Numerical Resolution of the Convection-Diffusion Equation
The Smith-Hutton Case

Giada Alessi
Master in Thermal Engineering
Universitat Politècnica de Catalunya

March 4, 2025

Contents

Abstract

This report presents a numerical study of the Smith-Hutton convection-diffusion problem, where the steady-state transport of a scalar property ϕ is investigated under a predefined velocity field. The numerical solution is obtained using the finite volume method with Upwind Difference Scheme (UDS) and Central Difference Scheme (CDS). The effects of varying the ρ/Γ ratio on the solution stability and accuracy are analyzed.

1 Introduction

The Smith-Hutton case is a reference problem for testing numerical schemes in convection-diffusion problems. It consists of a solenoidal flow in a 2D domain, where a transported property ϕ is studied under different transport regimes.
The governing convection-diffusion equation is given by:

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

where $v = (u, v)$ is the known velocity field and Γ is the diffusion coefficient.

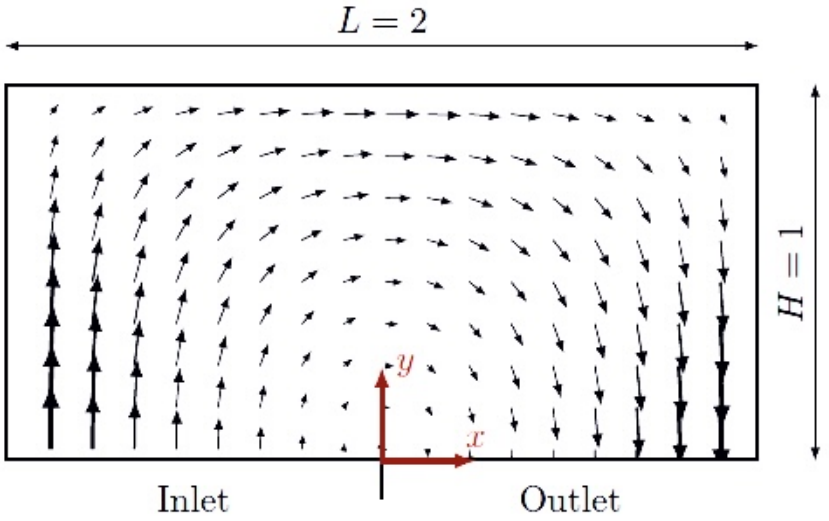

Figure 1: Smith-Hutton case

The boundary conditions for the present problem are:

- **Inlet:** $\phi = 1 + \tanh[(2x + 1)\alpha]$, for $x \in [-1, 0], y = 0$
- **Outlet:** $\frac{\partial \phi}{\partial y} = 0$, for $x \in [0, 1], y = 0$
- **Walls:** $\phi = 1 - \tanh[\alpha]$

where $\alpha = 10$.
The initial condition for the map is assumed to be $\phi = 0 \ \forall(x, y)$. The problem is solved using explicit temporal scheme and central difference scheme (CDS).

2 Theoretical Background

The convection-diffusion equation 1 provides a unified framework that can represent the Navier-Stokes, energy, and species transport equations by appropriately selecting the physical variable ϕ. Regardless of the specific application, these equations share a common structure consisting of transient (unsteady) terms, convective terms, diffusive terms, and additional source or sink terms. In its general form, the equation models the unsteady transport of a scalar (or vector) quantity, where:

- ϕ is the scalar or vector variable being transported.
- Γ_ϕ is the rate of diffusion for the transported quantity ϕ.
- $\dot{S}_\phi$ accounts for the generation or destruction of the quantity ϕ in the system.

Using the mass conservation equation, $\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{v}) = 0$, the previous convection-diffusion equation can be rewritten as follows:

$$\rho \frac{\partial \phi}{\partial t} + \rho \vec{v} \cdot \nabla \phi = \nabla \cdot (\Gamma \nabla \phi) + \dot{S}_\phi \tag{2}$$

Taking a small control volume, integrating it using the finite volume approach, and applying Gauss's theorem, the following equation is obtained:

$$\rho \frac{\partial \phi}{\partial t} \Delta V + \rho \sum (\vec{v}\phi)_f \cdot \Delta S = \sum (\Gamma \nabla \phi)_f \cdot \Delta S + \dot{S}_\phi \cdot \Delta V \tag{3}$$

where the subscript f denotes that the properties are evaluated at the control volume faces rather than at the cell centroids.
Analyzing the terms individually, we obtain:

- Convective term:

$$conv = \rho \sum (\vec{v}\phi)_f \cdot \Delta S = \rho \left(u_e \phi_e \Delta S_e - u_w \phi_w \Delta S_w + v_n \phi_n \Delta S_n - v_s \phi_s \Delta S_s \right) \tag{4}$$

- Diffusive term:

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

- Source/Sink term:

$$S = \dot{S}_\phi \cdot \Delta V = 0 \tag{6}$$

  this term is neglected in the present exercise.

- Transient term:

$$\rho \frac{\partial \phi}{\partial t} \Delta V = \rho \frac{\phi_P^{n+1} - \phi_P^n}{\Delta t} \Delta V \tag{7}$$

Applying explicit Forward Euler time discretization, for simplicity, the final form of equation 1 is:

$$\frac{\rho \phi_P^{n+1} - \phi_P^n}{\Delta t} = \frac{1}{\Delta V} (-\text{conv}^n + \text{diff}^n) \tag{8}$$

where both the convective and diffusive terms are evaluated at time instant t^n.

3 Numerical Implementation

The numerical implementation is structured using modular functions, with matrices handling data storage and processing. An iterative solver is used to advance the solution in time until steady-state convergence is reached. To streamline visualization, results are automatically plotted using Gnuplot directly within the C++ code. The following sections describe the key steps of the implementation

3.1 Computational Mesh and Velocity Field

The computational domain is a rectangular region which has been discretized using a structured mesh of $N_x = 100$ and $N_y = 50$ control volumes. The mesh is semi-uniform, consisting of $N_x + 2$ and $N_y + 2$ nodes, which are positioned at the center of the control volumes and along the boundaries. The mesh is generated in the function `MeshDefinition()`, where the positions of nodes, faces, and control volume centers are computed:

- Control volume positions are stored in matrices $x_{cv}[i][j]$ and $y_{cv}[i][j]$.
- Nodes' locations are derived from the control volume face coordinates and stored in $x_P[i][j]$ and $y_P[i][j]$.
- The control volume's volume is computed and stored in $v_P[i][j]$.
- Boundary nodes are explicitly initialized to ensure correct positioning.

For the purpose of this exercise, the velocity field is assumed to be known and is assigned in the function `VelocityField()`, where the velocity components are defined at the control volume faces as:

$$u = 2y(1 - x^2), \quad v = -2x(1 - y^2) \tag{9}$$

where the coordinates x and y refer, as specified earlier, to the coordinates of the corresponding face.

3.2 Discretization of the Governing Equation

The governing equation has been spatially discretized using the finite volume method and temporally using the explicit Forward Euler time integration method. The resulting equation from these implementations is as follows:

$$\begin{aligned}\frac{\rho\phi_P^{n+1} - \phi_P^n}{\Delta t} = \frac{1}{\Delta V}\Bigg[&- \rho\Big(u_e\phi_e^n\Delta S_e - u_w\phi_w^n\Delta S_w + v_n\phi_n^n\Delta S_n - v_s\phi_s^n\Delta S_s\Big) + \\ &+\Gamma\left(\frac{\phi_E^n - \phi_P^n}{d_{PE}}\Delta S_e + \frac{\phi_W^n - \phi_P^n}{d_{PW}}\Delta S_w + \frac{\phi_N^n - \phi_P^n}{d_{PN}}\Delta S_n + \frac{\phi_S^n - \phi_P^n}{d_{PS}}\Delta S_s\right)\Bigg]\end{aligned} \tag{10}$$

The transported property at each node/face in the right-hand side of the equation is evaluated at the time instant t^n. Since an explicit time discretization is employed, the updated value ϕ_P^{n+1} can be directly computed at each time step without requiring the solution of a system of equations. This approach ensures a straightforward time advancement, as ϕ_P^{n+1} depends only on known quantities from the previous time step, making the method computationally less expensive and easier to implement.

The convective term requires the transported property to be evaluated at the control volume face rather than at the node. Since the exact value at the face is unknown, an interpolation method must be applied. Initially, the Central Difference Scheme (CDS) was chosen as the first approach to solve the exercise. In the following, the solution using CDS will be explained in detail. Subsequently, the Upwind Difference Scheme (UDS) will be introduced to analyze the stability and convergence of the code.

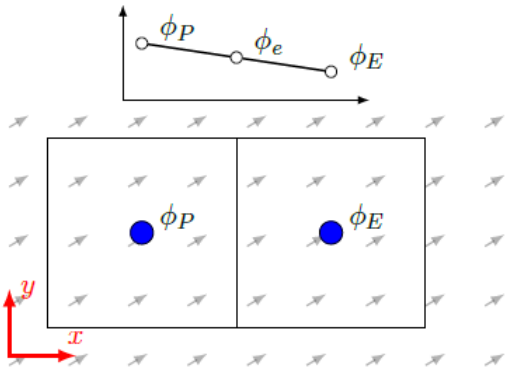

Figure 2: Central Difference Scheme (CDS)

The fundamental idea behind CDS is to approximate the value at the face using the known values of the surrounding nodes. This is achieved through linear interpolation. Following the example shown in Figure 2, and applying this method to each face, the interpolation is given by:

$$\phi_e^n = \frac{\phi_P^n + \phi_E^n}{2} \tag{11}$$

This scheme is not particularly dissipative but tends to be more unstable and may produce spurious oscillations if the chosen time step is not sufficiently small.
By introducing the CDS scheme into Equation 10, the resulting discretized equation is:

$$\begin{aligned}\frac{\rho\phi_P^{n+1} - \phi_P^n}{\Delta t} = \frac{1}{\Delta V}\Bigg[&- \rho\Bigg(u_e\frac{\phi_P^n + \phi_E^n}{2}\Delta S_e - u_w\frac{\phi_P^n + \phi_W^n}{2}\Delta S_w \\ &+ v_n\frac{\phi_P^n + \phi_N^n}{2}\Delta S_n - v_s\frac{\phi_P^n + \phi_S^n}{2}\Delta S_s\Bigg) \\ &+ \Gamma\Bigg(\frac{\phi_E^n - \phi_P^n}{d_{PE}}\Delta S_e + \frac{\phi_W^n - \phi_P^n}{d_{PW}}\Delta S_w \\ &+ \frac{\phi_N^n - \phi_P^n}{d_{PN}}\Delta S_n + \frac{\phi_S^n - \phi_P^n}{d_{PS}}\Delta S_s\Bigg)\Bigg]\end{aligned} \tag{12}$$

Although an explicit time discretization is used, the coefficients a_F $(F = N, S, E, W)$ are explicitly written to preserve consistency with the structured approach followed in previous exercises. This notation ensures a clearer representation of the finite volume formulation while explicitly distinguishing the contribution of neighboring nodes. Furthermore, the coefficients a_P and a_F naturally arise from the discretization process, making the transition between different schemes and numerical methods more intuitive.
The coefficients for each internal node of the mesh are then computed using the function `InternalNodesCoefficients()`, and they are defined as follows.

$$a_P = \frac{\rho}{\Delta t} \tag{13}$$

$$a_E = \frac{\Delta S_e}{\Delta V}\left(-\rho\frac{u_e}{2} + \frac{\Gamma}{d_{PE}}\right) \tag{14}$$

$$a_W = \frac{\Delta S_w}{\Delta V}\left(\rho\frac{u_w}{2} + \frac{\Gamma}{d_{PW}}\right) \tag{15}$$

$$a_N = \frac{\Delta S_n}{\Delta V}\left(-\rho\frac{v_n}{2} + \frac{\Gamma}{d_{PN}}\right) \tag{16}$$

$$a_S = \frac{\Delta S_s}{\Delta V}\left(\rho\frac{v_s}{2} + \frac{\Gamma}{d_{PS}}\right) \tag{17}$$

$$b_P = \frac{\rho}{\Delta t} + \frac{\Delta S_e}{\Delta V}\left(-\rho\frac{u_e}{2} - \frac{\Gamma}{d_{PE}}\right) + \frac{\Delta S_w}{\Delta V}\left(\rho\frac{u_w}{2} - \frac{\Gamma}{d_{PW}}\right) + \frac{\Delta S_n}{\Delta V}\left(-\rho\frac{v_n}{2} - \frac{\Gamma}{d_{PN}}\right) + \frac{\Delta S_s}{\Delta V}\left(\rho\frac{v_s}{2} - \frac{\Gamma}{d_{PS}}\right) \tag{18}$$

3.3 Boundary Conditions

The function `BoundaryConditions()` imposes Dirichlet and Neumann boundary conditions at the domain boundaries:

- **Inlet** ($y = 0$, for $x < 0$):

$$\phi = 1 + \tanh((2x+1)\alpha) \tag{19}$$

  resulting in:

$$a_E = a_W = a_S = a_N = 0 \tag{20}$$

$$a_P = 1, \quad b_P = 1 + \tanh((2x+1)\alpha) \tag{21}$$

- **Outlet** ($y = 0$, for $x > 0$):

$$\frac{\partial \phi}{\partial y} = 0 \tag{22}$$

  resulting in:

$$a_E = a_W = a_S = 0, \quad a_N = 1 \tag{23}$$

$$a_P = 1, \quad b_P = 0 \tag{24}$$

- **Remaining boundaries** ($y = H$, $x = -L/2$, $x = L/2$):

$$\phi = 1 - \tanh(\alpha) \tag{25}$$

  resulting in:

$$a_E = a_W = a_S = a_N = 0 \tag{26}$$

$$a_P = 1, \quad b_P = 1 - \tanh(\alpha) \tag{27}$$

3.4 Solution Algorithm

The numerical solution is obtained following this procedure:

- Definition of global variables and matrices outside of **main()** to be used in multiple functions.
- Definition of the necessary functions, as described earlier.

- Inside **main()**:
  - Definition of physical and numerical parameters.
  - Computation of the mesh using **MeshDefinition()**.
  - Computation of the velocity field using **VelocityField()**.
  - Initialization of the property map, setting $\phi = 0$.
  - Computation of the internal node coefficients for the initial map using **InternalNodesCoefficients()**.
  - Application of boundary conditions using **BoundaryConditions()**.

The steady-state map of the property ϕ is then obtained iteratively in time using a solver, implemented in the function `Solver()`:

- Initialization of the time counter to zero.
- Entry into the steady-state convergence loop:
  - Solve the discretized equation:
    - for internal nodes:

$$a_P \phi_P^{n+1} = a_W \phi_W^n + a_E \phi_W^n + a_N \phi_N^n + a_S \phi_S^n + b_P \phi_P^n \tag{28}$$

    - for boundary nodes:

$$a_P \phi_P^{n+1} = a_W \phi_W^n + a_E \phi_W^n + a_N \phi_N^n + a_S \phi_S^n + b_P \tag{29}$$

  - Assessment of convergence based on the maximum residual, ensuring that the changes in ϕ remain below a predefined tolerance:

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

  - Update of the time counter by adding Δt.
  - If convergence is not reached, the previous map is updated:

$$\phi^n = \phi^{n+1}$$

- The iterative process is repeated until steady-state conditions are met, meaning that the difference between consecutive iterations remains below the prescribed tolerance.

4 Results

The solution was obtained for three different values of ρ/Γ: 10, 1000, and $1,000,000$ and compared to the results given with the exercise. To solve the exercise, Γ was set to 1 and ρ was changed to fit the ratio ρ/Γ.

Position x	$\rho/\Gamma = 10$		$\rho/\Gamma = 1000$		$\rho/\Gamma = 1\ 000\ 000$	
	Given	Obtained	Given	Obtained	Given	Obtained
0.0	1.989	1.818	2.0000	1.9990	2.000	1.863
0.1	1.402	1.382	1.9990	1.9993	2.000	1.907
0.2	1.146	1.134	1.9997	1.9998	2.000	1.995
0.3	0.946	0.938	1.9850	1.9916	1.999	2.055
0.4	0.775	0.768	1.8410	1.8213	1.964	2.006
0.5	0.621	0.616	0.9510	0.9671	1.000	0.917
0.6	0.480	0.476	0.1540	0.1515	0.036	0.095
0.7	0.349	0.347	0.0010	0.0062	0.001	-0.020
0.8	0.227	0.225	0.0000	0.0001	0.000	0.001
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	0.000

Table 1: Table of provided and obtained results for different values of ρ/Γ.

Table 1 presents a comparison between the obtained and expected results. During the computation, some issues related to convergence and stability were encountered. For lower values of ρ/Γ, the algorithm exhibited convergence difficulties when using a time step of $\Delta t = 10^{-3}$ s, whereas it successfully converged with $\Delta t = 10^{-4}$ s. For the case of $\rho/\Gamma = 10^3$, no additional convergence issues were observed and the same time step was maintained. However, when dealing with high ρ/Γ values such as 10^6, the program required significantly more computational time to reach convergence. Ultimately, it was necessary to slightly relax the convergence tolerance to 10^{-5} and further reduce the time step to $\Delta t = 10^{-5}$ s to achieve better results.

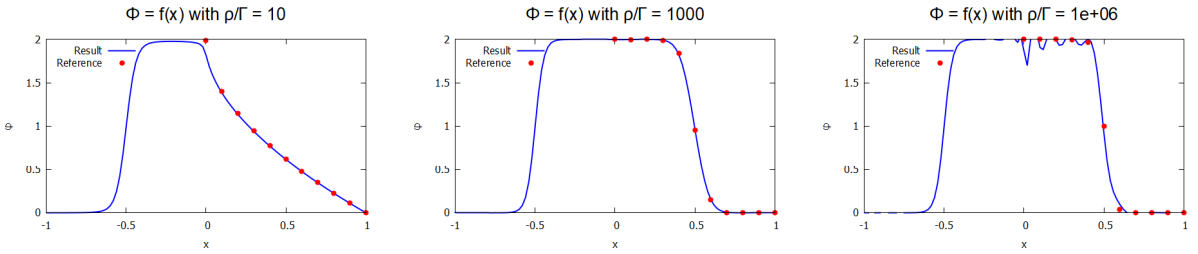

Figure 3: Plots of ϕ at the outlet for different ρ/Γ ratios compared to the reference data.

The images in Figure 3 show the plot of the property ϕ as a function of x, with the given reference values of ϕ at the outlet represented by red dots to assess the accuracy of the results.
It is of particular interest that in the case with $\rho/\Gamma = 10^6$, spurious oscillations appeared. These errors are caused by the dominance of convection over diffusion, leading to numerical instabilities. In high ρ/Γ regimes, central differencing schemes tend to produce non-physical oscillations due to the lack of sufficient numerical dissipation. Additionally, rough grids and large time steps amplify aliasing errors. To mitigate this issue, a more stable convective scheme, such as UDS or QUICK can be used. Additionally, reducing the time step Δt improves stability, refining the mesh in high-gradient regions enhances spatial resolution, and ensuring consistent boundary conditions prevents errors at the domain outlet.

For a complete visualization of the results, color maps depicting the steady-state distribution of the property have been plotted using Gnuplot.

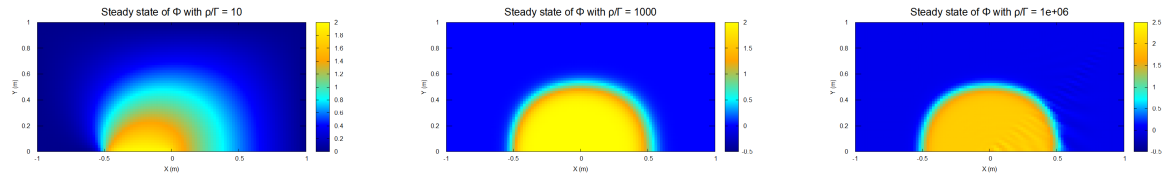

Figure 4: Color maps of ϕ at steady state of the Smith-Hutton case for different ρ/Γ ratios.

The color map for $\rho/\Gamma = 10^6$ shows some waviness on the right side, which corresponds to the spurious values observed in Figure 3. These are caused by numerical instabilities due to the central differencing scheme in convection-dominated cases. This highlights the need for more stable discretization methods or finer grid resolution to improve accuracy.

5 Code Enhancements

To improve the numerical stability and flexibility of the implementation, some modifications were introduced. First, the code was restructured using classes, allowing easy interchangeability between the Central Differencing Scheme (CDS) and the Upwind Differencing Scheme (UDS). After this modification, the UDS was tested to compare its results with those obtained using the CDS.

5.1 Class-Based Internal Nodes Coefficients Solver

The original `InternalNodesCoefficients()` function was implemented as a single global function, which made it harder to extend and modify when introducing different numerical methods such as CDS or UDS. To improve flexibility and modularity, the function was transformed into a class-based structure.
A base class, **InternalNodesCalculator**, was introduced to define a common interface for different numerical methods. This class provides a standardized framework for coefficient calculations and contains a pure virtual function, `computeCoefficients()`, which is implemented by specific subclasses.

Two main subclasses were created:

- **CDSMethod**: Implements the Central Differencing Scheme (CDS), which maintains the behavior of the original function.
- **UDSMethod**: Implements the Upwind Differencing Scheme (UDS), improving stability in convection-dominated problems.

This new approach facilitates the extension of the code, enabling the integration of additional discretization schemes without modifying the core solver structure.

To allow user selection of the method, the program asks the user to choose either 'CDS' or 'UDS' via command-line input. The selected method is created using the

`createCalculator()` function, which returns a pointer to the correct class. This method is then used to compute the internal coefficients before updating the solution over time until it reaches steady state.
In the **main()** function, the execution flow follows these steps:

1. The user inputs the desired discretization method ('CDS' or 'UDS').
2. The `createCalculator()` function returns an object of the corresponding class.
3. The selected method computes the internal node coefficients once, before time integration starts.
4. The explicit solver iteratively updates the transported property ϕ over time until convergence is achieved.
5. The dynamically allocated memory for the class instance is cleared at the end of the execution.

5.2 Comparison Between UDS and CDS for $\rho/\Gamma = 10^6$

The Upwind Differencing Scheme (UDS) was then tested and compared to the previously used Central Differencing Scheme (CDS).

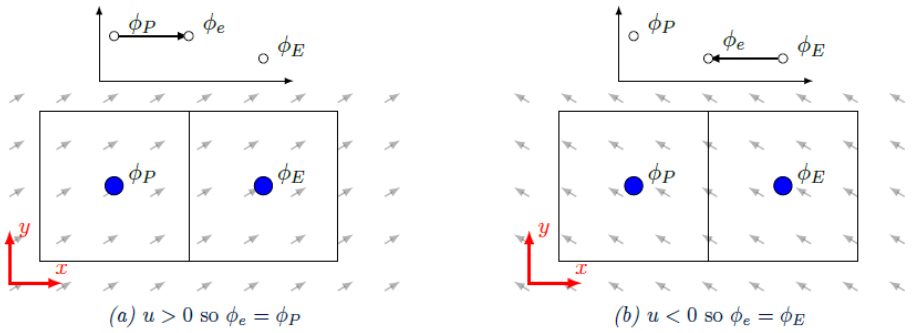

Figure 5: Upwind Difference Scheme (UDS) examples.

In the Upwind Difference Scheme (UDS), the value at a control volume face is assigned based on the direction of the velocity. Specifically, the face value is taken from the upstream node, meaning the node where the flow originates. This ensures that the transported quantity follows the natural direction of the fluid flow, improving numerical stability by introducing artificial diffusion. However, while UDS prevents oscillations and enhances robustness, it can also lead to excessive numerical diffusion, especially in high ρ/Γ regimes.

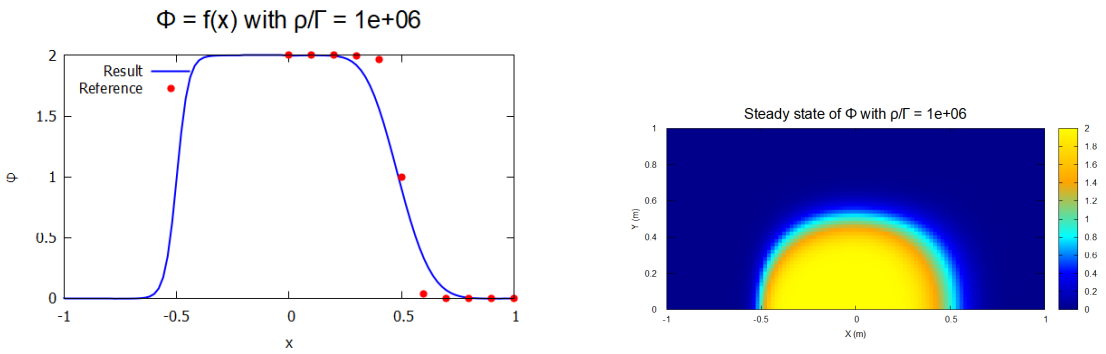

Figure 6: Plot at the outlet and color map at steady state of ϕ with $\rho/\Gamma = 10^6$, using Upwind Difference Scheme (UDS).

Figure 6 presents results obtained using the Upwind Difference Scheme with $\rho/\Gamma = 10^6$. The code turned out to be more stable and prone to convergence than when using CDS; a time step of 10^{-4} and a time convergence tolerance of 10^{-6} were sufficient to achieve convergence in less than 10 seconds.

Position x	$\rho/\Gamma = 1000000$		
	Given	CDS	UDS
0.0	2.000	1.863	1.998
0.1	2.000	1.907	2.000
0.2	2.000	1.995	1.995
0.3	1.999	2.055	1.919
0.4	1.964	2.006	1.563
0.5	1.000	0.917	0.906
0.6	0.036	-0.095	0.330
0.7	0.001	-0.020	0.068
0.8	0.000	-0.001	0.007
0.9	0.000	0.000	0.000
1.0	0.000	0.000	0.000

Table 2: Table of provided and obtained results for $\rho/\Gamma = 1000000$ using CDS and UDS.

As shown in the comparison graph and in Table 2, no spurious oscillations were observed. However, despite its improved stability, UDS introduces significant numerical diffusion, particularly in regions with steep gradients. This effect is visible in the smoothing of the sharp profile, where the solution deviates from the expected reference values.

This result highlights the trade-off between numerical diffusion and stability: while UDS is more robust and ensures a smooth solution, it tends to over-diffuse sharp transitions, reducing accuracy in capturing steep gradients.

6 Conclusion

The numerical results show that increasing the ρ/Γ ratio leads to a sharper gradient in ϕ, with possible oscillations when using the CDS scheme. The UDS scheme, while more

stable, introduces numerical diffusion. This highlights the trade-off between stability and accuracy when selecting convective schemes.

A C++ Code

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

using namespace std;
using namespace std::chrono;

// Global Variables
double const L = 2.0;            // Lenght
double const H = 1.0;            // Height
double const W = 1.0;            // Width
const int Nx = 100, Ny = 50;     // Number of control volumes (100x50)

// Vectors Definition
double phi[Ny+2][Nx+2], phi_1[Ny+2][Nx+2], phi_g[Ny+2][Nx+2];    // Transported property
double aP[Ny+2][Nx+2], aE[Ny+2][Nx+2], aW[Ny+2][Nx+2],           // Coefficients
       aN[Ny+2][Nx+2], aS[Ny+2][Nx+2], bP[Ny+2][Nx+2];
double xcv[Ny+2][Nx+2], ycv[Ny+2][Nx+2], xP[Ny+2][Nx+2],         // Mesh
       yP[Ny+2][Nx+2], vP[Ny+2][Nx+2];
double u[Ny+2][Nx+2], v[Ny+2][Nx+2];                             // Velocity components

// Function to generate the mesh and define position of nodes, faces and volume of control volumes
void MeshDefinition(double H, double L, double W, double Dx, double Dy) {
    for (int i = 0; i < Ny+2; i++) {
        for (int j = 0; j < Nx+2; j++) {
            if (i == 0 && j == 0)
                xcv[i][j] = -L / 2,
                ycv[i][j] = H,
                xP[i][j] = -L / 2,
                yP[i][j] = H,
                vP[i][j] = 0;
            else if (i == 0 && j == Nx+1)
                xcv[i][j] = L / 2,
                ycv[i][j] = H,
                xP[i][j] = L / 2,
                yP[i][j] = H,
                vP[i][j] = 0;
            else if (i == Ny+1 && j == 0)
                xcv[i][j] = -L / 2,
                ycv[i][j] = 0,
                xP[i][j] = -L / 2,
                yP[i][j] = 0,
                vP[i][j] = 0;
            else if (i == Ny+1 && j == Nx+1)
                xcv[i][j] = L / 2,
                ycv[i][j] = 0,
                xP[i][j] = L / 2,
                yP[i][j] = 0,
                vP[i][j] = 0;
            else if (i == 0)
                xcv[i][j] = -L / 2 + Dx * j,
                ycv[i][j] = H,
                xP[i][j] = (xcv[i][j] + xcv[i][j-1]) / 2,
                yP[i][j] = H,
                vP[i][j] = 0;
            else if (j == 0)
                xcv[i][j] = -L / 2,
                ycv[i][j] = H - Dy * i,
                xP[i][j] = -L / 2,
                yP[i][j] = (ycv[i][j] + ycv[i-1][j]) / 2,
                vP[i][j] = 0;
            else if (j == Nx+1)
                xcv[i][j] = L / 2,
                ycv[i][j] = H - Dy * i,
                xP[i][j] = L / 2,
                yP[i][j] = (ycv[i][j] + ycv[i-1][j]) / 2,
                vP[i][j] = 0;
            else if (i == Ny+1)
                xcv[i][j] = -L / 2 + Dx * j,
                ycv[i][j] = 0,
                xP[i][j] = (xcv[i][j] + xcv[i][j-1]) / 2,
                yP[i][j] = 0,
                vP[i][j] = 0;
        }
    }
    for (int i = 1; i < Ny+1; i++) {
        for (int j = 1; j < Nx+1; j++) {
            xcv[i][j] = -L / 2 + Dx * j;
            ycv[i][j] = H - Dy * i;
            xP[i][j] = (xcv[i][j] + xcv[i][j-1]) / 2;
            yP[i][j] = (ycv[i][j] + ycv[i-1][j]) / 2;
            vP[i][j] = W * fabs(xcv[i][j] - xcv[i][j-1]) * fabs(ycv[i][j] - ycv[i-1][j]);
        }
    }
}
```

```
// Function to assign velocity value at each face
void VelocityField() {
    for (int i = 0; i < Ny+2; i++) {
        for (int j = 0; j < Nx+2; j++) {
            u[i][j] = 2 * yP[i][j] * (1 - pow(xcv[i][j],2));     // ue of node P, if I want uw I'll
                write u[i][j-1]
            v[i][j] = -2 * xP[i][j] * (1 - pow(ycv[i][j],2));    // vs of node P, if I want vn I'll
                write u[i-1][j]
        }
    }
}

// Base Class to implement multiple methods
class InternalNodesCalculator {
public:
    virtual void computeCoefficients(double rho, double dt, double gamma) = 0;  // Pure virtual
        function
    virtual ~InternalNodesCalculator() = default;                                // Virtual destructor
};

// Central Differencing Scheme (CDS) method
class CDSMethod : public InternalNodesCalculator {
    // Now computeCoefficients() is a method of the class, which can be called with CDSMethod object
public:
    void computeCoefficients(double rho, double dt, double gamma) override {
        for (int i = 1; i < Ny + 1; i++) {
            for (int j = 1; j < Nx + 1; j++) {

                double DSe = (ycv[i - 1][j] - ycv[i][j]) * L;
                double DSw = (ycv[i - 1][j] - ycv[i][j]) * L;
                double DSn = (xcv[i][j] - xcv[i][j - 1]) * L;
                double DSs = (xcv[i][j] - xcv[i][j - 1]) * L;

                double dPE = xP[i][j + 1] - xP[i][j];
                double dPW = xP[i][j] - xP[i][j - 1];
                double dPN = yP[i - 1][j] - yP[i][j];
                double dPS = yP[i][j] - yP[i + 1][j];

                aE[i][j] = DSe / vP[i][j] * (-rho * u[i][j] / 2 + gamma / dPE);
                aW[i][j] = DSw / vP[i][j] * (rho * u[i][j - 1] / 2 + gamma / dPW);
                aN[i][j] = DSn / vP[i][j] * (-rho * v[i - 1][j] / 2 + gamma / dPN);
                aS[i][j] = DSs / vP[i][j] * (rho * v[i][j] / 2 + gamma / dPS);
                aP[i][j] = rho / dt;
                bP[i][j] = rho / dt + DSe / vP[i][j] * (-rho * u[i][j] / 2 - gamma / dPE) +
                                                      DSw / vP[i][j] * (rho * u[i][j - 1] / 2 - gamma / dPW
                                                          ) +
                                                      DSn / vP[i][j] * (-rho * v[i - 1][j] / 2 - gamma /
                                                          dPN) +
                                                      DSs / vP[i][j] * (rho * v[i][j] / 2 - gamma / dPS);
            }
        }
    }
};

// Upwind Differencing Scheme (UDS) method
class UDSMethod : public InternalNodesCalculator {
    // Now computeCoefficients() is another method of the class, which can be called with UDSMethod
        object
public:
    void computeCoefficients(double rho, double dt, double gamma) override {
        for (int i = 1; i < Ny + 1; i++) {
            for (int j = 1; j < Nx + 1; j++) {

                double DSe = (ycv[i - 1][j] - ycv[i][j]) * L;
                double DSw = (ycv[i - 1][j] - ycv[i][j]) * L;
                double DSn = (xcv[i][j] - xcv[i][j - 1]) * L;
                double DSs = (xcv[i][j] - xcv[i][j - 1]) * L;

                double dPE = xP[i][j + 1] - xP[i][j];
                double dPW = xP[i][j] - xP[i][j - 1];
                double dPN = yP[i - 1][j] - yP[i][j];
                double dPS = yP[i][j] - yP[i + 1][j];

                aE[i][j] = DSe / vP[i][j] * (max(-rho * u[i][j], 0.0) + gamma / dPE);
                aW[i][j] = DSw / vP[i][j] * (max(rho * u[i][j - 1], 0.0) + gamma / dPW);
                aN[i][j] = DSn / vP[i][j] * (max(-rho * v[i - 1][j], 0.0) + gamma / dPN);
                aS[i][j] = DSs / vP[i][j] * (max(rho * v[i][j], 0.0) + gamma / dPS);
                aP[i][j] = rho / dt;
                bP[i][j] = rho / dt + DSe / vP[i][j] * (min(-rho * u[i][j], 0.0) - gamma / dPE) +
                                                      DSw / vP[i][j] * (min(rho * u[i][j - 1], 0.0) - gamma
                                                          / dPW) +
                                                      DSn / vP[i][j] * (min(-rho * v[i - 1][j], 0.0) -
                                                          gamma / dPN) +
                                                      DSs / vP[i][j] * (min(rho * v[i][j], 0.0) - gamma /
                                                          dPS);
            }
        }
    }
};

// Function to choose the method
InternalNodesCalculator* createCalculator(string method) {
    if (method == "CDS") {
        return new CDSMethod();
    } else if (method == "UDS") {
        return new UDSMethod();
    } else {
        cerr << "Not a valid method!" << std::endl;
```

```
        return nullptr;
    }
}

// Function to apply boundary conditions
void BoundaryConditions(double alpha) {
    for (int i = 0; i < Ny+2; i++) {
        for (int j = 0; j < Nx+2; j++) {
            if (i == Ny+1 && xP[i][j] < 0)                                  // Inlet Condition
                aE[i][j] = 0,
                aW[i][j] = 0,
                aN[i][j] = 0,
                aS[i][j] = 0,
                aP[i][j] = 1,
                bP[i][j] = 1 + tanh((2 * xP[i][j] + 1) * alpha);
            else if (i == Ny+1 && xP[i][j] > 0)                             // Outlet Condition
                aE[i][j] = 0,
                aW[i][j] = 0,
                aN[i][j] = 1,
                aS[i][j] = 0,
                aP[i][j] = 1,
                bP[i][j] = 0;
            else if (i == 0 || j == 0 || j == Nx+1)                         // Rest of the boundaries
                aE[i][j] = 0,
                aW[i][j] = 0,
                aN[i][j] = 0,
                aS[i][j] = 0,
                aP[i][j] = 1,
                bP[i][j] = 1 - tanh(alpha);
        }
    }
}

void Solver(double maxRes, double &t_count, double dt, double rho, double gamma) {
    // Time Loop
    double res = maxRes + 1; // Condition to enter the loop
    while (res > maxRes) {
        double maxDiff = 0.0;

        for (int i = 0; i < Ny + 2; i++) {
            for (int j = 0; j < Nx + 2; j++) {
                if (i == 0 || i == Ny + 1 || j == 0 || j == Nx + 1) {
                    phi_1[i][j] = (aE[i][j] * phi[i][j + 1] + aW[i][j] * phi[i][j - 1] +
                                   aN[i][j] * phi[i - 1][j] + aS[i][j] * phi[i + 1][j] + bP[i][j]) /
                                      aP[i][j];
                } else {
                    phi_1[i][j] = (aE[i][j] * phi[i][j + 1] + aW[i][j] * phi[i][j - 1] +
                                   aN[i][j] * phi[i - 1][j] + aS[i][j] * phi[i + 1][j] + phi[i][j] *
                                      bP[i][j]) / aP[i][
                                     j];
                }

                double diff = fabs(phi_1[i][j] - phi[i][j]);
                if (diff > maxDiff) {
                    maxDiff = diff;
                }
            }
        }
        t_count += dt;
        res = maxDiff;

        // Update phi
        for (int i = 0; i < Ny + 2; i++) {
            for (int j = 0; j < Nx + 2; j++) {
                phi[i][j] = phi_1[i][j];
            }
        }
    }
    std::cout << "Steady-state-reached-in-" << t_count << "-seconds" << std::endl;
}

int main() {

    // Physical Data
    double phi_0 = 0.0;                             // Initial condition in all the domain
    double gamma = 1.0;                             // Assumed equal to 1
    double ratio = 1000000;                             // To be changed with 10, 1.000, 1.000.000
    double rho = gamma * ratio;                     // Uniform in the domain
    double alpha = 10;

    // Numerical Data
    double Dx = L / Nx, Dy = H / Ny;
    double dt = 1e-4;                               // Time step
    double t_count = 0.0;                           // Initial time count
    double maxRes = 1e-6;                           // Convergence tolerance
    string method;                                  // To choose the scheme

    // Start Timer
    auto start = high_resolution_clock::now();

    // Compute Mesh
    MeshDefinition(H, L, W, Dx, Dy);

    // Compute Velocity Field
    VelocityField();

    // Initial Map
    for (auto &row: phi) {
        for (auto &elem: row) {
```

```
            elem = phi_0;
        }
    }

    //Choose Interpolation Scheme
    cout << "Choose the method ('CDS' or 'UDS'): ";
    cin >> method;

    // Create an instance of the chosen method
    InternalNodesCalculator* calculator = createCalculator(method);
    if (!calculator) {
        cerr << "Error: Discretization not valid!" << endl;
        return 1;
    }

    // Compute Internal Nodes Coefficients
    calculator->computeCoefficients(rho, dt, gamma);

    // Compute Boundary Conditions
    BoundaryConditions(alpha);

    // Loop Until Steady-State is Reached
    Solver(maxRes, t_count, dt, rho, gamma);

    // Clear memory
    delete calculator;

    // File to check matrices
    ofstream TestFile ("Test.txt");
    for (int i = 0; i < Ny+2; i++) {
        for (int j = 0; j < Nx+2; j++) {
            TestFile << xP[i][j] << "-";
        }
        TestFile << endl;
    }

    // File to print values of phi at the outlet
    ofstream TransportedProperty ("TransportedProperty.txt");
    TransportedProperty << "X" << "-" << "Ratio=" << ratio << endl;
    for (int i = 0; i < Ny+2; i++) {
        for (int j = 0; j < Nx+2; j++) {
            if (i == Ny && xcv[i][j] >= 0)
                TransportedProperty << xcv[i][j] << "-" << (phi_1[i][j] + phi_1[i][j+1]) / 2 << endl;
        }
    }

    // File to print phi as a function of x
    ofstream PhiFunctionX ("PhiFx.txt");
    for (int i = 0; i < Ny+2; i++) {
        for (int j = 0; j < Nx+2; j++) {
            if (i == Ny)
                PhiFunctionX << xcv[i][j] << "-" << (phi_1[i][j] + phi_1[i][j+1]) / 2 << endl;
        }
    }

    // File to print data for Gnuplot 1
    ofstream GnuplotData1 ("GnuplotData1.txt");
    for (int i = 0; i < Ny+2; i++) {
        for (int j = 0; j < Nx+2; j++) {
           GnuplotData1 << xP[i][j] << "-" << yP[i][j] << "-" << phi_1[i][j] << endl;
        }
        GnuplotData1 << "\n";
    }

    // File to print data for Gnuplot 2
    ofstream GnuplotData2 ("GnuplotData2.txt");
    double ref[11][4] = {
        {0.0, 1.989, 2.0000, 2.000},
        {0.1, 1.402, 1.9990, 2.000},
        {0.2, 1.146, 1.9997, 2.000},
        {0.3, 0.946, 1.9850, 1.999},
        {0.4, 0.775, 1.8410, 1.964},
        {0.5, 0.621, 0.9510, 1.000},
        {0.6, 0.480, 0.1540, 0.036},
        {0.7, 0.349, 0.0010, 0.001},
        {0.8, 0.227, 0.0000, 0.000},
        {0.9, 0.111, 0.0000, 0.000},
        {1.0, 0.000, 0.0000, 0.000}
    };
    // Prints automatically the correct column
    int index = 1;
    if (ratio == 1000)
        index = 2;
    else if (ratio == 1000000)
        index = 3;
    for (int i = 0; i < 11; i++) {
        GnuplotData2 << ref[i][0] << "-" << ref[i][index] << std::endl;
    }

    // File to print property distribution
    ofstream PhiDistribution ("PhiDistribution.txt");
    for (int i = 0; i < Ny+2; i++) {
        for (int j = 0; j < Nx+2; j++) {
            PhiDistribution << phi_1[i][j] << "-";
        }
        PhiDistribution << endl;
    }

    TestFile.close();
```

```
    TransportedProperty.close();
    GnuplotData1.close();
    GnuplotData2.close();
    PhiDistribution.close();
    PhiFunctionX.close();

    // Generate Gnuplot script 1
    ofstream Gnuplot1("MagnitudeMap.plt");
    Gnuplot1 << "set terminal pngcairo size 1000,500 enhanced font 'Arial,12'\n";
    Gnuplot1 << "set output 'MagnitudeMap_Plot.png'\n";
    Gnuplot1 << "set pm3d map\n";
    Gnuplot1 << "set palette defined ("
        << "0 'dark-blue',"
        << "0.2 'blue',"
        << "0.4 'cyan',"
        << "0.7 'orange',"
        << "1 'yellow')\n";
    Gnuplot1 << "set colorbox\n";
    Gnuplot1 << "set xlabel 'X (m)'\n";
    Gnuplot1 << "set ylabel 'Y (m)'\n";
    Gnuplot1 << "set title 'Steady state of ϕ with {/Symbol r}/{/Symbol G} = " << ratio << "' font '
        Arial,20'\n";
    Gnuplot1 << "set xrange [-1:1]\n";
    Gnuplot1 << "set yrange [0:1]\n";
    Gnuplot1 << "set autoscale\n";
    Gnuplot1 << "set cbrange [*:*]\n";
    Gnuplot1 << "splot 'GnuplotData1.txt' using 1:2:3 with pm3d notitle\n";
    Gnuplot1.close();

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

    // Generate Gnuplot script 2
    ofstream Gnuplot2("ComparisonPlot.plt");
    Gnuplot2 << "set terminal pngcairo size 600,400\n";
    Gnuplot2 << "set output 'Comparison_Plot.png'\n";
    Gnuplot2 << "set xlabel 'x'\n";
    Gnuplot2 << "set ylabel 'ϕ'\n";
    Gnuplot2 << "set title 'ϕ = f(x) with {/Symbol r}/{/Symbol G} = " << ratio << "' font 'Arial,20'\n
        ";
    Gnuplot2 << "set key top left\n";
    Gnuplot2 << "set xrange [-1:1]\n";
    Gnuplot2 << "set yrange [0:2]\n";
    Gnuplot2 << "plot 'PhiFx.txt' with lines lw 2 lc rgb 'blue' title 'Result',"
             << "'GnuplotData2.txt' with points pt 7 ps 1.2 lc rgb 'red' title 'Reference'\n";
    Gnuplot2.close();

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

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

    return 0;
}
```