Numerical Solution of the Convection-Diffusion Equation

Author:
Sara-Medina Šehović

April 1, 2025

Abstract

This assignment deals with the numerical solution of the convection-diffusion equation. The emphasis is laid on solving the Smith-Hutton case, as proposed in the problem. The report consists of the Smith-Hutton problem statement, a short theoretical description of the convection-diffusion equation, solving methodology, and results and discussion. Additionally, the report deals with the comparison between the Central Difference Scheme (CDS) and Upwind Difference Scheme (UDS).

Contents

Contents **2**

1 Introduction **3**

2 Convection-Diffusion Equation **3**

3 Smith-Hutton Case **4**
3.1 Methodology 5
3.1.1 Discretization of the domain 5
3.1.2 Discretization equations 6
3.1.3 Discretization equations using Central Difference Scheme (CDS) 6
3.1.4 Discretization equations using Upwind Difference Scheme (UDS) 9
3.2 Code structure 10
3.3 Results 12
3.3.1 Results obtained using the Central Difference Scheme (CDS) 12
3.3.2 Results obtained using the Upwind Difference Scheme (UDS) 15
3.4 Discussion 18

4 Conclusions **19**

5 Future work **19**

6 Numerical solution of the Convection-Diffusion equation - Code **20**
6.1 UDS scheme coefficients 26

7 Matplotlib file for results plotting **27**

1 Introduction

This assignment deals with the numerical solution of the convection-diffusion equation. The goal of this assignment is to perform the numerical solution using C++ programming language. To do so, we will be focusing on the *Smith-Hutton case*, given in *Exercise 4*. Our focus is on creating a program which will solve the proposed case, and which will allow us to perform certain analyses to further understand the nature of the convection-diffusion equation.

2 Convection-Diffusion Equation

The convection-diffusion equation is a partial differential equation that describes the unsteady transport of different quantities, such as mass, momentum, energy, or species k, as well as transport equations of kinetic energy, entropy, etc. Transport equations of the species mentioned, all have a common structure, composed of unsteady terms, convective, diffusion, and other terms. Hence, the convection-diffusion equation can be written as:

$$\frac{\partial \rho\phi}{\partial t} \; + \; \nabla \cdot (\rho \boldsymbol{v} \phi) = \nabla \cdot (\Gamma \nabla \phi) \; + \; \mathcal{S}_{\mathcal{R}} \tag{1}$$

In order to obtain the numerical solution of Equation 1, we must first discretize the domain by generating control volumes, and then discretize each term of the Equation 1 by applying the Finite Volume Method (FVM), and Gauss theorem.

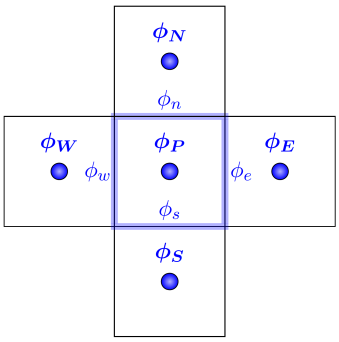

Figure 1: Mesh notation

Figure 1 represents the node notation, in which the observed node P is surrounded by the east W, west W, north N, and south S nodes. Values of quantity ϕ at each node are denoted with capital letters (ϕ_P, ϕ_E, etc.), and these values are known, while the quantity values at control volume faces are denoted by lowercase letters (ϕ_e, ϕ_w, etc.) and are unknown, and have to be calculated using convective schemes, which is one of the main problems of this assignment. To obtain the discretization equation, firstly, we must integrate the Equation 1 over a control volume $\mathcal{V}$:

$$\int_{\mathcal{V}} \frac{\partial \rho\phi}{\partial t} d\mathcal{V} + \int_{\mathcal{V}} \nabla \cdot (\rho \boldsymbol{v} \phi) \, d\mathcal{V} = \int_{\mathcal{V}} \nabla \cdot (\Gamma \nabla \phi) \, d\mathcal{V} + \int_{\mathcal{V}} \mathcal{S}_{\mathcal{R}} d\mathcal{V} \tag{2}$$

Next, the Gauss theorem is applied:

$$\int_{\mathcal{V}} \frac{\partial \rho\phi}{\partial t} d\mathcal{V} + \int_{S} (\rho \boldsymbol{v} \phi) \cdot d\boldsymbol{S} = \int_{S} \Gamma \nabla \phi \cdot d\boldsymbol{S} + \int_{\mathcal{V}} \mathcal{S}_{\mathcal{R}} d\mathcal{V} \tag{3}$$

Next, we follow with the finite volume method:

$$\frac{\partial \rho\phi}{\partial t} \Delta\mathcal{V} + \sum (\rho \boldsymbol{v} \phi)_f \cdot \Delta \boldsymbol{S} = \sum (\Gamma \nabla \phi)_f \cdot \Delta \boldsymbol{S} + \mathcal{S}_{\mathcal{R}} \Delta\mathcal{V} \tag{4}$$

Finally, for a 2D case, we obtain the following equations for the diffusive and convective terms:

$$diff = \sum (\Gamma \nabla \phi)_f \cdot \Delta \boldsymbol{S} = \Gamma_e \frac{\phi_E - \phi_P}{d_{PE}} + \Gamma_w \frac{\phi_W - \phi_P}{d_{PW}} + \Gamma_n \frac{\phi_N - \phi_P}{d_{PN}} + \Gamma_s \frac{\phi_S - \phi_P}{d_{PS}} \tag{5}$$

and

$$conv = \sum (\Gamma \nabla \phi)_f \cdot \Delta \boldsymbol{S} = \rho \left(u_e \phi_e \Delta S_e - u_w \phi_w \Delta S_w + v_n \phi_n \Delta S_n - v_s \phi_s \Delta S_s \right) \tag{6}$$

In Equations 5 and 6, u, represents the horizontal velocity component, while v represents the vertical component of the velocity vector $\boldsymbol{v} = (u, v)$.

As mentioned above, node values (ϕ_P, ϕ_E, etc.) are known, while the quantity values at faces (ϕ_e, ϕ_n, etc.) are unknown and need to be calculated using convective schemes. One of the main goals of this assignment is to numerically solve the convection-diffusion equation using convective schemes. Hence, we will be focusing on two convective schemes, namely Central Difference Scheme (CDS) and Upwind Difference Scheme (UDS), with application on the *Smith-Hutton case*.

3 Smith-Hutton Case

The Figure 2 represents our Smith-Hutton problem. The case deals with a bounded solenoidal flow, where we want to study the spatial evolution of a property ϕ at steady state for different ratios of ρ/Γ.

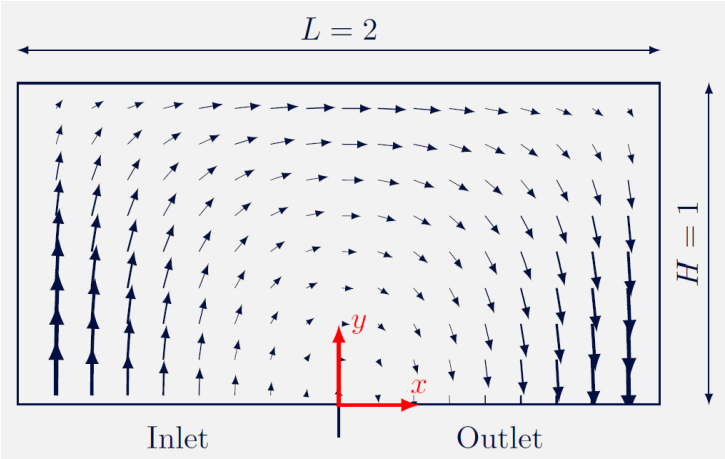

Figure 2: Smith-Hutton case

The velocity field $\boldsymbol{v} = (u, v)$ is known and is given as a function of the coordinates x and y. Note that $x = 0$ is at the center of the domain, so, $x \in [-1, 1]$ and $y \in [0, 1]$:

$$u\,(x, y) = 2y\left(1 - x^2\right) \tag{7}$$

$$v\,(x, y) = -2x\left(1 - y^2\right) \tag{8}$$

The boundary conditions for the magnitude ϕ are given as:

$$\phi = 1 + \tanh\left[(2x + 1)\,\alpha\right] \quad \text{For the inlet } (x \in [-1, 0]\,,\ y = 0) \tag{9}$$

$$\frac{\partial \phi}{\partial y} = 0 \quad \text{For the outlet } (x \in [1, 0]\,,\ y = 0) \tag{10}$$

$$\phi = 1 - \tanh[\alpha] \quad \text{For the rest of the walls} \tag{11}$$

where $\alpha = 10$. Our task is to solve the case and obtain the results for the steady state. In order to do so, we will give some initial conditions, which are set as $\phi = 0\ \forall\ (x, y)$. Then, we are required to solve the transient equation until it converges. To perform the calculations, we may use the UDS or the CDS scheme. Our tasks are as follows:

a) Solve for the following ρ/Γ ratios: $\rho/\Gamma = 10$, $\rho/\Gamma = 1000$, and $\rho/\Gamma = 1\ 000\ 000$.

b) Plot the value of ϕ at the outlet as a function of x.

c) Plot a color map of the three different cases.

Bonus tasks:

a) Try both UDS and CDS schemes, compare the results and behaviour differences.

b) Instead of using a uniform mesh, use a non-uniform one.

c) Implement the case using some classes or functions.

3.1 Methodology

To perform the required tasks, firstly, we must set up the simulation. The calculations were performed using the C++ programming language, while the graphics were obtained using the *Matplotlib* Python library. To write the code, we must discretize the convection-diffusion Equation 1, using the Finite Volume Method presented by the Equation 4. Then, these calculations must be implemented in the code and verified by the given reference data.

3.1.1 Discretization of the domain

To implement the Finite Volume Method, we will firstly discretize the domain. The domain is divided into a uniform mesh, as presented by the Figure 3 below:

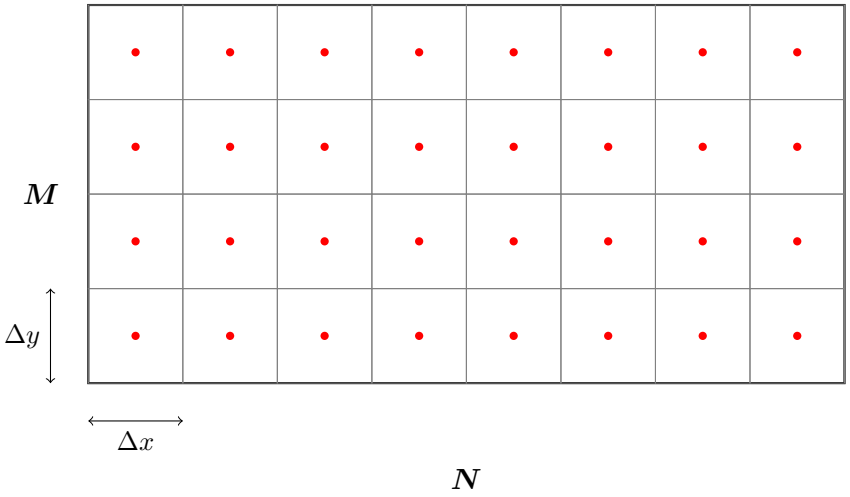

Figure 3: Uniform mesh

From the figure above, we can see that our mesh is divided into N control volumes in the x-direction and M control volumes in the y-direction. Each control volume, as being a part of a uniform mesh, has a size $\Delta x \times \Delta y$, where Δx and Δy are grid sizes in x- and y-direction, respectively. The nodes are placed at the center of each control volume and are marked with red dots in Figure 3. The mesh sizes are calculated as follows:

$$\Delta x = \frac{L}{N} \tag{12}$$

and

$$\Delta y = \frac{H}{M} \tag{13}$$

where $L = 2.0\ m$, and $H = 1.0\ m$ are the length and the height of the domain, respectively, as given by the problem statement.

After evaluating the grid sizes, we can calculate the node positions x_P and y_P, control volume positions x_{CV} and y_{CV}, etc. The position of control volume is calculated as:

$$x_{CV} = x[0] + i \cdot \Delta x \tag{14}$$

and

$$y_{CV} = y[0] + j \cdot \Delta y \tag{15}$$

where $x[0]$ and $y[0]$ are x and y positions of the bottom-left corner $(-1, 0)$, and i and j denote the number of a control volume. The control volumes are marked from $i = 1$ to $i = N$ in the x-direction, and $j = 1$ to $j = M$ in the y-direction. This is implemented similarly in the C++ program: the index [0] and $[N + 1]$ belong to the boundaries, and are defined as such to simplify the calculations; hence, the nodes in C++ are marked with 1 to N.

Node positions are calculated as:

$$x_P = \frac{x_{CV}[i] + x_{CV}[i+1]}{2} \tag{16}$$

and

$$y_P = \frac{y_{CV}[i] + y_{CV}[i+1]}{2} \tag{17}$$

for the defined nodes. In the program, the position of control volumes and nodes is calculated using C++ *functions*.

3.1.2 Discretization equations

Similarly to the previous assignment, in order to obtain the numerical solution of the convection-diffusion Equation 1, we must first obtain the discretized equation, and since we are dealing with the explicit time-integration scheme, the discretization equation will be in the form:

$$a_P T_P^{n+1} = a_E T_E^n + a_W T_W^n + a_N T_N^n + a_S T_S^n + b_P \tag{18}$$

We can write the Equation 4 after some rearrangements and time discretization as:

$$\rho \frac{\phi_P^{n+1} - \phi_P^n}{\Delta t} = \frac{1}{\Delta \mathcal{V}} \left(-conv^n - diff^n\right) + {\mathcal{S}_\mathcal{R}}^n \tag{19}$$

where convective and diffusive terms are given as:

$$conv = \rho \left(u_e \phi_e \Delta S_e - u_w \phi_w \Delta S_w + v_n \phi_n \Delta S_n - v_s \phi_s \Delta S_s\right) \tag{20}$$

$$diff = \Gamma_e \frac{\phi_E - \phi_P}{d_{PE}} \Delta S_e + \Gamma_w \frac{\phi_W - \phi_P}{d_{PW}} \Delta S_w + \Gamma_n \frac{\phi_N - \phi_P}{d_{PN}} \Delta S_n + \Gamma_s \frac{\phi_S - \phi_P}{d_{PS}} \Delta S_s \tag{21}$$

These equations are used for discretization of all nodes, and regardless of the convective scheme, since they serve as the base for further calculations. Both UDS and CDS schemes can be derived from them, as well as boundary conditions. As seen from above, Equation 19 is discretized in time using an explicit time-integration scheme. The following sections will describe the discretization equations for different convective schemes.

3.1.3 Discretization equations using Central Difference Scheme (CDS)

In this section, we will deal with discretization equations using the Central Difference Scheme (CDS). Since we are dealing with a uniform mesh, we will calculate the values at faces as the average between the observed and its neighboring node (e.g. the value ϕ_e at the eastern face is obtained as $\phi_e = \frac{\phi_P + \phi_E}{2}$).

However, the CDS scheme, despite being of the second order, is prone to stability problems. On the other hand, it is easy to implement, but it has some disadvantages, which result in instabilities.

As mentioned above, using a uniform mesh simplifies the implementation of the CDS scheme, and the values at faces are calculated simply as the average between the neighbouring nodes, since we are considering a linear distribution of the quantity ϕ. The graphical representation of the CDS scheme is presented in the following figure:

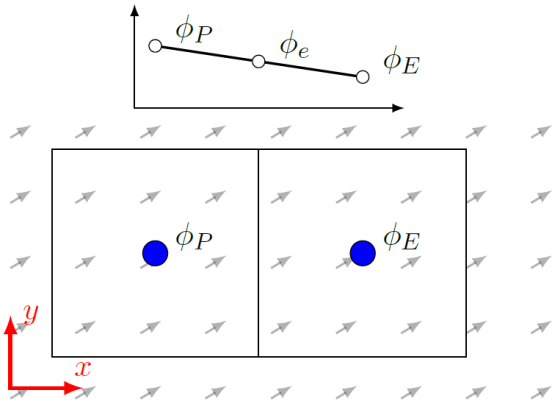

Figure 4: Central Difference Scheme (CDS) on a uniform mesh $\left(\frac{\phi_P+\phi_E}{2}\right)$

- **Discretization coefficients for internal nodes**

The internal nodes are considered the nodes that are not adjacent to the boundaries, that is, the nodes that have i-coefficients from [2] to $[N-1]$ and j-coefficients from [2] to $[M-1]$. Hence, the discretization Equation 18, when implemented in the convection-diffusion Equation 19 gives the following coefficients for the internal nodes:

$$a_P = \frac{\gamma}{\Delta t},\ a_E = \frac{-\gamma u_e}{2\Delta x} + \frac{1}{d_{PE}\Delta x},\ a_W = \frac{\gamma u_w}{2\Delta x} + \frac{1}{d_{PW}\Delta x},$$
$$a_N = \frac{-\gamma v_n}{2\Delta y} + \frac{1}{d_{PN}\Delta y},\ a_S = \frac{\gamma v_s}{2\Delta y} + \frac{1}{d_{PS}\Delta y},$$
$$b_P = \phi_P^n \left[\frac{-\gamma u_e}{2\Delta x} - \frac{1}{d_{PE}\Delta x} + \frac{\gamma u_w}{2\Delta x} - \frac{1}{d_{PW}\Delta x} + \frac{-\gamma v_n}{2\Delta y} - \frac{1}{d_{PN}\Delta y} + \frac{\gamma v_s}{2\Delta y} - \frac{1}{d_{PS}\Delta y} + \frac{\gamma}{\Delta t}\right]$$

where $\gamma = \rho/\Gamma$, used in order to simplify both the writing and the code implementation. These coefficients are calculated for each internal node, with the help of *for* loops in C++.

- **Discretization coefficients for boundary nodes**

Calculating the coefficients for boundary nodes is somewhat more complex, as we have different boundary conditions at different walls. Hence, we will observe five different boundary conditions: inlet, outlet, left wall, upper wall, and right wall. The discretization coefficients are described in the following passages.

– *Left boundary nodes*

Left boundary nodes are considered nodes with i-coefficient of [1] and j-coefficients ranging from [2] to $[M]$. Since we know the values of the quantity ϕ on the left boundary wall, we are dealing with a case of Dirichlet boundary conditions. We follow the same discretization approach as for the internal nodes, with coefficients a_W and b_P now calculated as:

$$a_W = \frac{\gamma u_w}{\Delta x} + \frac{1}{d_{PW}\Delta x}$$
$$b_p = \phi_P^n \left[\frac{-\gamma u_e}{2\Delta x} - \frac{1}{d_{PE}\Delta x} - \frac{1}{d_{PW}\Delta x} + \frac{-\gamma v_n}{2\Delta y} - \frac{1}{d_{PN}\Delta y} + \frac{\gamma v_s}{2\Delta y} - \frac{1}{d_{PS}\Delta y} + \frac{\gamma}{\Delta t}\right]$$

Furthermore, node [1][M], since placed in a corner with boundaries on the north and on the left, will, in addition to the coefficient a_W described above, have coefficients a_N and a_P calculated as:

$$a_N = -\frac{\gamma v_n}{\Delta x} + \frac{1}{d_{PN}\Delta y}$$

$$b_p = \phi_P^n \left[\frac{-\gamma u_e}{2\Delta x} - \frac{1}{d_{PE}\Delta x} - \frac{1}{d_{PW}\Delta x} - \frac{1}{d_{PN}\Delta y} + \frac{\gamma v_s}{2\Delta y} - \frac{1}{d_{PS}\Delta y} + \frac{\gamma}{\Delta t} \right]$$

– *Right boundary nodes*

Similarly to the previous case, the right boundary is also considered a Dirichlet boundary condition, and the nodes that belong to this are the ones with i-coefficient of [N] and j-coefficients ranging from [2] to [M]. Now, compared to the internal node discretization, we have different coefficients a_E and consequently b_P:

$$a_E = \frac{-\gamma u_e}{\Delta x} + \frac{1}{d_{PE}\Delta x}$$

$$b_p = \phi_P^n \left[-\frac{1}{d_{PE}\Delta x} + \frac{\gamma u_w}{2\Delta x} - \frac{1}{d_{PW}\Delta x} + \frac{-\gamma v_n}{2\Delta y} - \frac{1}{d_{PN}\Delta y} + \frac{\gamma v_s}{2\Delta y} - \frac{1}{d_{PS}\Delta y} + \frac{\gamma}{\Delta t} \right]$$

Again, the upper-right node is a subject to two boundary conditions, hence, we further change the coefficients a_N and b_P as:

$$a_N = -\frac{\gamma v_n}{\Delta y} + \frac{1}{d_{PN}\Delta y}$$

$$b_p = \phi_P^n \left[-\frac{1}{d_{PE}\Delta x} + \frac{\gamma u_w}{2\Delta x} - \frac{1}{d_{PW}\Delta x} - \frac{1}{d_{PN}\Delta y} + \frac{\gamma v_s}{2\Delta y} - \frac{1}{d_{PS}\Delta y} + \frac{\gamma}{\Delta t} \right]$$

– *Upper boundary nodes*

Upper boundary nodes adhere to the Dirichlet boundary condition as well. The nodes that belong to this range have the j-coefficient of M and i-coefficient ranging from [2] to [$N-1$]. Here, coefficients a_N and a_P will be calculated as:

$$a_N = -\frac{\gamma v_n}{\Delta y} + \frac{1}{d_{PN}\Delta y}$$

$$b_p = \phi_P^n \left[\frac{-\gamma u_e}{2\Delta x} - \frac{1}{d_{PE}\Delta x} + \frac{\gamma u_w}{2\Delta x} - \frac{1}{d_{PW}\Delta x} - \frac{1}{d_{PN}\Delta y} + \frac{\gamma v_s}{2\Delta y} - \frac{1}{d_{PS}\Delta y} + \frac{\gamma}{\Delta t} \right]$$

– *Inlet boundary nodes*

The inlet boundary nodes also follow the Dirichlet boundary condition. However, we must be careful with the definition of which nodes neighbour the inlet boundary, since we the bottom edge is divided into inlet and outlet. All the inlet boundary nodes have the j-coefficient of [1], while the i-coefficients are calculated depending on whether the node is placed at $x < 0$, as per the problem statement. This is implemented in the code using the *if* condition in the program, which will allow us to automatically change the number of nodes, without the need for further calculations. Similarly to the previous cases, we change the coefficients a_S and a_P as:

$$a_S = \frac{\gamma v_s}{\Delta y} + \frac{1}{d_{PS}\Delta y}$$

$$b_p = \phi_P^n \left[\frac{-\gamma u_e}{2\Delta x} - \frac{1}{d_{PE}\Delta x} + \frac{\gamma u_w}{2\Delta x} - \frac{1}{d_{PW}\Delta x} + \frac{-\gamma v_n}{2\Delta y} - \frac{1}{d_{PN}\Delta y} - \frac{1}{d_{PS}\Delta y} + \frac{\gamma}{\Delta t} \right]$$

Bottom-right node [1][1] adheres to two Dirichlet boundary conditions, hence we have altered coefficients a_W and a_P:

$$a_W = \frac{\gamma u_e}{\Delta x} + \frac{1}{d_{PW}\Delta x}$$

$$b_p = \phi_P^n \left[\frac{-\gamma u_w}{2\Delta x} - \frac{1}{d_{PW}\Delta x} - \frac{1}{d_{PE}\Delta x} + \frac{-\gamma v_n}{2\Delta y} - \frac{1}{d_{PN}\Delta y} - \frac{1}{d_{PS}\Delta y} + \frac{\gamma}{\Delta t} \right]$$

– *Outlet boundary nodes*

Outlet boundary condition differs from the previously defined boundary conditions, since the outlet is defined as the Neumann boundary condition per problem statement. The nodes that adhere to this boundary condition have j-coefficient of [1], however, the i-coefficients that belong to this rule are the ones placed at $x > 0$, which continues from the condition described for the inlet boundary condition. Hence, the discretization coefficients a_S and b_P will now be calculated as:

$$a_S = 0$$

$$b_p = \phi_P^n \left[\frac{-\gamma u_e}{2\Delta x} - \frac{1}{d_{PE}\Delta x} + \frac{\gamma u_w}{2\Delta x} - \frac{1}{d_{PW}\Delta x} + \frac{-\gamma v_n}{2\Delta y} - \frac{1}{d_{PN}\Delta y} + \frac{\gamma v_s}{\Delta y} - \frac{1}{d_{PS}\Delta y} + \frac{\gamma}{\Delta t} \right]$$

The node $[N][1]$ adheres to the Dirichlet boundary condition on the east and the Neumann boundary condition at the south. Hence, the coefficients a_E and b_P will now be:

$$a_E = \frac{-\gamma u_e}{\Delta x} + \frac{1}{d_{PE}\Delta x}$$

$$b_p = \phi_P^n \left[-\frac{1}{d_{PE}\Delta x} + \frac{\gamma u_w}{2\Delta x} - \frac{1}{d_{PW}\Delta x} + \frac{-\gamma v_n}{2\Delta y} - \frac{1}{d_{PN}\Delta y} + \frac{\gamma v_s}{\Delta y} - \frac{1}{d_{PS}\Delta y} + \frac{\gamma}{\Delta t} \right]$$

3.1.4 Discretization equations using Upwind Difference Scheme (UDS)

Upwind difference scheme (UDS) is a first-order accurate convective scheme. However, it is much more stable than the CDS scheme. For incompressible flows, or gases at low Mach, convective terms are more influenced by upstream than downstream conditions. This is why the UDS scheme serves as a good alternative to the unstable CDS scheme. The graphical representation of the UDS scheme is presented in the following figure:

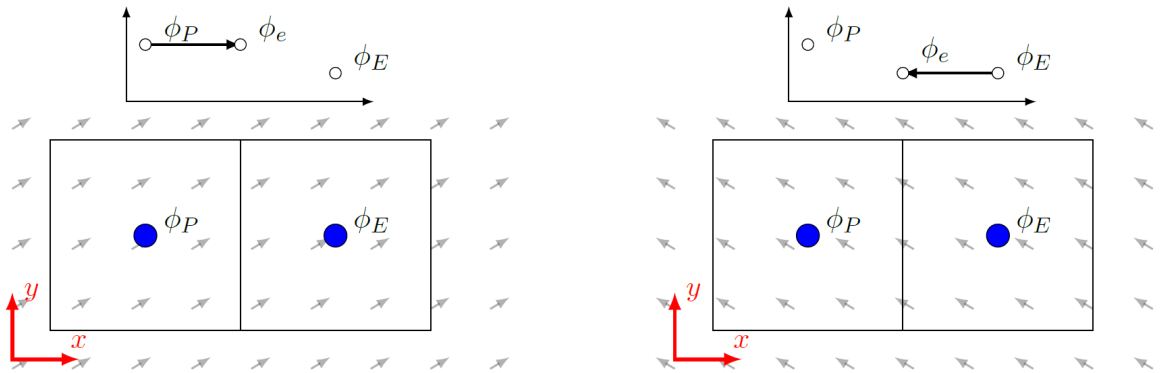

Figure 5: Upwind Difference Scheme (UDS) on a uniform mesh (examples)

From the figure above, we can see how the UDS scheme is implemented. In case of positive velocity at the face ϕ_e, the value assigned to ϕ_e is ϕ_P, since that is the upwind value in this case. In case of a negative velocity entering at face e, the value assigned to ϕ_e is ϕ_E, since that value is the upwind value for that case.

To implement the UDS scheme, we use a simple equation:

$$\phi_e - \phi_P = f_e \left(\phi_E - \phi_P \right) \tag{22}$$

from here, we can deduce:

$$\phi_e = f_e \phi_E + (1 - f_e)\, \phi_P \tag{23}$$

Here, $f_e = 0$ if $u_e > 0$ and $f_e = 1$ if $u_e < 0$. Similarly, we can deduce the same coefficients for west, north and south face. However, in case of the west and south face, $f_w = 1$ if $u_w > 0$ and $f_w = 1$ if $u_e > 0$. The same is applied to the south face. This happens because of the position of the nodes.

To implement these coefficients in our code, a special function was made, which is named `calculateUDSCoefficients()`, and it allows us to calculate the coefficients without unnecessary repetitions easily and with accordance to the given case.

Afterwards, we follow a similar approach as with the CDS scheme. However, in this case, we will calculate the same coefficients for all the nodes, starting from the node [1][1] and ending with the node $[N][M]$. The discretization coefficients are given as follows:

$$\begin{gathered}
a_E = -\gamma \frac{u_e f_e}{\Delta x} + \frac{1}{d_{PE}\Delta x}, \\
a_W = \gamma \frac{u_w f_w}{\Delta x} + \frac{1}{d_{PW}\Delta x}, \\
a_N = -\gamma \frac{v_n f_n}{\Delta y} + \frac{1}{d_{PN}\Delta y}, \\
a_S = \gamma \frac{v_s f_s}{\Delta y} + \frac{1}{d_{PS}\Delta y}, \\
b_P = \phi_P^n \left(-\gamma \frac{u_e(1-f_e)}{\Delta x} - \frac{1}{d_{PE}\Delta x} + \gamma \frac{u_w(1-f_w)}{\Delta x} - \frac{1}{d_{PW}\Delta x} \right. \\
\left. -\gamma \frac{v_n(1-f_n)}{\Delta y} - \frac{1}{d_{PN}\Delta y} + \gamma \frac{v_s(1-f_s)}{2\Delta y} - \frac{1}{d_{PS}\Delta y} + \frac{\gamma}{\Delta t} \right), \\
a_P = \frac{\gamma}{\Delta t}.
\end{gathered}$$

After calculating the coefficients, we enter the time loop, and proceed with calculations until steady state is reached. Once again, since all coefficients are constant, in order to simplify the code and make it faster, we will separate the coefficient b_P into ϕ_P^n and b_P^{const}, so we only have to calculate the coefficients once, while the value ϕ_P^n is calculated and updated at each time step.

Since the UDS convection scheme is numerically diffusive, we expect different behaviour from the CDS scheme. Hence, we will implement stricter time tolerance $\delta = 10^{-8}$, to ensure that proper steady state is reached.

The following section represents the code structure, used for both CDS and UDS convective schemes.

3.2 Code structure

Apart from the main part of the code, the code also consists of various functions, declared in order to simplify the calculations, and avoid excessive lines in the code. Furthermore, they help speed up the running of the code. For example, function `domainLength()` calculates the domain size according to the given coordinates. This simplifies calculations and allows us to automatically alter the domain size, without affecting other calculations. Furthermore, functions `gridSize()`, `cvPosition()`, and `nodePosition()`, calculate the mesh size and the position of control volumes and nodes. Additionally, functions `velocityU()` and `velocityV()` calculate the u and v components of velocity at different positions (x, y).

The `main()` in the C++ program contains the main code algorithm. The main code algorithm is presented in the following figure:

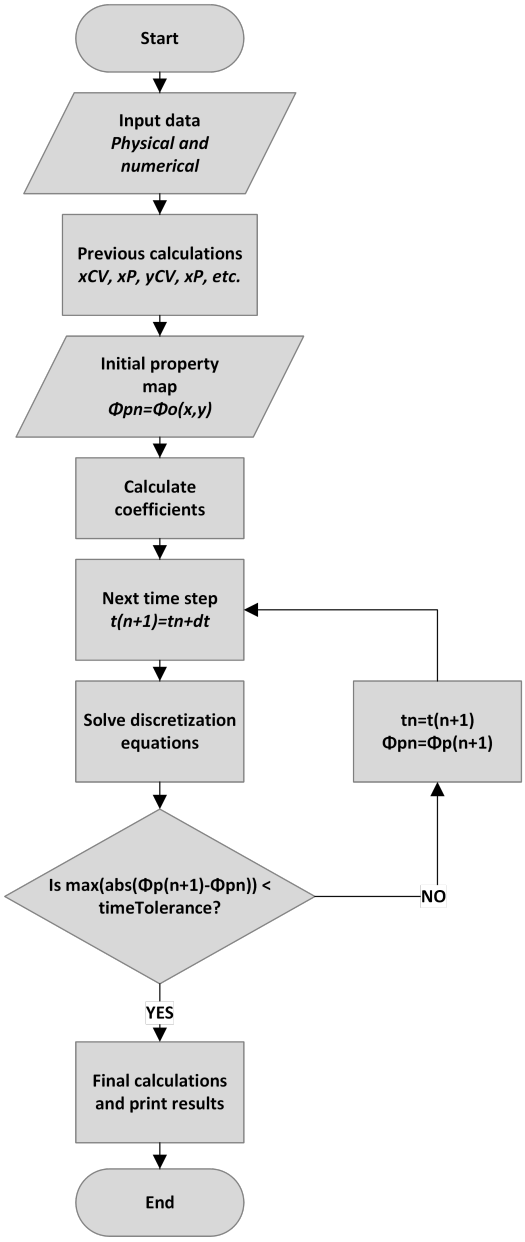

Figure 6: Global code algorithm

In this code, the time loop is achieved using the *do-while* loop, since we want to calculate the difference in ϕ quantity value at each point between two time steps until the defined criterion is satisfied. The *do-while* loop was chosen, since it allows us to perform calculations at least once, even if the condition is satisfied. Furthermore, *do-while* loop allows us to implement the convergence conditions easily, since we can reset the calculated value of $\phi_p^{n+1} - \phi_P^n$, to 0 and apply in the next time step. In case that the value between two time steps is smaller than the set tolerance, we can consider that our system has reached the steady state. Time step Δt and time tolerance δ are set according to the observed case (they have to be altered for different values of ρ/Γ.

Another important note is the calculation of discretization coefficients. Since all the coefficients

described in previous sections are constant in time, except the coefficient b_P, since it contains the quantity ϕ_P^n at the previous time step that changes at each iteration. We can improve the code performance by dividing the coefficients b_P into two parts: $\phi_P^n \times b_P^{const}$, where b_P^{const} is the constant part of the coefficient b_P. Hence, when solving the discretization equations, we will simply multiply the quantity ϕ_P^n at the previous time step by the constant part of the coefficient b_P.

3.3 Results

Our goal was to reproduce the results given in *Figure 6* of the paper. Furthermore, we want to further validate the results of the outlet by comparing them with the ones from the same paper, that are given in Table 1 and 2.

3.3.1 Results obtained using the Central Difference Scheme (CDS)

The color maps obtained using the CDS convective schemes are as follows:

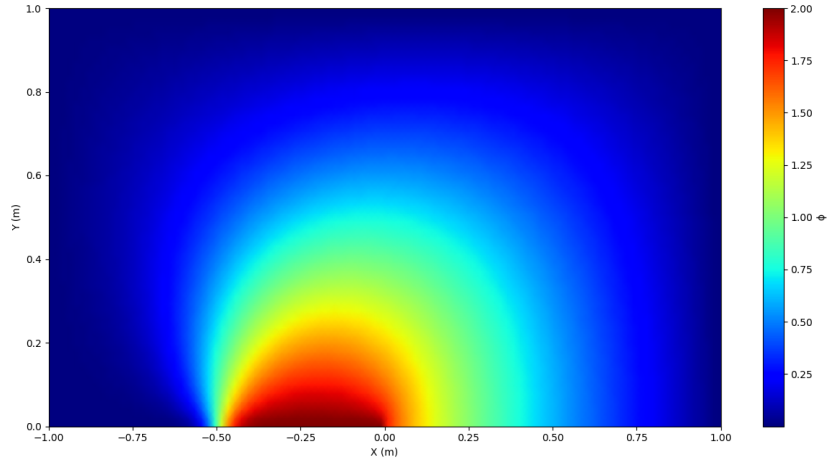

Figure 7: a) Color map of ϕ at steady state for $\rho/\Gamma = 10$ using the CDS convective scheme

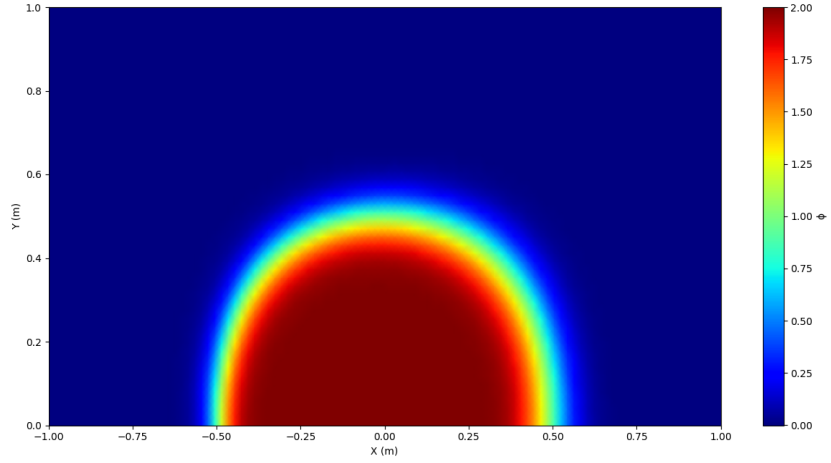

Figure 8: b) Color map of ϕ at steady state for $\rho/\Gamma = 1000$ using the CDS convective scheme

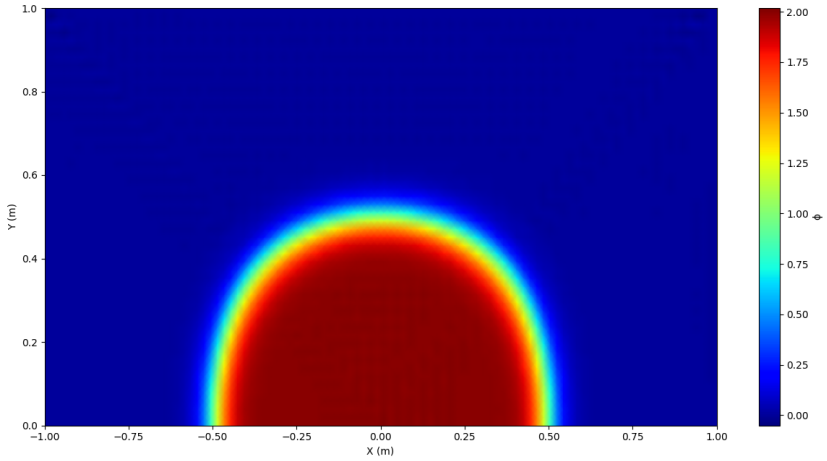

Figure 9: c) Color map of ϕ at steady state for $\rho/\Gamma = 1\ 000\ 000$ using the CDS convective scheme

If we compare Figures 7, 8, and 9, we can see that they closely resemble the results given by the reference paper in *Figure 6*. This confirms the right approach in our simulation. Time tolerance δ used in these simulations is $\delta = 10^{-6}$ for all three cases. The time step for $\rho/\Gamma = 10$ and $\rho/\Gamma = 1000$ was $\Delta t = 10^{-3}\ s$, while for the case $\rho/\Gamma = 1\ 000\ 000$, time step of $\Delta t = 10^{-5}\ s$ was used.

Next, we will compare the results obtained at the outlet with the ones given by the reference, which is presented in the following table:

Table 1: Numerical results at the outlet for different ratios of ρ/Γ; reference results and obtained results using the CDS convective scheme

Position x **x**	**ρ/Γ = 10**	**Results**	**ρ/Γ = 10000**	**Results**	**ρ/Γ = 1 000 000**	**Results**
0.0	1.989	1.871	2.0000	1.9999	2.000	2.009
0.1	1.402	1.383	1.9990	1.9999	2.000	1.999
0.2	1.146	1.136	1.9997	1.9998	2.000	2.000
0.3	0.946	0.940	1.9850	1.9917	1.999	2.000
0.4	0.775	0.771	1.8410	1.8221	1.964	1.961
0.5	0.621	0.618	0.9510	0.9673	1.000	1.000
0.6	0.480	0.478	0.1540	0.1510	0.036	0.039
0.7	0.349	0.348	0.0010	0.0062	0.001	0.001
0.8	0.227	0.227	0.0000	0.0001	0.000	0.000
0.9	0.111	0.111	0.0000	0.0000	0.000	0.000
1.0	0.000	0.000	0.0000	0.0000	0.000	-0.001

To better understand the table above, we can plot the obtained results and compare them to the reference:

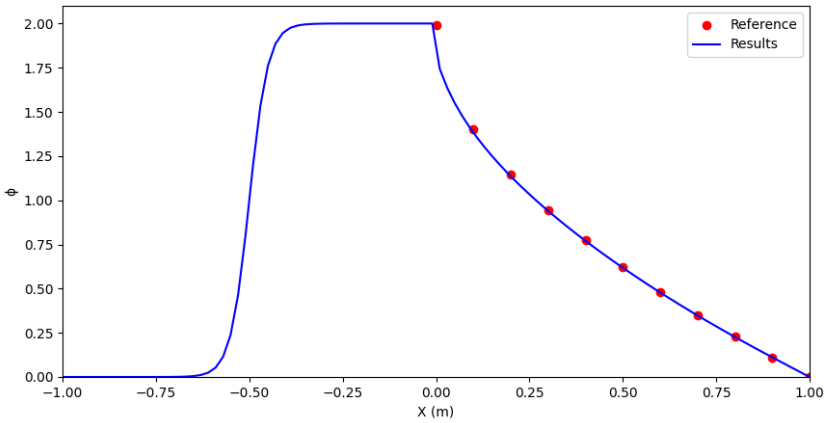

Figure 10: a) Plots of ϕ at the outlet, compared to the reference data, for $\rho/\Gamma = 10$, using the CDS convective scheme

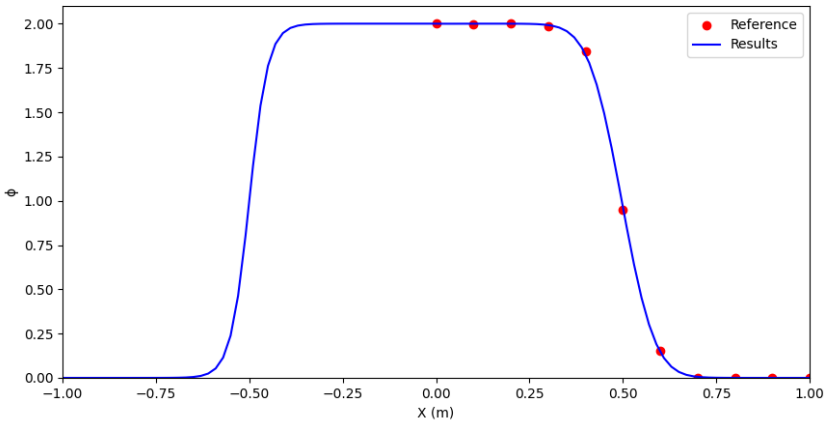

Figure 11: b) Plots of ϕ at the outlet, compared to the reference data, for $\rho/\Gamma = 1000$, using the CDS convective scheme

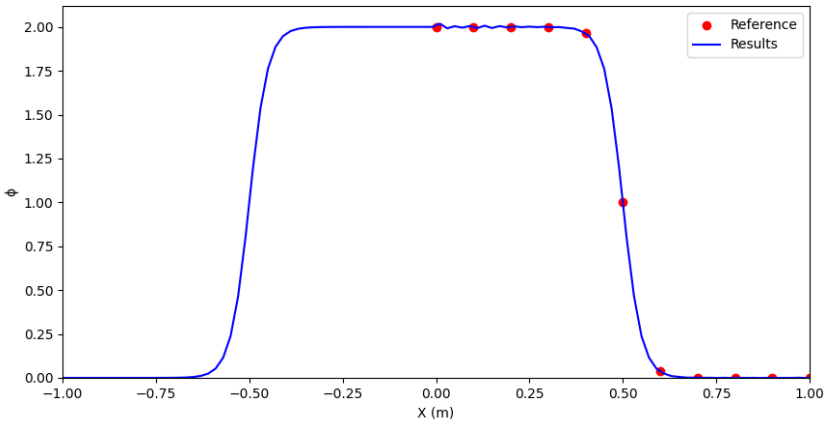

Figure 12: c) Plots of ϕ at the outlet, compared to the reference data, for $\rho/\Gamma = 1\ 000\ 000$, using the CDS convective scheme

From the figures above, we can observe the behaviour of our simulations. Results obtained by our simulations closely follow the reference results. This further validates the code.

Next, we want to observe the simulations using the Upwind Difference Scheme. The approach is the same, with the addition of calculating different discretization coefficients. According to the reference paper, CDS scheme causes the appearance of spurious velocities and oscilations, which can be observed in Figure 12, where we see various oscillations in the results. When using a bigger time step, for example $\Delta t = 10^{-4}$, the instabilities were even greater. Hence, the results are deeply influenced by the choice of time step. On the other hand, the reference paper states that UDS scheme may cause numerical diffusion, which will be further observed in the following section.

3.3.2 Results obtained using the Upwind Difference Scheme (UDS)

Similarly to the results obtained using the Central Difference Scheme, we want to compare our results to the reference results. Hence, the color maps obtained using the UDS convective scheme are as follows:

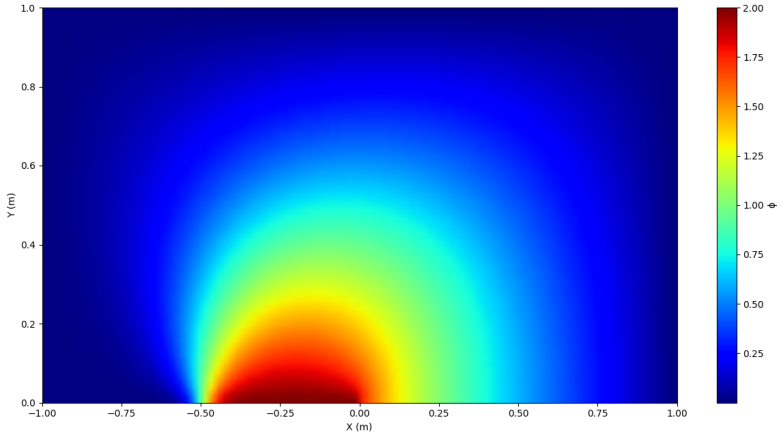

Figure 13: a) Color map of ϕ at steady state for $\rho/\Gamma = 10$ using the UDS convective scheme

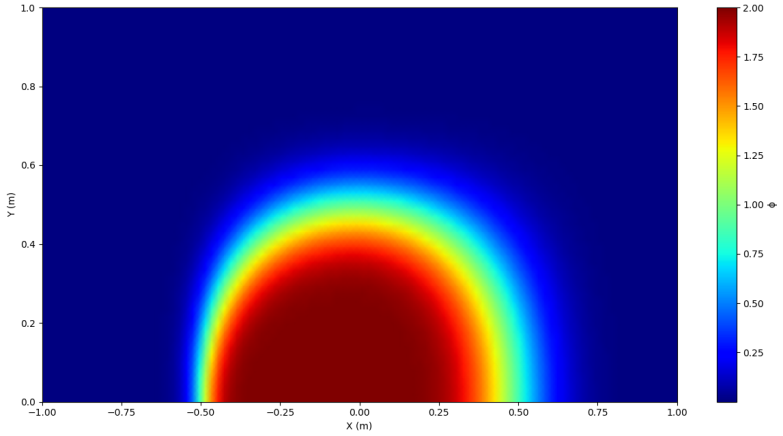

Figure 14: b) Color map of ϕ at steady state for $\rho/\Gamma = 1000$ using the UDS convective scheme

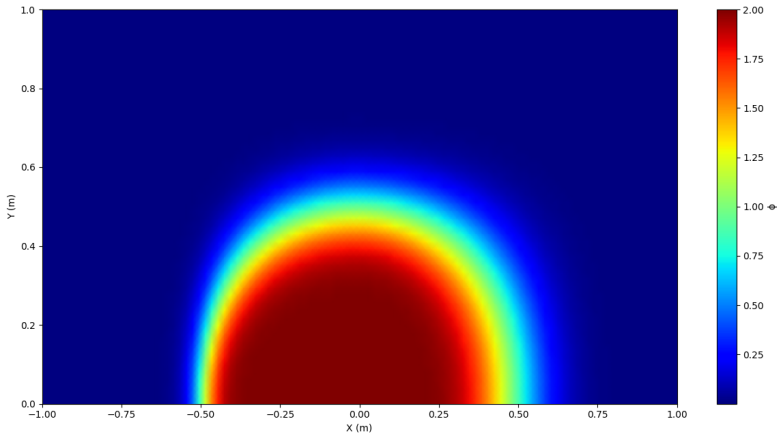

Figure 15: c) Color map of ϕ at steady state for $\rho/\Gamma = 1\ 000\ 000$ using the UDS convective scheme

The time tolerance used in all three cases was $\delta = 10^{-8}$ and a time step of $\Delta t = 10^{-4}$ s was also used for all three cases. Strict tolerance was imposed considering numerically diffusive nature of the UDS convective scheme. Using more relaxed tolerances results in incorrect results, since the steady state is not properly reached in those cases. These results closely resemble the ones given by the problem statement. However, to further validate our simulations, we need to compare the numerical results with the reference.

The following table compares the reference values given by the problem statement, and the values obtained in our simulations:

Table 2: Numerical results at the outlet for different ratios of ρ/Γ; reference results and obtained results using the UDS convective scheme

Position x **x**	**ρ/Γ = 10**	**Results**	**ρ/Γ = 10000**	**Results**	**ρ/Γ = 1 000 000**	**Results**
0.0	1.989	1.870	2.0000	2.0000	2.000	2.000
0.1	1.402	1.377	1.9990	1.9993	2.000	2.000
0.2	1.146	1.128	1.9997	1.9843	2.000	1.995
0.3	0.946	0.932	1.9850	1.8690	1.999	1.919
0.4	0.775	0.764	1.8410	1.4961	1.964	1.563
0.5	0.621	0.613	0.9510	0.8978	1.000	0.906
0.6	0.480	0.475	0.1540	0.3684	0.036	0.330
0.7	0.349	0.347	0.0010	0.0959	0.001	0.069
0.8	0.227	0.227	0.0000	0.0143	0.000	0.007
0.9	0.111	0.112	0.0000	0.0010	0.000	0.000
1.0	0.000	0.000	0.0000	0.0000	0.000	0.000

Once again, we can plot the obtained results of the UDS scheme as a function of position x at the bottom wall, for all three given cases of ρ/Γ ratio, and compare them with the reference results for better understanding:

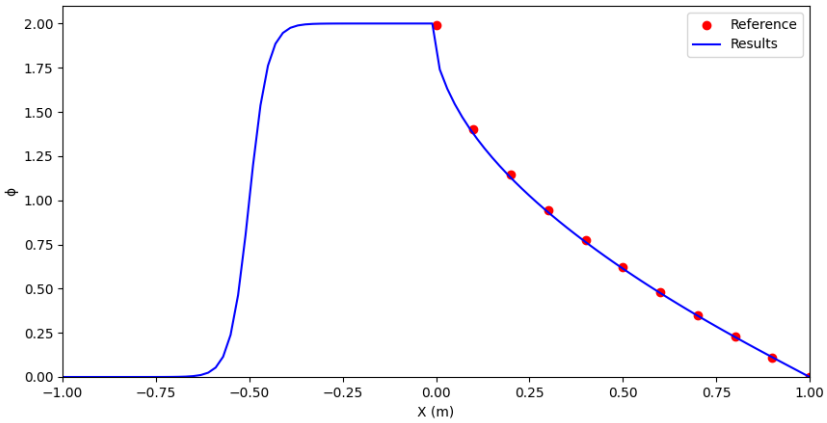

Figure 16: a) Plot of ϕ at the outlet, compared to the reference data, for $\rho/\Gamma = 10$, using the UDS convective scheme

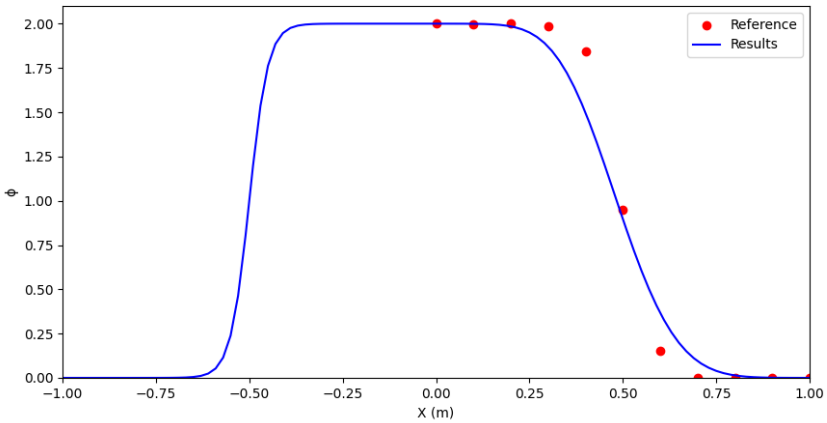

Figure 17: b) Plot of ϕ at the outlet, compared to the reference data, for $\rho/\Gamma = 1000$, using the UDS convective scheme

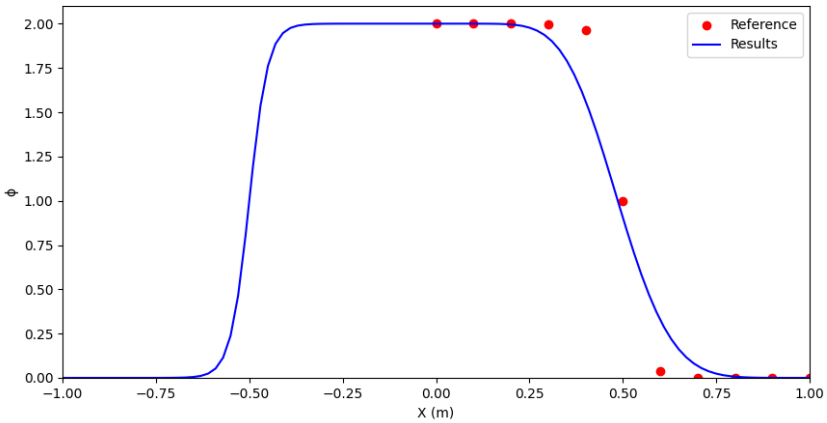

Figure 18: c) Plot of ϕ at the outlet, compared to the reference data, for $\rho/\Gamma = 1\ 000\ 000$, using the UDS convective scheme

3.4 Discussion

Previous sections provided results obtained by our simulations using the Central Difference Scheme and Upwind Difference Scheme. Both methods follow the same global algorithm, with different discretization coefficients. From the results, we notice multiple instances of different behaviour between the two convective schemes.

Results obtained by the CDS scheme, given in Table 1, closely resemble the ones given by the reference. This can be seen graphically in Figures 10, 11, and 12. Time tolerance δ was set to 10^{-6} for all three cases, ensuring that the steady state is reached and to avoid unnecessarily stringent convergence criteria. However, the time step used for ratios $\rho/\Gamma = 10$ and $\rho/\Gamma = 1000$ was $\Delta t = 10^{-3}$ s, while for ratio $\rho/\Gamma = 1\ 000\ 000$ $\Delta t = 10^{-5}$ s. Using bigger time steps in the third case would result in results divergence or incorrect results, depending on the time step chosen. However, despite numerically correct results, using small time steps slows down the code significantly, and the question that arises is whether it is computationally viable to use such stringent criteria in this case. For smaller ratios ρ/Γ, diffusive term dominates, which makes the system more stable and allows fo usage of bigger time steps. Hence, the computational time in these cases is significantly smaller. On the other hand, for large ratios of ρ/Γ, convective term dominates. This directly impacts the stability of the simulation, since the used value of time step does not adhere to the stability criteria (CFL, etc.). Hence, the simulation diverges and eventually blows up.

On the other hand, results obtained by the UDS scheme differ significantly from the ones obtained by the reference. This can be attributed to the numerical diffusion of the UDS scheme. However, the UDS scheme has some advantages over CDS scheme; despite the fact that we needed to impose stricter time convergence criteria of $\delta = 10^{-8}$, we could use a large time step for all three cases. The time step used in all three cases is $\Delta t = 10^{-4}$ s. This large time step allows for faster simulation and significantly reduces computational time. A graphical example of why strict time convergence criteria is required is presented in the following figure:

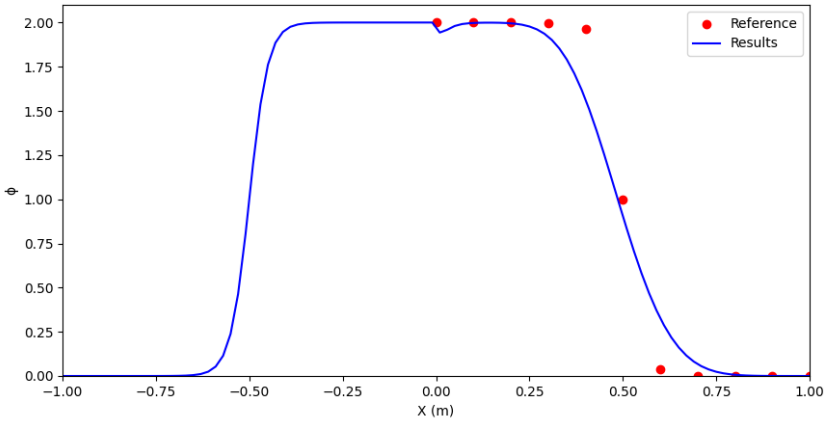

Figure 19: Plot of ϕ at the outlet, compared to the reference data, for $\rho/\Gamma = 1\ 000\ 000$, using the UDS convective scheme and time convergence criterion of $\delta = 10^{-5}$

The figure above uses a less strict time convergence criteria of $\delta = 10^{-5}$. While the computational time is significantly faster, the results obtained have clearly not yet reached the steady state, since these results differ from the ones obtained by a strict time convergence criterion, given in Figure 18. In case of using even bigger time tolerances, the results would contain even bigger oscillations, clearly proving that the steady state has not yet been reached.

4 Conclusions

In this assignment, we have dealt with the numerical solution of the Convection-Diffusion equation, with an emphasis on the practical implementation of the CD equation on the proposed Smith-Hutton case. The goal of this assignment was to understand the nature of the CD equation, and to create a program which would allow us to implement the Smith-Hutton case. Furthermore, we wanted to compare two different convective schemes, the Central Difference Scheme and the Upwind Difference Scheme.

Both convective schemes have their advantages and disadvantages; while the CDS scheme is second-order accurate, it is prone to stability problems. Hence, it requires small time steps, especially for convection-dominated cases. On the other hand, the UDS scheme, despite being first-order accurate, is more stable and allows for usage of large time steps, even for highly convection-dominated cases. However, the UDS scheme is numerically diffusive and it does not produce the results of desired accuracy like the CDS scheme does. Furthermore, the UDS scheme requires stricter time convergence criteria, since using less strict time steps might result in inaccurate results.

The choice of the convective scheme depends on the desired accuracy and computational time. Finally, a balance must be found between the two to produce optimal results.

5 Future work

To additionally enhance the code, we may implement various classes and functions, which will allow us to simply interchange between the schemes and easily implement them in the code. Furthermore, by creating different classes, we can create other types of meshes, and simply choose the type of mesh we want to use.

6 Numerical solution of the Convection-Diffusion equation - Code

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <algorithm>
#include <setjmp.h>
#include <iomanip>

using namespace std;

// Domain length
double domainLength(vector<double> domainCorners) {
  return domainCorners[1]-domainCorners[0];
}

// Grid size
double gridSize(double length, int N){
  return length/double(N);
}

// Position of control volumes
vector<double> cvPosition(int N, double d, vector<double> domainCorners){
  vector<double> posCV(N+2, 0.0);
  posCV[0] = domainCorners[0];
  posCV[N+1] = domainCorners[1];
  for(int i=1; i<=N; i++){
    posCV[i]=domainCorners[0]+i*d;
  }
  return posCV;
}

// Position of nodes
vector<double> nodePosition(vector<double> posCV, int N, vector<double>
↪  domainCorners){
  vector<double> posP(N+2, 0.0);
  posP[0] = domainCorners[0];
  posP[N+1] = domainCorners[1];
   for(int i=1; i<=N+1; i++){
     posP[i]=(posCV[i]+posCV[i-1])/2.0;
   }
  return posP;
}

double velocityU(double X, double Y) {
    double u=2*Y*(1-X*X);
    return u;
}

double velocityV(double X, double Y) {
    double v=-2*X*(1-Y*Y);
    return v;
}

//Numerical Solution of the Convection-Diffusion Equation
```

```
int main(){
  double PHI_o=0.0;                 // Initial conditions
  double alpha=10.0;                // Alpha coefficient
  double gamma=10.0;                // Ratio between rho and GAMMA

  int N=100, M=50;                  // Number of control volumes; mesh

  //Time definition
  double dt=1.0e-5;
    double time=0.0;

  vector<double> domainX = {-1, 1}, domainY = {0, 1};

  double lengthX = domainLength(domainX), lengthY = domainLength(domainY);

  double dx=gridSize(lengthX,N), dy=gridSize(lengthY,M);

  vector<double> xCV=cvPosition(N, dx, domainX), yCV=cvPosition(M, dy, domainY);
  vector<double> xP=nodePosition(xCV, N, domainX), yP=nodePosition(yCV, M, domainY);

  //Solver definition
  double timeTolerance=1e-6;        // Steady-state convergence
  double timeChange=0.0;            // Maximum difference between previous and current
  ↪  time step

  // Preallocate coefficients
  double aE[N+2][M+2], aW[N+2][M+2], aS[N+2][M+2], aN[N+2][M+2], bP[N+2][M+2],
  ↪  aP[N+2][M+2];

  // Initialize property  vectors
  double PHIo[N+2][M+2];     // Initial property map
    for (int i = 0; i <= N+1; i++) {
        for (int j = 0; j <= M+1; j++) {
            PHIo[i][j] = PHI_o;
        }
    }

    // Boundary conditions for PHI - Stays constant in time
    // Upper wall
    for (int i=1; i<=N;i++){
        PHIo[i][M+1]=1-tanh(alpha);
    }

    // Left wall
    for (int j=1;j<=M+1;j++){
        PHIo[0][j]=1-tanh(alpha);
    }

    // Right wall
    for (int j=1; j<=M+1;j++){
        PHIo[N+1][j]=1-tanh(alpha);
    }

    // Inlet and outlet - Initial conditions; Inlet will remain constant in time
    for (int i=0; i<=N+1;i++) {
        double x=xP[i];
        if (x<=0) {
            PHIo[i][0]=1+tanh((2.0*x+1)*alpha);
```

```
    } else {
        PHIo[i][0]=PHIo[i][1];
    }
}

double PHIpn[N+2][M+2];       // Initialize property vector at the previous time
↪  step t(n) at a node P
for (int i = 0; i <= N+1; i++) {
    for (int j = 0; j <= M+1; j++) {
        PHIpn[i][j] = PHIo[i][j];
    }
}

double PHIpn_1[N+2][M+2];
for (int i = 0; i <= N+1; i++) {
    for (int j = 0; j <= M+1; j++) {
        PHIpn_1[i][j] = PHIpn[i][j];
    }
}

// Inner coefficients
      for (int i=2; i<N;i++){
        for (int j=2; j<M;j++){
            double ue=velocityU(xCV[i],yP[j]);
            double uw=velocityU(xCV[i-1], yP[j]);
            double vn=velocityV(xP[i],yCV[j]);
            double vs=velocityV(xP[i], yCV[j-1]);

            aE[i][j]=-gamma*ue/(2.0*dx)+1.0/(dx*dx);
            aW[i][j]= gamma*uw/(2.0*dx)+1.0/(dx*dx);
            aN[i][j]=-gamma*vn/(2.0*dy)+1.0/(dy*dy);
            aS[i][j]= gamma*vs/(2.0*dy)+1.0/(dy*dy);
            bP[i][j]=(-gamma*ue/(2.0*dx)-2.0/(dx*dx)+gamma*uw/(2.0*dx) -
                      gamma*vn/(2.0*dy)-2.0/(dy*dy)+gamma*vs/(2.0*dy)+gamma/dt);
            aP[i][j]=gamma/dt;
        }
    }

    // Upper boundary nodes
    for (int i=2; i<=N-1;i++){
        double ue=velocityU(xCV[i],yP[M]);
        double uw=velocityU(xCV[i-1], yP[M]);
        double vn=velocityV(xP[i],yCV[M]);
        double vs=velocityV(xP[i], yCV[M-1]);

        aE[i][M]=-gamma*ue/(2.0*dx)+1.0/(dx*dx);
        aW[i][M]= gamma*uw/(2.0*dx)+1.0/(dx*dx);
        aN[i][M]=-gamma*vn/dy+2.0/(dy*dy);
        aS[i][M]= gamma*vs/(2.0*dy)+1.0/(dy*dy);
        bP[i][M]=(-gamma*ue/(2.0*dx)-2.0/(dx*dx)+gamma*uw/(2.0*dx) -
                  3.0/(dy*dy)+gamma*vs/(2.0*dy)+gamma/dt);
        aP[i][M]=gamma/dt;
      }

    // Left boundary nodes
      for (int j=2; j<=M;j++) {
          double ue=velocityU(xCV[1],yP[j]);
          double uw=velocityU(xCV[0], yP[j]);
```

```
          double vn=velocityV(xP[1],yCV[j]);
          double vs=velocityV(xP[1], yCV[j-1]);

          aE[1][j]=-gamma*ue/(2.0*dx)+1.0/(dx*dx);
          aS[1][j]= gamma*vs/(2.0*dy)+1.0/(dy*dy);
          aW[1][j]= gamma*uw/dx+2.0/(dx*dx);
          aP[1][j]=gamma/dt;
          if (j==M) {
              aN[1][j]=-gamma*vn/dy+2.0/(dy*dy);
              bP[1][j]=(-gamma*ue/(2.0*dx)-3.0/(dy*dy) -
                   3.0/(dx*dx)+gamma*vs/(2.0*dy)+gamma/dt);
          } else {
              aN[1][j]=-gamma*vn/(2.0*dy)+1.0/(dy*dy);
              bP[1][j]=(-gamma*ue/(2.0*dx)-2.0/(dy*dy)-gamma*vn/(2.0*dy) -
                   3.0/(dx*dx)+gamma*vs/(2.0*dy)+gamma/dt);
          }
        }

    // Right boundary nodes
    for (int j=2; j<=M;j++) {
        double ue=velocityU(xCV[N],yP[j]);
        double uw=velocityU(xCV[N-1], yP[j]);
        double vn=velocityV(xP[N],yCV[j]);
        double vs=velocityV(xP[N], yCV[j-1]);

        aE[N][j]=-gamma*ue/dx+2.0/(dx*dx);
        aS[N][j]= gamma*vs/(2.0*dy)+1.0/(dy*dy);
        aW[N][j]= gamma*uw/(2.0*dx)+1.0/(dx*dx);
        aP[N][j]=gamma/dt;
        if (j==M) {
            aN[N][j]=-gamma*vn/dy+2.0/(dy*dy);
            bP[N][j]=(gamma*uw/(2.0*dx)-3.0/(dy*dy) -
                 3.0/(dx*dx)+gamma*vs/(2.0*dy)+gamma/dt);
        } else {
            aN[N][j]=-gamma*vn/(2.0*dy)+1.0/(dy*dy);
            bP[N][j]=(gamma*uw/(2.0*dx)-2.0/(dy*dy)-gamma*vn/(2.0*dy) -
                 3.0/(dx*dx)+gamma*vs/(2.0*dy)+gamma/dt);
        }
    }

      // Bottom boundary nodes
      for (int i=1;i<=N;i++) {

          double ue=velocityU(xCV[i],yP[1]);
          double uw=velocityU(xCV[i-1], yP[1]);
          double vn=velocityV(xP[i],yCV[1]);
          double vs=velocityV(xP[i], yCV[0]);

          double x=xP[i];
          if (x<=0) {
              aE[i][1]=-gamma*ue/(2.0*dx)+1.0/(dx*dx);
              aN[i][1]=-gamma*vn/(2.0*dy)+1.0/(dy*dy);
              aS[i][1]= gamma*vs/dy+2.0/(dy*dy);
              aP[i][1]=gamma/dt;
              if (i==1){
                  aW[i][1]= gamma*uw/dx+2.0/(dx*dx);
                  bP[i][1]=(-gamma*ue/(2.0*dx)-3.0/(dx*dx) -
                       3.0/(dy*dy)-gamma*vn/(2.0*dy)+gamma/dt);
```

```
                } else {
                    aW[i][1]= gamma*uw/(2.0*dx)+1.0/(dx*dx);
                    bP[i][1]=(-gamma*ue/(2.0*dx)-2.0/(dx*dx)+gamma*uw/(2.0*dx) -
                         3.0/(dy*dy)-gamma*vn/(2.0*dy)+gamma/dt);
            }
            } else {
                aW[i][1]= gamma*uw/(2.0*dx)+1.0/(dx*dx);
                aN[i][1]=-gamma*vn/(2.0*dy)+1.0/(dy*dy);
                aS[i][1]= 0.0;
                aP[i][1]=gamma/dt;
                if (i==N) {
                    aE[i][1]=-gamma*ue/dx+2.0/(dx*dx);
                    bP[i][1]=(gamma*vs/dy-3.0/(dx*dx)+gamma*uw/(2.0*dx) -
                         1.0/(dy*dy)-gamma*vn/(2.0*dy)+gamma/dt);
                } else {
                    aE[i][1]=-gamma*ue/(2.0*dx)+1.0/(dx*dx);
                    bP[i][1]=(-gamma*ue/(2.0*dx)-2.0/(dx*dx)+gamma*uw/(2.0*dx) -
                         1.0/(dy*dy)-gamma*vn/(2.0*dy)+gamma*vs/dy+gamma/dt);
                }
            }
      }

// Start the time loop
do {
    time+=dt;
    cout<<"TIME = "<<time<<endl;
      timeChange=0.0;

          // Internal nodes
          for (int i=1;i<=N;i++){
              for (int j=1;j<=M;j++){
                  PHIpn_1[i][j]=(aE[i][j]*PHIpn[i+1][j]+aW[i][j]*PHIpn[i-1][j]+

                       ↪  aN[i][j]*PHIpn[i][j+1]+aS[i][j]*PHIpn[i][j-1]+PHIpn[i][j]*bP[i][j])/aP[i
              }
          }

    // Set new values for inlet/outlet
    // Inlet and outlet - Initial conditions; Inlet will remain constant in time
    for (int i=0; i<=N+1;i++) {
        double x=xP[i];
        if (x<=0) {
            continue;
        } else {
            PHIpn_1[i][0]=PHIpn_1[i][1];
        }
    }

      for (int i = 1; i < N ; i++) {
        for (int j = 1; j < M;  j++) {
            timeChange = max(timeChange, fabs(PHIpn_1[i][j]-PHIpn[i][j]));
        }
      }

    for (int i = 0; i <= N+1; i++) {
        for (int j = 0; j <= M+1; j++) {
            PHIpn[i][j] = PHIpn_1[i][j];
        }
```

```
    }
} while(timeChange>=timeTolerance);

  // Plotting data
  ofstream phiMap("PHI_data.csv");

  if (phiMap.is_open()) {
      for (int j = 0; j <= M + 1; j++) {
          for (int i = 0; i <= N + 1; i++) {
              phiMap << PHIpn[i][j];
              if (i < N + 1) phiMap << ",";
          }
          phiMap << "\n";
      }
      phiMap.close();
      cout << "Property data saved to 'PHI_data.csv'\n";
  } else {
      cerr << "Error: Unable to open file for writing\n";
  }

  ofstream phiOutlet("PHI_outlet.csv");
  if (phiOutlet.is_open()) {
      for (int i = 0; i <= N + 1; i++) {
          double x=xP[i];
          phiOutlet << x << "," << PHIpn[i][0] << "\n";
      }
      phiOutlet.close();
  }

  vector<double> target_x = {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0};

  ofstream phiInterpolated("PHI_interpolated.txt");
  phiInterpolated << std::fixed << std::setprecision(3); // Set decimal precision

  if (phiInterpolated.is_open()) {
      for (double x_target : target_x) {
          int i_left = 0;
          while (i_left < N && xP[i_left] < x_target) {
              i_left++;
          }

          if (i_left == 0) {
              phiInterpolated << x_target << " " << PHIpn[i_left][0] << "\n";
              continue;
          }

          int i_right = i_left;
          i_left--; // Now xP[i_left] < x_target < xP[i_right]

          double x1 = xP[i_left], x2 = xP[i_right];
          double phi1 = PHIpn[i_left][0], phi2 = PHIpn[i_right][0];

          double phi_interp = phi1 + (phi2 - phi1) * (x_target - x1) / (x2 - x1);

          phiInterpolated << x_target << " " << phi_interp << "\n";
      }
      phiInterpolated.close();
      cout << "Interpolated PHI values saved to 'PHI_interpolated.txt'\n";
```

```
    } else {
        cerr << "Error: Unable to open file for writing\n";
    }

  return 0;
}
```

6.1 UDS scheme coefficients

```
// Function for calculation of UDS scheme factors
    void calculateUDSCoefficients(double ue, double uw, double vn, double vs,
    double &fe, double &fw, double &fn, double &fs) {
    fe = (ue > 0) ? 0.0 : 1.0;
    fw = (uw > 0) ? 1.0 : 0.0;
    fn = (vn > 0) ? 0.0 : 1.0;
    fs = (vs > 0) ? 1.0 : 0.0;
}

// Coefficients
    for (int i=1; i<=N;i++){
        for (int j=1; j<=M;j++){
            double ue=velocityU(xCV[i],yP[j]);
            double uw=velocityU(xCV[i-1], yP[j]);
            double vn=velocityV(xP[i],yCV[j]);
            double vs=velocityV(xP[i], yCV[j-1]);

            calculateUDSCoefficients(ue, uw, vn, vs, fe, fw, fn, fs);

            aE[i][j]=-gamma*ue*fe/dx+1.0/((xP[i+1]-xP[i])*dx);
            aW[i][j]= gamma*uw*fw/dx+1.0/((xP[i]-xP[i-1])*dx);
            aN[i][j]=-gamma*vn*fn/dy+1.0/((yP[j+1]-yP[j])*dy);
            aS[i][j]= gamma*vs*fs/dy+1.0/((yP[j]-yP[j-1])*dy);
            bP[i][j]=(-gamma*ue*(1-fe)/dx-1.0/((xP[i+1]-xP[i])*dx)+
            gamma*uw*(1-fw)/dx-1.0/((xP[i]-xP[i-1])*dx) -
                        gamma*vn*(1-fn)/dy-1.0/((yP[j+1]-yP[j])*dy)+
                        gamma*vs*(1-fs)/dy-1.0/((yP[j]-yP[j-1])*dy)+gamma/dt);
            aP[i][j]=gamma/dt;
        }
    }
```

7 Matplotlib file for results plotting

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

data = np.loadtxt("PHI_dataUDS.csv", delimiter=",")

x = np.linspace(-1.0, 1.0, data.shape[1])
y = np.linspace(0.0, 1.0, data.shape[0])

X, Y = np.meshgrid(x, y)

plt.figure(figsize=(15, 7.5))
plt.pcolormesh(X, Y, data, shading="gouraud", cmap="jet")
plt.colorbar(label="")
plt.xlabel("X (m)")
plt.ylabel("Y (m)")

plt.savefig('PHI.png')

plt.show()

#Plotting outlet
referenceData=pd.read_csv("convection_diffusion_results.csv")
xData=referenceData["Position x"]
gamma10=referenceData["gamma=1000000"]

results = pd.read_csv("PHI_outletUDS.csv", header=None, names=["x", "PHI"])

plt.figure(figsize=(10, 5))
plt.scatter(xData, gamma10, color='r', marker='o', label="Reference")
plt.plot(results["x"], results["PHI"], linestyle='-', color='b', label="Results")
plt.xlim([-1.0, 1.0])
plt.ylim(0.0)
plt.xlabel("X (m)")
plt.ylabel("")
plt.legend()
plt.show()
```