# Numerical Solution of the Lid Driven Cavity Problem

Giada Alessi
Master in Thermal Engineering
Universitat Politècnica de Catalunya

March 21, 2025

## Contents

**Abstract**

This report presents the numerical solution of the Lid Driven Cavity problem using the Fractional Step Method. The steady-state velocity field is obtained for Reynolds numbers $Re = 100$ and $Re = 400$, and results are compared to reference data. Key plots include velocity profiles at domain mid-sections and velocity magnitude colormaps.

# 1 Introduction

The Lid Driven Cavity problem consists of a square cavity of size $L \times L$ filled with an incompressible fluid, with viscosity $\mu$ and density $\rho$. The bottom, left, and right walls are stationary, meaning that the velocity components $u$ and $v$ are both zero on these boundaries:

$$u = 0, \quad v = 0 \quad \text{on bottom, left, and right walls.} \tag{1}$$

The top wall, however, moves with a constant velocity $u(x, y = L) = U_{ref}$. The pressure boundary condition for all walls is given by:

$$\frac{\partial P}{\partial n} = 0. \tag{2}$$

This means that there is no pressure gradient normal to the walls.

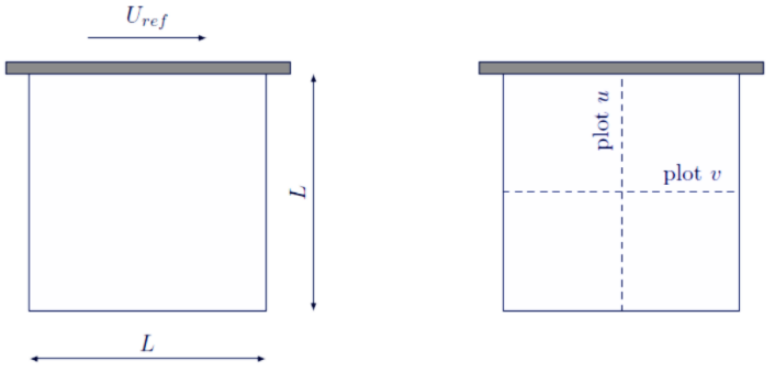



Figure 1: Lid driven cavity problem geometry and velocity plot locations.

The left image represents the physical domain with a moving top wall at velocity $U_{ref}$, while the right diagram indicates the locations where velocity profiles $u(y)$ at $x = 0.5L$ and $v(x)$ at $y = 0.5L$ are extracted. The flow characteristics are defined by the Reynolds number, calculated as:

$$Re = \frac{\rho U_{ref} L}{\mu} \tag{3}$$

In this study, the following parameters have been considered:

- Density: $\rho = 1.0 \;\; kg/m^3$
- Velocity of the lid: $U_{ref} = 1.0 \;\; m/s$

- Characteristic length: $L = 1 \ \ m$
- Dynamic viscosity: $\mu = \frac{\rho U_{ref} L}{Re}$

The objective of this analysis is to solve the case for $Re = 100$ and $Re = 400$ until the steady state is reached, to generate and compare the velocity profiles $u(y)$ at $x = 0.5$ and $v(x)$ at $y = 0.5$ with the reference case, and to visualize the velocity magnitude using color maps.

# 2 Theoretical Background

The motion of an incompressible Newtonian fluid, such as water or air at low velocities ($< 100$ m/s), can be described by a reduced form of the Navier-Stokes equations. For this exercise, the equations have been simplified by neglecting unnecessary terms, such as the gravity term ($\rho \cdot g$), which is approximated in the Boussinesq approximation in case of free/natural convection.

- **Mass Conservation:**

$$\nabla \cdot \mathbf{v} = 0 \tag{4}$$

  This equation states that mass is conserved if the density remains constant.

- **Momentum Conservation:**

$$\rho \frac{\partial \mathbf{v}}{\partial t} + \rho(\mathbf{v} \cdot \nabla)\mathbf{v} = -\nabla P + \mu \nabla^2 \mathbf{v} \tag{5}$$

  This equation is derived from Newton's Second Law of Motion and applies to forced convection.

- **Energy Conservation:**

$$\rho c_p \left( \frac{\partial T}{\partial t} + \nabla \cdot (vT) \right) = \nabla \cdot (\lambda \nabla T) + \dot{q}_v \tag{6}$$

  This is the simplified energy equation that describes the temperature field.

# 3 Numerical Implementation

In this section, key numerical concepts such as the Fractional Step Method and staggered meshes are introduced, as they are essential for solving the problem. Then, the spatial and temporal discretization of the governing equations and the overall solution algorithm are presented.

## 3.1 Fractional Step Method

When solving the Navier-Stokes equations, we aim to determine both the velocity and pressure fields simultaneously. If the discretization is not well-posed, spurious numerical solutions (such as a checkerboard pressure pattern) may arise, leading to incorrect velocity values. To address this issue, the Fractional Step Method (FSM) is introduced as an alternative numerical approach.

The FSM separates the computation of the velocity field into three steps: an intermediate prediction, a correction step, and the final computation of the velocity field. The steps are described as follows:

- **Step 1:**
$$\mathbf{v}^p = \mathbf{v}^n + \frac{\Delta t}{\rho}\left[\frac{3}{2}\mathbf{R}(\mathbf{v}^n) - \frac{1}{2}\mathbf{R}(\mathbf{v}^{n-1})\right] \tag{7}$$
- **Step 2:**
$$\nabla^2 P^{n+1} = \frac{\rho}{\Delta t}\nabla \cdot \mathbf{v}^p \tag{8}$$
- **Step 3:**
$$\mathbf{v}^{n+1} = \mathbf{v}^p - \frac{\Delta t}{\rho}\nabla P^{n+1} \tag{9}$$

In this method, the pressure field acts as a 'corrector' for the velocity, making the FSM a projection into a divergence-free velocity space that enforces the incompressibility constraint.

Despite the temporal discretization already shown in Equations 7, 8, and 9 — which will be further discussed in the dedicated chapter on *Equation Discretization* — the **Helmholtz-Hodge Theorem** has been applied to decouple the velocity and pressure gradient calculations by defining the predictor velocity $\mathbf{v}^p$.

In particular, the Helmholtz-Hodge theorem allows us to decompose a vector field into a divergence-free vector (the velocity field $\mathbf{v}$, where $\nabla \cdot \mathbf{v} = 0$) and the gradient of a scalar field (the pressure gradient $\nabla P$):

$$\boldsymbol{\omega} = \mathbf{A} + \nabla\phi \quad \rightarrow \quad \boldsymbol{\omega} = \mathbf{v} + \nabla P \tag{10}$$

## 3.2 Staggered Meshes

Instability problems can arise when solving the Navier-Stokes equations with the FSM using a collocated mesh, where both pressure and velocity are stored on the same grid. In such cases, the pressure gradient calculation may become imprecise, leading to numerical oscillations. As a direct consequence, if the pressure field exhibits a checkerboard pattern, the pressure gradient computed at velocity locations will be incorrect, resulting in physically unrealistic velocity values.

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

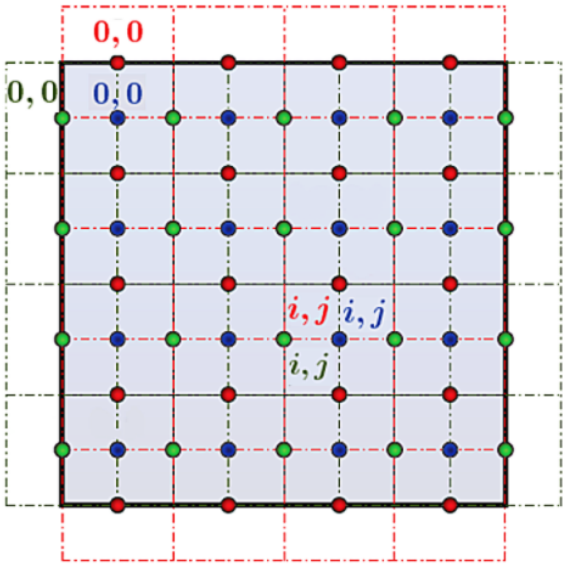



Figure 2: Staggered mesh representation: pressure mesh in blue, horizontal velocity mesh in green, and vertical velocity mesh in red.

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

In the implemented code for this exercise, matrices have been defined using

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

This staggered grid arrangement and the corresponding indexing help to prevent the checkerboard problem and similar numerical instabilities, ensuring a more stable and physically accurate solution.

## 3.3 Equation Discretization

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

### 3.3.1 Temporal Discretization

The second-order Adams-Bashforth explicit time integration method is applied to the momentum equation 2, resulting in:

$$\rho\frac{\mathbf{v}^{n+1} - \mathbf{v}^n}{\Delta t} = -\nabla P^{n+1} + \frac{3}{2}\mathbf{R}(\mathbf{v}^n) - \frac{1}{2}\mathbf{R}(\mathbf{v}^{n-1}) \tag{11}$$

where, for simplicity, the convection and diffusion terms are grouped as follows:

$$\mathbf{R}(\mathbf{v}) = -\rho(\mathbf{v}\cdot\nabla)\mathbf{v} + \mu\nabla^2\mathbf{v} \tag{12}$$

With this approach, the convective terms are evaluated at $n + \frac{1}{2}$ using an interpolation between the previous and pre-previous timesteps. A different time discretization is applied for the Poisson equation, which is evaluated implicitly at $t^{n+1}$, as shown in equation 8.
The predictor velocity $\mathbf{v}^p$, defined using the Helmholtz-Hodge theorem, is also discretized implicitly as follows:

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

By isolating $\mathbf{v}^{n+1}$ from Equation 13, we obtain the final equation 9 (step 3) of the previously described FSM. Then, substituting Equation 9 into Equation 11, Equation 7 is obtained (step 1). By applying the divergence operator to Equation 7, the Poisson equation 8 is derived (step 2).
At this point, the three-step equations of the Fractional Step Method are fully defined and discretized in time.

### 3.3.2 Spatial Discretization

The spatial discretization is performed using the finite volume method on the staggered grid described earlier. For simplicity, the grid is built uniformly, with $\Delta x = \Delta y$ constant across the meshes, and with constant $\rho$ and $\mu$. This approach simplifies the equations, considering, for example, that $\Delta S/d_{nb} = 1$ and $\Delta V = (\Delta x)^2$, etc.
Integrating equation 11 over a control volume and substituting the $R()$ term for the two different velocity components, $u$ and $v$, the following discretized equations are obtained:

- **Stagg-x mesh:**

$$u^p = u^n + \frac{\Delta t}{\rho \Delta V}\left[\frac{3}{2}R(u^n) - \frac{1}{2}R(u^{n-1})\right] \tag{14}$$

where:

$$R(u) = -conv_x + diff_x \tag{15}$$
$$conv_x = \rho \Delta x(v_n u_n - v_s u_s + u_e u_e - u_w u_w) \tag{16}$$
$$diff_x = \mu(u_N + u_S + u_E + u_W - 4u_P) \tag{17}$$

And, according to the indexing described previously:

$$v_n = \frac{1}{2}(v[i][j-1] + v[i][j]), \quad u_n = \frac{1}{2}(u[i][j] + u[i-1][j]$$
$$v_s = \frac{1}{2}(v[i+1][j-1] + v[i+1][j]), \quad u_s = \frac{1}{2}(u[i][j] + u[i+1][j])$$
$$u_e = \frac{1}{2}(u[i][j] + u[i][j+1]), \quad u_w = \frac{1}{2}(u[i][j] + u[i][j-1])n$$
$$u_N = u[i-1][j], \quad u_S = u[i+1][j], \quad u_E = u[i][j+1], \quad u_W = u[i][j-1]$$

- **Stagg-y mesh:**

$$v^p = v^n + \frac{\Delta t}{\rho \Delta V}\left[\frac{3}{2}R(v^n) - \frac{1}{2}R(v^{n-1})\right] \tag{18}$$

where:

$$R(v) = -conv_y + diff_y \tag{19}$$
$$conv_y = \rho \Delta x(v_n v_n - v_s v_s + u_e v_e - u_w v_w) \tag{20}$$
$$diff_y = \mu(v_N + v_S + v_E + v_W - 4v_P) \tag{21}$$

And, according to the indexing described previously:

$$v_n = \frac{1}{2}(v[i][j] + v[i-1][j]), \quad v_s = \frac{1}{2}(v[i][j] + v[i+1][j]$$
$$u_e = \frac{1}{2}(u[i-1][j+1] + u[i][j+1]), \quad v_e = \frac{1}{2}(v[i][j] + v[i][j+1])$$
$$u_w = \frac{1}{2}(u[i-1][j] + u[i][j]), \quad v_w = \frac{1}{2}(v[i][j] + v[i][j-1])n$$
$$v_N = v[i-1][j], \quad v_S = v[i+1][j], \quad v_E = v[i][j+1], \quad v_W = v[i][j-1]$$

- **Stagg-P mesh:**

$$P_N + P_S + P_E + P_W - 4P_P = \frac{\rho \Delta S}{\Delta t}(v_n^p - v_s^p + u_e^p - u_w^p) \tag{22}$$

And, according to the indexing described previously:

$$P_S = P[i+1][j] \quad P_E = P[i][j+1] \quad P_N = P[i-1][j] \quad P_W = P[i][j-1]$$
$$n_n^p = v^p[i][j] \quad v_s^p = v^p[i+1][j] \quad u_e^p = u^p[i][j+1] \quad u_w^p = u^p[i][j]$$

It should be noted that, in this case, a CDS scheme has been implemented in the algorithm to solve the convective term.

## 3.4 Boundary Conditions

The boundary conditions mentioned in the introduction have been applied to both the velocity meshes and the pressure mesh. The imposed conditions for each mesh are described below:

- **Horizontal velocity $u$ mesh:**
  The reference velocity $U_{ref}$ has been applied to the first row of control volumes, corresponding to the part of the fluid in contact with the moving lid. The horizontal velocity $u$ at all other boundaries has been set to zero:

$$u[0][j] = U_{ref}, \quad u[Ny-1][j] = u[i][0] = u[i][Nx] = 0 \tag{23}$$

- **Vertical velocity $v$ mesh:**
  The vertical velocity component has been set to zero at all boundaries, since the lid affects only the $u$-component and not the $v$-component of velocity:

$$v[0][j] = v[Ny][j] = v[i][0] = v[i][Nx-1] = 0 \tag{24}$$

- **Pressure mesh:**
  The Neumann boundary condition for pressure has been imposed at all walls and is given by:
  $$\frac{\partial P}{\partial n} = 0 \tag{25}$$
  which results in the following pressure boundary conditions:

  - **Top wall:**
  $$P^{n+1} = P_S^{n+1} - \frac{\rho \Delta x}{\Delta t}(v_n^p - v_s^p + u_e^p - u_w^p) \tag{26}$$
  - **Bottom wall:**
  $$P^{n+1} = P_N^{n+1} - \frac{\rho \Delta x}{\Delta t}(v_n^p - v_s^p + u_e^p - u_w^p) \tag{27}$$
  - **Left wall:**
  $$P^{n+1} = P_W^{n+1} - \frac{\rho \Delta x}{\Delta t}(v_n^p - v_s^p + u_e^p - u_w^p) \tag{28}$$
  - **Right wall:**
  $$P^{n+1} = P_E^{n+1} - \frac{\rho \Delta x}{\Delta t}(v_n^p - v_s^p + u_e^p - u_w^p) \tag{29}$$

It is important to note that the pressure boundary conditions have been derived using an approximation based on the assumption that the boundary nodes of the pressure mesh are located exactly at the cavity boundaries. In reality, this is not the case, yet this approach provides a reasonable approximation.

Moreover, the velocity boundary conditions are constant and independent of the fluid motion inside the cavity. Consequently, they are declared only once at the beginning of the simulation. In contrast, the pressure boundary conditions are updated at each time step, since they depend on the predictor velocity values from the previous time step, which are iteratively recalculated throughout the simulation.

## 3.5 Step-by-Step Algorithm

Merging all together the informations given before, the step-by-step resolution algorithm is presented and detailed.
The numerical solution is obtained following this procedure:

- Definition of global variables and matrices outside of **main()** to be used in multiple functions:
  - Pressure matrices: `P0`, `P1`, `Pg`.
  - Horizontal velocity matrices: `un_1`, `un`, `u1`, `uP`.
  - Vertical velocity matrices: `vn_1`, `vn`, `v1`, `vP`.
  - Convective-diffusive term matrices: `Ru`, `Run`, `Run_1`, `Rv`, `Rvn`, `Rvn_1`.
- Definition of the necessary functions:
  - **BoundaryConditions()** for defining boundary conditions of velocities u and v at any istant.
  - **ComputeRu()** and **ComputeRv()** for calculating convective-diffusive terms; these functions are modular, allowing different velocity matrices to be passed as arguments to compute and at different time steps without the need to create additional functions.
  - **PoissonSolver()** for solving the pressure correction equation; this function implements the Gauss-Seidel solver to compute the pressure values.
- Inside **main()**:
  - Definition of physical and numerical parameters.
  - Initialization of velocity fields (internal nodes only) and pressure field (each node) with zero values.
  - Application of constant boundary conditions of velocities using **BoundaryConditions()**.
  - Computation of initial convective-diffusive terms for each internal node of the velocity meshes using **ComputeRu()** and **ComputeRv()**: this step computes $R_u$ and $R_v$ at time steps $n$ and $n-1$ using the initial values of $u$ and $v$.

After these preliminary steps, the steady-state solution is iteratively obtained using the predictor-corrector scheme (FSM) within a while loop:

- Initialization of the time counter to zero.
- Entry into the steady-state convergence loop:

– **Predictor Step:** Compute predicted velocities at each internal node of the velocity meshes:

$$u^P = u^n + \frac{\Delta t}{\rho \Delta x^2}\left(\frac{3}{2}R(u^n) - \frac{1}{2}R(u^{n-1})\right) \tag{30}$$

$$v^P = v^n + \frac{\Delta t}{\rho \Delta x^2}\left(\frac{3}{2}R(v^n) - \frac{1}{2}R(v^{n-1})\right) \tag{31}$$

– **Pressure Correction Step:** Solve the Poisson equation using the Gauss-Seidel method implemented in the function **PoissonSolver()**:

$$P^{n+1} = \frac{1}{4}\left[P_N^{n+1} + P_S^{n+1} + P_E^{n+1} + P_W^{n+1} - \frac{\rho \Delta x}{\Delta t}(v_n^p - v_s^p + u_e^p - u_w^p)\right] \tag{32}$$

simultaneously applying the boundary conditions related to the pressure field, implemented inside the same **PoissonSolver()** function.

– **Evaluation Step:** Compute corrected velocity fields using the values evaluated in the previous two steps:

$$u^{n+1} = u^P - \frac{\Delta t}{\rho}\frac{(P^{n+1} - P_W^{n+1})}{\Delta x} \tag{33}$$

$$v^{n+1} = v^P - \frac{\Delta t}{\rho}\frac{(P_N^{n+1} - P^{n+1})}{\Delta x} \tag{34}$$

– Assessment of convergence based on the maximum residual, ensuring that the changes in $u$ and $v$ remain below a predefined tolerance:

$$\max(\text{Res}) < 10^{-6} \tag{35}$$

– If convergence is reached, exit the loop and print results in .txt files to be used for plotting with Gnuplot.

– Then, if needed, update the time counter by adding $\Delta t$ and continue.

– Update velocity fields:

$$u^{n-1} = u^n, \quad u^n = u^{n+1}$$
$$v^{n-1} = v^n, \quad v^n = v^{n+1}$$

- Compute $Ru()^{n+1}$ and $Rv()^{n+1}$ with **ComputeRu()** and **ComputeRv()** functions, using new values of velocity calculated in the Evaluation step.
- Update convective-diffusive terms:

$$Ru()^{n-1} = Ru()^n \quad Ru()^n = Ru()^{n+1} \tag{36}$$

$$Rv()^{n-1} = Rv()^n \quad Rv()^n = Rv()^{n+1} \tag{37}$$

- Restart while loop.

- The iterative process is repeated until steady-state conditions are met, meaning that the difference between consecutive iterations remains below the prescribed tolerance.
- The plotting is managed directly within the main code, which creates a script and runs it automatically without requiring manual interaction with the Gnuplot software.

# 4 Results and Discussion

In this section, the results are presented in two stages: preliminary results obtained with the initial code version previously described, and refined results achieved after implementing code improvements. The need for these improvements emerged during the following analysis of the preliminary results and a subsequent code review, which highlighted opportunities to enhance both the code structure — making it more modular and flexible — and the result manipulation process to improve accuracy. Additionally, an error analysis was introduced to assess how different mesh refinements affect the solution's performance.

## 4.1 Initial Results

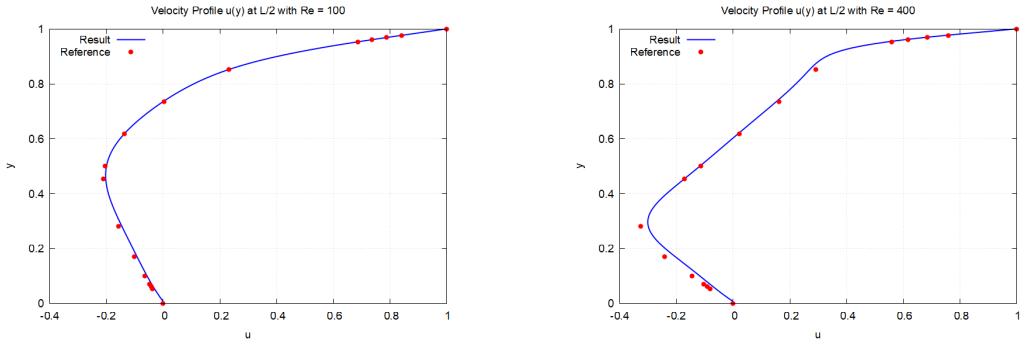
In this study, the recommended CFL condition for this type of problem did not lead to convergence. The stability condition for the convective and diffusive terms, shown in the following expressions:

$$C_{conv} = \frac{|\mathbf{v}|\Delta t}{\Delta x} < 0.35 \qquad C_{diff} = \frac{\mu \Delta t}{\rho \Delta x^2} < 0.2 \tag{38}$$

proved insufficient to ensure convergence for the selected parameters. Instead, a reduced time step of $\Delta t = 5^{-4} s$ was adopted, which successfully achieved convergence.

For the case of $Re = 100$, this adjusted time step allowed the solution to converge in less than two minutes. The convergence tolerance for the Gauss-Seidel method applied to the pressure field was set to $10^{-6}$, which is the same tolerance adopted for the time step convergence criterion. Similarly, for the $Re = 400$ case, the same reduced time step was employed, and convergence was achieved without issues in less than three minutes.
This result demonstrates that the implemented code performs reliably and efficiently across different Reynolds numbers. However, it is likely that adopting an upwind differencing scheme (UDS) instead of the applied central differencing scheme (CDS) could have improved convergence performance, though it would introduce additional numerical diffusion.

The numerical results obtained from the simulation are presented and compared with the reference data provided by Ghia et al.[1]
The velocity profiles at $x = 0.5$ and $y = 0.5$ are plotted to evaluate the solution's accuracy. Figures 3 and 4 display these profiles for Reynolds numbers equal to 100 and 400. Each plot is accompanied by corresponding numerical data presented in Tables 1 and 2 for a more detailed comparison.



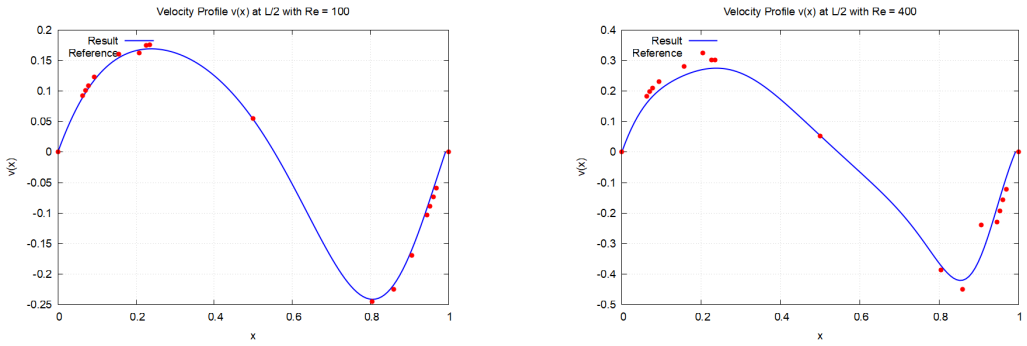

(a) $Re = 100$ (b) $Re = 400$

Figure 3: Velocity profiles $u(y)$ at $x = 0.5$ compared to reference data for different Reynolds numbers.

[1]GHIA, U. K. N. G.; GHIA, Kirti N.; SHIN, C. T. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics*, 1982, 48.3: 387-411.

Table 1: Values of u(y) at x = L/2 for different Re numbers - Comparison between reference data and obtained results.

| $y/L$ | $Re = 100$ | | $Re = 400$ | |
|---|---|---|---|---|
| | Reference | Results | Reference | Results |
| 1.0000 | 1.00000 | 1.00000 | 1.00000 | 1.00000 |
| 0.9766 | 0.84123 | 0.841629 | 0.75837 | 0.751334 |
| 0.9688 | 0.78871 | 0.789282 | 0.68439 | 0.675528 |
| 0.9609 | 0.73722 | 0.737964 | 0.61756 | 0.606951 |
| 0.9531 | 0.68717 | 0.688062 | 0.55892 | 0.546595 |
| 0.8516 | 0.23151 | 0.230092 | 0.29093 | 0.269514 |
| 0.7344 | 0.00332 | -0.00152995 | 0.16256 | 0.151146 |
| 0.6172 | -0.13641 | -0.138112 | 0.02135 | 0.0160491 |
| 0.5000 | -0.20581 | -0.200192 | -0.11477 | -0.118177 |
| 0.4531 | -0.21090 | -0.20266 | -0.17119 | -0.173697 |
| 0.2813 | -0.15662 | -0.14565 | -0.32726 | -0.299671 |
| 0.1719 | -0.10150 | -0.0927883 | -0.24299 | -0.205573 |
| 0.1016 | -0.06434 | -0.0572724 | -0.14612 | -0.118734 |
| 0.0703 | -0.04775 | -0.0400987 | -0.10338 | -0.0811636 |
| 0.0625 | -0.04192 | -0.0355746 | -0.09266 | -0.0716953 |
| 0.0547 | -0.03717 | -0.0309368 | -0.08186 | -0.0621251 |
| 0.0000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |




(a) $Re = 100$ (b) $Re = 400$

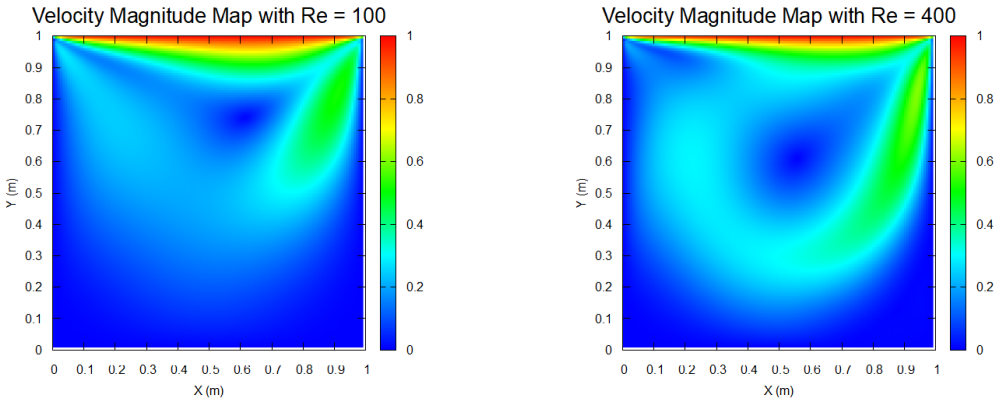
Figure 4: Velocity profiles $v(x)$ at $y = 0.5$ compared to reference data for different Reynolds numbers.

Table 2: Values of v(x) at y = L/2 for different Re numbers - Comparison between reference data and obtained results.

| $x/L$ | $Re = 100$ | | $Re = 400$ | |
|---|---|---|---|---|
| | Reference | Results | Reference | Results |
| 0.0000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| 0.0625 | 0.09233 | 0.0885496 | 0.18360 | 0.159979 |
| 0.0703 | 0.10091 | 0.0967942 | 0.19713 | 0.172172 |
| 0.0781 | 0.10890 | 0.104472 | 0.20920 | 0.183125 |
| 0.0938 | 0.12317 | 0.11821 | 0.22965 | 0.201884 |
| 0.1563 | 0.16077 | 0.154594 | 0.28124 | 0.250601 |
| 0.2266 | 0.17507 | 0.168746 | 0.30203 | 0.274158 |
| 0.2344 | 0.17527 | 0.168988 | 0.30174 | 0.27462 |
| 0.5000 | 0.05454 | 0.0534066 | 0.05186 | 0.0537795 |
| 0.8047 | -0.24533 | -0.241264 | -0.38598 | -0.372417 |
| 0.8594 | -0.22445 | -0.217811 | -0.44993 | -0.419908 |
| 0.9063 | -0.16914 | -0.159579 | -0.23827 | -0.337290 |
| 0.9453 | -0.10313 | -0.0910975 | -0.22847 | -0.183636 |
| 0.9531 | -0.08864 | -0.0761409 | -0.19254 | -0.149784 |
| 0.9609 | -0.07391 | -0.0609732 | -0.15663 | -0.116486 |
| 0.9688 | -0.05906 | -0.0456858 | -0.12146 | -0.084363 |
| 1.0000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |

Figure 3 and Table 1 specifically show the velocity profile $u(y)$ at $x = 0.5$, while Figure 4 and Table 2 present the velocity profile $v(x)$ at $y = 0.5$.
The obtained results align closely with the reference values, demonstrating the accuracy of the implemented numerical scheme. However, for $Re = 400$, the results show slightly less agreement compared to $Re = 100$, particularly for the $v(x)$ velocity profile, where larger discrepancies are observed.

To improve accuracy for higher Reynolds numbers, refining the grid is often necessary, as the mesh resolution should ideally be adapted to the Reynolds number to ensure consistent results (a concept that will be further explored in the following sections). Reducing the time step may not significantly enhance accuracy in laminar flows since the physical time scale is much larger than the stability time scale. Additionally, adopting an Upwind Differencing Scheme (UDS) may improve convergence performance, but at the cost of increased numerical diffusion, which tends to worsen with higher Reynolds numbers.
To further assess the overall flow behavior, Figure 5 presents the velocity magnitude colormaps for Reynolds numbers 100 and 400.




(a) $Re = 100$ (b) $Re = 400$

Figure 5: Colormaps of velocity magnitude for $Re = 100$ and $Re = 400$.

The good agreement between the computed velocity profiles and reference data confirms the reliability of the numerical method. The colormaps provide a clear visualization of the flow field, giving useful insight into its behavior at different Reynolds numbers.

## 4.2 Code Refinements and Implementations

While analyzing the preliminary results and reviewing the code, it became clear that some improvements could make the code more modular and flexible, while also improving the accuracy of the results. These changes focused on enhancing the code structure itself and refining the way results are processed. Additionally, an error analysis was introduced to evaluate how different mesh refinements impact the solution's performance.
The key improvements implemented are summarized as follows:

- **Dynamic Memory Allocation:** The initial code used static allocation for matrices, which limited flexibility when adjusting the grid resolution. Thanks to dynamic memory allocation, the code can now easily adapt to different mesh refinements chosen by the user, making it more flexible and efficient. However, the code still requires a uniform mesh since several formula simplifications rely on this assumption. Modifying this aspect would have required changes to all the related equations — an improvement that could be implemented in the future to make the code even more versatile.

- **User-Defined Mesh Refinement and Reynolds Number:** To enhance flexibility, the new version allows the user to specify the desired mesh refinement and Reynolds number directly from the terminal. This was made possible by implementing dynamic memory allocation for the matrices, which simplifies testing various conditions and improves the adaptability of the code.

- **Improved Data Extraction with Interpolation:** Previously, the data reported in the *Initial Results* subsection (Tables 1 and 2) had been manually selected by choosing the closest computed value to the reference positions indicated by Ghia et al. This manual approach was replaced by an interpolation method, implemented to extract data points at positions where no direct computation was available. This was achieved by creating a dedicated function for interpolation and integrating it into a comprehensive function for data extraction and error calculation.
  The interpolation function `Interpolate()` follows a linear interpolation approach, defined by the following formula:

$$f(x) = y_1 + (x - x_1)\frac{y_2 - y_1}{x_2 - x_1} \tag{39}$$

  Where:

  - $x_1$ and $x_2$ are the two known data points between which the interpolation is performed.
  - $y_1$ and $y_2$ are the corresponding function values at points $x_1$ and $x_2$.
  - $x$ is the desired position where the interpolated value is calculated.
  - $f(x)$ is the resulting interpolated value.

  This function was designed to efficiently identify the interval containing the desired position and compute the interpolated value accordingly.
  The interpolation function is employed by a second and more general function `ResultsManipulation()`, which is responsible for:

  - Reading reference positions from a predefined reference data matrix which contains the values taken from Ghia et al.
  - Loading computed values from previously generated text files used for graphical visualization.
  - Performing interpolation at the required positions using the interpolation function.

– Writing the interpolated values into a new output file, ensuring the data is presented in an organized format.

In addition to interpolated values, this function also computes the absolute error between the reference and interpolated values. The calculation of this error is discussed in the following bullet point.

- **Absolute Error Calculation:** To evaluate the accuracy of the numerical results across different mesh refinements, the calculation of the absolute error was added.
  For each data point, the absolute error was calculated as:

$$\text{Absolute Error} = |y_{\text{ref}} - y_{\text{comp}}| \tag{40}$$

  Where:

  – $y_{\text{ref}}$ is the reference value.

  – $y_{\text{comp}}$ is the computed interpolated value.

  To assess the overall accuracy across all data points, the **Mean Absolute Error (MAE)** was introduced, defined as:

$$\text{MAE} = \frac{\sum_{i=1}^{N} |y_{\text{ref},i} - y_{\text{comp},i}|}{N} \tag{41}$$

  Where $N$ is the total number of calculated points.

  In the final output file, the absolute error for each data point was printed alongside the corresponding reference and interpolated values. At the end of the file, the Mean Absolute Error was reported, providing a clear indicator of the global accuracy achieved with each mesh refinement.

The combined effect of these refinements significantly improved the code's performance, flexibility, and result accuracy. The following section presents the refined results and discusses the observed improvements compared to the preliminary outcomes.

## 4.3 Final Results

In this section, the final results are presented and discussed. The improved code was used to compute solutions for two Reynolds numbers, 100 and 400, using three different mesh refinements for each case. For this analysis, the horizontal velocity component $u(y)$ at $x = L/2$ was considered. To

evaluate the accuracy of the results, the Mean Absolute Error (MAE) was calculated for each configuration. This comparison highlights the impact of grid refinement on solution precision and computational cost.

Table 3: Comparison of results for Reynolds 100 with different mesh refinements.

| **Position** | **Reference** | **30 CV** | **60 CV** | **128 CV** |
|---|---|---|---|---|
| 1.0000 | 1.00000 | 1.00000 | 1.00000 | 1.00000 |
| 0.9766 | 0.84123 | 0.95476 | 0.89862 | 0.86914 |
| 0.9688 | 0.78871 | 0.90235 | 0.84647 | 0.81691 |
| 0.9609 | 0.73722 | 0.84927 | 0.79373 | 0.76479 |
| 0.9531 | 0.68717 | 0.79686 | 0.74326 | 0.71454 |
| 0.8516 | 0.23151 | 0.28524 | 0.25775 | 0.24413 |
| 0.7344 | 0.00332 | 0.02093 | 0.00951 | 0.00453 |
| 0.6172 | -0.13641 | -0.11786 | -0.12956 | -0.13578 |
| 0.5000 | -0.20581 | -0.17566 | -0.19205 | -0.20085 |
| 0.4531 | -0.21090 | -0.17864 | -0.19496 | -0.20411 |
| 0.2813 | -0.15662 | -0.12976 | -0.14100 | -0.14781 |
| 0.1719 | -0.10150 | -0.08295 | -0.09022 | -0.09473 |
| 0.1016 | -0.06434 | -0.04985 | -0.05578 | -0.05933 |
| 0.0703 | -0.04775 | -0.03321 | -0.03890 | -0.04229 |
| 0.0625 | -0.04192 | -0.02880 | -0.03456 | -0.03782 |
| 0.0547 | -0.03717 | -0.02439 | -0.03002 | -0.03325 |
| 0.0000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| **Mean Absolute Error** | | 0.04127 | 0.02091 | 0.01008 |

Table 4: Comparison of results for Reynolds 400 with different mesh refinements.

| **Position** | **Reference** | **30 CV** | **60 CV** | **128 CV** |
|---|---|---|---|---|
| 1.0000 | 1.00000 | 1.00000 | 1.00000 | 1.00000 |
| 0.9766 | 0.75837 | 0.92894 | 0.83597 | 0.79370 |
| 0.9688 | 0.68439 | 0.84661 | 0.75659 | 0.71575 |
| 0.9609 | 0.61756 | 0.76323 | 0.67764 | 0.64311 |
| 0.9531 | 0.55892 | 0.68090 | 0.60994 | 0.57902 |
| 0.8516 | 0.29093 | 0.23228 | 0.25749 | 0.27414 |
| 0.7344 | 0.16256 | 0.13490 | 0.14840 | 0.15545 |
| 0.6172 | 0.02135 | 0.02788 | 0.02228 | 0.02015 |
| 0.5000 | -0.11477 | -0.08888 | -0.10676 | -0.11397 |
| 0.4531 | -0.17119 | -0.13641 | -0.15987 | -0.16947 |
| 0.2813 | -0.32726 | -0.22878 | -0.27325 | -0.30136 |
| 0.1719 | -0.24299 | -0.16453 | -0.18910 | -0.21057 |
| 0.1016 | -0.14612 | -0.10061 | -0.11153 | -0.12336 |
| 0.0703 | -0.10338 | -0.06778 | -0.07681 | -0.08572 |
| 0.0625 | -0.09266 | -0.05897 | -0.06796 | -0.07632 |
| 0.0547 | -0.08186 | -0.05017 | -0.05894 | -0.06683 |
| 0.0000 | 0.00000 | 0.00000 | 0.00000 | 0.00000 |
| **Mean Absolute Error** | | 0.06338 | 0.03209 | 0.01589 |

The results shown in Tables 3 and 4 clearly demonstrate that refining the mesh improves the accuracy of the solution for both Reynolds numbers. This improvement is particularly evident for Reynolds 400, where the flow characteristics are more complex and sensitive to grid resolution. For instance, when comparing the same mesh refinement level (e.g., 30 control volumes), the Mean Absolute Error (MAE) for Reynolds 400 is 0.063, whereas for Reynolds 100 it is 0.041. This highlights the increased difficulty in accurately capturing flow details at higher Reynolds numbers and emphasizes the importance of adopting finer meshes in such cases.

It is important to note that the observed trend aligns with expectations — higher Reynolds numbers generally require finer grid resolutions to ensure accuracy. However, in this specific case, the finest mesh used in this study (128x128) corresponds to the reference mesh employed by Ghia et al. to solve the entire Reynolds number range, including values up to 10,000. Consequently, the key takeaway from this analysis is not just the importance of refining the grid for higher Reynolds numbers, but rather the potential to coarsen the grid for lower Reynolds numbers while still achieving satisfactory accuracy. For instance, in the Re = 100 case, the results indicate

that a coarse mesh of approximately 30x30 control volumes already provides reliable results. This finding underscores the opportunity to improve computational efficiency when simulating lower Reynolds number flows without significantly compromising solution accuracy.

# 5 Conclusions

The numerical solution of the Lid-Driven Cavity problem was successfully obtained using the Fractional Step Method. The computed velocity profiles showed good agreement with the reference data provided by Ghia et al., demonstrating the accuracy and reliability of the implemented numerical scheme.

During the initial phase of the study, the CFL condition proved insufficient to ensure convergence. However, reducing the time step allowed stable and accurate results to be achieved. The obtained preliminary results closely align with the reference values, confirming the accuracy of the implemented numerical scheme. However, for $Re = 400$, the results show slightly less agreement compared to $Re = 100$, particularly for the $v(x)$ velocity profile, where larger discrepancies are observed.

To improve accuracy for higher Reynolds numbers, a refined version of the code was implemented, introducing several improvements that enhanced both flexibility and accuracy. Dynamic memory allocation was incorporated, allowing the mesh refinement and Reynolds number to be selected directly from the terminal. Moreover, interpolation of results was introduced to improve accuracy, and an error analysis was performed to evaluate the performance of different mesh refinements.

The new results confirmed that refining the mesh improves solution accuracy, particularly for higher Reynolds numbers, where flow characteristics become more complex and sensitive to grid resolution. This effect is evident when comparing the Mean Absolute Error (MAE) values. However, this analysis also revealed that for lower Reynolds numbers, satisfactory accuracy can still be achieved with coarser meshes. This finding underscores the potential to improve computational efficiency when simulating lower Reynolds number flows without significantly compromising solution accuracy.

Finally, the presented colormaps effectively visualized the flow field, providing further insight into the fluid behavior at different Reynolds numbers. Overall, the improved code proved robust and efficient, capable of delivering accurate results while offering flexibility for future modifications and optimizations.

# A C++ Code

## Refined Version

```
#include "Lid_Driven_Cavity.h"
#include<iostream>
#include <fstream>
#include <chrono>
#include <cmath>
#include <cstdlib>
#include <iomanip>

using namespace std;
using namespace std::chrono;

// Function to allocate dynamic matrix
double** AllocateMatrix(int rows, int cols) {
    auto matrix = new double*[rows];
    for (int i = 0; i < rows; i++) {
        matrix[i] = new double[cols]();
    }
    return matrix;
}

// Function to deallocate dynamic matrix
void DeallocateMatrix(double**& matrix, int rows) {
    for (int i = 0; i < rows; i++) {
        delete[] matrix[i];
    }
    delete[] matrix;
    matrix = nullptr;
}

// Function to compute boundary conditions for velocity
void BoundaryConditions(double Uref, int Nx, int Ny, double **u1, double **un, double **
    un_1, double **v1, double **vn,
                        double **vn_1) {
    for (int i = 0; i < Ny; i++) {
        for (int j = 0; j < Nx+1; j++) {
            if (i == 0)
                u1[i][j] = Uref,
                un[i][j] = Uref,
                un_1[i][j] = Uref;
            if (i == Ny - 1 || j == 0 || j == Nx)
                u1[i][j] = 0,
                un[i][j] = 0,
                un_1[i][j] = 0;
        }
    }
    for (int i = 0; i < Ny + 1; i++) {
        for (int j = 0; j < Nx; j++) {
            if (i == 0)
                v1[i][j] = 0,
                vn[i][j] = 0,
                vn_1[i][j] = 0;
            if (i == Ny || j == 0 || j == Nx - 1)
                v1[i][j] = 0,
                vn[i][j] = 0,
                vn_1[i][j] = 0;
        }
    }
}

// Function to compute R(u) on internal nodes of stagg-x mesh
void ComputeRu(double** u, double** v, double** Ru, double rho, double dx, double mu,
    int Nx, int Ny) {
    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 (double** u, double** v, double** Rv, double rho, double dx, double mu,
     int Nx, int Ny) {
    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]);
        }
    }
}

// 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 **P1,
                   double **vP, double **uP, 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;
        for (int i = 0; i < Ny; i++) {
            for (int j = 0; j < Nx; j++) {
                Pg[i][j] = P1[i][j];
            }
        }
        iteP++;
    }
}

// Function to find the interpolated value
double Interpolate(double x, double** data, int dataSize) {
    if (x == data[dataSize - 1][0]) {
        return data[dataSize - 1][1]; // Explicitly handle the last value
    }
    for (int i = 0; i < dataSize - 1; i++) {
        if ((x >= data[i][0] && x <= data[i + 1][0]) ||
            (x <= data[i][0] && x >= data[i + 1][0])) {

            double x1 = data[i][0], x2 = data[i + 1][0];
            double y1 = data[i][1], y2 = data[i + 1][1];
            return y1 + (x - x1) * (y2 - y1) / (x2 - x1);
        }
    }
    return NAN; // If the value is out of range
}

// Function to extract, interpolate the requested values and calculate the mean absolute
     error
void ResultsManipulation(const string &sourceFile, const string &targetFile, double ref
    [][3], int refSize,
                         double **data, int N, int index) {
    ifstream inputFile(sourceFile);
    ofstream outputFile(targetFile);
```

```
    // Read data from file
    int dataSize = 0;
    double pos, value;
    while (inputFile >> pos >> value && dataSize < N+2) {
        data[dataSize][0] = pos;
        data[dataSize][1] = value;
        dataSize++;
    }

    // Formats output file
    outputFile << left << setw(15) << "Position"
               << setw(20) << "Reference"
               << setw(20) << "Interpolated Result"
               << setw(20) << "Absolute Error"
               << endl;
    outputFile << string(75, '-') << endl;

    // Extraction, interpolation and error calculation
    double sumError = 0.0;
    for (int i = 0; i < refSize; i++) {
        double position = ref[i][0];
        double referenceValue = ref[i][index];
        double interpolatedValue = Interpolate(position, data, dataSize);
        double MAError = fabs(referenceValue - interpolatedValue);

        if (!isnan(interpolatedValue)) {
            outputFile << left << setw(15) << position
                       << setw(20) << referenceValue
                       << setw(20) << interpolatedValue
                       << setw(20) << MAError << endl;
        } else {
            outputFile << left << setw(15) << position
                       << setw(20) << "NAN"
                       << setw(20) << "(out of range)"
                       << endl;
        }
        sumError += MAError;
    }
    outputFile << string(75, '-') << endl;
    outputFile << "Mean Absolute Error =" << sumError / refSize << endl;
    cout << "File " << targetFile << " successfully created." << endl;
    inputFile.close();
    outputFile.close();
}

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

    // Physical data
    int N;
    // Ask mesh refinement
    cout << "Insert number of control volumes:";
    cin >> N;
    int Nx = N;
    int Ny = N;
    double Re;
    double L = 1.0;
    double dx = L / Nx;
    double rho = 1.0;
    double Uref = 1.0;
    // Ask for desired Reynolds number
    cout << "Insert Reynolds number:";
    cin >> Re;
    double mu = rho * Uref * L / Re;

    // Numerical data
    // double Cconv = 0.35;
    // double Cdiff = 0.2;
    // CFL condition
    //double dt = min(Cconv * dx / Uref, Cdiff * pow(dx, 2) * rho / mu);     // Works
        with Re = 400 but not with Re = 100
    double dt = 0.001*0.5;                                                   // Works
        with both Re = 100 and Re = 400
    double maxRes = 1e-6;
    double maxIte = 1e6;
    double t_count = 0.0;
    double res1 = maxRes + 1;
    double res2 = maxRes + 1;
    int ite = 0;
    double** data = AllocateMatrix(N+2, 2);
```

```
// Pressure, u and v matrices
double** P1 = AllocateMatrix(Ny, Nx);
double** Pg = AllocateMatrix(Ny, Nx);
double** un_1 = AllocateMatrix(Ny, Nx+1);
double** un = AllocateMatrix(Ny, Nx+1);
double** u1 = AllocateMatrix(Ny, Nx+1);
double** uP = AllocateMatrix(Ny, Nx+1);
double** vn_1 = AllocateMatrix(Ny+1, Nx);
double** vn = AllocateMatrix(Ny+1, Nx);
double** v1 = AllocateMatrix(Ny+1, Nx);
double** vP = AllocateMatrix(Ny+1, Nx);

// Convective-diffusive term R() matrices
double** Ru1 = AllocateMatrix(Ny, Nx+1);
double** Run = AllocateMatrix(Ny, Nx+1);
double** Run_1 = AllocateMatrix(Ny, Nx+1);
double** Rv1 = AllocateMatrix(Ny+1, Nx);
double** Rvn = AllocateMatrix(Ny+1, Nx);
double** Rvn_1 = AllocateMatrix(Ny+1, Nx);

// Initial velocity fields = 0 for internal nodes
for (int i = 1; i < Ny - 1; i++) {
    for (int j = 1; j < Nx; j++) {
        un[i][j] = 0;
        un_1[i][j] = un[i][j];
    }
}
for (int i = 1; i < Ny; i++) {
    for (int j = 1; j < Nx - 1; j++) {
        vn[i][j] = 0;
        vn_1[i][j] = vn[i][j];
    }
}

// Initial pressure field
for (int i = 0; i < Ny; i++) {
    for (int j = 0; j < Nx; j++) {
        Pg[i][j] = 0;
    }
}

// Apply constant boundary conditions of velocity
BoundaryConditions(Uref, Nx, Ny, u1, un, un_1, v1, vn, vn_1);

// Compute R()^n-1
ComputeRu(un_1, vn_1, Run_1, rho, dx, mu, Nx, Ny);
ComputeRv(un_1, vn_1, Rvn_1, rho, dx, mu, Nx, Ny);

// Compute R()^n
ComputeRu(un, vn, Run, rho, dx, mu, Nx, Ny);
ComputeRv(un, vn, Rvn, rho, dx, mu, Nx, Ny);

// Time loop until convergence (aka steady state) is reached
while (res1 > maxRes && res2 > maxRes && ite < maxIte) {
    double maxDiff1 = 0.0;
    double maxDiff2 = 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
    for (int i = 1; i < Ny; i++) {
        for (int j = 1; j < Nx - 1; j++) {
            vP[i][j] = vn[i][j] + dt / (rho * pow(dx, 2)) * (3.0 / 2 * Rvn[i][j] -
                1.0 / 2 * Rvn_1[i][j]);
        }
    }

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

    // Step 3
    for (int i = 1; i < Ny - 1; i++) {
        for (int j = 1; j < Nx; j++) {
            u1[i][j] = uP[i][j] - dt / rho * (P1[i][j] - P1[i][j - 1]) / dx;
            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 * (P1[i - 1][j] - P1[i][j]) / dx;
            double diff2 = fabs(v1[i][j] - vn[i][j]);
            if (diff2 > maxDiff2) {
                maxDiff2 = diff2;
            }
        }
    }
    t_count += dt;
    res1 = maxDiff1;
    res2 = maxDiff2;

    // 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];
        }
    }

    // Compute R()^n+1
    ComputeRu(u1, v1, Ru1, rho, dx, mu, Nx, Ny);
    ComputeRv(u1, v1, Rv1, rho, dx, mu, Nx, Ny);

    // 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];
        }
    }

    // Go to next time step, if needed
    ite++;
}

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

// File to check the meshes
ofstream TestFile("Test.txt");
for (int i = 0; i < Ny; i++) {
    for (int j = 0; j < Nx; j++) {
        TestFile << u1[i][j] << "-";
    }
    TestFile << endl;
}
TestFile.close();

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

// File to print data for Gnuplot 1
ofstream GnuplotData1("GnuplotData1.txt");
double ref1[17][3] = {
    {1.0000, 1.00000, 1.00000},
    {0.9766, 0.84123, 0.75837},
    {0.9688, 0.78871, 0.68439},
    {0.9609, 0.73722, 0.61756},
    {0.9531, 0.68717, 0.55892},
```

```
    {0.8516, 0.23151, 0.29093},
    {0.7344, 0.00332, 0.16256},
    {0.6172, -0.13641, 0.02135},
    {0.5000, -0.20581, -0.11477},
    {0.4531, -0.21090, -0.17119},
    {0.2813, -0.15662, -0.32726},
    {0.1719, -0.10150, -0.24299},
    {0.1016, -0.06434, -0.14612},
    {0.0703, -0.04775, -0.10338},
    {0.0625, -0.04192, -0.09266},
    {0.0547, -0.03717, -0.08186},
    {0.0000, 0.00000, 0.00000}
};

int index1 = 0;
if (Re == 100)
    index1 = 1;
else if (Re == 400)
    index1 = 2;

for (auto & i : ref1) {
    GnuplotData1 << i[0] << "-" << i[index1] << std::endl;
}
GnuplotData1.close();

// Generate Gnuplot script 1
ofstream Gnuplot1("ComparisonUy.plt");

Gnuplot1 << "set terminal pngcairo size 800,600 enhanced font 'Arial,14'\n";
Gnuplot1 << "set output 'Comparison_Uy.png'\n";
Gnuplot1 << "set xlabel 'u'\n";
Gnuplot1 << "set ylabel 'y'\n";
Gnuplot1 << "set title 'Velocity Profile u(y) at L/2 with Re=" << Re << "'\n";
Gnuplot1 << "set key top left\n";
Gnuplot1 << "set grid\n";
Gnuplot1 << "set xrange [-0.4:1]\n";
Gnuplot1 << "set yrange [0:1]\n";
Gnuplot1 << "plot 'Uy.txt' using 2:1 with lines lw 2 lc rgb 'blue' title 'Result',"
        << "'GnuplotData1.txt' using 2:1 with points pt 7 ps 1.2 lc rgb 'red' title
            'Reference'\n";
Gnuplot1.close();

// Automatically run Gnuplot 1
system("gnuplot ComparisonUy.plt");
cout << "Gnuplot script 1 executed: 'Comparison_Uy.png' generated." << endl;

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

// File to print data for Gnuplot 2
ofstream GnuplotData2("GnuplotData2.txt");
double ref2[17][3] = {
    {0.0000, 0.00000, 0.00000},
    {0.0625, 0.09233, 0.18360},
    {0.0703, 0.10091, 0.19713},
    {0.0781, 0.10890, 0.20920},
    {0.0938, 0.12317, 0.22965},
    {0.1563, 0.16077, 0.28124},
    {0.2266, 0.17507, 0.30203},
    {0.2344, 0.17527, 0.30174},
    {0.5000, 0.05454, 0.05186},
    {0.8047, -0.24533, -0.38598},
    {0.8594, -0.22445, -0.44993},
    {0.9063, -0.16914, -0.23827},
    {0.9453, -0.10313, -0.22847},
    {0.9531, -0.08864, -0.19254},
    {0.9609, -0.07391, -0.15663},
    {0.9688, -0.05906, -0.12146},
    {1.0000, 0.00000, 0.00000}
};

int index2 = 0;
if (Re == 100)
    index2 = 1;
else if (Re == 400)
    index2 = 2;
```

```
for (auto & i : ref2) {
    GnuplotData2 << i[0] << "-" << i[index2] << std::endl;
}
GnuplotData2.close();

// Generate Gnuplot script 2
ofstream Gnuplot2("ComparisonVx.plt");

Gnuplot2 << "set terminal pngcairo size 800,600 enhanced font 'Arial,14'\n";
Gnuplot2 << "set output 'Comparison_Vx.png'\n";
Gnuplot2 << "set xlabel 'x'\n";
Gnuplot2 << "set ylabel 'v(x)'\n";
Gnuplot2 << "set title 'Velocity Profile v(x) at L/2 with Re=" << Re << "'\n";
Gnuplot2 << "set key top left\n";
Gnuplot2 << "set grid\n";
Gnuplot2 << "set xrange [0:1]\n";
Gnuplot2 << "set yrange [-0.5:0.4]\n";
Gnuplot2 << "plot 'Vx.txt' using 1:2 with lines lw 2 lc rgb 'blue' title 'Result',"
        << "'GnuplotData2.txt' using 1:2 with points pt 7 ps 1.2 lc rgb 'red' title
            'Reference'\n";
Gnuplot2.close();

// Automatically run Gnuplot 2
system("gnuplot ComparisonVx.plt");
cout << "Gnuplot script 2 executed: 'Comparison_Vx.png' generated." << endl;

// File to print velocity distribution for Plot 3
ofstream VelocityDistribution("VelocityDistribution.txt");
for (int i = 0; i < Ny; i++) {
    for (int j = 0; j < Nx; j++) {
        VelocityDistribution << j * dx + dx / 2 << "-" << L - i * dx - dx / 2 << "-"
             << sqrt(
            pow(v1[i][j], 2) + pow(u1[i][j], 2)) << endl;
    }
    VelocityDistribution << "\n";
}
VelocityDistribution.close();

// Generate Gnuplot script 3
ofstream Gnuplot3("MagnitudeMap.plt");

Gnuplot3 << "set terminal pngcairo size 600,600 enhanced font 'Arial,12'\n";
Gnuplot3 << "set output 'MagnitudeMap_Plot.png'\n";
Gnuplot3 << "set pm3d map\n";
Gnuplot3 << "set palette defined ("
         << "0 'blue',"
         << "0.3 'cyan',"
         << "0.5 'green',"
         << "0.7 'yellow',"
         << "1 'red')\n";
Gnuplot3 << "set colorbox vertical\n";
Gnuplot3 << "set xlabel 'X (m)'\n";
Gnuplot3 << "set ylabel 'Y (m)'\n";
Gnuplot3 << "set title 'Velocity Magnitude Map with Re=" << Re << "' font 'Arial
    ,20'\n";
Gnuplot3 << "set xrange [0:1]\n";
Gnuplot3 << "set yrange [0:1]\n";
Gnuplot3 << "set autoscale\n";
Gnuplot3 << "set cbrange [0:1]\n";
Gnuplot3 << "set size square\n";
Gnuplot3 << "splot 'VelocityDistribution.txt' using 1:2:3 with pm3d notitle\n";

Gnuplot3.close();

// Automatically run Gnuplot 3
system("gnuplot MagnitudeMap.plt");
cout << "Gnuplot script 3 executed: 'MagnitudeMap_Plot.png' generated." << endl;

// Extract and interpolate data for error analysis
ResultsManipulation("Uy.txt", "FinalResults_Uy.txt", ref1, 17, data, N, index1);
ResultsManipulation("Vx.txt", "FinalResults_Vx.txt", ref2, 17, data, N, index2);

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

// Deallocate matrix
DeallocateMatrix(P1, Ny);
DeallocateMatrix(Pg, Ny);
DeallocateMatrix(un_1, Ny);
DeallocateMatrix(un, Ny);
```

```
    DeallocateMatrix(u1, Ny);
    DeallocateMatrix(uP, Ny);
    DeallocateMatrix(vn_1, Ny+1);
    DeallocateMatrix(vn, Ny+1);
    DeallocateMatrix(v1, Ny+1);
    DeallocateMatrix(vP, Ny+1);
    DeallocateMatrix(Ru1, Ny);
    DeallocateMatrix(Run, Ny);
    DeallocateMatrix(Run_1, Ny);
    DeallocateMatrix(Rv1, Ny+1);
    DeallocateMatrix(Rvn, Ny+1);
    DeallocateMatrix(Rvn_1, Ny+1);
    DeallocateMatrix(data, N+2);

    return 0;
}
```