High-Rayleigh Simulation of a Rectangular Differentially Heated Cavity

Master's Degree in Thermal Engineering

Giada Alessi

Universitat Politècnica de Catalunya

June 9, 2025

Abstract

This report presents a two-dimensional direct numerical simulation (DNS) of a differentially heated cavity (DHC) with a rectangular geometry and high aspect ratio (1:4), subject to Rayleigh numbers up to $Ra \sim 10^9$. The objective is to investigate the onset of unsteady convective structures and quasi-turbulent behavior in natural convection. The numerical solver is based on a fractional step method applied on a staggered mesh, implemented in `C++` with post-processing in `Python`. Key flow quantities such as time-averaged fields, kinetic energy budget, turbulent kinetic energy (TKE), and Reynolds stresses are computed and analyzed. The study confirms the emergence of unsteady dynamics and quasi-periodic fluctuations for increasing Rayleigh numbers, and highlights the computational challenges related to pressure resolution and mesh refinement at high Ra.

Contents

1 Introduction

1.1 Physical Motivation and Scope

The present study represents a continuation of the numerical investigation of the Differentially Heated Cavity (DHC) problem. In the previous analysis, the classical square cavity configuration was explored across a range of Rayleigh numbers to capture the transition from conduction-dominated to convection-dominated heat transfer regimes. The flow characteristics were shown to evolve significantly as Ra increased, with the formation of vortices and convective structures. Temperature distributions and velocity fields were consistent with the expected physical behavior, validating the implementation of the energy and momentum equations under the Boussinesq approximation.
The objective of this extended work is to dive deeper into the dynamics of natural convection by increasing both the domain aspect ratio and the Rayleigh number. Specifically, the study considers a rectangular cavity with a 1:4 aspect ratio and three different Rayleigh numbers around the order of $10^8 - 10^9$, which are expected to show the transition from a steady state behavior to unsteady and chaotic flow features that resemble turbulence. It is important to emphasize, however, that true turbulence is a three-dimensional phenomenon. Thus, the present study should be considered as a preliminary investigation of quasi-turbulent patterns in a two-dimensional setting.

1.2 Extended DHC Configuration

The geometry of the cavity considered in this study is rectangular, with an aspect ratio of 1:4. As shown in the figure below, the width is $L = 1$ and the height is $H = 4L$, with gravity acting downward as indicated by the arrow. The left vertical wall is maintained at a constant hot temperature T_{hot}, while the right vertical wall is kept at a constant cold temperature T_{cold}; the top and bottom horizontal walls are considered adiabatic. The cavity is filled with air ($\text{Pr} = 0.71$), and natural convection occurs due to the imposed temperature difference, as all walls are stationary.

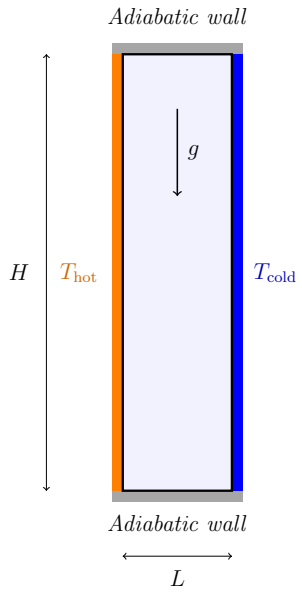


Figure 1: Schematic of the rectangular differentially heated cavity (aspect ratio 1:4) with boundary conditions.

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

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

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: $Ra = 1.0 \times 10^8$, 6.4×10^8, 1.07×10^9
- The gravitational acceleration g is computed from the definition of the Rayleigh number.

1.3 Objectives of the Study

As an extension of the previous study conducted on the square cavity (1×1), the objective of the present work is to investigate the convective flow behavior in a taller cavity of aspect ratio 1×4, at significantly higher Rayleigh numbers, where complex flow patterns and quasi-turbulent structures are expected to emerge. The following physical quantities will be analyzed to characterize the flow dynamics and heat transfer performance:

- Internal probes properties;
- Time-averaged velocity and temperature fields;
- Time-averaged stream function.
- Time-averaged Nusselt number;
- Maximum velocities and velocity profiles at mid-sections;
- Instantaneous kinetic energy budget;
- Turbulent kinetic energy and Reynolds stresses.

2 Theoretical Background

This chapter presents the theoretical background necessary to analyze natural convection flows at high Rayleigh numbers. It begins with an overview of turbulence, highlighting its key physical features and inside mechanisms. The main numerical strategies for simulating turbulent flows (RANS, LES, and DNS) are then introduced, with a focus on their applicability to two-dimensional configurations. The positioning of the present work is discussed in relation to the existing literature, justifying the use of a two-dimensional DNS-like approach. Finally, the governing equations used in the simulations are presented, including their spatial and temporal discretization.

2.1 Introduction to Turbulence

Turbulence is the most common flow regime we encounter in both engineering applications and in nature. It shows up everywhere, from the air flowing around cars, buildings, and airplanes, to the motion of fluids inside engines, turbines, and ventilation systems. Because it plays such a central role in how fluids behave in real-world situations, having a solid understanding of turbulence is essential for analyzing and designing efficient fluid systems.
Turbulence can be studied using the Navier–Stokes equations and different numerical methods, but it is usually described through a set of typical features, as outlined by Tennekes and Lumley [1].

- **Irregularity:** Turbulent flows are highly chaotic, random, and non-repetitive. The motion is characterized by vortical structures ranging from large energy-containing eddies to small dissipative scales.
- **Enhanced Diffusivity:** Turbulence significantly increases the rate of momentum, heat, and mass transport. This elevated diffusivity improves mixing and influences phenomena such as boundary layer separation and wall friction in internal flows.
- **High Reynolds Number:** Turbulence typically occurs at high Reynolds numbers, which in natural convection corresponds to high Rayleigh numbers. For example, in vertical natural convection between differentially heated walls, the transition to turbulence can occur for Ra values in the range of 10^8 to 10^9, depending on the geometry and boundary conditions.
- **Three-Dimensionality:** All turbulent flows are three-dimensional. Even if the mean flow is two-dimensional, the fluctuating components evolve into three-dimensional structures due to vortex stretching and tilting mechanisms.
- **Dissipation:** Turbulent kinetic energy undergoes a cascade process, originating from large eddies and successively transferring energy to smaller eddies, until it is finally dissipated into internal energy by viscous effects.
- **Continuum Hypothesis:** Despite the small scale of dissipative structures, they are still orders of magnitude larger than molecular scales. Hence, the continuum assumption remains valid throughout the turbulent flow field.

Turbulent flows are often described using a decomposition into a mean part (typically time-averaged) and a fluctuating part. This approach, known as Reynolds decomposition, forms the basis for many turbulence modeling strategies.
Several physical mechanisms are known to govern the generation, redistribution, and dissipation of turbulence across scales. These include:

- **Vortex Stretching:** A mechanism that increases the intensity of vorticity by elongating vortical structures, particularly relevant in three-dimensional flows.
- **Vortex Tilting:** The redirection of vorticity vectors due to velocity gradients, contributing to the redistribution of turbulence across directions.

- **Kolmogorov Scales:** Named after Andrey Kolmogorov, these define the smallest scales of turbulence where viscous dissipation dominates. They are characterized by specific velocity, length, and time scales derived from the viscosity and dissipation rate.
- **Energy Spectrum:** Describes the distribution of turbulent kinetic energy among different length scales, represented in wave number space. It consists of three distinct regions: the energy-containing range, dominated by large eddies that extract energy from the mean flow; the inertial subrange, where energy is transferred from large to small scales without dissipation; and the dissipation range, where small eddies convert kinetic energy into internal energy due to viscous effects.

2.2 Numerical Strategies for Turbulent Flow Simulation

The numerical simulation of turbulent flows involves a compromise between physical fidelity and computational cost. This trade-off affects the choice of turbulence modeling, which determines how the wide range of spatial and temporal scales is resolved or modeled. The three principal strategies in Computational Fluid Dynamics (CFD) are:

- **Reynolds-Averaged Navier–Stokes (RANS):** This approach is based on time-averaging the Navier–Stokes equations, resulting in a set of equations for the mean flow field. The effects of turbulence are entirely modeled using closure relations for the Reynolds stress tensor. Common models include the k-ε, k-ω, and Spalart–Allmaras models, which solve additional transport equations for turbulent quantities such as kinetic energy and dissipation. RANS is computationally inexpensive but it cannot capture the detailed, time-dependent turbulent structures, which are instead modeled in an averaged sense.
- **Large Eddy Simulation (LES):** LES directly resolves the large, energy-containing eddies and models only the smaller motions using a subgrid-scale (SGS) model. These smaller scales are more isotropic and behave in a more universal way, which makes them easier to model. LES offers much better accuracy than RANS, especially for unsteady flow features, but it requires finer grids and smaller time steps, making it more computationally expensive.
- **Direct Numerical Simulation (DNS):** DNS solves the full, unsteady Navier–Stokes equations without any turbulence model, capturing all scales of motion down to the Kolmogorov length and time scales. This requires extremely fine grids and very small time steps, making DNS feasible only for low Reynolds number flows or simplified geometries. DNS provides detailed insight into the fundamental physics of turbulence and is often used to validate models and theories, but its computational cost is too high for practical engineering applications.

In two-dimensional simulations, the absence of the third spatial dimension prevents the development of a complete turbulence cascade. Therefore, even if the 2D simulation with a DNS approach resolves all scales of motion and does not rely on turbulence models, it does not capture fully developed turbulence as observed in real 3D flows.

The present study uses a direct numerical simulation (DNS) approach, solving the incompressible Navier–Stokes and energy equations without any turbulence model. Since

the simulation is two-dimensional, the unsteady flow at high Rayleigh numbers is better described as quasi-turbulent or pseudo-turbulent. Although it does not capture fully 3D turbulence, this approach still provides insight into the unsteady and transitional convection typical of high-Rayleigh natural flows in confined domains.

2.3 Positioning of the Present Work

The current study, a direct numerical simulation in two dimensions, is aimed to capture the onset and evolution of quasi-turbulent flow structures driven by thermal gradients. A key consideration, however, is the validity of the two-dimensional assumption when modeling turbulent regimes. This issue has been addressed in several studies, with differing conclusions depending on the Rayleigh number and aspect ratio of the cavity.

The Rayleigh number (Ra) is the key dimensionless parameter that governs the onset and evolution of natural convection in buoyancy-driven systems. It quantifies the balance between the destabilizing thermal buoyancy forces and the stabilizing effects of viscous and diffusive transport. As Ra increases, the system transitions through a series of distinct flow regimes, as extensively described by Bejan [3]:

- For $Ra \lesssim 10^3$: conduction-dominated regime; the fluid remains at rest.
- For $10^4 \lesssim Ra \lesssim 10^7$: onset of steady convection; a large primary vortex forms in the cavity.
- For $10^8 \lesssim Ra \lesssim 10^9$: unsteady flow regime with periodic or quasi-periodic oscillations.
- For $Ra \gtrsim 10^10$: transition to fully turbulent regime with strong plumes and spatially complex behavior.

In the present case of a differentially heated cavity (DHC) with aspect ratio $H/L = 4$ and air as the working fluid ($Pr \approx 0.71$), it is acceptable to assume that the transition to unsteady or quasi-turbulent behavior begins around:

$$Ra_{\text{crit}} \approx 1.07 \times 10^9$$

This estimate is derived from the transition criterion for natural convection boundary layers along vertical isothermal walls reported by Bejan [3], where turbulence is observed for $Gr \sim 1.5 \times 10^9$, corresponding to $Ra \sim 10^9$ for air. However, for enclosures heated from the side, the flow may become unsteady at higher Rayleigh numbers, especially in tall geometries ($H/L > 2$), as in our case.

Based on the literature review, the critical Rayleigh number Ra_{crit} for the transition from steady laminar to unsteady or turbulent convection can be summarized as follows:

- For 2D classical DHC configurations (adiabatic horizontal walls and lateral heating):

$$Ra_{\text{crit}} \sim 10^8\text{–}10^9 \quad [4, 5]$$

- For 3D cavities with perfectly conducting horizontal walls and symmetry assumptions:

$$Ra_{\text{crit}} \sim 10^5\text{–}10^7 \quad [6, 7, 8, 9]$$

- For configurations with periodic spanwise boundary conditions:
  - Three regimes can be observed: 2D steady, 2D unsteady, and 3D unsteady.
  - Transition to unsteadiness and to 3D behavior are not necessarily simultaneous.
  - In tall cavities, the onset of 3D effects occurs after the 2D unsteady regime:

$$Ra \sim 10^8\text{–}10^9 \quad [10]$$

- Regarding the validity of the Boussinesq approximation, discrepancies emerge at:

$$Ra > 10^9\text{–}10^{10} \quad [11, 12, 13]$$

  affecting mainly turbulence statistics.

As highlighted by Trias et al. [14], while 2D results are not fully representative of realistic turbulent dynamics, they still offer physically meaningful insights into the behavior of mean thermal and flow fields, particularly at the early stages of turbulence onset. Based on the considerations discussed above, this study focuses on analyzing the transition from steady to turbulent natural convection using three different Rayleigh numbers. A value of $Ra = 10^8$ is selected to represent a purely steady regime. The transitional case at $Ra = 6.4 \times 10^8$ is chosen following Trias et al., as it allows for graphical comparisons with previously published results [15]. Finally, the case $Ra = 1.07 \times 10^9$, indicated by other sources as a critical value for the onset of turbulence, is used here to investigate the evolution of the flow under turbulent conditions.
Moreover, the present work adopts a two-dimensional DNS-like framework as a compromise between computational feasibility and physical resolution, with the understanding that certain turbulent features, particularly those associated with three-dimensional energy redistribution, cannot be captured in this configuration.

2.4 Governing Equations

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. 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.

- **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.

- **Kinetic Energy Equation:**

$$\frac{dE_k}{dt} = \int_\Omega \mathbf{v} \cdot \left(-\rho(\mathbf{v} \cdot \nabla)\mathbf{v} - \nabla p + \mu \nabla^2 \mathbf{v} - \rho\beta(T - T_\infty)\mathbf{g} \right) d\Omega \tag{5}$$

This equation represents the global kinetic energy balance for incompressible flows. It is obtained by projecting the momentum equation (3) onto the velocity field and integrating over the entire domain Ω. Each term reflects the contribution of a physical mechanism: convective transport, pressure work, viscous diffusion, and external forcing, such as buoyancy under the Boussinesq approximation.

3 Numerical Methods Overview

To numerically solve the case, several methods have been implemented. The governing equations have been discretized using the Finite Volume Method (FVM) on a staggered mesh configuration. The momentum equations have been solved using the fractional step method, with the Poisson equation being solved implicitly using the Gauss–Seidel method.The second-order Adams-Bashforth explicit time integration scheme has instead been employed to update the velocity and temperature fields.
A brief overview of the numerical methods, derivation of the kinetic energy, as well as the spatial and temporal discretization strategies adopted for the present case (aspect ratio 1:4), is provided in this section. To avoid redundancy, for a more detailed explanation, refer to the paper *Differentially Heated Square Cavity.*

3.1 Resolution Method

The Differentially Heated Cavity (DHC) case has been solved using the Fractional Step Method (FSM) followed by the resolution of the energy equation. The FSM is a numerical technique used to decouple the computation of the velocity and pressure fields in the solution of the Navier–Stokes equations. By introducing a predictor velocity and applying the Helmholtz–Hodge decomposition, the method splits the resolution into three steps:

- **Step 1 (FSM):** Compute the predictor velocity:

$$\mathbf{v}^p = \mathbf{v}^n + \frac{\Delta t}{\rho}\left[\frac{3}{2}\mathbf{R}(\mathbf{v}^n) - \frac{1}{2}\mathbf{R}(\mathbf{v}^{n-1}) + \frac{3}{2}\mathbf{R_B}(T^n) - \frac{1}{2}\mathbf{R_B}(T^{n-1})\right] \tag{6}$$

  where:

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

  and:

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

- **Step 2 (FSM):** Solve the pressure Poisson equation:

$$\nabla^2 P^{n+1} = \frac{\rho}{\Delta t}\nabla\cdot\mathbf{v}^p \tag{7}$$

- **Step 3 (FSM):** Correct the velocity field:

$$\mathbf{v}^{n+1} = \mathbf{v}^p - \frac{\Delta t}{\rho}\nabla P^{n+1} \tag{8}$$

  In this formulation, the terms $\mathbf{R}(\mathbf{v}) = (R(v_x), R(v_y))$ group the convective and diffusive contributions of the momentum equation for each velocity component. Specifically, $R(v_x) = conv_x + diff_x$ and $R(v_y) = conv_y + diff_y$. The term $\mathbf{R_B}(T)$ accounts for the buoyancy forces, evaluated through the Boussinesq approximation (only in y-direction).
  After completing the three steps of the FSM, the energy equation is solved to evaluate the temperature field and the kinetic energy field is obtained.

- **Step 4:** Evaluate temperature field:

$$T^{n+1} = T^n + \frac{\Delta t}{\rho c_P}\left[\frac{3}{2}\mathcal{R}(T^n) - \frac{1}{2}\mathcal{R}(T^{n-1})\right] \tag{9}$$

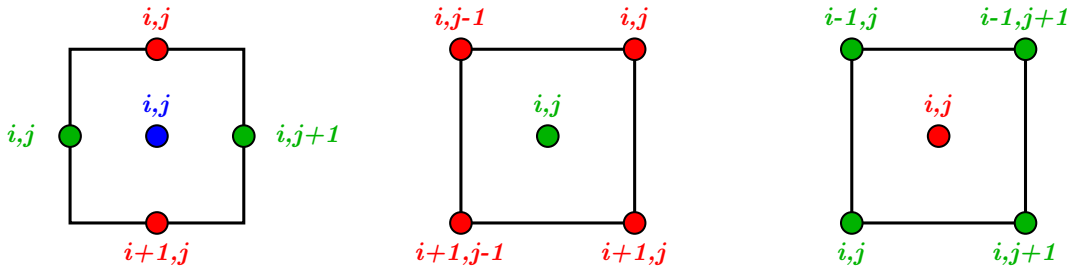
  where:

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

- **Step 5:** Evaluate kinetic energy budget, averaged fields and stream functions.

3.2 Staggered Meshes

Since the Fractional Step Method (FSM) is employed to solve the momentum equations, a staggered mesh configuration has been implemented to avoid numerical instabilities, such as spurious checkerboard pressure patterns, that commonly arise on collocated grids. The staggered layout ensures accurate pressure gradient evaluation, improving both stability and physical consistency of the solution.



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

Figure 2: Relative index of surrounding nodes for different meshes.

In this configuration, pressure and temperature values are stored at the center of each control volume. The horizontal velocity component u is defined at the vertical faces of the control volumes, while the vertical component v is located at the horizontal faces. The relative indexing between pressure and velocity nodes is illustrated in Figure 2.
The temperature field is defined on the same mesh as pressure, as it does not directly interact with the pressure correction step. This choice simplifies the implementation without introducing computational inconsistencies.

3.3 Temporal and Spatial Discretization

Here, the time and space discretized governing equations are presented as already adapted to the rectangular cavity considered in this study. To maintain consistency with the square cavity case and to keep both the implementation and analysis simple, a uniform mesh has been adopted: N_x control volumes are defined along the x-direction and $N_y = 4 \cdot N_x$ along the y-direction, resulting in square control volumes with $\Delta x = \Delta y$, and identical surface areas and volumes.
Thanks to this adaptation, all the simplifications applied to the square cavity case remain valid in the present configuration. As a result, the same spatial and temporal discretization schemes are employed and can be fully reviewed in the reference report *Differentially Heated Square Cavity*.

Predictor Velocities (6):

second-order Adams-Bashforth explicit time integration scheme.

$$u^p = u^n + \frac{\Delta t}{\rho(\Delta x)^2}\left(\frac{3}{2}R(u^n) - \frac{1}{2}R(u^{n-1})\right) \tag{10}$$

with:

$$R(u) = -\rho\Delta x\left(v_n u_n - v_s u_s + u_e u_e - u_w u_w\right) + \mu\left(u_N + u_S + u_E + u_W - 4u_P\right)$$

$$v^p = v^n + \frac{\Delta t}{\rho(\Delta x)^2}\left(\frac{3}{2}R(v^n) - \frac{1}{2}R(v^{n-1}) + \frac{3}{2}R_{B_y}^{n} - \frac{1}{2}R_{B_y}^{n-1}\right) \tag{11}$$

with:

$$R(v) = -\rho\Delta x\left(v_n v_n - v_s v_s + u_e v_e - u_w v_w\right) + \mu\left(v_N + v_S + v_E + v_W - 4v_P\right)$$

and:

$$R_{B,y} = \rho\beta(\Delta x)^2(T - T_\infty)g$$

Poisson Equation (7):

implicit time integration scheme.

$$P^{n+1} = \frac{1}{4}\left[P_N^{n+1} + P_S^{n+1} + P_E^{n+1} + P_W^{n+1} - \frac{\rho\,\Delta x}{\Delta t}\left(v_n^p - v_s^p + u_e^p - u_w^p\right)\right] \tag{12}$$

Corrected Velocities (8):

central differences between adjacent pressure nodes.

$$u^{n+1} = u^p - \frac{\Delta t}{\rho\,\Delta x}\left(P_P^{n+1} - P_W^{n+1}\right) \tag{13}$$

$$v^{n+1} = v^p - \frac{\Delta t}{\rho\,\Delta x}\left(P_P^{n+1} - P_S^{n+1}\right) \tag{14}$$

3.4 Derivation of the Kinetic Energy Equation

The concept of kinetic energy is here introduced; the derivation will be carried out using a matrix-based formulation rather than an integral form, though both approaches are mathematically equivalent.
We define the kinetic energy E_k as:

$$E_k = \frac{1}{2}\langle \mathbf{v}, \mathbf{v}\rangle_\Omega = \frac{1}{2}\mathbf{v}^T\Omega\mathbf{v} \tag{15}$$

where $\mathbf{v}$ is the discrete velocity field, and Ω is a symmetric matrix. In uniform finite volume discretizations, Ω is typically diagonal and proportional to the local control volume, $\Omega = \Delta x\,\Delta y \cdot I$, where I is the identity matrix.
However, in this case of a uniform grid where each node corresponds to a control volume of equal area, we can treat Ω as the scalar Δx^2, since multiplying by the identity matrix does not alter the vector.
To derive the evolution of kinetic energy in time, we differentiate E_k:

$$\frac{dE_k}{dt} = \frac{1}{2}\frac{d}{dt}\left(\mathbf{v}^T\Omega\mathbf{v}\right) \tag{16}$$

$$= \frac{1}{2}\left(\frac{d\mathbf{v}^T}{dt}\Omega\mathbf{v} + \mathbf{v}^T\Omega\frac{d\mathbf{v}}{dt}\right) \tag{17}$$

Knowing that Ω is symmetric ($\Omega^T = \Omega$), we can apply $\left(\frac{d\mathbf{v}^T}{dt}\Omega\mathbf{v}\right) = \left(\Omega\frac{d\mathbf{v}}{dt}\right)^T\mathbf{v}$, and obtain:

$$2\frac{dE_k}{dt} = \left(\Omega\frac{d\mathbf{v}}{dt}\right)^T\mathbf{v} + \mathbf{v}^T\Omega\frac{d\mathbf{v}}{dt} \tag{18}$$

Now, we consider the momentum equation 3 written in the following way:

$$\Omega\rho\frac{d\mathbf{v}}{dt} = \mu\nabla^2\mathbf{v} - \rho(\mathbf{v}\cdot\nabla)\mathbf{v} - \Omega\nabla p - \Omega\rho\beta(T - T_\infty)\mathbf{g} \tag{19}$$

According to equation 18, it is necessary to transpose 19, obtaining:

$$\left(\Omega\rho\frac{d\mathbf{v}}{dt}\right)^T = \left(\mu\nabla^2\mathbf{v}\right)^T - \left(\rho(\mathbf{v}\cdot\nabla)\mathbf{v}\right)^T - (\Omega\nabla p)^T - \left(\Omega\rho\beta(T-T_\infty)\mathbf{g}\right)^T \tag{20}$$

Handling each term individually:

- The viscous diffusion term becomes:

$$\left(\mu\nabla^2\mathbf{v}\right)^T = \mu\left(\nabla^2\mathbf{v}\right)^T = \mu\mathbf{v}^T\nabla^{2T}$$

- The convective term is transposed as:

$$\left(\rho(\mathbf{v}\cdot\nabla)\mathbf{v}\right)^T = \rho\mathbf{v}^T(\mathbf{v}\cdot\nabla)^T$$

- The pressure gradient term:

$$(\Omega\nabla p)^T = (\nabla p)^T\Omega$$

- The body force term:

$$\left(\Omega\rho\beta(T-T_\infty)\mathbf{g}\right)^T = \left(\rho\beta(T-T_\infty)\mathbf{g}\right)^T\Omega$$

Putting all the transposed terms together, the final expression becomes:

$$\left(\Omega\rho\frac{d\mathbf{v}}{dt}\right)^T = \mu\mathbf{v}^T\nabla^{2T} - \rho\mathbf{v}^T(\mathbf{v}\cdot\nabla)^T - (\nabla p)^T\Omega - \left(\rho\beta(T-T_\infty)\mathbf{g}\right)^T\Omega \tag{21}$$

Now, equations 19 and 21 can be substituted in 18, obtaining:

$$\begin{aligned} 2\frac{dE_k}{dt} = &\left[\mu\mathbf{v}^T\nabla^{2T} - \rho\mathbf{v}^T(\mathbf{v}\cdot\nabla)^T - (\nabla p)^T\Omega - \left(\rho\beta(T-T_\infty)\mathbf{g}\right)^T\Omega\right]\mathbf{v} \\ &+ \mathbf{v}^T\left[\mu\nabla^2\mathbf{v} - \rho(\mathbf{v}\cdot\nabla)\mathbf{v} - \Omega\nabla p - \Omega\rho\beta(T-T_\infty)\mathbf{g}\right] \end{aligned} \tag{22}$$

which can be reformulated and grouped as:

$$\begin{aligned} 2\frac{dE_k}{dt} = &\underbrace{\mathbf{v}^T\left(\mu\nabla^{2T} + \mu\nabla^2\right)\mathbf{v}}_{\text{Viscous diffusion}} - \underbrace{\rho\mathbf{v}^T\Big((\mathbf{v}\cdot\nabla)^T + (\mathbf{v}\cdot\nabla)\Big)\mathbf{v}}_{\text{Convective transport}} \\ &- \underbrace{\mathbf{v}^T\Big(\Omega\nabla p + (\nabla p)^T\Omega\Big)\mathbf{v}}_{\text{Pressure work}} - \underbrace{\mathbf{v}^T\Big(\Omega\rho\beta(T-T_\infty)\mathbf{g} + (\Omega\rho\beta(T-T_\infty)\mathbf{g})^T\Big)\mathbf{v}}_{\text{Buoyancy (Boussinesq)}} \end{aligned} \tag{23}$$

The evolution of kinetic energy reflects how different physical processes inject, redistribute, or dissipate energy throughout the domain. Each term, in this balance, has a distinct physical meaning:

- **Viscous Diffusion:**

$$\mathbf{v}^T \left(\mu\nabla^{2T} + \mu\nabla^2\right)\mathbf{v}$$

This term represents viscous dissipation. It quantifies how the internal friction within the fluid converts kinetic energy into dissipated heat due to velocity gradients.

- **Convective Transport:**

$$\rho\mathbf{v}^T\Big((\mathbf{v}\cdot\nabla)^T + (\mathbf{v}\cdot\nabla)\Big)\mathbf{v}$$

This term models the transport of momentum by the flow itself. In an ideal situation, it would not generate or dissipate any kinetic energy, it would simply move it from one region of the domain to another. However, in practice, especially at high Ra, the numerical discretization may cause this term to act as if it's adding or removing energy, introducing artificial effects in the solution.

- **Pressure Work:**

$$\mathbf{v}^T\Big(\Omega\nabla p + (\nabla p)^T\Omega\Big)\mathbf{v}$$

This term is associated with the mechanical work done by pressure forces. Although it does not create or destroy kinetic energy in a closed, incompressible system (its global contribution vanishes), it is important for redistributing energy between regions of the domain.

- **Buoyancy Forcing (Boussinesq):**

$$\mathbf{v}^T\Big(\Omega\rho\beta(T-T_\infty)\mathbf{g} + (\Omega\rho\beta(T-T_\infty)\mathbf{g})^T\Big)\mathbf{v}$$

This term represents the energy input from external body forces, specifically the buoyancy term under the Boussinesq approximation. It is the only source term in the equation and is directly responsible for injecting kinetic energy into the flow.

Equation 23 can be rewritten in its classical integral form, which corresponds to equation 5.

3.5 Boundary Conditions

In the present 1×4 rectangular cavity, boundary conditions are defined to ensure the physical accuracy of the simulation and to induce natural convection and transitional or semi-turbulent flow behavior.

- **Velocity BC:** the velocity field is subject to no-slip conditions on all walls, meaning that both horizontal and vertical components are set to zero: $u = 0$, $v = 0$. This models the fact that the cavity boundaries are stationary.
- **Pressure BC:** for the pressure field, Neumann boundary conditions are applied on all sides ($\partial P/\partial n = 0$) along all boundaries. This is implemented by assigning each pressure value at the boundary equal to its immediate neighbor inside the domain (e.g., $P = P_E$ on the West wall).

- **Temperature BC:** temperature boundary conditions reflect the thermal driving of the flow. The left (West) vertical wall is maintained at a hot temperature $T = 1$, while the right (East) vertical wall is kept cold at $T = 0$. These are imposed using Dirichlet conditions. On the horizontal walls (bottom and top), adiabatic conditions are assumed, meaning that no heat transfer occurs through those walls. This corresponds to Neumann boundary condition ($\partial T/\partial n = 0$), implemented by copying the temperature from the nearest internal cell (e.g., $T = T_N$ on the bottom wall).
- **Kinetic Energy BC:** the kinetic energy at the domain boundaries is zero, as all walls are stationary. This condition is automatically satisfied as a consequence of the no-slip velocity boundary conditions previously imposed, which enforce both horizontal and vertical velocity components to be zero at the boundaries. Consequently, the kinetic energy will also be zero.

All boundary conditions are applied consistently with the staggered mesh layout described in Section 3.2, ensuring correct positioning of variables and avoiding errors at the domain boundaries.

3.6 Adaptive Convective Schemes and Relaxation Strategy

The same adaptive automatic switching scheme implemented in the square DHC simulation has been applied here. The simulation employs both the upwind differencing scheme (UDS) and the central differencing scheme (CDS), switching from the first to the second based on the temperature residual. This dynamic switching is implemented to enhance numerical robustness in the early stages of the simulation (by using UDS) and to improve accuracy once the flow becomes sufficiently resolved (by switching to CDS). Since both schemes have already been described in detail in the square DHC study, they are not discussed further here.
A significant change has been introduced regarding the relaxation strategy. To allow the flow to fully develop, under-relaxation on the velocity and temperature fields has been removed. This decision is justified by the fact that under-relaxation, while useful to enhance stability, may excessively attenuate physical transients or delay flow development.
A range of over- and under-relaxation factors were tested for the Gauss-Seidel solver used for the pressure Poisson equation, and an over-relaxation factor of 1.3 proved effective.

3.7 Numerical Stability and Convergence

The numerical simulations were performed under a CFL-based time-stepping condition, which was slightly relaxed to accelerate the simulation process.

$$\Delta t_c = 0.35\frac{\Delta x L}{\mathbf{v}_{\max}}, \quad \Delta t_d = 0.20\frac{\Delta x^2}{\nu}, \quad \Delta t_T = 0.20\frac{\Delta x^2}{\alpha} \tag{24}$$

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

$$\Delta t = \min\{\Delta t_c, \Delta t_d, \Delta t_T\} \tag{25}$$

In the case of $Ra = 10^8$, the simulation initially progressed very slowly during the early transient phase. However, once the flow approached a steady state, the convergence speed

increased significantly. This behavior was likely due to the Gauss–Seidel pressure solver, which may have required many iterations to stabilize the pressure field at the beginning. Once the system became steady, the residuals dropped and the solver appeared to perform much more efficiently.
To investigate this hypothesis, a time counter was introduced inside the Gauss-Seidel loop. By comparing the computational time required for the pressure solver with the total runtime of the simulation, it became evident that the solver accounted for a disproportionate share of the total computational cost, more than 97% of the total simulation time for each Ra.
This analysis clearly indicates that implementing a more efficient pressure solver would allow the use of time steps strictly limited by the CFL condition without concern for excessive runtime.
Initially, a proper grid independence study was not feasible due to the high computational cost. As a result, a relatively coarse mesh was used to carry out the simulations. Nonetheless, it is strongly recommended to revisit this point as part of future work, particularly for high Ra cases, where accurate resolution of boundary layers and thermal plumes is crucial. A refined mesh, enabled by a faster solver, would allow for a more reliable quantitative validation of the results.

3.8 Stop Criteria and Probe Strategy

In transient or quasi-turbulent regimes, such as the one investigated in this study, the temperature and velocity fields do not converge to a steady state. Instead, they display unsteady dynamics driven by the development of convective structures and thermally-induced plumes. These fluctuations signal the transition toward a statistically stationary regime, where the flow no longer settles into a fixed configuration but oscillates around a mean state. Under these conditions, it becomes necessary to compute time-averaged fields by averaging thermo-fluid dynamic quantities over a sufficiently long time interval. To do so, the simulation must first go through an initial development phase, during which transients decay and the statistical properties of the flow stabilize. Determining the appropriate duration of both this transient phase and the averaging window is not straightforward. Initially, a statistical convergence test based on the stabilization of the temperature's standard deviation was used: the flow was considered statistically stationary when the relative variation of this quantity remained below a predefined threshold for a sufficient number of time steps.
However, at high Rayleigh numbers, the small time steps required lead to long simulation times and may trigger the convergence logic prematurely, causing the simulation to stop before the full development of the flow field.
For this reason, a probe monitoring strategy was adopted, which proved to be simpler and more reliable. The temporal evolution of temperature and velocity components was recorded at four probe locations positioned near the corners of the cavity, where unsteady dynamics are more pronounced compared to the central region, which may falsely appear steady due to weaker gradients. For the case with Rayleigh number $\mathrm{Ra} = 6.4 \times 10^8$, the simulation was run for approximately 0.7 s. After an initial transient, the signals exhibit periodic or quasi-periodic oscillations around a mean value, indicating that the flow has reached a statistically stationary state suitable for initiating time averaging.

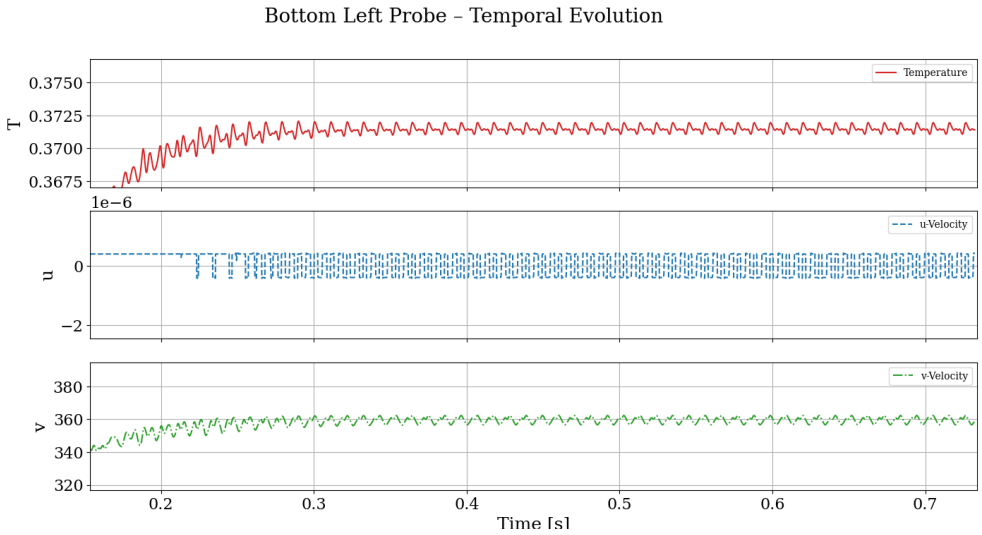


Figure 3: Temporal evolution of temperature and velocity components (u, v) at the bottom-left corner probe of the cavity.

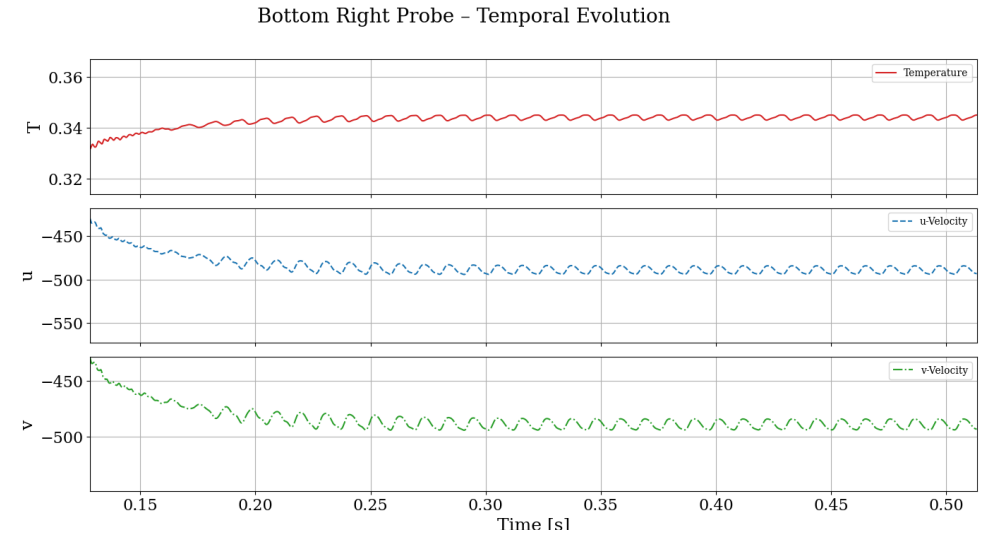


Figure 4: Temporal evolution of temperature and velocity components (u, v) at the bottom-right corner probe of the cavity.

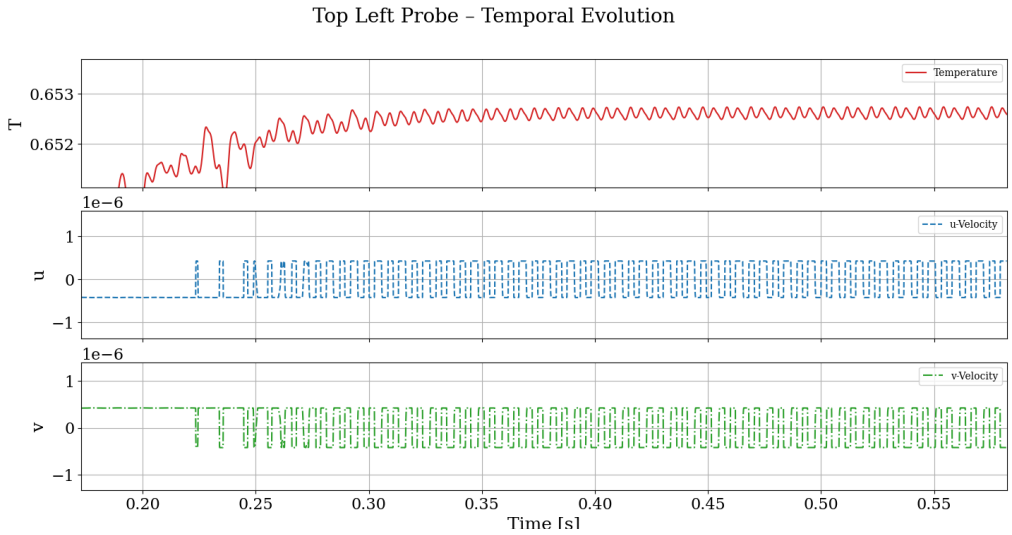


Figure 5: Temporal evolution of temperature and velocity components (u, v) at the top-left corner probe of the cavity.

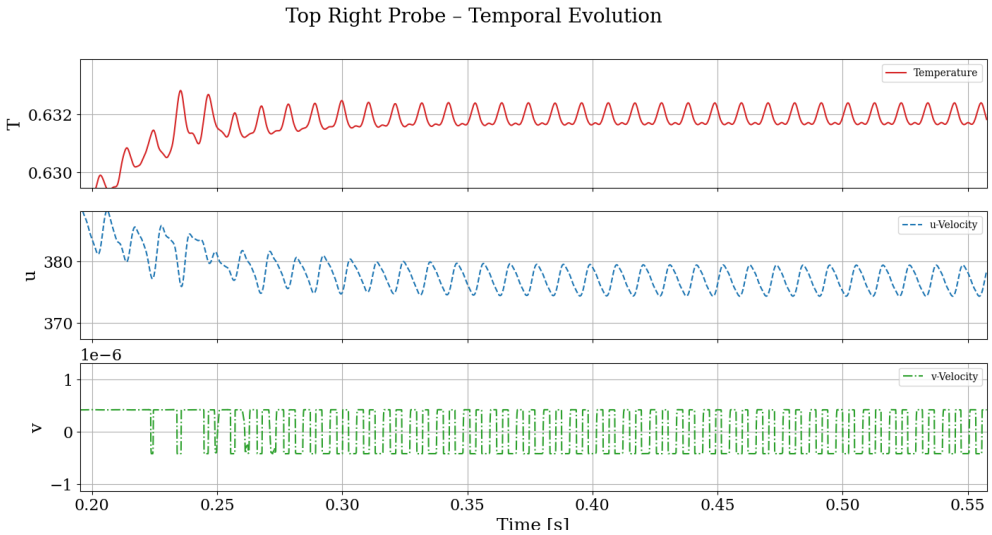


Figure 6: Temporal evolution of temperature and velocity components (u, v) at the top-right corner probe of the cavity.

By analyzing the probe plots, it is conservatively estimated that the system reaches a statistically stationary regime at approximately $t = 0.4\,\mathrm{s}$ for the case with $\mathrm{Ra} = 6.4 \times 10^8$. Consequently, in order to ensure a sufficiently long averaging window and obtain accurate time-averaged results, the simulation was extended up to 2 s. This duration was chosen because the fluid exhibited periodic fluctuations, as can be clearly seen from the plots, which do not require an excessively long time window to be properly averaged.
The same procedure was applied to determine the appropriate averaging intervals for the two additional test cases used for comparison: $\mathrm{Ra} = 1.0 \times 10^8$ and $\mathrm{Ra} = 1.07 \times 10^9$. The selected intervals are summarized in Table 1.

Rayleigh Number (Ra)	**Averaging Start Time** (t_0) [s]	**Total Simulation Time** (Δt) [s]
1.0×10^8	0.8	1.5
6.4×10^8	0.6	2.0
1.07×10^9	0.6	2.5

Table 1: Time intervals selected for averaging in each simulation case.

For the case with $\mathrm{Ra} = 1.0 \times 10^8$, after an initial transient phase, a moment of strong velocity instability was observed between 0.5 s and 0.6 s, followed by a sudden stabilization. Therefore, the time-averaging process was conservatively delayed until $t = 0.8$ s to ensure that only the statistically stationary regime was captured. Moreover, since $\mathrm{Ra} = 10^8$ does not yet represent a turbulent regime, the flow does not exhibit sustained oscillations but rather reaches a true steady state. For this reason, the simulation was stopped at $t = 1.5$ s, as no further temporal evolution was expected beyond that point.
On the other hand, for the case with $\mathrm{Ra} = 1.07 \times 10^9$, after the transient phase, the system reached an oscillatory regime that was less regular compared to the one observed at $\mathrm{Ra} = 6.4 \times 10^8$. The oscillations appeared to vary in shape but remained within a consistent range of values. This behavior allowed the selection of 0.6 s as the starting point for time-averaging. Due to the non-periodic nature of the fluctuations, the averaging window was extended to 2.5 s in order to better capture the fluid's dynamic behavior.

4 Post-Processed Quantities of Interest and Discussion

This section presents the main results obtained from the simulations, including the time-averaged fields of temperature and velocity, the stream function distributions, the computed average Nusselt number, velocity peaks and profiles along characteristic sections of the domain. Additionally, a detailed kinetic energy budget analysis is carried out to assess the energetic consistency of the numerical solver, and turbulence-related quantities such as the turbulent kinetic energy (TKE) and Reynolds stresses are evaluated.
The main simulation parameters and post-processing settings adopted for each Rayleigh number are summarized in Table 2. This includes mesh resolution, convective scheme, solver tolerance, time-averaging window, total simulated time, and computational effort associated with the Gauss–Seidel pressure solver.

Ra	Mesh size	Scheme	ϵ_{GS}	t_0 [s]	t_{sim} [s]	t_{exec} [h]	t_{GS} [h]	%GS
10^8	24×96	CDS	$1e^{-4}$	0.8	1.5	2.17	2.13	97.99%
6.4×10^8	24×96	CDS	$1e^{-4}$	0.6	2.0	5.17	5.05	97.57%
1.07×10^9	20×80	CDS	$1e^{-3}/1e^{-4}$	0.6	2.5	15.52	15.45	99.53%

Table 2: Summary of numerical and post-processing parameters for each Rayleigh number case. Mesh size refers to the number of control volumes used in the simulation. 'Scheme' indicates the convective discretization method. ϵ_{GS} is the convergence tolerance for the Gauss–Seidel pressure solver. t_0 is the time at which time-averaging is initiated. t_{sim} is the total physical time simulated, t_{exec} is the total execution time of the simulation, t_{GS} is the cumulative time spent in the Gauss–Seidel solver, and %GS is the percentage of total execution time consumed by the pressure solver.

As the Rayleigh number increases, both the flow intensity and the required time-averaging period increase. This leads to a substantial rise in total execution time, which is especially evident in the case of $Ra = 1.07 \times 10^9$. To manage computational cost, a coarser mesh (20×80) was adopted for this case. In addition, the convergence tolerance of the Gauss–Seidel pressure solver was initially relaxed to 10^{-3} during the transient phase. From $t_0 = 0.6\,\text{s}$ onward, when time-averaging begins, the tolerance was tightened to 10^{-4} to improve accuracy in the final averaged fields.
The table also shows that the pressure solver consistently dominates the total execution time, consuming over 97% of the simulation time in all cases. This confirms and highlights the need to implement a different, non-iterative pressure solver in future developments, in order to reduce the computational cost associated with enforcing mass conservation, especially at higher Rayleigh numbers and longer simulation times.

4.1 Time-Averaged Fields

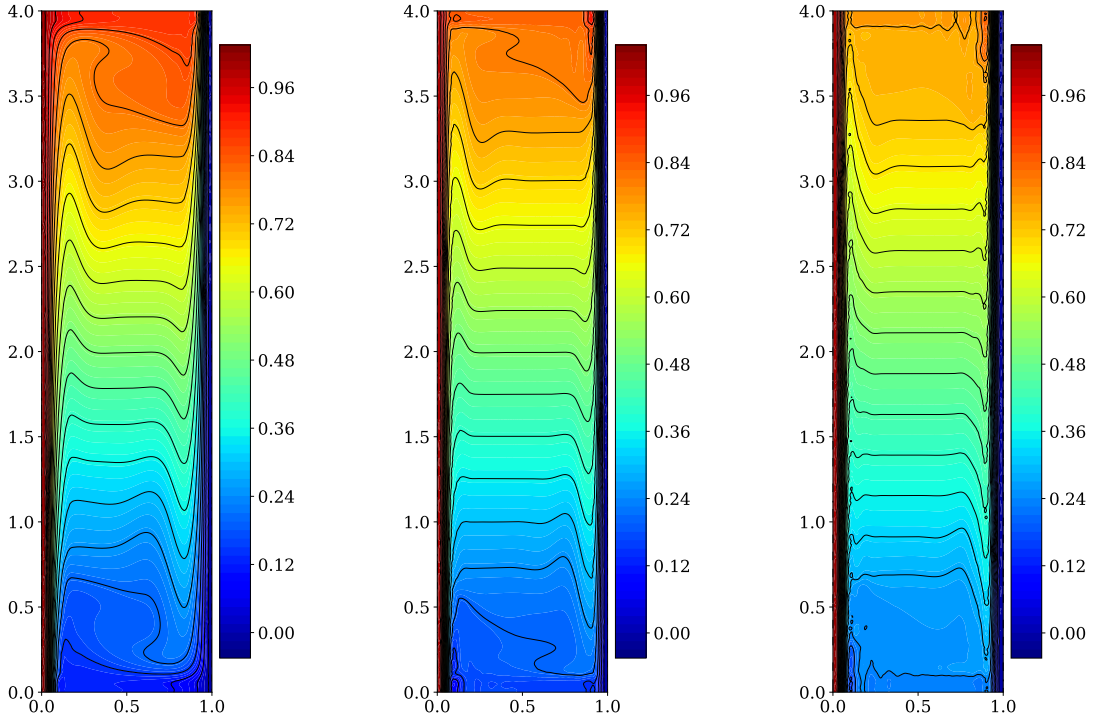
As previously discussed, in order to capture and visualize the quasi-turbulent behavior of the flow, the relevant physical quantities must be averaged over time. Given a generic scalar or vector field $\phi(x, y, t)$, the time-averaged field is defined as:

$$\overline{\phi}(x, y) = \frac{1}{\Delta t} \int_{t_0}^{t_0+\Delta t} \phi(x, y, t)\, dt \tag{26}$$

where t_0 represents the time at which the averaging process begins, and Δt is the time window over which the averaging is performed. This interval must be sufficiently long to cover the dominant temporal oscillations of the system, once the initial transient has decayed.
In the implementation, Equation 26 is approximated by accumulating the instantaneous values of the field, multiplied by the relative time step, at each iteration within the averaging interval, and subsequently dividing the result by the total seconds of the averaging window to obtain the average field.

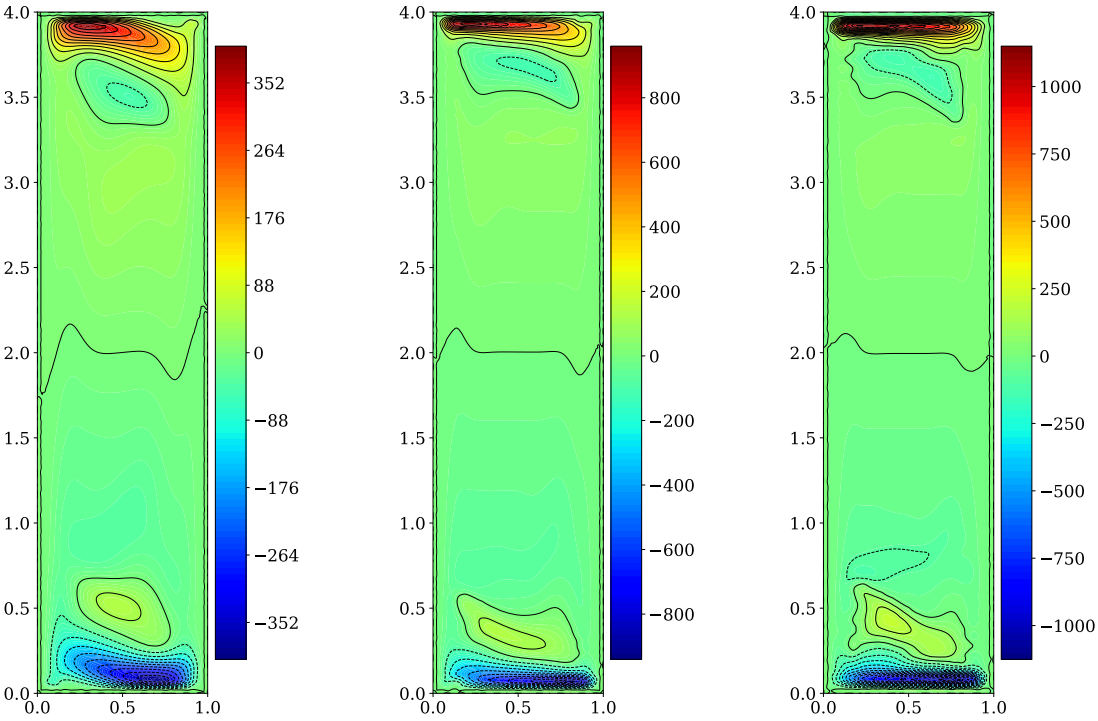
Data Output and Visualization

The computed time-averaged fields are exported in `.txt` files and visualized using a dedicated Python script. Below, the colormaps of the time-averaged temperature and velocity components are presented for the three cases with different Rayleigh numbers.



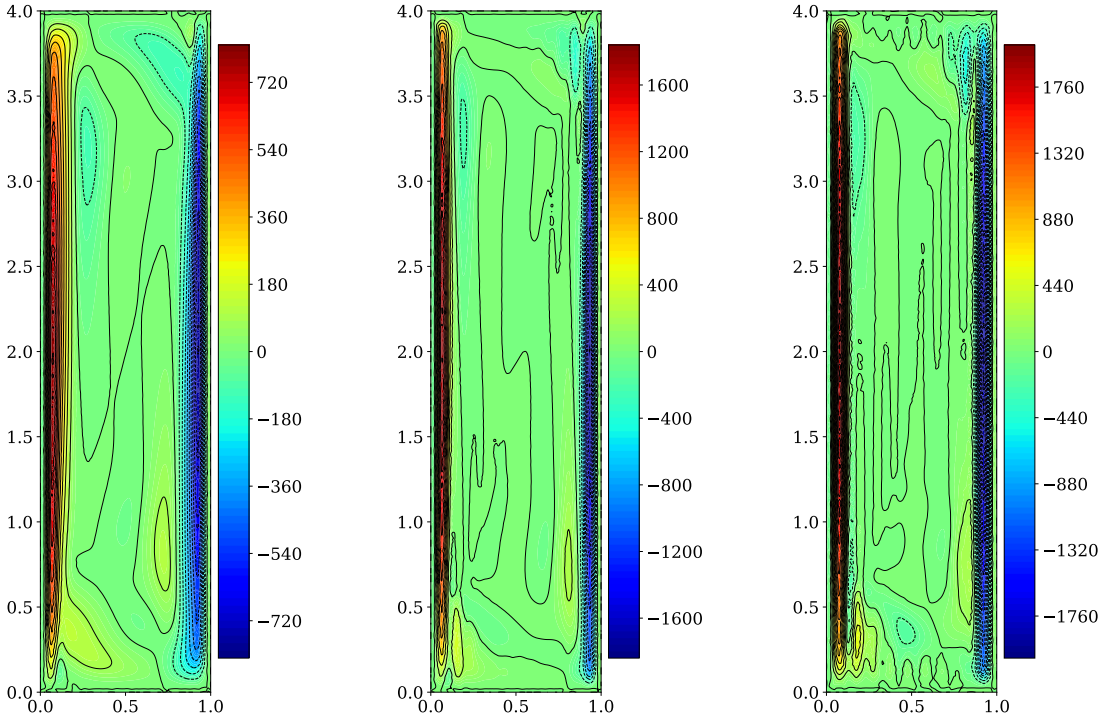
(a) $Ra = 10^8$ (b) $Ra = 6.4 \times 10^8$ (c) $Ra = 1.07 \times 10^9$

Figure 7: Colormaps of time-averaged temperature fields $\overline{T}(x, y)$ for different Rayleigh numbers.



(a) $Ra = 10^8$ (b) $Ra = 6.4 \times 10^8$ (c) $Ra = 1.07 \times 10^9$

Figure 8: Colormaps of time-averaged horizontal velocity fields $\overline{u}(x, y)$ for different Rayleigh numbers.



(a) $Ra = 10^8$ (b) $Ra = 6.4 \times 10^8$ (c) $Ra = 1.07 \times 10^9$

Figure 9: Colormaps of time-averaged vertical velocity fields $\overline{v}(x, y)$ for different Rayleigh numbers.

At $Ra = 10^8$, the temperature field exhibits smooth isotherms and relatively thicker thermal boundary layers, characteristic of an organized convective regime. The velocity fields support this behavior: $\overline{u}(x, y)$ is confined to symmetric recirculation zones near the top and bottom walls, while $\overline{v}(x, y)$ is concentrated along the vertical boundaries, forming laminar jet structures.
For $Ra = 6.4 \times 10^8$ and $Ra = 1.07 \times 10^9$, the temperature field shows a thinner boundary layer, increased waviness and localized distortions in the isotherms, especially in the corners of the cavity, indicating the development of small-scale unsteady motions. Recirculations in vertical and horizontal components of the velocity become more evident and irregular, corresponding to the growth of secondary convection cells and unsteady flow structures. The vertical velocity jets become more pronounced, and irregular fluctuations appear throughout the domain, reflecting the onset of flow destabilization and enhanced vertical mixing.
It is important to note that, due to the high computational cost, the simulation at $Ra = 1.07 \times 10^9$ was carried out with a coarser spatial grid. This limits the resolution of fine-scale structures but still captures the main features of the turbulent regime.

4.2 Stream Function

To visualize the flow patterns and highlight the vortical structures of the velocity field, the stream function $\psi(x, y)$ was computed from the time-averaged velocity components $\overline{u}$ and $\overline{v}$.
According to the classical definition provided by Lamb and Batchelor [2], the stream function ψ is defined as the line integral:

$$\psi(x, y, t) = \int_A^P (\overline{u}\ dy - \overline{v}\ dx) \tag{27}$$

where A is a reference point at which the stream function is set to zero, and P is the point of interest. The stream function thus represents the volumetric flux through a surface perpendicular to the plane of flow, per unit thickness.
Assuming an infinitesimal displacement $dP = (dx, dy)$, the corresponding variation in stream function is:

$$d\psi = \overline{u}\ dy - \overline{v}\ dx \tag{28}$$

Comparing this with the exact differential form:

$$d\psi = \frac{\partial \psi}{\partial x} dx + \frac{\partial \psi}{\partial y} dy, \tag{29}$$

we obtain the following expressions for the velocity components in terms of the stream function:

$$\overline{u} = \frac{\partial \psi}{\partial y}, \quad \overline{v} = -\frac{\partial \psi}{\partial x} \tag{30}$$

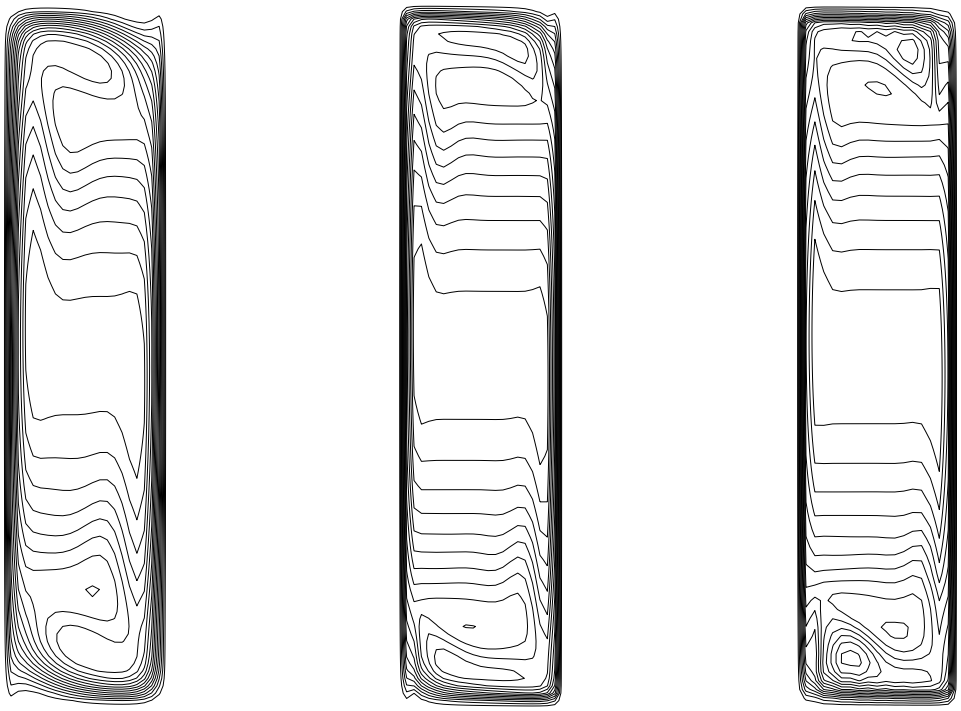
This formulation ensures that the incompressibility condition $\nabla \cdot \mathbf{v} = 0$ is automatically satisfied, since:

$$\frac{\partial \overline{u}}{\partial x} + \frac{\partial \overline{v}}{\partial y} = \frac{\partial^2 \psi}{\partial x \partial y} - \frac{\partial^2 \psi}{\partial y \partial x} = 0 \tag{31}$$

The stream function was numerically evaluated on the same structured grid as the pressure/temperature, assuming a uniform spacing $\Delta x = \Delta y$ and interpolating velocity components to the cell center usind a CDS scheme. The integration was carried out in two steps: first, a vertical integration was performed along the first left column of the domain by summing the horizontal velocity component from bottom to top, setting the bottom-left corner value to zero. Then, for each row, a horizontal integration from left to right was applied using the vertical velocity component, allowing the complete reconstruction of the stream function field across the domain.

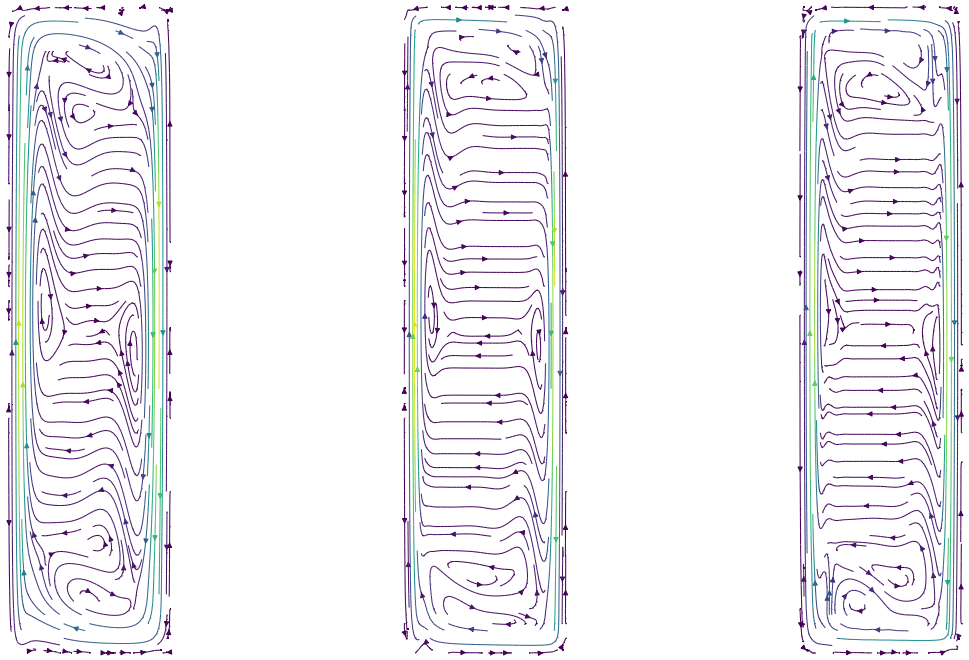
Data Output and Visualization

After computing the stream function at all grid points, the values are exported to a text file. The file is then post-processed using `Python` to plot contour lines corresponding to constant values of ψ. Figures 10, 11 and 12 present the stream function visualizations for the three Rayleigh number cases. For each case, three types of plots are displayed: contour lines, streamlines, and colormap representations of the stream function $\psi(x, y)$, computed from the time-averaged velocity fields.



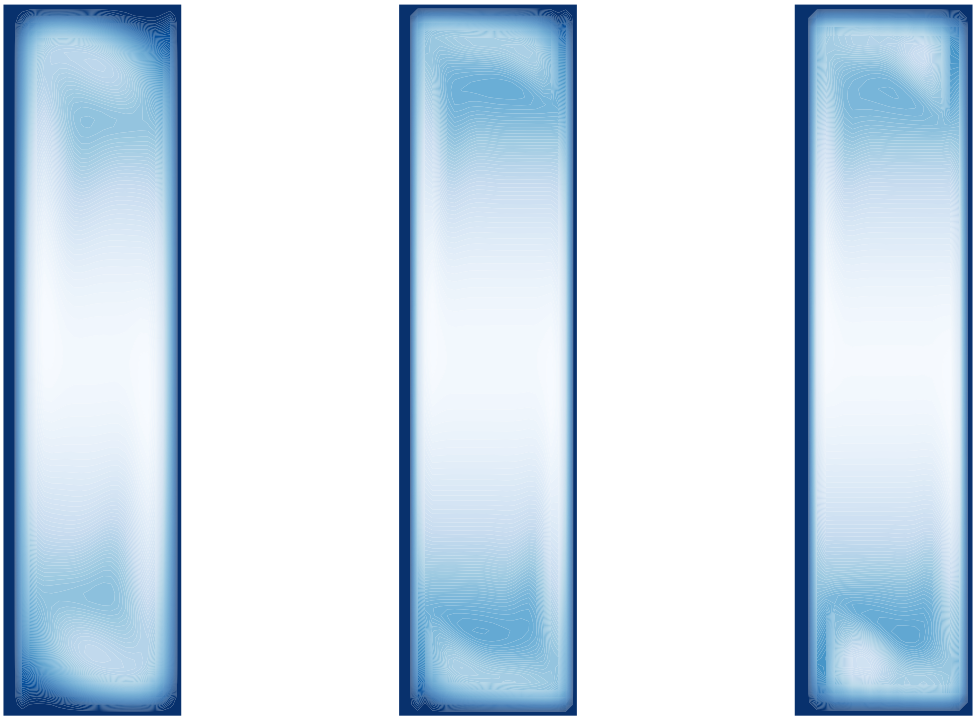
(a) $Ra = 10^8$ (b) $Ra = 6.4 \times 10^8$ (c) $Ra = 1.07 \times 10^9$

Figure 10: Contour plots of the stream function computed from time-averaged velocity fields, for different Rayleigh numbers.



(a) $Ra = 10^8$ (b) $Ra = 6.4 \times 10^8$ (c) $Ra = 1.07 \times 10^9$

Figure 11: Streamline plots of the stream function based on time-averaged velocity fields, for different Rayleigh numbers.



(a) $Ra = 10^8$ (b) $Ra = 6.4 \times 10^8$ (c) $Ra = 1.07 \times 10^9$

Figure 12: Colormap visualizations of the stream function based on time-averaged velocity fields, for different Rayleigh numbers.

Note that the stream function is defined up to an arbitrary constant. Here, it is conventionally set to zero at the bottom-left corner.
Figure 10 show the initial formation of Tollmien–Schlichting-like waves, which disturb the boundary layers and eject large unsteady eddies into the core of the cavity. These eddies enhance mixing by transporting hot or cold fluid away from the isothermal walls, leading to nearly uniform temperature regions near the top and bottom boundaries, as observed in Figure 7.
According to theoretical expectations, the location where this phenomenon begins should move upstream as the Rayleigh number increases. A first indication of this behavior can be seen in Figure 10, although the vertical shift is not clearly visible at the moderate Rayleigh numbers considered here. This effect is more evident in the work of Trias et al. [14], where higher turbulent Rayleigh numbers were investigated.
The mixing process also forces the temperature drop in the core of the cavity to occur in a smaller region, but this effect is, again, not clearly visible in the current results.
Moreover, Figure 11 presents the time-averaged velocity vector fields, which help visualize the flow direction and identify the structure of convection cells. At $Ra = 10^8$, the flow is dominated by a large primary convection cell spanning the full height of the cavity, characteristic of a laminar regime. Although the flow is not yet turbulent, the high aspect ratio of the cavity induces the appearance of localized recirculations. In particular, two symmetric lateral recirculations are observed near the vertical walls at mid-height. These structures arise from the interaction between the buoyancy-driven vertical jets and the viscous shear imposed by the no-slip boundary conditions. As Ra increases to 6.4×10^8, the lateral recirculations become thinner and more energetic, while the primary vortex starts to deform. Secondary vortical structures begin to emerge, especially near the

corners of the cavity. The streamlines become increasingly irregular, indicating the onset of complex unsteady behavior and the transition toward a more mixed and potentially turbulent regime.

4.3 Nusselt Number, Maximum Velocities and Mid-Section Profiles

To assess the thermal and fluid dynamic behavior of the cavity under natural convection, a quantitative analysis is carried out including the evaluation of the average Nusselt number, the identification of maximum velocities and their spatial locations, as well as the velocity profiles along the mid-height and mid-width sections of the domain.
To quantify the convective heat transfer within the rectangular cavity, the average Nusselt number is computed from the local dimensionless heat flux along vertical lines across the domain. The nondimensional formulation adopts H as the reference length for both spatial coordinates and velocity components. The following nondimensional variables are defined:

$$x^* = \frac{x}{H}, \quad y^* = \frac{y}{H}, \quad T^* = \frac{T - T_{\text{cold}}}{T_{\text{hot}} - T_{\text{cold}}}, \quad u^* = \frac{uH}{\alpha}, \quad v^* = \frac{vH}{\alpha}, \quad t^* = \frac{t\alpha}{H^2}, \quad \dot{q}_x^* = \frac{\dot{q}_x H}{\alpha \Delta T}$$

The local heat flux in the x-direction is defined as:

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

and the corresponding local Nusselt number is obtained by integrating this quantity over the vertical direction:

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

In the numerical procedure, this integral is discretized as a summation over vertical control volumes. The convective term is interpolated using either UDS or CDS schemes, and the temperature gradient is approximated with finite differences. This computation is repeated along multiple vertical sections, and the average Nusselt number is finally obtained as the arithmetic mean of all local contributions.
It is important to note that the direction of integration affects the physical meaning of the Nusselt number. In a differentially heated cavity, where the temperature gradient is applied horizontally between the hot left wall and the cold right wall, heat transfer mainly occurs in the x-direction.
For this reason, computing the local Nusselt number along vertical lines is the most appropriate, as it measures the heat flux across sections aligned with the imposed gradient. Alternatively, integrating along horizontal lines (at fixed y^*) would capture vertical heat transport due to convection, such as rising plumes or recirculation zones. However, this does not match the standard definition of the Nusselt number for this configuration.
Table 3 reports the computed average Nusselt number $\overline{Nu}$, as well as the maximum dimensionless velocities (u^* and v^*) and their corresponding positions inside the cavity for each Rayleigh number investigated.

Ra	$\overline{Nu}$	Max u^*	y/H at $x = 0.5L$	Max v^*	x/L at $y = 0.5H$
10^8	27.36	786.15	0.97	2130.52	0.02
6.4×10^8	37.10	1935.37	0.98	4978.67	0.02
1.07×10^9	34.79	2695.29	0.98	5546.49	0.02

Table 3: Summary of computed global quantities: average Nusselt number $\overline{Nu}$, maximum dimensionless horizontal velocity u^* at mid-width, and maximum dimensionless vertical velocity v^* at mid-height, with their respective positions.

Table 3 reports the main global quantities computed for the three Rayleigh numbers. As expected, the average Nusselt number $\overline{Nu}$ increases significantly between $Ra = 10^8$ and $Ra = 6.4 \times 10^8$, confirming that convective heat transfer becomes more efficient with stronger thermal forcing. For the highest Rayleigh number, $Ra = 1.07 \times 10^9$, a slight decrease in $\overline{Nu}$ is observed, which can be attributed to the use of a coarser spatial discretization compared to the other cases. Furthermore, in this simulation, the tolerance for the Gauss–Seidel pressure solver was initially set to 10^{-3}, and only reduced to 10^{-4} from the beginning of the averaging phase onward. These factors may have affected the accuracy of the computed time-averaged quantities.
The maximum values of horizontal and vertical velocity, u^* and v^*, increase with Rayleigh number, indicating the presence of stronger flow structures. The horizontal velocity peaks are located near the upper part of the cavity ($y/H \approx 0.97{-}0.98$), while the maximum vertical velocities are found close to the sidewalls at mid-height ($x/L \approx 0.02$). This distribution of peak velocities matches the typical flow pattern found in differentially heated cavities, where the hot and cold vertical walls produce narrow upward and downward plumes, connected by horizontal flow near the top and bottom boundaries.
In addition to the maximum values, velocity profiles along the vertical line at mid-width ($x = 0.5L$) and the horizontal line at mid-height ($y = 0.5H$) have been included to provide further insight into the development of boundary layers and the internal flow structure.

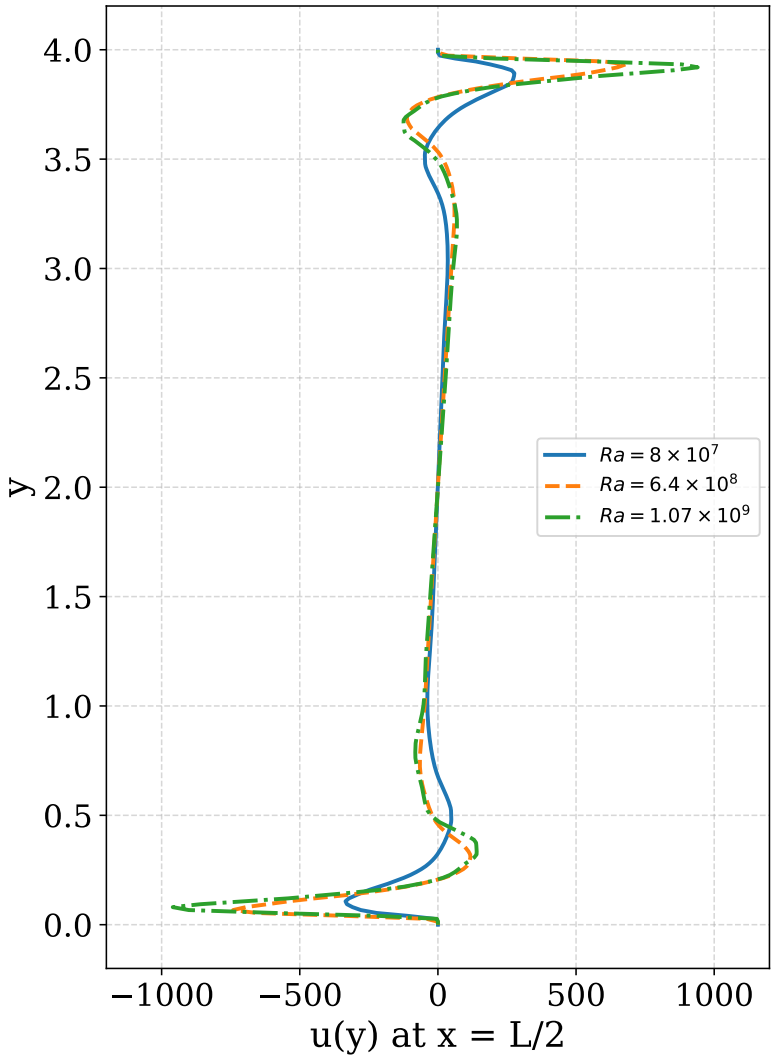


Figure 13: Horizontal velocity profiles $u(y)$ at $x = 0.5L$ for three different Rayleigh numbers.

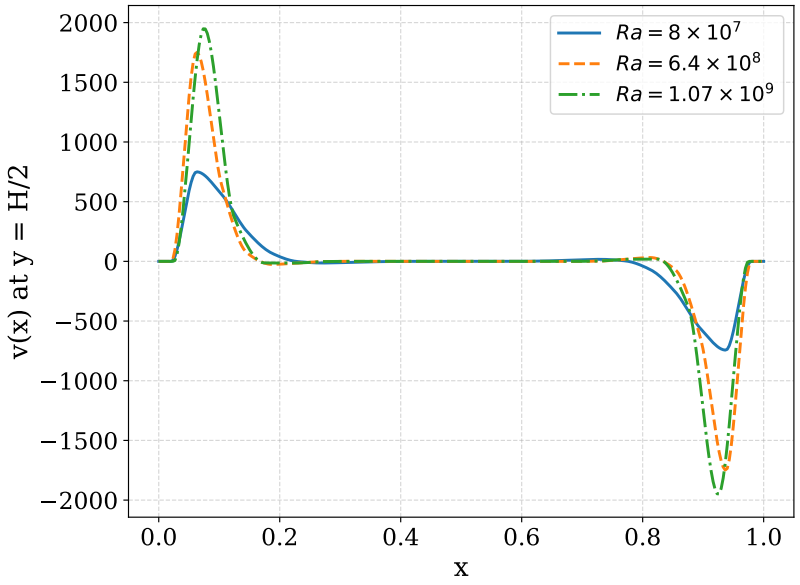


Figure 14: Vertical velocity profiles $v(x)$ at $y = 0.5H$ for three different Rayleigh numbers.

The velocity profiles at the cavity mid-sections provide important insight into the flow development as the Rayleigh number increases. The vertical velocity profiles $v(x)$ at mid-height ($y = 0.5H$) show that the main ascending and descending plumes near the hot and cold walls become narrower and shift towards the cavity center for higher Ra. This behavior suggests that the buoyant plumes are no longer strictly confined to the thermal boundary layers along the vertical walls, but instead begin to detach and move into the core region. Similar observations have been reported in previous DNS studies, such as [15].
At the same time, the horizontal velocity profiles $u(y)$ at the vertical mid-plane ($x = 0.5L$) show the formation of strong horizontal jets near the bottom and top boundaries. These jets grow in magnitude with increasing Ra and are associated with the horizontal transport of fluid between the ascending and descending plumes. The velocity profiles also become steeper near the walls, indicating thinner boundary layers and stronger shear. In the central region, the horizontal velocity remains close to zero, consistent with the recirculating nature of the cavity flow.
Overall, these changes in the velocity profiles highlight the transition from moderate to vigorous convection, characterized by thin boundary layers, strong vertical and horizontal jets, and plume detachment from the walls.

4.4 Kinetic Energy Budget and Turbulent Flow Quantities

In order to quantify the dynamic behavior of the flow inside the cavity and validate the energetic consistency of the simulation, a complete kinetic energy analysis was carried out. This includes both the instantaneous energy balance and the evaluation of turbulence-related quantities, based on the velocity fields and their temporal evolution.
The instantaneous kinetic energy of the system is defined as:

$$E_k(t) = \frac{1}{2} \int_\Omega \left(u^2 + v^2\right) \, d\Omega \tag{32}$$

In the discrete framework, this becomes:

$$E_k(t) = \frac{1}{2}\rho \sum_{i,j} \left(u_c^2 + v_c^2\right) \, \Delta x^2 \tag{33}$$

where u_c and v_c are the velocity components interpolated at the center of each control volume.
To assess the evolution of kinetic energy over time, the individual physical contributions to its rate of change are computed at each time step:

- **Viscous diffusion** (R_{diff}) is evaluated as $\mu\,\mathbf{v} \cdot \nabla^2\mathbf{v}$, using a discrete Laplacian operator.
- **Convective transport** (R_{conv}) corresponds to $-\rho\,\mathbf{v}\cdot(\mathbf{v}\cdot\nabla)\mathbf{v}$, discretized via central differences.
- **Pressure work** (R_{press}) is computed as $-\nabla p \cdot \mathbf{v}$, again using central differences.
- **Buoyancy production** (R_{buoy}) represents the work done by buoyancy forces under the Boussinesq approximation, calculated as $\rho\beta(T - T_\infty)\,\mathbf{g} \cdot \mathbf{v}$.

The theoretical rate of kinetic energy change is then given by:

$$\left.\frac{dE_k}{dt}\right|_{\text{theory}} = R_{\text{diff}} + R_{\text{conv}} + R_{\text{press}} + R_{\text{buoy}} \tag{34}$$

Simultaneously, the numerical time derivative is computed using finite differences:

$$\left.\frac{dE_k}{dt}\right|_{\text{numeric}} = \frac{E_k(t + \Delta t) - E_k(t)}{\Delta t} \tag{35}$$

The consistency of the energy balance is verified by monitoring the instantaneous error:

$$\text{Error}(t) = \left| \left.\frac{dE_k}{dt}\right|_{\text{theory}} - \left.\frac{dE_k}{dt}\right|_{\text{numeric}} \right| \tag{36}$$

This methodology not only ensures that the numerical solution remains physically consistent over time, but also allows to assess the approach to a statistically steady state. When the time-averaged residual $|\mathcal{R}_{\text{tot}} - dE_k/dt|$ falls below a predefined threshold, the flow can be considered statistically stationary, enabling the extraction of averaged turbulent quantities.

Validation of the Solver via Energy Budget (Case: Ra = 10^8)

To validate the implementation of the energy budget, a detailed analysis was carried out for the test case with Rayleigh number Ra = 10^8. The following non-dimensional quantities were monitored at the end of the transient (time $t \approx 1.5$ s):

Symbol	**Value**	**Physical Meaning**
t	1.49997	Non-dimensional time
E_k	54894.7	Total kinetic energy
R_{diff}	-4.734×10^7	Viscous diffusion (dissipation)
R_{conv}	$+2.41 \times 10^5$	Convective transport (net energy flux)
R_{press}	-1.04×10^5	Pressure work
R_{buoy}	$+4.742 \times 10^7$	Buoyancy energy input
R_{tot}	$+2.21 \times 10^5$	Total theoretical RHS
$\left.\frac{dE_k}{dt}\right\|_{\text{num}}$	0	Numerical time derivative
Error	2.21×10^5	Energy budget residual

Table 4: Instantaneous values of the non-dimensional energy budget terms for the test case $\text{Ra} = 10^8$ at time $t = 1.49997$. All quantities are expressed in non-dimensional form.

The numerical derivative of kinetic energy being zero confirms that the system has reached a statistically steady state. The residual error is:

$$\frac{\text{Error}}{R_{\text{buoy}}} \approx \frac{2.21 \times 10^5}{4.742 \times 10^7} \approx 0.0047$$

This small residual (about 0.47%) is consistent with the expected numerical truncation and interpolation errors. Moreover, the physical balance is well preserved:

- $R_{\text{buoy}} > 0$: buoyancy injects energy into the system, as expected.
- $R_{\text{diff}} < 0$: viscous forces dissipate energy, confirming physical realism.
- $R_{\text{conv}} > 0$: minor net convective transport.
- $R_{\text{press}} < 0$: minor pressure forces redistribution.

The dominant terms, buoyancy production and viscous dissipation, nearly cancel each other out, resulting in a net rate of energy change that is close to zero, as expected in a statistically steady regime.
In conclusion, the energy budget confirms the physical validity and numerical consistency of the solver.

Turbulent Kinetic Energy Analysis and Reynolds Stresses

In the context of Direct Numerical Simulation (DNS), the turbulent kinetic energy (TKE) and Reynolds stresses can be computed directly from the instantaneous flow field, without relying on turbulence models. These quantities are essential to characterize the energy transfer mechanisms and anisotropic structures arising in turbulent convection.
The turbulent kinetic energy per unit mass is defined as:

$$TKE(x, y) = \frac{1}{2}\left(\overline{u'^2} + \overline{v'^2}\right) \tag{37}$$

The fluctuating components of velocity are obtained by subtracting the square of the mean from the mean of the square:

$$\overline{u'^2} = \overline{u^2} - \overline{u}^2, \qquad \overline{v'^2} = \overline{v^2} - \overline{v}^2 \tag{38}$$

where:

- $\overline{u}, \overline{v}$ are the time-averaged velocity components;
- $\overline{u^2}, \overline{v^2}$ are the time-averaged squares of the velocity components;
- all averages are computed over a time window Δt, once the system reaches a statistically stationary regime.

The Reynolds stress tensor describes the turbulent transport of momentum and is defined as:

$$\tau_{ij}^{\mathrm{Re}} = -\rho\, \overline{u_i' u_j'} \tag{39}$$

In a 2D case, the relevant components become:

$$\overline{u'^2} = \overline{u^2} - \overline{u}^2 \tag{40}$$

$$\overline{v'^2} = \overline{v^2} - \overline{v}^2 \tag{41}$$

$$\overline{u'v'} = \overline{uv} - \overline{u} \cdot \overline{v} \tag{42}$$

The numerical evaluation of TKE and Reynolds stresses is performed over a structured mesh indexed by (i, j). The computation follows these steps:

- First, accumulate temporally integrated fields during the averaging window:

$$\begin{aligned}
u_{\mathrm{avg}}[i][j] &= \frac{1}{\Delta t} \sum u[i][j] \cdot dt \\
uu_{\mathrm{avg}}[i][j] &= \frac{1}{\Delta t} \sum u[i][j]^2 \cdot dt \\
v_{\mathrm{avg}}[i][j] &= \frac{1}{\Delta t} \sum v[i][j] \cdot dt \\
vv_{\mathrm{avg}}[i][j] &= \frac{1}{\Delta t} \sum v[i][j]^2 \cdot dt \\
uv_{\mathrm{avg}}[i][j] &= \frac{1}{\Delta t} \sum u[i][j] \cdot v[i][j] \cdot dt
\end{aligned}$$

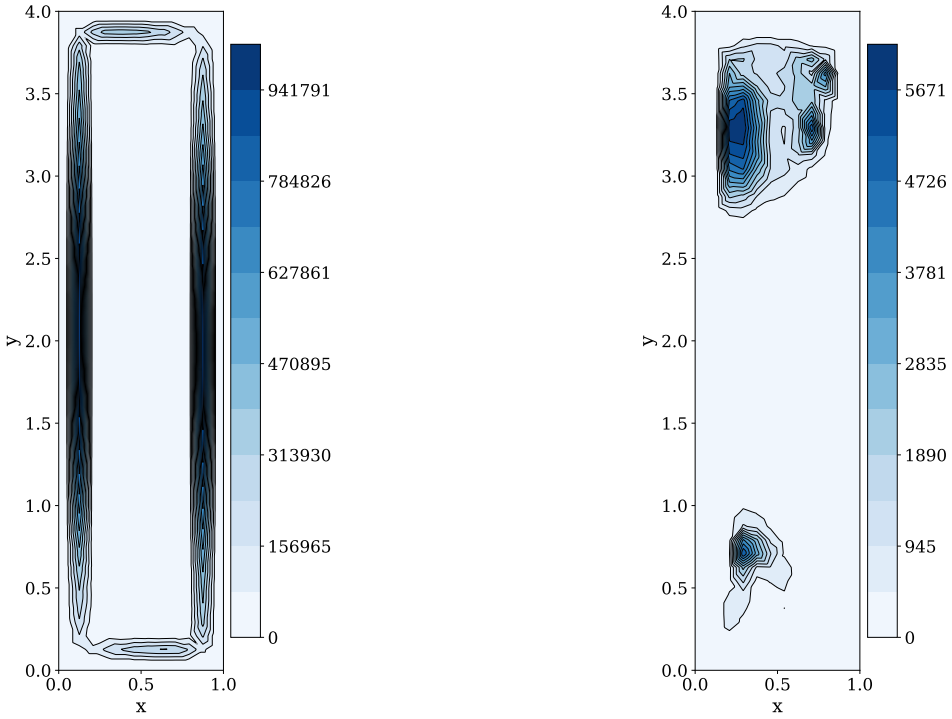
- Then compute the second-order moments of fluctuations:

$$\begin{aligned}
\overline{u'^2} &= uu_{\mathrm{avg}} - u_{\mathrm{avg}}^2 \\
\overline{v'^2} &= vv_{\mathrm{avg}} - v_{\mathrm{avg}}^2 \\
\overline{u'v'} &= uv_{\mathrm{avg}} - u_{\mathrm{avg}} \cdot v_{\mathrm{avg}} \\
TKE &= \frac{1}{2}\left(\overline{u'^2} + \overline{v'^2}\right)
\end{aligned}$$

To complement the analysis, the mean kinetic energy (KE) of the time-averaged flow was also computed as:

$$\mathrm{KE} = \frac{1}{2}\left(\overline{u}^2 + \overline{v}^2\right) \tag{43}$$

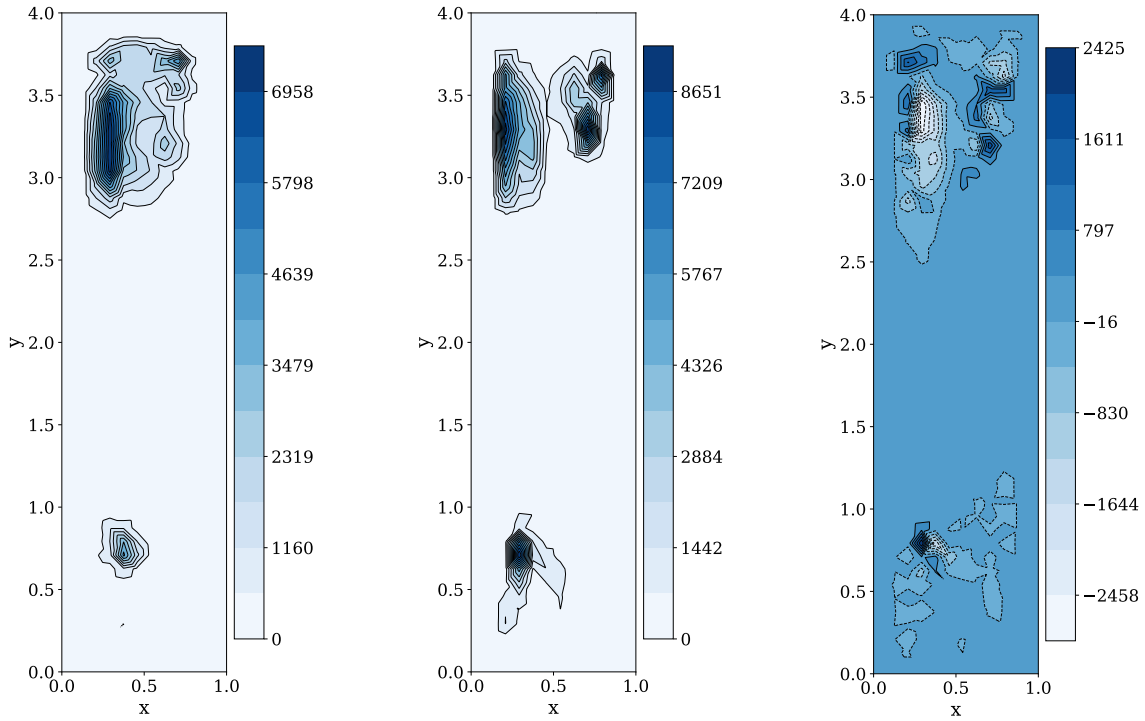
Here, the colormaps of the computed fields are presented and analyzed for the case $\mathrm{Ra} = 1.07 \times 10^9$. It is important to note that the results were obtained using a coarser mesh (18x72 control volumes) and over a short simulation time (2 s). Therefore, while the results are not highly accurate, they provide a reasonable first approximation and display reliable flow features. These features have been qualitatively compared with the 3D results reported in the study by Trias [14].



(a) Mean Kinetic Energy (KE)

(b) Turbulent Kinetic Energy (TKE)

Figure 15: Comparison of mean and turbulent kinetic energy distributions for the case $Ra = 1.07 \times 10^9$, computed on a coarse 10×40 grid.



(a) Reynolds stress $\overline{u'^2}$ (b) Reynolds stress $\overline{v'^2}$ (c) Shear stress $\overline{u'v'}$

Figure 16: Reynolds stress components for the case $Ra = 1.07 \times 10^9$, showing the distribution of turbulent velocity fluctuations and shear stress in the domain.

By comparing the spatial distributions of KE and TKE, one can identify the regions where the flow is primarily organized (high KE) or where turbulence and unsteady fluctuations dominate (high TKE). Regions with high $\overline{u'v'}$ indicate strong coupling between vertical and horizontal momentum transfer, typically associated with thermal plumes and large-scale convective structures.

Figures 15 and 16 present the computed distributions of kinetic energy and Reynolds stresses for the case $Ra = 1.07 \times 10^9$, based on a coarse 18×72 grid.

As expected, the mean kinetic energy (KE) field exhibits a highly symmetric structure, with two intense vertical layers located near the side walls. These regions correspond to the strong vertical jets associated with the main convective circulation, where the velocity magnitudes are highest. The central part of the domain remains nearly motionless in the time-averaged sense, resulting in low kinetic energy values.

In contrast, the turbulent kinetic energy (TKE) and the Reynolds stress components $\overline{u'^2}$, $\overline{v'^2}$, and $\overline{u'v'}$ show noticeable asymmetries. In particular, the turbulent activity appears to develop in the upper part of the cavity, while the lower region lacks a symmetric counterpart. This imbalance is likely due to the limited temporal resolution of the simulation. The coarse mesh may not fully capture small-scale turbulent fluctuations near the bottom wall, and the relatively short time-averaging window might be insufficient to reach statistical convergence. These factors can lead to underestimation of fluctuation intensities in certain regions, resulting in non-physical asymmetries in the TKE and Reynolds stress fields.

Despite these limitations, the regions of peak turbulent intensity are consistent with the areas where large-scale unsteady structures were observed in the streamline plots, and they show similar patterns and magnitudes to those reported by Trias in a comparable

two-dimensional configuration [15].

5 Conclusions

The study successfully simulated natural convection in a tall, differentially heated cavity at high Rayleigh numbers using a two-dimensional DNS approach. The results revealed:

- A clear transition from steady to unsteady convective behavior as Ra increases from 10^8 to 1.07×10^9, with signs of quasi-turbulent flow structures.
- Time-averaged fields and velocity profiles confirmed the emergence of boundary layer thinning, vortex deformation, and intensified plume motion.
- The kinetic energy budget validated the energetic consistency of the solver, with buoyancy injection and viscous dissipation being in near balance.
- The turbulent kinetic energy and Reynolds stress analysis revealed anisotropic and spatially inhomogeneous turbulent features.
- The Gauss–Seidel pressure solver accounted for more than 97% of total computational time, emphasizing the need for more efficient solvers in future developments.

While the simulations captured relevant physical behaviors of high-Rayleigh natural convection, further improvements are recommended. These include grid refinement, implementation of faster solvers, and more advanced statistical analysis to better resolve and quantify turbulent structures.

References

[1] H. Tennekes and J. L. Lumley, *A First Course in Turbulence*, MIT Press, 1972.

[2] H. Lamb and G. K. Batchelor, *Hydrodynamics*, 6th ed., Cambridge University Press, 1932.

[3] A. Bejan, *Convection Heat Transfer*, 4th ed., John Wiley & Sons, 2013.

[4] P. Le Quere and M. Behnia, "From onset of unsteadiness to chaos in a differentially heated square cavity," *J. Fluid Mech.*, vol. 359, pp. 81–107, Mar. 1998.

[5] S. Xin and P. Le Quere, "Direct numerical simulations of two-dimensional chaotic natural convection in a differentially heated cavity of aspect ratio 4," *J. Fluid Mech.*, vol. 304, pp. 87–118, Dec. 1995.

[6] E. Jansen and R. A. W. M. Henkes, "Transition to three-dimensional natural convection flows in enclosures," *Int. J. Heat Fluid Flow*, vol. 16, no. 4, pp. 317–325, 1995.

[7] T. Fusegi, J. M. Hyun, K. Kuwahara, and B. Farouk, "Numerical solutions of two-dimensional natural convection in a differentially heated square cavity," *Int. J. Heat Mass Transf.*, vol. 34, no. 6, pp. 1543–1557, 1991.

[8] R. A. W. M. Henkes and C. J. Hoogendoorn, "Natural convection in a square cavity calculated with low-Reynolds-number turbulence models," *Int. J. Heat Mass Transf.*, vol. 37, pp. 313–325, 1994.

[9] G. Labrosse, S. Xin, and P. Le Quere, "Non-symmetric effects in 3D natural convection in a cubic cavity," *Int. J. Heat Fluid Flow*, vol. 19, no. 4, pp. 387–396, 1998.

[10] R. A. W. M. Henkes and P. Le Quere, "Transition and three-dimensional effects in natural convection in differentially heated cavities," *Int. J. Heat Mass Transf.*, vol. 40, no. 2, pp. 319–333, 1997.

[11] S. Paolucci and D. R. Chenoweth, "Direct numerical simulation of two-dimensional turbulent natural convection," *J. Fluid Mech.*, vol. 201, pp. 379–410, 1989.

[12] S. Paolucci, "Direct numerical simulation of two-dimensional turbulent natural convection in an enclosure," *Phys. Fluids A*, vol. 2, no. 12, pp. 2091–2103, 1990.

[13] H. Mlaouah, M. Vynnycky, and J. R. King, "Non-Boussinesq effects on natural convection in a square cavity," *Int. J. Heat Mass Transf.*, vol. 49, nos. 11–12, pp. 2045–2060, 2006.

[14] F. X. Trias, O. Lehmkuhl, A. Oliva, M. Soria, and C. D. Perez-Segarra, "Direct numerical simulation of a differentially heated cavity of aspect ratio 4 with Rayleigh numbers up to 10^{11}," *Int. J. Heat Mass Transf.*, vol. 53, nos. 3–4, pp. 665–673, 2010.

[15] F. X. Trias, M. Soria, C. D. Perez-Segarra, and A. Oliva, "Direct numerical simulation of a three-dimensional natural convection flow in a differentially heated cavity of aspect ratio 4," *Numer. Heat Transf. A Appl.*, vol. 45, no. 7, pp. 649–673, 2004.

[16] J. Soria, M. Hossain, P. Betts, and L. Djenidi, "Direct numerical simulation of a 3D differentially heated cavity at Rayleigh number 10^9," in *Proc. 15th Australasian Fluid Mech. Conf.*, 2004.

[17] A. T. Berkovsky and V. B. Polevikov, "Numerical investigation of two-dimensional convection in a closed rectangular cavity," in *Natural Convection: Fundamentals and Applications*, S. Kakac, R. K. Shah, and W. Aung, Eds., Hemisphere Publishing, 1985, pp. 323–352.

[18] L. Davidson, *An Introduction to Turbulence Models*, Dept. of Thermo and Fluid Dynamics, Chalmers University of Technology, Publication 97/2, 2022.

A C++ Main Code

```
#include "DHC_Turbulent.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, double dystar) = 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 H, double alpha, double dx, double
            &Nux_avg,
                vector<vector<double>> &T1star, vector<vector<double>> &T_avg, vector<vector<double>> &qx
                    ,
                vector<vector<double>> &u1star, vector<vector<double>> &u_avg, vector<vector<double>> &
                    dTdx,
                vector<double> &Nux, double dxstar, double dystar) override {

            // Adimensionalizzazione basata su H (altezza della cavita)
            dxstar = dx / H;
            dystar = dx / H;

            // Temperatura adimensionale
            for (int i = 0; i < Ny; i++) {
                for (int j = 0; j < Nx; j++) {
                    T1star[i][j] = (T_avg[i][j] - Tcold) / (Thot - Tcold);
                }
            }

            // Velocita orizzontale adimensionale: u* = u H / alpha
            for (int i = 0; i < Ny; i++) {
                for (int j = 0; j < Nx + 1; j++) {
                    u1star[i][j] = u_avg[i][j] * H / alpha;
                }
            }

            // Derivata e flusso sulla parete calda (x = 0)
            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];
            }

            // Celle interne: schema 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];
                }
            }

            // Integrazione verticale (in y*)
            for (int j = 0; j < Nx; ++j) {
                double sum = 0.0;
                for (int i = 0; i < Ny; ++i) {
                    sum += qx[i][j] * dystar;
                }
                Nux[j] = sum;
            }

            // Nusselt medio lungo la parete
            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
                           ]);
        }
    }
}

// Funzione per il calcolo del Nusselt medio (UDS) con adimensionalizzazione basata su H
void Nusselt(int Nx, int Ny, double Tcold, double Thot, double H, double alpha, double dx, double
    &Nux_avg,
             vector<vector<double> > &T1star, vector<vector<double> > &T_avg, vector<vector<double
                 > > &qx,
             vector<vector<double> > &u1star, vector<vector<double> > &u_avg, vector<vector<double
                 > > &dTdx,
             vector<double> &Nux, double dxstar, double dystar) override {
    // Adimensionalizzazione su H
    dxstar = dx / H;
    dystar = dx / H;

    // Temperatura adimensionale
    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx; j++) {
            T1star[i][j] = (T_avg[i][j] - Tcold) / (Thot - Tcold);
        }
    }

    // Velocit  adimensionale u* = u * H /
    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx + 1; j++) {
            u1star[i][j] = u_avg[i][j] * H / alpha;
        }
    }

    // Calcolo flusso UDS (Upwind Differencing Scheme)
    for (int i = 0; i < Ny; ++i) {
        // Parete calda 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];

        // Seconda colonna
        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];

        // Celle interne
        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];
        }
    }

    // Integrazione verticale (in y*)
    for (int j = 0; j < Nx; ++j) {
        double sum = 0.0;
        for (int i = 0; i < Ny; ++i) {
            sum += qx[i][j] * dystar;
```

```
            }
            Nux[j] = sum;
        }

        // Nusselt medio
        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 Nx;
    int Ny;

    // Ask for mesh refinement
    cout << "Insert number of control volumes on x direction:";
    cin >> Nx;
    Ny = Nx * 4;
    double Ra;
    double L = 1.0;
    double H = 4.0;
    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;

    // Ask for desired Rayleigh number
    cout << "Insert Rayleigh number:";
    cin >> Ra;
    double g = Ra * mu * lambda / (cp * beta * pow(rho, 2) * (Thot - Tcold) * pow(H, 3));   //HO
        SCAMBIATO L CON H
    double alpha = lambda / (rho * cp);
    double Nux_avg = 0.0;
    double Vmax;
    double dxstar;
    double dystar;
    double Ek_old = 0.0;

    /////// Numerical data ///////

    int ite = 0;
    int sample_count = 0;
    double maxResP = 1e-3;
    double maxIte = 1e10;
    double maxResT = 1e-3;
    double maxResU = maxResT;
    double maxResV = maxResT;
    double res1 = maxResU + 1;
    double res2 = maxResV + 1;
    double res3 = maxResT + 1;
    double res3_old = res3;
    double t_count = 0.0;
    double t_count_avg = 0.0;
    double total_GS_time = 0.0;
    // Starting time for averaging
    double t_0 = 0.6;
    // Total duration of the simulation
    const double t_min = 2.5;
    double dt = 1e-6;
    double dtc, dtd, dtt;
    double omega_P = 1.8;
    bool switched = false;
    string method;

    // Ask for convective scheme
    cout << "Choose the method ('CDS', 'UDS', or 'AUTO'):";
    cin >> method;
    ranges::transform(method, method.begin(), ::toupper);
    ConvectiveScheme* calculator = nullptr;
    ConvectiveScheme* calculator_UDS = nullptr;
    ConvectiveScheme* calculator_CDS = nullptr;
    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;
    }
```

```
/////// Definition of matrices and vectors ///////

// Pressure, Temperature, u and v
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);

// Nusselt number
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);

// Averaged fields
auto T_avg = CreateMatrix(Ny, Nx, 0.0);
auto u_avg = CreateMatrix(Ny, Nx + 1, 0.0);
auto v_avg = CreateMatrix(Ny + 1, Nx, 0.0);
auto uu_avg = CreateMatrix(Ny, Nx + 1, 0.0);
auto vv_avg = CreateMatrix(Ny + 1, Nx, 0.0);
auto uv_avg = CreateMatrix(Ny, Nx, 0.0);
auto psi = CreateMatrix(Ny, Nx, 0.0);

// Kinetic Energy
auto u_var = CreateMatrix(Ny, Nx + 1, 0.0);
auto v_var = CreateMatrix(Ny + 1, Nx, 0.0);
auto uv_fluct = CreateMatrix(Ny, Nx, 0.0);
auto ke = CreateMatrix(Ny, Nx, 0.0);
auto tke = CreateMatrix(Ny, Nx, 0.0);
auto u_interp_sum = CreateMatrix(Ny, Nx, 0.0);
auto uu_interp_sum = CreateMatrix(Ny, Nx, 0.0);
auto v_interp_sum = CreateMatrix(Ny + 1, Nx, 0.0);
auto vv_interp_sum = CreateMatrix(Ny + 1, Nx, 0.0);
auto uv_interp_sum = CreateMatrix(Ny + 1, Nx, 0.0);

/////// Probes analysis ///////

int i_probe_bottom_left = Ny - 2;
int j_probe_bottom_left = 1;
int i_probe_bottom_right = Ny - 2;
int j_probe_bottom_right = Nx - 2;
int i_probe_top_left = 1;
int j_probe_top_left = 1;
int i_probe_top_right = 1;
int j_probe_top_right = Nx - 2;

/////// Initialization ///////

// 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 and linearly distributed temperature field
for (int i = 0; i < Ny; i++) {
    for (int j = Nx-1; j >= 0; --j) {
        Pg[i][j] = 0.0;
        T1[i][j] = 0;
        Tn[i][j] = T1[i][j];
        Tn_1[i][j] = Tn[i][j];
    }
```

```
}

// Apply 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);

// Open Info file
ofstream Infos ("Info_Ra_" + to_string(static_cast<int>(Ra)) + ".txt");
Infos << "===== SIMULATION INFO =====" << endl;
Infos << "RAYLEIGH =" << Ra << endl;
Infos << "Mesh size =" << Nx << " x " << Ny << endl;
Infos << "Convective scheme =" << method << endl;
Infos << "Gauss-Seidel tollerance =" << maxResP << endl;
Infos << "Relaxation Poisson =" << omega_P << endl;
Infos << "Start averaging time =" << t_0 << " s" << endl;
Infos << "Averaging window ====== =" << t_min - t_0 << " s" << endl;
Infos << "Total simulation time =" << t_min << " s" << endl;

// Open probes and kinetic budget script files
ofstream probes_out("ProbesHistory_Ra_" + to_string(static_cast<int>(Ra)) + ".txt");
ofstream energy_out("EnergyTerms_Ra_" + to_string(static_cast<int>(Ra)) + ".txt");

/////// Time loop ///////

while (t_count <= t_min && 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++) {
            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;
            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
    if (t_count > t_0) {
        maxResP = 1e-4;
    }
    auto start_GS = std::chrono::high_resolution_clock::now();
    PoissonSolver(maxResP, maxIte, rho, dx, dt, Nx, Ny, omega_P, P1, vP, uP, Pg);
    auto end_GS = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> dt_GS = end_GS - start_GS;
    total_GS_time += dt_GS.count();

    // 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;
            }
        }
    }

    // Step 4 - Temperature evaluation
    for (int i = 1; i < Ny - 1; i++) {
        for (int j = 1; j < Nx - 1; j++) {
            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 temperature boundary conditions
for (int j = 0; j < Nx; j++) {
    T1[0][j] = T1[1][j];
    T1[Ny - 1][j] = T1[Ny - 2][j];
}

// Step 5: Non-dimensional kinetic energy balance

double Ek = 0.0;
double error = 0.0;
double R_total = 0.0;
double R_diff = 0.0;
double R_conv = 0.0;
double R_press = 0.0;
double R_buoy = 0.0;

// Reference energy scale: rho * alpha^2
double Ek_ref = rho * alpha * alpha;

for (int i = 1; i < Ny-1; i++) {
    for (int j = 1; j < Nx-1; j++) {
        // Interpolate u and v to cell center (dimensional)
        double u_c = 0.5 * (u1[i][j] + u1[i][j + 1]);
        double v_c = 0.5 * (v1[i][j] + v1[i + 1][j]);

        // Kinetic energy (adimensionalized at the end)
        Ek += 0.5 * (u_c * u_c + v_c * v_c) * dx * dx;

        // Viscous diffusion term
        double lap_u = (u1[i][j + 1] + u1[i][j - 1] + u1[i - 1][j] + u1[i + 1][j] - 4.0 * u1[i
            ][j]) / (dx * dx);
        double lap_v = (v1[i][j + 1] + v1[i][j - 1] + v1[i - 1][j] + v1[i + 1][j] - 4.0 * v1[i
            ][j]) / (dx * dx);
        R_diff += mu * (u_c * lap_u + v_c * lap_v) * dx * dx;

        // Convective transport term
        double dudx = (u1[i][j + 1] - u1[i][j - 1]) / (2.0 * dx);
        double dudy = (u1[i - 1][j] - u1[i + 1][j]) / (2.0 * dx);
        double dvdx = (v1[i][j + 1] - v1[i][j - 1]) / (2.0 * dx);
        double dvdy = (v1[i - 1][j] - v1[i + 1][j]) / (2.0 * dx);
        double conv_u = u_c * dudx + v_c * dudy;
        double conv_v = u_c * dvdx + v_c * dvdy;
        R_conv -= rho * (u_c * conv_u + v_c * conv_v) * dx * dx;

        // Pressure work
        double dpdx = (P1[i][j + 1] - P1[i][j - 1]) / (2.0 * dx);
        double dpdy = (P1[i - 1][j] - P1[i + 1][j]) / (2.0 * dx);
        R_press -= (u_c * dpdx + v_c * dpdy) * dx * dx;

        // Buoyancy term
        double Fb = rho * beta * (T1[i][j] - Tinf) * g;
        R_buoy += Fb * v_c * dx * dx;
    }
}

// Final nondimensionalization
Ek /= Ek_ref;
R_diff /= Ek_ref;
R_conv /= Ek_ref;
R_press /= Ek_ref;
R_buoy /= Ek_ref;

// Time derivative (dimensionless time)
double dt_star = dt * alpha / (H * H);
double dEk_num;
if (t_count > 0) {
    dEk_num = (Ek - Ek_old) / dt_star;
} else {
    dEk_num = 0.0;
}

// Total residual and mismatch
R_total = R_diff + R_conv + R_press + R_buoy;
error = fabs(R_total - dEk_num);

// Output (already nondimensional)
energy_out << t_count << "-"
        << Ek << "-"
        << R_diff << "-"
        << R_conv << "-"
        << R_press << "-"
        << R_buoy << "-"
        << R_total << "-"
        << dEk_num << "-"
        << error << endl;

// Update old energy
Ek_old = Ek;

// Update time counter and residual
t_count += dt;
res1 = maxDiff1;
res2 = maxDiff2;
res3 = maxDiff3;

// Switch method criteria (when 'AUTO' is selected)
if (method == "AUTO" && !switched && res3 < 1e-3 && fabs(res3 - res3_old) / res3_old < 1e-2) {
    calculator = calculator_CDS;
    switched = true;
```

```
    cout << "Switched from UDS to CDS at iteration " << ite << " (res3 = " << res3 << ")\n";
}

// Update temperature residual
res3_old = res3;

// Update u^n and u^n-1
for (int i = 1; i < Ny - 1; i++) {
    for (int j = 1; j < Nx; j++) {
        un_1[i][j] = un[i][j];
        un[i][j] = u1[i][j];
    }
}

// Update v^n and v^n-1
for (int i = 1; i < Ny; i++) {
    for (int j = 1; j < Nx - 1; j++) {
        vn_1[i][j] = vn[i][j];
        vn[i][j] = v1[i][j];
    }
}

// Update T^n and T^n-1
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];
    }
}

// Print probes history
probes_out << t_count << " "
           << T1[i_probe_bottom_left][j_probe_bottom_left] << " " << u1[i_probe_bottom_left][
               j_probe_bottom_left] << " " << v1[i_probe_bottom_left][j_probe_bottom_left] <<
               " "
           << T1[i_probe_bottom_right][j_probe_bottom_right] << " " << u1[i_probe_bottom_right
               ][j_probe_bottom_right] << " " << v1[i_probe_bottom_right][j_probe_bottom_right
               ] << " "
           << T1[i_probe_top_left][j_probe_top_left] << " " << u1[i_probe_top_left][
               j_probe_top_left] << " " << v1[i_probe_top_left][j_probe_top_left] << " "
           << T1[i_probe_top_right][j_probe_top_right] << " " << u1[i_probe_top_right][
               j_probe_top_right] << " " << v1[i_probe_top_right][j_probe_top_right]
           << endl;

// Sum of the fields for averaging
if (t_count > t_0) {
    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx; j++) {
            T_avg[i][j] += T1[i][j] * dt;

            u_avg[i][j] += 0.5 * (u1[i][j] + u1[i][j + 1]) * dt;
            v_avg[i][j] += 0.5 * (v1[i][j] + v1[i + 1][j]) * dt;

            uu_avg[i][j] += pow(0.5 * (u1[i][j] + u1[i][j + 1]), 2) * dt;
            vv_avg[i][j] += pow(0.5 * (v1[i][j] + v1[i + 1][j]), 2) * dt;
            uv_avg[i][j] += 0.5 * (u1[i][j] + u1[i][j + 1]) * 0.5 * (v1[i][j] + v1[i + 1][j])
                * dt;
        }
    }
    t_count_avg += dt;
    sample_count++;
}

//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 and print values of interest for debugging purposes
        ite++;
        if (ite % 100 == 0) {
            cout << "SUMMARY: ite=" << ite
                 << "||resT=" << res3
                 << "||resU=" << res1
                 << "||resV=" << res2
                 << "||t=" << t_count
                 << "||dt=" << dt
                 << endl;
        }
    }

    // Close probes history and kinetic budget files
    probes_out.close();
    energy_out.close();

    // Average Fields
    if (sample_count > 0) {
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx; j++) {
                T_avg[i][j] /= t_count_avg;
                u_avg[i][j] /= t_count_avg;
                v_avg[i][j] /= t_count_avg;
                uu_avg[i][j] /= t_count_avg;
                vv_avg[i][j] /= t_count_avg;
                uv_avg[i][j] /= t_count_avg;
            }
        }
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx; j++) {
                u_var[i][j] = uu_avg[i][j] - pow(u_avg[i][j], 2);
                v_var[i][j] = vv_avg[i][j] - pow(v_avg[i][j], 2);
                uv_fluct[i][j] = uv_avg[i][j] - u_avg[i][j] * v_avg[i][j];
                ke[i][j] = 0.5 * (pow(u_avg[i][j],2) + pow(v_avg[i][j], 2));
                tke[i][j] = 0.5 * (u_var[i][j] + v_var[i][j]);
            }
        }
    }
    Infos << "Fields averaged at" << t_count << "seconds" << endl;

    // Stream Function Calculation
    psi[0][0] = 0.0;
    // Vertical integration in the first column
    for (int i = 1; i < Ny; ++i) {
        psi[i][0] = psi[i - 1][0] + dx * u_avg[i - 1][0];
    }
    // Horizontal integration in the rows
    for (int i = 0; i < Ny; ++i) {
        for (int j = 1; j < Nx; ++j) {
            psi[i][j] = psi[i][j - 1] - dx * v_avg[i][j - 1];
        }
    }

    /////// .txt files for plotting ///////

    // Averaged temperature distribution
    ofstream TemperatureDistribution("TemperatureDistribution_Ra_" + to_string(static_cast<int>(Ra)) +
        ".txt");
    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx; j++) {
            TemperatureDistribution << j * dx + dx / 2 << "," << H - i * dx - dx / 2 << "," << T_avg[i
                ][j] << endl;
        }
        TemperatureDistribution << "\n";
    }

    // Averaged velocity u distribution
    ofstream VelocityUDistribution("VelocityUDistribution_Ra_" + to_string(static_cast<int>(Ra)) + ".
        txt");
    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx + 1; j++) {
            VelocityUDistribution << j * dx + dx / 2 << "," << H - i * dx - dx / 2 << "," << u_avg[i][
                j] << endl;
        }
        VelocityUDistribution << "\n";
    }

    // Averaged velocity v distribution
    ofstream VelocityVDistribution("VelocityVDistribution_Ra_" + to_string(static_cast<int>(Ra)) + ".
        txt");
    for (int i = 0; i < Ny + 1; i++) {
        for (int j = 0; j < Nx; j++) {
            VelocityVDistribution << j * dx + dx / 2 << "," << H - i * dx - dx / 2 << "," << v_avg[i][
                j] << endl;
        }
        VelocityVDistribution << "\n";
    }

    // u(y) at L/2
    ofstream Uyl2("UyL2_Ra_" + to_string(static_cast<int>(Ra)) + ".txt");
    Uyl2 << 4 << "," << 0 << endl;
    for (int i = 0; i < Ny; i++) {
        Uyl2 << H - i * dx - dx / 2 << "," << (u_avg[i][Nx / 2] + u_avg[i][Nx / 2 + 1]) / 2 << endl;
    }
    Uyl2 << 0 << "," << 0 << endl;

    // v(x) at L/2
```

```
ofstream Vxl2("VxL2_Ra_" + to_string(static_cast<int>(Ra)) + ".txt");
Vxl2 << 0 << "-" << 0 << endl;
for (int j = 0; j < Nx; j++) {
    Vxl2 << j * dx + dx / 2 << "-" << (v_avg[Ny / 2][j] + v_avg[Ny / 2 + 1][j]) / 2 << endl;
}
Vxl2 << 1 << "-" << 0 << endl;

// Stream Function
ofstream StreamFunction("StreamFunction_Ra_" + to_string(static_cast<int>(Ra)) + ".txt");
for (int i = 0; i < Ny; i++) {
    for (int j = 0; j < Nx; j++) {
        double x = j * dx + dx / 2;
        double y = H - i * dx - dx / 2;
        StreamFunction << x << "-" << y << "-" << psi[i][j] << endl;
    }
    StreamFunction << "\n";
}

// Reynolds stresses, turbulent kinetic energy and mean kinetic energy
ofstream Turbulent_kinetic("Turbulent_KE_" + to_string(static_cast<int>(Ra)) + ".txt");
for (int i = 0; i < Ny; i++) {
    for (int j = 0; j < Nx; j++) {
        double x = j * dx + dx / 2;
        double y = H - i * dx - dx / 2;
        Turbulent_kinetic << x << "-" << y << "-" << u_var[i][j] << "-" << v_var[i][j] << "-" <<
            uv_fluct[i][j] <<
                "-" << tke[i][j] << "-" << ke[i][j] << endl;
    }
    Turbulent_kinetic << "\n";
}

// Adding Boundaries
// BORDO SUPERIORE (i = -1)
for (int j = 0; j < Nx; j++) {
    double x = j * dx + dx / 2;
    double y = H;
    TemperatureDistribution << x << "-" << y << "-" << T_avg[0][j] << endl;
    VelocityUDistribution << x << "-" << y << "-" << "0" << endl;
    VelocityVDistribution << x << "-" << y << "-" << "0" << endl;
    StreamFunction << x << "-" << y << "-" << "0" << endl;
    Turbulent_kinetic << x << "-" << y << "-" << "0" << "-" << "0" << "-" << "0" << "-" <<
            "0" << "-" << "0" << endl;
}

// BORDO INFERIORE (i = Ny)
for (int j = 0; j < Nx; j++) {
    double x = j * dx + dx / 2;
    double y = 0.0;
    TemperatureDistribution << x << "-" << y << "-" <<  T_avg[Ny-1][j] << endl;
    VelocityUDistribution << x << "-" << y << "-" << "0" << endl;
    VelocityVDistribution << x << "-" << y << "-" << "0" << endl;
    StreamFunction << x << "-" << y << "-" << "0" << endl;
    Turbulent_kinetic << x << "-" << y << "-" << "0" << "-" << "0" << "-" << "0" << "-" <<
            "0" << "-" << "0" << endl;
}

// BORDO SINISTRO (j = -1)
for (int i = 0; i < Ny; i++) {
    double x = 0.0;
    double y = H - i * dx - dx / 2;
    TemperatureDistribution << x << "-" << y << "-" << "1" << endl;
    VelocityUDistribution << x << "-" << y << "-" << "0" << endl;
    VelocityVDistribution << x << "-" << y << "-" << "0" << endl;
    StreamFunction << x << "-" << y << "-" << "0" << endl;
    Turbulent_kinetic << x << "-" << y << "-" << "0" << "-" << "0" << "-" << "0" << "-" <<
            "0" << "-" << "0" << endl;
}

// BORDO DESTRO (j = Nx)
for (int i = 0; i < Ny; i++) {
    double x = 1.0;
    double y = H - i * dx - dx / 2;
    TemperatureDistribution << x << "-" << y << "-" << "0" << endl;
    VelocityUDistribution << x << "-" << y << "-" << "0" << endl;
    VelocityVDistribution << x << "-" << y << "-" << "0" << endl;
    StreamFunction << x << "-" << y << "-" << "0" << endl;
    Turbulent_kinetic << x << "-" << y << "-" << "0" << "-" << "0" << "-" << "0" << "-" <<
            "0" << "-" << "0" << endl;
}

// ANGOLO ALTO SINISTRO
TemperatureDistribution << 0.0 << "-" << H << "-" << "1" << endl;
VelocityUDistribution << 0.0 << "-" << H << "-" << "0" << endl;
VelocityVDistribution << 0.0 << "-" << H << "-" << "0" << endl;
StreamFunction << 0.0 << "-" << H << "-" << "0" << endl;
Turbulent_kinetic << 0.0 << "-" << H << "-" << "0" << "-" << "0" << "-" << "0" << "-" <<
            "0" << "-" << "0" << endl;
// ANGOLO ALTO DESTRO
TemperatureDistribution << 1.0 << "-" << H << "-" << "0" << endl;
VelocityUDistribution << 1.0 << "-" << H << "-" << "0" << endl;
VelocityVDistribution << 1.0 << "-" << H << "-" << "0" << endl;
StreamFunction << 1.0 << "-" << H << "-" << "0" << endl;
Turbulent_kinetic << 1.0 << "-" << H << "-" << "0" << "-" << "0" << "-" << "0" << "-" <<
            "0" << "-" << "0" << endl;
// ANGOLO BASSO SINISTRO
TemperatureDistribution << 0.0 << "-" << 0.0 << "-" << "1" << endl;
VelocityUDistribution << 0.0 << "-" << 0.0 << "-" << "0" << endl;
VelocityVDistribution << 0.0 << "-" << 0.0 << "-" << "0" << endl;
StreamFunction << 0.0 << "-" << 0.0 << "-" << "0" << endl;
```

```
    Turbulent_kinetic << 0.0 << "-" << 0.0 << "-" << "0" << "-" << "0" << "-" << "0" << "-" <<
                    "0" << "-" << "0" << endl;
    // ANGOLO BASSO DESTRO
    TemperatureDistribution << 1.0 << "-" << 0.0 << "-" << "0" << endl;
    VelocityUDistribution << 1.0 << "-" << 0.0 << "-" << "0" << endl;
    VelocityVDistribution << 1.0 << "-" << 0.0 << "-" << "0" << endl;
    StreamFunction << 1.0 << "-" << 0.0 << "-" << "0" << endl;
    Turbulent_kinetic << 1.0 << "-" << 0.0 << "-" << "0" << "-" << "0" << "-" << "0" << "-" <<
                    "0" << "-" << "0" << endl;


    TemperatureDistribution.close();
    VelocityUDistribution.close();
    VelocityVDistribution.close();
    Uyl2.close();
    Vxl2.close();
    StreamFunction.close();
    Turbulent_kinetic.close();

    /////// Final calculations ///////

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

    // Maximum u* velocity at x = L/2, adimensionalizzata con H
    for (int i = 0; i < Ny; i++) {
        u1star_L2[i] = (u_avg[i][Nx / 2] + u_avg[i][Nx / 2 + 1]) / 2 * H / alpha;
    }
    auto ustar_max = ranges::max_element(u1star_L2);
    int u_index = distance(u1star_L2.begin(), ustar_max);
    double y_u = (Ny - u_index - 0.5) * dx;  // y fisico
    double ystar_u = y_u / H;  // y adimensionale con H
    Infos << "Max u* velocity=" << *ustar_max << " at y/H=" << ystar_u << endl;

    // Maximum v* velocity at y = H/2, adimensionalizzata con H
    for (int j = 0; j < Nx; j++) {
        v1star_L2[j] = (v_avg[Ny / 2][j] + v_avg[Ny / 2 + 1][j]) / 2 * H / 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;  // x fisico
    double xstar_v = x_v / H;
    Infos << "Max v* velocity=" << *vstar_max << " at x/H=" << xstar_v << endl;

    // Stop timer and print total duration
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<seconds>(stop - start);
    Infos << "Total Execution Time:" << duration.count() << " seconds" << endl;
    Infos << "Total GS Solver Time:" << total_GS_time << " seconds=" << total_GS_time * 100 /
        duration.count() <<
            "% of the total execution time" << endl;

    Infos.close();

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

    return 0;
}
```

B Python Post-Processing Script

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

plt.rcParams.update({
    "text.usetex": False,
    "font.family": "serif",
    "font.size": 18,
    "axes.titlesize": 20,
    "axes.labelsize": 18,
    "xtick.labelsize": 16,
    "ytick.labelsize": 16,
    "legend.fontsize": 14,
    "figure.titlesize": 22
})

#Rayleigh
Ra = float(input("Insert Rayleigh number: "))
print(f"Loading files for Ra = {int(Ra)}...")

#Domain parameters
domainWidth = 1.0
aspectRatio = 4.0
fine_N = 500
x_fine = np.linspace(0, domainWidth, fine_N)
y_fine = np.linspace(0, domainWidth * aspectRatio, fine_N)
X_fine, Y_fine = np.meshgrid(x_fine, y_fine)


######################################################################

################## COLORMAPS - SEPARATE FIGURES ##################

######################################################################

quantities = [
    (f"TemperatureDistribution_Ra_{int(Ra)}.txt", "Temperature"),
    (f"VelocityUDistribution_Ra_{int(Ra)}.txt", "u-Velocity"),
    (f"VelocityVDistribution_Ra_{int(Ra)}.txt", "v-Velocity")
]

for filename, quantity_name in quantities:
    # Load data
    data = pd.read_csv(filename, sep=r"\s+", header=None)
    x, y, value = data[0].values, data[1].values, data[2].values

    # Interpolation
    value_fine = griddata((x, y), value, (X_fine, Y_fine), method='cubic')

    fig, ax = plt.subplots(figsize=(6, 10))
    img = ax.contourf(X_fine, Y_fine, value_fine, levels=100, cmap="jet")
    contours = ax.contour(X_fine, Y_fine, value_fine, colors='black', levels=30, linewidths=0.8)
    # ax.clabel(contours, inline=True, fontsize=8)
    ax.set_xlim([0, domainWidth])
    ax.set_ylim([0, domainWidth * aspectRatio])
    ax.set_aspect('equal')
    #ax.axis("off")
    #fig.suptitle(f"{quantity_name} Field", fontsize=22)
    cbar = fig.colorbar(img, ax=ax, shrink=0.9, pad=0.02)
    output_name = f"{quantity_name.replace('-','')}_Colormap_Ra_{int(Ra)}.pdf"
    plt.tight_layout()
    plt.savefig(output_name, dpi=300, bbox_inches='tight')
    plt.show()
    plt.close()


############################################################

################## TURBULENT KINETIC ENERGY ##################

############################################################

# === Load data
df = pd.read_csv(f"Turbulent_KE_{int(Ra)}.txt", sep=r"\s+", header=None,
                 names=["x", "y", "u_var", "v_var", "uv_fluct", "tke", "ke"])
points = df[["x", "y"]].values

# === Interpolation
fields = {
    "TKE": df["tke"].values,
    "KE": df["ke"].values,
    "u'u'": df["u_var"].values,
    "v'v'": df["v_var"].values,
    "u'v'": df["uv_fluct"].values
}

interpolated_fields = {
    name: griddata(points, values, (X_fine, Y_fine), method='cubic')
    for name, values in fields.items()
}

def plot_field(Z, filename, num_levels=14, cmap="Blues", force_positive=False):
```

```
    levels = np.linspace(np.nanmin(Z), np.nanmax(Z), num_levels)

    fig, ax = plt.subplots(figsize=(6, 10))
    cf = ax.contourf(X_fine, Y_fine, Z, levels=levels, cmap=cmap)
    ax.contour(X_fine, Y_fine, Z, levels=levels, colors='black', linewidths=0.8)

    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_xlim([0, domainWidth])
    ax.set_ylim([0, domainWidth * aspectRatio])
    ax.set_aspect('equal')

    fig.colorbar(cf, ax=ax, shrink=0.9, pad=0.02)

    plt.tight_layout()
    plt.savefig(f"{filename}.pdf", dpi=300, bbox_inches='tight')
    plt.show()
    plt.close()


plot_field(interpolated_fields["TKE"], f"TKE_Ra_{int(Ra)}")
plot_field(interpolated_fields["KE"],  f"KE_Ra_{int(Ra)}")
plot_field(interpolated_fields["u'u'"], f"u_var_Ra_{int(Ra)}")
plot_field(interpolated_fields["v'v'"], f"v_var_Ra_{int(Ra)}")
plot_field(interpolated_fields["u'v'"], f"uv_fluct_Ra_{int(Ra)}")



#########################################################################

################## STREAM FUNCTION - SEPARATE FIGURES ##################

#########################################################################

df_psi = pd.read_csv(f"StreamFunction_Ra_{int(Ra)}.txt", sep=r"\s+", header=None, names=["x", "y", "
    psi"])
df_u = pd.read_csv(f"VelocityUDistribution_Ra_{int(Ra)}.txt", sep=r"\s+", header=None, names=["x", "y"
    , "u"])
df_v = pd.read_csv(f"VelocityVDistribution_Ra_{int(Ra)}.txt", sep=r"\s+", header=None, names=["x", "y"
    , "v"])
df_psi_sorted = df_psi.sort_values(by=["y", "x"], ascending=[True, True])
Z = df_psi_sorted.pivot(index="y", columns="x", values="psi").values

x_vals = np.sort(df_psi["x"].unique())
y_vals = np.sort(df_psi["y"].unique())
X, Y = np.meshgrid(x_vals, y_vals)

# === Interpolation
scale = 3
x_all = np.linspace(x_vals.min(), x_vals.max(), len(x_vals) * scale)
y_all = np.linspace(y_vals.min(), y_vals.max(), len(y_vals) * scale)
Xi, Yi = np.meshgrid(x_all, y_all)
Ui = griddata((df_u["x"], df_u["y"]), df_u["u"], (Xi, Yi), method="cubic")
Vi = griddata((df_v["x"], df_v["y"]), df_v["v"], (Xi, Yi), method="cubic")
speed = np.sqrt(Ui**2 + Vi**2)

# === Figure 1: Isolines ===
fig1, ax1 = plt.subplots(figsize=(6, 10))
levels = np.linspace(Z.min(), Z.max(), 20)
cs = ax1.contour(X, Y, Z, levels=levels, colors='black', linewidths=0.8, linestyles='solid')
ax1.set_aspect('equal')
ax1.axis('off')
plt.tight_layout()
plt.savefig(f"StreamFunction_Contours_Ra_{int(Ra)}.pdf", dpi=300, bbox_inches='tight')
plt.show()
plt.close()

# === Figure 2: Streamlines vectors ===
fig2, ax2 = plt.subplots(figsize=(6, 10))
stream = ax2.streamplot(Xi, Yi, Ui, Vi, color=speed, cmap='viridis', density=1.5, linewidth=1,
    arrowsize=1)
ax2.set_aspect('equal')
ax2.axis('off')
plt.tight_layout()
plt.savefig(f"StreamFunction_Streamlines_Ra_{int(Ra)}.pdf", dpi=300, bbox_inches='tight')
plt.show()
plt.close()

# === Figure 3: Colormap ===
fig3, ax3 = plt.subplots(figsize=(6, 10))
levels = np.linspace(Z.min(), Z.max(), 100)
cf = ax3.contourf(X, Y, Z, levels=levels, cmap='Blues')
#cbar3 = fig3.colorbar(cf, ax=ax3, label=r"ψ")
ax3.set_aspect('equal')
ax3.axis('off')
plt.tight_layout()
plt.savefig(f"StreamFunction_Colormap_Ra_{int(Ra)}.pdf", dpi=300, bbox_inches='tight')
plt.show()
plt.close()



#########################################################################

################## VELOCITY PROFILES AT MID SECTIONS ##################

#########################################################################
```

```
Ra_list = [1e8, 6.4e8, 1.07e9]
Ra_labels = [r"Ra = 8 × 10^7", r"Ra = 6.4 × 10^8", r"Ra = 1.07 × 10^9"]
colors = ['tab:blue', 'tab:orange', 'tab:green']
styles = ['-', '--', '-.']

# === PROFILE U(y) at x = L/2 ===
plt.figure(figsize=(6, 8))
for Ra, label, color, style in zip(Ra_list, Ra_labels, colors, styles):
    fname = f"UyL2_Ra_{int(Ra)}.txt"
    data = np.loadtxt(fname)
    data = data[(data[:, 0] != 0.05) & (data[:, 0] != 3.95)]
    data = data[np.argsort(data[:, 0])]

    y_vals = data[:, 0]
    u_vals = data[:, 1]

    interp_u = PchipInterpolator(y_vals, u_vals)
    y_smooth = np.linspace(y_vals.min(), y_vals.max(), 300)
    u_smooth = interp_u(y_smooth)

    plt.plot(u_smooth, y_smooth, linestyle=style, color=color, linewidth=2, label=label)

plt.grid(True, linestyle='--', alpha=0.5)
plt.xlabel('u(y) at x = L/2')
plt.ylabel('y')
plt.xlim(-1200, 1200)
plt.legend(loc='center right', fontsize=10, frameon=True)
plt.tight_layout()
plt.savefig("U_Profile_Comparison.pdf")
plt.show()


# === PROFILE V(x) at y = H/2 ===
plt.figure(figsize=(8, 6))
for Ra, label, color, style in zip(Ra_list, Ra_labels, colors, styles):
    fname = f"VxL2_Ra_{int(Ra)}.txt"
    data = np.loadtxt(fname)
    data = data[(data[:, 0] != 0.05) & (data[:, 0] != 0.95)]
    data = data[np.argsort(data[:, 0])]

    x_vals = data[:, 0]
    v_vals = data[:, 1]

    interp_v = PchipInterpolator(x_vals, v_vals)
    x_smooth = np.linspace(x_vals.min(), x_vals.max(), 300)
    v_smooth = interp_v(x_smooth)

    plt.plot(x_smooth, v_smooth, linestyle=style, color=color, linewidth=2, label=label)

plt.grid(True, linestyle='--', alpha=0.5)
plt.xlabel('x')
plt.ylabel('v(x) at y = H/2')
plt.legend()
plt.tight_layout()
plt.savefig("V_Profile_Comparison.pdf")
plt.show()



############################################################

################## KINETIC ENERGY BUDGET ##################

############################################################

# Load data: columns = t, Ek, R_diff, R_conv, R_press, R_bous, R_total, dEk_num, error
data = np.loadtxt(f"EnergyTerms_Ra_{int(Ra)}.txt")

# Extract individual arrays
t = data[:, 0]              # Time
Ek = data[:, 1]             # Total kinetic energy
R_diff = data[:, 2]         # Viscous diffusion term
R_conv = data[:, 3]         # Convective transport term
R_press = data[:, 4]        # Pressure work term
R_bous = data[:, 5]         # Buoyancy forcing
R_total = data[:, 6]        # Sum of all budget terms
dEk_num = data[:, 7]        # Numerical derivative of Ek
error = data[:, 8]          # Absolute error

# === Plot 1: Kinetic Energy vs Time ===
plt.figure(figsize=(10, 6))
plt.plot(t, Ek, label='Kinetic Energy $E_k$', color='navy')
plt.xlabel('Time')
plt.ylabel('Kinetic Energy')
#plt.title('Time Evolution of Kinetic Energy')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.savefig(f'kinetic_energy_Ra_{int(Ra)}.pdf')
plt.show()

# === Plot 2: Buoyancy and Viscous Diffusion ===
plt.figure(figsize=(10, 6))
plt.plot(t, R_bous, label='Buoyancy Forcing', linestyle='-', color='purple')
plt.plot(t, R_diff, label='Viscous Diffusion', linestyle='--', color='red')
plt.xlabel('Time')
plt.ylabel('Energy Contribution')
#plt.title('Kinetic Energy Budget: Buoyancy and Viscous Diffusion')
plt.grid(True)
```

```
plt.legend()
plt.tight_layout()
plt.savefig(f'budget_buoyancy_viscous_Ra_{int(Ra)}.pdf')
plt.show()

# === Plot 3: Convective and Pressure Terms ===
plt.figure(figsize=(10, 6))
plt.plot(t, R_conv, label='Convective Transport', linestyle='-.', color='green')
plt.plot(t, R_press, label='Pressure Work', linestyle=':', color='orange')
plt.xlabel('Time')
plt.ylabel('Energy Contribution')
#plt.title('Kinetic Energy Budget: Convective and Pressure Terms')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.savefig(f'budget_convective_pressure_Ra_{int(Ra)}.pdf')
plt.show()

# === Plot 4: Compare Kinetic Energy vs Total Budget Term ===
plt.figure(figsize=(10, 6))
plt.plot(t, Ek, label='Kinetic Energy $E_k$', color='blue')
plt.plot(t, R_total, label=r'Total Budget Term $\mathcal{R}_{\tot}$', color='black', linestyle='--')
plt.xlabel('Time')
plt.ylabel('Value')
#plt.title('Kinetic Energy vs Total Budget Term')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.savefig(f'Ek_vs_Rtotal_Ra_{int(Ra)}.pdf')
plt.show()

# === Plot 4: dEk_num vs R_total ===
plt.figure(figsize=(10, 6))
plt.plot(t, dEk_num, label='Numerical $\frac{dE_k}{dt}$', color='darkcyan')
plt.plot(t, R_total, label='Analytical $\mathcal{R}_{\text{tot}}$', linestyle='--', color='black')
plt.xlabel('Time')
plt.ylabel('Rate of Change')
#plt.title('Comparison: Numerical vs Analytical Derivative')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.savefig(f'dEk_vs_Rtotal_Ra_{int(Ra)}.pdf')
plt.show()

# === Plot 5: Error vs Time ===
plt.figure(figsize=(10, 6))
plt.plot(t, error, label='|dEk_num - R_total|', color='crimson')
plt.xlabel('Time')
plt.ylabel('Absolute Error')
#plt.title('Error Between Numerical and Analytical dEk/dt')
plt.yscale('log')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.savefig(f'error_vs_time_Ra_{int(Ra)}.pdf')
plt.show()


#######################################################

################## PROBES ANALYSIS ##################

#######################################################

# Load the probes history
data = np.loadtxt(f"ProbesHistory_Ra_{int(Ra)}.txt")

time = data[:, 0]

# Bottom left probe
T_bl = data[:, 1]
u_bl = data[:, 2]
v_bl = data[:, 3]

# Bottom right probe
T_br = data[:, 4]
u_br = data[:, 5]
v_br = data[:, 6]

# Top left probe
T_tl = data[:, 7]
u_tl = data[:, 8]
v_tl = data[:, 9]

# Top right probe
T_tr = data[:, 10]
u_tr = data[:, 11]
v_tr = data[:, 12]

def plot_probe(time, T, u, v, label, filename):
    fig, axes = plt.subplots(3, 1, figsize=(10, 12), sharex=True)
    fig.suptitle(f"{label}    -Temporal Evolution", fontsize=20)

    axes[0].plot(time, T, color='tab:red', linestyle='-',
                 label='Temperature')
    axes[0].set_ylabel('T')
    axes[0].grid(True)
    axes[0].legend(fontsize=10, loc='upper right')
```

```
    axes[1].plot(time, u, color='tab:blue', linestyle='-',
                 label='u-Velocity')
    axes[1].set_ylabel('u')
    axes[1].grid(True)
    axes[1].legend(fontsize=10, loc='upper right')

    axes[2].plot(time, v, color='tab:green', linestyle='-.',
                 label='v-Velocity')
    axes[2].set_ylabel('v')
    axes[2].set_xlabel('Time [s]')
    axes[2].grid(True)
    axes[2].legend(fontsize=10, loc='upper right')

    plt.tight_layout(rect=[0, 0, 1, 0.95])
    plt.savefig(filename, dpi=300, bbox_inches='tight')
    plt.show()

plot_probe(time, T_bl, u_bl, v_bl, "Bottom-Left-Probe", f"Probe_BL_Ra_{int(Ra)}.pdf")
plot_probe(time, T_br, u_br, v_br, "Bottom-Right-Probe", f"Probe_BR_Ra_{int(Ra)}.pdf")
plot_probe(time, T_tl, u_tl, v_tl, "Top-Left-Probe", f"Probe_TL_Ra_{int(Ra)}.pdf")
plot_probe(time, T_tr, u_tr, v_tr, "Top-Right-Probe", f"Probe_TR_Ra_{int(Ra)}.pdf")
```