MASTER'S DEGREE IN AERONAUTICAL ENGINEERING

Generic convection-diffusion transport equation

COMPUTATIONAL ENGINEERING

Authors: Ricard Arbat Carandell

Professor: Carlos David Perez Segarra

Escola Superior d'Enginyeria Industrial, Aeroespacial i Audiovisual de Terrassa (ESEIAAT)

Dated: October 20, 2024

Contents

Index of Figures

1 Convection and diffusion

In fluid dynamics, **convection** and **diffusion** describe two mechanisms through which properties such as heat, mass, or momentum are transported within a fluid. Convection refers to the transport of these properties through the bulk motion of the fluid, while diffusion refers to their transport due to molecular motion. Now, some of the most relevant equations and parameters used for modelling this effects will be analyzed:

1.1 Péclet number

The **Péclet number** (Pe) is a dimensionless quantity that characterizes the relative importance of convection versus diffusion in a flow. It is defined as:

$$Pe = \frac{\text{Convective Transport}}{\text{Diffusive Transport}} = \frac{\rho v L}{\Gamma} \tag{1}$$

where L is the characteristic length scale of the problem, and Γ is the diffusion coefficient.

For large Pe, convection dominates, while for small Pe, diffusion dominates. The behavior of numerical schemes can vary significantly depending on the value of Pe, with some schemes performing better for convection-dominated flows and others being more suited to diffusion-dominated flows.

1.2 The convection-diffusion equation

The equation describes the transport of any scalar quantity ϕ (e.g., temperature, concentration, or momentum) due to both convection and diffusion. Starting from the conservation principles and applying the simplifications discussed, we arrive at the generic form of the convection-diffusion equation:

$$\frac{\partial(\rho\phi)}{\partial t} + \nabla \cdot (\rho \mathbf{v}\phi) = \nabla \cdot (\Gamma_\phi \nabla\phi) + S_\phi \tag{2}$$

In this equation:

- ϕ represents a scalar quantity (e.g., temperature, species concentration, or velocity component).
- Γ_ϕ is the diffusion coefficient for the scalar ϕ (e.g., thermal diffusivity for temperature or molecular diffusivity for concentration).
- S_ϕ represents any external source or sink of ϕ (e.g., heat sources, chemical reactions).

The left-hand side of the equation represents unsteady ($\partial(\rho\phi)/\partial t$) and convective ($\nabla \cdot (\rho \mathbf{v}\phi)$) transport. The right-hand side represents diffusive transport ($\nabla \cdot (\Gamma_\phi \nabla\phi)$) and any sources or sinks.

For steady-state problems (i.e., $\partial/\partial t = 0$), the equation simplifies to:

$$\nabla \cdot (\rho \mathbf{v}\phi) = \nabla \cdot (\Gamma_\phi \nabla\phi) + S_\phi \tag{3}$$

1.3 Numerical discretization of the convection-diffusion equation

To solve the convection-diffusion equation numerically, we must discretize the spatial and temporal derivatives, it this case using a finite voluem method (FVM)

In the FVM, the domain is divided into small control volumes (CVs), and the governing equations are integrated over each CV. Integrating the convection-diffusion equation over the control volume yields:

$$\int_{V_p} \frac{\partial(\rho\phi)}{\partial t} dV + \int_{S_p} \mathbf{n} \cdot (\rho \mathbf{v}\phi) dS = \int_{S_p} \mathbf{n} \cdot (\Gamma_\phi \nabla\phi) dS + \int_{V_p} S_\phi dV \tag{4}$$

The integrals represent fluxes of ϕ through the control volume surfaces. The convection and diffusion terms are evaluated at the faces of the control volumes using appropriate numerical schemes. Considering a second order approach for ϕ, the equations can be further developed, yielding:

$$\begin{aligned}\frac{\rho_P\phi_P - \rho_P^0\phi_P^0}{\Delta t} V_P + F_e(\phi_E - \phi_P) - F_w(\phi_W - \phi_P) + F_n(\phi_N - \phi_P) - F_s(\phi_S - \phi_P) \\ = D_e(\phi_E - \phi_P) - D_w(\phi_P - \phi_W) + D_n(\phi_N - \phi_P) - D_s(\phi_P - \phi_S) + S_\phi C + S_\phi P\phi_P\end{aligned}$$

1.4 Resolution schemes

When discretizing the convection and diffusion terms, we use schemes that differ in terms of accuracy and stability. Three common schemes for discretizing the convective fluxes at control volume faces are:

1.4.1 Upwind difference scheme (UDS)

The UDS assumes that the value of ϕ at a control volume face is equal to the value of ϕ at the upstream node. This ensures numerical stability, particularly for high Péclet number flows, but it is only first-order accurate.

$$\phi_e = \begin{cases} \phi_P, & \text{if } F_e > 0 \\ \phi_E, & \text{if } F_e < 0 \end{cases} \tag{5}$$

Here, $F_e = \rho u_e S_e$ is the convective flux across the east face of the control volume.

1.4.2 Central difference scheme (CDS)

The CDS assumes that the value of ϕ at a face is the average of the values at the adjacent nodes. This scheme is second-order accurate, but can become unstable in high Péclet numbers.

$$\phi_e = \frac{\phi_P + \phi_E}{2} \tag{6}$$

1.4.3 Hybrid difference Scheme (HDS)

The HDS is a combination of the UDS and CDS schemes. It uses the CDS for low velocities (or low Péclet numbers) and switches to UDS for high velocities. This offers a balance between stability and accuracy.

1.4.4 Power-law difference scheme (PDS)

The Power-Law Difference Scheme (PDS) is based on the power-law formulation of the convection-diffusion equation. It offers a compromise between the stability of UDS in high convection regimes and the accuracy of CDS in low convection regimes, making it well-suited for flows where both convection and diffusion are significant.

The PDS is given by:

$$\phi_e = \frac{\phi_P}{(1 + 0.5|Pe_e|)} + \frac{\phi_E}{(1 + 0.5|Pe_e|)} \left(1 - \frac{0.5|Pe_e|}{1 + 0.5|Pe_e|}\right) \tag{7}$$

1.5 Generalized discretization equation

The general discretized convection-diffusion equation can be rearranged into a more usable form.

$$a_P \phi_P = a_E \phi_E + a_W \phi_W + a_N \phi_N + a_S \phi_S + b_P \tag{8}$$

Where:

$$a_E = D_e \cdot A(|P_e|) + \max(-F_e, 0)$$

$$a_W = D_w \cdot A(|P_w|) + \max(F_w, 0)$$

$$a_N = D_n \cdot A(|P_n|) + \max(-F_n, 0)$$

$$a_S = D_s \cdot A(|P_s|) + \max(F_s, 0)$$

$$a_P = a_E + a_W + a_N + a_S + \frac{\rho \Delta x \Delta y}{\Delta t}$$

$$b_P = \frac{\rho \Delta x \Delta y}{\Delta t} \phi_P^0 + S_P \Delta x \Delta y$$

In the above terms:

- F represents the convective fluxes across the east, west, north, and south faces, respectively. These are computed as $F = \rho u A$, where A is the area of the face.
- D represents the diffusive fluxes across the corresponding faces. These are computed as $D = \frac{\Gamma A}{d}$, where d is the distance between nodes.
- ϕ are the values of ϕ at the east, west, north, and south neighboring nodes.

The different schemes can be applied by using the equations found in Table I in place of the A functions of Equation 8

Scheme	Formula for A(|P|)
CDS	$1 - 0.5\|P\|$
UDS	1
HDS	$\max(0, 1 - 0.5(\|P\|))$
PDS	$\max(0, (1 - 0.5(\|P\|)^5))$

Table I. Numerical schemes available in function of the Péclet number

1.6 Resolution algorithm

1. Input parameters:
   - Physical properties: Velocity, channel size, initial value ϕ, time step Δt...
   - Numerical parameters: Converge criteria Δe, mesh cell number...
   - Other options: Scheme to be used (CDS, HDS, UDS, PDS...), exporting format, file name, storage folder...
2. Initial settings:
   - Mesh building for X, Y and t
   - Initial value of ϕ for t = 0
3. Iteration along the cells of the domain:
   - If fluid cell:
     - Speeds for the cell (u, v)
     - Mass flow terms (F)
     - Péclet number and scheme implementation $(A(|P|))$
     - Evaluation of the equation parameters $(a_p, a_e, a_w, a_n, a_s, b)$
     - Calculation of the next value (ϕ_p) using Gauss - Seidel

- If boundary cell:
  - Set boundary value (ϕ_p)

4. Error with last values. Is $|\phi_{old} - \phi_p < \Delta e|$?
   - Yes. Skip to 5 No. Go to 3 and $\phi_{old} = \phi_p$
5. According to parameters, go back to 2 and repeat for multiple cases of:
   - Schemes
   - Péclet numbers
   - Time step size
   - Mesh refinement
   - Type of flow
6. Exporting data to .csv files
7. Plotting using Matlab, and comparison with reference results

2 Problem definition

2.1 Diagonal flow

The first case encompasses a diagonal flow, as seen in Figure 1. The velocity of the flow is constant at an angle of 45º, and the base speed of the flow (V_0) is set to 1 m/s

$$u = V_0 cos(\theta) \quad v = V_0 sin(\theta)$$

As for the boundary conditions, walls above the diagonal are set to a higher variable $\phi_h = 2$, while the other ones to a lower value $\phi_l = 0$.

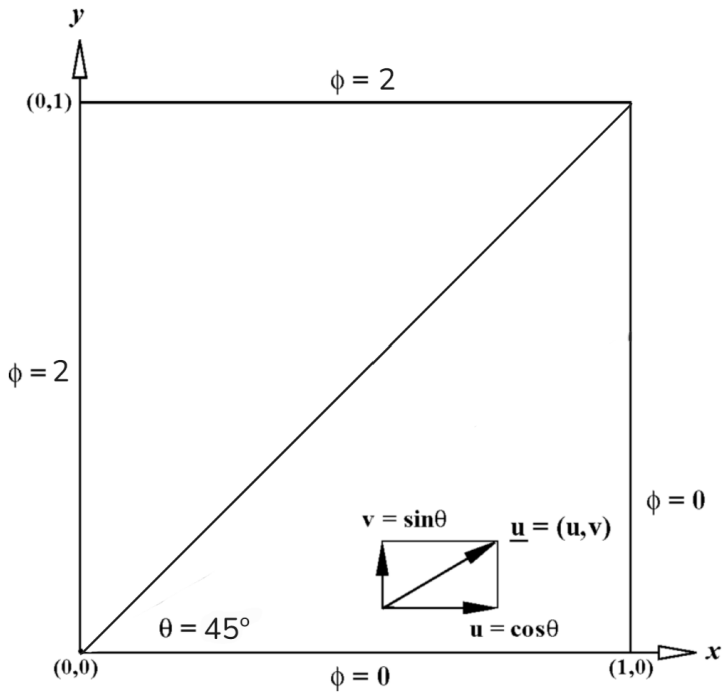


Figure 1. Diagonal flow diagram

2.2 Smith Hutton flow

The second case considers a Smith Hutton constant flow, as seen in Figure 2. The flow comes in from an inlet and follows a velocity field up to the exit. This field is defined as:

$$u = 2y(1 - x^2) \quad v = -2x(1 - y^2)$$

As for the boundary conditions, there are three different cases:

- **Inlet:** $\phi = 1 + tanh(10(2x + 1))$
- **Walls:** $\phi = 1 + tanh(10)$
- **Outlet:** $\frac{\delta\phi}{\delta y} = 0)$

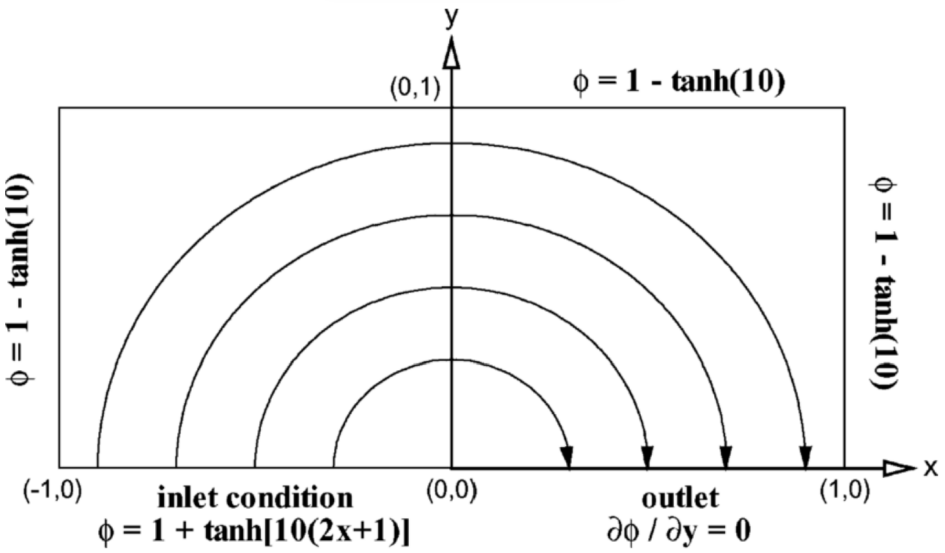


Figure 2. Smith Hutton flow diagram

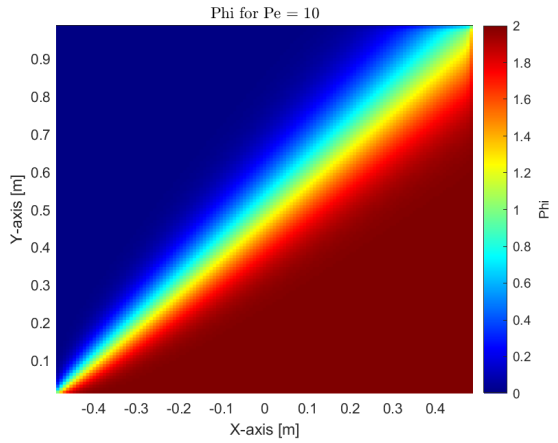
3 Results

3.1 Diagonal flow

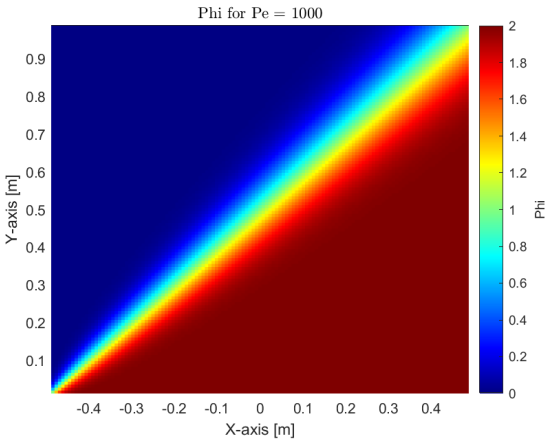
The diagonal flow was obtained as a first simple case, and it was used to check whether the solved worked as expected. The parameters used for the flow were:

- Cells: $M = 128 \;\; N = 128$
- Size: $H = 1m \;\; L = 1m$
- Time step size: $\Delta t = 0.01s$
- Convergence error: $\Delta e = 0.00001$
- Scheme: UDS

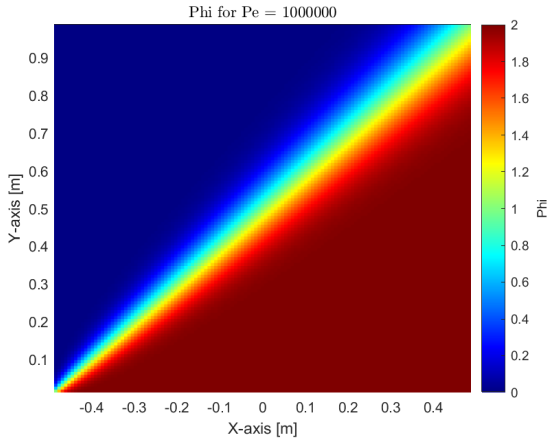
The flow has a clear division between zones of high phi (ϕ_h) and zones with a lower phi (ϕ_l). Also, for lower Péclet numbers, the division is smoother, but as it gets higher, the jump is faster and more abrupt.



(a) Diagonal flow for $Pe = 10$



(b) Diagonal flow for $Pe = 10^3$



(c) Diagonal flow for $Pe = 10^6$

Figure 3. Diagonal flow for multiple Péclet numbers obtained using the UDS scheme

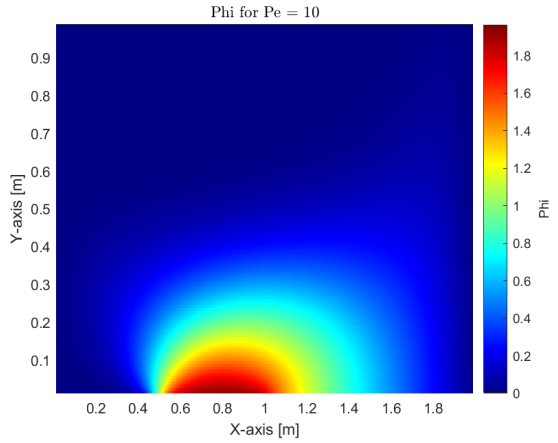
3.2 Smith-Hutton flow

The Smith-Hutton flow was obtained for a case with a relatively high mesh refinement. As it used an HDS scheme, time step size was not very relevant. The parameters used were:

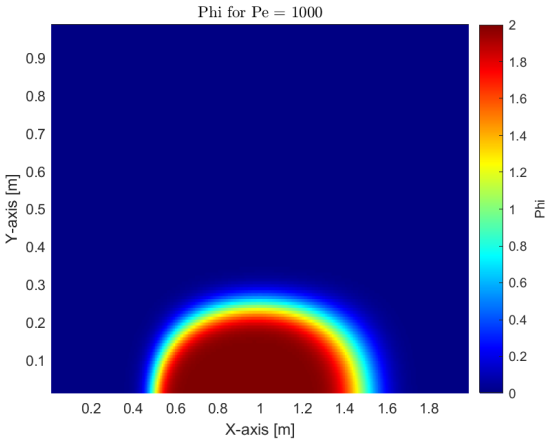
- Cells: $M = 128 \;\; N = 256$
- Convergence error: $\Delta e = 0.00001$
- Size: $H = 1m \;\; L = 2m$
- Time step size: $\Delta t = 0.001s$
- Scheme: HDS

As seen in Figure 4, the resulting 2D maps behave as expected. The value of ϕ defined at the inlet gets transported by the velocity field as expected, reaching the outlet at the end.

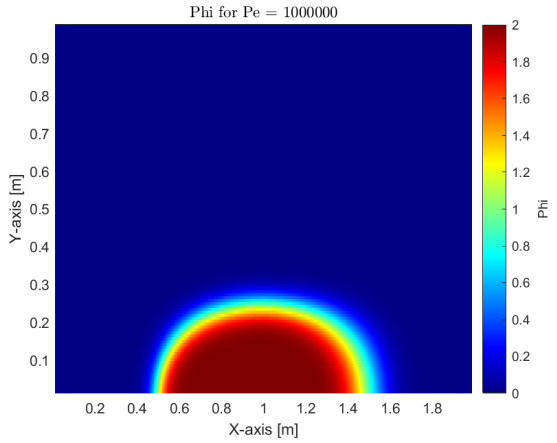
When the Péclet number increases, the change in ϕ becomes more abrupt and fast. This is due to the fact that at higher Péclet numbers, convection is the main transporter of ϕ, while at lower values, diffusion takes more importance.



(a) Smith-Hutton flow for $Pe = 10$



(b) Smith-Hutton flow for $Pe = 10^3$



(c) Smith-Hutton flow for $Pe = 10^6$

Figure 4. Smith-Hutton for multiple Péclet numbers

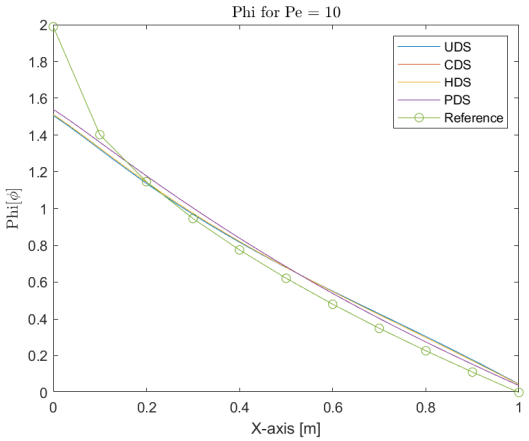
4 Verification

4.1 Schemes

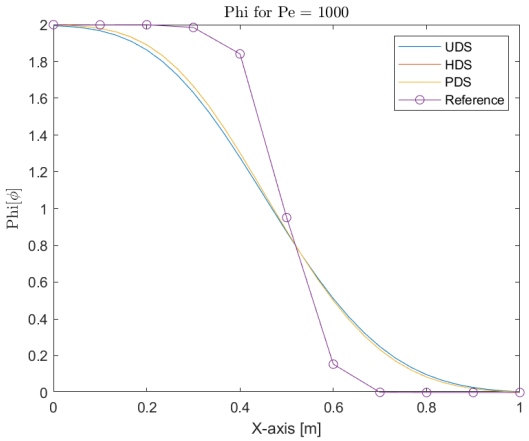
Firstly, the different schemes (UDS, CSD, PDS and HDS) were compared. All of them were compared using some simple parameters:

- Cells: $M = 32 \;\; N = 64$
- Time step size: $\Delta t = 0.01s$
- Size: $H = 1m \;\; L = 2m$
- Convergence error: $\Delta e = 0.0001$

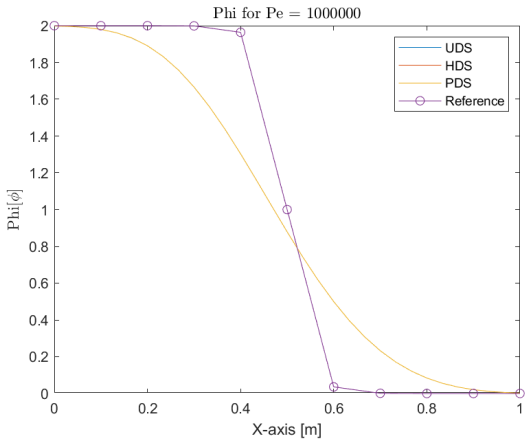
Two main things were observed: Firstly, all the solution schemes gave quite similar results, even though CDS became ustable al higher Pe numbers. And secondly, as the Pe gets higher, the solutions diverge more from the reference result. Both behaviors may mean that more mesh or time refinement are needed.



(a) Parameter (ϕ) at the outlet nodes for a Smith-Hutton flow, $Pe = 10$



(b) Parameter (ϕ) at the outlet nodes for a Smith-Hutton flow, $Pe = 10^3$



(c) Parameter (ϕ) at the outlet nodes for a Smith-Hutton flow, $Pe = 10^6$

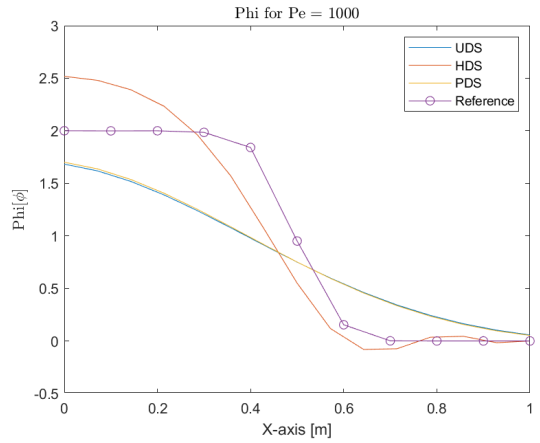
Figure 5. Comparison of ϕ values at the outlet for different schemes and values of Pe

4.2 Convergence to reference results

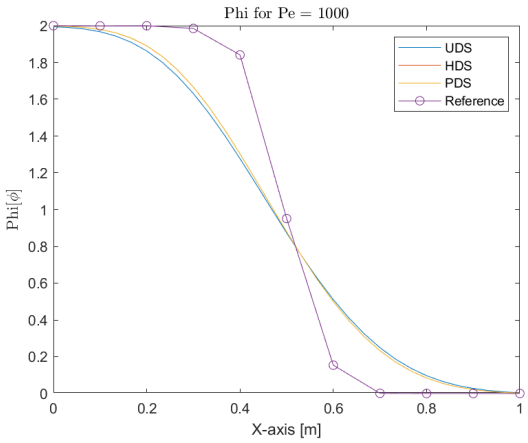
In order to analyze the convergence of the solutions to the reference result, some cases regarding mesh and time delta were studied. The convergence error for the next cases was set to $\Delta e = 0.0001$

4.2.1 Mesh size ($\delta x, \delta y$)

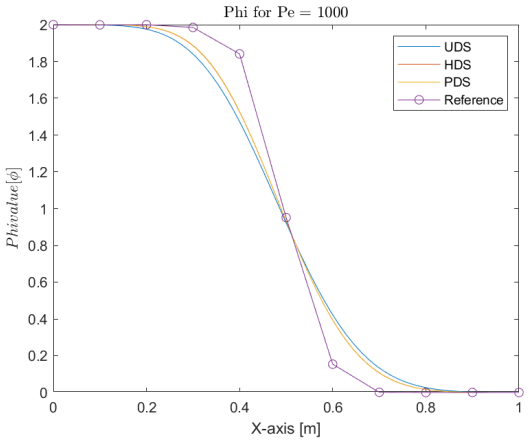
Firstly, the convergence with the mesh refinement was studied. It was seen how (See Figure 6), as the meshes get finer, the numerical approximation gets closer and closer to the reference result. The HDS scheme behaved particularly well, even at low mesh refinements. As it was seen in Figure 5, smaller Pe numbers did not need so much refinement, but as Pe got higher, the difference between the numerical and reference solutions also go higher.



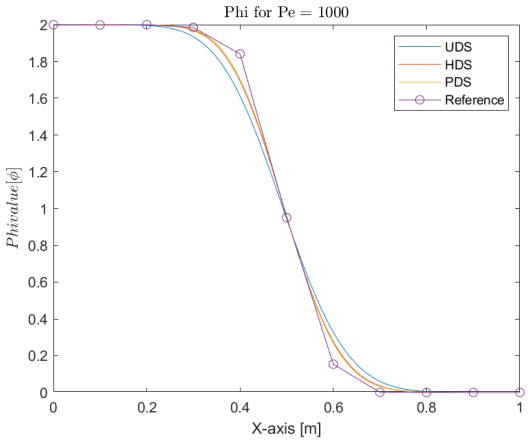
(a) ϕ at the outlet nodes for a Smith-Hutton flow, with $Pe = 10^3$ and $M = 16$ $N = 32$



(b) ϕ at the outlet nodes for a Smith-Hutton flow, with $Pe = 10^3$ and $M = 32$ $N = 64$



(c) ϕ at the outlet nodes for a Smith-Hutton flow, with $Pe = 10^3$ and $M = 64$ $N = 128$



(d) ϕ at the outlet nodes for a Smith-Hutton flow, with $Pe = 10^3$ and $M = 128$ $N = 256$

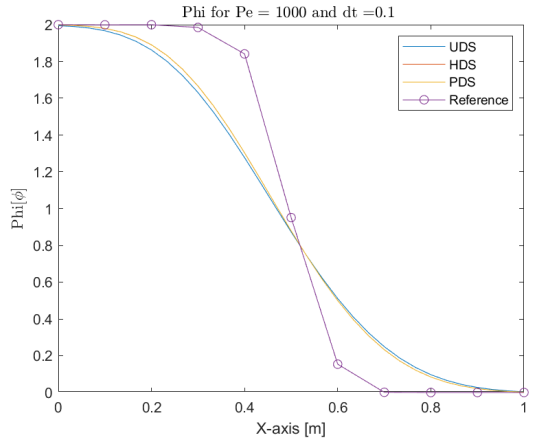
Figure 6. ϕ at the outlet for meshes with different refinement, ranging from M = 16 to M = 128 nodes

As the Péclet number was quite high at 1000, and the numerical parameters used were $\Delta t = 0.01s$ and $\Delta e = 0.0001$, the CDS solution was not stable and diverged. That's why it is not available in the plots. In the next section 4.2.2, this will be analyzed.

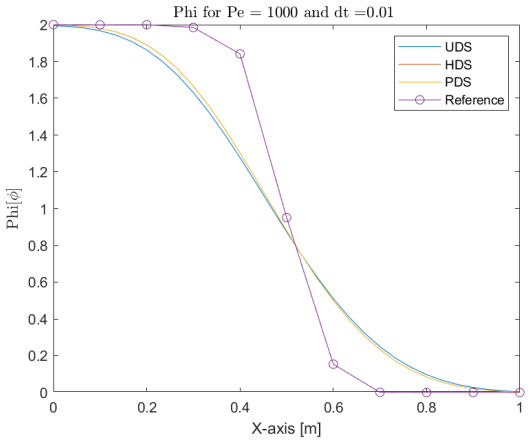
4.2.2 Time step size (Δt)

As for the time step size, some behaviors were also studied. It was observed that the CDS scheme could obtain great results with low mesh refinements, but at the cost of smaller time stepping (See Figure 7).

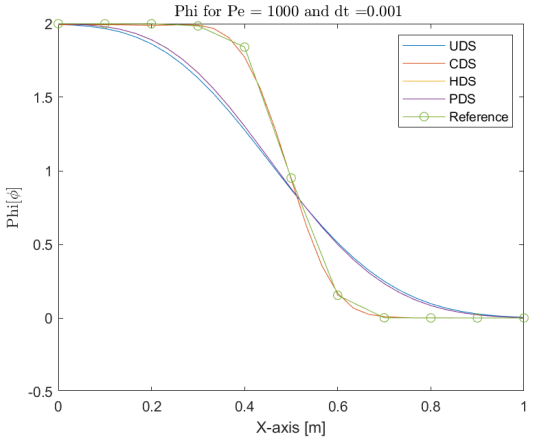
If the time step was too big, for example $\Delta t = 0.01$, the CDS scheme diverged, but as it got smaller, at values of around $\Delta t = 0.001$,even if $M = 32$, the CDS could yield great results, as in Figure 7c



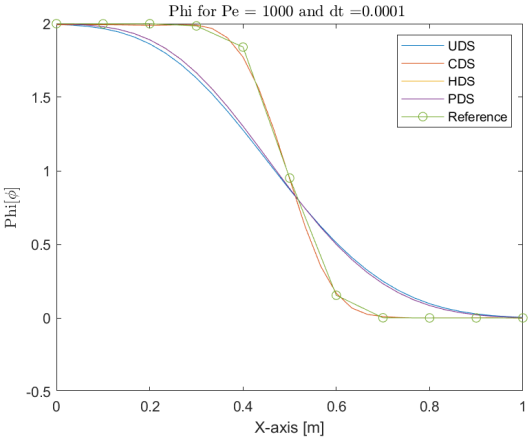
(a) ϕ at the outlet nodes for a Smith-Hutton flow, with $Pe = 10^3$, $M = 32$ and $\Delta t = 0.1$



(b) ϕ at the outlet nodes for a Smith-Hutton flow, with $Pe = 10^3$, $M = 32$ and $\Delta t = 0.01$



(c) ϕ at the outlet nodes for a Smith-Hutton flow, with $Pe = 10^3$, $M = 32$ and $\Delta t = 0.001$



(d) ϕ at the outlet nodes for a Smith-Hutton flow, with $Pe = 10^3$, $M = 32$ and $\Delta t = 0.0001$

Figure 7. ϕ at the outlet for different time refinements, ranging from $\Delta t = 0.1$ to $\Delta t = 0.0001$

References

[1] CTTC. (n.d.). *Course on numerical methods in heat transfer and fluid dynamics: Non-viscous flows*. (accessed: 23-09-2024).

[2] W3Schools. (n.d.). *Learn c++*. https://www.w3schools.com/cpp/ (accessed: 25-09-2024).

[3] Patankar, S. V. (1980). *Numerical heat transfer and fluid flow*. Hemisphere Publishing Corporation.

A C++ Code

The following code has been developed for the report. It allows defining the parameters for either a diagonal or a Smith-Hutton flow, solving it, and then exporting the data for plotting and verification in csv files.

A few Matlab files have also been used in order to plot the data obtained. However, these are not relevant for this report, so they will not be commented, but are available in the *.zip* folder.

A.1 Parameters

The *parameters.cpp* file includes many general parameters used along the code.

```
1
2
3  // Dimensions
4  int M = 128;                // number of columns
5  int N = 2 * M;              // number of rows
6  double H = 1;               // height of the channel
7  double L = 2 * H;           // length of the channel
8  const double dx = L / N; // x step
9  const double dy = L / M; // y step
10
11 // Numerical
12 const int time_steps = 20000;              // number of time steps
13 const double delta_convergence = 0.00001; // maximum delta for the error
14 const double initial_phi = 0;              // initial value
15 double delta_t = 0.001;                    // time step
16
17 // Physical
18 const double rho = 1;    // inlet density
19 const double S_phi = 0; // source term depenant on phi
20 const double S_c = 0;    // constant source term
21
22 // Diagonal flow
23 const double u0 = 50;    // inlet velocity for diagonal flow
24 const double phi_h = 2; // phi value for high side
25 const double phi_l = 0; // phi value for low side
```

A.2 Meshing

The *mesh.cpp* file contains functions used to build the mesh, set its values and export the data stored in it.

```
1 // Last update: 2024/10/19
2 // Author: Ricard Arbat Carandell
```

```cpp

// Master in Aerospace Engineering - Computational Engineering
// Universitat Politècnica de Catalunya (UPC) - BarcelonaTech
// Overview: Mesh definition functions

#include <iostream>
#include <vector>
#include <cmath>
#include <fstream>
#include <string>
#include <sstream>
#include "parameters.cpp"
using namespace std;

// node struct
struct node
{
    double x, y, u, v, phi;
};

/**
 * Fills the mesh with nodes as it defines their positions.
 *
 * @param mesh Mesh matrix (vector of vectors) to be filled with Node structs
 * @param type Type of problem to solve
 */
void build_mesh(vector<vector<vector<node>>> &mesh, string type)
{
    for (int t = 0; t < time_steps; t++)
    {
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < M; j++)
            {
                mesh[t][i][j].x = -1 + (i * dx) + (0.5 * dx);
                mesh[t][i][j].y = (j * dy) + (0.5 * dy);
            }
        }
    }
}

/**
 * Sets the initial density of the mesh nodes.
 * also checks if the node is solid and sets the solid density
 *
 * @param mesh_t Mesh matrix at time t
```

```
 * @param variable Value to set
 * @param name Name of the variable to set
 */
void set_mesh_value(vector<vector<node>> &mesh_t, float variable, string name)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
        {
            if (name == "u")
            {
                mesh_t[i][j].u = variable;
            }
            else if (name == "v")
            {
                mesh_t[i][j].v = variable;
            }
            else if (name == "phi")
            {
                mesh_t[i][j].phi = variable;
            }
        }
    }
}

/**
 * Exports the mesh data to a CSV file
 *
 * @param mesh Mesh matrix at time t
 * @param filename Name of the file to be exported
 * @param folder Folder where the file will be saved
 * @param how_many How many time steps to export. Either "all" or "last".
 **/
void export_data(vector<vector<vector<node>>> &mesh, string filename = "output.csv", string folder = "smith_hutton", string how_many = "all")
{

    string header = "s,X,Y,phi,U,V";
    string rows = header + "\n";

    if (how_many == "all")
    {
        for (int t = 0; t < time_steps; t++)
        {
            cout << "Exporting data at time " << t << endl;
            for (int i = 1; i < N - 1; i += 1)
```

```
            {
                for (int j = 1; j < M - 1; j += 1)
                {
                    string row = to_string(t * delta_t) + "," + to_string(
    mesh[t][i][j].x) + "," + to_string(mesh[t][i][j].y) + "," + to_string(
    mesh[t][i][j].phi) + "," + to_string(mesh[t][i][j].u) + "," + to_string(
    mesh[t][i][j].v);
                    rows.append(row + "\n");
                }
            }
        }
    }
    else if (how_many == "last")
    {
        int t = time_steps - 1;
        cout << "Exporting data last timestep " << endl;
        for (int i = 1; i < N - 1; i += 1)
        {
            for (int j = 1; j < M - 1; j += 1)
            {
                string row = to_string(t * delta_t) + "," + to_string(mesh[t
    ][i][j].x) + "," + to_string(mesh[t][i][j].y) + "," + to_string(mesh[t][i
    ][j].phi) + "," + to_string(mesh[t][i][j].u) + "," + to_string(mesh[t][i
    ][j].v);
                rows.append(row + "\n");
            }
        }
    }

    string path = folder + "/" + filename;

    ofstream outfile(path);
    outfile << rows << endl;
    outfile.close();
}

/**
 * Exports the mesh data to a CSV file
 *
 * @param mesh Mesh matrix at time t
 * @param scheme Scheme used
 * @param outfile File to write the data
 **/
void export_data_at_outlet(vector<vector<vector<node>>> &mesh, string scheme
    , ofstream &outfile)
{

```

```
134     string rows = scheme + ",";
135 
136     int j = 2;
137     for (int i = N / 2; i < N - 1; i++)
138     {
139         string row = to_string(mesh[time_steps - 1][i][j].phi) + ",";
140         rows.append(row);
141         cout << "x" << mesh[time_steps - 1][i][j].x << " y" << mesh[
    time_steps - 1][i][j].y << " phi" << mesh[time_steps - 1][i][j].phi <<
    endl;
142     }
143     rows.append("\n");
144 
145     outfile << rows << endl;
146 }
147 
148 /**
149  * Makes a filename with the time and name
150  *
151  * @param Pe Peclet number
152  * @param scheme Scheme used
153  * @param name Name of the file
154  * @param type Type of problem
155  * @param delta_t Time step size
156  *
157  */
158 string file_name(double Pe, string scheme, string name, string type, double
    delta_t = 0.001)
159 {
160     std::ostringstream oss;
161     if (Pe > 100000.0)
162     {
163         string Pe_s = "1000000";
164         oss << name << "_Pe_" << Pe_s << "_S_" << scheme << "_M_" << M << "
    _type_" << type << "_t_" << delta_t << ".csv";
165     }
166     else
167     {
168         oss << name << "_Pe_" << Pe << "_S_" << scheme << "_M_" << M << "
    _type_" << type << "_t_" << delta_t << ".csv";
169     }
170 
171     std::string var = oss.str();
172 
173     return var;
174 }
175 
```

```
/**
 * Prints the value of phi in the mesh graphically, like a matrix
 *
 * @param values Matrix (Vector of vectors) containing the phi values
 */

void print_phi_matrix(vector<vector<double>> &values)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
        {
            cout << values[i][j] << " ";
        }
        cout << endl;
    }
}
```

A.3 Convection-diffusion solver

The *compute_convection_diffusion.cpp* file contains all the algorithms used to compute the simulation.

```
// Last update: 2024/10/19
// Author: Ricard Arbat Carandell

// Master in Aerospace Engineering - Computational Engineering
// Universitat Politècnica de Catalunya (UPC)
// Overview: Convection-diffusion calculations

#define pass (void)0

#include <iostream>
#include <vector>
#include <cmath>
#include <fstream>
#include "mesh.cpp"
using namespace std;

double cds(double P);
double hds(double P);
double eds(double P);
double pds(double P);
double calculate_error(vector<vector<double>> v1, vector<vector<double>> v2)
    ;
void fill_values_at_t(vector<vector<double>> &values, double value);
```

```
23 void evaluate_time_step_smith_hutton(int t, vector<vector<vector<node>>> &
       mesh, double gamma, double delta_t, string scheme);
24 void evaluate_time_step_diagonal(int t, vector<vector<vector<node>>> &mesh,
       double gamma, string scheme);
25
26 /**
27  * Computes iterates the evalueate time step function for each time step.
28  * Also selects the type of problem to solve.
29  *
30  * @param mesh Mesh matrix (vector of vectors)
31  * @param gamma Diffusion coefficient
32  * @param delta_t Time step size
33  * @param scheme Scheme to be used
34  * @param type Type of problem
35  */
36
37 void compute_diffusive_convective(vector<vector<vector<node>>> &mesh, double
       gamma, double delta_t, string scheme, string type)
38 {
39
40     // Time loop
41     for (int t = 1; t < time_steps; t++)
42     {
43         if (type == "smith-hutton")
44         {
45             evaluate_time_step_smith_hutton(t, mesh, gamma, delta_t, scheme)
       ;
46         }
47         else if (type == "diagonal")
48         {
49             evaluate_time_step_diagonal(t, mesh, gamma, scheme);
50         }
51     }
52 }
53
54 /**
55  * Evaluates the time step of the mesh for the Smith-Hutton problem.
56  * Iterates over the mesh and calculates the new phi value for each node
       using the selected scheme.
57  *
58  * @param t Time instant of the simulation
59  * @param mesh Mesh matrix (vector of vectors)
60  * @param gamma Diffusion coefficient
61  * @param delta_t Time step size
62  * @param scheme Scheme to be used
63  */
```

```
void evaluate_time_step_smith_hutton(int t, vector<vector<vector<node>>> &
    mesh, double gamma, double delta_t, string scheme)
{
    // Initializing variables for each iteration
    double aE, aW, aN, aS, aP, aP0, b;
    double De, Dw, Dn, Ds;
    double Fe, Fw, Fn, Fs;
    double ue, uw, vn, vs;
    double Pe, Pw, Pn, Ps;
    double Ae, Aw, An, As;
    double phiE, phiW, phiN, phiS;
    double error = 10;
    int cont = 0;

    vector<vector<double>> last_phi(N, vector<double>(M, 0));
    vector<vector<double>> next_phi(N, vector<double>(M, 0));

    // Copy values from last timestep
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
        {
            last_phi[i][j] = mesh[t - 1][i][j].phi; // copy phi from
    previous time step
        }
    }

    while (error > delta_convergence)
    {
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < M; j++)
            {
                ue = 2 * mesh[t][i][j].y * (1 - ((mesh[t][i][j].x + 0.5 * dx
    ) * (mesh[t][i][j].x + 0.5 * dx)));
                uw = 2 * mesh[t][i][j].y * (1 - ((mesh[t][i][j].x - 0.5 * dx
    ) * (mesh[t][i][j].x - 0.5 * dx)));
                vn = -2 * mesh[t][i][j].x * (1 - ((mesh[t][i][j].y + 0.5 *
    dy) * (mesh[t][i][j].y + 0.5 * dy)));
                vs = -2 * mesh[t][i][j].x * (1 - ((mesh[t][i][j].y - 0.5 *
    dy) * (mesh[t][i][j].y - 0.5 * dy)));

                De = gamma * (dy / dx);
                Dw = gamma * (dy / dx);
                Dn = gamma * (dx / dy);
                Ds = gamma * (dx / dy);

```

```
                Fe = rho * dy * ue;
                Fw = rho * dy * uw;
                Fn = rho * dx * vn;
                Fs = rho * dx * vs;

                // Peclet numbers
                Pe = Fe / De;
                Pw = Fw / Dw;
                Pn = Fn / Dn;
                Ps = Fs / Ds;

                // Schemes
                if (scheme == "UDS")
                {
                    Ae = 1;
                    Aw = 1;
                    An = 1;
                    As = 1;
                }
                else if (scheme == "CDS")
                {
                    Ae = cds(Pe);
                    Aw = cds(Pw);
                    An = cds(Pn);
                    As = cds(Ps);
                }
                else if (scheme == "HDS")
                {
                    Ae = hds(Pe);
                    Aw = hds(Pw);
                    An = hds(Pn);
                    As = hds(Ps);
                }
                else if (scheme == "PDS")
                {
                    Ae = pds(Pe);
                    Aw = pds(Pw);
                    An = pds(Pn);
                    As = pds(Ps);
                }
                else if (scheme == "EDS")
                {
                    Ae = eds(Pe);
                    Aw = eds(Pw);
                    An = eds(Pn);
                    As = eds(Ps);
                }
```

```
152
153                 aE = De * Ae + max(-Fe, 0.0);
154                 aW = Dw * Aw + max(Fw, 0.0);
155                 aN = Dn * An + max(-Fn, 0.0);
156                 aS = Ds * As + max(Fs, 0.0);
157
158                 aP = aE + aW + aN + aS + (rho * dx * dy / delta_t) - (S_phi
    * dx * dy);
159
160                 b = ((rho * dx * dy * mesh[t - 1][i][j].phi) / delta_t) + (
    S_c * dx * dy);
161
162                 if (i == 0)
163                 {
164                     next_phi[i][j] = 1 - tanh(10); // bc
165                 }
166                 else if (i == N - 1)
167                 {
168                     next_phi[i][j] = 1 - tanh(10); // bc
169                 }
170                 else if (j == 0)
171                 {
172                     if (i < N / 2) // inlet
173                     {
174                         next_phi[i][j] = 1 + tanh(10 * (2 * mesh[t][i][j].x
    + 1)); // bc
175                     }
176                     else // outlet
177                     {
178                         phiE = last_phi[i + 1][j];
179                         phiW = last_phi[i - 1][j];
180                         phiN = last_phi[i][j + 1];
181                         phiS = last_phi[i][j]; // bc
182                         next_phi[i][j] = (aE * phiE + aW * phiW + aN * phiN
    + aS * phiS + b) / aP;
183                     }
184                 }
185                 else if (j == M - 1)
186                 {
187                     next_phi[i][j] = 1 - tanh(10); // bc
188                 }
189                 else
190                 {
191                     phiE = last_phi[i + 1][j];
192                     phiW = last_phi[i - 1][j];
193                     phiN = last_phi[i][j + 1];
194                     phiS = last_phi[i][j - 1];
```

```
195                     next_phi[i][j] = (aE * phiE + aW * phiW + aN * phiN + aS
    * phiS + b) / aP;
196                 }
197             }
198         }
199 
200         error = calculate_error(next_phi, last_phi);
201 
202         if (cont % 100 == 0)
203             cout << "Time step: " << t << " Error: " << error << endl;
204         cont++;
205 
206         for (int i = 0; i < N; i++)
207         {
208             for (int j = 0; j < M; j++)
209             {
210                 last_phi[i][j] = next_phi[i][j];
211             }
212         }
213     }
214 
215     for (int i = 0; i < N; i++)
216     {
217         for (int j = 0; j < M; j++)
218         {
219             mesh[t][i][j].phi = next_phi[i][j];
220         }
221     }
222 }
223 
224 /**
225  * Evaluates the time step of the mesh for the diagonal flow problem.
226  * Iterates over the mesh and calculates the new phi value for each node
    using the selected scheme.
227  *
228  * @param t Time instant of the simulation
229  * @param mesh Mesh matrix (vector of vectors)
230  * @param gamma Diffusion coefficient
231  * @param scheme Scheme to be used
232  */
233 void evaluate_time_step_diagonal(int t, vector<vector<vector<node>>> &mesh,
    double gamma, string scheme)
234 {
235     // Initializing variables for each iteration
236     double aE, aW, aN, aS, aP, aP0, b;
237     double De, Dw, Dn, Ds;
238     double Fe, Fw, Fn, Fs;
```

```
    double ue, uw, vn, vs;
    double Pe, Pw, Pn, Ps;
    double Ae, Aw, An, As;
    double phiE, phiW, phiN, phiS;
    double error = 10;
    int cont = 0;

    vector<vector<double>> last_phi(N, vector<double>(M, 0));
    vector<vector<double>> next_phi(N, vector<double>(M, 0));

    // Copy values from last timestep
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
        {
            last_phi[i][j] = mesh[t - 1][i][j].phi; // copy phi from
previous time step
        }
    }

    while (error > delta_convergence)
    {
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < M; j++)
            {
                ue = u0 * cos(0.785398);
                uw = u0 * cos(0.785398);
                vn = u0 * sin(0.785398);
                vs = u0 * sin(0.785398);

                De = gamma * (dy / dx);
                Dw = gamma * (dy / dx);
                Dn = gamma * (dx / dy);
                Ds = gamma * (dx / dy);

                Fe = rho * dy * ue;
                Fw = rho * dy * uw;
                Fn = rho * dx * vn;
                Fs = rho * dx * vs;

                // Peclet numbers
                Pe = Fe / De;
                Pw = Fw / Dw;
                Pn = Fn / Dn;
                Ps = Fs / Ds;

```

```
285                 // Schemes
286                 if (scheme == "UDS")
287                 {
288                     Ae = 1;
289                     Aw = 1;
290                     An = 1;
291                     As = 1;
292                 }
293                 else if (scheme == "CDS")
294                 {
295                     Ae = cds(Pe);
296                     Aw = cds(Pw);
297                     An = cds(Pn);
298                     As = cds(Ps);
299                 }
300                 else if (scheme == "HDS")
301                 {
302                     Ae = hds(Pe);
303                     Aw = hds(Pw);
304                     An = hds(Pn);
305                     As = hds(Ps);
306                 }
307                 else if (scheme == "PDS")
308                 {
309                     Ae = pds(Pe);
310                     Aw = pds(Pw);
311                     An = pds(Pn);
312                     As = pds(Ps);
313                 }
314                 else if (scheme == "EDS")
315                 {
316                     Ae = eds(Pe);
317                     Aw = eds(Pw);
318                     An = eds(Pn);
319                     As = eds(Ps);
320                 }
321
322                 aE = De * Ae + max(-Fe, 0.0);
323                 aW = Dw * Aw + max(Fw, 0.0);
324                 aN = Dn * An + max(-Fn, 0.0);
325                 aS = Ds * As + max(Fs, 0.0);
326
327                 aP = aE + aW + aN + aS + (rho * dx * dy / delta_t) - (S_phi
    * dx * dy);
328
329                 b = ((rho * dx * dy * mesh[t - 1][i][j].phi) / delta_t) + (
    S_c * dx * dy);
```

```
330
331                 if (i == 0) // Left BC
332                 {
333                     next_phi[i][j] = phi_l;
334                 }
335                 else if (i == N - 1) // Right BC
336                 {
337                     next_phi[i][j] = phi_h;
338                 }
339                 else if (j == M - 1) // Top BC
340                 {
341                     next_phi[i][j] = phi_l;
342                 }
343                 else if (j == 0) // Bottom inlet BC nodes
344                 {
345                     next_phi[i][j] = phi_h;
346                 }
347                 else
348                 {
349                     phiE = last_phi[i + 1][j];
350                     phiW = last_phi[i - 1][j];
351                     phiN = last_phi[i][j + 1];
352                     phiS = last_phi[i][j - 1];
353                     next_phi[i][j] = (aE * phiE + aW * phiW + aN * phiN + aS
    * phiS + b) / aP;
354                 }
355             }
356         }
357
358         error = calculate_error(next_phi, last_phi);
359
360         if (cont % 100 == 0)
361             cout << "Time step: " << t << " Error: " << error << endl;
362         cont++;
363
364         for (int i = 0; i < N; i++)
365         {
366             for (int j = 0; j < M; j++)
367             {
368                 last_phi[i][j] = next_phi[i][j];
369             }
370         }
371     }
372
373     for (int i = 0; i < N; i++)
374     {
375         for (int j = 0; j < M; j++)
```

```
376         {
377             mesh[t][i][j].phi = next_phi[i][j];
378         }
379     }
380 }
381 
382 /**
383  * Compares two matrices and returns the maximum error between them.
384  *
385  * @param v1 Matrix (Vector of vectors) to compare
386  * @param v2 Matrix (Vector of vectors) to compare
387  */
388 double calculate_error(vector<vector<double>> v1, vector<vector<double>> v2)
389 {
390     double max_error = delta_convergence;
391     for (int i = 1; i < v1.size(); i++)
392     {
393         for (int j = 1; j < v1[i].size(); j++)
394         {
395             double difference = abs(v1[i][j] - v2[i][j]);
396             if (difference > max_error)
397             {
398                 max_error = difference;
399             }
400         }
401     }
402     return max_error;
403 }
404 
405 /**
406  * Fills the initial values of the phi temporal mesh
407  *
408  * @param values Matrix (Vector of vectors) containing the next stream
    values
409  * @param value Value to set for all the nodes
410  */
411 void fill_values_at_t(vector<vector<double>> &values, double value)
412 {
413     for (int i = 0; i < N; i++)
414     {
415         for (int j = 0; j < M; j++)
416         {
417             values[i][j] = value;
418         }
419     }
420 }
421 
```

```
// Schemes

// Central Difference Scheme
double cds(double P)
{
    return 1 - 0.5 * abs(P);
}

// Hybrid Difference Scheme
double hds(double P)
{
    return max(1 - 0.5 * abs(P), 0.0);
}

// Power Difference Scheme
double pds(double P)
{
    return max(pow(1 - 0.5 * abs(P), 5), 0.0);
}

// Exponential Difference Scheme
double eds(double P)
{
    double val = abs(P) / (exp(abs(P)) - 1) + 0.0000000000001;
    return val;
}
```

A.4 Main

Finally, the main file uses all the defined functions in order to define the parameters, build the mesh, and compute the numerical resolution. It is the entry point of the program, and the one to be executed. A few other files are present in the provided .zip file, each one used to generate different solutions to different cases in a simple way. These codes will not be written here, as they are not relevant enough, but are attached in a *.zip* file:

- *generate_diag_flows.cpp*
- *generate_output_curves.cpp*
- *generate_output_curves_deltat.cpp*
- *generate_smith_hutt_flows.cpp*

And the main file is the following

```
// Last update: 2024/10/19
```

```cpp
// Author: Ricard Arbat Carandell

// Master in Aerospace Engineering - Computational Engineering
// Universitat Politècnica de Catalunya (UPC) - BarcelonaTech
// Overview: Main function to solve the convection-diffusion problem.

// Libraries
#include <iostream>
#include <vector>
#include <cmath>
#include <fstream>
#include "compute_diff_conv.cpp"
#include <chrono>
using namespace std;
using namespace std::chrono;

/**
 * Main function of the program. It initializes the mesh, computes the stream
 * function and calculates the forces around the cilinder.
 */
int main(void)
{

    double Pe = 1000000;       // peclet number
    double gamma = rho / Pe;   // diffusion coefficient
    string scheme = "PDS";     // scheme to be used
    string type = "diagonal"; // type of problem

    if (type == "smith-hutton")
    {
        N = 2 * M;
    }
    else if (type == "diagonal")
    {
        N = M;
    }

    // Main mesh
    vector<vector<vector<node>>> mesh(time_steps, vector<vector<node>>(N, vector<node>(M)));
    build_mesh(mesh, type); // creating the mesh

    set_mesh_value(mesh[0], initial_phi, "phi");

    // Compute stream function
    auto start = high_resolution_clock::now();
```

```
47      compute_diffusive_convective(mesh, gamma, delta_t, scheme, type); //
        stream solver
48      auto stop = high_resolution_clock::now();
49      auto duration = duration_cast<microseconds>(stop - start);
50
51      // Export data
52      export_data(mesh, file_name(Pe, scheme, "output", type), "output", "last
        ");
53
54      // Print final results
55      cout << "### CONVECTION DIFFUSION FLOW RESULTS ###" << endl;
56      cout << "Computation time = " << duration.count() / 1000 << "ms" << endl
        ;
57      return 0;
58  }
```