ME5302 Assignment: Natural convection in a square cavity

Er Qi Yang A0164661A

May 1, 2025

Abstract

This assignment investigates natural convection in a square cavity using the **Finite Difference Method (FDM)** applied to the **vorticity-streamfunction formulation** of the non-dimensional Navier-Stokes equations. The governing equations—vorticity transport, Poisson equation for streamfunction, and heat equation—are discretized via **central differencing** and solved numerically in **Python**. Key steps include:

- Derivation of the vorticity-streamfunction form to eliminate pressure and enforce incompressibility.
- Time integration using an **explicit Euler scheme** for vorticity and temperature, coupled with a **Point Jacobi relaxation method** for the streamfunction.
- Implementation of **second-order accurate boundary conditions** for vorticity (via Taylor expansion), streamfunction (via one-sided differences), and temperature (Dirichlet/Neumann conditions).
- Computation of velocity fields (u, v) from the streamfunction.

The solver is validated for $Pr = 0.7$ at $Ra = 3.5 \times 10^3$ (61×61 mesh) and $Ra = 2.5 \times 10^4$ (121×121 mesh), reporting extremal velocities and Nusselt numbers. Results demonstrate the method's capability to capture convective dynamics while highlighting trade-offs between accuracy and computational cost.

1 Solution procedure

1.1 Deriving the vorticity-stream form of the non-dimensional Navier Stokes equations

Non-Dimensional Incompressible Navier-Stokes Equations (2D, Scalar Form)

Continuity Equation

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{1}$$

x-Momentum Equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{\partial p}{\partial x} + Pr\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{2}$$

y-Momentum Equation

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{\partial p}{\partial y} + Pr\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) + Pr \cdot Ra \cdot T \tag{3}$$

Non-Dimensional Heat Equation (with Advection)

$$\frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \tag{4}$$

From Primitive to Vorticity–Streamfunction Form

The vorticity is defined as:

$$\omega = -\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \tag{5}$$

To automatically satisfy the continuity equation, introduce the streamfunction ψ:

$$u = \frac{\partial \psi}{\partial y}, \qquad v = -\frac{\partial \psi}{\partial x} \tag{6}$$

Substituting these into the continuity equation:

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

which is automatically satisfied due to equality of mixed partial derivatives.

Now, take the curl of the momentum equations (subtract the x-derivative of the y-momentum from the y-derivative of the x-momentum) which eliminates the pressure terms:

$$\begin{aligned}
&\frac{\partial}{\partial y}\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) - \frac{\partial}{\partial x}\left(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) \\
=&Pr\left(\frac{\partial}{\partial y}\left(\nabla^2 u\right) - \frac{\partial}{\partial x}\left(\nabla^2 v\right)\right) - Pr \cdot Ra \cdot \frac{\partial T}{\partial x}
\end{aligned}$$

Simplifying, and recognizing the left side as the material derivative of ω, yields:

$$\frac{\partial \omega}{\partial t} + u\frac{\partial \omega}{\partial x} + v\frac{\partial \omega}{\partial y} = Pr\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right) - Pr \cdot Ra \cdot \frac{\partial T}{\partial x} \tag{7}$$

Poisson Equation for Streamfunction From the vorticity definition and streamfunction relations:

$$\omega = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = -\left(\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2}\right) \tag{8}$$

which gives the Poisson equation:

$$\nabla^2 \psi = \omega \tag{9}$$

Vorticity–Streamfunction Form (2D, Non-Dimensional)

Vorticity Transport Equation

$$\boldsymbol{\frac{\partial \omega}{\partial t} + u\frac{\partial \omega}{\partial x} + v\frac{\partial \omega}{\partial y} = Pr\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right) - Pr \cdot Ra \cdot \frac{\partial T}{\partial x}} \tag{10}$$

Poisson Equation (Streamfunction–Vorticity Relation)

$$\boldsymbol{\nabla^2 \psi = \omega} \tag{11}$$

Non-Dimensional Scalar Heat Equation

$$\boldsymbol{\frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}} \tag{12}$$

1.2 Defining residual functions

In order to compute the vorticity and temperature fields at the next time step, we first define *residual functions*, which act as spatial operators. These residuals allow us to conveniently apply implicit or explicit time-stepping schemes for the numerical solution.

Vorticity Residual Function (Spatial Operator $S(\omega_{i,j})$)

The continuous form of the vorticity residual is given by:

$$R^{\omega} = Pr\left(\frac{\partial^2\omega}{\partial x^2} + \frac{\partial^2\omega}{\partial y^2}\right) + Pr \cdot Ra \cdot \frac{\partial T}{\partial x} - u\frac{\partial \omega}{\partial x} - v\frac{\partial \omega}{\partial y} \tag{13}$$

Discretizing this expression using central differences in both space directions yields:

$$\begin{aligned} R^{\omega}_{i,j} = Pr&\left(\frac{\omega_{i-1,j} - 2\omega_{i,j} + \omega_{i+1,j}}{\Delta x^2} + \frac{\omega_{i,j-1} - 2\omega_{i,j} + \omega_{i,j+1}}{\Delta y^2}\right) \\ &- Pr \cdot Ra \cdot \frac{T_{i+1,j} - T_{i-1,j}}{2\Delta x} \\ &- u_{i,j} \cdot \frac{\omega_{i+1,j} - \omega_{i-1,j}}{2\Delta x} - v_{i,j} \cdot \frac{\omega_{i,j+1} - \omega_{i,j-1}}{2\Delta y} \end{aligned} \tag{14}$$

Temperature Residual Function (Spatial Operator $S(T_{i,j})$)

Similarly, for the temperature field, the continuous residual is:

$$R^{Temp} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} - u\frac{\partial T}{\partial x} - v\frac{\partial T}{\partial y} \tag{15}$$

And its discretized form using central differences is:

$$\begin{aligned} R^{Temp}_{i,j} = &\frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{\Delta x^2} + \frac{T_{i,j-1} - 2T_{i,j} + T_{i,j+1}}{\Delta y^2} \\ &- u_{i,j} \cdot \frac{T_{i+1,j} - T_{i-1,j}}{2\Delta x} - v_{i,j} \cdot \frac{T_{i,j+1} - T_{i,j-1}}{2\Delta y} \end{aligned} \tag{16}$$

Stream function Residual Function (Spatial Operator $S(\psi_{i,j})$)

The residual of the poisson equation is slightly different as it uses the solution of vorticity $\omega^{n+1}_{i,j}$ in the next time step to solve for $\psi^{n+1}_{i,j}$. This is also because I chose to solve the stream function using the Δ-form due to its easy implementation and boundary conditions.

$$S^n_{i,j} = \omega^{n+1}_{i,j} - \left(\frac{\psi^n_{i-1,j} - 2\psi^n_{i,j} + \psi^n_{i+1,j}}{\Delta x^2} + \frac{\psi^n_{i,j-1} - 2\psi^n_{i,j} + \psi^n_{i,j+1}}{\Delta y^2}\right) \tag{17}$$

These residual functions form the basis of numerical solution algorithms, where the time advancement for vorticity and temperature is carried out by applying an explicit scheme (Euler explicit) using these spatial operators. The residual function of the stream function is used in an iterative method after applying the Euler implicit method in order to solve for the stream function in the next time step. This method is the Jacobi-relaxation solver.

1.3 Explicit scheme to march in time

Time Integration Methods for Vorticity and Temperature Fields

In this section, we outline the numerical schemes used to advance the vorticity, $\omega_{i,j}$, temperature, $T_{i,j}$ and stream function $\psi_{i,j}$ fields in time. These schemes include the Euler method, a modified Runge-Kutta 4 (RK4) scheme (both explicit schemes), and the Point Jacobi relaxation method (implicit iterative scheme). **Only the Euler explicit for $T_{i,j}, \omega_{i,j}$ and Point Jacobi for $\psi_{i,j}$ was used as the RK4 method did not significantly change the run time**

Explicit Euler Scheme

The Explicit Euler method is a simple first-order time integration scheme, updating the vorticity (and similarly, temperature) as follows:

$$\omega^{n+1}_{i,j} = \omega^n_{i,j} + \Delta t \cdot R^{\omega^n}_{i,j} \tag{18}$$

$$T^{n+1}_{i,j} = T^n_{i,j} + \Delta t \cdot R^{Temp^n}_{i,j} \tag{19}$$

where $R^n_{i,j}$ is the residual evaluated at the current time step.

Explicit Runge-Kutta 4 (RK4) Scheme

A higher-order, more accurate time integration scheme is the modified Runge-Kutta 4 (RK4) method, proposed by James. This scheme involves the following stages:

1. Compute intermediate stages:

$$k_1 = 0.25 \cdot \Delta t \cdot R\left(\omega_{i,j}^n\right) \tag{20}$$

$$k_2 = \frac{\Delta t}{3} \cdot R\left(\omega_{i,j}^n + k_1\right) \tag{21}$$

$$k_3 = 0.5 \cdot \Delta t \cdot R\left(\omega_{i,j}^n + k_2\right) \tag{22}$$

2. Update the vorticity field:

$$\omega_{i,j}^{n+1} = \omega_{i,j}^n + \Delta t \cdot R\left(\omega_{i,j}^n + k_3\right) \tag{23}$$

This method improves accuracy by estimating the residual at multiple intermediate points within each time step.

Point Jacobi Relaxation Method

For certain problems, especially those involving implicit time-stepping or steady-state solutions, the Point Jacobi relaxation method is employed. This iterative approach updates each point in the vorticity (or temperature) field based on the following procedure. First we cast the discretised poisson equation into a Δ-form with coefficients b_W, b_E, b_S, b_N, b_P.
The discretized form of the Poisson equation using finite differences can be expressed as:

$$b_W \Delta\psi_{i-1,j}^{n+1} + b_E \Delta\psi_{i+1,j}^{n+1} + b_S \Delta\psi_{i,j-1}^{n+1} + b_N \Delta\psi_{i,j+1}^{n+1} + b_P \Delta\psi_{i,j}^{n+1} = S_{i,j}^n \tag{24}$$

$$S_{i,j}^n = \omega_{i,j}^{n+1} - \left(\frac{\psi_{i-1,j}^n - 2\psi_{i,j}^n + \psi_{i+1,j}^n}{\Delta x^2} + \frac{\psi_{i,j-1}^n - 2\psi_{i,j}^n + \psi_{i,j+1}^n}{\Delta y^2}\right) \tag{25}$$

where:

- $\Delta\psi_{i,j}^{n+1}$ is the correction to the potential at grid point (i, j) at iteration $n + 1$,
- b_W, b_E, b_S, b_N, b_P are the coefficients for the west, east, south, north, and central points, respectively,
- $S_{i,j}^n$ is the residual at grid point (i, j) at iteration n.

1. Compute the coefficient for the central point:

$$b_p = \left(\frac{2}{\Delta x^2} + \frac{2}{\Delta y^2}\right) + \frac{1}{\Delta t} \tag{26}$$

2. Update the vorticity field iteratively:

$$\psi_{i,j}^{n+1} = \psi_{i,j}^n + \beta \cdot \frac{R_{i,j}^{\psi}}{b_p} \tag{27}$$

where β is the relaxation factor chosen to improve convergence.This update is applied to all interior points except for $i \in \{1, 2, N-1, N-2\}$ and $j \in \{1, 2, M-1, M-2\}$, while keeping the boundary conditions fixed at each iteration.At the boundary, we impose two conditions for ψ: $\psi = 0$ and $\frac{\partial\psi}{\partial n} = 0$.

These time integration methods collectively provide a framework for evolving the vorticity and temperature fields in time, balancing between computational efficiency and numerical accuracy.

1.4 Updating boundary conditions for T, ψ, ω after stepping in time

Updating of Stream Function ψ at the Boundary

We assume the following boundary conditions:

$$\psi = \frac{\partial \psi}{\partial x} = \frac{\partial \psi}{\partial y} = 0 \quad \text{at the boundary.}$$

These conditions imply that both the stream function and its normal derivatives are zero along the domain boundaries. To approximate the boundary values just inside the domain, we update the stream function at the second and second-last grid points using their inner neighboring values:

$$\psi_{2,j} = \frac{1}{4}\psi_{3,j} \tag{28}$$

$$\psi_{N-1,j} = \frac{1}{4}\psi_{N-2,j} \tag{29}$$

$$\psi_{i,2} = \frac{1}{4}\psi_{i,3} \tag{30}$$

$$\psi_{i,M-1} = \frac{1}{4}\psi_{i,M-2} \tag{31}$$

This is derived using the one sided second order finite difference scheme to approximate the derivative condition as shown in Professor Shu's lecture notes in page 140:
At the left boundary ($i = 1$) for a grid point $(1, j)$, the one-sided second-order finite difference approximation for the first derivative in the x-direction is:

$$\left.\frac{\partial \psi}{\partial x}\right|_{i=1,j} \approx \frac{-3\psi_{1,j} + 4\psi_{2,j} - \psi_{3,j}}{2\Delta x}$$

Given the boundary conditions:

$$\psi_{1,j} = 0, \quad \frac{\partial \psi}{\partial x} = 0$$

Substituting these into the finite difference formula:

$$0 = \frac{-3(0) + 4\psi_{2,j} - \psi_{3,j}}{2\Delta x}$$

Solving for $\psi_{2,j}$:

$$4\psi_{2,j} = \psi_{3,j}$$

$$\psi_{2,j} = \frac{1}{4}\psi_{3,j}$$

The other 3 relations can be derived using the similar inner neighbouring points.

Updating of Vorticity ω at the Boundary

Similarly we use the second order approximation of ψ at the inner grid point to update ω at the boundary by doing a Taylor expansion up to the third order.
We start with the governing Poisson equation:

$$\omega = \left(\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2}\right)$$

Left Wall (at $x = 0$, $i = 1$)
The third-order Taylor expansion of ψ about the boundary point $(1, j)$ is:

$$\psi_{2,j} = \psi_{1,j} + \Delta x \left.\frac{\partial \psi}{\partial x}\right|_{1,j} + \frac{\Delta x^2}{2!}\left.\frac{\partial^2 \psi}{\partial x^2}\right|_{1,j} + \frac{\Delta x^3}{3!}\left.\frac{\partial^3 \psi}{\partial x^3}\right|_{1,j} + \mathcal{O}(\Delta x^4)$$

Using the boundary conditions:

$$\psi_{1,j} = 0, \quad \left.\frac{\partial \psi}{\partial x}\right|_{1,j} = 0$$

Simplifying:

$$\psi_{2,j} = \frac{\Delta x^2}{2}\left.\frac{\partial^2 \psi}{\partial x^2}\right|_{1,j} + \frac{\Delta x^3}{6}\left.\frac{\partial^3 \psi}{\partial x^3}\right|_{1,j}$$

From the Poisson equation:

$$\left.\frac{\partial^2 \psi}{\partial x^2}\right|_{1,j} = -\omega_{1,j}$$

Also, differentiating the Poisson equation:

$$\frac{\partial \omega}{\partial x}|_{1,j} = -\frac{\partial^3 \psi}{\partial x^3}|_{1,j}$$

Using a finite difference approximation:

$$\frac{\partial \omega}{\partial x}|_{1,j} \approx \frac{\omega_{2,j} - \omega_{1,j}}{\Delta x}$$

Substituting back:

$$\psi_{2,j} = -\frac{\Delta x^2}{2}\omega_{1,j} + \frac{\Delta x^3}{6}\left(-\frac{\omega_{2,j} - \omega_{1,j}}{\Delta x}\right)$$

Simplifying:

$$\psi_{2,j} = -\frac{\Delta x^2}{3}\omega_{1,j} - \frac{\Delta x^2}{6}\omega_{2,j}$$

Solving for $\omega_{1,j}$:

$$\omega_{1,j} = \frac{3\psi_{2,j}}{\Delta x^2} - \frac{1}{2}\omega_{2,j}$$

By symmetry:

$$\omega_{1,j} = \frac{3\psi_{2,j}}{\Delta x^2} - \frac{1}{2}\omega_{2,j} \tag{32}$$

$$\omega_{N,j} = \frac{3\psi_{N-1,j}}{\Delta x^2} - \frac{1}{2}\omega_{N-1,j} \tag{33}$$

$$\omega_{i,1} = \frac{3\psi_{i,2}}{\Delta y^2} - \frac{1}{2}\omega_{i,2} \tag{34}$$

$$\omega_{i,M} = \frac{3\psi_{i,M-1}}{\Delta y^2} - \frac{1}{2}\omega_{i,M-1} \tag{35}$$

Updating of Temperature T at Boundary

- Left and Right Boundaries: $T_{1,j},\ T_{N,j}$

To approximate the temperature at the left and right boundaries, we use the one-sided second-order Taylor expansion for the first derivative of temperature in the x-direction. This is similar to the boundary condition of vorticity except now our Temperature at the boundaries are not 0.
The second-order accurate one-sided finite difference approximations for the derivative are:

$$\left.\frac{\partial T}{\partial x}\right|_{1,j} = \frac{-3T_{1,j} + 4T_{2,j} - T_{3,j}}{2\Delta x} + \mathcal{O}(\Delta x^2)$$
$$\left.\frac{\partial T}{\partial x}\right|_{N,j} = \frac{3T_{N,j} - 4T_{N-1,j} + T_{N-2,j}}{2\Delta x} + \mathcal{O}(\Delta x^2)$$

Since the physical boundary condition imposes:

$$\left.\frac{\partial T}{\partial x}\right|_{1,j} = \left.\frac{\partial T}{\partial x}\right|_{N,j} = 0$$

we can set the expressions equal to zero and solve for $T_{1,j}$ and $T_{N,j}$:

$$\begin{aligned} T_{1,j} &= \frac{4}{3}T_{2,j} - \frac{1}{3}T_{3,j} \\ T_{N,j} &= \frac{4}{3}T_{N-1,j} - \frac{1}{3}T_{N-2,j} \end{aligned}$$

These formulas are used to iteratively update the temperature values at the left and right boundaries, ensuring a second-order accurate, Neumann-type (zero-gradient) boundary condition.

- Top Boundary: $T_{i,N} = 0$

At the top boundary, we explicitly enforce a Dirichlet boundary condition:

$$\boldsymbol{T_{i,N} = 0}$$

This is necessary to fully constrain the temperature field and prevent divergence of the numerical solution. Without this, the system would be under-determined, leading to instability.

1.5 Update and compute velocity components u, v

Calculating the Velocity Components from the Stream Function

Before computing the velocity components, we first apply the **no-slip boundary condition**, which imposes that the velocity at all boundaries of the domain is zero:

$$u = v = 0 \quad \text{at the boundaries.}$$

This means that we only compute the velocity components at the **inner grid points** of the domain.

Velocity Computation at Inner Grid Points

The velocity components can be derived from the stream function ψ using the following relations:

$$\begin{aligned} u &= \frac{\partial \psi}{\partial y} \approx \frac{\psi_{i,j+1} - \psi_{i,j-1}}{2\Delta y} \\ v &= -\frac{\partial \psi}{\partial x} \approx -\left(\frac{\psi_{i+1,j} - \psi_{i-1,j}}{2\Delta x}\right) \end{aligned}$$

These finite difference approximations are applied only at the inner points of the grid, while maintaining zero velocity at the boundaries, as required by the no-slip condition.

1.6 Sequence in pipeline

Before initialising the iterations all variables are initialised to 0. Stream function could be set to an arbitrary constant but for simplicity I chose 0. The variables initialised are $u, v, T, \omega, \psi, R^{Temp}, R^{\omega}, S$. The sequence for each iteration is as such:

1. **Calculate residual vorticity** R^{ω}
2. **Solve vorticity** ω^{n+1}
3. **Calculate residual stream function** S
4. **Solve stream function** ψ^{n+1}
5. **Update boundary** ψ
6. **Update boundary** ω
7. **Update and compute velocities** u, v
8. **Compute residual temperature** R^{Temp}
9. **Solve temperature** T^{n+1}
10. **Update boundary** T

2 Numerical results

2.1 $Pr = 0.7, Ra = 3.5 \times 10^3$ with mesh size of 61x61

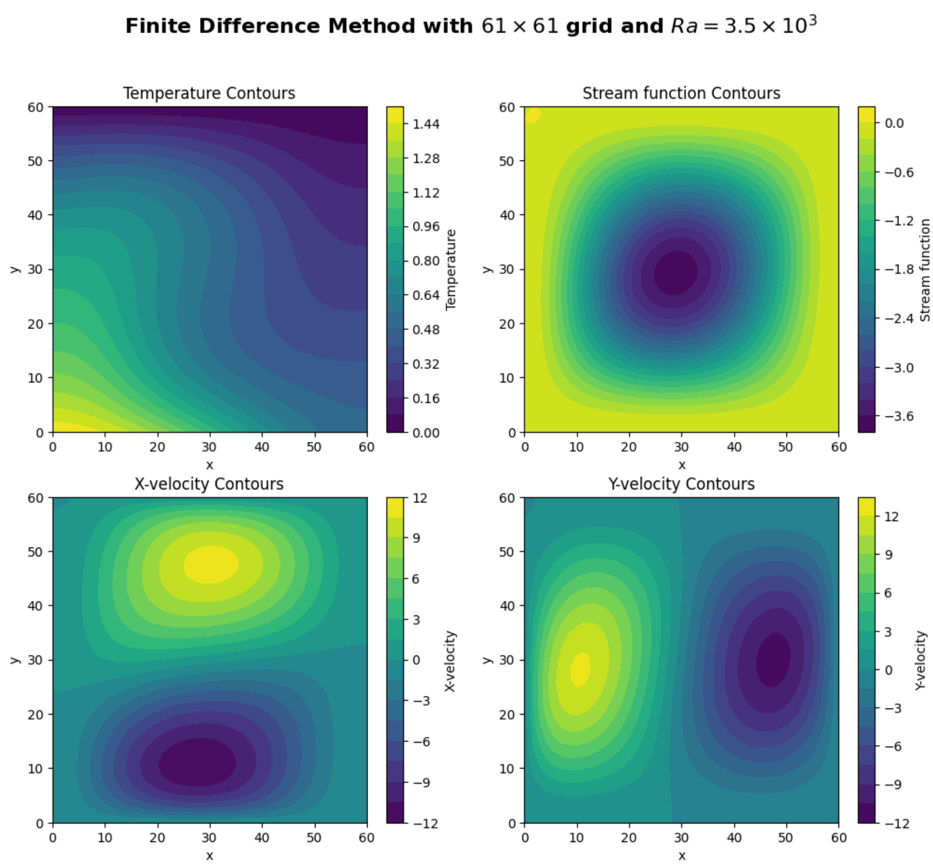

Figure 1: Contour plot of results of $Ra = 3.5 \times 10^3$

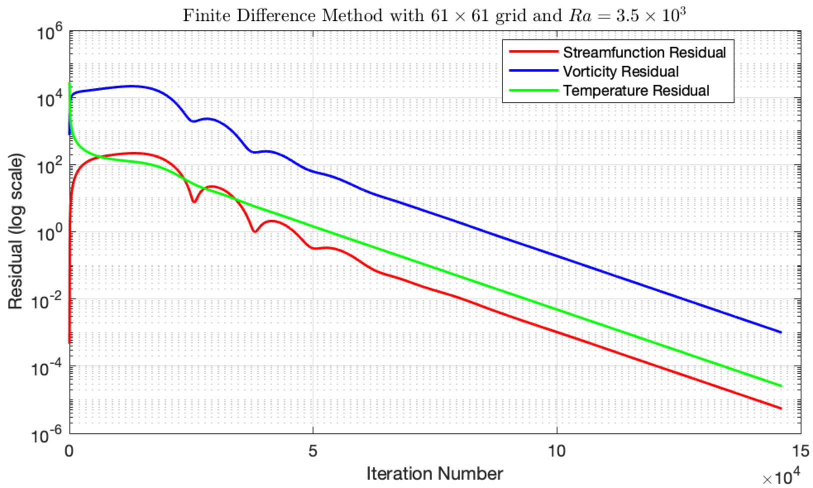

Figure 2: Log convergence plot of residuals of T, ω, ψ generated in Matlab for $Ra = 3.5 \times 10^3$

Parameter	**1e-3 Tolerance**	**1e-4 Tolerance**
Time step	0.000010	0.000010
Total simulation time	39.095450	39.540010
Number of iterations	146123	166358
$u_{\max}$	11.645524 @ $y = 0.1833$	11.645524 @ $y = 0.1833$
$v_{\max}$	12.168678 @ $x = 0.1833$	12.168678 @ $x = 0.1833$
Nu_0 (avg at $y = 0$)	1.908897	1.908897
$Nu_{0.5}$ (avg at $y = 0.5$)	0.191326	0.191326
$Nu_{\max}$ (at $y = 0$)	3.000745 @ $x = 0.3833$	3.000745 @ $x = 0.3833$

Table 1: Simulation results for $Ra = 3.5 \times 10^3$ using finite difference method on a 61×61 grid with tolerances of 1×10^{-3} and 1×10^{-4} on Matlab with a shorter computation time

From table 1 we can see no difference between simulation results when a tolerance of 10^-3 or 10^-4 is used. From the convergence plot in Figure 2 we can also see that the residual errors fluctuate initially and then becoming monotonically decreasing after around 50000 iterations.

2.2 $Pr = 0.7, Ra = 2.5 \times 10^4$ with mesh size of 121x121

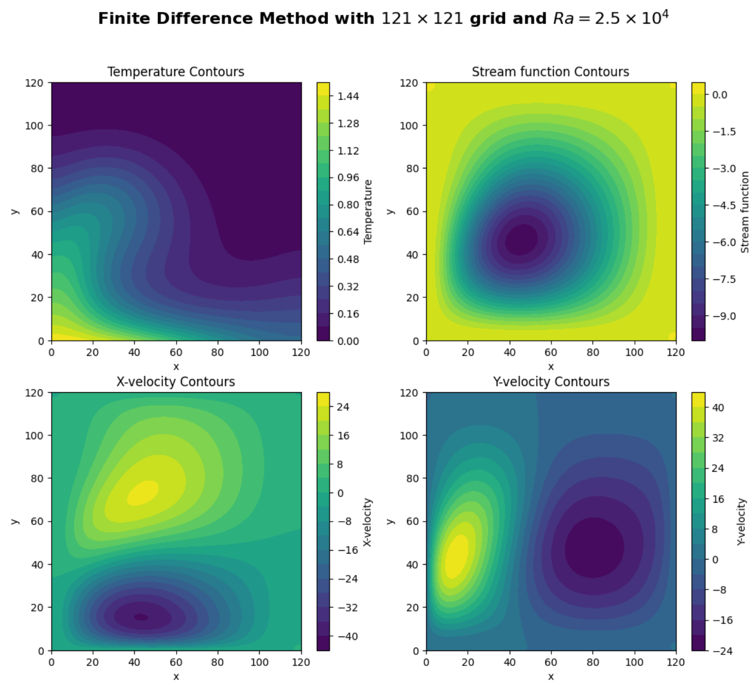

Figure 3: **PARTIAL** contour plot of results of $Ra = 2.5 \times 10^4$ due to too long simulation time in python notebook

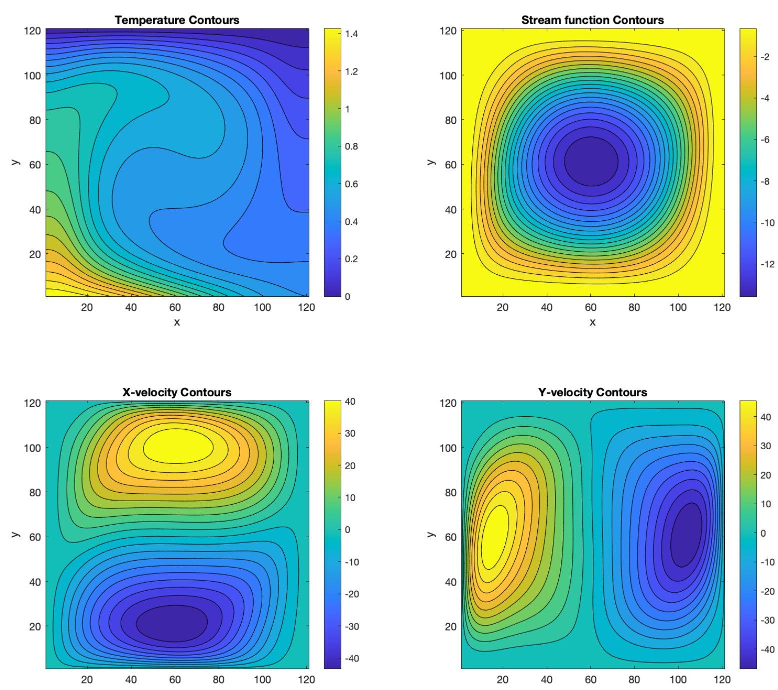

Figure 4: **FULL** contour plot of results of $Ra = 2.5 \times 10^4$ in Matlab

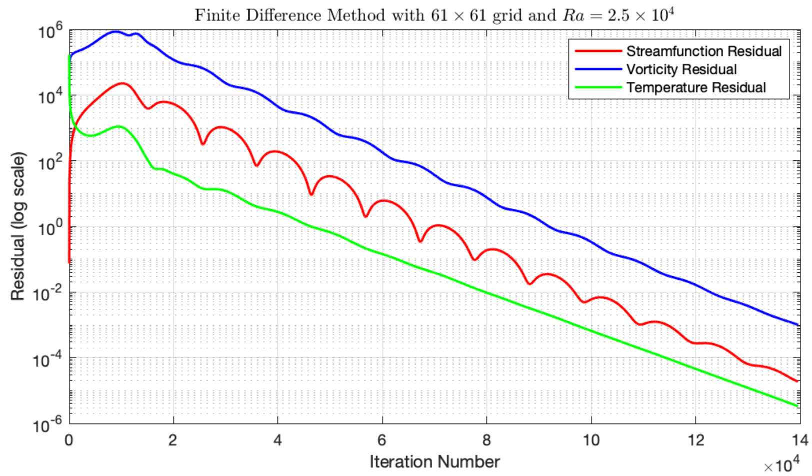

Figure 5: Log convergence plot of residuals of T, ω, ψ generated in Matlab for $Ra = 2.5 \times 10^4$

Parameter	**1e-3 Tolerance**	**1e-4 Tolerance**
Time step	0.0000010	0.0000010
Total simulation time	1523.593627	1363.598932
Number of iterations	1536717	1723736
$u_{\max}$	44.284072 @ $y = 0.8333$	44.284072 @ $y = 0.8333$
$v_{\max}$	50.077433 @ $x = 0.1250$	50.077433 @ $x = 0.1250$
Nu_0 (avg at $y = 0$)	3.391385	3.391385
$Nu_{0.5}$ (avg at $y = 0.5$)	-0.156429	-0.156429
$Nu_{\max}$ (at $y = 0$)	5.495347 @ $x = 0.3583$	5.495347 @ $x = 0.3583$

Table 2: Simulation results for $Ra = 2.5 \times 10^4$ using finite difference method on a 61 × 61 grid with tolerances of 1×10^{-3} and 1×10^{-4} on Matlab with a shorter computation time

From table 2 we can see no difference between simulation results when a tolerance of 10^-3 or 10^-4 is used. From the convergence plot in Figure 5 we can see that the residual of Stream function error fluctuates more than the other 2 errors in comparison to the previous lower Rayleigh number. A higher Rayleigh number signifies a stronger convection property and a transition to a more turbulent flow. This might explain why the residual errors fluctuate more and neccessitate a smaller time discretisation of 10^-6 used instead of 10^-5.

2.2.1 Comparison of Results for Different Rayleigh Numbers and Tolerances

The simulation results for two Rayleigh numbers, $Ra = 3.5 \times 10^3$ and $Ra = 2.5 \times 10^4$, show distinct differences due to the variation in buoyancy-driven flow strength. The results for two convergence tolerances, 10^{-3} and 10^{-4}, are identical in both cases, indicating that the solution accuracy is not significantly impacted by the residual tolerance within this range. The summary of key differences is presented in Table 3.

Parameter	$Ra = 3.5 \times 10^3$	$Ra = 2.5 \times 10^4$
$u_{\max}$	11.65 @ $y = 0.1833$	44.28 @ $y = 0.8333$
$v_{\max}$	12.17 @ $x = 0.1833$	50.08 @ $x = 0.1250$
Nu_0 (avg)	1.91	3.39
$Nu_{\max}$	3.00 @ $x = 0.3833$	5.50 @ $x = 0.3583$

Table 3: Comparison of key parameters between low and high Rayleigh number cases (tolerance has negligible effect)

The increased Rayleigh number in the $Ra = 2.5 \times 10^4$ case represents a stronger thermal driving force (buoyancy) due to a greater temperature difference across the cavity. This results in more vigorous natural convection currents, manifested as significantly higher maximum velocities in both the horizontal and vertical directions. Physically, this occurs because the fluid becomes more unstable at higher Ra, enhancing convective mixing and accelerating the flow within the cavity.

Additionally, the Nusselt numbers—representing the ratio of convective to conductive heat transfer—also increase with Ra, indicating a more dominant convective heat transfer mechanism. The steepening of temperature gradients near the heated wall (as evidenced by higher $Nu_{\max}$) reflects the formation of thinner thermal boundary layers, which is typical in high-Rayleigh-number flows.

3 Code

I used both a python notebook and matlab to code the Finite Difference Method. Python notebook was used to improve readability for the marker, but it was slow especially for the case of $Ra = 2.5 \times 10^5$ so I used Matlab as well. The numerical results in the above tables are all from Matlab while the first contour plot corresponding to $Ra = 3.5 \times 10^4$ is from the python notebook. Both notebook and the Matlab file is attached with this report, and the notebook is also in my github which can be found in the appendix A. I attach also the matlab code in B.

A GitHub Repository

The code and additional resources for this report are available at my github repository here: ME5302-Assignment-2

B MATLAB Code

The following MATLAB code was used in the implementation:

Listing 1: Main MATLAB Script

```
% Finite Difference Method
% of the steady Natural Convection in a square cavity
% Created by ER Qi Yang A0164661A %
clc; close all; clear all;

% Defining parameters
N = 121; % Number of points along x axis
dx = 1.0 / (N - 1); % Grid spacing in x direction
M = 121; % Number of points along y axis
dy = 1.0 / (M - 1); % Grid spacing in y direction
Ra = 2.5e4; % Rayleigh number
Pr = 0.7; % Prandtl number
h = 0.000001; % Time step
beta = 0.4; % relaxation factor for iterative methods to solve algebraic
    equations
tol = 1e-3;

% Initialisation at t=0 with boundary conditions
u = zeros(N, M); % x-velocity
v = zeros(N, M); % y-velocity
T = zeros(N, M); % Temperature
rT = zeros(N, M); % Residual temperature
vor = zeros(N, M); % Vorticity
rvor = zeros(N, M); % Residual vorticity
p = zeros(N, M); % Stream function initialised to be 0
rp = zeros(N, M); % Residual stream function

```

```
% Boundary condition for temperature
for i = 1:N
    T(i, 1) = 0.5 * cos(pi * (i-1) / (N-1)) + 1; % Bottom boundary condition
        of T = 0.5cos(pi*x)+1
end

% Function to calculate vorticity residual
function rvor = resvor(vor, u, v, T, dx, dy, Pr, Ra)
    [N, M] = size(vor);
    rvor = zeros(size(vor));
    for i = 2:(N-1)
        for j = 2:(M-1)
            dvorx2 = (vor(i+1, j) - 2*vor(i, j) + vor(i-1, j)) / (dx^2);
            dvory2 = (vor(i, j+1) - 2*vor(i, j) + vor(i, j-1)) / (dy^2);
            dvorx1 = u(i, j) * (vor(i+1, j) - vor(i-1, j)) / (2*dx);
            dvory1 = v(i, j) * (vor(i, j+1) - vor(i, j-1)) / (2*dy);
            dTx = (T(i+1, j) - T(i-1, j)) / (2*dx);

            rvor(i, j) = (dvorx2 + dvory2) * Pr - Pr * Ra * dTx - dvorx1 -
                dvory1;
        end
    end
end

% Function to calculate temperature residual
function rtemp = restemp(T, u, v, dx, dy)
    [N, M] = size(T);
    rtemp = zeros(size(T));
    for i = 2:(N-1)
        for j = 2:(M-1)
            dTx2 = (T(i+1, j) - 2*T(i, j) + T(i-1, j)) / (dx^2);
            dTy2 = (T(i, j+1) - 2*T(i, j) + T(i, j-1)) / (dy^2);
            dTx1 = u(i, j) * (T(i+1, j) - T(i-1, j)) / (2*dx);
            dTy1 = v(i, j) * (T(i, j+1) - T(i, j-1)) / (2*dy);

            rtemp(i, j) = dTx2 + dTy2 - dTx1 - dTy1;
        end
    end
end

% Function to solve temperature field
function T = solT(T, rT, h, method)
    [N, M] = size(T);

    if strcmp(method, 'euler')
        T(2:N-1, 2:M-1) = T(2:N-1, 2:M-1) + h * rT(2:N-1, 2:M-1);
    end
end

% Function to solve vorticity field
function vor = solvor(beta, vor, rvor, h, dx, dy, Pr, method)
    [N, M] = size(vor);

    if strcmp(method, 'euler')
        vor(2:N-1, 2:M-1) = vor(2:N-1, 2:M-1) + h * rvor(2:N-1, 2:M-1);
    end
end

% Calculation of residual of the Poisson equation
function rp = resp(p, vor, dx, dy)
    [N, M] = size(p);
    rp = zeros(size(p));
```

```
    for i = 3:(N-2)
        for j = 3:(M-2)
            rp(i, j) = vor(i, j) ...
                - (p(i+1, j) - 2*p(i, j) + p(i-1, j)) / (dx^2) ...
                - (p(i, j+1) - 2*p(i, j) + p(i, j-1)) / (dy^2);
        end
    end
end

% Function to solve stream function
function p = solp(p, rp, dx, dy, beta)
    [N, M] = size(p);
    % Coefficients for iterative methods
    b_W = 1 / dx^2;
    b_S = 1 / dy^2;
    b_P = -2 * (b_W + b_S);

    for i = 3:(N-2)
        for j = 3:(M-2)
            p(i, j) = p(i, j) + beta * rp(i, j) / b_P;
        end
    end
end

% Function to apply boundary conditions to stream function
function p = BCp(p)
    [N, M] = size(p);
    % Update p along the vertical boundaries
    for j = 2:(M-1)
        p(2, j) = 0.25 * p(3, j);  % Left
        p(N-1, j) = 0.25 * p(N-2, j);  % Right
    end

    % Update p along the horizontal boundaries
    for i = 2:(N-1)
        p(i, 2) = 0.25 * p(i, 3); % Bottom
        p(i, M-1) = 0.25 * p(i, M-2); % Top
    end

    % Update p at the boundaries
    p(1, :) = 0; % Left
    p(N, :) = 0; % Right
    p(:, 1) = 0; % Bottom
    p(:, M) = 0; % Top
end

% Function to apply boundary conditions to vorticity
function vor = BCvor(vor, p, dx, dy)
    [N, M] = size(vor);
    % Update vorticity at the boundaries using 2nd order approximation
    for j = 1:M
        vor(1, j) = 3.0 * p(2, j) / (dx^2) - 0.5 * vor(2, j);
        vor(N, j) = 3.0 * p(N-1, j) / (dx^2) - 0.5 * vor(N-1, j);
    end

    % Update along the horizontal boundaries (i-loop)
    for i = 2:(N-1)
        vor(i, 1) = 3.0 * p(i, 2) / (dy^2) - 0.5 * vor(i, 2);
        vor(i, M) = 3.0 * p(i, M-1) / (dy^2) - 0.5 * vor(i, M-1);
    end

```

```
end

% Function to apply boundary conditions to temperature
function T = BCT(T)
    [N, M] = size(T);
    % Update temperature at the left boundary
    for j = 1:M
        T(1, j) = (4/3) * T(2, j) - (1/3) * T(3, j);
    end

    % Update temperature at the right boundary
    for j = 1:M
        T(N, j) = (4/3) * T(N-1, j) - (1/3) * T(N-2, j);
    end

    % Update temperature at the top boundary (added: isothermal condition T=0)
    for i = 1:N
        T(i, N) = 0.0;
    end
end

% Function to calculate velocity components from stream function
function [u, v, T] = caluv(T, u, v, p, dx, dy)
    [N, M] = size(u);
    % Apply physical boundary conditions of 0 velocity
    for j = 1:M
        u(1, j) = 0;
        u(N, j) = 0;
        v(1, j) = 0;
        v(N, j) = 0;
    end

    for i = 2:(N-1)
        u(i, 1) = 0;
        v(i, 1) = 0;
        u(i, M) = 0;
        v(i, M) = 0;
    end

    % Update velocity components based on stream function
    for i = 2:(N-1)
        for j = 2:(M-1)
            u(i, j) = 0.5 * (p(i, j+1) - p(i, j-1)) / dy;
            v(i, j) = 0.5 * (p(i-1, j) - p(i+1, j)) / dx;
        end
    end
end

% Initialise errors for convergence check
iter_no = 0;
errp_list = [];
errvor_list = [];
errT_list = [];
iter_list = [];

% Start timer
tic;

% Main simulation loop
while true
    % Compute residual vorticity and update vorticity
    rvor = resvor(vor, u, v, T, dx, dy, Pr, Ra);
```

```
    vor = solvor(beta, vor, rvor, h, dx, dy, Pr, 'euler');

    % Compute residual Poisson equation and update stream function
    rp = resp(p, vor, dx, dy);
    p = solp(p, rp, dx, dy, beta);

    % Update boundary conditions for stream function
    p = BCp(p);

    % Update boundary conditions for vorticity
    vor = BCvor(vor, p, dx, dy);

    % Update velocity components based on stream function
    [u, v, T] = caluv(T, u, v, p, dx, dy);

    % Compute residual temperature and update temperature
    rT = restemp(T, u, v, dx, dy);
    T = solT(T, rT, h, 'euler');

    % Update Temperature field
    T = BCT(T);

    % Update iteration number
    iter_no = iter_no + 1;

    % Calculate errors
    errvor = sqrt(sum(sum(rvor.^2)));
    errp = sqrt(sum(sum(rp.^2)));
    errT = sqrt(sum(sum(rT.^2)));
    errp_list(end+1) = errp;
    errvor_list(end+1) = errvor;
    errT_list(end+1) = errT;
    iter_list(end+1) = iter_no;


    if mod(iter_no, 100) == 0
        fprintf('Iteration number %d, errp: %f, errvor: %f, errT: %f\n', iter_no, errp, errvor, errT);
    end

        % Check convergence
    if errp < tol && errvor < tol && errT < tol
        converged = true;
        fprintf('Converged at iteration %d: errp = %f, errvor = %f, errT = %f\n', iter_no, errp, errvor, errT);
        break;
    end
end

% End timer and calculate elapsed time
elapsed_time = toc;

fprintf('My time step is: %f\n', h);
fprintf('Total time elapsed is: %f\n', h*iter_no);
fprintf('Total time taken for the simulation: %f seconds\n', elapsed_time);

% Create figure with 2x2 subplots
figure('Position', [100, 100, 900, 750]);

% Plot Temperature contours
subplot(2, 2, 1);
[~, contour1] = contourf(T', 20);
```

```
colorbar;
title('Temperature Contours');
xlabel('x');
ylabel('y');

% Plot Stream function contours
subplot(2, 2, 2);
[~, contour2] = contourf(p', 20);
colorbar;
title('Stream function Contours');
xlabel('x');
ylabel('y');

% Plot X-velocity contours
subplot(2, 2, 3);
[~, contour3] = contourf(u', 20);
colorbar;
title('X-velocity Contours');
xlabel('x');
ylabel('y');

% Plot Y-velocity contours
subplot(2, 2, 4);
[~, contour4] = contourf(v', 20);
colorbar;
title('Y-velocity Contours');
xlabel('x');
ylabel('y');

figure;
semilogy(iter_list, errp_list, 'r', 'LineWidth', 1.5); hold on;
semilogy(iter_list, errvor_list, 'b', 'LineWidth', 1.5);
semilogy(iter_list, errT_list, 'g', 'LineWidth', 1.5);
grid on;

title('Finite Difference Method with $61 \times 61$ grid and $Ra = 2.5 \times
    10^4$', 'Interpreter', 'latex');
xlabel('Iteration Number');
ylabel('Residual (log scale)');
legend('Streamfunction Residual', 'Vorticity Residual', 'Temperature Residual'
    , 'Location', 'best');


% --- Inputs ---
% u, v: Velocity fields (size NxM)
% T: Temperature field (size NxM)
% dx, dy: Grid spacings
% N, M: Number of grid points in x and y directions

%% (1) Calculate umax (max horizontal velocity on vertical mid-plane x=0.5)
x_mid_idx = round(N/2);  % Index of vertical mid-plane (x=0.5)
u_midplane = u(x_mid_idx, :);  % Horizontal velocity on x=0.5
[umax, umax_idx] = max(abs(u_midplane));  % Maximum magnitude and its index
umax_y_location = (umax_idx - 1) * dy;  % y-coordinate of umax
umax_sign = sign(u_midplane(umax_idx));  % Sign of umax (direction)

fprintf('umax = %.6f at y = %.4f (direction: %d)\n', umax, umax_y_location,
    umax_sign);

%% (2) Calculate vmax (max vertical velocity on horizontal mid-plane y=0.5)
y_mid_idx = round(M/2);  % Index of horizontal mid-plane (y=0.5)
v_midplane = v(:, y_mid_idx);  % Vertical velocity on y=0.5
```

```
[vmax, vmax_idx] = max(abs(v_midplane));  % Maximum magnitude and its index
vmax_x_location = (vmax_idx - 1) * dx;  % x-coordinate of vmax
vmax_sign = sign(v_midplane(vmax_idx));  % Sign of vmax (direction)

fprintf('vmax = %.6f at x = %.4f (direction: %d)\n', vmax, vmax_x_location,
    vmax_sign);

%% (3) Calculate Nu0 (avg Nusselt number at bottom wall y=0)
dTdy_bottom = (T(:, 2) - T(:, 1)) / dy;  % Temperature gradient at y=0 (
    forward difference)
Nu_local_bottom = -dTdy_bottom;  % Local Nusselt number (T_wall = T(:,1))
Nu0 = mean(Nu_local_bottom);  % Average Nusselt number at y=0

fprintf('Nu0 (avg at y=0) = %.6f\n', Nu0);

%% (4) Calculate Nu1 (avg Nusselt number at mid-plane y=0.5)
dTdy_mid = (T(:, y_mid_idx+1) - T(:, y_mid_idx-1)) / (2*dy);  % Central
    difference at y=0.5
Nu_local_mid = -dTdy_mid;  % Local Nusselt number at y=0.5
Nu1 = mean(Nu_local_mid);  % Average Nusselt number at y=0.5

fprintf('Nu1 (avg at y=0.5) = %.6f\n', Nu1);

%% (5) Calculate Numax (max Nusselt number at bottom wall y=0)
[Numax, Numax_idx] = max(Nu_local_bottom);  % Max Nusselt number and its index
Numax_x_location = (Numax_idx - 1) * dx;  % x-coordinate of Numax

fprintf('Numax (max at y=0) = %.6f at x = %.4f\n', Numax, Numax_x_location);
```