ME663 - Computational Fluid Dynamics
Assignment 1

Tommaso Bocchietti

A.Y. 2023/24 - W24

Contents

List of Figures

List of Tables

Listings

1 Requests

This assignment was about writing a computer program using the language we prefer, to solve the incompressible Navier-Stokes equations based on Vanka's Symmetric Coupled Gauss-Seidel (SCGS) method [2].
The objective here is to find out the numerical solutions for a flow in a square cavity with the top wall moving at a constant velocity U_{lid}. Different suggestions / requests were given, such as:

- Use UDS, Hybrid, and/or QUICK convection schemes on a mesh of $40x40$ and $80x80$ nodes at $Re = 400$ and $Re = 1000$, where $Re = \frac{U_{lid} \cdot L}{\nu}$ and L is the length of the cavity.
- Set the convergence criterion to 10^{-4}.
- Report the wall-clock time and total number of iterations for each case.
- Compare your results in terms of $u(y)@x = 0.5L$ and $v(x)@y = 0.5L$ with results in Tables I and II from *Ghia et al.* [1] paper.
- What is the optimal under-relaxation factor for each case?
- If you try more than one convection scheme, which scheme is more accurate compared to *Ghia et al.* [1] 'exact solutions'?
- Use $2^{nd} - order$ and $4^{th} - order$ schemes to approximate the diffusion term $\nu \left[\frac{\partial^2(\phi)}{\partial x^2} + \frac{\partial^2(\phi)}{\partial y^2} \right]$ in the Navier-Stokes equation.

2 Methods

For this assignment, we have chosen to implement the code using C-language, and to implement different schemes and methods to solve the incompressible Navier-Stokes equations. In particular, we have implemented the following schemes for the convection term:

- Upwind Differencing Scheme (UDS)
- Central Differencing Scheme (CDS)
- Quadratic Upstream Interpolation for Convective Kinematics (QUICK)

We have also implemented the following schemes for the diffusion term:

- Second-order central difference scheme
- Fourth-order central difference scheme

For the methods to solve the linear system of equations, we have implemented the following method:

- Symmetric Coupled Gauss-Seidel (SCGS)

The main idea behind the implementation of these schemes and methods was to compare the various combinations of those, and to find out which one would be the most accurate and efficient for the given problem.
In the following sections, we will describe the implementation of each of these schemes and methods, and we will also present the results obtained from the simulations.
Complete code can be found at `https://github.com/Bocchio01/CFD_Simulation_Engine`, along with its documentation and instructions on how to run it.

3 Derivation of discretized governing equations

In this section, we will derive the discretized governing equations for the incompressible Navier-Stokes equations, which will be used to solve the problem at hand.
The set of incompressible Navier-Stokes equations is given by:

$$\nabla \cdot \mathbf{u} = 0$$

$$\rho \frac{\partial \mathbf{u}}{\partial t} + \rho \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f}$$

In the rest of the document, the following hypotheses will be considered:

- Steady-state problem: $\frac{\partial \mathbf{u}}{\partial t} = 0$
- Constant density: $\rho = \text{const}$
- Constant dynamic viscosity: $\mu = \text{const}$
- Zero body forces: $\mathbf{f} = 0$

Based on these hypotheses, the incompressible Navier-Stokes equations can be simplified and expanded as follows:

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

$$\frac{\partial uu}{\partial x} + \frac{\partial vu}{\partial y} = -\frac{\partial p}{\partial x} + \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{2}$$

$$\frac{\partial uv}{\partial x} + \frac{\partial vv}{\partial y} = -\frac{\partial p}{\partial y} + \nu \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) \tag{3}$$

Where $\nu = \frac{\mu}{\rho}$ is the kinematic viscosity and $p = \frac{p}{\rho}$ is the non-dimensional pressure.
Obviously, to solve the problem using a discrete calculator, the equations must be therefore discretized.

3.1 Finite Volume Method

The Finite Volume Method (FVM) is a numerical technique used to discretize partial differential equations, and is particularly well suited for the discretization of the Navier-Stokes equations.
The idea here is to divide the domain into a set of control volumes, and then integrate the governing equations over each control volume. The resulting set of equations will be a set of algebraic equations, which can be solved using a discrete calculator.

3.1.1 Control volumes

Before proceeding with the discretization of the governing equations, we need to define what a control volume is and the notations used in the rest of the document.
In particular, we will assume from now on to have a Cartesian grid, with a uniform mesh spacing in both the x and y directions.
From the Figure 1, we can appreciate graphically how the domain is divided.

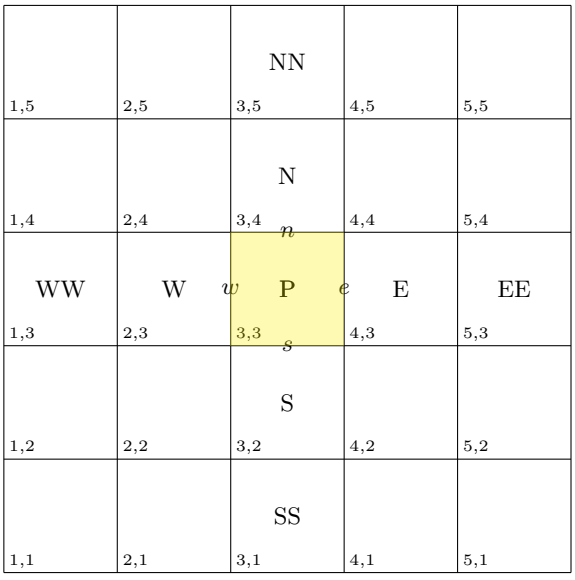

Figure 1: Control volumes and control volume faces.

In particular, Figure 1 shows:

- A grid of control volumes, with the subscript (i, j) indicating the position of the control volume in the x and y directions, respectively. For example, the control volume in the center of the grid has indices $(i, j) = (3, 3)$.

- The control volume centers with capital letters, $P, N, S, E, W, \ldots$.
- The control volume faces with lowercase letters, n, s, e, w.

Notice also that the capital letters always refers to a relative position with respect to the control volume in consideration. For example, P refers to the control volume in consideration, N refers to the control volume to the north of P, and so on. For this reason, in Figure 1, the control volume P is highlighted in yellow so to indicate that is the control volume in consideration.

3.1.2 Staggered grid and L_{shape}

As we will see later during the formulation of the solving solution, for the purpose of this work, we will use a specific type of grid, called staggered grid.
We can also give a brief definition of the two types of grids available in the literature, which are:

- **Collocated grid**: all the variables are located at the same point in the control volume (e.g. the center of the control volume).
- **Staggered grid**: the variables are located at different points in the control volume (e.g. the velocity components are located at the center of the faces of the control volume, and the pressure is located at the center of the control volume).

Given the formulation of the staggered grid, it's now useful to define the so called L_{shape}, which is a frame used to define in a compact and clear way the position of the variables in the control volume.
The L_{shape} has been reported for control volume P in Figure 2.

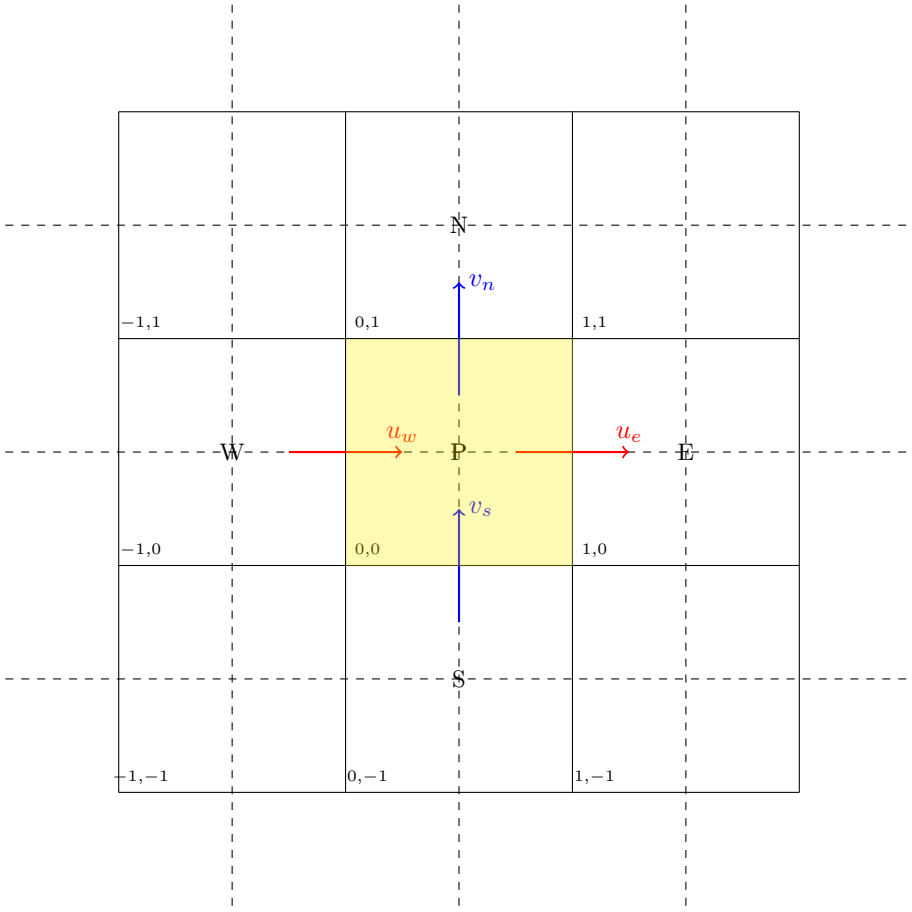

Figure 2: L_{shape} for control volume P.

Basically, the L_{shape} for control volume P links the velocity components to the control volume P itself, and it's used to define the indexes of the system.
In particular, the same Figure 2 can be represented using the index notations, as shown in Figure 3.

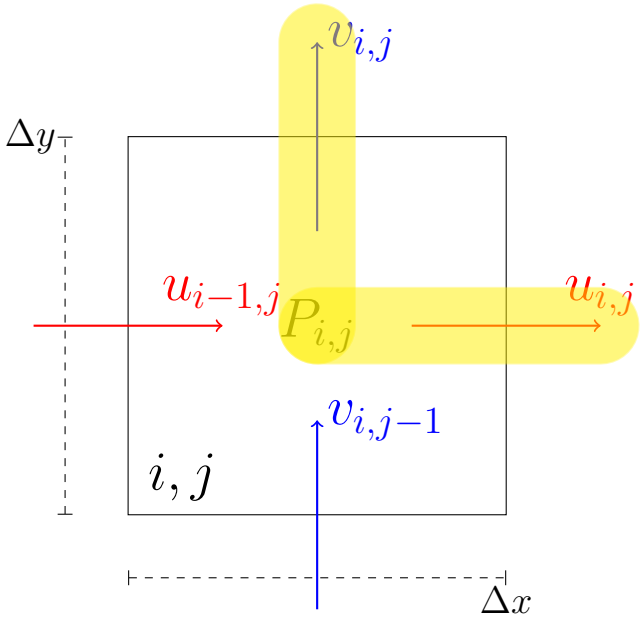

Figure 3: L_{shape} for control volume P using index notations.

From Figure 3, we can appreciate how the L_{shape} works well with index notations, and can be useful when working purely with indexes to refer to the variables.
Finally, from Figures 4 5 & 6, we can appreciate how the staggered grid is used to define different control volumes for different variables.
During the rest of the document, we will avoid the specification of the control volume taken in consideration, because it will be chosen according to the variable in consideration. If we are working with the pressure, we will refer to the control volume as P_{CV}, if we are working with the velocity components, we will refer to the control volume as U_{CV} or V_{CV}, and so on.
So for example, when dealing with the u velocity component, we will refer to the control volume as U_{CV}, so that:

- u_P will refer to the velocity component centered in the control volume U_{CV}. Equivalent to $u_{i,j}$ in the index notation.
- u_e instead, will be the east component with respect to the control volume U_{CV}.
- u_w will be the west component with respect to the control volume U_{CV}. Spatially, u_w will be centered in the P_{CV}.

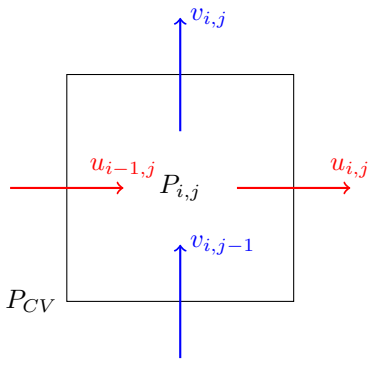

Figure 4: $P_{ControlVolume}$ (same as in Figure 3)

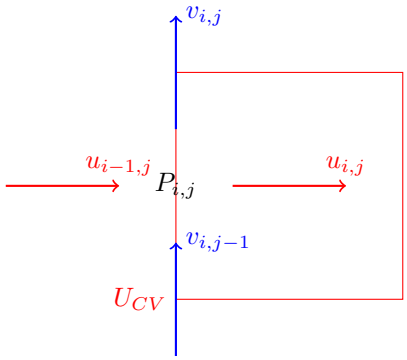

Figure 5: $U_{ControlVolume}$

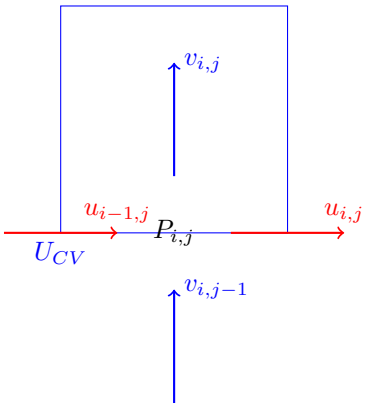

Figure 6: $V_{ControlVolume}$

3.2 Application of the Finite Volume Method

Having defined our working framework, we can now proceed with the application of the Finite Volume Method (FVM) to the incompressible Navier-Stokes equations. We know that the definition of the FVM is based on the integration of the given PDE equation over the control volume, such as:

$$\text{FVM} := \int_V \text{PDE} dV = \int_s^n \int_w^e \text{PDE} dx dy \tag{4}$$

Where s, n, w and e are the south, north, west and east faces of the control volume, respectively.
For our case, we have a set of three equations, the continuity equation 1, and the momentum equations 2 and 3.
We can now proceed by applying the FVM to each of these equations separately.
Notice that in the followings, we will assume:

$$u = u(x) \tag{5}$$
$$v = v(y) \tag{6}$$

Which underline the independence of the velocity components from the y and x coordinates, respectively.

3.2.1 Continuity Equation (FVM)

The FVM for the continuity equation 1 is given by:

$$\int_V \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} dV = 0 \tag{7}$$

Evaluating the integral based on the assumption 6, we obtain:

$$\int_s^n \int_w^e \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} dx dy = 0 \tag{8}$$
$$\int_s^n \left[\int_w^e \frac{\partial u}{\partial x} dx \right] dy + \int_w^e \left[\int_s^n \frac{\partial v}{\partial y} dy \right] dx = 0 \tag{9}$$
$$\int_s^n \left[u_e - u_w \right] dy + \int_w^e \left[v_n - v_s \right] dx = 0 \tag{10}$$
$$(u_e - u_w)\Delta y + (v_n - v_s)\Delta x = 0 \tag{11}$$

Where Δx and Δy are the dimensions of the control volume in the x and y directions, respectively.
Since we will be dealing with indexes for the implementation of the code, we can rewrite the equation as:

$$(u_i - u_{i-1})\Delta y + (v_j - v_{j-1})\Delta x = 0 \tag{12}$$

3.2.2 Momentum Equations (FVM)

The FVM for the momentum equations 2 and 3 is given by:

$$\int_V \frac{\partial uu}{\partial x} + \frac{\partial vu}{\partial y} + \frac{\partial p}{\partial x} - \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) dV = 0 \tag{13}$$
$$\int_V \frac{\partial uv}{\partial x} + \frac{\partial vv}{\partial y} + \frac{\partial p}{\partial y} - \nu \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) dV = 0 \tag{14}$$

Since the equations are very similar, we will only derive the FVM for the first momentum equation (along the x direction), and then we will present the final form for both equations.
Evaluating the integral based on the assumption 6, we obtain:

$$\int_V \underbrace{\frac{\partial uu}{\partial x} + \frac{\partial vu}{\partial y}}_{\text{Convection terms}} + \underbrace{\frac{\partial p}{\partial x}}_{\text{Source term}} \underbrace{- \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)}_{\text{Diffusion term}} dV = 0 \tag{15}$$

We can now proceed by discretizing the convection, diffusion and source terms separately.

Convection Term The finite volume discretization of the convection term is given by:

$$\int_V \frac{\partial uu}{\partial x} + \frac{\partial vu}{\partial y} dV = 0 \tag{16}$$

We further split the convection term into its directional components, so to solve them separately:

$$\int_s^n \int_w^e \frac{\partial uu}{\partial x} dxdy = \tag{17}$$
$$\int_s^n \left[\int_w^e \frac{\partial uu}{\partial x} dx \right] dy = \tag{18}$$
$$\int_s^n \left[u_e u_e - u_w u_w \right] dy = \tag{19}$$
$$(u_e u_e - u_w u_w)\Delta y \tag{20}$$

Where Δy is the dimension of the control volume in the y direction.
Since our equation is highly non-linear, we can try to linearize it by introducing the concept of:

- Advecting velocity: previous step velocity $\hat{u_e} = \frac{1}{2}(u_P + u_E), \hat{u_w} = \frac{1}{2}(u_W + u_P) \rightarrow$ Known.
- Advected velocity: current step velocity $u_e, u_w \rightarrow$ Unknown, evaluated using the Convection schemes (following chapter).

We can then rewrite the convection term for the x direction as:

$$\int_V \frac{\partial uu}{\partial x} dV = (\hat{u_e} u_e - \hat{u_w} u_w)\Delta y \tag{21}$$

The same procedure can be applied to the convection term for the y direction, and the final form for the convection term is given by:

$$\int_V \frac{\partial uu}{\partial x} + \frac{\partial vu}{\partial y} dV = (\hat{u_e} u_e - \hat{u_w} u_w)\Delta y + (\hat{v_n} u_n - \hat{v_s} u_s)\Delta x \tag{22}$$

Diffusion Term The finite volume discretization of the diffusion term is given by:

$$\int_V \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) dV = 0 \tag{23}$$

We further split the diffusion term into its directional components, so to solve them separately:

$$\int_s^n \int_w^e \nu \left(\frac{\partial^2 u}{\partial x^2} \right) dxdy = \tag{24}$$
$$\int_s^n \left[\int_w^e \frac{\partial}{\partial x} \left(\nu \frac{\partial u}{\partial x} \right) dx \right] dy = \tag{25}$$
$$\int_s^n \left[\nu \frac{\partial u}{\partial x}\bigg|_e - \nu \frac{\partial u}{\partial x}\bigg|_w \right] dy = \tag{26}$$
$$\nu \left(\frac{\partial u}{\partial x}\bigg|_e - \frac{\partial u}{\partial x}\bigg|_w \right) \Delta y \tag{27}$$

Where the terms $\left(\frac{\partial u}{\partial x}\Big|_{Position} \right)$ will be approximated using a Taylor expansion.
In particular, our final form for the diffusion term for the x direction is given by:

$$\int_V \nu \left(\frac{\partial^2 u}{\partial x^2} \right) dV = \nu \left(\approx \frac{\partial^2 u}{\partial x^2}\bigg|_{x \rightarrow 0} \right) \Delta x \Delta y \tag{28}$$

The same procedure can be applied to the diffusion term for the y direction, and the final form for the diffusion term is given by:

$$\int_V \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) dV = \nu \left(\approx \frac{\partial^2 u}{\partial x^2}\bigg|_{x \rightarrow 0} \right) \Delta x \Delta y + \nu \left(\approx \frac{\partial^2 u}{\partial y^2}\bigg|_{y \rightarrow 0} \right) \Delta x \Delta y \tag{29}$$

More details on the approximation of the second order derivatives will be reported in the following chapters.

Source Term The finite volume discretization of the source term is given by:

$$\int_V -\frac{\partial p}{\partial x} dV = 0 \tag{30}$$

Evaluating the integral based on the assumption 6, we obtain:

$$\int_s^n \int_w^e -\frac{\partial p}{\partial x} dx dy = \tag{31}$$

$$\int_s^n \left[-\int_w^e \frac{\partial p}{\partial x} dx \right] dy = \tag{32}$$

$$\int_s^n \left[-p_e + p_w \right] dy = \tag{33}$$

$$-(p_e - p_w)\Delta y \tag{34}$$

Final Form We can now substitute the discretized convection, diffusion and source terms into the momentum equations for both the x and y directions (13 14), and obtain the final form for the discretized momentum equations:

$$(\hat{u_e} u_e - \hat{u_w} u_w)\Delta y + (\hat{v_n} u_n - \hat{v_s} u_s)\Delta x - \nu \left(\approx \left. \frac{\partial^2 u}{\partial x^2} \right|_{x \to 0} + \approx \left. \frac{\partial^2 u}{\partial y^2} \right|_{y \to 0} \right) \Delta x \Delta y - (p_e - p_w)\Delta y = 0 \tag{35}$$

$$(\hat{u_e} v_e - \hat{u_w} v_w)\Delta y + (\hat{v_n} v_n - \hat{v_s} v_s)\Delta x - \nu \left(\approx \left. \frac{\partial^2 v}{\partial x^2} \right|_{x \to 0} + \approx \left. \frac{\partial^2 v}{\partial y^2} \right|_{y \to 0} \right) \Delta x \Delta y - (p_n - p_s)\Delta x = 0 \tag{36}$$

4 Schemes

In this section, we will present the schemes used to solve the previously quasi-discretized equations 22 and 29. Those schemes are the "tools" that will allow us to bind the currently considered cell values with its neighbor's cell values. This means that at the end of this section, we will have a set of coefficients (based on the scheme adopted) that will be used to assemble the solving matrix of the system.
To further clarify, what we want to achieve is to compute the following functions (nb indicates a generic neighbor cell, one or more cells away from the current cell P):

$$u_P = f(u_{nb}, v_{nb}, p_{nb}) \tag{37}$$

$$v_P = f(u_{nb}, v_{nb}, p_{nb}) \tag{38}$$

As we will see, the result of the schemes will be a set of coefficients that will be used to compute the values of u_P and v_P based on the values of the neighbor cells. In particular, the final form of the system will be:

$$A_P^u u_P = \sum_{nb} A_{nb}^u u_{nb} + b_P^u \tag{39}$$

$$A_P^v v_P = \sum_{nb} A_{nb}^v v_{nb} + b_P^v \tag{40}$$

Notice how there is no direct correlation between the u and v equations, that are instead coupled through the pressure term p.
Using the indices' notation system (i, j) becomes:

$$(A_P^u)_{i,j} u_{i,j} = \sum_{nb} (A_{nb}^u)_{i,j} u_{i,j} + (p_{i,j} - p_{i+1,j})\Delta y \tag{41}$$

$$(A_P^v)_{i,j} v_{i,j} = \sum_{nb} (A_{nb}^v)_{i,j} v_{i,j} + (p_{i,j} - p_{i,j+1})\Delta x \tag{42}$$

Similar to the approach used in the previous section, we will treat separately the convection and diffusion terms. In this way we will obtain two different sets of coefficients, one for the convection term and one for the diffusion

term. Those coefficients will be then reassembled together to reduce the equations to the final form presented above.
In particular, we will have that the coefficients A_P^ϕ and A_{nb}^ϕ will be the difference between the convection and diffusion coefficients, given that from Equation 15, we know:

$$FVM_{\text{Convection}} - FVM_{\text{Diffusion}} = 0 \rightarrow (A^\phi) = (A^\phi)_{\text{Convection}} - (A^\phi)_{\text{Diffusion}} \tag{43}$$

Note: In the following sections, we will refer to the use of `Mathematica`. The complete notebook used to obtain the final form of the coefficients is left in the 7.5 section of this document.

4.1 Convection schemes

In this section, we will present the schemes related to the convection terms of the discretized governing equations, which were derived in Section 3.2.

4.1.1 Upwind Differencing Scheme (UDS)

The Upwind Differencing Scheme (UDS) is the simplest convection scheme, and it is a $1^{th} - order$ scheme.
The idea behind the UDS is to consider the velocity at the face of the U_{CV} equal to the upwind value between the current cell center value and the neighbor cell center value.
This means that UDS scheme can be visualized as:

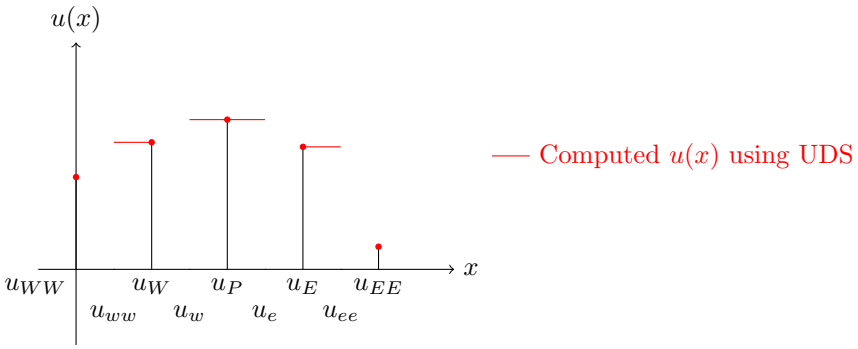

Figure 7: Example of the UDS scheme applied to the u velocity component.

We can obtain a formula for the UDS that will be used in the implementation of the code. In particular, by defining the Volume Fluxes as: $F_e = \hat{u_e}\Delta y$ & $F_w = \hat{u_w}\Delta y$, we can write that:

$$u_e = \begin{cases} u_P & \text{if } F_e > 0 \\ u_E & \text{if } F_e < 0 \end{cases} \tag{44}$$

$$u_w = \begin{cases} u_W & \text{if } F_w > 0 \\ u_P & \text{if } F_w < 0 \end{cases} \tag{45}$$

The same apply for the v velocity component, but in this case, we will have $F_n = \hat{v_n}\Delta x$, $F_s = \hat{v_s}\Delta x$.
In the end, our Upwind Differencing Scheme (UDS) scheme applied to the convection term will be:

$$\begin{gathered} (\hat{u_e}u_e - \hat{u_w}u_w)\,\Delta y + (\hat{v_n}u_n - \hat{v_s}u_s)\,\Delta x = \\ (F_e u_e - F_w u_w) + (F_n u_n - F_s u_s) = \\ u_P * max(F_e, 0) + u_E * min(F_e, 0) + u_W * max(F_w, 0) + u_P * min(F_w, 0) + \\ u_P * max(F_n, 0) + u_N * min(F_n, 0) + u_S * max(F_s, 0) + u_P * min(F_s, 0) \end{gathered}$$

Since we are interested in the A_P^ϕ and A_{nb}^ϕ coefficients as written in the form of Equation 40, with the help of `Mathematica`, we can regroup the terms based on the velocity components, perform some sign manipulations and obtain the coefficients for the convection term using the UDS scheme for both the u and v momentum equations.

$$\text{Convection Coefficients UDS: } = \textbf{See table 1} \tag{46}$$

4.1.2 Central Differencing Scheme (CDS)

The Central Differencing Scheme (CDS) is a $2^{nd} - order$ scheme.
The idea behind the CDS is to consider the velocity at the face of the control volume as the average between the current cell center value and the neighbor cell center value.
This means that CDS scheme can be visualized as:

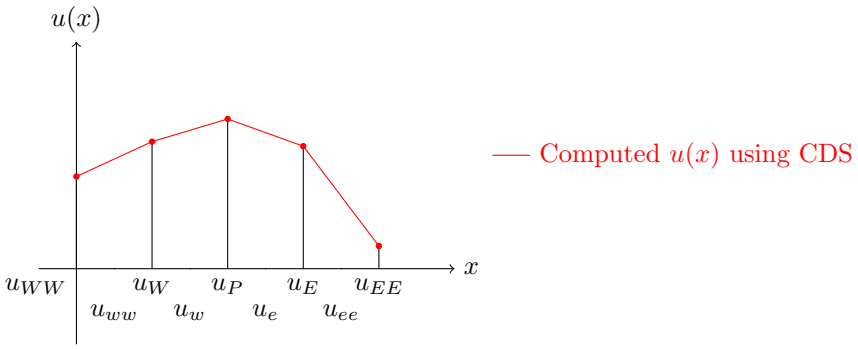

Figure 8: Example of the CDS scheme applied to the u velocity component.

We can obtain a formula for the CDS that will be used in the implementation of the code. In particular, by using the same definition of the Volume Fluxes as before, we can write that:

$$u_e = \frac{u_P + u_E}{2} \tag{47}$$

$$u_w = \frac{u_W + u_P}{2} \tag{48}$$

The same apply for the v velocity component.
In the end, our CDS scheme applied to the convection term will be:

$$\begin{gathered}
(\hat{u_e} u_e - \hat{u_w} u_w)\,\Delta y + (\hat{v_n} u_n - \hat{v_s} u_s)\,\Delta x = \\
(F_e u_e - F_w u_w) + (F_n u_n - F_s u_s) = \\
F_e \frac{u_P + u_E}{2} - F_w \frac{u_W + u_P}{2} + F_n \frac{u_P + u_N}{2} - F_s \frac{u_S + u_P}{2}
\end{gathered}$$

Since we are interested in the A_P^ϕ and A_{nb}^ϕ coefficients as written in the form of Equation 40, with the help of `Mathematica`, we can regroup the terms based on the velocity components, perform some sign manipulations and obtain the coefficients for the convection term using the CDS scheme for both the u and v momentum equations.

Convection Coefficients CDS: = **See table 1**

4.1.3 Quadratic Upstream Interpolation for Convective Kinematics (QUICK)

The Quadratic Upstream Interpolation for Convective Kinematics (QUICK) is a $3^{rd} - order$ scheme.
The idea behind the QUICK scheme is to interpolate the velocity at the center of 3 cells to then compute the velocity at the face of the cell. The choice of the cells to interpolate is based on the direction of the velocity at previous step ($\hat{u}$ or $\hat{v}$) similarly to the UDS scheme. For example, if we are computing the u_e velocity, we will pick as interpolations points the u_P, u_E velocity and u_{EE} or u_W based on the direction of the velocity at the face.
The QUICK scheme can be visualized as:

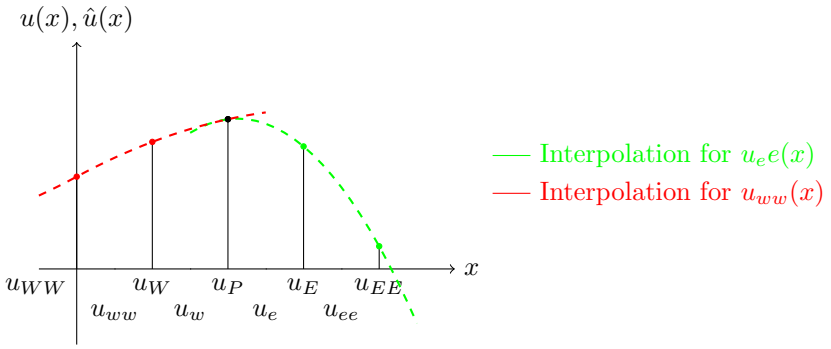

Figure 9: Example of the QUICK scheme applied to the u velocity component.

We can obtain a formula for the QUICK that will be used in the implementation of the code. In particular, we can recall the definition of the Lagrange interpolation polynomial of degree 2:

$$P(x) = \sum_{i=0}^{2} y_i \prod_{j=0, j\neq i}^{2} \frac{x - x_j}{x_i - x_j} \tag{49}$$

In our case, it's convenient to fix the reference system in correspondence of the center of the control volume considered. If that is the case, then for example the evaluation of the velocity at the face u_e will be:

$$u_e = P(x)\Big|_{x=\Delta x/2} \tag{50}$$

As we said before, the choice of the interpolation points is based on the direction of the velocity at the face of the control volume. In particular, if we are computing the u_e velocity, the possible polynomials ($P_e(x)$) will be:

$$P_e(x) = \begin{cases} P(u_W, u_P, u_E) & \text{if } \hat{u_e} > 0 \\ P(u_P, u_E, u_{EE}) & \text{if } \hat{u_e} < 0 \end{cases} \tag{51}$$

By using `Mathematica`, we can compute the polynomials for the u velocity component in both cases.
The results of the symbolic analysis are:

$$u_e = \begin{cases} -\frac{1}{8}u_W + \frac{3}{4}u_P + \frac{3}{8}u_E & \text{if } \hat{u_e} > 0 \\ +\frac{3}{8}u_P + \frac{3}{4}u_E - \frac{1}{8}u_{EE} & \text{if } \hat{u_e} < 0 \end{cases} \tag{52}$$

$$u_w = \begin{cases} -\frac{1}{8}u_{WW} + \frac{3}{4}u_W + \frac{3}{8}u_P & \text{if } \hat{u_w} > 0 \\ +\frac{3}{8}u_W + \frac{3}{4}u_P - \frac{1}{8}u_E & \text{if } \hat{u_w} < 0 \end{cases} \tag{53}$$

The same apply for the v velocity component, but in this case, the variables that decide the interpolation points are the $\hat{v_n}$ and $\hat{v_s}$.
In the end, our QUICK scheme applied to the convection term will be:

$$(\hat{u_e}u_e - \hat{u_w}u_w)\,\Delta y + (\hat{v_n}u_n - \hat{v_s}u_s)\,\Delta x = \tag{54}$$

$$(F_e u_e - F_w u_w) + (F_n u_n - F_s u_s) = \tag{55}$$

$$\frac{1}{8}((3u_E + 6u_P - u_W)max(0, F_e) + (-u_{EE} + 6u_E + 3u_P)min(0, F_e) + \tag{56}$$

$$(-3u_P - 6u_W + u_{WW})max(0, F_w) + (u_E - 6u_P - 3u_W)min(0, F_w) + \tag{57}$$

$$(3u_N + 6u_P - u_S)max(0, F_n) + (6u_N - u_{NN} + 3u_P)min(0, F_n) + \tag{58}$$

$$(-3u_P - 6u_S + u_{SS})max(0, F_s) + (u_N - 6u_P - 3u_S)min(0, F_s)) \tag{59}$$

Since we are interested in the A_P^ϕ and A_{nb}^ϕ coefficients as written in the form of Equation 40, with the help of `Mathematica`, we can regroup the terms based on the velocity components, perform some sign manipulations and obtain the coefficients for the convection term using the QUICK scheme for both the u and v momentum equations.

Convection Coefficients QUICK: = **See table 1**

4.2 Diffusion Schemes

In this section, we will present the schemes related to the diffusion terms of the discretized governing equations, which were derived in Section 3.2.
The general idea here is to approximate the diffusion term using its Taylor expansion and then evaluate the derivative at the cell faces. As a recall, the definition of the Taylor expansion of order n is:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \cdots + \frac{f^n(x_0)}{n!}(x - x_0)^n + \mathbf{O}((x - x_0)^{n+1}) \tag{60}$$

4.2.1 2^{nd}-order scheme

The 2^{nd}-order scheme approximate the diffusion term using the Taylor expansion up to the second order.
We can start by writing the polynomial that approximates by interpolation the values of the variable u at the cell faces. To do so, we can use the Lagrange interpolation polynomial, previously defined in Equation 49. Since we are now analyzing the 2^{nd}-order scheme, we will use $n = 2 + 1 = 3$ points to interpolate.
As before, we can use `Mathematica` to obtain the polynomial that approximates function $u(x)$ as:

$$u(x) \approx \frac{2(\Delta x)^2 u_P + x^2(u_E - 2u_P + u_W) + \Delta x x(u_E - u_W)}{2(\Delta x)^2} \tag{61}$$

We can now proceed by evaluating the second derivative of the polynomial as:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_E - 2u_P + u_W}{(\Delta x)^2} \tag{62}$$

And finally, we can the approximated second derivative at the point of interest $x = 0$ as:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_P \approx \frac{u_E - 2u_P + u_W}{(\Delta x)^2} \tag{63}$$

Given the definition of the diffusion term of u along x in Equation 29, we can write the 2^{nd}-order scheme as:

$$\nu\frac{\partial^2 u}{\partial x^2} = \nu\frac{u_E - 2u_P + u_W}{(\Delta x)^2} \tag{64}$$

With a similar approach, we can obtain the 2^{nd}-order scheme for the y direction as:

$$\nu\frac{\partial^2 u}{\partial y^2} = \nu\frac{u_N - 2u_P + u_S}{(\Delta y)^2} \tag{65}$$

The same procedure can be applied to the v momentum equation, obtaining the following results:

$$\nu\frac{\partial^2 u}{\partial x^2} + \nu\frac{\partial^2 u}{\partial y^2} = \tag{66}$$

$$\frac{\Delta y \nu(u_E - 2u_P + u_W)}{\Delta x} + \tag{67}$$

$$\frac{\Delta x \nu(u_N - 2u_P + u_S)}{\Delta y} \tag{68}$$

Since we are interested in the A_P^ϕ and A_{nb}^ϕ coefficients as written in the form of Equation 40, with the help of `Mathematica`, we can regroup the terms based on the velocity components, perform some sign manipulations and obtain the coefficients for the diffusion term using the 2^{nd}-order scheme for both the u and v momentum equations.

$$\text{Diffusion Coefficients order} 2^{nd} = \textbf{See table 1} \tag{69}$$

4.2.2 4^{th}-order scheme

The 4^{th}-order scheme approximate the diffusion term using the Taylor expansion up to the fourth order.
Since the reasoning is similar to the 2^{nd}-order scheme, we can directly write the approximated second derivative at the point of interest $x = 0$ as:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_P \approx -\frac{u_{WW} - 16u_W + 30u_P - 16u_E + u_{EE}}{12(\Delta x)^2} \tag{70}$$

Given the definition of the diffusion term of u along x in Equation 29, we can write the 4^{th}-order scheme as:

$$\nu\frac{\partial^2 u}{\partial x^2} = \nu - \frac{u_{WW} - 16u_W + 30u_P - 16u_E + u_{EE}}{12(\Delta x)^2} \tag{71}$$

With a similar approach, we can obtain the 4^{th}-order scheme for the y direction as:

$$\nu\frac{\partial^2 u}{\partial y^2} = \nu - \frac{u_{WW} - 16u_W + 30u_P - 16u_E + u_{EE}}{12(\Delta y)^2} \tag{72}$$

The same procedure can be applied to the v momentum equation, obtaining the following results:

$$\int_V \nu\frac{\partial^2 u}{\partial x^2} + \nu\frac{\partial^2 u}{\partial y^2} dV = \tag{73}$$

$$-\frac{\nu}{12}\left(\frac{u_{WW} - 16u_W + 30u_P - 16u_E + u_{EE}}{(\Delta x)^2} + \frac{u_{WW} - 16u_W + 30u_P - 16u_E + u_{EE}}{(\Delta y)^2}\right)\Delta x\Delta y \tag{74}$$

Since we are interested in the A_P^ϕ and A_{nb}^ϕ coefficients as written in the form of Equation 40, with the help of `Mathematica`, we can regroup the terms based on the velocity components, perform some sign manipulations and obtain the coefficients for the diffusion term using the 4^{th}-order scheme for both the u and v momentum equations.

$$\text{Diffusion Coefficients order } 4^{th} = \textbf{See table 1} \tag{75}$$

4.3 Final coefficients

The final coefficients are obtained by reassembling the convection and diffusion coefficients, based on the FVM result:

$$FVM_{\text{Convection}} - FVM_{\text{Diffusion}} = 0 \rightarrow (A^\phi) = (A^\phi)_{\text{Convection}} - (A^\phi)_{\text{Diffusion}} \tag{76}$$

Table 1: Final Ap coefficients for convection and diffusion using different schemes

Ap	**Convection UDS**	**Convection CDS**	**Convection QUICK**	**Diffusion 2° order**	**Diffusion 4° order**
$Ap\phi WWSS$	0	0	0	0	0
$Ap\phi WWS$	0	0	0	0	0
$Ap\phi WWP$	0	0	$-\frac{1}{8}\max(0, Fw\phi)$	0	$-\frac{\Delta y\nu}{12\Delta x}$
$Ap\phi WWN$	0	0	0	0	0
$Ap\phi WWNN$	0	0	0	0	0
$Ap\phi WSS$	0	0	0	0	0
$Ap\phi WS$	0	0	0	0	0
$Ap\phi WP$	$\max(0, Fw\phi)$	$\frac{Fw\phi}{2}$	$\frac{1}{8}\max(0, Fe\phi) + \frac{3}{4}\max(0, Fw\phi) + \frac{3}{8}\min(0, Fw\phi)$	$\frac{\Delta y\nu}{\Delta x}$	$\frac{4\Delta y\nu}{3\Delta x}$
$Ap\phi WN$	0	0	0	0	0
$Ap\phi WNN$	0	0	0	0	0
$Ap\phi PSS$	0	0	$-\frac{1}{8}\max(0, Fs\phi)$	0	$-\frac{\Delta x\nu}{12\Delta y}$
$Ap\phi PS$	$\max(0, Fs\phi)$	$\frac{Fs\phi}{2}$	$\frac{1}{8}\max(0, Fn\phi) + \frac{3}{4}\max(0, Fs\phi) + \frac{3}{8}\min(0, Fs\phi)$	$\frac{\Delta x\nu}{\Delta y}$	$\frac{4\Delta x\nu}{3\Delta y}$
$Ap\phi PP$	$\sum_{nb} Ap\phi$	$\sum_{nb} Ap\phi$	$\sum_{nb} Ap\phi$	$\sum_{nb} Ap\phi$	$\sum_{nb} Ap\phi$
$Ap\phi PN$	$-\min(0, Fn\phi)$	$-\frac{Fn\phi}{2}$	$\frac{1}{8}(-3)\max(0, Fn\phi) - \frac{3}{4}\min(0, Fn\phi) - \frac{1}{8}\min(0, Fs\phi)$	$\frac{\Delta x\nu}{\Delta y}$	$\frac{4\Delta x\nu}{3\Delta y}$
$Ap\phi PNN$	0	0	$\frac{1}{8}\min(0, Fn\phi)$	0	$-\frac{\Delta x\nu}{12\Delta y}$
$Ap\phi ESS$	0	0	0	0	0
$Ap\phi ES$	0	0	0	0	0
$Ap\phi EP$	$-\min(0, Fe\phi)$	$-\frac{Fe\phi}{2}$	$\frac{1}{8}(-3)\max(0, Fe\phi) - \frac{3}{4}\min(0, Fe\phi) - \frac{1}{8}\min(0, Fw\phi)$	$\frac{\Delta y\nu}{\Delta x}$	$\frac{4\Delta y\nu}{3\Delta x}$
$Ap\phi EN$	0	0	0	0	0
$Ap\phi ENN$	0	0	0	0	0
$Ap\phi EESS$	0	0	0	0	0
$Ap\phi EES$	0	0	0	0	0
$Ap\phi EEP$	0	0	$\frac{1}{8}\min(0, Fe\phi)$	0	$-\frac{\Delta y\nu}{12\Delta x}$
$Ap\phi EEN$	0	0	0	0	0
$Ap\phi EENN$	0	0	0	0	0

5 Symmetric Coupled Gauss-Seidel (SCGS)

So far we have presented the set of differential equations that govern the fluid flow, and we have also discretized them using the FVM and applying different schemes (for the convection and diffusion terms).
In the following we will discuss the Symmetric Coupled Gauss-Seidel (SCGS) method by [2, Vanka (1986)], which combine all the information we have presented so far, and will allow us to solve the discretized equations iteratively.
Notice that when we say *solve the equations*, we mean to find the values of the velocity components and the pressure at each cell of the domain, namely $U(x_i, y_j)$ & $V(x_i, y_j)$ & $P(x_i, y_j)$, so to have a complete description of the flow field.

5.1 Variable Correction Concept

Before presenting the SCGS method itself, it's now useful to introduce the concept of variable correction, which is crucial for the understanding of the method.
The idea here is to observe a generic current state of the system $\phi^{(n)}$, as the sum of a known part $\phi^{(n-1)}$ and a correction $\delta\phi^{(n)}$:

$$\phi^{(n)} = \phi^{(n-1)} + \delta\phi^{(n)} \tag{77}$$

Where ϕ is a generic variable, and n is the iteration index.
For simplicity of notation, we will drop the iteration index in the following, and we will rewrite Equation 77 as:

$$\phi = \phi^* + \phi' \tag{78}$$

Where ϕ^* is the known part, and ϕ' is the correction.
By doing so, we can rewrite the state of the system at the current iteration as:

$$u_{i,j} = u^*_{i,j} + u'_{i,j} \tag{79}$$
$$v_{i,j} = v^*_{i,j} + v'_{i,j} \tag{80}$$
$$p_{i,j} = p^*_{i,j} + p'_{i,j} \tag{81}$$

5.2 Equations Coupling

At this point, one could ask how the three equations we have presented so far are related to each other. The answer is in the continuity equation, which is the key to couple the velocity and pressure fields (and indeed, have a close form of the system with 3 equations 12 35 36 and 3 unknowns).
Given that the continuity equation is defined for a given $p - CV$ and that involve the all the four velocity components at the faces of the control volume, it should be intuitive that in order to solve the equilibrium across one cell, we need to have a system of equations that involve all the four velocity components.
By recalling the representation with index notation for the $p - CV$, we can compose our system of equations.

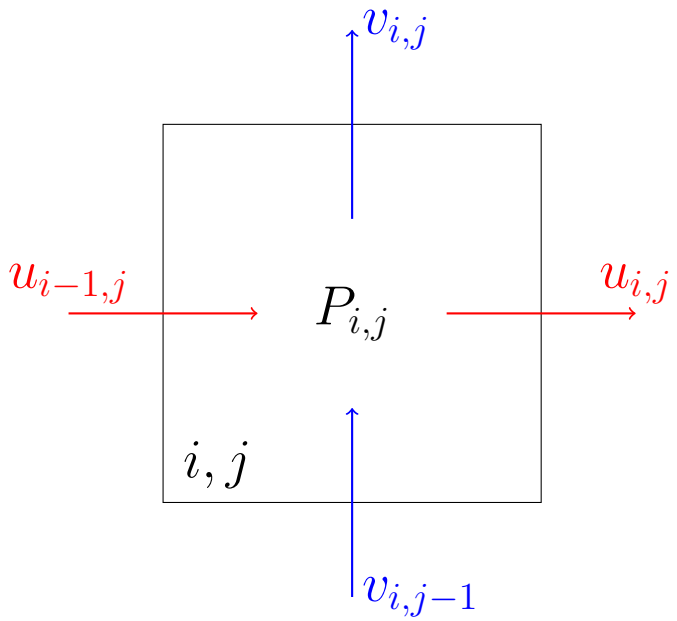

Figure 10: Generic $p - CV$ with the velocity components at the faces.

$$\begin{cases} \text{Mom. } u_{i-1,j} \\ \text{Mom. } u_{i,j} \\ \text{Mom. } v_{i,j-1} \\ \text{Mom. } v_{i,j} \\ \text{Con.} \end{cases} = \begin{cases} (A_P^u)_{i-1,j} u_{i-1,j} & = \sum_{nb} (A_{nb}^u)_{i-1,j} (u_{nb})_{i-1,j} + (p_{i-1,j} - p_{i,j}) \Delta y \\ (A_P^u)_{i,j} u_{i,j} & = \sum_{nb} (A_{nb}^u)_{i,j} (u_{nb})_{i,j} + (p_{i,j} - p_{i+1,j}) \Delta y \\ (A_P^v)_{i,j-1} v_{i,j-1} & = \sum_{nb} (A_{nb}^v)_{i,j-1} (v_{nb})_{i,j-1} + (p_{i,j-1} - p_{i,j}) \Delta x \\ (A_P^v)_{i,j} v_{i,j} & = \sum_{nb} (A_{nb}^v)_{i,j} (v_{nb})_{i,j} + (p_{i,j} - p_{i,j+1}) \Delta x \\ (u_{i,j} - u_{i-1,j}) \Delta y + (v_{i,j} - v_{i,j-1}) \Delta x & = 0 \end{cases} \tag{82}$$

The system of equations is composed by 5 equations and 5 unknowns ($u_{i-1,j}$, $u_{i,j}$, $v_{i,j-1}$, $v_{i,j}$, $p_{i,j}$), and it's the result of the coupling of the momentum and continuity equations across a generic $p - CV$.

5.3 Residual Concept

Being the SCGS an iterative method, it's crucial to introduce the concept of residual, which is the key to understand the convergence of the method.
The residual is the difference between the left-hand side and the right-hand side of the discretized equations, and it's a measure of the error of the current state of the system.
For our system of 5 equations, we can define 5 residuals, one for each equation.
To simplify the set of equations (and have a simpler matrix resolution afterwards), we can choose to apply the correction to only some variables, and not to all of them.
In particular, we can choose to neglect the correction for the velocity and pressure components regarding the neighbor cells, and apply the correction only to the pressure and the velocity components at the current cell.
By doing so, we can rewrite the system of equations as:

$$(A_P^u)_{i-1,j}(u^*_{i-1,j} + u'_{i-1,j}) = \sum_{nb} (A_{nb}^u)_{i-1,j} (u^*_{nb})_{i-1,j} + (p^*_{i-1,j} - (p^*_{i,j} + p'_{i,j})) \Delta y \tag{83}$$

$$(A_P^u)_{i,j}(u^*_{i,j} + u'_{i,j}) = \sum_{nb} (A_{nb}^u)_{i,j} (u^*_{nb})_{i,j} + ((p^*_{i,j} + p'_{i,j}) - p^*_{i+1,j}) \Delta y \tag{84}$$

$$(A_P^v)_{i,j-1}(v^*_{i,j-1} + v'_{i,j-1}) = \sum_{nb} (A_{nb}^v)_{i,j-1} (v^*_{nb})_{i,j-1} + (p^*_{i,j-1} - (p^*_{i,j} + p'_{i,j})) \Delta x \tag{85}$$

$$(A_P^v)_{i,j}(v^*_{i,j} + v'_{i,j}) = \sum_{nb} (A_{nb}^v)_{i,j} (v^*_{nb})_{i,j} + ((p^*_{i,j} + p'_{i,j}) - p^*_{i,j+1}) \Delta x \tag{86}$$

$$((u^*_{i,j} + u'_{i,j}) - (u^*_{i-1,j} + u'_{i-1,j})) \Delta y + ((v^*_{i,j} + v'_{i,j}) - (v^*_{i,j-1} + v'_{i,j-1})) \Delta x = 0 \tag{87}$$

As explained in Subsection 5.1, the * superscript is used to indicate the value of the variable at the previous iteration, and the $'$ superscript is used to indicate the correction to apply to the variable. This means that the ϕ^* are known values, and the ϕ' are the unknowns to solve for.

We can now rearrange each equation to have the unknowns on the left-hand side and the known on the right-hand side:

$$(A_P^u)_{i-1,j}u'_{i-1,j} + p'_{i,j}\Delta y = R^u_{i-1,j} \tag{88}$$
$$(A_P^u)_{i,j}u'_{i,j} - p'_{i,j}\Delta y = R^u_{i,j} \tag{89}$$
$$(A_P^v)_{i,j-1}v'_{i,j-1} + p'_{i,j}\Delta x = R^v_{i,j-1} \tag{90}$$
$$(A_P^v)_{i,j}v'_{i,j} - p'_{i,j}\Delta x = R^v_{i,j} \tag{91}$$
$$(u'_{i,j} - u'_{i-1,j})\Delta y + (v'_{i,j} - v'_{i,j-1})\Delta x = R^c_{i,j} \tag{92}$$

Where $R^\phi_{i,j}$ is the residual of the ϕ equation at the i,j cell.
The residuals, computed from the previous system by moving the known terms to the right-hand side, are defined as:

$$R^u_{i-1,j} = \sum_{nb}(A^u_{nb})_{i-1,j}(u^*_{nb})_{i-1,j} - (A^u_P)_{i-1,j}u^*_{i-1,j} + (p^*_{i-1,j} - p^*_{i,j})\Delta y \tag{93}$$
$$R^u_{i,j} = \sum_{nb}(A^u_{nb})_{i,j}(u^*_{nb})_{i,j} - (A^u_P)_{i,j}u^*_{i,j} + (p^*_{i,j} - p^*_{i+1,j})\Delta y \tag{94}$$
$$R^v_{i,j-1} = \sum_{nb}(A^v_{nb})_{i,j-1}(v^*_{nb})_{i,j-1} - (A^v_P)_{i,j-1}v^*_{i,j-1} + (p^*_{i,j-1} - p^*_{i,j})\Delta x \tag{95}$$
$$R^v_{i,j} = \sum_{nb}(A^v_{nb})_{i,j}(v^*_{nb})_{i,j} - (A^v_P)_{i,j}v^*_{i,j} + (p^*_{i,j} - p^*_{i,j+1})\Delta x \tag{96}$$
$$R^c_{i,j} = -\left[(u^*_{i,j} - u^*_{i-1,j})\Delta y + (v^*_{i,j} - v^*_{i,j-1})\Delta x\right] \tag{97}$$

5.4 Gauss-Seidel Iterative Method

Before, proceeding with the solution of the SCGS method, we need to introduce the Gauss-Seidel Iterative Method (GS) method.
The GS method is an iterative method used to solve a system of linear equations, and it's based on the idea of solving one equation at a time, and using the updated values of the variables to solve the next equation.
For our purpose, is interesting to observe how the introduction of the 'under-relaxation' factor α can help to improve the convergence of the method.
In particular, we can apply the GS method to solve a similar equation of our system, as for example:

$$(A^\phi_P)\phi_P = \sum_{nb}(A^\phi_{nb})\phi_{nb} + S^\phi \tag{98}$$

Where ϕ is a generic variable, and S^ϕ is the source term of the equation.
The GS method can be written as:

$$\phi_P = \frac{\sum_{nb}(A^\phi_{nb})\phi^*_{nb} + S^\phi}{A^\phi_P} + \phi^*_P \tag{99}$$
$$\phi_P = \alpha\left(\frac{\sum_{nb}(A^\phi_{nb})\phi^*_{nb} + S^\phi}{A^\phi_P} - \phi^*_P\right) + \phi^*_P \tag{100}$$

Where ϕ^*_P is the value of the variable at the previous iteration, and α is the under-relaxation factor.
By rearranging the last equation, we have:

$$\phi'_P = \frac{R^\phi_P}{\left(\frac{A^\phi_P}{\alpha_\phi}\right)} \tag{101}$$

Where ϕ'_P is the correction to apply to the variable, and R^ϕ_P is the residual of the equation at the P cell.
This result will be useful in the next section.

5.5 Vanka's approach

The SCGS method is based on the idea of solving the equations in a coupled way.
This means that for each iteration, we will obtain the correction for each of the 5 variables simultaneously. To make this possible, we need to introduce a new matrix, called the **Vanka matrix [2]**.
Considering the system of equations 92, we can write them in a matrix form as:

$$\begin{bmatrix} (A_P^u)_{i-1,j} & 0 & 0 & 0 & \Delta y \\ 0 & (A_P^u)_{i,j} & 0 & 0 & -\Delta y \\ 0 & 0 & (A_P^v)_{i,j-1} & 0 & \Delta x \\ 0 & 0 & 0 & (A_P^v)_{i,j} & -\Delta x \\ -\Delta y & \Delta y & -\Delta x & \Delta x & 0 \end{bmatrix} \begin{bmatrix} u'_{i-1,j} \\ u'_{i,j} \\ v'_{i,j-1} \\ v'_{i,j} \\ p'_{i,j} \end{bmatrix} = \begin{bmatrix} R^u_{i-1,j} \\ R^u_{i,j} \\ R^v_{i,j-1} \\ R^v_{i,j} \\ R^c_{i,j} \end{bmatrix} \tag{102}$$

As we can see, the Vanka matrix is a 5×5 matrix, and it's a function of the coefficients of the discretized equations, and of the grid spacing.
Our unknown is the vector of the corrections, being the A_P^ϕ coefficients and the R^ϕ residuals known from the previous iteration.
Given that, we can solve the system of equations 102 to obtain the corrections for the velocity and pressure components at each cell of the domain.
To do so, we must invert the Vanka matrix, and multiply it by the residuals vector:

$$\left[\phi'\right] = \left[A^\phi\right]^{-1} \left[R^\phi\right] \tag{103}$$

As stated in [2], the inverse of the Vanka matrix can be computed analytically using the following algorithm:

$$r_i = \frac{a_{5i}}{a_{ii}}, \quad i = 1, 2, 3, 4 \tag{104}$$

$$DEN = \sum_{i=1}^{4} a_{5i} r_i \tag{105}$$

$$x_5 = \frac{\left[\sum_{i=1}^{4} r_i b_i - b_5\right]}{DEN} \tag{106}$$

$$x_i = \frac{b_i - a_{i5} x_5}{a_{ii}}, \quad i = 1, 2, 3, 4 \tag{107}$$

Where a_{ij} are the elements of the Vanka matrix A^ϕ, b_i are the elements of the residuals vector R^ϕ, and x_i are the elements of the corrections vector ϕ'.
In case we want to apply an under-relaxation factor to the corrections, we can modify Vanka's Matrix as follows:

$$\begin{bmatrix} \frac{(A_P^u)_{i-1,j}}{\alpha_u} & 0 & 0 & 0 & \Delta y \\ 0 & \frac{(A_P^u)_{i,j}}{\alpha_u} & 0 & 0 & -\Delta y \\ 0 & 0 & \frac{(A_P^v)_{i,j-1}}{\alpha_v} & 0 & \Delta x \\ 0 & 0 & 0 & \frac{(A_P^v)_{i,j}}{\alpha_v} & -\Delta x \\ -\Delta y & \Delta y & -\Delta x & \Delta x & 0 \end{bmatrix} \tag{108}$$

Where α_u and α_v are the under-relaxation factors for the velocity components u and v respectively.

5.6 Boundary conditions for Lid-Driven Cavity problem

So far, we have presented all the tools and concepts required to solve a generic fluid flow problem (except for the algorithm itself) that respect the hypothesis and conditions stated at the beginning of Section 3.
In this section we will present the boundary conditions that will be used to solve the Lid-Driven Cavity problem and their implementation inside the SCGS method.

5.6.1 Ghosts cells

Consider that our domain has already been discretized (subdivided) into $N_x \times N_y$ cells.
If we think of the scheme for both the convection and diffusion presented before, we can see that at the boundary of our domain, the coefficients $A_p\phi$ 1 involve cells that are outside the domain. For this reason, we need to add a layer of cells outside the physical domain, called ghost cells, to calculate the coefficients at the boundary.

The number of ghost cells required depends on the order of the scheme used to calculate the coefficients. For a second-order scheme, we need at least one ghost cell, and for a fourth-order scheme, we need at least two ghost cells.
Here, and also in the implementation of the SCGS method in the code, we will consider the possibility to adopt a mixed strategy scheme and use:

- Away from boundaries: higher-order schemes
- Near boundaries: UDS scheme for convection and 2nd-order scheme for diffusion

By doing so, we ensure that at the boundaries both the schemes will be applicable with just a single layer of ghost cells.

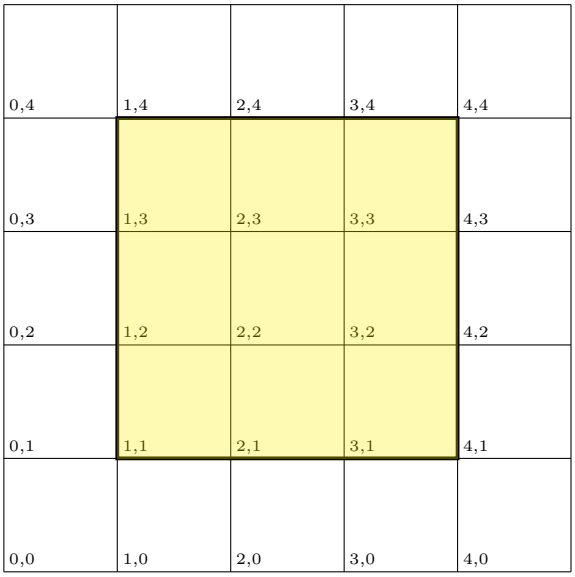

Figure 11: Physical domain and ghost cells

In Figure 11, the yellow area represents the physical domain, while the frame surrounding it represents the ghost cells.
As we can see, the indexes of the cells stick to the physical domain, while the ghost cells may have negative indexes.
In the case above, we can declare some of the problem that will be recalled during the following sections:

- Physical domain: $N_x \times N_y$ grid. In the example above, $N_x = N_y = 3$
- Physical domain indexes: from $(1, 1)$ to $(3, 3)$. In general, from $(1, 1)$ to (N_x, N_y).
- Ghost cells layer: may have different sizes along the x and y directions. In the example above, the ghost cells layer has a size of $N_g = 1$ cell in both directions. We will stick with this size for the rest of the document.
- Ghost cells indexes: may have negative indexes. In the example above, the ghost cells have indexes from $(0, 0)$ to $(4, 4)$, that is equivalent to $(1 - N_g, 1 - N_g)$ to $(N_x + 2 * N_g - 1, N_y + 2 * N_g - 1)$.

We can now proceed to identify the boundary conditions for the Lid-Driven Cavity problem.

5.6.2 No-slip condition

As boundary conditions for the Lid-Driven Cavity problem (and for most of the fluid flow problems), we have the no-slip condition, which gives us the constraint for two direction of the velocity field:

- Component normal to the wall: given a condition of impenetrability, the velocity component normal to the wall must be zero (no-penetration)
- Component tangential to the wall: given a condition of no-slip, the velocity component tangential to the wall must equal to the wall velocity

As an example, we can consider the cell in position $(1, N_y)$, which is the top-left corner of the physical domain (Figure 11 for reference).

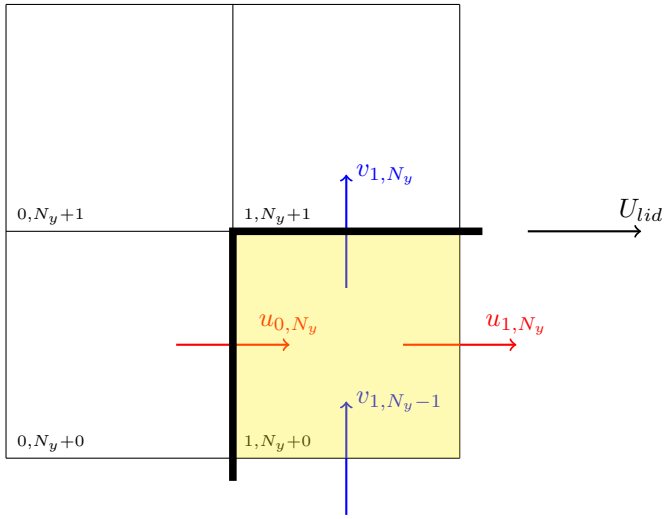

Figure 12: Boundary conditions for the cell $(1, N_y)$

In this case, we must accomplish the following conditions:

- $u_{0,N_y} = 0$ (no-penetration)
- $v_{1,N_y} = 0$ (no-penetration)
- $u_{1,N_y} = U_{lid}$ (no-slip)

Intuitively, we can understand that similar conditions must be applied to the other walls of the domain given the correction of the indexes and the direction of the velocity field.
In general, we can state that the boundary conditions for the Lid-Driven Cavity problem are:

- Bottom wall: $v_{i,0} = 0$ (no-penetration) + $u_{i,1} = 0$ (no-slip)
- Left wall: $u_{0,j} = 0$ (no-penetration) + $v_{1,j} = 0$ (no-slip)
- Top wall: $v_{i,N_y} = 0$ (no-penetration) + $u_{i,N_y} = U_{lid}$ (no-slip)
- Right wall: $u_{N_x,j} = 0$ (no-penetration) + $v_{N_x,j} = 0$ (no-slip)

We can now proceed to see how these conditions are accounted for in the SCGS method.

5.6.3 Boundary conditions inside SCGS method

By adopting the SCGS method, we can impose the boundary conditions in the following way:

- No-penetration condition: impose specific coefficients in the Vanka matrix to be zero
- No-slip condition: impose specific velocity values in the ghost cells

Moreover, as we are going to see, the no-penetration condition imply the velocity to be null and can be imposed to the system before solving the linear system of equations. Instead, given the mathematical formulation of the no-slip condition, we can impose the velocity values in the ghost cells only after solving the linear system of equations (and will affect the next iteration).

No-penetration condition The no-penetration condition imply the normal velocity component at the boundary to be null. This means that if we start from an initial guess of the velocity field equal to zero everywhere, we must ensure that correction of the velocity field at the boundary is null for each iteration.
Taken for example the cell $(1, N_y)$ as in Figure 12, we know that we must impose $u_{0,N_y} = 0$ & $v_{1,N_y} = 0$, which means that the correction of the velocity field at the boundary must be null as well $u'_{0,N_y} = 0$ & $v'_{1,N_y} = 0$.
By recalling the formulation for both the continuity and momentum equations, we can see that multiple terms must be canceled out (known null solution).

For the **continuity equation** written at the cell $(1, N_y)$, we have:

$$-u'_{0,N_y}\Delta y + u'_{1,N_y}\Delta y - v'_{1,N_y-1}\Delta x + v'_{1,N_y}\Delta x = R^c_{1,N_y} \tag{109}$$

By imposing $[A^\phi]_{5,1} = \Delta y = 0$, we can cancel out the first term, and by imposing $[A^\phi]_{5,4} = \Delta x = 0$, we can cancel out the third term.
In this way, even if we are not imposing the no-penetration condition directly, we are imposing an equivalent condition that will lead to the same result.
More in general, given the physical domain of size $N_x \times N_y$, we can impose the no-penetration condition for the continuity equation by setting:

$$@i = 1, u'_{i-1,j} = 0 \rightarrow [A^\phi]_{5,1} = -\Delta y = 0 \quad \forall j = 1, 2, ..., N_y \tag{110}$$
$$@i = N_x, u'_{i,j} = 0 \rightarrow [A^\phi]_{5,2} = \Delta y = 0 \quad \forall j = 1, 2, ..., N_y \tag{111}$$
$$@j = 1, v'_{i,j-1} = 0 \rightarrow [A^\phi]_{5,3} = -\Delta x = 0 \quad \forall i = 1, 2, ..., N_x \tag{112}$$
$$@j = N_y, v'_{i,j} = 0 \rightarrow [A^\phi]_{5,4} = \Delta x = 0 \quad \forall i = 1, 2, ..., N_x \tag{113}$$

For the **momentum equations**, written at the cell $(1, N_y)$ for the two components that are controlled by the no-penetration condition, we have:

$$(A^u_P)_{0,N_y} u'_{0,N_y} + p'_{0,N_y}\Delta y = R^u_{0,N_y} \tag{114}$$
$$(A^v_P)_{1,N_y} v'_{1,N_y} - p'_{1,N_y}\Delta x = R^v_{1,N_y} \tag{115}$$

By imposing $[A^\phi]_{1,5} = \Delta y = 0$ and $[R^\phi]_1 = 0$, we can cancel out the first equation, and by imposing $[A^\phi]_{4,5} = -\Delta x = 0$ and $[R^\phi]_4 = 0$, we can cancel out the second equation.
More in general, given the physical domain of size $N_x \times N_y$, we can impose the no-penetration condition for the momentum equations by setting:

$$@i = 1, u'_{i-1,j} = 0 \rightarrow [A^\phi]_{1,5} = \Delta y = 0 \quad \forall j = 1, 2, ..., N_y \tag{116}$$
$$@i = N_x, u'_{i,j} = 0 \rightarrow [A^\phi]_{2,5} = -\Delta y = 0 \quad \forall j = 1, 2, ..., N_y \tag{117}$$
$$@j = 1, v'_{i,j-1} = 0 \rightarrow [A^\phi]_{3,5} = \Delta x = 0 \quad \forall i = 1, 2, ..., N_x \tag{118}$$
$$@j = N_y, v'_{i,j} = 0 \rightarrow [A^\phi]_{4,5} = -\Delta x = 0 \quad \forall i = 1, 2, ..., N_x \tag{119}$$

No-slip condition The idea behind the no-slip condition for the tangential velocity component is to impose the velocity field in the ghost cells so that the interpolation with the velocity field in the physical domain gives the correct value at the boundary.
As an example, we can consider the cell $(1, N_y)$ as in Figure 12.
We know that we must impose $u_{1,N_y} = U_{lid}$. Given that the condition can't be imposed directly to the correction term as for the no-penetration condition, here we have to solve the system and then impose the condition over the ghost cells velocity based on the current solution of the velocity field.
In particular, we can think of the wall velocity as the average of the velocity in the ghost cell and the velocity in the physical domain. This implies to impose the following condition for the cell $(1, N_y)$:

$$U_{lid} = \frac{1}{2}(u_{1,N_y} + u_{1,N_y+1}) \rightarrow u_{1,N_y+1} = 2U_{lid} - u_{1,N_y} \tag{120}$$

More in general, given the physical domain of size $N_x \times N_y$, we can impose the no-slip condition for the velocity field by setting:

$$@i = 1, v_{0,j} = -v_{1,j} \quad \forall j = 1, 2, ..., N_y \tag{121}$$
$$@i = N_x, v_{N_x+1,j} = -v_{N_x,j} \quad \forall j = 1, 2, ..., N_y \tag{122}$$
$$@j = 1, u_{i,0} = -u_{i,1} \quad \forall i = 1, 2, ..., N_x \tag{123}$$
$$@j = N_y, u_{i,N_y+1} = 2U_{lid} - u_{i,N_y} \quad \forall i = 1, 2, ..., N_x \tag{124}$$

5.7 Convergence criterion

Being the SCGS method an iterative method, we need to define a criterion to stop the iterations, and to consider the solution as converged.
We could have adopted many criteria, but probably the most intuitive one is to monitor the residuals of the equations over the entire domain for each iteration, and to stop the cycle when the maximum of the residual is below a certain threshold.
So far we have defined 5 residuals, one for each equation, and we can define other 3 residuals derived from the previous ones, as follows:

$$\begin{aligned} \text{Continuity residual} : R^p &= |R^c| \\ \text{Momentum residual u} : R^u &= \frac{\left(|R^u_{i-1}| + |R^u_i|\right)}{2} \\ \text{Momentum residual v} : R^v &= \frac{\left(|R^v_{j-1}| + |R^v_j|\right)}{2} \end{aligned} \tag{125}$$

By doing so, our convergence criterion will be:

$$\max\left(R^p, R^u, R^v\right) < \varepsilon \tag{126}$$

Where ε is the threshold and may vary between 10^{-3} and 10^{-6}, depending on the problem and the computational resources available.

6 Code implementation

When it comes to the implementation of the SCGS algorithm, lots of details need to be taken into account, such as the data structures, the memory allocation, the data exchange between the different modules of the code, and so on.
The main blocks of the SCGS algorithm can be found in the appendix section 7.5 of this document.
The complete codebase is available on GitHub at `https://github.com/Bocchio01/CFD_Simulation_Engine`, along with its technical documentation.

7 Results

The SCGS algorithm has been implemented in `C` and tested on a simple 2D lid-driven cavity flow problem.
In the following we are going to compare the results obtained using our code with the ones reported in Ghia et al. [1].
Note: we will see that our code has some issues related to convergence and stability. This affect the under-relaxation factors required to stabilize the solution that have to be decreased to very low values (e.g. $\alpha_u = 0.01$ and $\alpha_v = 0.01$ or less). This is not a good practice and it's not a solution to the problem. We are currently investigating the issue and we will update the codebase as soon as possible.

7.1 Ghia's exact solution

The exact solution for the lid-driven cavity flow at $Re = 1000$ is reported in Ghia et al. [1], and it is used as a benchmark to validate the results obtained using our code.
In Ghia's solution, a uniform grid of 129×129 points is used, and the Reynolds number is imposed as follows:

$$Re = \begin{cases} 100 \\ 400 \\ 1000 \end{cases} \rightarrow \nu = \frac{U_{lid} L_{domain}}{Re} = \begin{cases} 0.0100 \\ 0.0025 \\ 0.0010 \end{cases} \tag{127}$$

Where U_{lid} is the velocity of the lid and L_{domain} is the characteristic dimension of the domain.
For reference, in Figure 13 and Figure 14 we report the solution from Ghia et al. [1] for the lid-driven cavity flow varying the Reynolds number.

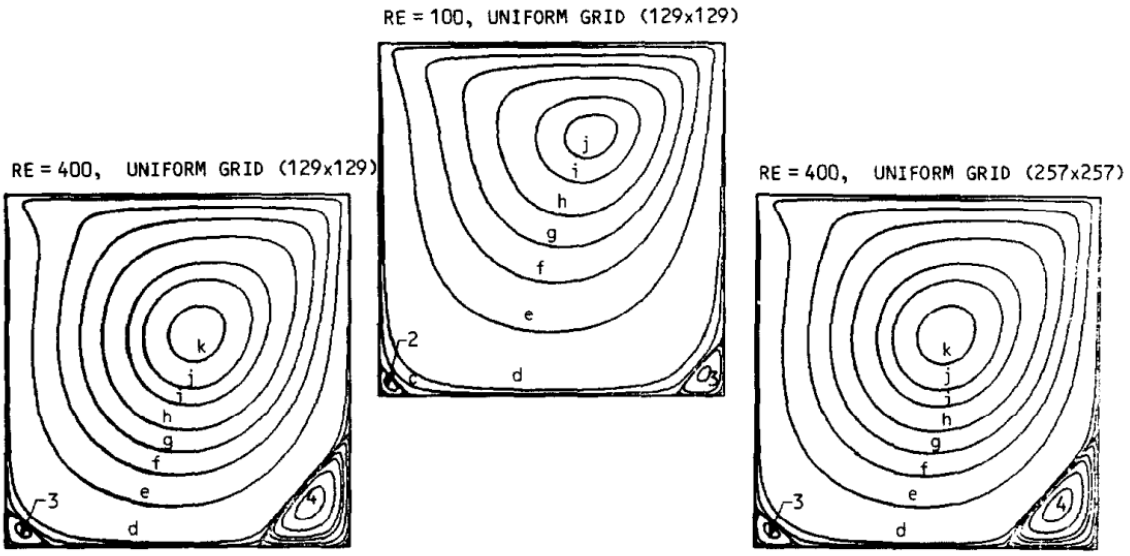

Figure 13: Ghia's solutions for the lid-driven cavity flow at different Reynolds numbers.

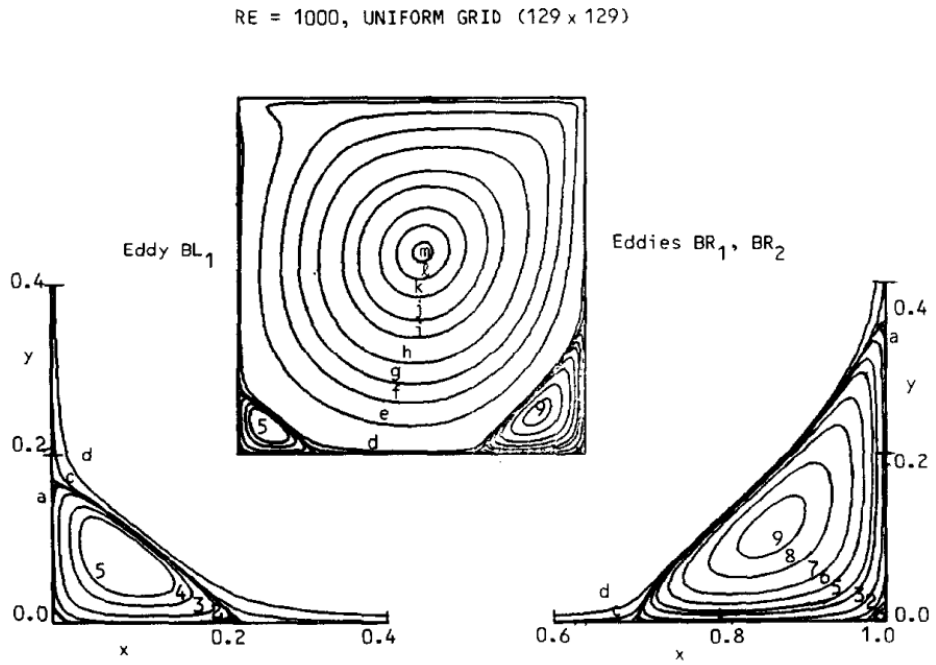

FIG. 3. Streamline pattern for primary, secondary, and additional corner vortices.

Figure 14: Highlights of the eddy structures at the bottom corners of the cavity at $Re = 1000$.

We can now proceed with the comparison between our results and Ghia's solution.

7.2 Comparison with Ghia's solution

Instead of just comparing the results by looking at the streamlines of the vector field which can be misleading, we are going to compare the velocity field at different points of the domain. In particular, in Ghia's paper [1] are reported the velocity profiles of $u(y)@x/L_x = 0.5$ and $v(x)@y/L_y = 0.5$ for different Reynolds numbers. By using `MATLAB`, we can easily plot those profiles and compare them with the ones obtained using our code. **Note:** to distinguish the different solutions generated by our code, we are going to use the following naming convention:

$$\#\#\text{-Nx-Ny-Re-ConvectionScheme-DiffusionScheme-}\alpha_u\text{-}\alpha_v \tag{128}$$

Where:

- ## is the ID of the solution;
- Nx is the number of points in the x direction;
- Ny is the number of points in the y direction;
- Re is the Reynolds number;
- $ConvectionScheme$ is the convection scheme used;
- $DiffusionScheme$ is the diffusion scheme used;
- α_u is the under-relaxation factor for the u velocity (e.g. 08 for 0.8, 005 for 0.05, etc.);
- α_v is the under-relaxation factor for the v velocity (e.g. 08 for 0.8, 005 for 0.05, etc.).

By benchmarking our code with different parameters, we have obtained the following velocity profiles.

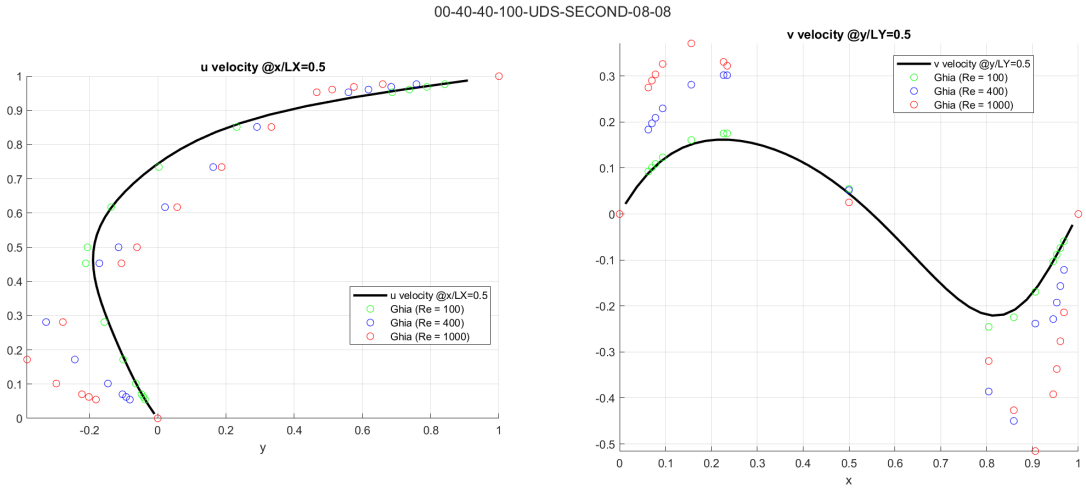

Figure 15: Velocity comparison for solution 00-40-40-100-UDS-SECOND-08-08.

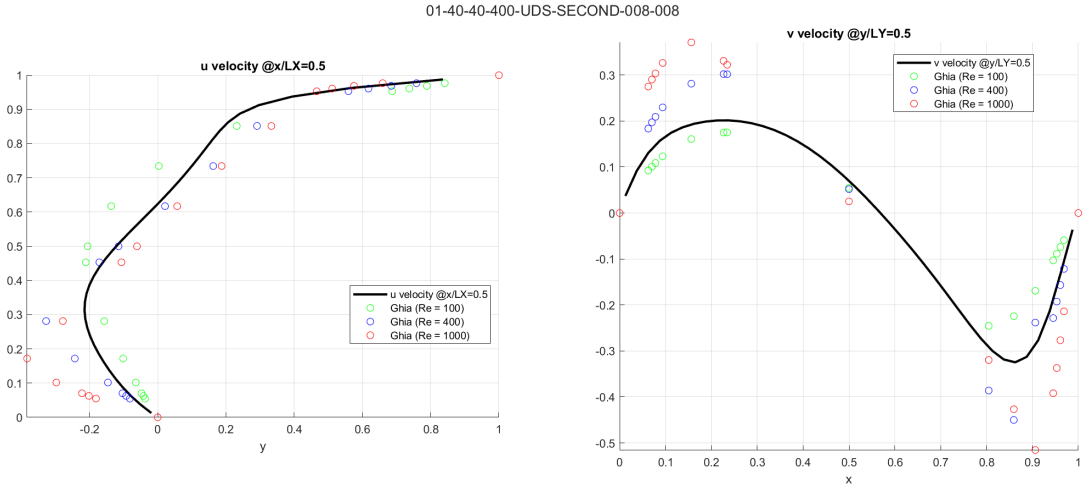

Figure 16: Velocity comparison for solution 01-40-40-400-UDS-SECOND-008-008.

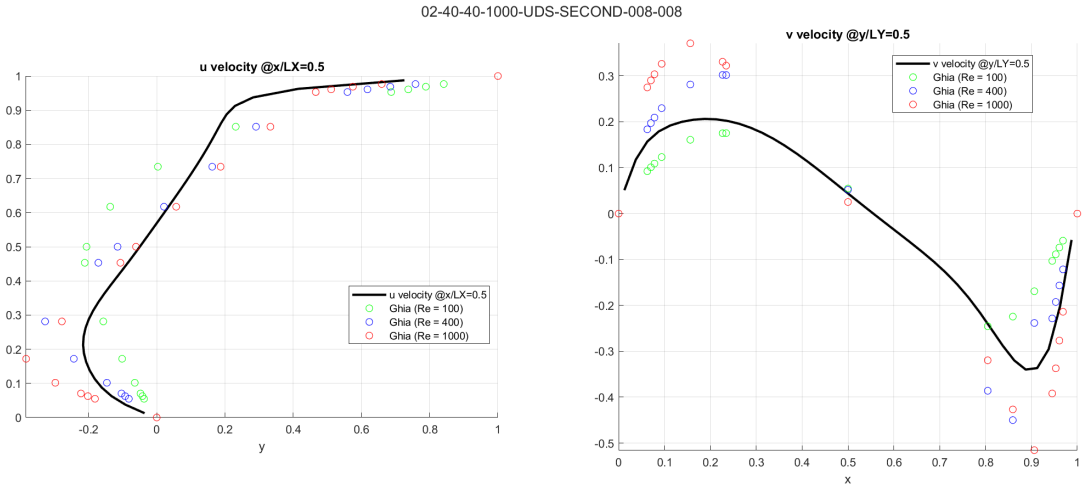

Figure 17: Velocity comparison for solution 02-40-40-1000-UDS-SECOND-008-008.

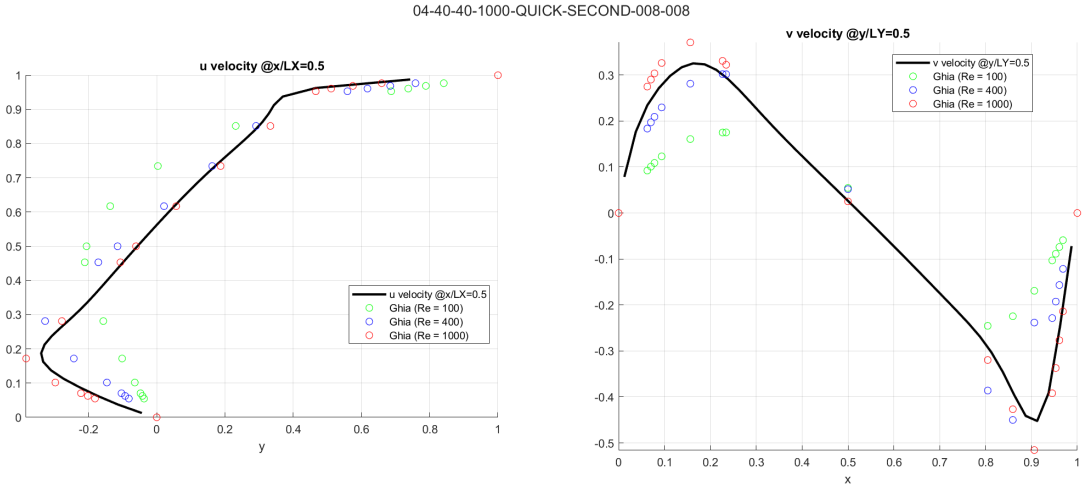

Figure 18: Velocity comparison for solution 04-40-40-1000-QUICK-SECOND-008-008.

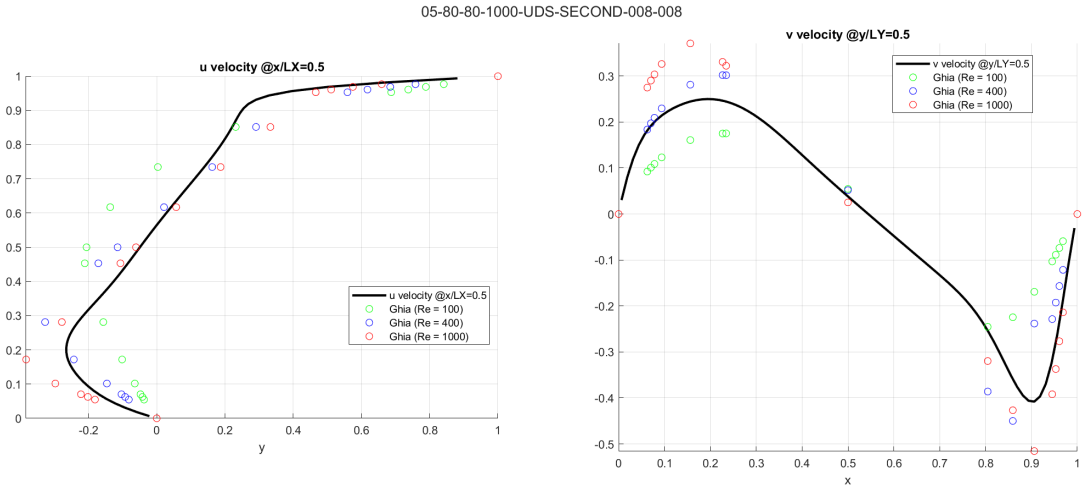

Figure 19: Velocity comparison for solution 05-80-80-1000-UDS-SECOND-008-008.

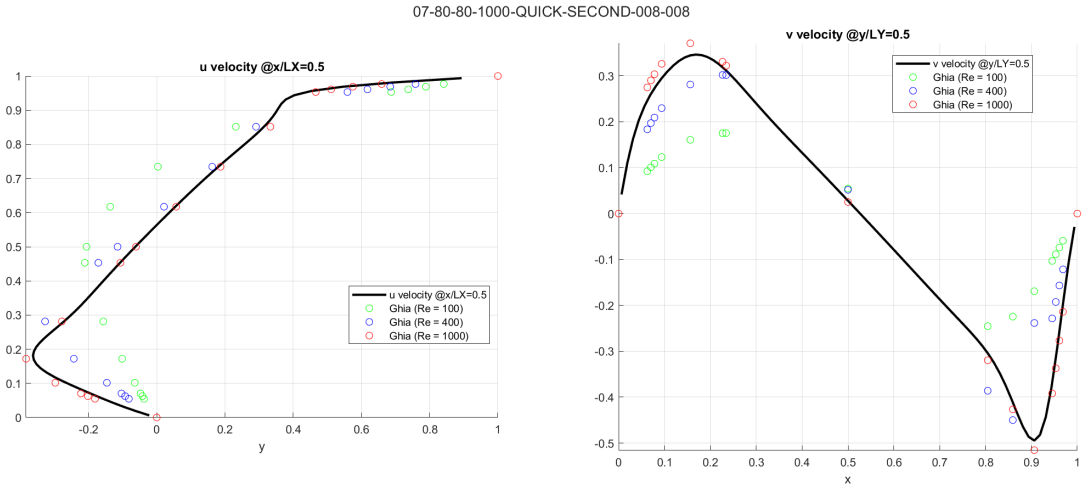

Figure 20: Velocity comparison for solution 07-80-80-1000-QUICK-SECOND-008-008.

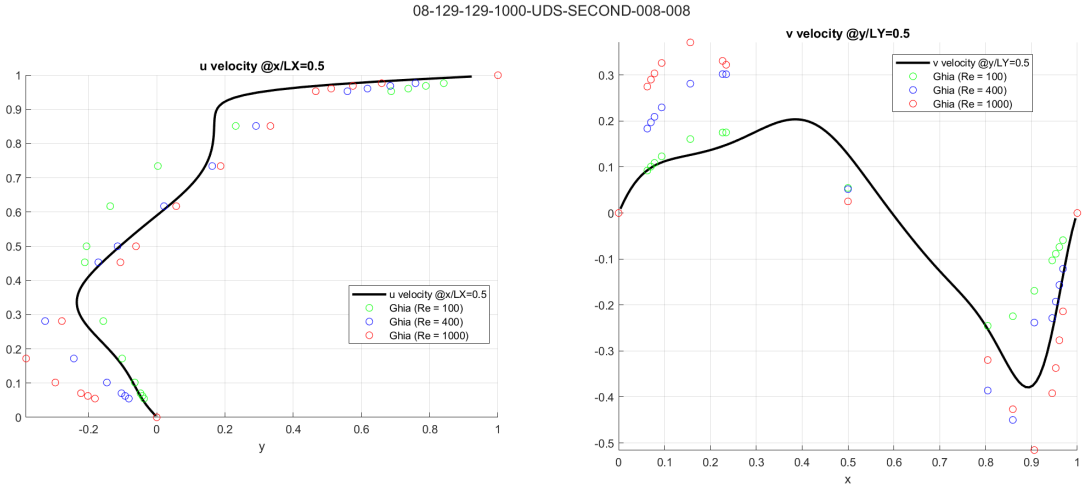

Figure 21: Velocity comparison for solution 08-129-129-1000-UDS-SECOND-008-008.

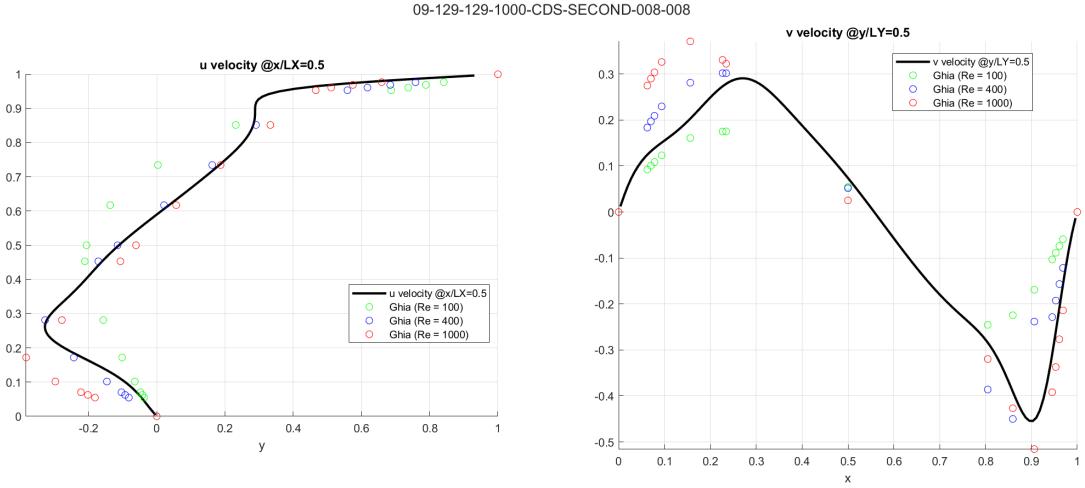

Figure 22: Velocity comparison for solution 09-129-129-1000-CDS-SECOND-008-008.

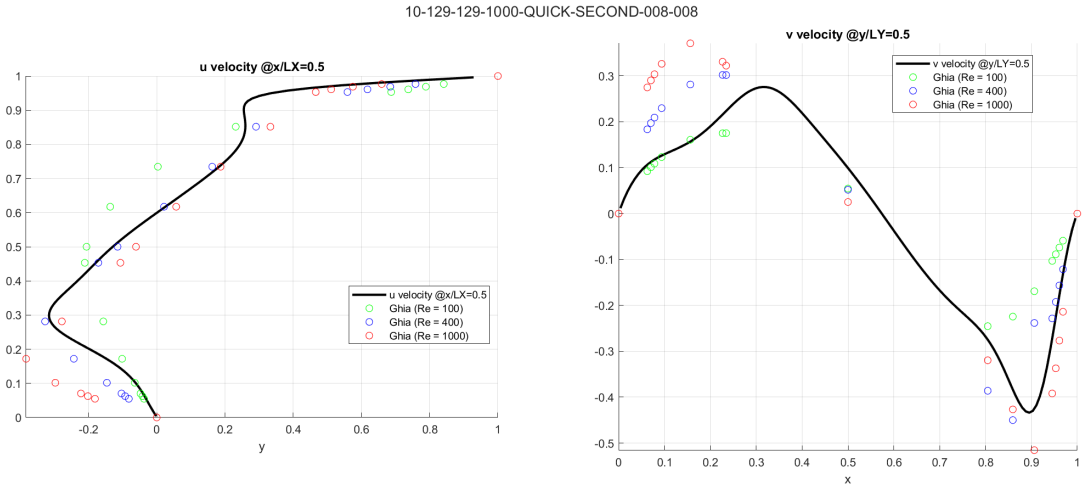

Figure 23: Velocity comparison for solution 10-129-129-1000-QUICK-SECOND-008-008.

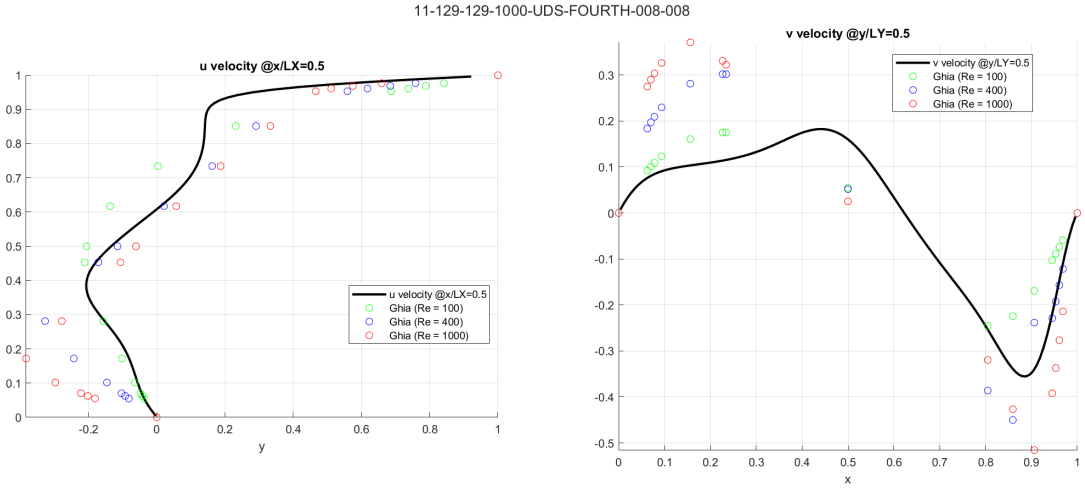

Figure 24: Velocity comparison for solution 11-129-129-1000-UDS-FOURTH-008-008.

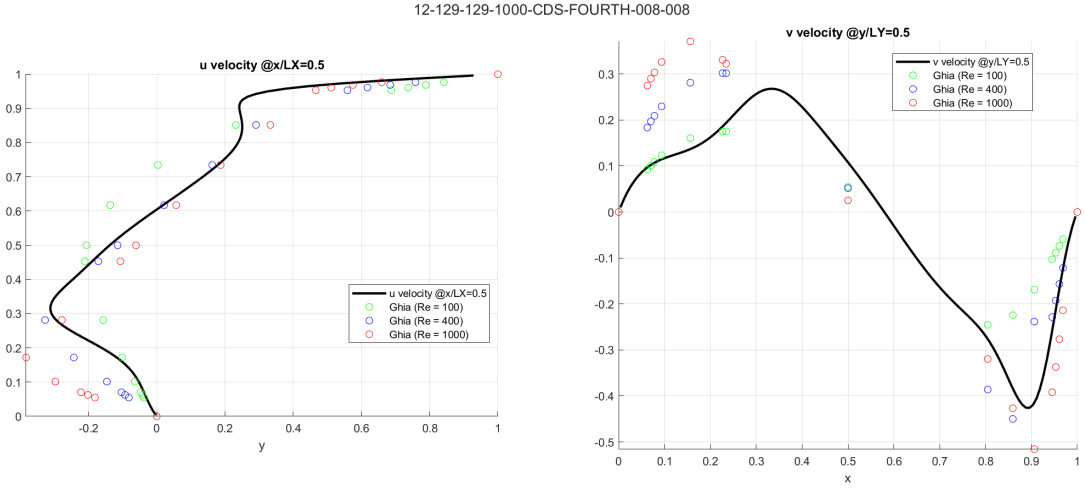

Figure 25: Velocity comparison for solution 12-129-129-1000-CDS-FOURTH-008-008.

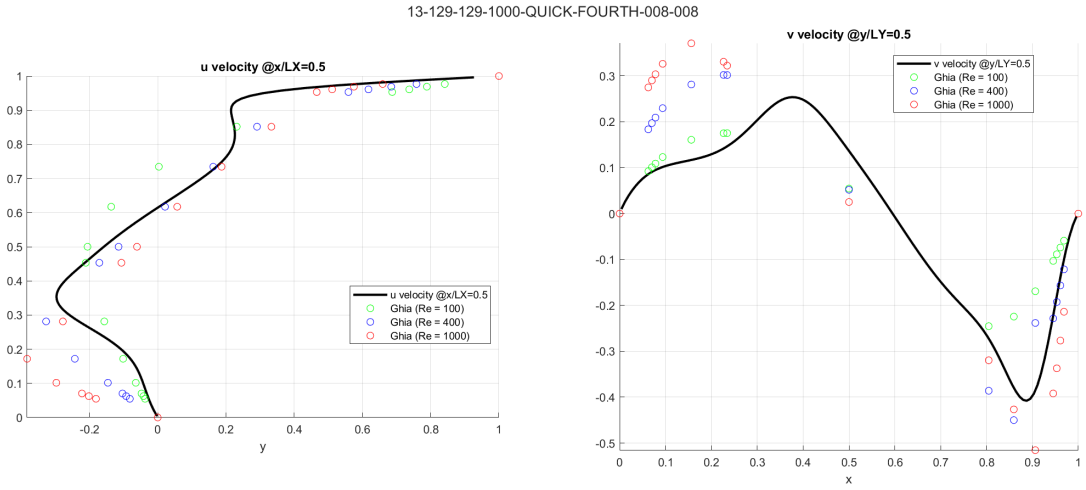

Figure 26: Velocity comparison for solution 13-129-129-1000-QUICK-FOURTH-008-008.

7.3 07-80-80-1000-QUICK-SECOND-008-008

Considering just only the results obtained for $Re = 1000$, it is easy to see how `07-80-80-1000-QUICK-SECOND-008-008` is the one that better approximates the exact solution reported by Ghia et al. [1].
In the following figure we report all the data results obtained for the simulation runned with the following parameters:

Variable	Value
Re	1000
Nx	80
Ny	80
α_u	0.08
α_v	0.08
Convection scheme	QUICK
Diffusion scheme	SECOND

Table 2: Parameters for simulation 07.

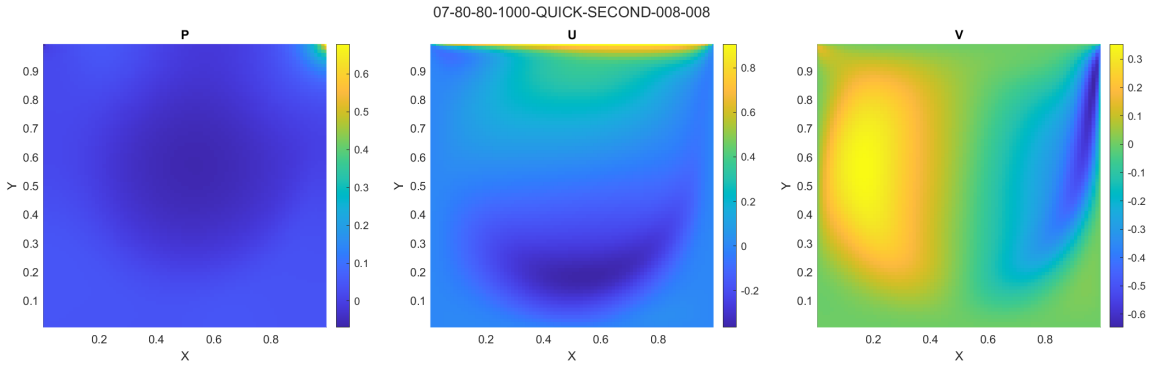

Figure 27: Final state of the simulation 07.

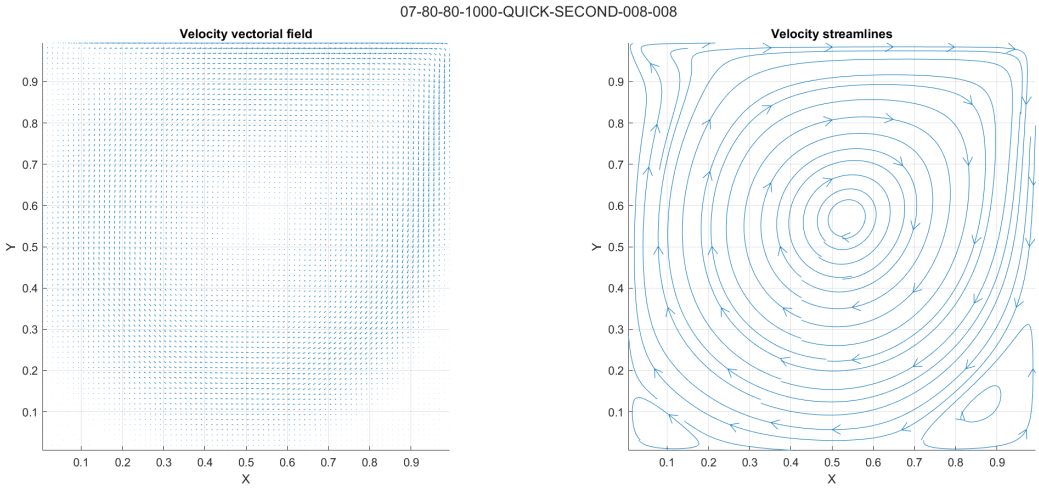

Figure 28: Streamlines of the simulation 07.

7.4 Convergence analysis

In this section we report the convergence analysis of the SCGS algorithm.

Residual In Figure 29 we report the residual of the algorithm for the different simulations runned.

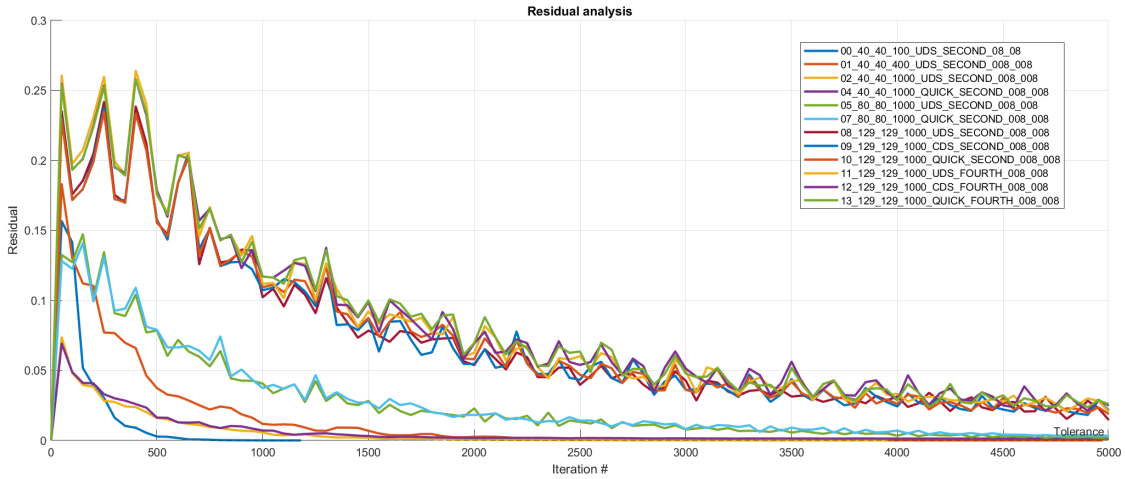

Figure 29: Residuals of the SCGS algorithm.

CPU time In Figure 30 we report the CPU time of the algorithm for the different simulations runned.

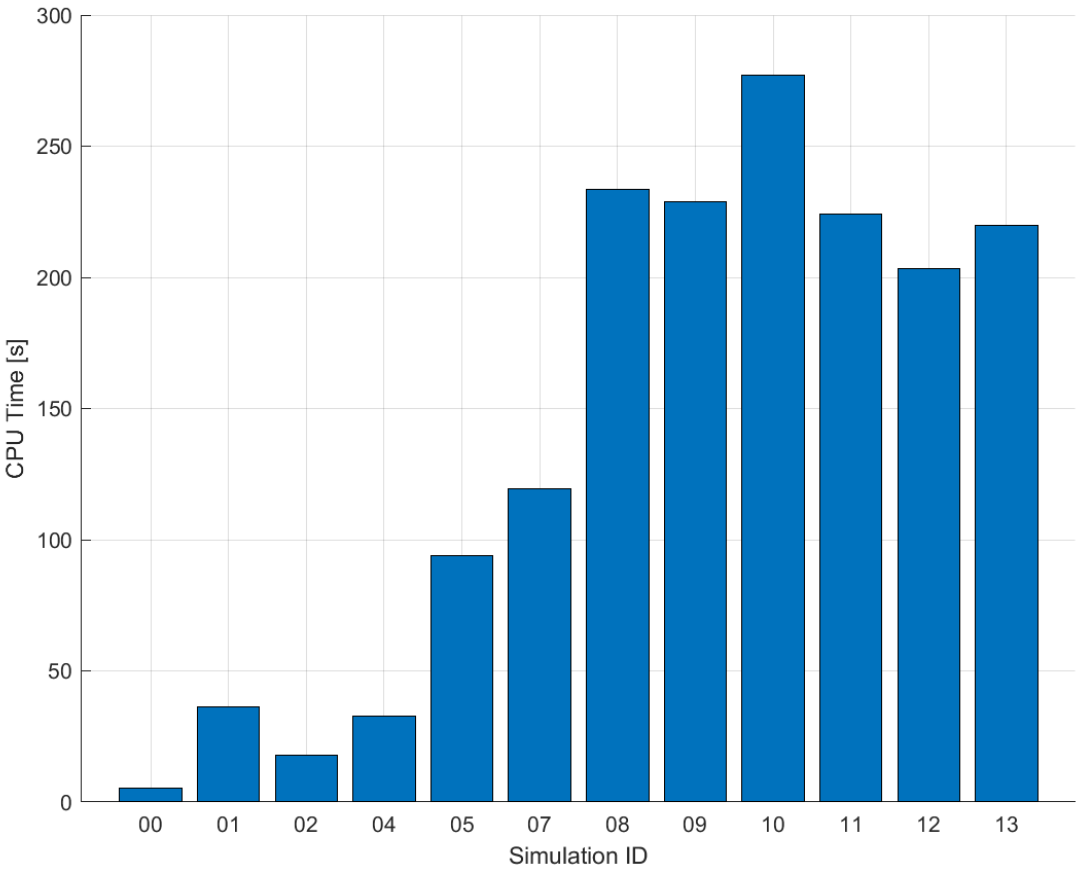

Figure 30: CPU time of the SCGS algorithm.

7.5 Final remarks

As we have seen, our code has some issues related to convergence and stability.
In particular, in order to have a convergent solution, we have to decrease the under-relaxation factors to very low values (e.g. $\alpha_u = 0.01$ and $\alpha_v = 0.01$ or less).
By debugging through the code, we can clearly see that this is due to the low or absent diagonal dominance of the coefficient matrix (Vanka's matrix).
Moreover, by deepening the problem, we have found that the convergence of the algorithm is ensured only if the following condition is satisfied:

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}| \quad \forall i = 1, 2, 3, 4 \tag{129}$$

Where a_{ij} are the elements of the coefficient matrix.
In the SCGS algorithm the coefficient matrix is in the form of Equation 102, so we can states that the diagonal dominance is ensured only if the following condition is satisfied:

$$|a_{ii}| \geq |a_{5i}| \quad \forall i = 1, 2, 3, 4 \tag{130}$$

For how the system has been constructed, we know that:

$$a_{ii} = f(\nu, \alpha_u, \alpha_v) \tag{131}$$
$$a_{5i} = f(Nx, Ny) \tag{132}$$

Where f indicate a generic function.
Based on the equations and derivations done in the previous sections, we have that diagonal dominance is compromised when:

- The grid is not fine enough (i.e. Nx and Ny are low) $\Rightarrow a_{5i}$ = Cell sixe is high

- The under-relaxation factors are high $\Rightarrow a_{ii} = A_p^{\phi}/\alpha_{\phi}$ is low
- The viscosity is low $\Rightarrow$ Diffusion term is low $\Rightarrow a_{ii}$ is low

When Equation 129 is not satisfied once, then the algorithm will diverge imposing too high corrections to the state variables, causing the Convective terms to become too high at the next iteration (or neighboring cells).

Note: we know this should not append, but as of today we still haven't found the problem in our code.

References

[1] Urmila Ghia, Karman Ghia, and C. T. Shin. High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method. *Journal of Computational Physics*, 48:387–411, 1982.

[2] S.P Vanka. Block-implicit multigrid solution of navier-stokes equations in primitive variables. *Journal of Computational Physics*, 65(1):138–158, 1986.

A Mathematica code

Here follows the `Mathematica` notebook used for symbolic analysis of the discretized schemes.

```
(*Tommaso Bocchietti*)
(*ME663 - Computational Fluid Dynamics*)
(*Ap coefficient calculations*)
Clear["Global`*"];

(*Define the Uvelocities and Vvelocities lists*)
Hlabels={"WW","W","P","E","EE"};
Vlabels={"SS","S","P","N","NN"};

(*Initialize empty matrices*)
\[Phi]=ConstantArray[0,{Length[Hlabels],Length[Vlabels]}];

(*Populate the matrices using for loops*)
For[i=1,i<=Length[Hlabels],i++,
For[j=1,j<=Length[Vlabels],j++,
\[Phi][[i,j]]=Symbol["\[Phi]"<>Hlabels[[i]]<>Vlabels[[j]]];
];
];

(*Extrapolate velocity at (i,j)=(i,0) and (i,j)=(0,j)*)
velWWtoEE=\[Phi][[All,Ceiling[Length[Vlabels]/2]]];
velSStoNN=\[Phi][[Ceiling[Length[Hlabels]/2]]];

(*Fix lenght as adapting*)
ptsWWtoEE=Table[{i*\[CapitalDelta]x,velWWtoEE[[i+3]]},{i,-2,2}];
ptsSStoNN = Table[{i*\[CapitalDelta]y,velSStoNN[[i+3]]},{i,-2,2}];

(*phi is an alias to access the element of \[Phi] more easly*)
phi[i_,j_]:= \[Phi][[i-3,j-3]];
(*phi[i_,j_]:= 0;*)

(*Define a list of labels for final Ap coefficients*)
ApLabels="Ap"<>ToString[#]&/@Flatten[\[Phi]];

(*General definition of convection and diffusion discretized equations*)
Convection[\[Phi]e_, \[Phi]w_, \[Phi]n_, \[Phi]s_] :=(
   (\[Phi]e[[1]]*Max[Fe\[Phi], 0] + \[Phi]e[[2]]*Min[Fe\[Phi], 0])-
(\[Phi]w[[1]]*Max[Fw\[Phi], 0] + \[Phi]w[[2]]*Min[Fw\[Phi], 0]) +
(\[Phi]n[[1]]*Max[Fn\[Phi], 0] + \[Phi]n[[2]]*Min[Fn\[Phi], 0])-
(\[Phi]s[[1]]*Max[Fs\[Phi], 0] + \[Phi]s[[2]]*Min[Fs\[Phi], 0])
);

ConvectionCDS[\[Phi]e_, \[Phi]w_, \[Phi]n_, \[Phi]s_] :=(
   (\[Phi]e*Fe\[Phi] )-(\[Phi]w*Fw\[Phi]) +
(\[Phi]n*Fn\[Phi])- (\[Phi]s*Fs\[Phi])
);

Diffusion[dd\[Phi]x_ , dd\[Phi]y_]:=\[Nu]*(dd\[Phi]x+dd\[Phi]y)*\[CapitalDelta]x*\[
    CapitalDelta]y;

(*Convection schemes*)
(*UDS Scheme for convection*)
UDS\[Phi]e = {phi[+0,+0], phi[+1,+0]};
UDS\[Phi]w = {phi[-1,+0], phi[+0,+0]};
UDS\[Phi]n = {phi[+0,+0], phi[+0,+1]};
UDS\[Phi]s = {phi[+0,-1], phi[+0,+0]};

```

```
(*USD computing and sign correction (change sign to all elements except the ApPP one)*)
UDSAp=Table[D[Convection[UDS\[Phi]e, UDS\[Phi]w, UDS\[Phi]n, UDS\[Phi]s],\[Phi]Vel],{\[Phi]
    Vel,Flatten[\[Phi]]}];
UDSAp=MapIndexed[If[First[#2]!=Ceiling[Length[UDSAp]/2],-#1,#1]&,UDSAp];

(*UDS Scheme for convection*)
CDS\[Phi]e = 1/2*(phi[+0,+0]+phi[+1,+0]);
CDS\[Phi]w = 1/2*(phi[-1,+0]+phi[+0,+0]);
CDS\[Phi]n = 1/2*(phi[+0,+0]+phi[+0,+1]);
CDS\[Phi]s = 1/2*(phi[+0,-1]+phi[+0,+0]);

(*CDS computing and sign correction (change sign to all elements except the ApPP one)*)
CDSAp=Table[D[ConvectionCDS[CDS\[Phi]e, CDS\[Phi]w, CDS\[Phi]n, CDS\[Phi]s],\[Phi]Vel],{\[
    Phi]Vel,Flatten[\[Phi]]}];
CDSAp=MapIndexed[If[First[#2]!=Ceiling[Length[CDSAp]/2],-#1,#1]&,CDSAp];

(*QUICK Scheme for convection*)
(*Partition[points[[1]],3,1] -> Sliding windows of 3 points amplitude*)
(*InterpolatingPolynomial[#,x]&/@ -> Apply interpolation over each set of points*)
QUICK\[Phi]e=InterpolatingPolynomial[#,x]&/@Partition[ptsWWtoEE,3,1][[2;;3]]/. x->\[
    CapitalDelta]x/2;
QUICK\[Phi]w=InterpolatingPolynomial[#,x]&/@Partition[ptsWWtoEE,3,1][[1;;2]]/. x->-\[
    CapitalDelta]x/2;
QUICK\[Phi]n=InterpolatingPolynomial[#,y]&/@Partition[ptsSStoNN,3,1][[2;;3]]/. y->\[
    CapitalDelta]y/2;
QUICK\[Phi]s=InterpolatingPolynomial[#,y]&/@Partition[ptsSStoNN,3,1][[1;;2]]/. y->-\[
    CapitalDelta]y/2;

(*QUICK computing and sign correction (change sign to all elements except the ApPP one)*)
QUICKAp=Table[D[Convection[QUICK\[Phi]e, QUICK\[Phi]w, QUICK\[Phi]n, QUICK\[Phi]s],\[Phi]Vel
    ],{\[Phi]Vel,Flatten[\[Phi]]}];
QUICKAp=MapIndexed[If[First[#2]!=Ceiling[Length[QUICKAp]/2],-#1,#1]&,QUICKAp];

(*Diffusion schemes*)
(*Second order scheme for diffusion*)
dd\[Phi]x2=D[InterpolatingPolynomial[ptsWWtoEE[[2;;4]],x], {x,2}]/. x->0;
dd\[Phi]y2=D[InterpolatingPolynomial[ptsSStoNN[[2;;4]],y], {y,2}]/. y->0;

(*DIF2 computing and sign correction (change sign to the ApPP element only)*)
DIF2Ap=Table[D[Diffusion[dd\[Phi]x2,dd\[Phi]y2],\[Phi]Vel],{\[Phi]Vel,Flatten[\[Phi]]}];
DIF2Ap=MapIndexed[If[First[#2]==Ceiling[Length[DIF2Ap]/2],-#1,#1]&,DIF2Ap];

(*Fourth order scheme for diffusion*)
dd\[Phi]x4=D[InterpolatingPolynomial[ptsWWtoEE[[1;;5]],x], {x,2}]/. x->0;
dd\[Phi]y4=D[InterpolatingPolynomial[ptsSStoNN[[1;;5]],y], {y,2}]/. y->0;

(*DIF4 computing and sign correction (change sign to the ApPP element only)*)
DIF4Ap=Table[D[Diffusion[dd\[Phi]x4,dd\[Phi]y4],\[Phi]Vel],{\[Phi]Vel,Flatten[\[Phi]]}];
DIF4Ap=MapIndexed[If[First[#2]==Ceiling[Length[DIF4Ap]/2],-#1,#1]&,DIF4Ap];

(*Final coefficients table*)
TableForm[Transpose[{ApLabels,UDSAp,CDSAp, QUICKAp, DIF2Ap, DIF4Ap}], TableHeadings->{None,{
    "Ap\[Phi]HV","Convection UDS","Convection CDS", "Convection QUICK", "Diffusion 2", "
    DIffusion 4"}}]
```

Listing 1: Mathematica notebook used for symbolic analysis of the discretized schemes.

B C code

The complete codebase is available on GitHub at `https://github.com/Bocchio01/CFD_Simulation_Engine`, along with its technical documentation.

```c
#include "SCGS.h"

typedef struct CFD_t CFD_t;

#include <stdlib.h>
#include <math.h>
#include "methods.h"
#include "../../CFD.h"
#include "../schemes/schemes.h"
#include "../../utils/cALGEBRA/dMAT.h"
#include "../../utils/cALGEBRA/cVEC.h"
#include "../../utils/cLOG/cLOG.h"

void CFD_SCGS(CFD_t *cfd)
{
    SCGS_t *scgs;

    scgs = CFD_SCGS_Allocate();

    for (cfd->engine->method->iteractions = 0;
         cfd->engine->method->iteractions < cfd->engine->method->maxIter;
         cfd->engine->method->iteractions++)
    {
        CFD_SCGS_Reset(scgs);

        for (cfd->engine->method->index->j = cfd->engine->mesh->n_ghosts;
             cfd->engine->method->index->j < cfd->engine->mesh->nodes->Ny + cfd->engine->mesh->n_ghosts;
             cfd->engine->method->index->j++)
        {
            for (cfd->engine->method->index->i = cfd->engine->mesh->n_ghosts;
                 cfd->engine->method->index->i < cfd->engine->mesh->nodes->Nx + cfd->engine->mesh->n_ghosts;
                 cfd->engine->method->index->i++)
            {

                CFD_SCGS_System_Compose(cfd, scgs);

                CFD_SCGS_BC_NoSlip_Normal(cfd, scgs);

                CFD_SCGS_System_Solve(scgs);

                CFD_SCGS_Apply_Correction(cfd, scgs);

                CFD_SCGS_Update_Residual(scgs);
            }
        }

        CFD_SCGS_BC_NoSlip_Tangetial(cfd);

        cfd->engine->method->residual->data[cfd->engine->method->iteractions] = fmax(fmax(scgs->residual->u, scgs->residual->v), scgs->residual->p);

        if (isnan(cfd->engine->method->residual->data[cfd->engine->method->iteractions]) ||
            isinf(cfd->engine->method->residual->data[cfd->engine->method->iteractions]))
        {
```

```
54              log_fatal("Algorithm diverged at iteration %d", cfd->engine->method->iteractions
    );
55              CFD_SCGS_Free(scgs);
56              exit(EXIT_FAILURE);
57          }
58
59          if (cfd->engine->method->iteractions % 100 == 0 ||
60              cfd->engine->method->residual->data[cfd->engine->method->iteractions] < cfd->
    engine->method->tolerance)
61          {
62              log_info("\nIteration:\t%d\nResidual:\t%.10f",
63                       cfd->engine->method->iteractions,
64                       cfd->engine->method->residual->data[cfd->engine->method->iteractions]);
65          }
66
67          if (cfd->engine->method->residual->data[cfd->engine->method->iteractions] < cfd->
    engine->method->tolerance &&
68              cfd->engine->method->iteractions > 1)
69          {
70              log_info("Algorithm converged in %d iterations", cfd->engine->method->
    iteractions);
71              CFD_SCGS_Free(scgs);
72              break;
73          }
74      }
75  }
76
77  SCGS_t *CFD_SCGS_Allocate()
78  {
79      SCGS_t *scgs;
80
81      scgs = (SCGS_t *)malloc(sizeof(SCGS_t));
82
83      if (scgs != NULL)
84      {
85          scgs->vanka = (Vanka_t *)malloc(sizeof(Vanka_t));
86          scgs->residual = (residual_t *)malloc(sizeof(residual_t));
87          scgs->A_coefficients = VEC_Init(EENN + 1);
88
89          if (scgs->vanka != NULL &&
90              scgs->residual != NULL &&
91              scgs->A_coefficients != NULL)
92          {
93              scgs->vanka->A = MAT_Init(5, 5);
94              scgs->vanka->R = VEC_Init(5);
95              scgs->vanka->x = VEC_Init(5);
96
97              if (scgs->vanka->A != NULL &&
98                  scgs->vanka->R != NULL &&
99                  scgs->vanka->x != NULL)
100             {
101                 return scgs;
102             }
103         }
104     }
105
106     log_fatal("Failed to allocate memory for SCGS method");
107     exit(EXIT_FAILURE);
108 }
109
110 void CFD_SCGS_Free(SCGS_t *scgs)
```

```
111 {
112     MAT_Free(scgs->vanka->A);
113     VEC_Free(scgs->vanka->R);
114     VEC_Free(scgs->vanka->x);
115     free(scgs->residual);
116     free(scgs->vanka);
117     free(scgs->A_coefficients);
118     free(scgs);
119 }
120
121 void CFD_SCGS_Reset(SCGS_t *scgs)
122 {
123     scgs->residual->u = 0.0;
124     scgs->residual->v = 0.0;
125     scgs->residual->p = 0.0;
126 }
127
128 void CFD_SCGS_System_Compose(CFD_t *cfd, SCGS_t *scgs)
129 {
130     double Ap[4];
131
132     uint16_t i = cfd->engine->method->index->i;
133     uint16_t j = cfd->engine->method->index->j;
134
135     typedef struct
136     {
137         int i;
138         int j;
139         phi_t phi;
140     } position;
141
142     position positions[] = {
143         {-1, +0, u},
144         {+0, +0, u},
145         {+0, -1, v},
146         {+0, +0, v}};
147
148     for (int el = 0; el < 4; el++)
149     {
150         scgs->vanka->R->data[el] = 0.0;
151
152         if (i == cfd->engine->mesh->n_ghosts ||
153             j == cfd->engine->mesh->n_ghosts ||
154             i == cfd->engine->mesh->nodes->Nx + cfd->engine->mesh->n_ghosts - 1 ||
155             j == cfd->engine->mesh->nodes->Ny + cfd->engine->mesh->n_ghosts - 1)
156         {
157             CFD_Scheme_Convection_UDS(cfd, i + positions[el].i, j + positions[el].j,
    positions[el].phi);
158             CFD_Scheme_Diffusion_SECOND(cfd);
159         }
160         else
161         {
162             cfd->engine->schemes->convection->callable(cfd, i + positions[el].i, j +
    positions[el].j, positions[el].phi);
163             cfd->engine->schemes->diffusion->callable(cfd);
164         }
165
166         for (uint16_t k = 0; k < scgs->A_coefficients->length; k++)
167         {
168             double phi = CFD_Get_State(cfd, positions[el].phi, i + positions[el].i + ((k /
    5) % 5) - 2, j + positions[el].j + (k % 5) - 2);
```

```
169             scgs->A_coefficients->data[k] = cfd->engine->schemes->convection->coefficients->
    data[k] + cfd->engine->schemes->diffusion->coefficients->data[k];
170
171             if (k == PP)
172             {
173                 Ap[el] = scgs->A_coefficients->data[k];
174                 scgs->vanka->R->data[el] -= (scgs->A_coefficients->data[k]) * phi;
175             }
176             else
177             {
178                 scgs->vanka->R->data[el] += (scgs->A_coefficients->data[k]) * phi;
179             }
180         }
181     }
182
183     scgs->vanka->R->data[0] += (CFD_Get_State(cfd, p, i - 1, j + 0) - CFD_Get_State(cfd, p,
    i + 0, j + 0)) * cfd->engine->mesh->element->size->dy;
184     scgs->vanka->R->data[1] += (CFD_Get_State(cfd, p, i + 0, j + 0) - CFD_Get_State(cfd, p,
    i + 1, j + 0)) * cfd->engine->mesh->element->size->dy;
185     scgs->vanka->R->data[2] += (CFD_Get_State(cfd, p, i + 0, j - 1) - CFD_Get_State(cfd, p,
    i + 0, j + 0)) * cfd->engine->mesh->element->size->dx;
186     scgs->vanka->R->data[3] += (CFD_Get_State(cfd, p, i + 0, j + 0) - CFD_Get_State(cfd, p,
    i + 0, j + 1)) * cfd->engine->mesh->element->size->dx;
187     scgs->vanka->R->data[4] = -((CFD_Get_State(cfd, u, i + 0, j + 0) - CFD_Get_State(cfd, u,
     i - 1, j + 0)) * cfd->engine->mesh->element->size->dy +
188                                 (CFD_Get_State(cfd, v, i + 0, j + 0) - CFD_Get_State(cfd, v,
     i + 0, j - 1)) * cfd->engine->mesh->element->size->dx);
189
190     scgs->vanka->A->data[0][0] = Ap[0] / cfd->engine->method->under_relaxation_factors->u;
191     scgs->vanka->A->data[1][1] = Ap[1] / cfd->engine->method->under_relaxation_factors->u;
192     scgs->vanka->A->data[2][2] = Ap[2] / cfd->engine->method->under_relaxation_factors->v;
193     scgs->vanka->A->data[3][3] = Ap[3] / cfd->engine->method->under_relaxation_factors->v;
194
195     scgs->vanka->A->data[4][0] = -1.0 * cfd->engine->mesh->element->size->dy;
196     scgs->vanka->A->data[4][1] = +1.0 * cfd->engine->mesh->element->size->dy;
197     scgs->vanka->A->data[4][2] = -1.0 * cfd->engine->mesh->element->size->dx;
198     scgs->vanka->A->data[4][3] = +1.0 * cfd->engine->mesh->element->size->dx;
199
200     scgs->vanka->A->data[0][4] = +1.0 * cfd->engine->mesh->element->size->dy;
201     scgs->vanka->A->data[1][4] = -1.0 * cfd->engine->mesh->element->size->dy;
202     scgs->vanka->A->data[2][4] = +1.0 * cfd->engine->mesh->element->size->dx;
203     scgs->vanka->A->data[3][4] = -1.0 * cfd->engine->mesh->element->size->dx;
204
205     if (abs(scgs->vanka->A->data[0][0]) < abs(scgs->vanka->A->data[4][0]) ||
206         abs(scgs->vanka->A->data[1][1]) < abs(scgs->vanka->A->data[4][1]) ||
207         abs(scgs->vanka->A->data[2][2]) < abs(scgs->vanka->A->data[4][2]) ||
208         abs(scgs->vanka->A->data[3][3]) < abs(scgs->vanka->A->data[4][3]))
209     {
210         log_info("Matrix A is not diagonally dominant @ (it=%d, i=%d, j=%d)", cfd->engine->
    method->iteractions, i, j);
211     }
212 }
213
214 void CFD_SCGS_System_Solve(SCGS_t *scgs)
215 {
216     double DEN;
217     double r1;
218     double r2;
219     double r3;
220     double r4;
221
```

```
    r1 = scgs->vanka->A->data[4][0] / scgs->vanka->A->data[0][0];
    r2 = scgs->vanka->A->data[4][1] / scgs->vanka->A->data[1][1];
    r3 = scgs->vanka->A->data[4][2] / scgs->vanka->A->data[2][2];
    r4 = scgs->vanka->A->data[4][3] / scgs->vanka->A->data[3][3];

    DEN = r1 * scgs->vanka->A->data[0][4] +
          r2 * scgs->vanka->A->data[1][4] +
          r3 * scgs->vanka->A->data[2][4] +
          r4 * scgs->vanka->A->data[3][4];

    scgs->vanka->x->data[4] = (r1 * scgs->vanka->R->data[0] +
                               r2 * scgs->vanka->R->data[1] +
                               r3 * scgs->vanka->R->data[2] +
                               r4 * scgs->vanka->R->data[3] -
                               scgs->vanka->R->data[4]) /
                              DEN;

    scgs->vanka->x->data[0] = (scgs->vanka->R->data[0] - scgs->vanka->A->data[0][4] * scgs->
    vanka->x->data[4]) / scgs->vanka->A->data[0][0];
    scgs->vanka->x->data[1] = (scgs->vanka->R->data[1] - scgs->vanka->A->data[1][4] * scgs->
    vanka->x->data[4]) / scgs->vanka->A->data[1][1];
    scgs->vanka->x->data[2] = (scgs->vanka->R->data[2] - scgs->vanka->A->data[2][4] * scgs->
    vanka->x->data[4]) / scgs->vanka->A->data[2][2];
    scgs->vanka->x->data[3] = (scgs->vanka->R->data[3] - scgs->vanka->A->data[3][4] * scgs->
    vanka->x->data[4]) / scgs->vanka->A->data[3][3];
}

void CFD_SCGS_Apply_Correction(CFD_t *cfd, SCGS_t *scgs)
{
    uint16_t i = cfd->engine->method->index->i;
    uint16_t j = cfd->engine->method->index->j;

    cfd->engine->method->state->u->data[j + 0][i - 1] += scgs->vanka->x->data[0];
    cfd->engine->method->state->u->data[j + 0][i + 0] += scgs->vanka->x->data[1];
    cfd->engine->method->state->v->data[j - 1][i + 0] += scgs->vanka->x->data[2];
    cfd->engine->method->state->v->data[j + 0][i + 0] += scgs->vanka->x->data[3];
    cfd->engine->method->state->p->data[j + 0][i + 0] += scgs->vanka->x->data[4];
}

void CFD_SCGS_Update_Residual(SCGS_t *scgs)
{
    scgs->residual->u += (fabs(scgs->vanka->R->data[0]) + fabs(scgs->vanka->R->data[1])) /
    2.0;
    scgs->residual->v += (fabs(scgs->vanka->R->data[2]) + fabs(scgs->vanka->R->data[3])) /
    2.0;
    scgs->residual->p += fabs(scgs->vanka->R->data[4]);
}
```

Listing 2: SCGS Core algorithm implementation in C.

```
#include "SCGS_BC.h"

#include "SCGS.h"
#include "../../CFD.h"
#include "../../utils/cALGEBRA/dMAT.h"

void CFD_SCGS_BC_NoSlip_Normal(CFD_t *cfd, SCGS_t *scgs)
{
    if (cfd->engine->method->index->i == cfd->engine->mesh->n_ghosts)
    {
        scgs->vanka->A->data[0][0] = 1.0;
```

```
12          scgs->vanka->A->data[0][4] = 0.0;
13          scgs->vanka->R->data[0] = 0.0;
14
15          scgs->vanka->A->data[4][0] = 0.0;
16      }
17
18      if (cfd->engine->method->index->i == cfd->engine->mesh->nodes->Nx + cfd->engine->mesh->
        n_ghosts - 1)
19      {
20          scgs->vanka->A->data[1][1] = 1.0;
21          scgs->vanka->A->data[1][4] = 0.0;
22          scgs->vanka->R->data[1] = 0.0;
23
24          scgs->vanka->A->data[4][1] = 0.0;
25      }
26
27      if (cfd->engine->method->index->j == cfd->engine->mesh->n_ghosts)
28      {
29          scgs->vanka->A->data[2][2] = 1.0;
30          scgs->vanka->A->data[2][4] = 0.0;
31          scgs->vanka->R->data[2] = 0.0;
32
33          scgs->vanka->A->data[4][2] = 0.0;
34      }
35
36      if (cfd->engine->method->index->j == cfd->engine->mesh->nodes->Ny + cfd->engine->mesh->
        n_ghosts - 1)
37      {
38          scgs->vanka->A->data[3][3] = 1.0;
39          scgs->vanka->A->data[3][4] = 0.0;
40          scgs->vanka->R->data[3] = 0.0;
41
42          scgs->vanka->A->data[4][3] = 0.0;
43      }
44  }
45
46  void CFD_SCGS_BC_NoSlip_Tangetial(CFD_t *cfd)
47  {
48      // Horizontal walls
49      for (uint16_t i = 0;
50           i < cfd->engine->mesh->nodes->Nx + 2 * cfd->engine->mesh->n_ghosts;
51           i++)
52      {
53          cfd->engine->method->state->u->data[cfd->engine->mesh->n_ghosts - 1][i] = 2.0 * 0.0
        - cfd->engine->method->state->u->data[cfd->engine->mesh->n_ghosts][i];
54          cfd->engine->method->state->u->data[cfd->engine->mesh->n_ghosts + cfd->engine->mesh
        ->nodes->Ny][i] = 2.0 * cfd->in->uLid - cfd->engine->method->state->u->data[cfd->engine
        ->mesh->n_ghosts + cfd->engine->mesh->nodes->Ny - 1][i];
55      }
56
57      // Vertical walls
58      for (uint16_t j = 0;
59           j < cfd->engine->mesh->nodes->Ny + 2 * cfd->engine->mesh->n_ghosts;
60           j++)
61      {
62          cfd->engine->method->state->v->data[j][cfd->engine->mesh->n_ghosts - 1] = 2.0 * 0.0
        - cfd->engine->method->state->v->data[j][cfd->engine->mesh->n_ghosts];
63          cfd->engine->method->state->v->data[j][cfd->engine->mesh->n_ghosts + cfd->engine->
        mesh->nodes->Nx] = 2.0 * 0.0 - cfd->engine->method->state->v->data[j][cfd->engine->mesh
        ->n_ghosts + cfd->engine->mesh->nodes->Nx - 1];
64      }
```

```
}
```

Listing 3: SCGS Boundary Conditions implementation in C.

```c
#include "convection.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "../../CFD.h"
#include "../../utils/cLOG/cLOG.h"
#include "../../utils/cALGEBRA/cVEC.h"
#include "schemes.h"

void CFD_Setup_Convection(CFD_t *cfd)
{
    switch (cfd->engine->schemes->convection->type)
    {
    case UDS:
        cfd->engine->schemes->convection->callable = CFD_Scheme_Convection_UDS;
        break;
    case CDS:
        cfd->engine->schemes->convection->callable = CFD_Scheme_Convection_CDS;
        break;
    case QUICK:
        cfd->engine->schemes->convection->callable = CFD_Scheme_Convection_QUICK;
        break;
    case HYBRID:
        cfd->engine->schemes->convection->callable = CFD_Scheme_Convection_HYBRID;
        break;
    }
}

void CFD_Scheme_Get_Flux(CFD_t *cfd, int i, int j, phi_t phi)
{

    double uPP = CFD_Get_State(cfd, u, i + 0, j + 0);
    double vPP = CFD_Get_State(cfd, v, i + 0, j + 0);

    double uPN = CFD_Get_State(cfd, u, i + 0, j + 1);
    double vPN = CFD_Get_State(cfd, v, i + 0, j + 1);

    // double uEN = CFD_Get_State(cfd, u, i + 1, j + 1);
    // double vEN = CFD_Get_State(cfd, v, i + 1, j + 1);

    double uEP = CFD_Get_State(cfd, u, i + 1, j + 0);
    double vEP = CFD_Get_State(cfd, v, i + 1, j + 0);

    // double uES = CFD_Get_State(cfd, u, i + 1, j - 1);
    double vES = CFD_Get_State(cfd, v, i + 1, j - 1);

    // double uPS = CFD_Get_State(cfd, u, i + 0, j - 1);
    double vPS = CFD_Get_State(cfd, v, i + 0, j - 1);

    // double uWS = CFD_Get_State(cfd, u, i - 1, j - 1);
    // double vWS = CFD_Get_State(cfd, v, i - 1, j - 1);

    double uWP = CFD_Get_State(cfd, u, i - 1, j + 0);
    // double vWP = CFD_Get_State(cfd, v, i - 1, j + 0);

    double uWN = CFD_Get_State(cfd, u, i - 1, j + 1);
```

```
58      // double vWN = CFD_Get_State(cfd, v, i - 1, j + 1);
59
60      double dx = cfd->engine->mesh->element->size->dx;
61      double dy = cfd->engine->mesh->element->size->dy;
62
63      switch (phi)
64      {
65      case u:
66
67          cfd->engine->schemes->convection->F->w = 1.0 / 2.0 * (uWP + uPP) * dy;
68          cfd->engine->schemes->convection->F->e = 1.0 / 2.0 * (uPP + uEP) * dy;
69          cfd->engine->schemes->convection->F->s = 1.0 / 2.0 * (vPS + vES) * dx;
70          cfd->engine->schemes->convection->F->n = 1.0 / 2.0 * (vPP + vEP) * dx;
71
72          break;
73
74      case v:
75
76          cfd->engine->schemes->convection->F->w = 1.0 / 2.0 * (uWP + uWN) * dy;
77          cfd->engine->schemes->convection->F->e = 1.0 / 2.0 * (uPP + uPN) * dy;
78          cfd->engine->schemes->convection->F->s = 1.0 / 2.0 * (vPS + vPP) * dx;
79          cfd->engine->schemes->convection->F->n = 1.0 / 2.0 * (vPP + vPN) * dx;
80
81          break;
82
83      default:
84          log_fatal("Error: CFD_Scheme_Get_Flux phi not found");
85          exit(EXIT_FAILURE);
86          break;
87      }
88  }
89
90  void CFD_Scheme_Convection_UDS(CFD_t *cfd, uint16_t i, uint16_t j, phi_t phi)
91  {
92
93      CFD_Scheme_Get_Flux(cfd, i, j, phi);
94
95      F_coefficients_t *F = cfd->engine->schemes->convection->F;
96      cVEC_t *Ap = cfd->engine->schemes->convection->coefficients;
97
98      Ap->data[WWSS] = 0.0;
99      Ap->data[WWS] = 0.0;
100     Ap->data[WWP] = 0.0;
101     Ap->data[WWN] = 0.0;
102     Ap->data[WWNN] = 0.0;
103     Ap->data[WSS] = 0.0;
104     Ap->data[WS] = 0.0;
105     Ap->data[WP] = fmax(0.0, F->w);
106     Ap->data[WN] = 0.0;
107     Ap->data[WNN] = 0.0;
108     Ap->data[PSS] = 0.0;
109     Ap->data[PS] = fmax(0.0, F->s);
110     Ap->data[PP] = fmax(0.0, F->e) + fmax(0.0, F->n) - fmin(0.0, F->s) - fmin(0.0, F->w);
111     Ap->data[PN] = -fmin(0.0, F->n);
112     Ap->data[PNN] = 0.0;
113     Ap->data[ESS] = 0.0;
114     Ap->data[ES] = 0.0;
115     Ap->data[EP] = -fmin(0.0, F->e);
116     Ap->data[EN] = 0.0;
117     Ap->data[ENN] = 0.0;
118     Ap->data[EESS] = 0.0;
```

```
119     Ap->data[EES] = 0.0;
120     Ap->data[EEP] = 0.0;
121     Ap->data[EEN] = 0.0;
122     Ap->data[EENN] = 0.0;
123 }
124 
125 void CFD_Scheme_Convection_CDS(CFD_t *cfd, uint16_t i, uint16_t j, phi_t phi)
126 {
127     CFD_Scheme_Get_Flux(cfd, i, j, phi);
128 
129     F_coefficients_t *F = cfd->engine->schemes->convection->F;
130     cVEC_t *Ap = cfd->engine->schemes->convection->coefficients;
131 
132     Ap->data[WWSS] = 0.0;
133     Ap->data[WWS] = 0.0;
134     Ap->data[WWP] = 0.0;
135     Ap->data[WWN] = 0.0;
136     Ap->data[WWNN] = 0.0;
137     Ap->data[WSS] = 0.0;
138     Ap->data[WS] = 0.0;
139     Ap->data[WP] = F->w / 2.0;
140     Ap->data[WN] = 0.0;
141     Ap->data[WNN] = 0.0;
142     Ap->data[PSS] = 0.0;
143     Ap->data[PS] = F->s / 2.0;
144     Ap->data[PP] = F->e / 2.0 + F->n / 2.0 - F->s / 2.0 - F->w / 2.0;
145     Ap->data[PN] = -F->n / 2.0;
146     Ap->data[PNN] = 0.0;
147     Ap->data[ESS] = 0.0;
148     Ap->data[ES] = 0.0;
149     Ap->data[EP] = -F->e / 2.0;
150     Ap->data[EN] = 0.0;
151     Ap->data[ENN] = 0.0;
152     Ap->data[EESS] = 0.0;
153     Ap->data[EES] = 0.0;
154     Ap->data[EEP] = 0.0;
155     Ap->data[EEN] = 0.0;
156     Ap->data[EENN] = 0.0;
157 }
158 
159 void CFD_Scheme_Convection_QUICK(CFD_t *cfd, uint16_t i, uint16_t j, phi_t phi)
160 {
161     CFD_Scheme_Get_Flux(cfd, i, j, phi);
162 
163     F_coefficients_t *F = cfd->engine->schemes->convection->F;
164     cVEC_t *Ap = cfd->engine->schemes->convection->coefficients;
165 
166     Ap->data[WWSS] = 0.0;
167     Ap->data[WWS] = 0.0;
168     Ap->data[WWP] = -1.0 / 8.0 * fmax(0.0, F->w);
169     Ap->data[WWN] = 0.0;
170     Ap->data[WWNN] = 0.0;
171     Ap->data[WSS] = 0.0;
172     Ap->data[WS] = 0.0;
173     Ap->data[WP] = 1.0 / 8.0 * fmax(0.0, F->e) + 3.0 / 4.0 * fmax(0.0, F->w) + 3.0 / 8.0 *
        fmin(0.0, F->w);
174     Ap->data[WN] = 0.0;
175     Ap->data[WNN] = 0.0;
176     Ap->data[PSS] = -1.0 / 8.0 * fmax(0.0, F->s);
177     Ap->data[PS] = 1.0 / 8.0 * fmax(0.0, F->n) + 3.0 / 4.0 * fmax(0.0, F->s) + 3.0 / 8.0 *
        fmin(0.0, F->s);
```

```
178     Ap->data[PP] = 3.0 / 4.0 * fmax(0.0, F->e) + 3.0 / 8.0 * fmin(0.0, F->e) + 3.0 / 4.0 *
        fmax(0.0, F->n) + 3.0 / 8.0 * fmin(0.0, F->n) - 3.0 / 8.0 * fmax(0.0, F->s) - 3.0 / 4.0
        * fmin(0.0, F->s) - 3.0 / 8.0 * fmax(0.0, F->w) - 3.0 / 4.0 * fmin(0.0, F->w);
179     Ap->data[PN] = 1.0 / 8.0 * (-3.0) * fmax(0.0, F->n) - 3.0 / 4.0 * fmin(0.0, F->n) - 1.0
        / 8.0 * fmin(0.0, F->s);
180     Ap->data[PNN] = 1.0 / 8.0 * fmin(0.0, F->n);
181     Ap->data[ESS] = 0.0;
182     Ap->data[ES] = 0.0;
183     Ap->data[EP] = 1.0 / 8.0 * (-3.0) * fmax(0.0, F->e) - 3.0 / 4.0 * fmin(0.0, F->e) - 1.0
        / 8.0 * fmin(0.0, F->w);
184     Ap->data[EN] = 0.0;
185     Ap->data[ENN] = 0.0;
186     Ap->data[EESS] = 0.0;
187     Ap->data[EES] = 0.0;
188     Ap->data[EEP] = 1.0 / 8.0 * fmin(0.0, F->e);
189     Ap->data[EEN] = 0.0;
190     Ap->data[EENN] = 0.0;
191 }
192
193 void CFD_Scheme_Convection_HYBRID(CFD_t *cfd, uint16_t i, uint16_t j, phi_t phi)
194 {
195     CFD_Scheme_Get_Flux(cfd, i, j, phi);
196
197     // F_coefficients_t *F = cfd->engine->schemes->convection->F;
198     // cVEC_t *Ap = cfd->engine->schemes->convection->coefficients;
199
200     log_fatal("HYBRID not implemented yet");
201     exit(EXIT_FAILURE);
202 }
```

Listing 4: Convection Schemes implementation in C.

```
1  #include "diffusion.h"
2
3  #include "../../CFD.h"
4  #include "../../utils/cLOG/cLOG.h"
5  #include "../../utils/cALGEBRA/cVEC.h"
6  #include "schemes.h"
7
8  void CFD_Setup_Diffusion(CFD_t *cfd)
9  {
10     switch (cfd->engine->schemes->diffusion->type)
11     {
12     case SECOND:
13         cfd->engine->schemes->diffusion->callable = CFD_Scheme_Diffusion_SECOND;
14         break;
15     case FOURTH:
16         cfd->engine->schemes->diffusion->callable = CFD_Scheme_Diffusion_FOURTH;
17         break;
18     }
19 }
20
21 void CFD_Scheme_Diffusion_SECOND(CFD_t *cfd)
22 {
23     cVEC_t *Ap = cfd->engine->schemes->diffusion->coefficients;
24
25     double dx = cfd->engine->mesh->element->size->dx;
26     double dy = cfd->engine->mesh->element->size->dy;
27     double nu = cfd->in->fluid->nu;
28
29     Ap->data[WWSS] = 0.0;
```

```
    Ap->data[WWS] = 0.0;
    Ap->data[WWP] = 0.0;
    Ap->data[WWN] = 0.0;
    Ap->data[WWNN] = 0.0;
    Ap->data[WSS] = 0.0;
    Ap->data[WS] = 0.0;
    Ap->data[WP] = (dy * nu) / (dx);
    Ap->data[WN] = 0.0;
    Ap->data[WNN] = 0.0;
    Ap->data[PSS] = 0.0;
    Ap->data[PS] = (dx * nu) / (dy);
    Ap->data[PP] = -1.0 * dx * dy * nu * (-2.0 / (dx * dx) - 2.0 / (dy * dy));
    Ap->data[PN] = (dx * nu) / (dy);
    Ap->data[PNN] = 0.0;
    Ap->data[ESS] = 0.0;
    Ap->data[ES] = 0.0;
    Ap->data[EP] = (dy * nu) / (dx);
    Ap->data[EN] = 0.0;
    Ap->data[ENN] = 0.0;
    Ap->data[EESS] = 0.0;
    Ap->data[EES] = 0.0;
    Ap->data[EEP] = 0.0;
    Ap->data[EEN] = 0.0;
    Ap->data[EENN] = 0.0;
}

void CFD_Scheme_Diffusion_FOURTH(CFD_t *cfd)
{
    cVEC_t *Ap = cfd->engine->schemes->diffusion->coefficients;

    double dx = cfd->engine->mesh->element->size->dx;
    double dy = cfd->engine->mesh->element->size->dy;
    double nu = cfd->in->fluid->nu;

    Ap->data[WWSS] = 0.0;
    Ap->data[WWS] = 0.0;
    Ap->data[WWP] = -(dy * nu) / (12.0 * dx);
    Ap->data[WWN] = 0.0;
    Ap->data[WWNN] = 0.0;
    Ap->data[WSS] = 0.0;
    Ap->data[WS] = 0.0;
    Ap->data[WP] = (4.0 * dy * nu) / (3.0 * dx);
    Ap->data[WN] = 0.0;
    Ap->data[WNN] = 0.0;
    Ap->data[PSS] = -(dx * nu) / (12.0 * dy);
    Ap->data[PS] = (4.0 * dx * nu) / (3.0 * dy);
    Ap->data[PP] = -dx * dy * nu * (-5.0 / (2.0 * dx * dx) - 5.0 / (2.0 * dy * dy));
    Ap->data[PN] = (4.0 * dx * nu) / (3.0 * dy);
    Ap->data[PNN] = -(dx * nu) / (12.0 * dy);
    Ap->data[ESS] = 0.0;
    Ap->data[ES] = 0.0;
    Ap->data[EP] = (4.0 * dy * nu) / (3.0 * dx);
    Ap->data[EN] = 0.0;
    Ap->data[ENN] = 0.0;
    Ap->data[EESS] = 0.0;
    Ap->data[EES] = 0.0;
    Ap->data[EEP] = -(dy * nu) / (12.0 * dx);
    Ap->data[EEN] = 0.0;
    Ap->data[EENN] = 0.0;
}
```

Listing 5: Diffusion Schemes implementation in C.