Numerical Simulation of the Differentially Heated Cavity Problem

Master's Degree in Thermal Engineering

Giada Alessi

Universitat Politècnica de Catalunya

April 24, 2025

Abstract

This report presents the numerical solution of the Navier–Stokes equations to study and analyze the benchmark problem of the Differentially Heated Cavity (DHC). The main objective of this work is to understand how to introduce the energy equation into the previously developed lid-driven cavity simulation and how to incorporate buoyancy forces in natural convection problems. The study includes the implementation of the Fractional Step Method, staggered meshes, and different convective schemes for comparison. Results are analyzed for various Rayleigh numbers, $Ra = 10^3, 10^4, 10^5, 10^6$, with particular attention to the development of the thermal and velocity fields. The algorithm has been implemented in `C++`, while the post-processing and visualization of the results have been carried out using a Python script.

Contents

1 Introduction

This section introduces the physical configuration and governing equations of the problem under investigation. The first part provides a detailed description of the differentially heated cavity and the associated boundary and physical conditions, while the second part outlines the theoretical background and the fundamental conservation equations employed in the numerical model.

1.1 Definition of the Problem

The Differentially Heated Cavity (DHC) problem is a classical benchmark in the field of computational fluid dynamics. It represents a fundamental case study for the analysis of natural convection and often serves as a starting point for investigating more complex thermal-fluid phenomena.
The configuration consists of a square cavity of size $L \times L$, filled with air and subject to a horizontal temperature gradient. The left vertical wall is maintained at a hot temperature T_{hot}, while the right vertical wall is kept at a cold temperature T_{cold}. The horizontal walls (top and bottom) are assumed to be adiabatic and stationary. The imposed temperature difference induces a buoyancy-driven flow inside the cavity, leading to natural convection. The fluid motion is modeled using the Boussinesq approximation, which accounts for density variations only in the buoyancy term of the momentum equation.

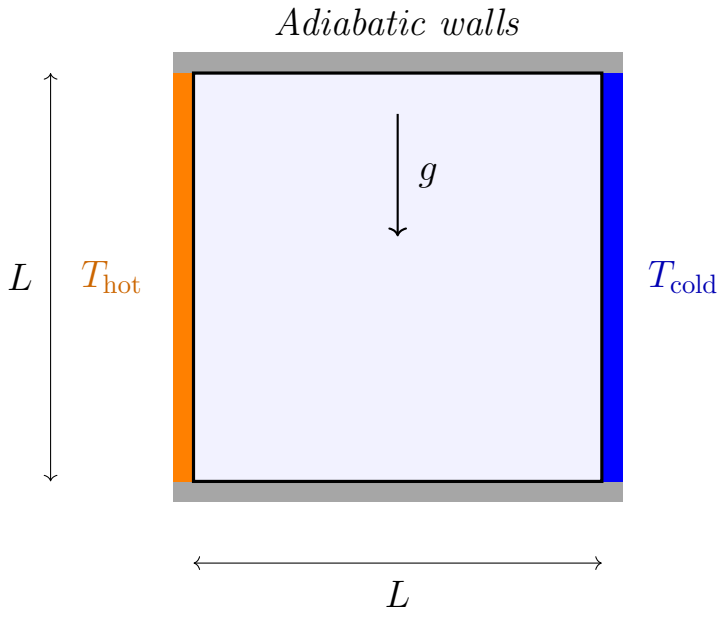


Figure 1: Differentially heated cavity with hot and cold vertical walls and adiabatic horizontal boundaries.

The problem is characterized by the Rayleigh number Ra and the Prandtl number Pr, defined as:

$$Ra = \frac{c_p g \beta \rho^2 (T_{hot} - T_{cold}) L^3}{\mu \lambda}, \qquad Pr = \frac{\mu c_p}{\lambda} \tag{1}$$

The objective is to analyze the resulting convective flow and temperature field once steady state is reached. The following results are presented:

- Isotherms and colormaps of u, v, and T;
- Average Nusselt number Nu;
- Velocity profiles $u(y)$ at $x = 0.5$ and $v(x)$ at $y = 0.5$.

To solve the problem, the following non-dimensional and physical parameters are adopted:

- Temperatures: $T_{hot} = 1, \quad T_{cold} = 0, \quad T_\infty = 0.5$
- Fluid properties: $\mu = 1, \quad \rho = 1, \quad \lambda = 1, \quad \beta = 1$
- Specific heat: $c_p = 0.71$, confirming $Pr = \frac{\mu c_p}{\lambda} = 0.71$ (air)
- Rayleigh numbers considered: $Ra = 10^3, 10^4, 10^5, 10^6$
- The gravitational acceleration g is computed from the definition of the Rayleigh number.

The numerical study presented in this report is based on the benchmark configuration proposed by G. de Vahl Davis [1], adopting the same physical parameters and boundary conditions in order to validate and compare the obtained results with those available in the literature

1.2 Theoretical Background

The governing equations used in this study are a reduced form of the Navier-Stokes equations, suitable for Newtonian and incompressible fluids with velocities below 100 m/s. This approximation is valid for the present problem, assuming that the air inside the cavity remains below this threshold.

- **Mass Conservation:**

$$\nabla \cdot \mathbf{v} = 0 \tag{2}$$

  This equation ensures mass conservation as long as the density remains constant.

- **Momentum Conservation:**

$$\rho \frac{\partial \mathbf{v}}{\partial t} = -\rho(\mathbf{v} \cdot \nabla)\mathbf{v} - \nabla P + \mu \nabla^2 \mathbf{v} - \rho\beta(T - T_\infty)\mathbf{g} \tag{3}$$

  This equation is derived from Newton's Second Law of Motion and includes the Boussinesq approximation to model natural convection. In the nondimensional formulation adopted, the density is set to unity ($\rho = 1$) and the buoyancy term is expressed in terms of the Rayleigh number. Hence, it is the Rayleigh number that governs the natural convection instead of an explicit temperature-dependent density.

[1] de Vahl Davis, G. *Natural convection of air in a square cavity: a benchmark numerical solution.* International Journal for Numerical Methods in Fluids, 1983, Vol. 3, No. 3, pp. 249–264.

- **Energy Conservation:**

$$\rho c_p \left(\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{v}T) \right) = \nabla \cdot (\lambda \nabla T) + \dot{q}_v \tag{4}$$

This is the energy equation governing the temperature distribution, written in a simplified form. The simplifications include neglecting viscous dissipation and pressure work, assuming constant thermophysical properties (such as specific heat and thermal conductivity), and neglecting radiative heat transfer by assuming a non-participating medium. Moreover, the fluid is considered incompressible, and the volumetric heat source term ($\dot{q}_v$) is displayed here for completeness but will be neglected, as it is not present in the problem under study.

The reference temperature T_∞ is chosen as the average of T_{hot} and T_{cold}. Since the air inside the cavity is set in motion by temperature gradients, inducing natural convection, the Boussinesq approximation is applied and will be considered only in the vertical direction.

2 Numerical Implementation

In this section, key numerical concepts such as the Fractional Step Method, time discretizations and staggered meshes are introduced, as they are essential to solve the problem. Then, the spatial discretization of the governing equations, along with the boundary conditions of the cavity, are presented.

2.1 Fractional Step Method

When solving the Navier-Stokes equations, we aim to determine both the velocity and pressure fields simultaneously. If the discretization is not well-posed, spurious numerical solutions (such as a checkerboard pressure pattern) may arise, leading to incorrect velocity values. To address this issue, the Fractional Step Method (FSM) is introduced as an alternative numerical approach. This method separates the computation of the velocity field into three steps: an intermediate prediction, a correction step, and the final computation of the velocity field.

The main objective of the FSM is to decouple the velocity and pressure gradient calculations. To achieve this, a predictor velocity $\mathbf{v}^p$ is introduced by applying the **Helmholtz–Hodge Theorem**. This theorem allows the decomposition of any vector field into the sum of a divergence-free component (the velocity field $\mathbf{v}$, for which $\nabla \cdot \mathbf{v} = 0$) and the gradient of a scalar potential (in this context, the pressure gradient ∇P):

$$\boldsymbol{\omega} = \mathbf{A} + \nabla\phi \quad \rightarrow \quad \boldsymbol{\omega} = \mathbf{v} + \nabla P \tag{5}$$

Specifically, the Helmholtz–Hodge theorem enables the elimination of the pressure gradient term from the momentum conservation equation. To do so, the second-order Adams–Bashforth explicit time integration method is applied to the momentum equation 3, resulting in:

$$\rho \frac{\mathbf{v}^{n+1} - \mathbf{v}^n}{\Delta t} = -\nabla P^{n+1} + \frac{3}{2}\mathbf{R}(\mathbf{v}^n) - \frac{1}{2}\mathbf{R}(\mathbf{v}^{n-1}) + \frac{3}{2}\mathbf{R}_\mathbf{B}(T^n) - \frac{1}{2}\mathbf{R}_\mathbf{B}(T^{n-1}) \tag{6}$$

Here, for simplicity, the convective and diffusive terms are grouped as follows:

$$\mathbf{R}(\mathbf{v}) = -\rho(\mathbf{v}\cdot\nabla)\mathbf{v} + \mu\nabla^2\mathbf{v} \tag{7}$$

and the buoyancy forces, represented using the Boussinesq approximation, are summarized by:

$$\mathbf{R_B}(T) = -\rho\beta(T - T_\infty)\mathbf{g} \tag{8}$$

With this approach, the convective terms are evaluated at $n + \frac{1}{2}$ using an interpolation between the previous and pre-previous timesteps.
Now, the Helmholtz–Hodge theorem can be applied by defining the predictor velocity as:

$$\mathbf{v}^p \equiv \mathbf{v}^{n+1} + \frac{\Delta t}{\rho}\nabla P^{n+1} \tag{9}$$

The implicit temporal discretization of the predictor velocity allows us to strategically substitute it into the momentum conservation equation. First, $\mathbf{v}^{n+1}$ is isolated from Eq. 9:

$$\mathbf{v}^{n+1} = \mathbf{v}^p - \frac{\Delta t}{\rho}\nabla P^{n+1} \tag{10}$$

Then, it is substituted into Eq. 6 to obtain the first step of the FSM:

$$\mathbf{v}^p = \mathbf{v}^n + \frac{\Delta t}{\rho}\left[\frac{3}{2}\mathbf{R}(\mathbf{v}^n) - \frac{1}{2}\mathbf{R}(\mathbf{v}^{n-1}) + \frac{3}{2}\mathbf{R_B}(T^n) - \frac{1}{2}\mathbf{R_B}(T^{n-1})\right] \tag{11}$$

The predictor velocity can now be computed explicitly from previous time steps.
The second step of the FSM is represented by a Poisson equation, which is obtained by applying the divergence operator to both sides of Eq. 9:

$$\nabla\cdot\mathbf{v}^p \equiv \nabla\cdot\mathbf{v}^{n+1} + \frac{\Delta t}{\rho}\nabla\cdot\nabla P^{n+1} \tag{12}$$

The mass is conserved by substituting Eq. 2 ($\nabla\cdot\mathbf{v}^{n+1} = 0$). Recalling that $\nabla\cdot\nabla P^{n+1} = \nabla^2 P$, the Poisson equation is obtained, representing the second step of the FSM:

$$\nabla^2 P^{n+1} = \frac{\rho}{\Delta t}\nabla\cdot\mathbf{v}^p \tag{13}$$

Note that, compared to the momentum equation, a different time discretization is applied in the Poisson equation, which is evaluated implicitly at t^{n+1}, and thus requires the use of an iterative solver such as Gauss–Seidel.
Lastly, the third step is derived by substituting the pressure correction from the Poisson equation into Eq. 10:

$$\mathbf{v}^{n+1} = \mathbf{v}^p - \frac{\Delta t}{\rho}\nabla P^{n+1} \tag{14}$$

Finally, the steps of the Fractional Step Method can be summarized as follows:

- **Step 1:**

$$\mathbf{v}^p = \mathbf{v}^n + \frac{\Delta t}{\rho}\left[\frac{3}{2}\mathbf{R}(\mathbf{v}^n) - \frac{1}{2}\mathbf{R}(\mathbf{v}^{n-1}) + \frac{3}{2}\mathbf{R_B}(T^n) - \frac{1}{2}\mathbf{R_B}(T^{n-1})\right] \tag{15}$$

- **Step 2:**

$$\nabla^2 P^{n+1} = \frac{\rho}{\Delta t} \nabla \cdot \mathbf{v}^p \tag{16}$$

- **Step 3:**

$$\mathbf{v}^{n+1} = \mathbf{v}^p - \frac{\Delta t}{\rho} \nabla P^{n+1} \tag{17}$$

Comparing the steps used in this study to those of the lid-driven cavity, a new term can be observed. The first equation of the FSM includes the Boussinesq approximation in the $\mathbf{R_B}$ term, to account for natural convection effects. The equations are written in a compact vectorial form, but it is important to note that the gravity vector acts only in the y-direction. As a result, the Boussinesq term appears only in the momentum equation for the y-component and not in the x-direction.

2.2 Energy Equation

To solve the Differentially Heated Cavity problem, the energy equation introduced in Equation (4) is considered.
The present study does not consider internal heat sources, meaning that $\dot{q}_v = 0$. The energy conservation equation has been discretized in time using a second-order Adams–Bashforth method, resulting in:

$$\rho c_p \frac{T^{n+1} - T^n}{\Delta t} = \frac{3}{2}\mathcal{R}(T^n) - \frac{1}{2}\mathcal{R}(T^{n-1}) \tag{18}$$

where:

$$\mathcal{R}(T) = -\rho c_p (\mathbf{v} \cdot \nabla T) + \lambda \nabla^2 T \tag{19}$$

2.3 Staggered Meshes

Before proceeding with the spatial discretization of the governing equations, it is important to address a common issue encountered in the numerical resolution of convective regimes.
Instability problems can arise when solving the Navier–Stokes equations using the FSM on a collocated mesh, where both pressure and velocity are stored at the same grid locations. In such cases, the calculation of the pressure gradient may become inaccurate, leading to numerical oscillations. As a direct consequence, if the pressure field exhibits a checkerboard pattern, the computed pressure gradient at the velocity locations will be incorrect, resulting in physically unrealistic velocity values.

To mitigate this issue, a staggered grid approach is introduced, in which three distinct meshes are defined:

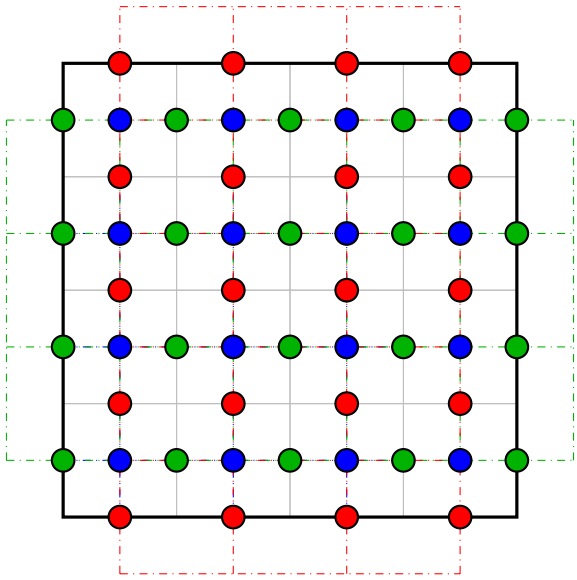
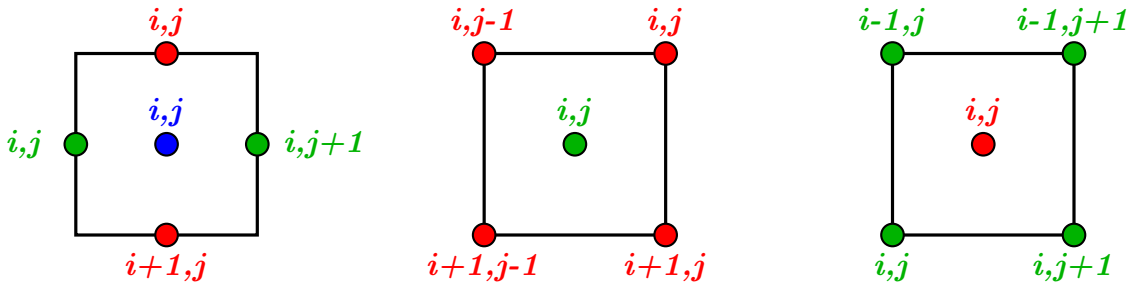


Figure 2: Staggered mesh representation: pressure and temperature mesh in blue, horizontal velocity mesh in green, and vertical velocity mesh in red.

As shown in Figure 2, the pressure mesh shared between P and T, shown in blue, has its nodes and control volumes located at the cell centers, with dimensions $[N_y][N_x]$. The velocity mesh for the horizontal component u, shown in green, places its nodes at the vertical faces of the pressure/temperature mesh and has dimensions $[N_y][N_x + 1]$. Lastly, the velocity mesh for the vertical component v, shown in red, places its nodes at the horizontal faces of the pressure/temperature mesh, with dimensions $[N_y + 1][N_x]$.

In the implemented code for this exercise, matrices have been defined using the notation mentioned above. Consequently, the first node of each mesh is located at the top-left corner, as shown in the figure. The index i, corresponding to rows, ranges from 0 to $Ny - 1/Ny$, proceeding from top to bottom. Similarly, the index j, corresponding to columns, ranges from 0 to $Nx - 1/Nx$, proceeding from left to right. This notation is crucial, as it will be applied in the 'Spatial Discretization' section.



(a) Mesh $P - T$ (b) Mesh u (c) Mesh v

Figure 3: Relative index of surrounding nodes for different meshes.

This staggered grid arrangement, along with the associated indexing, helps to prevent the checkerboard problem and similar numerical instabilities, ensuring a more stable and physically accurate solution.

Unlike the lid-driven cavity problem, the present case also involves the solution of the energy equation. Since the temperature field does not directly interact with the pressure correction step, it has been defined on the same mesh as the pressure. This choice simplifies the implementation without introducing computational conflicts.

2.4 Spatial Discretization

Taking into account all the previous statements and considerations—the Helmholtz–Hodge theorem, the FSM method, temporal discretization, and staggered meshes—the spatial discretization of the governing equations is now presented, term by term.

The spatial discretization is performed using the finite volume method (FVM) on the staggered grid described earlier. For simplicity, the grid is assumed to be uniform, with $\Delta x = \Delta y$ constant across the domain, and with constant ρ and μ. This approach simplifies the equations, allowing us to assume, for instance, that $\Delta S/d_{nb} = 1$ and $\Delta V = (\Delta x)^2$, among other simplifications.
All integrals involved in the discretization are computed over control volumes defined with respect to the reference mesh, denoted respectively as $\mathcal{V}_u$, $\mathcal{V}_v$, $\mathcal{V}_P$, and $\mathcal{V}_T$, corresponding to the control volumes for the horizontal and vertical velocity components, pressure, and temperature, respectively.
Lastly, for clarity in the indexing convention, capital letters denote quantities evaluated at the center of a control volume (node-centered), while lowercase letters indicate quantities evaluated at the control volume faces. The letter P refers to the current (central) node, whereas N, S, E, and W refer to the adjacent nodes in the north, south, east, and west directions, respectively.

2.4.1 Momentum Equation

Convective Term

The convective term in the momentum Navier–Stokes equation 3 is:

$$conv = -\rho\,(\mathbf{v}\cdot\nabla)\,\mathbf{v} \tag{20}$$

First, the general form of the convective term, using the outer product and assuming $\Delta S = \Delta x$, is integrated over the appropriate control volume $\mathcal{V}_u$ or $\mathcal{V}_v$, depending on the velocity component:

$$-\rho\int_{\mathcal{V}} \nabla\cdot(\mathbf{v}\otimes\mathbf{v})\;\, d\mathcal{V} = -\rho\int_{\mathcal{V}} \mathbf{v}\otimes\mathbf{v}\cdot d\boldsymbol{S} \tag{21}$$

Finally, after applying the finite volume method and considering a uniform 2D mesh, the x- and y-components of the convective term become:

$$conv_x = -\rho\int_{\mathcal{V}_u} (\mathbf{v}\cdot\nabla)\;\, d\mathcal{V} = -\rho\Delta x\,(v_n u_n - v_s u_s + u_e u_e - u_w u_w) \tag{22}$$

$$conv_y = -\rho\int_{\mathcal{V}_v} (\mathbf{v}\cdot\nabla)\;\, d\mathcal{V} = -\rho\Delta x\,(v_n v_n - v_s v_s + u_e v_e - u_w v_w) \tag{23}$$

Diffusive Term

The diffusive term in the momentum equation 3 is given by:

$$diff = \mu \nabla^2 \mathbf{v} \tag{24}$$

Integrating over a control volume and applying Gauss's theorem to the two components of velocity, u and v, assuming constant viscosity μ, we obtain:

$$\int_{\mathcal{V}_u} \mu \nabla^2 u \; d\mathcal{V} = \int_{\mathcal{V}_u} \mu \nabla \cdot \nabla u \; d\mathcal{V} = \int_{S_u} \mu \nabla u \cdot d\boldsymbol{S} \simeq \mu \sum_{nb} \frac{u_{nb} - u_p}{d_{nb}} \Delta S \tag{25}$$

$$\int_{\mathcal{V}_v} \mu \nabla^2 v \; d\mathcal{V} = \int_{\mathcal{V}_v} \mu \nabla \cdot \nabla v \; d\mathcal{V} = \int_{S_v} \mu \nabla v \cdot d\boldsymbol{S} \simeq \mu \sum_{nb} \frac{v_{nb} - v_p}{d_{nb}} \Delta S \tag{26}$$

For a uniform 2D mesh, assuming $\Delta S / d_{nb} = 1$, the diffusive terms for the u- and v-components simplify to:

$$diff_x = \mu \left((u_N - u_P) + (u_S - u_P) + (u_E - u_P) + (u_W - u_P) \right) \tag{27}$$

$$diff_y = \mu \left((v_N - v_P) + (v_S - v_P) + (v_E - v_P) + (v_W - v_P) \right) \tag{28}$$

Boussinesq Approximation Term

The Boussinesq approximation term in the Navier–Stokes equation 3 is given by:

$$\mathbf{R_B} = -\rho \beta (T - T_\infty) \mathbf{g} \tag{29}$$

Since gravity acts only in the vertical direction, this term contributes only to the y-momentum equation and is therefore integrated over the control volume $\mathcal{V}_v$. Assuming a 2D uniform mesh with $\Delta \mathcal{V}_v = (\Delta x)^2$ and constant ρ and β, we obtain:

$$- \int_{\mathcal{V}_v} \rho \beta (T - T_\infty) \mathbf{g} \, d\mathcal{V} \tag{30}$$

The discretized form becomes:

$$R_{Bx} = 0 \tag{31}$$

$$R_{By} = -\rho \beta (T - T_\infty)(-g) \; (\Delta x)^2 = \rho \beta (T - T_\infty) g \; (\Delta x)^2 \tag{32}$$

2.4.2 Poisson Equation

To discretize the Poisson equation 16, for a non-boundary node in a uniform 2D mesh, we integrate over an infinitesimal control volume belonging to the staggered-p mesh, denoted as $\mathcal{V}_p$:

$$\int_{\mathcal{V}_p} \nabla \cdot \nabla P \; d\mathcal{V} = \int_{\mathcal{V}_p} \frac{\rho}{\Delta t} \nabla \cdot \mathbf{v}^p \; d\mathcal{V} \tag{33}$$

then, applying the Gauss theorem:

$$\int_{\mathcal{S}_p} \nabla P \; d\mathcal{S} = \frac{\rho}{\Delta t} \int_{\mathcal{S}_p} \mathbf{v}^p \; d\mathcal{S} \tag{34}$$

$$\sum \frac{P_{nb} - P_P}{d_{nb}} \Delta S = \frac{\rho}{\Delta t} \sum \mathbf{v}^p_{nb} \cdot \Delta \boldsymbol{S} \tag{35}$$

$$P_N + P_S + P_E + P_W - 4P_P = \frac{\rho \Delta x}{\Delta t} \left(v^p_n - v^p_s + u^p_e - u^p_w \right) \tag{36}$$

2.4.3 Energy Equation

The resolution of the energy equation 4 allows us to compute the evolution of the temperature field over time. The second-order Adams–Bashforth explicit time integration scheme has been applied, leading to the following equation, where internal sources are neglected:

$$\rho c_p \frac{T^{n+1} - T^n}{\Delta t} = \frac{3}{2}\mathcal{R}(T^n) - \frac{1}{2}\mathcal{R}(T^{n-1}) \tag{37}$$

where, once again, the term $\mathcal{R}(T)$ is given by:

$$\mathcal{R}(T) = -\rho c_p(\mathbf{v} \cdot \nabla T) + \lambda \nabla^2 T \tag{38}$$

Although the temperature is stored at the same locations as the pressure (both share the same reference mesh), we denote the associated control volume as $\mathcal{V}_T$ to maintain consistency with the thermal variable and avoid confusion across equations.
Applying the finite volume method and integrating Eq. 37 over a control volume $\mathcal{V}_T$, we obtain:

$$\int_{\mathcal{V}_T} \rho c_p \frac{T^{n+1} - T^n}{\Delta t}\, d\mathcal{V} = -\int_{\mathcal{V}_T} \rho c_p \nabla \cdot (\mathbf{v}T)\, d\mathcal{V} + \int_{\mathcal{V}_T} \nabla \cdot (\lambda \nabla T)\, d\mathcal{V} \tag{39}$$

Then, the Gauss divergence theorem is applied to each term of the equation, assuming a uniform mesh and the simplifications introduced previously.

Transient Term

The first term of the equation, representing the internal energy variation over time, becomes:

$$\int_{\mathcal{V}_T} \rho c_p \frac{T^{n+1} - T^n}{\Delta t}\, d\mathcal{V} = \rho c_p \frac{T^{n+1} - T^n}{\Delta t}(\Delta x)^2 \tag{40}$$

Convective Term

The convective term, representing the transport of energy due to fluid motion, becomes:

$$-\int_{\mathcal{V}_T} \rho c_p \nabla \cdot (\mathbf{v}T)\, d\mathcal{V} = -\rho c_p \int_{\mathcal{S}_T} \mathbf{v}T\, d\mathbf{S} = -\rho c_p \Delta x (v_n T_n - v_s T_s + u_e T_e - u_w T_w) \tag{41}$$

Diffusive Term

Finally, the diffusive term, representing heat conduction due to temperature gradients, becomes:

$$\int_{\mathcal{V}_T} \nabla \cdot (\lambda \nabla T)\, d\mathcal{V} = \lambda \int_{\mathcal{S}_T} \nabla T\, d\boldsymbol{S}$$

$$\lambda \int_{\mathcal{S}_T} \nabla T\, d\boldsymbol{S} = \lambda \Delta x \left(\frac{T_N - T_P}{d_{PN}} + \frac{T_S - T_P}{d_{PS}} + \frac{T_E - T_P}{d_{PE}} + \frac{T_W - T_P}{d_{PW}} \right) \tag{42}$$

In Eq. 42, the terms d_{PN}, d_{PS}, d_{PE}, and d_{PW} represent the distances between the control volume center P and its neighboring nodes N, S, E, and W, respectively. For internal nodes, the distance is $d = \Delta x$, and thus Eq. 42 can be further simplified as follows:

$$\lambda \int_{\mathcal{S}_T} \nabla T\, d\boldsymbol{S} = \lambda \left(T_N + T_S + T_E + T_W - 4T_P\right) \tag{43}$$

2.5 Convective Schemes

It must be noted that the convective term in the momentum equations, as well as part of the Poisson equation, requires the velocity to be evaluated at the control volume faces rather than at the nodes. Since the exact value at the face is not known, an interpolation method must be applied.
Initially, the Central Difference Scheme (CDS) was selected as a first approach to solving the exercise. Later, the Upwind Difference Scheme (UDS) was implemented to analyze how the code behaves with the introduction of increased numerical diffusion.

In this section, the two schemes are described using a generic scalar property ϕ, which can also represent a velocity component, as in the present case.

2.5.1 Central Difference Scheme (CDS)

The central idea behind CDS is to approximate the value at the face using the known values at the adjacent nodes through linear interpolation.

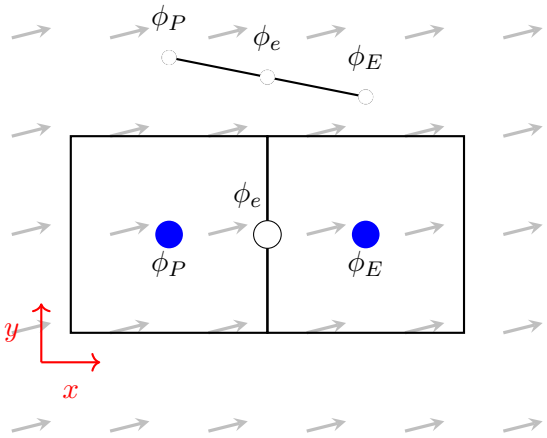


Figure 4: Central Difference Scheme (CDS)

Following the example shown in Figure 4, and applying this method to each face, the interpolation is given by:

$$\phi_e = \frac{\phi_P + \phi_E}{2} \tag{44}$$

This scheme is not particularly dissipative, but it tends to be less stable and may produce spurious oscillations if the chosen time step is not sufficiently small.

2.5.2 Upwind Difference Scheme (UDS)

The Upwind Difference Scheme (UDS) is based on the idea that the value of a transported quantity at a face should be taken from the upstream (or upwind) side of the flow. This approach assumes that the quantity transported by the flow keeps the characteristics of the region it comes from.
In practice, this means that the face value ϕ_e is assigned based on the sign of the velocity at the face. If the velocity is positive, the fluid is coming from the left (the west cell), and the upstream value is ϕ_P. If the velocity is negative, the fluid is coming from the right (the east cell), and the upstream value is ϕ_E. This logic is illustrated in Figure 5.

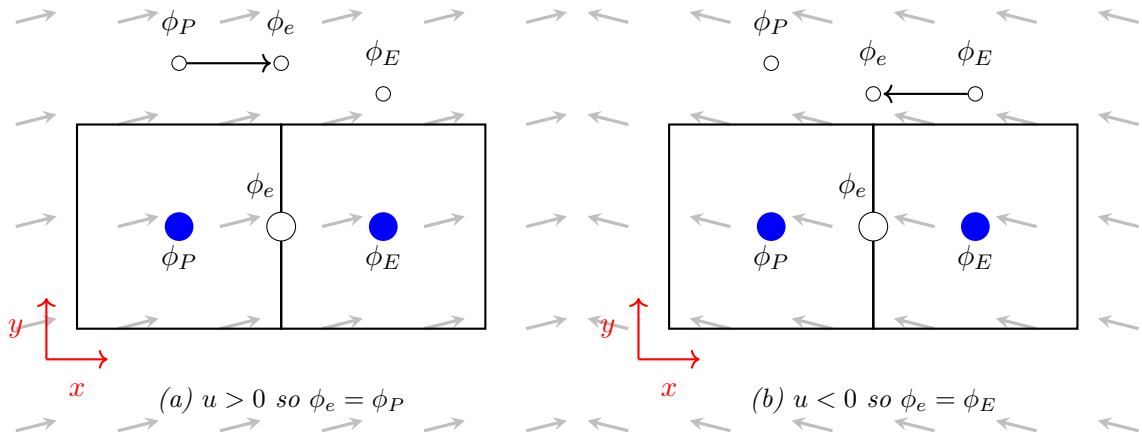


Figure 5: Upwind Difference Scheme (UDS)

Accordingly, the face interpolation is expressed as:

$$\phi_e = \begin{cases} \phi_P, & \text{if } u > 0 \\ \phi_E, & \text{if } u < 0 \end{cases} \tag{45}$$

This scheme is more diffusive than the Central Difference Scheme (CDS), which improves its stability properties. However, the downside is that it can introduce excessive numerical diffusion, particularly in regions with steep gradients, thus reducing the accuracy of the solution.

2.6 Boundary Conditions

The boundary conditions applied to the domain are essential to ensure the physical correctness and numerical stability of the solution. In this problem, the following conditions are imposed on the velocity, pressure, and temperature fields:

Velocity field (u, v):
A stationary walls condition is applied on all boundaries of the cavity, which corresponds to setting both velocity components to zero:

$$u = 0, \quad v = 0 \qquad \text{on all walls}$$

Pressure field P:
Neumann boundary conditions are imposed along all domain boundaries, implying zero pressure gradient normal to the walls:

$$\frac{\partial P}{\partial n} = 0 \qquad \text{on all boundaries}$$

where $\partial/\partial n$ denotes the derivative in the direction normal to the wall. In discrete form, this condition is implemented by assigning the pressure at the boundary node equal to that of its adjacent interior node. For instance:

- On the West wall: $P = P_E$

- On the East wall: $P = P_W$
- On the South wall: $P = P_N$
- On the North wall: $P = P_S$

Temperature field T:

- On the vertical boundaries, Dirichlet conditions are applied:

$$T = T_{\text{hot}} = 1 \quad \text{on the West wall,} \qquad T = T_{\text{cold}} = 0 \quad \text{on the East wall}$$

  These conditions are imposed directly by assigning the temperature value at the boundary nodes.

- On the horizontal boundaries (top and bottom), adiabatic conditions are assumed, corresponding to zero temperature gradient in the direction normal to the wall:

$$\frac{\partial T}{\partial n} = 0 \qquad \text{on the North and South walls}$$

  In discrete form, this is implemented by setting the boundary temperature equal to that of the adjacent interior node:

  - On the South wall: $T = T_N$
  - On the North wall: $T = T_S$

These conditions are applied according to the staggered grid setup explained in Section 2.3, and they are essential to correctly simulate the natural convection inside the cavity.

2.7 Nusselt Number

To quantify the convective heat transfer, the average Nusselt number is computed based on the local Nusselt number Nu_{x^*}, evaluated by integrating the dimensionless heat flux along a vertical line at a given position x^*. Figure 6 shows a schematic representation of the heat flux and the definition of Nu_{x^*}.

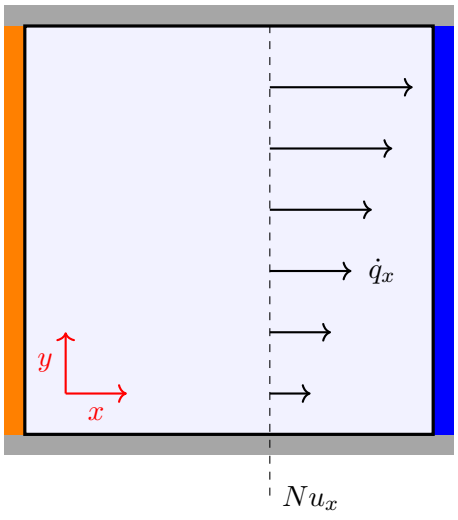


Figure 6: Heat flux $\dot{q}_x$ and local Nusselt number Nu_x.

The nondimensional formulation adopts:

$$x^* = \frac{x}{L}, \quad y^* = \frac{y}{L}, \quad T^* = \frac{T - T_{\text{cold}}}{T_{\text{hot}} - T_{\text{cold}}}, \quad u^* = \frac{uL}{\alpha}, \quad \alpha = \frac{\lambda}{\rho c_p}$$

The local heat flux in the x-direction is given by:

$$\dot{q}_x = u^* T^* - \frac{\partial T^*}{\partial x^*}$$

and the corresponding local Nusselt number is:

$$Nu_{x^*} = \int_0^1 \left(u^* T^* - \frac{\partial T^*}{\partial x^*} \right) dy^*$$

In the numerical implementation, $\dot{q}_x$ is computed at each vertical column of control volumes by discretizing the convective and conductive terms. The temperature at the face is interpolated using either an upwind (UDS) or central (CDS) scheme, and the spatial derivative is approximated by finite differences consistent with the chosen method. The integral is evaluated as a summation over the vertical grid cells:

$$Nu_{x^*} \approx \sum_i \dot{q}_x \, \Delta y^*$$

This operation is repeated for all vertical lines across the domain. Finally, the average Nusselt number is computed as the arithmetic mean of the local values. Although the theoretical flux should remain constant across the domain, small variations may appear due to discretization and interpolation errors.

2.8 Resolution Procedure

In this section, the resolution procedure is described by applying the previously introduced discretized equations and numerical concepts. Although the velocity and temperature fields are coupled through the Boussinesq approximation, their computation can proceed sequentially. In this work, the solution starts with the evaluation of the velocity field using the Fractional Step Method, followed by the computation of the steady-state temperature distribution based on the resulting flow field. The chosen convective scheme and boundary conditions are consistently applied throughout the process. Finally, the average Nusselt number is calculated and compared with benchmark results provided by G. de Vahl Davis.
This step-by-step description does not include details on the indexing implementation, as it depends on the chosen convective scheme, nor the boundary conditions applied to the border nodes. For implementation details, refer to the staggered mesh paragraph in Section 2.3 and the boundary conditions in Section 2.6.
First, the velocity field is computed by following the steps of the Fractional Step Method (FSM):

- **Step 1a: Predictor Velocity** u^p
  In the first step, the horizontal component of the predictor velocity is computed. Since the second-order Adams–Bashforth time integration scheme is adopted, the convective–diffusive terms at both the previous time step $R(u^n)$ and the one before

that $R(u^{n-1})$ must be evaluated. Here, $R(u)$ represents the sum of the convective and diffusive terms in the x-momentum equation, i.e., $R(u) = \text{conv}_x + \text{diff}_x$, as described in Section 2.4.1 and defined explicitly in Eqs. (22) and (27). Once the convective–diffusive terms have been computed, the horizontal component of the predictor velocity u^p can be evaluated as:

$$u^p = u^n + \frac{\Delta t}{\rho(\Delta x)^2}\left(\frac{3}{2}R(u^n) - \frac{1}{2}R(u^{n-1})\right) \tag{46}$$

For the proper indexing of the quantities involved in the computation of u^p, refer to the indexing of the u-mesh shown in Fig. 3.

- **Step 1b: Predictor Velocity v^p**
  Following the same procedure used for u^p and applying the same time integration scheme, the vertical component of the predictor velocity v^p is evaluated. In this case, the buoyancy term must also be taken into account. Here, $R(v)$ represents the sum of the convective and diffusive terms in the y-momentum equation, i.e., $R(v) = \text{conv}_y + \text{diff}_y$, as described in Section 2.4.1 and defined in Eqs. (23) and (28). The buoyancy contributions $R_{B_y}^{\,n}$ and $R_{B_y}^{\,n-1}$ are defined in Eq. (32).

  The vertical predictor velocity is computed as:

$$v^p = v^n + \frac{\Delta t}{\rho(\Delta x)^2}\left(\frac{3}{2}R(v^n) - \frac{1}{2}R(v^{n-1}) + \frac{3}{2}R_{B_y}^{\,n} - \frac{1}{2}R_{B_y}^{\,n-1}\right) \tag{47}$$

  For the proper indexing of the quantities involved in the computation of v^p, refer to the indexing of the v-mesh shown in Fig. 3.

- **Step 2: Pressure P^{n+1}**
  The second step of the FSM consists in solving the Poisson equation to obtain the pressure at each control volume of the P-mesh. The pressure is computed implicitly using a Gauss–Seidel solver applied to the following discretized equation:

$$P^{n+1} = \frac{1}{4}\left[P_N^{n+1} + P_S^{n+1} + P_E^{n+1} + P_W^{n+1} - \frac{\rho\,\Delta x}{\Delta t}\left(v_n^p - v_s^p + u_e^p - u_w^p\right)\right] \tag{48}$$

  This equation is valid only for a uniform grid, where the grid spacing is the same in both directions ($\Delta x = \Delta y$). In this case, the terms involving Δx and Δy cancel out, leading to this simplified form. If the grid were non-uniform, a more general version of the Poisson equation would have to be used.

  Once the Poisson equation is solved, the corrected velocity field can be obtained in the third and final step.

- **Step 3: Velocities u^{n+1} and v^{n+1}**
  In this final step, the updated velocity field of the cavity is computed using the pressure values obtained from the previous step. The velocity components are corrected as follows:

$$u^{n+1} = u^p - \frac{\Delta t}{\rho\,\Delta x}\left(P_P^{n+1} - P_W^{n+1}\right) \tag{49}$$

$$v^{n+1} = v^p - \frac{\Delta t}{\rho\,\Delta x}\left(P_P^{n+1} - P_S^{n+1}\right) \tag{50}$$

The pressure gradient required for this velocity correction step is discretized using central differences between adjacent pressure nodes: the x-component is computed between nodes P and W to update the velocity u located at the east face, while the y-component is computed between P and S to update the velocity v located at the north face. This is consistent with the positioning of variables in the staggered mesh.

At this point, all the steps of the Fractional Step Method have been completed, and the velocity field is fully evaluated.

- **Step 4: Temperature** T^{n+1}
  Having obtained the velocity field, it is now possible to compute the temperature field by solving the previously discretized energy equation. Also in this case, the Adams–Bashforth second-order time integration scheme is adopted. As for the velocity prediction in Step 1, the convective–diffusive terms must be evaluated first.

  Here, $R_T(T)$ denotes the sum of the convective and diffusive contributions in the energy equation, as described in Section 2.4.3 and defined in Eqs. (41) and (42). The terms $R_T(T^n)$ and $R_T(T^{n-1})$ are evaluated at time levels t^n and t^{n-1}, respectively.

  Finally, the temperature at the current time step t^{n+1} is computed as:

$$T^{n+1} = T^n + \frac{\Delta t}{\rho c_p (\Delta x)^2}\left(\frac{3}{2}R_T(T^n) - \frac{1}{2}R_T(T^{n-1})\right) \tag{51}$$

The full resolution procedure described above must be iteratively repeated over successive time steps until convergence to a steady-state solution is achieved. This is verified by monitoring the residuals of the governing equations and ensuring that their values fall below a predefined threshold.

2.9 Step-by-Step Algorithm

Merging all the previously described elements, the step-by-step resolution algorithm for the Differentially Heated Cavity (DHC) is presented. The algorithm makes use of dedicated functions to evaluate convective-diffusive terms, implement the Gauss-Seidel solver for the Poisson equation, impose boundary conditions, and compute the average Nusselt number of the simulation. An abstract class is implemented to improve modularity and to allow the selection between two different schemes and convergence behaviors. The code also enables the user to input specific data to characterize the problem, such as the Rayleigh number, as well as to define the appropriate mesh refinement and the desired convective scheme.

A summarized flow of the code is now presented, with a brief explanation of the purpose of the classes, functions, and the various convergence criteria and auxiliary routines:

- **Definition of modular classes and functions:**
  - `ConvectiveScheme` is an abstract class implementing the interface for convective-diffusive terms and post-processing.

  - `CDSMethod` and `UDSMethod` are the two methods implemented in the `ConvectiveScheme` class, providing specific implementations of the convective scheme using the Central Differencing Scheme or the Upwind Differencing Scheme.
  - **BoundaryConditions()** imposes Dirichlet and Neumann boundary conditions for the temperature field.
  - **PoissonSolver()** solves the pressure correction equation using a Gauss-Seidel iterative solver, including over-relaxation (ω_P) to accelerate convergence.

- **Definition of global matrices and parameters** inside the **main()** function:
  - Pressure matrices: `P1`, `Pg`.
  - Temperature matrices: `Tn_1`, `Tn`, `T1`.
  - Velocity field matrices (staggered): `un_1`, `un`, `u1`, `uP`, `vn_1`, `vn`, `v1`, `vP`.
  - Convective-diffusive terms: `Run_1`, `Run`, `Ru1`, `Rvn_1`, `Rvn`, `Rv1`, `Rtn_1`, `Rtn`, `Rt1`.
  - Buoyancy force terms: `Rbn`, `Rbn_1`.

- **Initialization phase:**
  - Physical and numerical parameters are defined, including the Rayleigh number, mesh refinement, and convective scheme as input by the user.
  - Initial velocity fields are set to zero; the initial temperature field is linearly distributed to improve convergence.
  - Temperature boundary conditions are applied.
  - Convective-diffusive terms R^u, R^v, and R^T are computed for time levels $n-1$ and n.

The steady-state solution is iteratively achieved through the following time-marching procedure:

- **Predictor step for velocities:**
  - The predicted horizontal velocity field is computed as:

$$u_{i,j}^P = u_{i,j}^n + \frac{\Delta t}{\rho \Delta x^2}\left(\frac{3}{2}R(u)^n - \frac{1}{2}R(u)^{n-1}\right)$$

  - The predicted vertical velocity includes buoyancy terms via the Boussinesq approximation:

$$v_{i,j}^P = v_{i,j}^n + \frac{\Delta t}{\rho \Delta x^2}\left(\frac{3}{2}R(v)^n - \frac{1}{2}R(v)^{n-1} + \frac{3}{2}R_B^n - \frac{1}{2}R_B^{n-1}\right)$$

    Both predictor velocities are evaluated with Explicit Euler time integration for the first five iterations to better stimulate the system in the initial steps, after that, the Adams–Bashforth time integration scheme is applied.

- **Pressure correction step:**

  - The Poisson equation is solved implicitly on the pressure mesh using the Gauss-Seidel solver implemented in `PoissonSolver()`:

$$\nabla^2 P^{n+1} = \frac{\rho}{\Delta t} \nabla \cdot \mathbf{u}^P$$

  - Over-relaxation is applied to stabilize and improve convergence.

- **Corrector step:**
  - Velocities are corrected and finally computed using pressure gradients:

$$u^{n+1} = u^P - \frac{\Delta t}{\rho \Delta x} \left(P_{i,j} - P_{i,j-1} \right), \quad v^{n+1} = v^P - \frac{\Delta t}{\rho \Delta x} \left(P_{i-1,j} - P_{i,j} \right)$$

  - Residuals for both velocity components are evaluated.

- **Temperature evaluation step:**
  - The temperature field is advanced in time with:

$$T_{i,j}^{n+1} = T_{i,j}^n + \frac{\Delta t}{\rho c_p \Delta x^2} \left(\frac{3}{2} \mathcal{R}(T)^n - \frac{1}{2} \mathcal{R}(T)^{n-1} \right)$$

  - Neumann and Dirichlet boundary conditions for temperature are applied at each time step. The same time integration scheme as for the predictor velocities is used.

- **Field updates:**
  - All fields (u, v, T) are updated using a linear under-relaxation:

$$\phi^{n+1} = \phi^n + \omega_\phi (\phi^{n+1} - \phi^n)$$

    where ω_ϕ is the relaxation factor for the respective field.
  - Previous values are shifted for the next iteration: $\phi^{n-1} \leftarrow \phi^n \leftarrow \phi^{n+1}$. Convective-diffusive terms are re-computed with the updated values.

- **Adaptive time stepping:**
  - The time step is dynamically adjusted based on three criteria:

$$\Delta t_c = 0.35 \frac{\Delta x L}{\mathbf{v}_{\max}}, \quad \Delta t_d = 0.20 \frac{\Delta x^2}{\nu}, \quad \Delta t_T = 0.20 \frac{\Delta x^2}{\alpha}$$

    where $\nu = \frac{\mu}{\rho}$ and $\alpha = \frac{\lambda}{\rho c_P}$.
  - The minimum among these is selected to ensure stability:

$$\Delta t = \min \{ \Delta t_c, \Delta t_d, \Delta t_T \}$$

- **Convergence check and loop restart:**
  - Residuals for u, v, and T are checked:

$$\max(\text{Res}) < 10^{-6} \quad \text{(velocity)}, \quad \max(\text{Res}_T) < 10^{-4} \quad \text{(temperature)}$$

  - Only the temperature residual represents the convergence criterion, for reasons that will be explained later in Section 3. If not satisfied, the loop continues. Otherwise, the solution is assumed to have reached steady-state, and the loop exits before further updates, to obtain a clean result not influenced by relaxation factors.

- **Post-processing:**
  - Several files are generated:
    - Temperature distribution (`TemperatureDistribution.txt`)
    - Velocity fields (`VelocityUDistribution.txt`, `VelocityVDistribution.txt`)
    - Central profiles (`Uy.txt`, `Vx.txt`)
  - The average Nusselt number is computed, as well as the maximum dimensionless velocities u^* and v^* along the midplanes.
  - Colormaps and plots are generated by a Python script, which uses the `pandas` library to read the `.txt` files produced by the C++ code, and `matplotlib` to generate the plots.

For better visualization and to summarize the overall procedure, the flowchart of the implemented algorithm is provided:

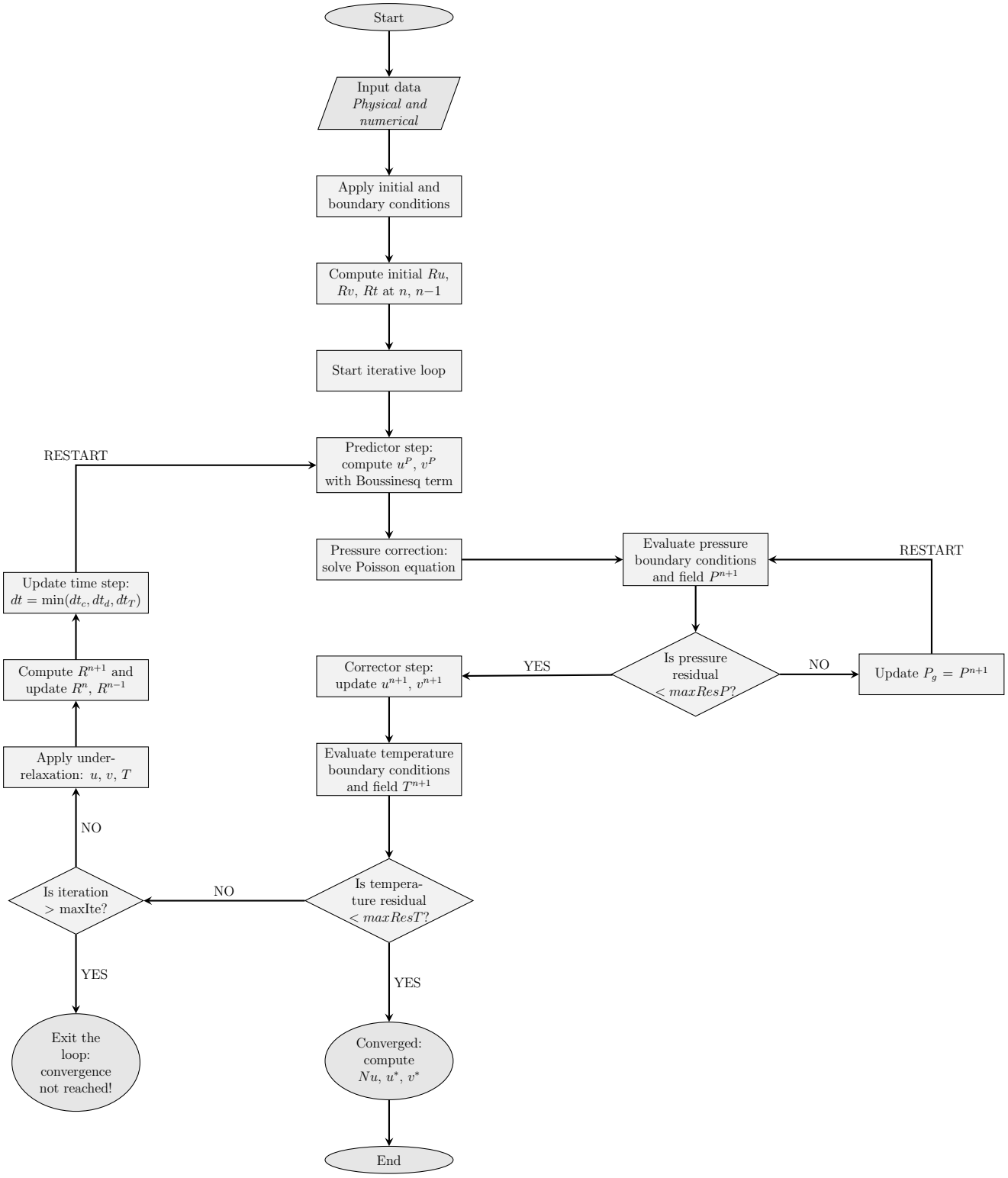


Figure 7: Algorithmic flowchart for the DHC numerical simulation.

3 Results and Discussion

Results are provided for $Pr = 0.71$ and Rayleigh numbers $Ra = 10^3$, 10^4, 10^5, and 10^6. For each case, the velocity field, temperature distribution, and Nusselt number are reported and compared to reference data.

3.1 Preliminary Considerations

The results presented in this section were obtained using an adaptive convective scheme: the simulation initially employs an upwind differencing scheme (UDS) during the first iterations to ensure numerical stability and allow the transient to evolve. Once the temperature residual drops below a predefined threshold and its relative variation becomes sufficiently small, the solver automatically switches to a central differencing scheme (CDS) to improve the accuracy of the solution.

Moreover, the steady-state condition was considered to be reached once the temperature residual dropped below a predefined tolerance of `maxResT < 1e-4`. Initially, a stricter threshold of 10^{-6} was adopted for all residuals, including velocity and temperature. However, it was soon observed that such a restrictive criterion often resulted in poor convergence of the velocity fields, with some simulations exhibiting unphysical oscillations or stagnation of the residuals, especially at higher Rayleigh numbers.
Therefore, the temperature tolerance was relaxed to 10^{-4}, which proved to be sufficiently accurate for capturing thermal behavior. The final convergence check was performed by verifying that all three residuals—horizontal velocity, vertical velocity, and temperature—remained below their respective tolerances. This approach ensured both physical consistency and numerical stability, leading to solutions that closely matched benchmark results.

Lastly, to further accelerate convergence, the following over- and under-relaxation factors were applied throughout the simulations:

Relaxation Factor Type	**Value**
Over-relaxation for pressure (Gauss–Seidel)	1.3
Under-relaxation for velocity u	0.5
Under-relaxation for velocity v	0.5
Under-relaxation for temperature T	0.5

Table 1: Over- and under-relaxation factors used in the simulations.

These values were selected according to standard practices in simulations involving structured staggered grids and iterative solvers. Over-relaxation factors slightly above 1 are known to accelerate convergence of Gauss–Seidel solvers when solving pressure correction equations such as Poisson, while under-relaxation values between 0.3 and 0.7 are typically used for velocity and temperature to improve numerical stability.[2]

[2]See: J. H. Ferziger and M. Perić, *Computational Methods for Fluid Dynamics*, 3rd ed., Springer, 2002.

3.2 Comparison with Benchmark Results

In this section, the results obtained are compared with the benchmark solutions provided by G. de Vahl Davis[3].
All simulations were performed using a uniform grid of **21×21** control volumes for $Ra = 10^3$ and 10^4, and **41×41** control volumes for $Ra = 10^5$ and 10^6. This choice was made to ensure consistency with the benchmark results presented by *G. de Vahl Davis* and to adequately capture the steep thermal and velocity gradients that develop at higher Rayleigh numbers. The reference study includes a detailed grid convergence analysis, using multiple mesh refinements and Richardson extrapolation to determine grid-independent benchmark results with an estimated accuracy better than 1%. These validated results are used here for comparison.

Along with the average Nusselt number, the relative error (RE) is also presented to compare the accuracy of the results. It is computed as the ratio between the absolute error and the reference value and is expressed as a percentage:

$$RE = \left| \frac{\overline{Nu}_{\text{reference}} - \overline{Nu}_{\text{result}}}{\overline{Nu}_{\text{reference}}} \right| \times 100\%$$

Ra	Reference $\overline{Nu}$	Result $\overline{Nu}$	RE (%)
10^3	1.118	1.13981	1.95
10^4	2.243	2.15965	3.72
10^5	4.519	4.37167	3.26
10^6	8.800	8.18352	7.01

Table 2: Relative error between reference and computed $\overline{Nu}$ values (21×21 for $Ra = 10^3, 10^4$, 41×41 for $Ra = 10^5, 10^6$).

From the results presented in Table 2, we can observe how both the Rayleigh number and the mesh resolution affect the accuracy of the computed average Nusselt number.
For $Ra = 10^3$ and 10^4, which are both computed using a 21×21 mesh, the relative error increases from 1.95% to 3.72%, showing how higher Rayleigh numbers introduce greater numerical discrepancies on the same mesh. For $Ra = 10^5$ and 10^6, the use of a finer 41×41 mesh results in better accuracy compared to coarser grids, but the relative error still increases to 7.01% at $Ra = 10^6$.
This indicates that as the Rayleigh number grows and convection becomes stronger, finer meshes are needed to accurately capture the steeper thermal gradients that develop near the cavity boundaries.

[3]G. de Vahl Davis, "Natural Convection of Air in a Square Cavity: A Bench Mark Numerical Solution", *Int. J. Numer. Meth. Fluids*, 3, 249–264 (1983).

Ra	Max u^* velocity (at $x = 0.5L$)			Position y/L of $u^*_{\max}$		
	Ref.	Res.	RE (%)	Ref.	Res.	RE (%)
10^3	3.649	3.33483	8.61	0.813	0.795	2.21
10^4	16.178	15.32840	5.25	0.823	0.795	3.40
10^5	34.730	33.09120	4.72	0.855	0.845	1.17
10^6	64.630	60.86300	5.83	0.850	0.845	0.59

Table 3: Relative error between reference and result for max horizontal velocity $u^*_{\max}$ and its vertical position at $x = 0.5L$ (21×21 for $Ra = 10^3, 10^4$, 41×41 for $Ra = 10^5, 10^6$).

Ra	Max v^* velocity (at $y = 0.5L$)			Position x/L of $v^*_{\max}$		
	Ref.	Res.	RE (%)	Ref.	Res.	RE (%)
10^3	3.697	3.37471	8.72	0.178	0.205	15.17
10^4	19.617	18.47490	5.82	0.119	0.159	33.61
10^5	68.590	66.99370	2.33	0.066	0.083	25.76
10^6	219.360	204.52600	6.76	0.0379	0.060	58.31

Table 4: Relative error between reference and result for max vertical velocity $v^*_{\max}$ and its horizontal position at $y = 0.5L$ (21×21 for $Ra = 10^3, 10^4$, 41×41 for $Ra = 10^5, 10^6$).

Tables 3 and 4 show a detailed comparison between the computed and reference values for the maximum dimensionless velocities (u^* and v^*) and their positions inside the cavity.
In Table 3, the values of $u^*_{\max}$ are predicted with good accuracy, with relative errors always below 9%. The vertical position (y/L) of this peak is also well captured, with small errors under 3.5%. This confirms that the horizontal velocity component is well predicted, especially when the mesh is sufficiently refined.
On the other hand, Table 4 shows that although the computed values of $v^*_{\max}$ are close to the reference (with errors under 7%), the prediction of its horizontal position (x/L) is much less accurate. The error becomes especially large at high Rayleigh numbers, reaching over 58% for $Ra = 10^6$. This is likely due to the fact that vertical velocity peaks occur in narrower regions, which are harder to resolve unless a very fine grid is used.
Overall, the results show that while the magnitude of the velocity peaks is predicted quite well, the exact location of these peaks—especially for the vertical velocity—requires finer grids to avoid large errors, particularly at high Rayleigh numbers where the flow becomes more complex.

3.3 Grid Independence Study

Although the benchmark study by de Vahl Davis provides validated grid-independent solutions for each Rayleigh number, an independent grid convergence analysis was carried out to assess the robustness of the present implementation and verify that the computed results are not significantly affected by the chosen spatial resolution.
For each Rayleigh number tested ($Ra = 10^3, 10^4, 10^5, 10^6$), simulations were performed using four progressively refined uniform grids: 21×21, 31×31, 41×41, and 51×51. The

average Nusselt number $\overline{Nu}$, the maximum horizontal velocity $u_{\max}$, and the maximum vertical velocity $v_{\max}$ were monitored.

Table 5: Grid data for $Ra = 10^3$.

Grid	$\overline{Nu}$	$u_{\max}$	$v_{\max}$
21×21	1.13981	3.33483	3.37471
31×31	1.13233	3.43568	3.46992
41×41	1.12839	3.48021	3.52910
51×51	1.12587	3.50169	3.55500

Table 6: Grid data for $Ra = 10^4$.

Grid	$\overline{Nu}$	$u_{\max}$	$v_{\max}$
21×21	2.15965	15.32840	18.47490
31×31	2.19354	15.59510	19.07850
41×41	2.21031	15.77050	19.28470
51×51	2.21872	15.87800	19.36150

Table 7: Grid data for $Ra = 10^5$.

Grid	$\overline{Nu}$	$u_{\max}$	$v_{\max}$
21×21	4.12213	31.18900	62.86220
31×31	4.29788	32.43010	67.15470
41×41	4.37167	33.09120	66.99370
51×51	4.41107	33.40630	67.26190

Table 8: Grid data for $Ra = 10^6$.

Grid	$\overline{Nu}$	$u_{\max}$	$v_{\max}$
21×21	6.93599	55.65840	205.95700
31×31	7.81209	58.39870	212.38000
41×41	8.18352	60.86300	204.52600
51×51	8.37582	61.86640	216.94700

To evaluate convergence, the relative error of each quantity was computed with respect to the values obtained on the finest grid (51×51), according to the following expression:

$$\varepsilon_{\text{rel}} = \left| \frac{\phi_h - \phi_{51}}{\phi_{51}} \right|$$

where ϕ_h denotes the value computed on a coarser grid.
The convergence trends are illustrated in the following figure, where the relative errors are plotted against the total number of control volumes in each mesh for all Rayleigh numbers considered.

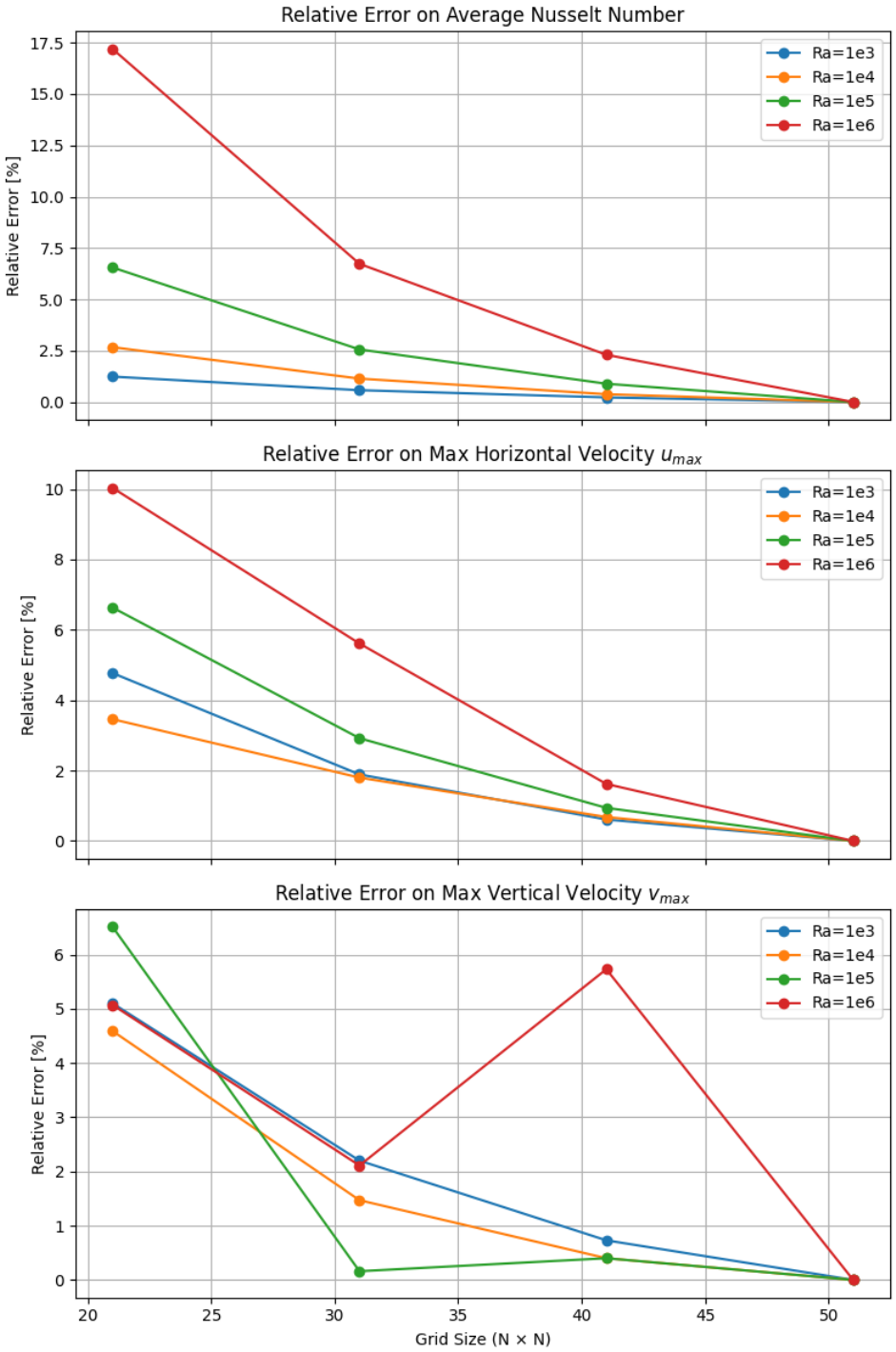


Figure 8: Relative error on $\overline{Nu}$, $u_{\max}$, and $v_{\max}$ with respect to the 51×51 grid, as a function of grid refinement, for different Rayleigh numbers.

Figure 8 shows the relative errors on the average Nusselt number, maximum horizontal velocity $u^*_{\max}$, and maximum vertical velocity $v^*_{\max}$ as the mesh is refined, for different Rayleigh numbers.
For $Ra = 10^3$ and 10^4, the error on the Nusselt number is already below 2% with the coarser 21×21 grid. This means that, in these cases, a coarse mesh is accurate enough and using a finer one would only increase computation time without much benefit. For $Ra = 10^5$ and 10^6, a finer 41×41 grid is necessary to reduce the error below 2%, since

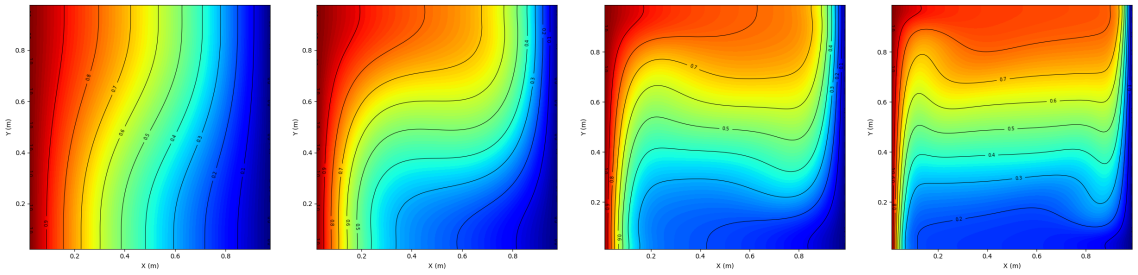
the flow and temperature gradients are more intense.
The behavior of $u^*_{\max}$ is similar: errors drop consistently with grid refinement and become small from the 41×41 grid onward.
For $v^*_{\max}$, however, the error does not decrease as smoothly, especially for $Ra = 10^6$, where the error temporarily increases at 41×41. This is likely caused by strong vertical jets that are difficult to resolve and sensitive to small changes in the grid.
Overall, we can say that:

- 21×21 is sufficient for $Ra = 10^3$ and 10^4, although a 31×31 grid might be preferred to better capture velocity peaks.
- 41×41 is a good choice for $Ra = 10^5$ and 10^6.

This strategy balances accuracy and computational time, which becomes very important at higher Rayleigh numbers.

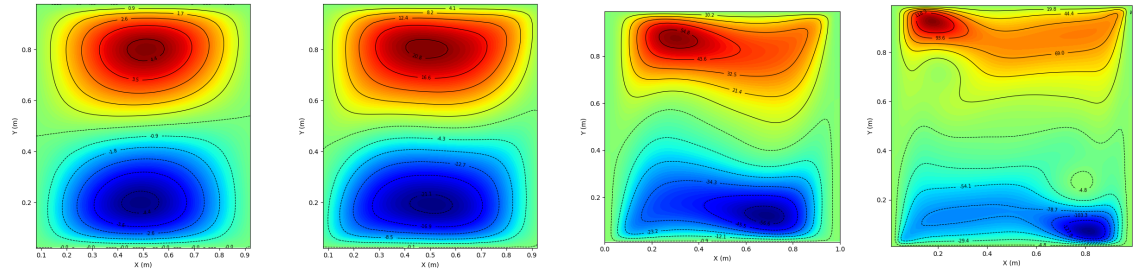
3.4 Colormaps and Profiles

To better understand the physical phenomena occurring in the differentially heated cavity, several plots are presented below.



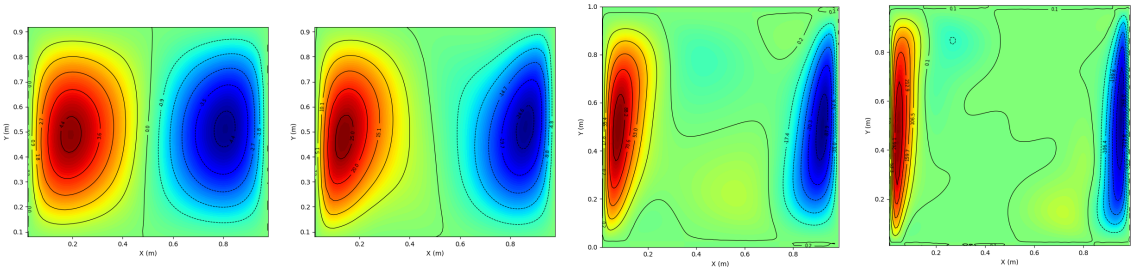
(a) $Ra = 10^3$ (b) $Ra = 10^4$ (c) $Ra = 10^5$ (d) $Ra = 10^6$

Figure 9: Temperature maps for different Rayleigh numbers



(a) $Ra = 10^3$ (b) $Ra = 10^4$ (c) $Ra = 10^5$ (d) $Ra = 10^6$

Figure 10: u velocity maps for different Rayleigh numbers



(a) $Ra = 10^3$ (b) $Ra = 10^4$ (c) $Ra = 10^5$ (d) $Ra = 10^6$

Figure 11: v velocity maps for different Rayleigh numbers

The colormaps presented in Figures 9, 10, and 11 illustrate the evolution of the temperature field, the horizontal velocity component u, and the vertical velocity component v, respectively, as the Rayleigh number increases from 10^3 to 10^6.
In the temperature maps (Figure 9), we observe a clear transition from a conduction-dominated regime at low Rayleigh numbers to convection-dominated behavior at higher Ra. At $Ra = 10^3$, the isotherms are almost parallel and indicate a linear temperature gradient. As Ra increases, the isotherms start to bend and curve, showing the formation of thermal currents caused by the natural convection.
The horizontal velocity maps (Figure 10) highlight the formation of a clockwise convective regime. The fluid moves to the right (positive u) in the upper part of the cavity and to the left (negative u) near the bottom, forming a coherent circulation pattern. This behavior is further confirmed by the u velocity profiles along the vertical centerline (Figure 13, top row), where the velocity transitions from positive to negative, indicating the opposing directions of motion at the top and bottom of the cavity.
In the vertical velocity maps (Figure 11), the same convective structure is evident: fluid rises along the hot left wall (positive v values) and descends near the cold right wall (negative v values). The vertical velocity profiles extracted at mid-height (Figure 13, bottom row) clearly confirm this upward and downward movement. The profiles were extracted along the lines indicated in Figure 12.

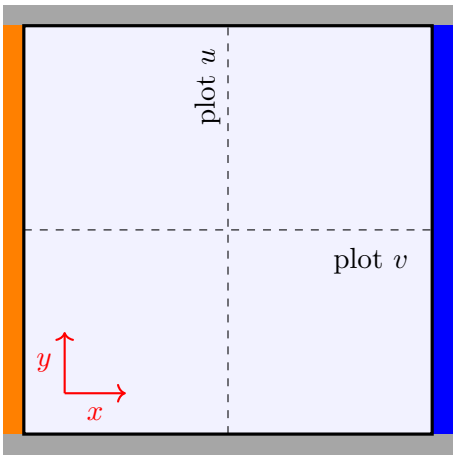
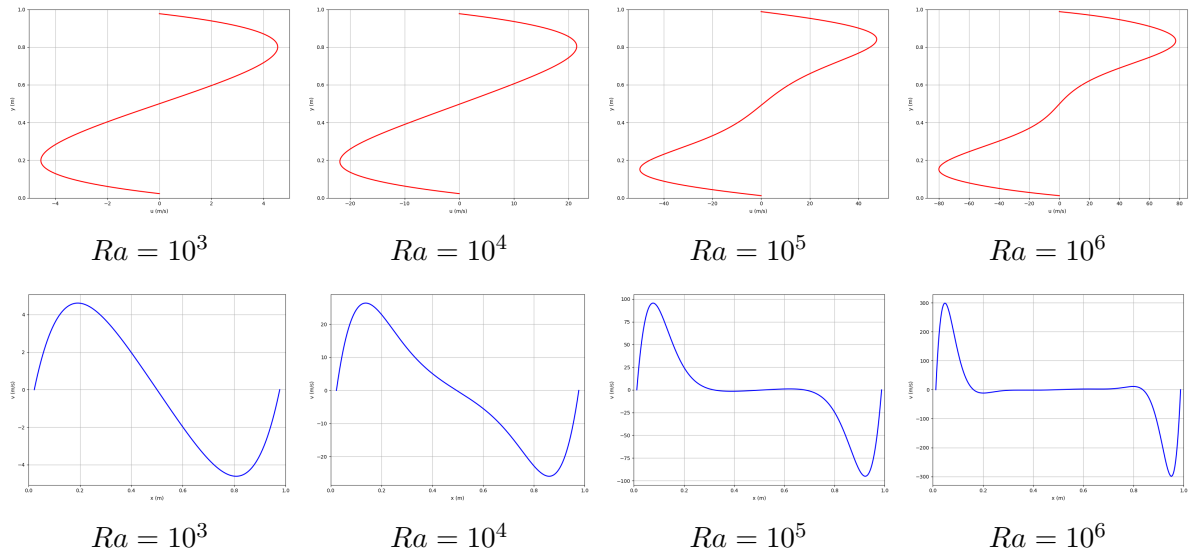


Figure 12: Velocity profiles: $u(y)$ at mid-width and $v(x)$ at mid-height.



$Ra = 10^3$ $Ra = 10^4$ $Ra = 10^5$ $Ra = 10^6$

$Ra = 10^3$ $Ra = 10^4$ $Ra = 10^5$ $Ra = 10^6$

Figure 13: Velocity profiles at mid-sections. Top row: $u(y)$ at $x = 0.5L$. Bottom row: $v(x)$ at $y = 0.5L$.

This flow pattern is a typical representation of natural convection in a square cavity with differentially heated vertical walls. The fluid adjacent to the hot left wall absorbs thermal energy, causing its temperature to increase. As a result, its density decreases and the fluid rises. On the opposite side, near the cold right wall, the fluid loses heat, becomes denser, and descends. As Ra increases, both thermal and flow fields exhibit sharper variations, indicating a stronger influence of convection over conduction. This effect is particularly evident at $Ra = 10^6$, where the flow shows sharp transitions and more complex structures, which require fine spatial resolution to be captured accurately.

4 Conclusions

The Differentially Heated Cavity (DHC) problem was solved using a finite volume-based fractional step solver with staggered meshes. The main extension with respect to the lid-driven cavity was the inclusion of the energy equation and the coupling of temperature and velocity fields via the Boussinesq approximation.
The computed results showed good agreement with reference data, capturing the expected flow patterns and their evolution with increasing Rayleigh number. The use of refined grids improved the accuracy of the predicted quantities, particularly for higher Ra.
The code was able to reproduce the main characteristics of natural convection, such as the circulating flow, the temperature distribution, and the expected values of Nusselt number and peak velocities. These results confirm the robustness of the solver and provide a reliable basis for future developments.

A C++ Code

```
#include "Differentially_Heated_Cavity.h"
#include<iostream>
#include <fstream>
#include <chrono>
#include <cmath>
#include <vector>
#include <algorithm>
#include <ranges>

using namespace std;
using namespace std::chrono;

// Function to allocate matrices
vector<vector<double>> CreateMatrix(int rows, int cols, double init_val = 0.0) {
    return vector(rows, vector(cols, init_val));
}

// Base Class to implement multiple methods
class ConvectiveScheme {
public:
    virtual void ComputeRt(double rho, int Ny, int Nx, double dx, double lambda, double cp, vector<
        vector<double> > &u,
                           vector<vector<double> > &v, vector<vector<double> > &Rt,
                           vector<vector<double> > &T) = 0; // Pure virtual function

    virtual void ComputeRu(vector<vector<double> > &u, vector<vector<double> > &v, vector<vector<
        double> > &Ru,
                           double rho, double dx, double mu, int Nx, int Ny) = 0;

    virtual void ComputeRv(vector<vector<double> > &u, vector<vector<double> > &v, vector<vector<
        double> > &Rv,
                           double rho, double dx, double mu, int Nx, int Ny) = 0;

    virtual void Nusselt(int Nx, int Ny, double Tcold, double Thot, double L, double alpha, double dx,
         double &Nux_avg,
                         vector<vector<double> > &T1star, vector<vector<double> > &T1, vector<vector<
                             double> > &qx,
                         vector<vector<double> > &u1star, vector<vector<double> > &u1, vector<vector<
                             double> > &dTdx,
                         vector<double> &Nux, double dxstar) = 0;

    virtual ~ConvectiveScheme() = default; // Virtual destructor
};


// Central Differencing Scheme (CDS) method
class CDSMethod : public ConvectiveScheme {

public:
    // Function to compute R(t) on each node of stagg-T mesh (same as stagg-P)
    void ComputeRt(double rho, int Ny, int Nx, double dx, double lambda, double cp, vector<vector<
        double> > &u,
                   vector<vector<double> > &v, vector<vector<double> > &Rt, vector<vector<double> > &T
                       ) override {
        for (int i = 1; i < Ny-1; i++) {
            for (int j = 1; j < Nx-1; j++) {
                Rt[i][j] = -rho * dx * cp * (u[i][j + 1] * 1.0 / 2 * (T[i][j] + T[i][j + 1]) - u[i][j]
                    * 1.0 / 2 * (
                                                 T[i][j] + T[i][j - 1]) + v[i][j] * 1.0 / 2 * (T[i][j] + T[
                                                     i - 1][j]) - v[
                                                 i + 1][j] * 1.0 / 2 * (T[i][j] + T[i + 1][j])) + lambda *
                                                     (
                                    T[i][j + 1] + T[i][j - 1] + T[i - 1][j] + T[i + 1][j] - 4 * T[i][j]);
            }
        }
    }

    // Function to compute R(u) on internal nodes of stagg-x mesh
    void ComputeRu(vector<vector<double> > &u, vector<vector<double> > &v, vector<vector<double> > &Ru
        , double rho,
                   double dx, double mu, int Nx, int Ny) override {
        for (int i = 1; i < Ny - 1; i++) {
            for (int j = 1; j < Nx; j++) {
                Ru[i][j] = -rho * dx * (1.0 / 2 * (v[i][j - 1] + v[i][j]) * 1.0 / 2 * (u[i][j] + u[i -
                    1][j])
                                        - 1.0 / 2 * (v[i + 1][j - 1] + v[i + 1][j]) * 1.0 / 2 * (u[i][
                                            j] + u[i + 1][j])
                                        + 1.0 / 2 * (u[i][j] + u[i][j + 1]) * 1.0 / 2 * (u[i][j] + u[i
                                            ][j + 1])
                                        - 1.0 / 2 * (u[i][j] + u[i][j - 1]) * 1.0 / 2 * (u[i][j] + u[i
                                            ][j - 1]))
                                   + mu * (u[i - 1][j] + u[i + 1][j] + u[i][j - 1] + u[i][j + 1] - 4 * u[i][j
                                       ]);
            }
        }
    }

    // Function to compute R(v) on internal nodes of stagg-y mesh
    void ComputeRv(vector<vector<double> > &u, vector<vector<double> > &v, vector<vector<double> > &Rv
        , double rho,
                   double dx, double mu, int Nx, int Ny) override {
        for (int i = 1; i < Ny; i++) {
            for (int j = 1; j < Nx - 1; j++) {
                Rv[i][j] = -rho * dx * (1.0 / 2 * (v[i][j] + v[i - 1][j]) * 1.0 / 2 * (v[i][j] + v[i -
                    1][j])
```

```
                                        - 1.0 / 2 * (v[i][j] + v[i + 1][j]) * 1.0 / 2 * (v[i][j] + v[i
                                            + 1][j])
                                        + 1.0 / 2 * (u[i - 1][j + 1] + u[i][j + 1]) * 1.0 / 2 * (v[i][
                                            j] + v[i][j + 1])
                                        - 1.0 / 2 * (u[i - 1][j] + u[i][j]) * 1.0 / 2 * (v[i][j] + v[i
                                            ][j - 1]))
                                + mu * (v[i - 1][j] + v[i + 1][j] + v[i][j - 1] + v[i][j + 1] - 4 * v[i][j
                                    ]);
            }
        }
    }

    void Nusselt(int Nx, int Ny, double Tcold, double Thot, double L, double alpha, double dx, double
        &Nux_avg,
                 vector<vector<double> > &T1star, vector<vector<double> > &T1, vector<vector<double> >
                     &qx,
                 vector<vector<double> > &u1star, vector<vector<double> > &u1, vector<vector<double> >
                     &dTdx,
                 vector<double> &Nux, double dxstar) override {

        // Normalize distance, temperature and horizontal velocity
        dxstar = dx / L;
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx; j++) {
                T1star[i][j] = (T1[i][j] - Tcold) / (Thot - Tcold);
            }
        }
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx + 1; j++) {
                u1star[i][j] = u1[i][j] * L / alpha;
            }
        }
        // Hot wall (x = 0) - dT/dx
        for (int i = 0; i < Ny; ++i) {
            dTdx[i][0] = (T1star[i][1] - T1star[i][0]) / dxstar;
            double T_central = 0.5 * (T1star[i][1] + T1star[i][0]);
            qx[i][0] = u1star[i][0] * T_central - dTdx[i][0];
        }
        // Internal cells (CDS)
        for (int i = 0; i < Ny; ++i) {
            for (int j = 1; j < Nx; ++j) {
                dTdx[i][j] = (T1star[i][j] - T1star[i][j - 1]) / dxstar;
                double T_central = 0.5 * (T1star[i][j] + T1star[i][j - 1]);
                qx[i][j] = u1star[i][j] * T_central - dTdx[i][j];
            }
        }
        // Vertical integration
        for (int j = 0; j < Nx; ++j) {
            double sum = 0.0;
            for (int i = 0; i < Ny; ++i) {
                sum += qx[i][j] * dxstar;
            }
            Nux[j] = sum;
        }
        // Average Nusselt
        Nux_avg = 0.0;
        for (int j = 0; j < Nx; ++j) {
            Nux_avg += Nux[j];
        }
        Nux_avg /= Nx;
    }
};

// Upwind Differencing Scheme (UDS) method
class UDSMethod : public ConvectiveScheme {

public:
    // Function to compute R(t) on each node of stagg-T mesh (same as stagg-P)
    void ComputeRt(double rho, int Ny, int Nx, double dx, double lambda, double cp, vector<vector<
        double> > &u,
                   vector<vector<double> > &v, vector<vector<double> > &Rt, vector<vector<double> > &T
                       ) override {
        for (int i = 1; i < Ny-1; i++) {
            for (int j = 1; j < Nx-1; j++) {
                double convW = u[i][j] > 0 ? u[i][j] * T[i][j - 1] : u[i][j] * T[i][j];

                double convE = u[i][j + 1] > 0 ? u[i][j + 1] * T[i][j] : u[i][j + 1] * T[i][j + 1];

                double convS = v[i + 1][j] > 0 ? v[i + 1][j] * T[i + 1][j] : v[i + 1][j] * T[i][j];

                double convN = v[i][j] > 0 ? v[i][j] * T[i][j] : v[i][j] * T[i - 1][j];

                Rt[i][j] = -rho * dx * cp * (convE - convW + convN - convS) +
                           lambda * (T[i][j + 1] + T[i][j - 1] + T[i - 1][j] + T[i + 1][j] - 4 * T[i][
                               j]);
            }
        }
    }

    // Function to compute R(u) on internal nodes of stagg-x mesh
    void ComputeRu(vector<vector<double> > &u, vector<vector<double> > &v,
                   vector<vector<double> > &Ru, double rho, double dx, double mu, int Nx, int Ny)
                       override {
        for (int i = 1; i < Ny - 1; i++) {
            for (int j = 1; j < Nx; j++) {
                double v_n = (v[i][j - 1] + v[i][j]) / 2;
                double conv_n = v_n > 0 ? v_n * u[i][j] : v_n * u[i - 1][j];

                double v_s = (v[i + 1][j - 1] + v[i + 1][j]) / 2;
```

```
                double conv_s = v_s > 0 ? v_s * u[i + 1][j] : v_s * u[i][j];

                double conv_w = u[i][j] > 0 ? u[i][j] * u[i][j - 1] : u[i][j] * u[i][j];

                double conv_e = u[i][j + 1] > 0 ? u[i][j + 1] * u[i][j] : u[i][j + 1] * u[i][j + 1];

                Ru[i][j] = -rho * dx * (conv_e - conv_w + conv_n - conv_s)
                           + mu * (u[i - 1][j] + u[i + 1][j] + u[i][j - 1] + u[i][j + 1] - 4 * u[i][j
                               ]);
            }
        }
    }

    //Function to compute R(v) on internal nodes of stagg-y mesh
    void ComputeRv(vector<vector<double> > &u, vector<vector<double> > &v, vector<vector<double> > &Rv
        , double rho,
                   double dx, double mu, int Nx, int Ny) override {
        for (int i = 1; i < Ny; i++) {
            for (int j = 1; j < Nx - 1; j++) {
                double conv_n = v[i][j] > 0 ? v[i][j] * v[i - 1][j] : v[i][j] * v[i][j];

                double conv_s = v[i + 1][j] > 0 ? v[i + 1][j] * v[i][j] : v[i + 1][j] * v[i + 1][j];

                double u_e = 0.5 * (u[i - 1][j + 1] + u[i][j + 1]);
                double conv_e = u_e > 0 ? u_e * v[i][j] : u_e * v[i][j + 1];

                double u_w = 0.5 * (u[i - 1][j] + u[i][j]);
                double conv_w = u_w > 0 ? u_w * v[i][j - 1] : u_w * v[i][j];

                Rv[i][j] = -rho * dx * (conv_e - conv_w + conv_n - conv_s)
                           + mu * (v[i - 1][j] + v[i + 1][j] + v[i][j - 1] + v[i][j + 1] - 4 * v[i][j
                               ]);
            }
        }
    }

    // Function to evaluate average Nusselt number
    void Nusselt(int Nx, int Ny, double Tcold, double Thot, double L, double alpha, double dx, double
        &Nux_avg,
                 vector<vector<double> > &T1star, vector<vector<double> > &T1, vector<vector<double> >
                     &qx,
                 vector<vector<double> > &u1star, vector<vector<double> > &u1, vector<vector<double> >
                     &dTdx,
                 vector<double> &Nux, double dxstar) override {
        // Normalize distance, temperature and horizontal velocity
        dxstar = dx / L;
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx; j++) {
                T1star[i][j] = (T1[i][j] - Tcold) / (Thot - Tcold);
            }
        }
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx + 1; j++) {
                u1star[i][j] = u1[i][j] * L / alpha;
            }
        }
        // Heat flux (UDS)
        for (int i = 0; i < Ny; ++i) {
            // (x = 0)
            dTdx[i][0] = (T1star[i][1] - T1star[i][0]) / dxstar;
            double T_upwind_0 = (u1star[i][0] > 0.0) ? T1star[i][0] : T1star[i][1];
            qx[i][0] = u1star[i][0] * T_upwind_0 - dTdx[i][0];

            // (j = 1)
            dTdx[i][1] = (T1star[i][1] - T1star[i][0]) / dxstar;
            double T_upwind_1 = (u1star[i][1] > 0.0) ? T1star[i][0] : T1star[i][1];
            qx[i][1] = u1star[i][1] * T_upwind_1 - dTdx[i][1];

            // Internals (j >= 2)
            for (int j = 2; j < Nx; ++j) {
                if (u1star[i][j] > 0.0) {
                    dTdx[i][j] = (T1star[i][j - 1] - T1star[i][j - 2]) / dxstar;
                } else {
                    dTdx[i][j] = (T1star[i][j] - T1star[i][j - 1]) / dxstar;
                }
                double T_upwind = (u1star[i][j] > 0.0) ? T1star[i][j - 1] : T1star[i][j];
                qx[i][j] = u1star[i][j] * T_upwind - dTdx[i][j];
            }
        }
        // Vertical integration
        for (int j = 0; j < Nx; ++j) {
            double sum = 0.0;
            for (int i = 0; i < Ny; ++i) {
                sum += qx[i][j] * dxstar;
            }
            Nux[j] = sum;
        }
        // Average Nusselt
        Nux_avg = 0.0;
        for (int j = 0; j < Nx; ++j) {
            Nux_avg += Nux[j];
        }
        Nux_avg /= Nx;
    }
};

// Function to choose the method
ConvectiveScheme* createCalculator(const string &method) {
    if (method == "CDS") {
```

```
        return new CDSMethod();
    } else if (method == "UDS") {
        return new UDSMethod();
    } else {
        cerr << "Not a valid method!" << endl;
        return nullptr;
    }
}

// Function to apply boundary conditions
void BoundaryConditions(int Nx, int Ny, double Thot, double Tcold, vector<vector<double> > &Tn_1,
                        vector<vector<double> > &Tn, vector<vector<double> > &T1) {
    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx; j++) {
            // LEFT WALL (x = 0): Dirichlet BC
            if (j == 0) {
                Tn_1[i][j] = Thot;
                Tn[i][j] = Thot;
                T1[i][j] = Thot;
            }
            // RIGHT WALL (x = L): Dirichlet BC
            else if (j == Nx - 1) {
                Tn_1[i][j] = Tcold;
                Tn[i][j] = Tcold;
                T1[i][j] = Tcold;
            }
        }
    }
    // TOP WALL (y = L -> i = 0): Neumann dT/dz = 0 -> T[0][j] = T[1][j]
    for (int j = 0; j < Nx; j++) {
        Tn_1[0][j] = Tn_1[1][j];
        Tn[0][j] = Tn[1][j];
        T1[0][j] = T1[1][j];
    }
    // BOTTOM WALL (y = 0 -> i = Ny - 1): Neumann dT/dz = 0 -> T[Ny-1][j] = T[Ny-2][j]
    for (int j = 0; j < Nx; j++) {
        Tn_1[Ny - 1][j] = Tn_1[Ny - 2][j];
        Tn[Ny - 1][j] = Tn[Ny - 2][j];
        T1[Ny - 1][j] = T1[Ny - 2][j];
    }
}

// Function to solve the pressure field with Gauss-Seidel solver on all nodes of stagg-P mesh
void PoissonSolver(double maxResP, double maxIteP, double rho, double dx, double dt, int Nx, int Ny,
     double omega_P,
                    vector<vector<double> > &P1, vector<vector<double> > &vP, vector<vector<double> > &
                        uP,
                    vector<vector<double> > &Pg) {
    double resP = maxResP + 1;
    int iteP = 0;
    while (resP > maxResP && iteP < maxIteP) {
        double maxDiffP = 0.0;
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx; j++) {
                if (i == 0)
                    P1[i][j] = 1.0 / 1 * (P1[i+1][j] - rho * dx / dt * (vP[i][j] - vP[i+1][j] + uP[i][
                        j+1] - uP[i][j]));
                else if (j == 0)
                    P1[i][j] = 1.0 / 1 * (P1[i][j+1] - rho * dx / dt * (vP[i][j] - vP[i+1][j] + uP[i][
                        j+1] - uP[i][j]));
                else if (i == Ny - 1)
                    P1[i][j] = 1.0 / 1 * (P1[i-1][j] - rho * dx / dt * (vP[i][j] - vP[i+1][j] + uP[i][
                        j+1] - uP[i][j]));
                else if (j == Nx - 1)
                    P1[i][j] = 1.0 / 1 * (P1[i][j-1] - rho * dx / dt * (vP[i][j] - vP[i+1][j] + uP[i][
                        j+1] - uP[i][j]));
                else
                    P1[i][j] = 1.0 / 4 * (P1[i+1][j] + P1[i][j+1] + P1[i-1][j] + P1[i][j-1]
                        - rho * dx / dt * (vP[i][j] - vP[i+1][j] + uP[i][j+1] - uP[i][j]));
                double diffP = fabs(P1[i][j] - Pg[i][j]);
                if (diffP > maxDiffP) {
                    maxDiffP = diffP;
                }
            }
        }
        resP = maxDiffP;
        // Update P guess with over-relaxation
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx; j++) {
                if (omega_P < 1e-8) {
                    Pg[i][j] = P1[i][j];
                } else {
                    Pg[i][j] = Pg[i][j] + omega_P * (P1[i][j] - Pg[i][j]);
                }
            }
        }
        iteP++;
    }
}

int main () {
    // Start timer
    auto start = high_resolution_clock::now();

    // Physical data
    int N;
    cout << "Insert number of control volumes:-";                                    // Ask for mesh
         refinement
    cin >> N;
```

```
int Nx = N;
int Ny = N;
double Ra;
double L = 1.0;
double dx = L / Nx;
double Pr = 0.71;
double Thot = 1.0;
double Tcold = 0.0;
double Tinf = 0.5;
double cp = 0.71;
double lambda = 1.0;
double mu = Pr * lambda / cp;
double rho = 1.0;
double beta = 1.0;
cout << "Insert Rayleigh number:"; // Ask for desired
    Rayleigh number
cin >> Ra;
double g = Ra * mu * lambda / (cp * beta * pow(rho, 2) * (Thot - Tcold) * pow(L, 3));
double alpha = lambda / (rho * cp);
double Nux_avg = 0.0;
double Vmax;
double dxstar;

// Numerical data
double maxResP = 1e-6;
double maxIte = 1e6;
double maxResT = 1e-4;
double maxResU, maxResV = maxResT;
double t_count = 0.0;
double res1 = maxResU + 1;
double res2 = maxResV + 1;
double res3 = maxResT + 1;
double dt = 1e-4;
double dtc, dtd, dtt;
int ite = 0;
string method;
string graph;
double omega_T = 0.5; // Relaxation factors
double omega_u = 0.5;
double omega_v = 0.5;
double omega_P = 1.3;

cout << "Choose the method ('CDS', 'UDS', or 'AUTO'):"; // Ask for convective
    scheme
cin >> method;
ranges::transform(method, method.begin(), ::toupper);

ConvectiveScheme* calculator = nullptr;
ConvectiveScheme* calculator_UDS = nullptr;
ConvectiveScheme* calculator_CDS = nullptr;

bool switched = false;
double res3_old = res3;

if (method == "CDS" || method == "UDS") {
    calculator = createCalculator(method);
    if (!calculator) {
        cerr << "Error: Invalid method selected." << endl;
        return 1;
    }
    cout << "Running with fixed " << method << " scheme.\n";
} else if (method == "AUTO") {
    calculator_UDS = createCalculator("UDS");
    calculator_CDS = createCalculator("CDS");
    if (!calculator_UDS || !calculator_CDS) {
        cerr << "Error: Could not create convection schemes." << endl;
        return 1;
    }
    calculator = calculator_UDS; // start with UDS
    cout << "AUTO mode: starting with UDS, will switch to CDS based on residuals.\n";
} else {
    cerr << "Error: Method must be 'CDS', 'UDS', or 'AUTO'." << endl;
    return 1;
}

// Pressure, Temperature, u and v matrices
auto P1    = CreateMatrix(Ny, Nx);
auto Pg    = CreateMatrix(Ny, Nx);
auto Tn_1 = CreateMatrix(Ny, Nx);
auto Tn = CreateMatrix(Ny, Nx);
auto T1 = CreateMatrix(Ny, Nx);
auto un_1  = CreateMatrix(Ny, Nx + 1);
auto un    = CreateMatrix(Ny, Nx + 1);
auto u1    = CreateMatrix(Ny, Nx + 1);
auto uP    = CreateMatrix(Ny, Nx + 1);
auto vn_1  = CreateMatrix(Ny + 1, Nx);
auto vn    = CreateMatrix(Ny + 1, Nx);
auto v1    = CreateMatrix(Ny + 1, Nx);
auto vP    = CreateMatrix(Ny + 1, Nx);

// Convective-diffusive, Buoyancy term R() matrices
auto Ru1    = CreateMatrix(Ny, Nx + 1);
auto Run    = CreateMatrix(Ny, Nx + 1);
auto Run_1  = CreateMatrix(Ny, Nx + 1);
auto Rv1    = CreateMatrix(Ny + 1, Nx);
auto Rvn    = CreateMatrix(Ny + 1, Nx);
auto Rvn_1  = CreateMatrix(Ny + 1, Nx);
auto Rt1 = CreateMatrix(Ny, Nx);
auto Rtn = CreateMatrix(Ny, Nx);
```

```
auto Rtn_1 = CreateMatrix(Ny, Nx);
auto Rbn = CreateMatrix(Ny + 1, Nx);
auto Rbn_1 = CreateMatrix(Ny + 1, Nx);

// Adimensional analysis matrices and vectors
auto T1star = CreateMatrix(Ny, Nx);
auto u1star = CreateMatrix(Ny, Nx + 1);
auto v1star = CreateMatrix(Ny + 1, Nx);
auto dTdx = CreateMatrix(Ny, Nx);
auto qx = CreateMatrix(Ny, Nx);
auto V = CreateMatrix(Ny, Nx);
vector Nux(Ny, 0.0);
vector u1star_L2(Ny, 0.0);
vector v1star_L2(Ny, 0.0);

// Initial velocity fields = 0.0
for (int i = 1; i < Ny; i++) {
    for (int j = 1; j < Nx + 1; j++) {
        u1[i][j] = 0.0;
        un[i][j] = u1[i][j];
        un_1[i][j] = un[i][j];
    }
}
for (int i = 1; i < Ny + 1; i++) {
    for (int j = 1; j < Nx; j++) {
        v1[i][j] = 0.0;
        vn[i][j] = v1[i][j];
        vn_1[i][j] = vn[i][j];
    }
}
// Initial pressure field = 0.0 and linearly distributed temperature field
for (int i = 0; i < Ny; i++) {
    for (int j = 0; j < Nx; j++) {
        Pg[i][j] = 0.0;
        T1[i][j] = 1.0 - i * dx;
        Tn[i][j] = T1[i][j];
        Tn_1[i][j] = Tn[i][j];
    }
}

// Apply temperature boundary conditions
BoundaryConditions(Nx, Ny, Thot, Tcold, Tn_1, Tn, T1);

// Compute R()^n-1
calculator->ComputeRu(un_1, vn_1, Run_1, rho, dx, mu, Nx, Ny);
calculator->ComputeRv(un_1, vn_1, Rvn_1, rho, dx, mu, Nx, Ny);
calculator->ComputeRt(rho, Ny, Nx, dx, lambda, cp, un_1, vn_1, Rtn_1, Tn_1);

// Compute R()^n
calculator->ComputeRu(un, vn, Run, rho, dx, mu, Nx, Ny);
calculator->ComputeRv(un, vn, Rvn, rho, dx, mu, Nx, Ny);
calculator->ComputeRt(rho, Ny, Nx, dx, lambda, cp, un, vn, Rtn, Tn);

// Time loop until convergence is reached
while ((/*res1 > maxResU || res2 > maxResV ||*/ res3 > maxResT) && ite < maxIte) {
    double maxDiff1 = 0.0;
    double maxDiff2 = 0.0;
    double maxDiff3 = 0.0;

    // Step 1a
    for (int i = 1; i < Ny - 1; i++) {
        for (int j = 1; j < Nx; j++) {
            if (ite < 5) {                                          // Using Euler Explicit for the
                  first time steps
                uP[i][j] = un[i][j] + dt / (rho * pow(dx, 2)) * Run[i][j];
            } else {
                uP[i][j] = un[i][j] + dt / (rho * pow(dx, 2)) * (3.0 / 2 * Run[i][j] - 1.0 / 2 *
                      Run_1[i][j]);
            }
        }
    }

    // Step 1b + Boussinesq approx.
    for (int i = 1; i < Ny; i++) {
        for (int j = 1; j < Nx - 1; j++) {
            Rbn[i][j] = rho * pow(dx, 2) * beta * (Tn[i][j] - Tinf) * g;
            Rbn_1[i][j] = rho * pow(dx, 2) * beta * (Tn_1[i][j] - Tinf) * g;
            if (ite < 5) {
                vP[i][j] = vn[i][j] + dt / (rho * pow(dx, 2)) * (Rvn[i][j] + Rbn[i][j]);
            } else {
                vP[i][j] = vn[i][j] + dt / (rho * pow(dx, 2)) * (
                            3.0 / 2 * Rvn[i][j] - 1.0 / 2 * Rvn_1[i][j] + 3.0 / 2 * Rbn[i][j] -
                                  1.0 / 2 * Rbn_1[
                                i][j]);
            }
        }
    }

    // Step 2
    PoissonSolver(maxResP, maxIte, rho, dx, dt, Nx, Ny, omega_P, P1, vP, uP, Pg);

    // Step 3
    for (int i = 1; i < Ny - 1; i++) {
        for (int j = 1; j < Nx; j++) {
            u1[i][j] = uP[i][j] - dt / (rho * dx) * (P1[i][j] - P1[i][j - 1]);
            double diff1 = fabs(u1[i][j] - un[i][j]);
            if (diff1 > maxDiff1) {
                maxDiff1 = diff1;
            }
```

```
    }
}
for (int i = 1; i < Ny; i++) {
    for (int j = 1; j < Nx - 1; j++) {
        v1[i][j] = vP[i][j] - dt / (rho * dx) * (P1[i - 1][j] - P1[i][j]);
        double diff2 = fabs(v1[i][j] - vn[i][j]);
        if (diff2 > maxDiff2) {
            maxDiff2 = diff2;
        }
    }
}

// Temperature boundary conditions
BoundaryConditions(Nx, Ny, Thot, Tcold, Tn_1, Tn, T1);

// Temperature evaluation
for (int i = 1; i < Ny - 1; i++) {
    for (int j = 1; j < Nx - 1; j++) {
        if (ite < 5) {
            T1[i][j] = Tn[i][j] + dt / (pow(dx, 2) * rho * cp) * Rtn[i][j];
        } else {
            T1[i][j] = Tn[i][j] + dt / (pow(dx, 2) * rho * cp) * (3.0 / 2 * Rtn[i][j] - 1.0 /
                2 * Rtn_1[i][j]);
        }
        double diff3 = fabs(T1[i][j] - Tn[i][j]);
        if (diff3 > maxDiff3) {
            maxDiff3 = diff3;
        }
    }
}

// Update residuals
t_count += dt;
res1 = maxDiff1;
res2 = maxDiff2;
res3 = maxDiff3;
// Exits the loop if res3 < maxResT; if not, continue

if (method == "AUTO" && !switched && res3 < 1e-2 && fabs(res3 - res3_old) / res3_old < 5e-2) {
    calculator = calculator_CDS;
    switched = true;
    cout << "Switched from UDS to CDS at iteration " << ite << " (res3 = " << res3 << ")\n";
}
res3_old = res3;

// Under-relaxation and update u^n and u^n-1
for (int i = 1; i < Ny - 1; i++) {
    for (int j = 1; j < Nx; j++) {
        if (omega_u < 1e-8) {
            un_1[i][j] = un[i][j];
            un[i][j] = u1[i][j];
        } else {
            u1[i][j] = un[i][j] + omega_u * (u1[i][j] - un[i][j]);
            un_1[i][j] = un[i][j];
            un[i][j] = u1[i][j];
        }
    }
}

// Under-relaxation and update v^n and v^n-1
for (int i = 1; i < Ny; i++) {
    for (int j = 1; j < Nx - 1; j++) {
        if (omega_v < 1e-8) {
            vn_1[i][j] = vn[i][j];
            vn[i][j] = v1[i][j];
        } else {
            v1[i][j] = vn[i][j] + omega_v * (v1[i][j] - vn[i][j]);
            vn_1[i][j] = vn[i][j];
            vn[i][j] = v1[i][j];
        }
    }
}

// Under-relaxation and update T^n and T^n-1
if (omega_T < 1e-8) {
    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx; j++) {
            Tn_1[i][j] = Tn[i][j];
            Tn[i][j] = T1[i][j];
        }
    }
} else {
    for (int i = 1; i < Ny - 1; i++) {
        for (int j = 1; j < Nx - 1; j++) {
            T1[i][j] = Tn[i][j] + omega_T * (T1[i][j] - Tn[i][j]);
        }
    }
    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx; j++) {
            Tn_1[i][j] = Tn[i][j];
            Tn[i][j] = T1[i][j];
        }
    }
}

// Compute R()^n+1
calculator->ComputeRu(u1, v1, Ru1, rho, dx, mu, Nx, Ny);
calculator->ComputeRv(u1, v1, Rv1, rho, dx, mu, Nx, Ny);
calculator->ComputeRt(rho, Ny, Nx, dx, lambda, cp, u1, v1, Rt1, T1);
```

```
        // Update R()^n and R()^n-1
        for (int i = 1; i < Ny - 1; i++) {
            for (int j = 1; j < Nx; j++) {
                Run_1[i][j] = Run[i][j];
                Run[i][j] = Ru1[i][j];
            }
        }
        for (int i = 1; i < Ny; i++) {
            for (int j = 1; j < Nx - 1; j++) {
                Rvn_1[i][j] = Rvn[i][j];
                Rvn[i][j] = Rv1[i][j];
            }
        }
        for (int i = 1; i < Ny-1; i++) {
            for (int j = 1; j < Nx-1; j++) {
                Rtn_1[i][j] = Rtn[i][j];
                Rtn[i][j] = Rt1[i][j];
            }
        }

        //Update time step
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx; j++) {
                V[i][j] = sqrt(pow(u1[i][j], 2) + pow(v1[i][j], 2));
            }
        }
        Vmax = std::ranges::max(std::views::join(V));
        dtc = 0.35 * dx * L / Vmax;
        dtd = 0.20 * pow(dx, 2) / (mu / rho);
        dtt = 0.20 * pow(dx, 2) / (lambda / (rho * cp));
        dt = min ({dtc, dtd, dtt});

        // Go to next time step
        ite++;
        cout << "ite=" << ite << "|res1=" << res1 << "|res2=" << res2 << "|res3=" <<
            res3 << "|dt=" << dt << endl;
    }

    cout << "Steady state reached in " << t_count << " seconds" << endl;

    // .txt files for plotting

    // File to print temperature distribution for Plot 1
    ofstream TemperatureDistribution("TemperatureDistribution.txt");
    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx; j++) {
            TemperatureDistribution << j * dx + dx / 2 << "-" << L - i * dx - dx / 2 << "-" << T1[i][j
                ] << endl;
        }
        TemperatureDistribution << "\n";
    }
    TemperatureDistribution.close();

    // File to print velocity u distribution for Plot 2
    ofstream VelocityUDistribution("VelocityUDistribution.txt");
    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx + 1; j++) {
            VelocityUDistribution << j * dx << "-" << L - i * dx - dx / 2 << "-" << u1[i][j] << endl;
        }
        VelocityUDistribution << "\n";
    }
    VelocityUDistribution.close();

    // File to print velocity v distribution for Plot 3
    ofstream VelocityVDistribution("VelocityVDistribution.txt");
    for (int i = 0; i < Ny + 1; i++) {
        for (int j = 0; j < Nx; j++) {
            VelocityVDistribution << j * dx + dx / 2 << "-" << L - i * dx << "-" << v1[i][j] << endl;
        }
        VelocityVDistribution << "\n";
    }
    VelocityVDistribution.close();

    // File to print u(y) at L/2 for Plot 4
    ofstream Uyl2("Uy.txt");
    Uyl2 << 1 << "-" << 0 << endl;
    for (int i = 0; i < Ny; i++) {
        Uyl2 << L - i * dx - dx / 2 << "-" << (u1[i][Nx / 2] + u1[i][Nx / 2 + 1]) / 2 << endl;
    }
    Uyl2 << 0 << "-" << 0 << endl;
    Uyl2.close();

    // File to print v(x) at L/2 for Plot 5
    ofstream Vxl2("Vx.txt");
    Vxl2 << 0 << "-" << 0 << endl;
    for (int j = 0; j < Nx; j++) {
        Vxl2 << j * dx + dx / 2 << "-" << (v1[Ny / 2][j] + v1[Ny / 2 + 1][j]) / 2 << endl;
    }
    Vxl2 << 1 << "-" << 0 << endl;
    Vxl2.close();

    // Average Nusselt number
    calculator->Nusselt(Nx, Ny, Tcold, Thot, L, alpha, dx, Nux_avg, T1star, T1, qx, u1star, u1, dTdx,
        Nux, dxstar);
    cout << "Average Nusselt for Ra=" << Ra << "---->" << Nux_avg << endl;

    //Maximum u* velocity at x = L/2
    for (int i = 0; i < Ny; i++) {
```

```
        u1star_L2[i] = (u1[i][Nx / 2] + u1[i][Nx / 2 + 1]) / 2 * L / alpha;
    }
    auto ustar_max = ranges::max_element(u1star_L2);
    int u_index = distance(u1star_L2.begin(), ustar_max);
    double y_u = L - (u_index + 0.5) * dx;
    double ystar_u = y_u / L;
    cout << "Max u* velocity = " << *ustar_max << " at y/L = " << ystar_u << endl;

    // Maximum v* velocity at y = L/2
    for (int j = 0; j < Nx; j++) {
        v1star_L2[j] = (v1[Ny / 2][j] + v1[Ny / 2 + 1][j]) / 2 * L / alpha;
    }
    auto vstar_max = ranges::max_element(v1star_L2);
    int v_index = distance(v1star_L2.begin(), vstar_max);
    double x_v = (v_index + 0.5) * dx;
    double xstar_v = x_v / L;
    cout << "Max v* velocity = " << *vstar_max << " at x/L = " << xstar_v << endl;

    // Stop timer and print total duration
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<seconds>(stop - start);
    cout << "Total Execution Time: " << duration.count() << " seconds" << endl;

    if (method == "AUTO") {
        delete calculator_UDS;
        delete calculator_CDS;
    } else {
        delete calculator;
    }

    return 0;
}
```

B Python Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import griddata
from scipy.interpolate import make_interp_spline

# Physical data
Ra = 1e-6
L = 1.0
lambd = 1.0
rho = 1.0
cp = 0.71
alpha = lambd / (rho * cp)

# Choose plots
print("Choose the graphs you want to visualize:")
print("- To display colormaps of T, u, and v, type: 'COLOR'")
print("- To plot velocity profiles at L/2, type: 'VELOCITY'")
graph = input("Plot: ").strip().upper()

if graph == 'COLOR':

    ################### COLORMAP: TEMPERATURE T ##################

    # Read temperature data file (formatted with 3 columns)
    df = pd.read_csv("TemperatureDistribution.txt", sep=r"\s+", engine="python", header=None, names=["
        x", "y", "T"])
    x = df["x"].values
    y = df["y"].values
    T = df["T"].values

    # Original grid resolution
    nx = len(np.unique(x))
    ny = len(np.unique(y))
    scale = 3  # Upscaling for finer interpolation

    # Create interpolated grid
    xi = np.linspace(np.min(x), np.max(x), nx * scale)
    yi = np.linspace(np.min(y), np.max(y), ny * scale)
    Xi, Yi = np.meshgrid(xi, yi)
    Ti = griddata((x, y), T, (Xi, Yi), method="cubic")

    # Plot contourf map
    fig, ax = plt.subplots(figsize=(6, 6))
    cf = ax.contourf(Xi, Yi, Ti, levels=100, cmap='jet', vmin=0, vmax=1)

    # Black isolines
    contours = ax.contour(Xi, Yi, Ti, levels=np.arange(0, 1.01, 0.1), colors='black', linewidths=0.7)
    ax.clabel(contours, inline=True, fontsize=7, fmt="%.1f")

    # Colorbar
    #cbar = plt.colorbar(cf, ax=ax)
    #cbar.set_label("Temperature")

    # Labels and title
    ax.set_xlabel("X (m)")
    ax.set_ylabel("Y (m)")

    #ax.set_title("Temperature Map with Ra = {:.0e}".format(Ra))
    #ax.set_aspect("equal")

    # Layout and save
    plt.tight_layout()
    plt.savefig("TemperatureMap_Ra_{:.0e}.png".format(Ra), dpi=100)
    plt.show()


    ################### COLORMAP: VELOCITY U ##################

    # Read velocity-u data file
    df_u = pd.read_csv("VelocityUDistribution.txt", sep=r"\s+", engine="python", header=None, names=["
        x", "y", "u"])
    x_u = df_u["x"].values
    y_u = df_u["y"].values
    u_vals = df_u["u"].values

    # Original grid resolution
    nx_u = len(np.unique(x_u))
    ny_u = len(np.unique(y_u))
    scale_u = 3  # Upscaling for finer interpolation

    # Create interpolated grid
    xi_u = np.linspace(np.min(x_u), np.max(x_u), nx_u * scale_u)
    yi_u = np.linspace(np.min(y_u), np.max(y_u), ny_u * scale_u)
    Xi_u, Yi_u = np.meshgrid(xi_u, yi_u)
    Ui = griddata((x_u, y_u), u_vals, (Xi_u, Yi_u), method="cubic")

    # Plot contourf map
    fig_u, ax_u = plt.subplots(figsize=(6, 6))
    cf_u = ax_u.contourf(Xi_u, Yi_u, Ui, levels=100, cmap='jet')

    # Black isolines
    vmin_u = np.percentile(Ui, 1)
    vmax_u = np.percentile(Ui, 99)
    levels_u = np.linspace(vmin_u, vmax_u, 11)  # 10 intervals
```

```
    contours_u = ax_u.contour(Xi_u, Yi_u, Ui, levels=levels_u, colors='black', linewidths=0.8)
    ax_u.clabel(contours_u, inline=True, fontsize=7, fmt="%.1f")

    # Colorbar
    #cbar_u = plt.colorbar(cf_u, ax=ax_u)
    #cbar_u.set_label("Velocity u (m/s)")

    # Labels and title
    ax_u.set_xlabel("X (m)")
    ax_u.set_ylabel("Y (m)")

    #ax_u.set_title("Velocity u(y) Map with Ra = {:.0e}".format(Ra))
    #ax_u.set_aspect("equal")
    ax_u.set_xlim(0.01, 0.99)
    ax_u.set_ylim(0, 1)
    ax_u.set_aspect('equal')

    # Layout and save
    plt.tight_layout()
    plt.savefig("VelocityUMap_Ra_{:.0e}.png".format(Ra), dpi=100)
    plt.show()


    ################### COLORMAP: VELOCITY V ###################

    # Read velocity-v data file
    df_v = pd.read_csv("VelocityVDistribution.txt", sep=r"\s+", engine="python", header=None, names=["
        x", "y", "v"])
    x_v = df_v["x"].values
    y_v = df_v["y"].values
    v_vals = df_v["v"].values

    # Original grid resolution
    nx_v = len(np.unique(x_v))
    ny_v = len(np.unique(y_v))
    scale_v = 3  # Upscaling for finer interpolation

    # Create interpolated grid
    xi_v = np.linspace(np.min(x_v), np.max(x_v), nx_v * scale_v)
    yi_v = np.linspace(np.min(y_v), np.max(y_v), ny_v * scale_v)
    Xi_v, Yi_v = np.meshgrid(xi_v, yi_v)
    Vi = griddata((x_v, y_v), v_vals, (Xi_v, Yi_v), method="cubic")

    # Plot contourf map
    fig_v, ax_v = plt.subplots(figsize=(6, 6))
    cf_v = ax_v.contourf(Xi_v, Yi_v, Vi, levels=100, cmap='jet')

    # Black isolines
    vmin_v = np.percentile(Vi, 1)
    vmax_v = np.percentile(Vi, 99)
    levels_v = np.linspace(vmin_v, vmax_v, 11)
    contours_v = ax_v.contour(Xi_v, Yi_v, Vi, levels=levels_v, colors='black', linewidths=0.8)
    ax_v.clabel(contours_v, inline=True, fontsize=7, fmt="%.1f")

    # Colorbar
    #cbar_v = plt.colorbar(cf_v, ax=ax_v)
    #cbar_v.set_label("Velocity v (m/s)")

    # Labels and title
    ax_v.set_xlabel("X (m)")
    ax_v.set_ylabel("Y (m)")
    #ax_v.set_title("Velocity v(x) Map with Ra = {:.0e}".format(Ra))
    #ax_v.set_aspect("equal")
    ax_v.set_xlim(0, 1)
    ax_v.set_ylim(0.01, 0.99)
    ax_v.set_aspect('equal')

    # Layout and save
    plt.tight_layout()
    plt.savefig("VelocityVMap_Ra_{:.0e}.png".format(Ra), dpi=100)
    plt.show()


elif graph == 'VELOCITY':

    ################### VELOCITY PROFILE u(y) at x = L/2 ###################

    # Read data from 3-column file
    df = pd.read_csv("VelocityUDistribution.txt", delim_whitespace=True, header=None, names=["x", "y",
        "u"])

    # Filter rows where x = L/2 (= 0.5)
    filtered = df[abs(df["x"] - 0.5) < 1e-6]

    # Extract data for plotting
    y_vals_u = filtered["y"].values
    u_vals = filtered["u"].values

    # Sort for smooth plotting
    sorted_indices = y_vals_u.argsort()
    y_vals_u = y_vals_u[sorted_indices]
    u_vals = u_vals[sorted_indices]

    # Spline interpolation for smoother curve
    y_smooth = np.linspace(y_vals_u.min(), y_vals_u.max(), 300)
    spline = make_interp_spline(y_vals_u, u_vals, k=3)
```

```
    u_smooth = spline(y_smooth)

    # Plot interpolated curve
    fig, ax = plt.subplots(figsize=(8, 6))
    ax.plot(u_smooth, y_smooth, color='red', linewidth=2)
    ax.set_xlabel("u (m/s)")
    ax.set_ylabel("y (m)")
    # ax.set_title(f"Velocity Profile u(y) at x = L/2 with Ra = {Ra:.0e}")
    ax.grid(True)
    ax.set_ylim([0, 1])
    plt.tight_layout()
    plt.savefig(f"Profile_Uy_Ra_{Ra:.0e}.png", dpi=100)
    plt.show()


    # Print maximum u* velocity and y position at x = L/2
    filename = 'VelocityUDistribution.txt'

    # Load data without header
    df = pd.read_csv(filename, delim_whitespace=True, header=None, names=['x', 'y', 'velocity'])

    # Filter rows where x = 0.5
    filtered = df[abs(df['x'] - 0.5) < 1e-6]

    # Find row with max velocity
    if not filtered.empty:
        max_row = filtered.loc[filtered['velocity'].idxmax()]
        print(f"Maximum u* velocity at x = L/2 is {max_row['velocity'] * L / alpha}, occurring at y =
            {max_row['y']}")
    else:
        print("No data found with x = 0.5")



    ################### VELOCITY PROFILE v(x) at y = L/2 ##################

    # Read data from 3-column file
    df = pd.read_csv("VelocityVDistribution.txt", delim_whitespace=True, header=None, names=["x", "y",
        "v"])

    # Filter rows where y = L/2 (= 0.5)
    filtered = df[abs(df["y"] - 0.5) < 1e-6]

    # Extract data for plotting
    x_vals_v = filtered["x"].values
    v_vals = filtered["v"].values

    # Sort for smooth plotting
    sorted_indices = x_vals_v.argsort()
    x_vals_v = x_vals_v[sorted_indices]
    v_vals = v_vals[sorted_indices]

    # Spline interpolation for smoother curve
    x_smooth = np.linspace(x_vals_v.min(), x_vals_v.max(), 300)
    spline = make_interp_spline(x_vals_v, v_vals, k=3)
    v_smooth = spline(x_smooth)

    # Plot interpolated curve
    fig, ax = plt.subplots(figsize=(8, 6))
    ax.plot(x_smooth, v_smooth, color='blue', linewidth=2)
    ax.set_xlabel("x (m)")
    ax.set_ylabel("v (m/s)")
    # ax.set_title(f"Velocity Profile v(x) at y = L/2 with Ra = {Ra:.0e}")
    ax.grid(True)
    ax.set_xlim([0, 1])
    plt.tight_layout()
    plt.savefig(f"Profile_Vx_Ra_{Ra:.0e}.png", dpi=100)
    plt.show()


    # Print maximum v* velocity and x position at y = L/2
    filename = 'VelocityVDistribution.txt'

    # Load data without header
    df = pd.read_csv(filename, delim_whitespace=True, header=None, names=['x', 'y', 'velocity'])

    # Filter rows where y = 0.5
    filtered = df[abs(df['y'] - 0.5) < 1e-6]

    # Find row with max velocity
    if not filtered.empty:
        max_row = filtered.loc[filtered['velocity'].idxmax()]
        print(f"Maximum v* velocity at y = L/2 is {max_row['velocity'] * L / alpha}, occurring at x =
            {max_row['x']}")
    else:
        print("No data found with y = 0.5")

else:
    print("Invalid input. Please choose 'COLOR' or 'VELOCITY'.")
```