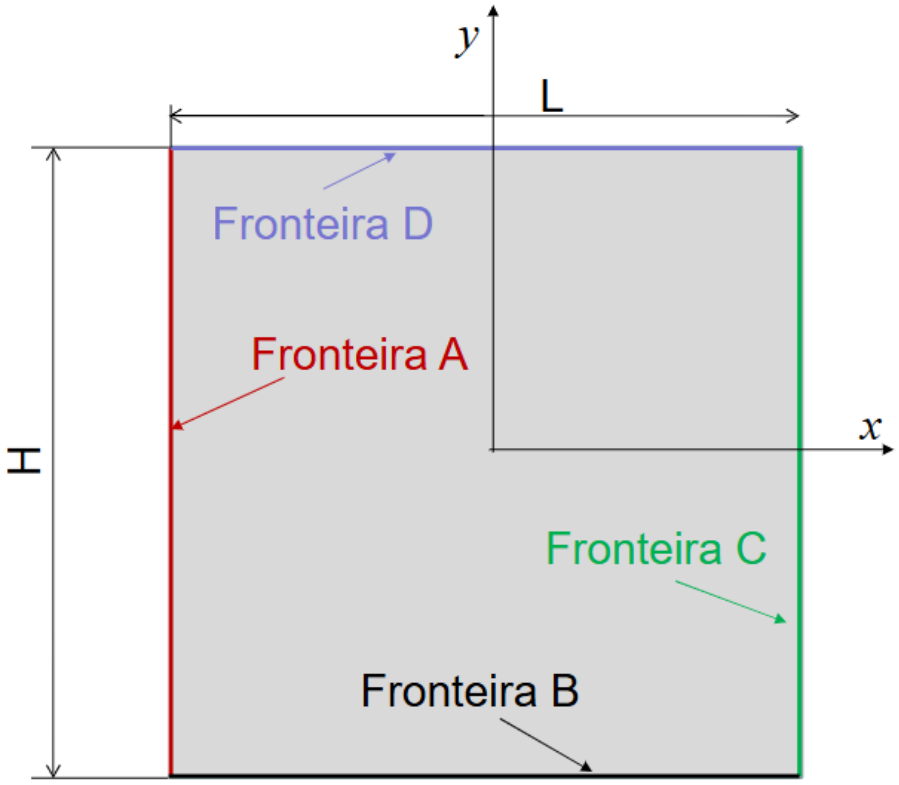

Figure 1: 2D Domain [1].

Problem Data:

$$
\begin{aligned}
&L = 2\text{m} \\
&H = 1\text{m} \\
&k_{ref} = 0.15\textbf{W/(m.K)} \\
&k_y = 0 \\
&k_x = 1 \\
&\text{A - Dirichlet Condition} \\
&\text{B - Dirichlet Condition} \\
&\text{C - Neumann Condition} \\
&\text{D - Neumann Condition}
\end{aligned}
\tag{E.1}
$$

F Plots and Results

F.1 20×10 Mesh

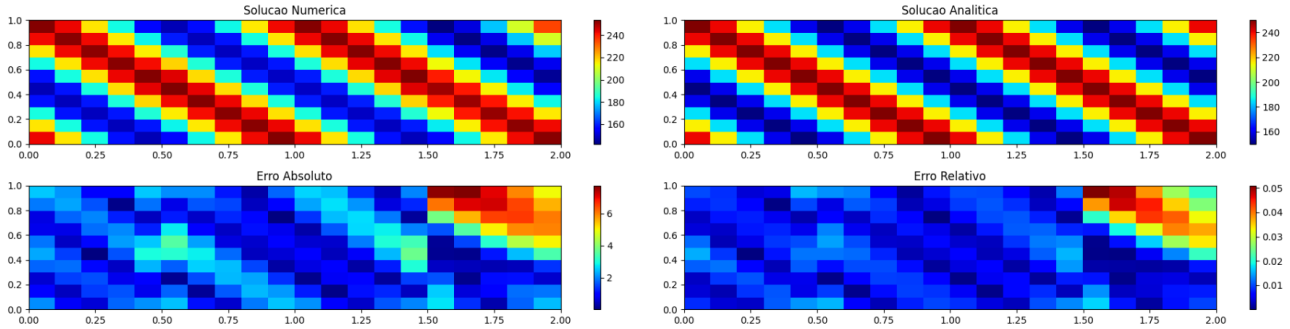

Figure 2: Results for Direct Solver, 20×10 Mesh.

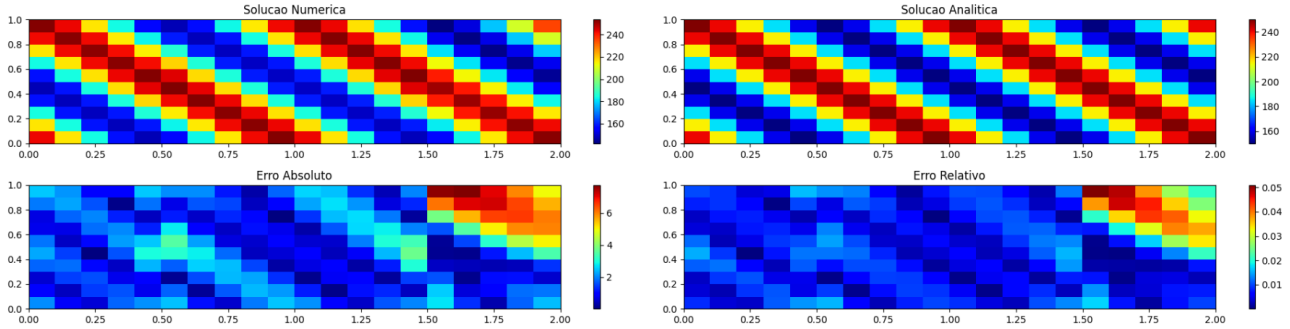

Figure 3: Results for Gauss-Seidel Iterative Method, 20×10 Mesh.

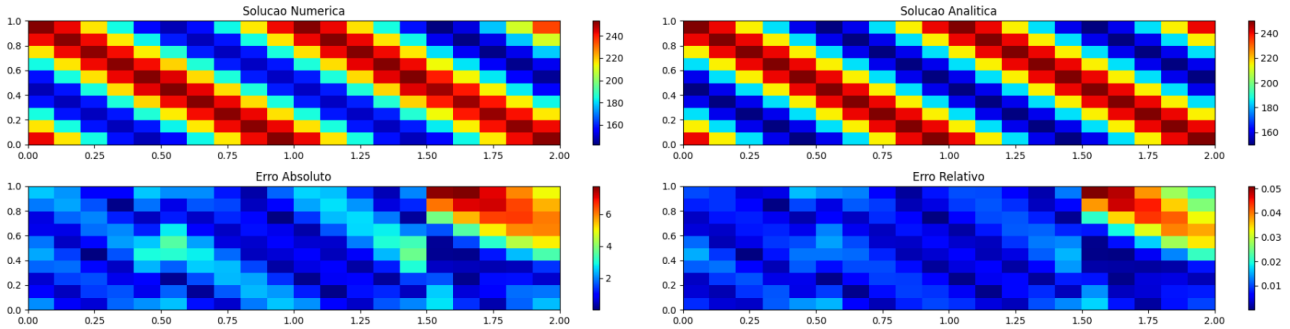

Figure 4: Results for Gauss-Seidel SOR Iterative Method, 20×10 Mesh.

F.2 40×20 Mesh

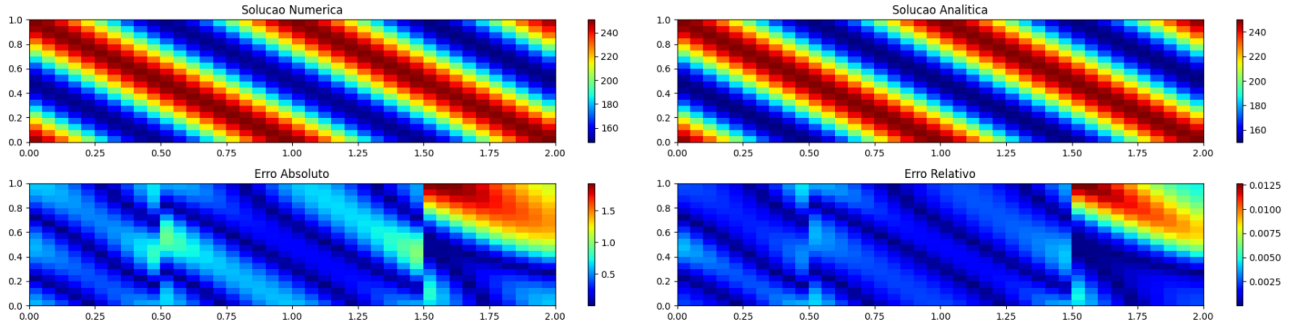

Figure 5: Results for Direct Solver, 40×20 Mesh.

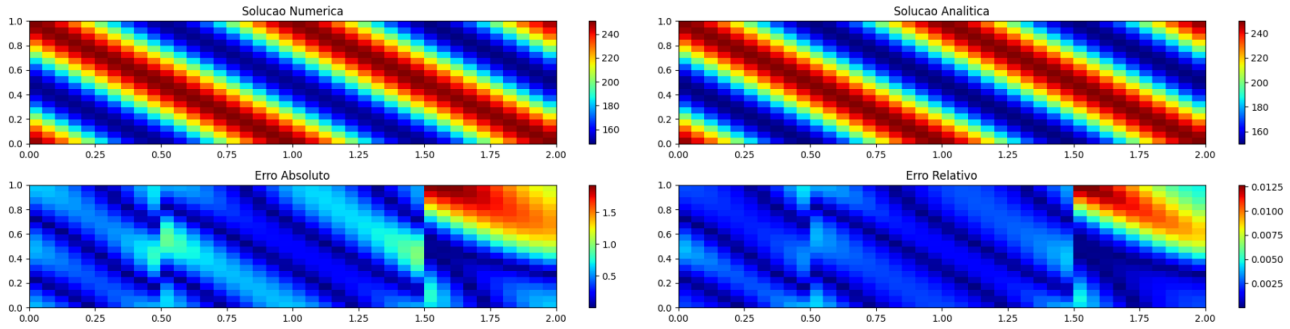

Figure 6: Results for Gauss-Seidel Iterative Method, 40×20 Mesh.

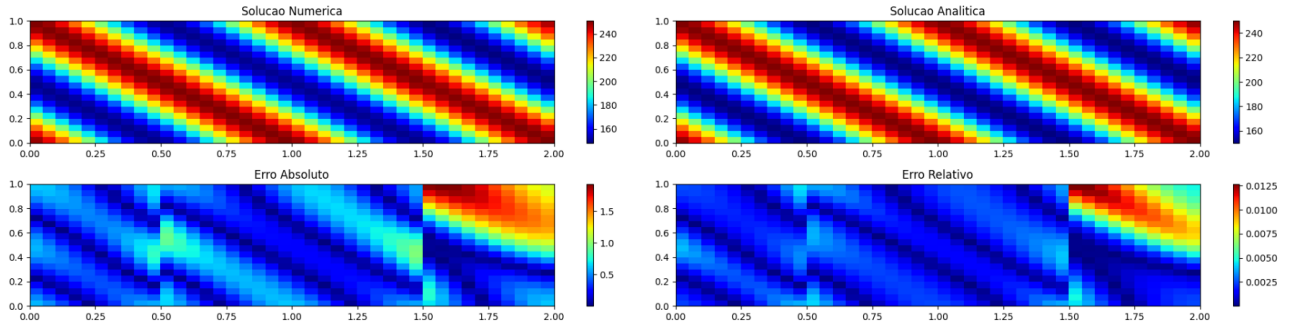

Figure 7: Results for Gauss-Seidel SOR Iterative Method, 40×20 Mesh.

F.3 80×40 Mesh

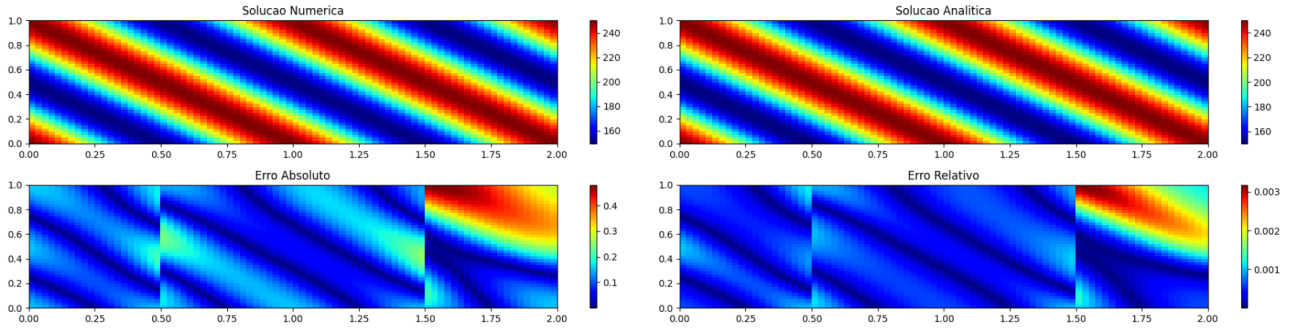

Figure 8: Results for Direct Solver, 80×40 Mesh.

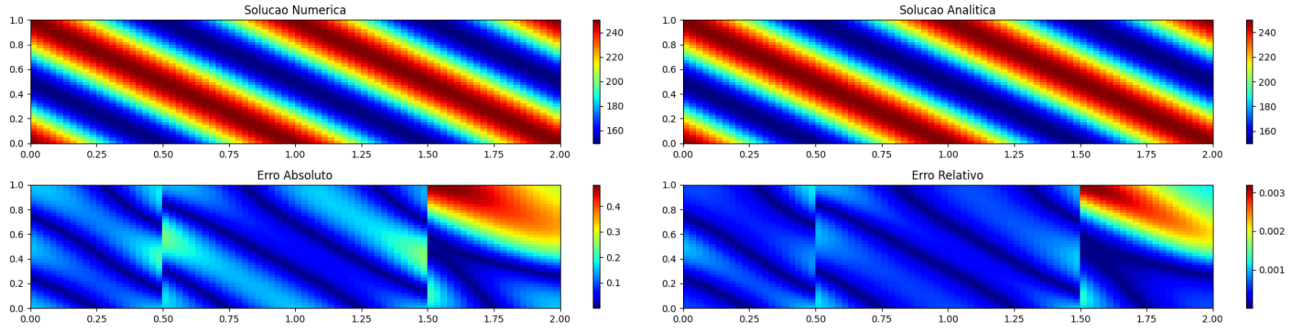

Figure 9: Results for Gauss-Seidel Iterative Method, 80×40 Mesh.

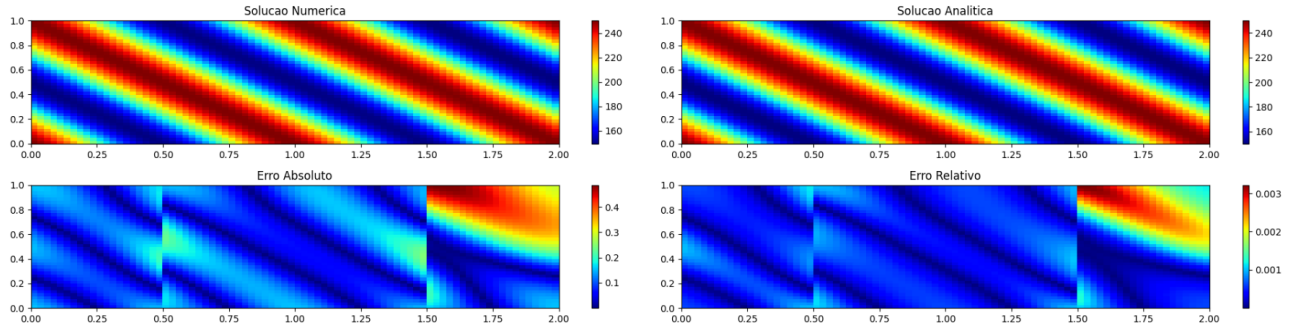

Figure 10: Results for Gauss-Seidel SOR Iterative Method, 80×40 Mesh.

F.4 160×80 Mesh

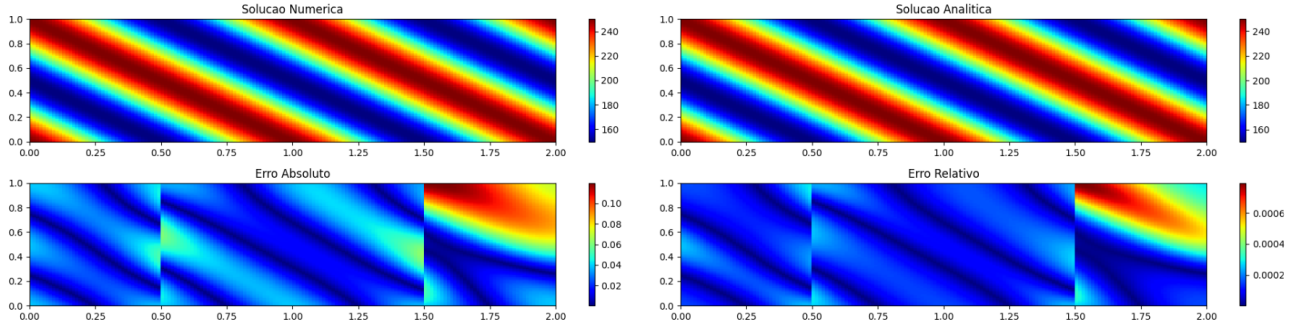

Figure 11: Results for Direct Solver, 160×80 Mesh.

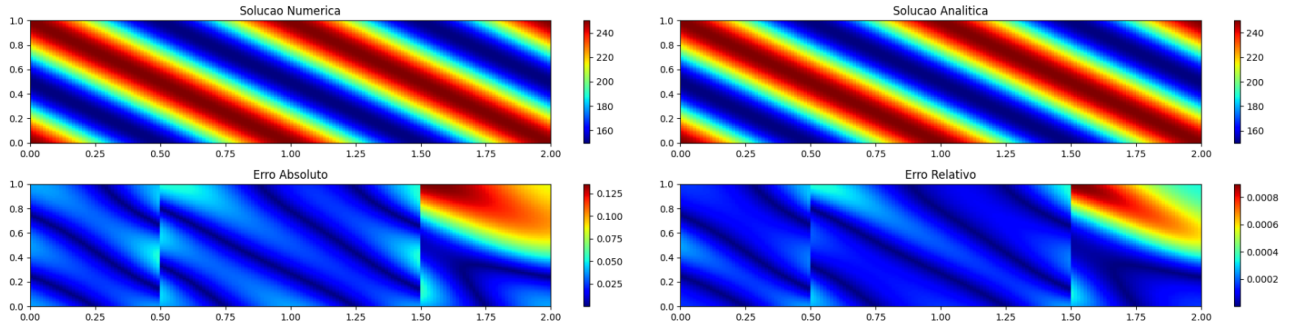

Figure 12: Results for Gauss-Seidel Iterative Method, 160×80 Mesh.

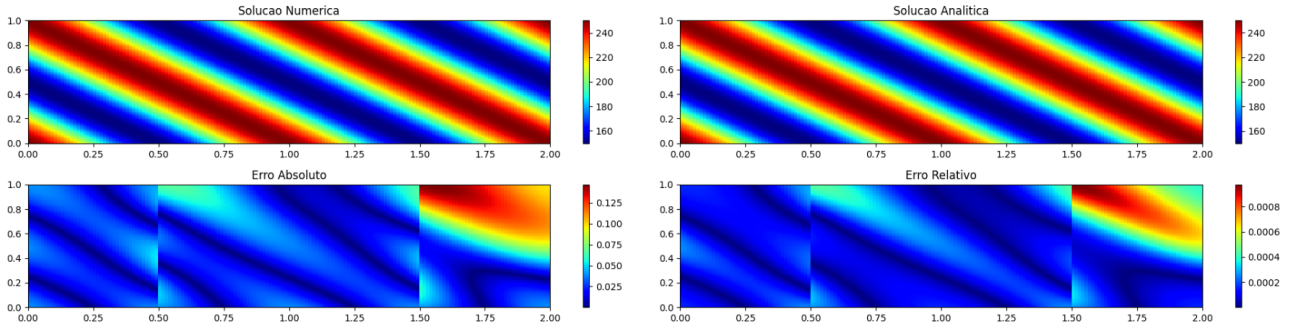

Figure 13: Results for Gauss-Seidel SOR Iterative Method, 160×80 Mesh.

F.5 320×160 Mesh

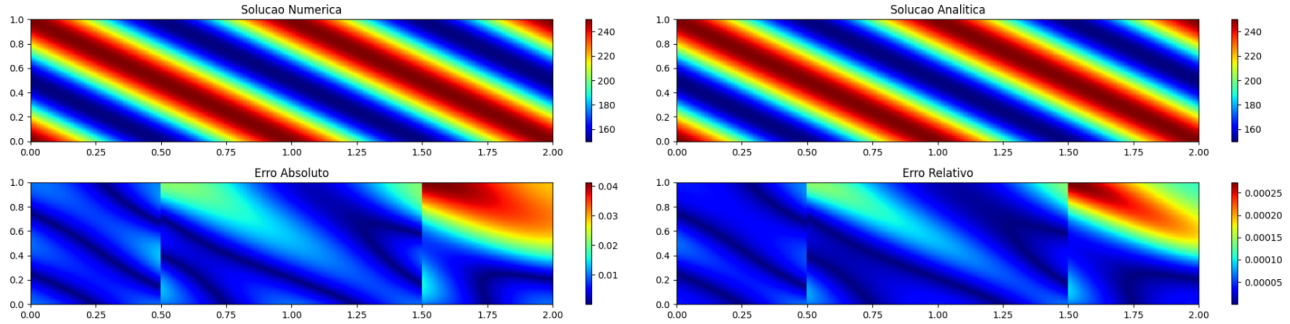

Figure 14: Results for Gauss-Seidel SOR Iterative Method, 320×160 Mesh.