Simulation of Reactive Multi-Component Flow using SIMPLE Algorithm.

by

Arshia Fazeli

ME663 Individual Project

Waterloo, Ontario, Canada, 2024

Abstract

Multi-component reactive flows are prevalent in chemical and industrial applications such as reactors, fuel cells and combustion processes. Understanding the hydrodynamic behaviour of these flows allows us to enhance the predictive capabilities in process design and control. This enables the development of safer, cleaner and more efficient processes. The simplified reaction models that are extensively used in industry such as continuous stirred tank reactor (CSTR) and plug-flow reactor (PFR) models assume perfect mixing in the tank and reactor's cross-section respectively. Both these models are one-dimensional and do not account for the hydrodynamics which might have a significant impact on the distribution of reactants and products.

In this work, an incompressible Navier Stokes (INS) solver using the semi-implicit method for pressure-linked equations (SIMPLE) algorithm is coupled with a solver for convection-diffusion equation with reaction source to predict distributions of components within a reactive fluid. Upwind, hybrid and QUICK convection schemes and 2nd-order and 4th-order diffusion schemes are implemented for both INS and convection-diffusion equation schemes. The INS solver is tested on the lid-driven cavity test case by Ghia to validate the accuracy of this solver. It has been shown that there is a good agreement between Ghia's data and the simulation results where the QUICK convection scheme is used. Furthermore, the concentration profiles from the lid-driven cavity simulation are compared to the simulation results by `openCMP` which is a computational multiphysics software package based on the finite element method.

Additionally, the developed solver is used to simulate a 2D plug-flow reactor test case. The simulation results are averaged over the reactor's cross-section and the resulting profiles are compared to the analytical solutions by PFR model.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Multi-component reactive flows have several applications in chemical reactors, fuel cells and combustion processes. Understanding the hydrodynamic behaviour of these flows allows us to enhance the predictive capabilities in reactor design and process optimization. This enables the development of safer, cleaner and more efficient chemical processes. [1] The simplified reaction models that are extensively used in industry such as continuous stirred tank reactor (CSTR) and plug-flow reactor (PFR) models assume perfect mixing in the tank and reactor's cross-section respectively.[1] However, this is not the case in numerous industrial applications which brings the importance of consideration of the hydrodynamics for modelling this kind of multi-component reactive flows.

In this project, an incompressible Navier-Stokes solver will be implemented to evaluate the velocity and pressure profiles within the flow. This hydrodynamic solver will be coupled to a solver for the component transfer equation (convection-diffusion) with reaction source terms to evaluate the concentration distribution within the system.

For simplicity, the developed solver will be tested and validated for a system with two dissolved components A and B. Component A is a reactant and component B is the product

and the reaction is a first-order reaction.

$$\begin{aligned} A \rightarrow B \qquad & k = 0.5s^{-1} \\ \text{reaction rate :} \qquad & r = -kc_A \end{aligned} \tag{1.1}$$

In this case, the governing equations for each component's transfer are as follows.

$$\boldsymbol{\nabla} \cdot (\mathbf{u}c_A) - D_A \nabla^2 c_A = -c_A \tag{1.2}$$

$$\boldsymbol{\nabla} \cdot (\mathbf{u}c_B) - D_B \nabla^2 c_B = c_A \tag{1.3}$$

Where the following boundary conditions apply:

$$c_A = 1, c_B = 0 \quad \text{at inlet} \tag{1.4}$$

$$\nabla c_A = 0, \nabla c_B = 0 \quad \text{at the other boundaries} \tag{1.5}$$

1.2 Objectives

The objectives of this project are as follows:

1. Develop a steady-state numerical solver based on the semi-implicit method for pressure-linked Equations (SIMPLE) algorithm to solve the 2D incompressible Navier-Stokes equations.

2. Integrate the solver to include the convection-diffusion equation with reaction source term to analyze the concentration distribution of two dissolved reacting species A and B.

3. Test the developed solver in a uniformly meshed 2D rectangular grid with unidirectional flow to predict the concentration distribution of components A and B.

Chapter 2

Numerical Solver Implementation

2.1 Discretization

2.1.1 Conservation of Momentum Equations

The discretized conservation of momentum equation system for the staggered grid:

$$(A_P^u)_{i,j}u_{i,j} = \sum_{nb}(A_{nb}^u)_{i,j}u_{i,j} + (p_{i,j} - p_{i,j+1})\Delta y \tag{2.1}$$

$$(A_P^u)_{i,j-1}u_{i,j-1} = \sum_{nb}(A_{nb}^u)_{i,j-1}u_{i,j-1} + (p_{i,j-1} - p_{i,j})\Delta y \tag{2.2}$$

$$(A_P^v)_{i,j}v_{i,j} = \sum_{nb}(A_{nb}^v)_{i,j}v_{i,j} + (p_{i,j} - p_{i-1,j})\Delta x \tag{2.3}$$

$$(A_P^v)_{i+1,j}v_{i+1,j} = \sum_{nb}(A_{nb}^v)_{i+1,j}v_{i+1,j} + (p_{i+1,j} - p_{i,j})\Delta x \tag{2.4}$$

Where the convective (C_{nb}) and diffusive (D_{nb}) contributions of the neighbouring coefficients ($A_{nb} = C_{nb} + D_{nb}$) can be found using different schemes listed in tables 2.1 and 2.2.

The coefficients at the cell (A_p^ϕ) can be found by summation of the neighbouring coef-

C_{nb}	UDS	QUICK
C_E	$dy \max(0, -u^*{}_e)$	$-dy\,(-0.75\max(0, -u^*{}_e) + 0.375\max(0, u^*{}_e) - 0.125\max(0, -u^*{}_w))$
C_W	$dy \max(0, u^*{}_w)$	$-dy\,(-0.125\max(0, u^*{}_e) + 0.375\max(0, -u^*{}_w) - 0.75\max(0, u^*{}_w))$
C_N	$dx \max(0, -v^*{}_n)$	$-dx\,(-0.75\max(0, -u^*{}_n) + 0.375\max(0, u^*{}_n) - 0.125\max(0, -u^*{}_s))$
C_S	$dx \max(0, v^*{}_s)$	$-dx\,(-0.125\max(0, u^*{}_n) + 0.375\max(0, -u^*{}_s) - 0.75\max(0, u^*{}_s))$
C_{EE}	0	$-0.125dy \max(0, -u^*{}_e)$
C_{WW}	0	$-0.125dy \max(0, u^*{}_w)$
C_{NN}	0	$-0.125dx \max(0, -u^*{}_n)$
C_{SS}	0	$-0.125dx \max(0, u^*{}_s)$

Table 2.1: Convective contributions of neighbour coefficients with UDS and QUICK schemes.

ficients. [2]

$$A_p^\phi = \sum_{nb} A_{nb}^\phi \tag{2.5}$$

In the SIMPLE method, we express every unknown variable ϕ as a summation of correction ϕ' and previous iteration ϕ^* terms. The above system of equations becomes:

$$(A_P^u)_{i,j} u'_{i,j} = \sum_{nb} (A_{nb}^u)_{i,j} u'_{i,j} + (p'_{i,j} - p'_{i,j+1})\Delta y + R_{i,j}^u \tag{2.6}$$

$$(A_P^u)_{i,j-1} u'_{i,j-1} = \sum_{nb} (A_{nb}^u)_{i,j-1} u'_{i,j-1} + (p'_{i,j-1} - p'_{i,j})\Delta y + R_{i,j-1}^u \tag{2.7}$$

$$(A_P^v)_{i,j} v'_{i,j} = \sum_{nb} (A_{nb}^v)_{i,j} v'_{i,j} + (p'_{i,j} - p'_{i-1,j})\Delta x + R_{i,j}^v \tag{2.8}$$

$$(A_P^v)_{i+1,j} v'_{i+1,j} = \sum_{nb} (A_{nb}^v)_{i+1,j} v'_{i+1,j} + (p'_{i+1,j} - p'_{i,j})\Delta x + R_{i+1,j}^u \tag{2.9}$$

We cancel out the neighbouring terms and the residual term since as $\phi' \to 0$ (or as solution converges), the residual term also becomes zero $R^\phi \to 0$. The final form of the momentum equations is as follows.

C_{nb}	2nd order	4th order	A_{nb}	Hybrid
D_E	$\frac{\nu dy}{dx}$	$\frac{4dy\nu}{3dx}$	A_E	$dy \max\left(0, -u^*{}_e, \frac{\nu}{dx} - u^*{}_e\right)$
D_W	$\frac{\nu dy}{dx}$	$\frac{4dy\nu}{3dx}$	A_W	$dy \max\left(0, u^*{}_w, \frac{\nu}{dx} - u^*{}_w\right)$
D_N	$\frac{\nu dx}{dy}$	$\frac{4dx\nu}{3dy}$	A_N	$dx \max\left(0, -v^*{}_n, \frac{\nu}{dy} - u^*{}_n\right)$
D_S	$\frac{\nu dx}{dy}$	$\frac{4dy\nu}{3dx}$	A_S	$dx \max\left(0, v^*{}_s, \frac{\nu}{dy} - u^*{}_s\right)$
D_{EE}	0	$-\frac{dy\nu}{12dx}$	A_{EE}	0
D_{WW}	0	$-\frac{dy\nu}{12dx}$	A_{WW}	0
D_{NN}	0	$-\frac{dx\nu}{12dy}$	A_{NN}	0
D_{SS}	0	$-\frac{dx\nu}{12dy}$	A_{SS}	0

Table 2.2: Diffusive contributions of neighbour coefficients with 2nd order and 4th order finite difference methods. Additionally, the total coefficient for the hybrid scheme can be found in this table.

$$u'_{i,j} = \frac{\Delta y}{(A^u_P)_{i,j}}(p'_{i,j} - p'_{i,j+1}) \tag{2.10}$$

$$u'_{i,j-1} = \frac{\Delta y}{(A^u_P)_{i,j-1}}(p'_{i,j-1} - p'_{i,j}) \tag{2.11}$$

$$v'_{i,j} = \frac{\Delta x}{(A^v_P)_{i,j}}(p'_{i,j} - p'_{i-1,j}) \tag{2.12}$$

$$v'_{i+1,j} = \frac{\Delta x}{(A^v_P)_{i+1,j}}(p'_{i+1,j} - p'_{i,j}) \tag{2.13}$$

2.1.2 Conservation of Mass Equation

To derive the discretized continuity equation, we integrate it over the p control volume.

$$(u'_{i,j} - u'_{i,j-1})dy + (v'_{i,j} - v'_{i+1,j})dx = R^c_{i,j} \tag{2.14}$$

$$R^c_{i,j} = -[(u^*_{i,j} - u^*_{i,j-1})dy + (v^*_{i,j} - v^*_{i+1,j})dx] \tag{2.15}$$

The correction terms in eq. (2.14), can be replaced by eqs. (2.10) to (2.13) to obtain

the pressure correction pressure-correction equation.

$$(A_P^p)_{i,j}(p'_{i,j}) = \sum_{nb}(A_{nb}^p)_{i,j}(p'_{nb}) + R_{i,j}^c \tag{2.16}$$

Where:

$$(A_E^p)_{i,j} = \frac{(dy)^2}{(A_P^u)_{i,j}} \quad (A_W^p)_{i,j} = \frac{(dy)^2}{(A_P^u)_{i,j}} \tag{2.17}$$

$$(A_N^p)_{i,j} = \frac{(dx)^2}{(A_P^v)_{i,j}} \quad (A_{np}^p)_{i,j} = \frac{(dx)^2}{(A_P^v)_{i,j}} \tag{2.18}$$

$$(A_P^p)_{i,j} = \sum_{nb}(A_{nb}^p)_{i,j} \tag{2.19}$$

2.1.3 Conservation of Mass of Species

The conservation of mass for each component k can be written as follows:

$$(A_P^c)_{i,j}(c_k)_{i,j} = \sum_{nb}(A_{nb}^c)_{i,j}(c_k)_{i,j} + r(c_k)_{i,j}\Delta y\Delta x \tag{2.20}$$

Where $r(c_k)$ corresponds to the net rate of generation/consumption of component k. Partitioning c_k into the linearized and correction terms, we get:

$$(A_P^c)_{i,j}(c'_k)_{i,j} = \sum_{nb}(A_{nb}^c)_{i,j}(c'_k)_{i,j} + r(c'_k)_{i,j}\Delta y\Delta x + R_{i,j}^k \tag{2.21}$$

$$R_{i,j}^k = \sum_{nb}(A_{nb}^c)_{i,j}(c_k^*)_{i,j} - (A_P^c)_{i,j}(c_k^*)_{i,j} + r(c_k^*)_{i,j}\Delta y\Delta x$$

The neighbouring coefficients for species balance equations can be found using the expressions in tables 2.1 and 2.2. The only difference is that the diffusivity of each component should be replaced by the kinematic viscosity in table table 2.2. However, the convective contributions are exactly the same.

2.2 Boundary Conditions

The boundary conditions used for the lid-driven cavity and plug-flow reactor test cases can be classified into the following three categories.

$$\begin{aligned} &\text{walls}: \quad u = 0, v = 0, \frac{\partial c_k}{\partial n} = 0, \frac{\partial p}{\partial n} = 0 \\ &\text{outlet}: \quad \frac{\partial u}{\partial n} = 0, \frac{\partial v}{\partial n} = 0, \frac{\partial c_k}{\partial n}, \frac{\partial p}{\partial n} = 0 \\ &\text{inlet}: \quad u = u_{in}, v = v_{in}, c_k = (c_k)_{in}, \frac{\partial p}{\partial n} = 0 \end{aligned} \tag{2.22}$$

Hence, there is a combination of Dirichlet and Neumann boundary conditions. The general framework for imposing each of these boundary conditions will be discussed in this section.

2.2.1 Dirichlet Boundary Conditions

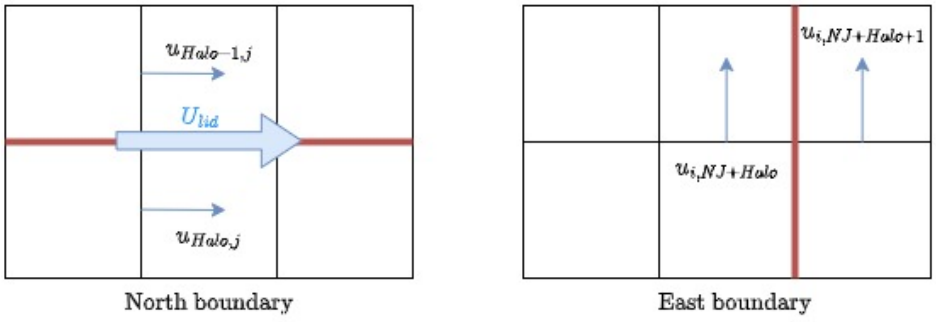


Figure 2.1: Dirichlet boundary conditions for u at the north boundary and for v at the East boundary

For the lid-driven cavity test case, the no-slip velocity boundary condition is imposed at every physical boundary in the domain. The u and v velocities are zero at the south, east and west boundaries. The u velocity at the north boundary is equal to a constant velocity U_{lid}. Since there are no u nodes at this boundary, we do a linear interpolation using two adjacent cells to the boundary. According to fig. 2.1:

$$\frac{u_{Halo,j} + u_{Halo-1,j}}{2} = U_{lid} \rightarrow u_{Halo-1,j} = 2 - u_{Halo,j}$$

For the concentrations at the north boundary, we got:

$$\frac{(c_k)_{Halo,j} + (c_k)_{Halo-1,j}}{2} = (c_k)_{in}$$

If we assume to have to components where A is the reactant with $(c_A)_{in} = 1$ and B is a product, the above equation becomes:

$$(c_A)_{Halo-1,j} = 2 - (c_A)_{Halo,j}$$
$$(c_B)_{Halo-1,j} = -(c_B)_{Halo,j}$$

Similarly to have zero v-velocity at the east boundary, we impose:

$$\frac{u_{i,Halo+NJ} + u_{i,Halo+NJ+1}}{2} = 0 \rightarrow u_{i,Halo+NJ+1} = -u_{i,Halo+NJ}$$

2.2.2 Neumann Boundary Conditions

According to eq. (2.22), there are numerous zero-gradient boundary conditions, particularly at the walls and outlet boundaries. To impose these Neumann boundary conditions the difference between the degree of freedom closest to the boundary and the value at the Halo cell can be set to 0. According to fig. 2.2, at the south boundary we have:

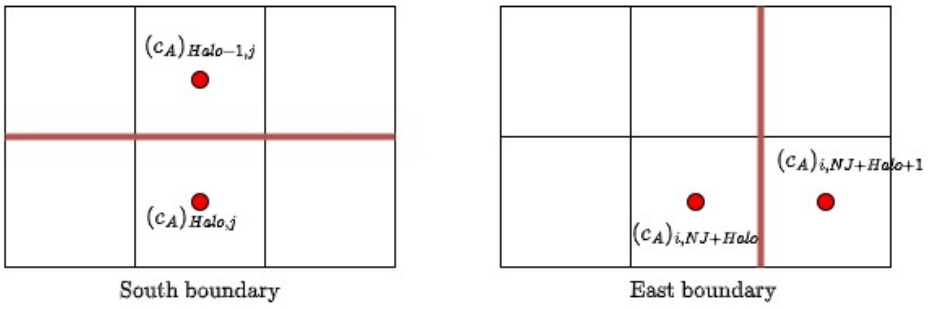


Figure 2.2: Neumann boundary conditions for c_A at the south boundary and for v at the East boundary

$$\frac{\partial c_A}{\partial n} = \frac{(c_A)_{Halo,j} - (c_k)_{Halo-1,j}}{dy} = 0 \rightarrow (c_A)_{Halo,j} = (c_k)_{Halo-1,j}$$

Similarly for the concentration of component A at the east boundary, we have:

$$\frac{(c_A)_{i,Halo+NJ} - (c_A)_{i,Halo+NJ+1}}{2} = 0 \rightarrow (c_A)_{i,Halo+NJ+1} = (c_A)_{i,Halo+NJ}$$

2.3 SIMPLE Algorithm

1. Specify initial conditions using p^*, u^*, v^*, c_k^*.

2. use Gauss-Seidel/under-relaxation to solve the conservation of momentum discretized equations. The general formulation for this method is as follows:

$$\phi_P = \alpha_\phi [\frac{\sum_{nb} A_{nb}^\phi \phi_{nb}^* + S_u^\phi}{A_P^\phi}] + (1 - \alpha_\phi)\phi_P^* \tag{2.23}$$

   Where α_ϕ is the relaxation factor.

3. calculate the mass residual using eq. (2.14).

4. Solve the pressure-correction equation eq. (2.16) using Gauss-Seidel method.

$$(p'_{i,j}) = \frac{\sum_{nb}(A_{nb}^p)_{i,j}(p'_{nb}) + R_{i,j}^c}{(A_P^p)_{i,j}} \tag{2.24}$$

5. Correct the u, v, p terms.

$$p_{i,j} = p_{i,j}^* + \alpha_p p'_{i,j} \tag{2.25}$$
$$u_{i,j} = u_{i,j}^* + \alpha_u u'_{i,j} \tag{2.26}$$
$$v_{i,j} = v_{i,j}^* + \alpha_v v'_{i,j} \tag{2.27}$$

6. Solve the conservation of mass for each of the components using GS method on the eq. (2.21).

7. Repeat steps 2-6 until the maximum residual for every variable reaches below the specified tolerance.

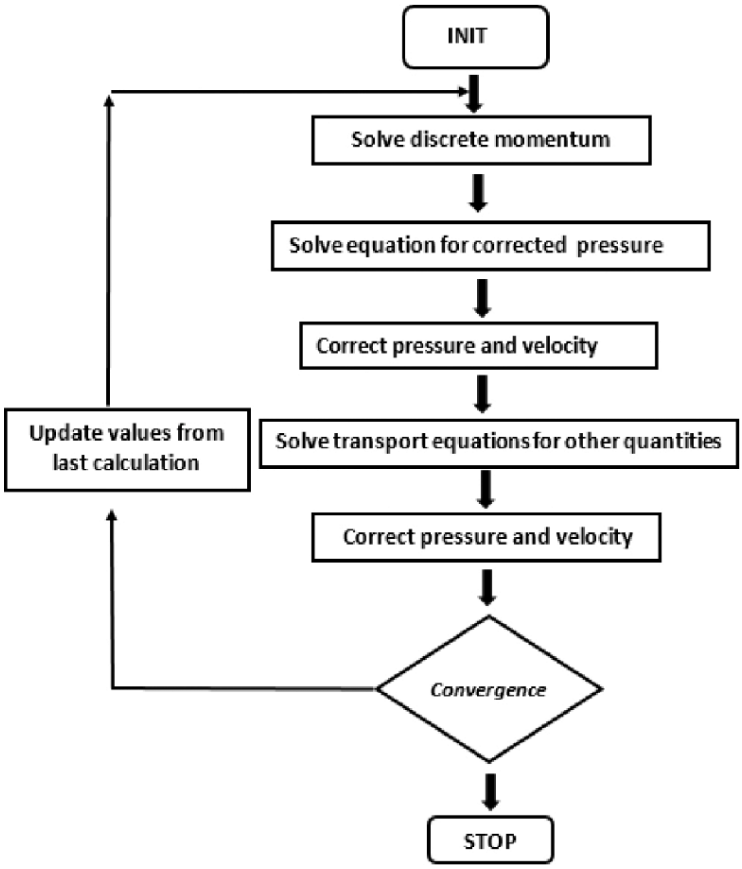


Figure 2.3: Summary of Simple Algorithm. [3]

Chapter 3

Results

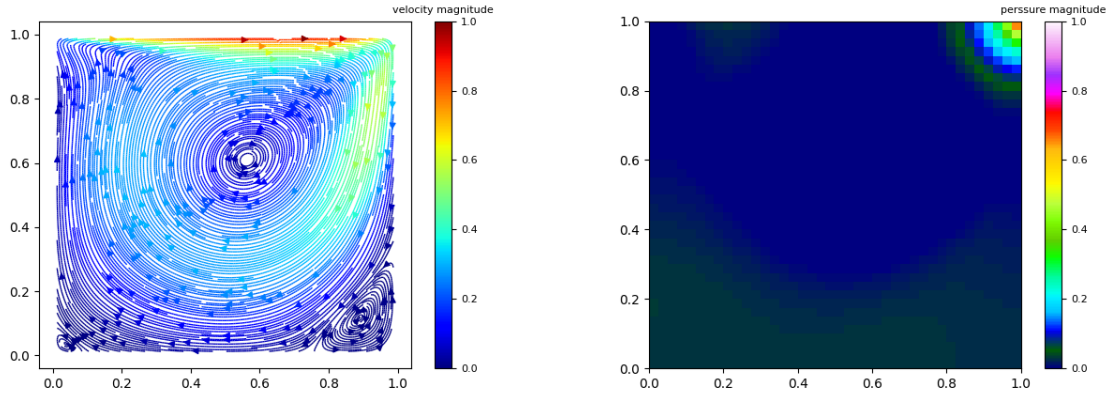
3.1 Lid-Driven Cavity

The Lid-Driven Cavity test case by Ghia [4] is selected for validation of the hydrodynamic predictions by the solver. The simulation parameters used in the majority of simulations (unless specified) are listed here:

- The tolerance is set to 0.0001.
- The number of inner loop iterations is set to 8 for the pressure-correction equation and 4 for the other equations.
- Except for the QUICK scheme at Re=1000 test case, the relaxation factor is set to 0.3 for the pressure-correction equation and 0.5 for the other equations.
- For the QUICK scheme at Re=1000 test case, the relaxation factor is set to 0.2 for the pressure-correction equation and 0.3 for the other equations. The reasons for using lower relaxation factors is that the solver is not stable with higher relaxation factors.
- The reaction rate is set to a constant value of 0.5 s^{-1} and the diffusivity of both components is set to 0.05 $m^2 s^{-1}$.

3.1.1 Streamlines and Pressure Profile

Different grid sizes, Reynolds numbers, convection and diffusion schemes are used depending on the simulation. The general streamlines and pressure profile at Re=400 and Re=1000 are shown in fig. 3.1 and fig. 3.2 respectively.



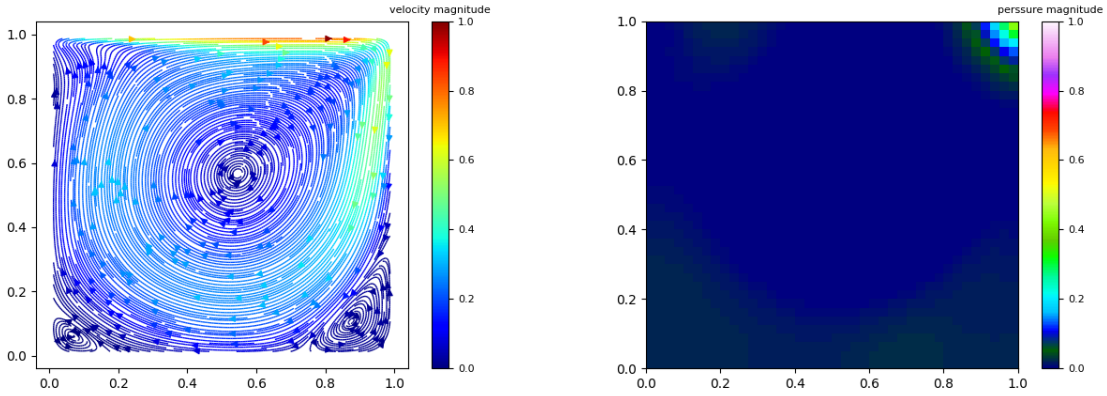
(a) streamlines

(b) pressure profile

Figure 3.1: Velocity streamlines and pressure profile at Re = 400 in the lid-driven cavity test case. The grid is a 40*40 structured grid, QUICK and 4th-order diffusion schemes are used.

3.1.2 Computation Time

The developed SIMPLE solver is compared to the SCGS solver implemented for assignment 1 in table 3.1. As it can be seen the SIMPLE algorithm solver generally converges faster than the SCGS solver. Particularly this difference is significant for finer mesh. For example, the computation time of the SIMPLE solver at the 120*120 grid is more than half the computation time for the similar simulation with the SCGS solver which is a huge improvement.



(a) streamlines

(b) pressure profile

Figure 3.2: Velocity streamlines and pressure profile at Re = 1000 in the lid-driven cavity test case. The grid is a 40*40 structured grid, QUICK and 4th-order diffusion schemes are used.

SCGS		
Grid	Re=400	Re=1000
40*40	12.89	12.25
80*80	37.08	33.57
120*120	126.49	109.67

SIMPLE		
Grid	Re=400	Re=1000
40*40	10.05	10.34
80*80	18.42	17.08
120*120	52.30	40.59

Table 3.1: CPU time in CPU.S for SIMPLE (right table) vs SCGS (left table) for convergence at different Reynolds numbers and grid numbers.

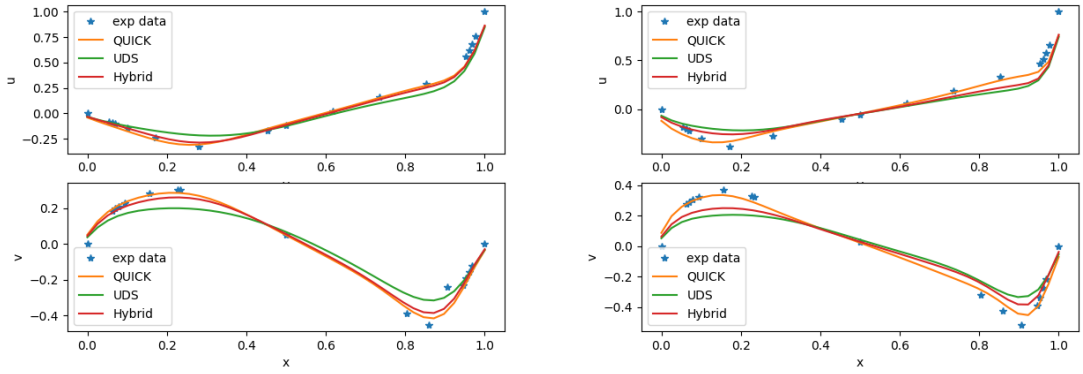
3.1.3 Validation

In this section, the simulation results obtained using different convection schemes and grids at Re=400 are compared with the data found in Ghia's [4] paper which are in one the following categories:

- The x-components of the velocity across the vertical line through the geometric centre of the cavity.
- The y-components of the velocity across the horizontal line through the geometric

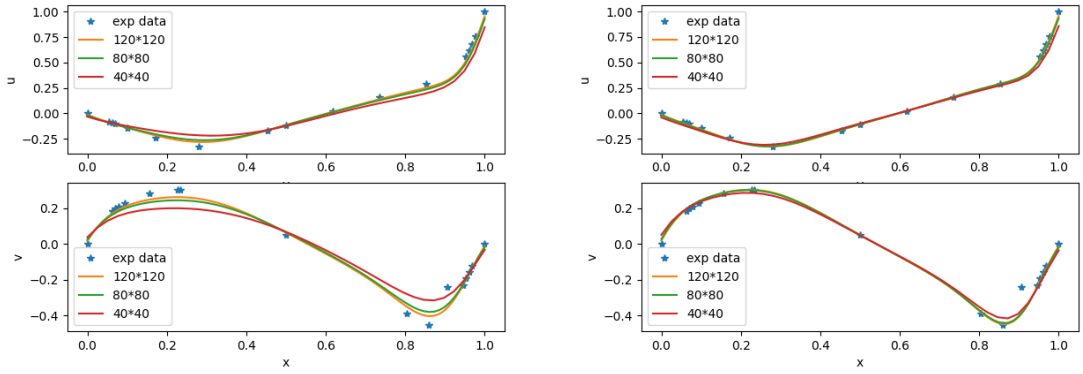
centre of the cavity.

According to fig. 3.3, the predictions by the QUICK and Hybrid scheme are in a good agreement with the experimental results. For this reason these two schemes are mostly selected over UDS for the future simulations. Furthermore fig. 3.4 shows that the grid refinement from 40*40 mesh to 80*80 causes improvement in the accuracy of results. However, using a finer mesh than 80*80 does not cause that much improvement considering how much more computationally intensive it is, in particular for the case where the QUICK scheme has been used.



(a) Re=400 (b) Re=1000

Figure 3.3: Comparison of Ghia's data at Re=400 and Re=1000 with simulation results using different convection schemes.



(a) UDS (b) QUICK

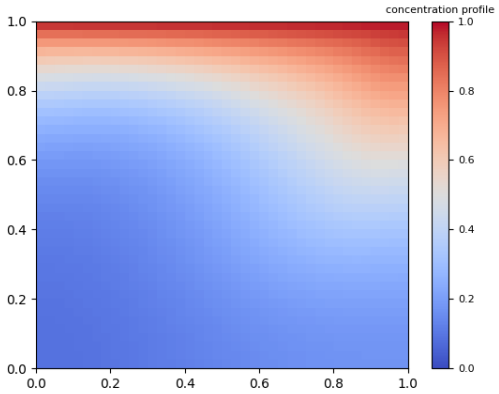
Figure 3.4: Comparison of Ghia's data at Re=400 with simulation results at different grid sizes using UDS and QUICK schemes.

3.1.4 Concentration Profiles

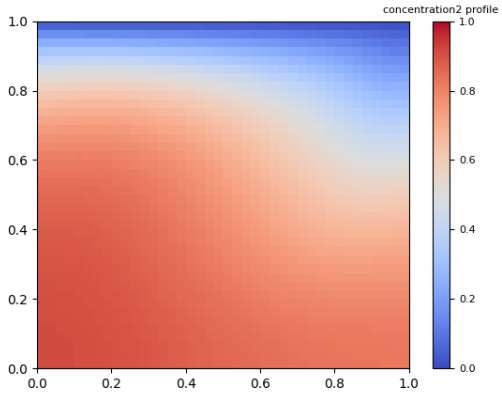
The concentration of components A (reactant) and B (product) are set to 1 and 0 at the top boundary of the computational domain respectively. On every other boundary, zero gradient boundary conditions are applied for the concentration of both components. The reaction is assumed to be non-reversible with the reaction rate constant of 0.5 s^{-1}. The concentration profile of components A and B from the SIMPLE solver can be found in fig. 3.5(a) and fig. 3.5(b) respectively.

To assess the accuracy of the results, a similar simulation has been running using `openCMP` which is a finite element method package that easily allows the user to select the order of interpolating polynomials. Hence, a solution with a high-order of accuracy (3rd order in this case) can be achieved. The results of this simulation can be found in fig. 3.5(c).

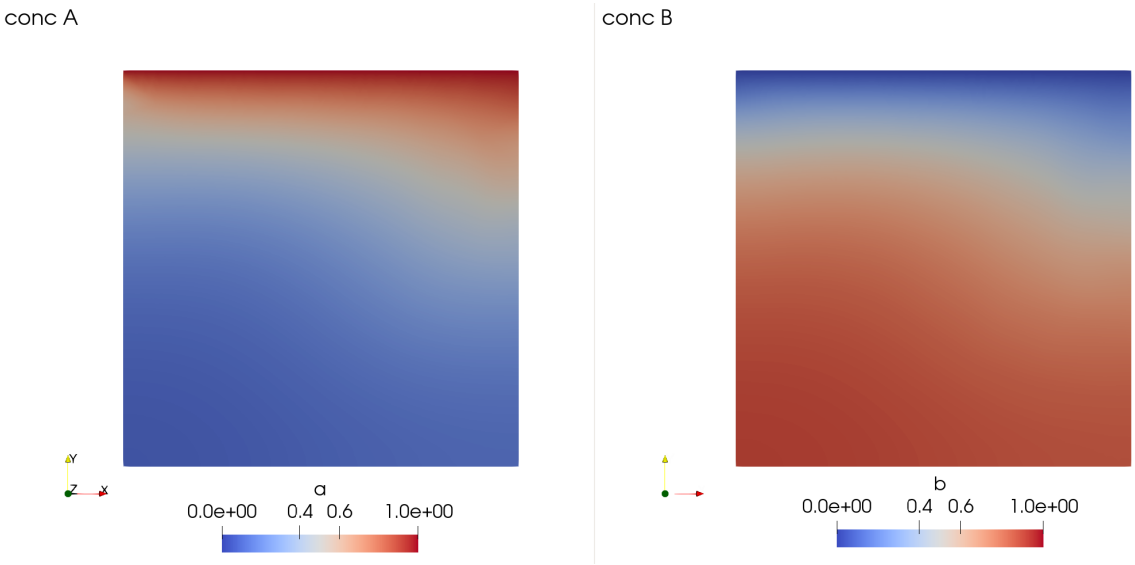
In fig. 3.5(d), the concentrations at line x=0.8 from two different simulations are compared. It can be seen that there is a relatively good general agreement between the concentration profiles particularly closer to the top boundary.



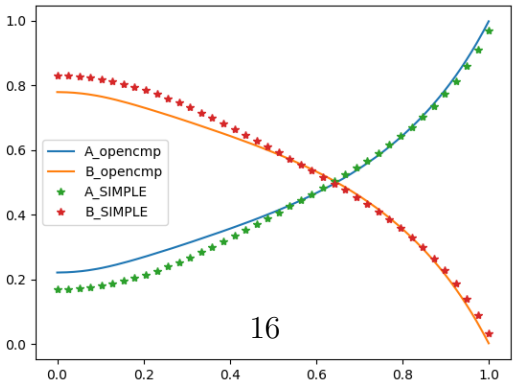
(a) c_A SIMPLE results



(b) c_B SIMPLE results



(c) `openCMP results`



(d) `Comparison`

Figure 3.5: SIMPLE solver vs openCMP results.

3.2 Plug-Flow Reactor

To run the plug-flow reactor simulation, a 2D rectangular grid with 40*80 mesh cells is created. The computational domain and the boundary conditions used can be found in fig. 3.6.

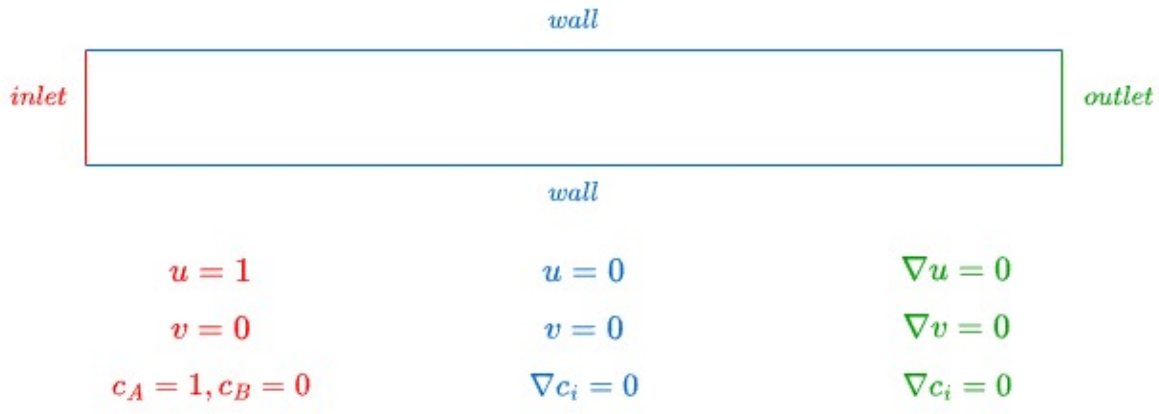
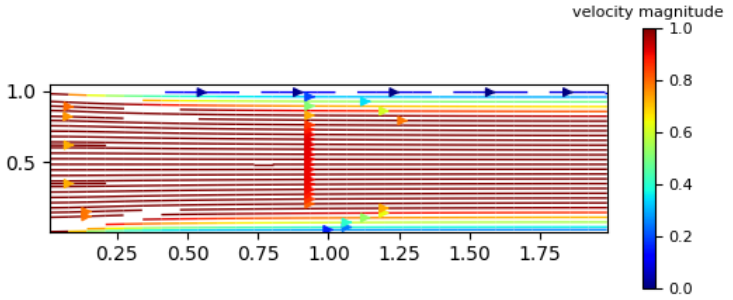


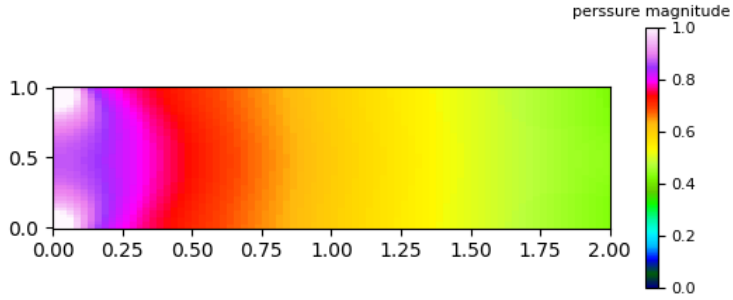
Figure 3.6: Computational domain and boundary conditions for the plug flow reactor test case.

3.2.1 Simulation

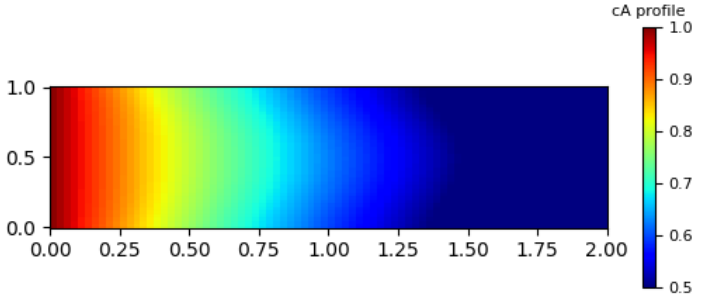
The simulation is conducted at Re=400, using QUICK convection scheme and 4th order diffusion schemes. The reaction rate and diffusivity are set to 0.5 s^{-1} and 0.05 m^2/s^{-1} respectively. The velocity streamlines, pressure and concentration profiles are shown in fig. 3.7.



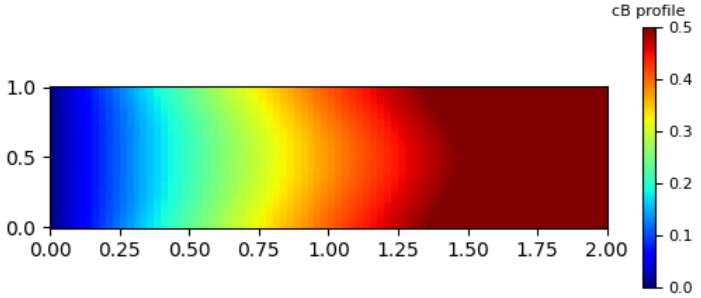
(a) Figure A



(b) Figure B



(c) Figure A



(d) Figure B

Figure 3.7: Velocity streamlines, pressure and concentration profiles in the plug-flow test case.

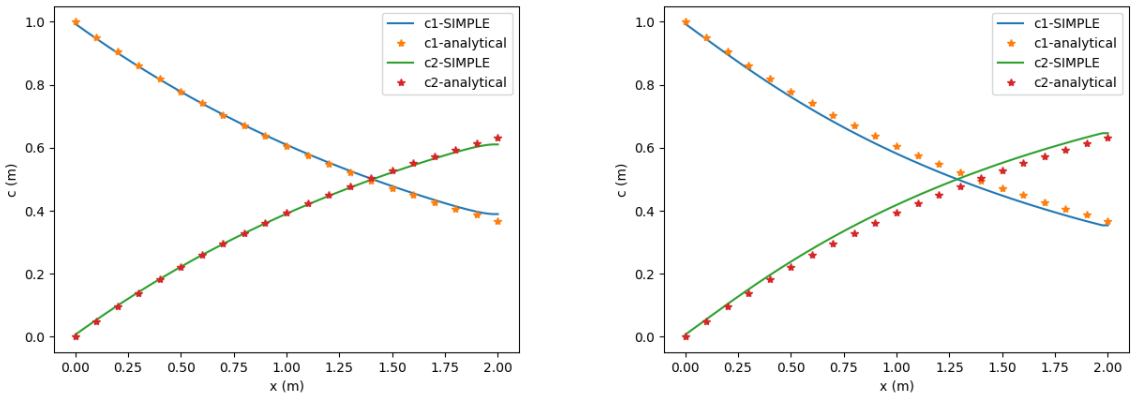
3.2.2 Analytical Solution

Using the 2D PFR model to find the analytical solution:

$$\frac{1}{\Delta y}\frac{\partial c_A}{\partial x} = -kc_A \rightarrow c_A = exp(-\Delta ykx) \tag{3.1}$$

$$\frac{1}{\Delta y}\frac{\partial c_B}{\partial x} = +kc_A \rightarrow c_B = 1 - exp(-\Delta ykx) \tag{3.2}$$

The analytical solution is compared to the surface averaged of the simulation results over the cross-section of the reactor. As it can be seen in fig. 3.8(a), there is a good agreement between the simulation results and analytical solution when the diffusivity is 0.05. However, when the diffusivity is a lower value fig. 3.8(b), the error starts to increase. The reason for this is that when the diffusivity is higher, there is better mixing across the cross-section of the reactor. The PFR model is derived by assuming that there is no concentration gradient across the cross-section which is only a good assumption when the diffusivity of the component is relatively high. Hence, the analytical solution from the PFR model is not valid for the case where the diffusivity is low.



(a) D=0.05

(b) D=0.001

Figure 3.8: Comparison of simulation results vs the analytical solution from the plug-flow reactor (PFR) model.

References

[1] H Scott Fogler. *Elements of chemical reaction engineering*. Pearson, 2020.

[2] Joel H Ferziger and Milovan PeriC. Computational methods for fluid dynamics, 2002.

[3] Martin Guay, Fabrice Colin, and Richard Egli. Simple and fast fluids. In *GPU PRO 360 Guide to GPGPU*, pages 47–58. AK Peters/CRC Press, 2018.

[4] UKNG Ghia, Kirti N Ghia, and CT Shin. High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method. *Journal of computational physics*, 48(3):387–411, 1982.

.1 main.py

```
import os.path
import time
import numpy as np
from solver import init, SIMPLE, calculate_cell_centre_values
from user_setting import NI,NJ,Ihalo,Jhalo,Re, maxit,relaxation_factor_u,
    relaxation_factor_v,convection_scheme,diffusion_order
import csv

# create u, v and p numpy arrays
u, v, p = init(NI, NJ, Ihalo, Jhalo)
c = np.ones((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
c2 = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
# start recording the time
start_time = time.process_time()
# Solve
c,c2, u, v, p, res = SIMPLE(c,c2, u, v, p, maxit, Re, relaxation_factor_u
    , relaxation_factor_v, convection_scheme, diffusion_order)
# Display the process time
print(f"process time in cpus:{time.process_time()-start_time}")

# find the values of u and v at the centre of p-control volumes
uc,vc = calculate_cell_centre_values(u, v)

# create solution directory
if os.path.isdir("./sol") == False:
    os.mkdir("./sol")
# Save the solutions in csv files
np.savetxt("./sol/u.csv", u, delimiter=",")
np.savetxt("./sol/v.csv", v, delimiter=",")
np.savetxt("./sol/uc.csv", uc, delimiter=",")
np.savetxt("./sol/vc.csv", vc, delimiter=",")
np.savetxt("./sol/p.csv", p, delimiter=",")
np.savetxt("./sol/c.csv", c, delimiter=",")
np.savetxt("./sol/c2.csv", c2, delimiter=",")
# record information about iterations and residuals
```

```
with open('./sol/alog.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(res)
```

.2 solver.py

```
import numpy as np
from numba import njit
from user_setting import NI, NJ, Ihalo, Jhalo, error_tolerance,
    Hybrid_switch, dx, dy, maxit_inner_u, maxit_inner_v, \
     maxit_inner_p, relaxation_factor_p, reaction_rate_constant,
    diffusivity


@njit
def init(NI, NJ, Ihalo, Jhalo):
    '''
    creates numpy array for u-velocity, v-velocity and p solution spaces.
    args:
    NI : number of mesh cells in x direction
    NJ : number of mesh cells in y direction
    Ihalo : number of halo cells on one side of x-axis
    Jhalo : number of halo cells on one side of y-axis

    returns:
    u : np.ndarray for u velocity
    v: np.ndarray for v velocity
    p: np.ndarray for p
    '''
    u = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
    v = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo  +1))
    p = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
    return u, v, p
@njit
def init_coef_arrays(NI, NJ, Ihalo, Jhalo):
    '''
    creates numpy arrays which includes the coefficients for each mesh
```

```
    cell.
    args:
    NI : number of mesh cells in x direction
    NJ : number of mesh cells in y direction
    Ihalo : number of halo cells on one side of x-axis
    Jhalo : number of halo cells on one side of y-axis

    returns:
    AE,AW,AN,AS,AEE,AWW,ANN,ASS,AP (all np.array)
    '''
    AE = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
    AW = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo  +1))
    AN = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
    AS = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
    AEE = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
    AWW = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo  +1))
    ANN = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
    ASS = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
    AP = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
    return AE,AW,AN,AS,AEE,AWW,ANN,ASS,AP
@njit
def grid(NI, NJ, Ihalo, Jhalo):
    '''
    creates grid which includes halo cells.
    NI : number of mesh cells in x direction
    NJ : number of mesh cells in y direction
    Ihalo : number of halo cells on one side of x-axis
    Jhalo : number of halo cells on one side of y-axis

    return: np.ndarray for coordinates of mesh cell edges
    x,y

    '''
    # uniform grid in x and y
    dx = 1.0 / NI
    dy = 1.0 / NJ

```

```
    # including halo data region
    x = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
    y = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))

    for i in range(-Ihalo, NI + Ihalo + 1):
        for j in range(-Jhalo, NJ + Jhalo + 1):
            x[i + Ihalo, j + Jhalo] = dx * i
            y[i + Ihalo, j + Jhalo] = dy * j

    return x, y
@njit
def grid_centre(NI,NJ):
    '''
    creates grid which does NOT include the halo cells.
    NI : number of mesh cells in x direction
    NJ : number of mesh cells in y direction

    return: np.ndarray for coordinates of mesh cell centres
    x,y
    '''
    # uniform grid in x and y
    dx = 1.0/NI
    dy = 1.0/NJ

    x = np.zeros((NI , NJ))
    y = np.zeros((NI , NJ))
    for i in range(NI):
        for j in range(NJ):
            x[i] = dx * i + dx/2
            y[j] = dy * j + dx/2
    return x, y
@njit
def get_convective_coefficients(ue, uw, vn, vs, convection_scheme,i,j):
    """
    evaluates the convection part of the neighbouring coefficients (A_nb)
    and A_P using
    different convection schemes
```

```
102
103     args:
104     ue: u velocity at the east face
105     uw: u velocity at the west face
106     vn: v velocity at the north face
107     vs: v velocity at the south face
108     i : reference index in x-axis position (used to activate UDS on
    boundaries for quick)
109     j : reference index in y-axis position (used to activate UDS on
    boundaries for quick)
110     convection_scheme: "UDS" or "QUICK""
111
112     return:
113     [AE,AW,AN,AS,AEE,AWW,ANN,ASS,AP]
114     """
115     if convection_scheme=="UDS":
116         AE = dy * max(0, -ue)
117         AW = dy * max(0, uw)
118         AN = dx * max(0, -vn)
119         AS = dx * max(0, vs)
120         AEE = 0
121         AWW = 0
122         ANN = 0
123         ASS = 0
124         AP = AS+AW+AN+AE+ASS+AWW+ANN+AEE
125     elif convection_scheme =="QUICK":
126         # at the boundaries use UDS
127         if j == Jhalo + 1 or j == NJ + Jhalo or i == NI + Ihalo - 1 or i
    == Ihalo:
128             [AE,AW,AN,AS,AEE,AWW,ANN,ASS,AP]= \
129                 get_convective_coefficients(ue, uw, vn, vs, "UDS", i, j)
130         else:
131             AE = -dy*(-0.75*max(0, -ue) + 0.375*max(0,ue) - 0.125*max(0,
    -uw))
132             AW = -dy*(-0.125*max(0, ue) + 0.375*max(0, -uw) - 0.75*max(0,
     uw))
133             AN = -dx * (-0.75 * max(0, -vn) + 0.375 * max(0, vn) - 0.125
```

```
    * max(0, -vs))
134             AS = -dx*(-0.125*max(0, vn) + 0.375*max(0, -vn) - 0.75*max(0,
     vs))
135             AEE = -0.125*dy*max(0, -ue)
136             AWW = -0.125*dy*max(0, uw)
137             ANN = -0.125*dx*max(0, -vn)
138             ASS = -0.125*dx*max(0, vs)
139             AP = AS+AW+AN+AE+ASS+AWW+ANN+AEE
140     else:
141         raise Exception("Select one of the following convection schemes:
    UDS or QUICK. "
142                         "For Hybrid, change Hybrid in user_setting.py to
    True ")
143
144     return np.array([AE,AW,AN,AS,AEE,AWW,ANN,ASS,AP])
145
146 @njit
147 def Hybrid(ue, uw, vn, vs,nu, convection_scheme,diffusion_order,i,j):
148     """
149     evaluates the net (convection part + diffusion part)  neighbouring
    coefficients (A_nb) and A_P using
150     hybrid scheme.
151
152     args:
153     ue: u velocity at the east face
154     uw: u velocity at the west face
155     vn: v velocity at the north face
156     vs: v velocity at the south face
157     i : reference index in x-axis position (used to activate UDS on
    boundaries for quick)
158     j : reference index in y-axis position (used to activate UDS on
    boundaries for quick)
159     convection_scheme: "UDS" or "QUICK""
160
161     return:
162     [AE,AW,AN,AS,AEE,AWW,ANN,ASS,AP]
163     """
```

```
    if convection_scheme != "UDS":
        raise Exception("Hybrid scheme is only implemented for UDS convection scheme")
    # use the second-order diffusion approximation
    [DE,DW,DN,DS,DEE,DWW,DNN,DSS,DP] = get_diffusive_coefficients(nu,2,i,j)
    AE = dy * max(0, -ue , DE/dy -ue / 2)
    AW = dy * max(0, uw, DW/dy + uw /2)
    AN = dx * max(0, -vn, DN/dx -vn / 2)
    AS = dx * max(0, vs, DS/dx +vs / 2)
    AEE = 0
    AWW = 0
    ANN = 0
    ASS = 0
    AP = AS+AW+AN+AE+ASS+AWW+ANN+AEE
    return np.array([AE, AW, AN, AS, AEE, AWW, ANN, ASS, AP])
@njit
def get_diffusive_coefficients(nu, diffusion_order,i,j):
    """
    evaluates the diffusion part of the neighbouring coefficients (A_nb).
    args:
    nu: kinematic viscosity
    diffusion_order: order of the scheme used for closing the diffusive terms

    return:
    [AE,AW,AN,AS,AEE,AWW,ANN,ASS]
    """
    # the coefficients for the second order approximation
    if diffusion_order == 2:
        AE = dy * nu / dx
        AW = dy * nu / dx
        AN = dx * nu / dy
        AS = dx * nu / dy
        AEE = 0
        AWW = 0
        ANN = 0
```

```
198         ASS = 0
199         AP =  - nu*(-2*dx/dy - 2*dy/dx)
200     # the coefficients for the fourth order approximation
201     elif diffusion_order == 4:
202         if j == Jhalo+1 or j == NJ + Jhalo or i == NI + Ihalo - 1 or i ==
    Ihalo:
203             [AE,AW,AN,AS,AEE,AWW,ANN,ASS,AP]= \
204                 get_diffusive_coefficients(nu, 2,i,j)
205         else:
206             AE = 4/3*dy*nu/dx
207             AW = 4/3*dy*nu/dx
208             AN = 4/3*dx*nu/dy
209             AS = 4/3*dx*nu/dy
210             AEE = -1/12*dy*nu/dx
211             AWW = -1/12*dy*nu/dx
212             ANN = -1/12*dx*nu/dy
213             ASS = -1/12*dx*nu/dy
214             AP = - nu*(-2.5*dx/dy - 2.5*dy/dx)
215     else:
216         raise Exception("the implemented diffusion schemes: 2nd order, 4
    th order")
217     return np.array([AE,AW,AN,AS,AEE,AWW,ANN,ASS,AP])
218 @njit
219 def calculate_cell_centre_values(u, v):
220     """
221     calculates the velocity values at the centre of p-control volume
222 
223     args:
224     u : numpy array for u velocity
225     v : numpy array for v velocity
226 
227     returns:
228     uc: numpy array for u-velocity values at the centre of p-control
    volume
229     vc: numpy array for v-velocity values at the centre of p-control
    volume
230     """
```

```
231     uc = np.zeros((NI , NJ ))
232     vc = np.zeros((NI , NJ))
233     for i in range(NI):
234         vc[i , :] = (v[i + Ihalo , Jhalo + 1:NJ+Jhalo+1] + v[i + Ihalo +
    1,Jhalo+1:NJ+Jhalo+1]) / 2
235     for j in range(NJ):
236         uc[: , j] = (u[Ihalo:Ihalo+NI,Jhalo+j]+u[Ihalo:Ihalo+NI,Jhalo+j
    +1])/2
237     return uc, vc
238
239 #@njit
240 def SIMPLE(c ,c2, u, v, p, maxit, Re, relaxation_factor_u,
    relaxation_factor_v, convection_scheme, diffusion_order):
241     nu = 1 / Re
242     res = []
243     for iter in range(1, maxit + 1):
244
245         # solve u-equation
246         u, BU, res_u, APU = calculate_u(u, v, p, nu, NI, NJ, Ihalo, Jhalo
    , relaxation_factor_u, convection_scheme, diffusion_order)
247
248         # solve v-equation
249         v, BV, res_v, APV = calculate_v(u, v, p, nu, NI, NJ, Ihalo, Jhalo
    , relaxation_factor_v, convection_scheme, diffusion_order)
250
251         # solve p'-equation
252         p, BP, res_p = calculate_p(u, v, p, NI, NJ, Ihalo, Jhalo,
    relaxation_factor_u,APU,APV)
253
254         c, BC, res_c = calculate_c(c, u, v, p, nu, NI, NJ, Ihalo, Jhalo,
    relaxation_factor_u, convection_scheme, diffusion_order)
255
256         c2, BC2, res_c2 = calculate_c2(c2,c, u, v, p, nu, NI, NJ, Ihalo,
    Jhalo, relaxation_factor_u, convection_scheme, diffusion_order)
257
258         res.append([iter, res_u, res_v, res_p, res_c,res_c2])
259         if iter % 10 == 0:
```

```
260             print(f"{iter} {res_u} {res_v} {res_p} {res_c} {res_c2}")
261 
262         # define convergence criterion
263         if max(res_u, res_v, res_p, res_c,res_c2) <= error_tolerance and
    iter > 1:
264             print("converged")
265             return c,c2, u, v, p, res
266         elif iter == maxit - 1:
267             print("did not converge")
268             return c,c2, u, v, p, res
269 
270 @njit
271 def calculate_u(u, v, p, nu, NI, NJ, Ihalo, Jhalo, relaxation_factor_u,
    convection_scheme, diffusion_order):
272     AEU, AWU, ANU, ASU, AEEU, AWWU, ANNU, ASSU, APU = init_coef_arrays(NI
    ,NJ,Ihalo,Jhalo)
273     BU = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
274     # # Swift direction: south to north
275     for i in range(NI + Ihalo - 1, Ihalo-1 , -1):
276         # Swift direction: west to east (does not include the most
    eastern cell)
277         for j in range(Jhalo , NJ + Jhalo+1):
278             # define coefficients for u-equation
279             # linear interpolation to find variables on the CV boundary
280             U_1_ue = (u[i, j] + u[i, j + 1]) / 2
281             U_1_uw = (u[i, j] + u[i, j - 1]) / 2
282             U_1_vn = (v[i, j] + v[i, j + 1]) / 2
283             U_1_vs = (v[i + 1, j] + v[i + 1, j + 1]) / 2
284             if Hybrid_switch == False:
285                 # calculate the coefficients for u_i,j
286                 [AEU[i,j], AWU[i,j], ANU[i,j], ASU[i,j], AEEU[i,j], AWWU[
    i,j], ANNU[i,j], ASSU[i,j], APU[i,j]] = \
287                     get_convective_coefficients(U_1_ue, U_1_uw, U_1_vn,
    U_1_vs, convection_scheme, i, j) \
288                     + get_diffusive_coefficients(nu, diffusion_order,i,j)
289             # if Hybrid_switch == True, then do hybrid scheme
290             else:
```

```
291                 [AEU[i, j], AWU[i, j], ANU[i, j], ASU[i, j], AEEU[i, j],
    AWWU[i, j], ANNU[i, j], ASSU[i, j], APU[i, j]]\
292                     = Hybrid(U_1_ue, U_1_uw, U_1_vn, U_1_vs, nu,
    convection_scheme, diffusion_order,i,j)
293
294     # Apply BCs to the south and north boundaries
295     # south
296     u[-Ihalo + 1, Jhalo + 1:NJ + Jhalo + 1] = - u[-Ihalo, Jhalo + 1:NJ +
    Jhalo + 1]
297     # north
298     u[Ihalo - 1, Jhalo + 1:NJ + Jhalo + 1] =  - u[Ihalo, Jhalo + 1:NJ +
    Jhalo + 1]
299     # west
300     u[Ihalo:Ihalo + NI, Jhalo] = 2 -u[Ihalo:Ihalo + NI, Jhalo + 1]
301     # east
302     u[Ihalo:NI + Ihalo, NJ + Jhalo] = +u[Ihalo:NI + Ihalo, NJ + Jhalo-1]
303
304     # calculate residuals
305     res_u = 0.0
306     for i in range(NI + Ihalo - 1, Ihalo - 1, -1):
307         for j in range(Jhalo + 1, NJ + Jhalo):
308             BU[i,j] = (AEU[i,j] * u[i, j + 1] + AWU[i,j] * u[i, j - 1] +
    ANU[i,j] * u[i - 1, j] + ASU[i,j] * u[i + 1, j]) \
309                     + (AEEU[i,j] * u[i, j + 2] + AWWU[i,j] * u[i, j - 2] +
     ANNU[i,j] * u[i - 2, j] + ASSU[i,j] * u[i + 2, j]) \
310                     + (p[i, j] - p[i, j + 1]) * dy - APU[i,j] * u[i, j]
311             res_u += abs(BU[i,j])
312
313     # inner iterations for the u equation
314     for inner_iter in range(1, maxit_inner_u + 1):
315         for i in range(NI + Ihalo - 1, Ihalo - 1, -1):
316             for j in range(Jhalo + 1, NJ + Jhalo):
317                 u[i,j] =  relaxation_factor_u * ((AEU[i,j] * u[i, j + 1]
    + AWU[i,j] * u[i, j - 1] + ANU[i,j] * u[i - 1, j] + ASU[i,j] * u[i +
    1, j]) \
318                     + (AEEU[i,j] * u[i, j + 2] + AWWU[i,j] * u[i, j - 2] +
     ANNU[i,j] * u[i - 2, j] + ASSU[i,j] * u[i + 2, j]) \
```

```
                    + (p[i, j] - p[i, j + 1]) * dy )/APU[i,j] + (1 -
    relaxation_factor_u) * u[i, j]


     return u, BU, res_u, APU


@njit
def calculate_v(u, v, p, nu, NI, NJ, Ihalo, Jhalo, relaxation_factor_u,
    convection_scheme, diffusion_order):
     AEV, AWV, ANV, ASV, AEEV, AWWV, ANNV, ASSV, APV = init_coef_arrays(NI
    ,NJ,Ihalo,Jhalo)
     BV = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
     # # Swift direction: south to north
     for i in range(NI + Ihalo+1, Ihalo-1, -1):
         # Swift direction: west to east (does not include the most
    eastern cell)
         for j in range(Jhalo , NJ + Jhalo + 2):
             # define coefficients for v-equation
             # linear interpolation to find variables on the CV boundary
             V_1_ue = (u[i - 1, j] + u[i, j]) / 2
             V_1_uw = (u[i, j - 1] + u[i - 1, j - 1]) / 2
             V_1_vn = (v[i, j] + v[i - 1, j]) / 2
             V_1_vs = (v[i, j] + v[i + 1, j]) / 2

             # calculate the coefficients for v_i,j
             if Hybrid_switch == False:
                 [AEV[i,j], AWV[i,j], ANV[i,j], ASV[i,j], AEEV[i,j], AWWV[
    i,j], ANNV[i,j], ASSV[i,j], APV[i,j]] = \
                     get_convective_coefficients(V_1_ue, V_1_uw, V_1_vn,
    V_1_vs, convection_scheme, i, j) \
                     + get_diffusive_coefficients(nu, diffusion_order,i,j)
             else:
                 [AEV[i,j], AWV[i,j], ANV[i,j], ASV[i,j], AEEV[i,j], AWWV[
    i,j], ANNV[i,j], ASSV[i,j], APV[i,j]]= \
                     Hybrid(V_1_ue, V_1_uw, V_1_vn, V_1_vs, nu,
    convection_scheme, diffusion_order,i,j)
```

```
348
349
350     # Apply the boundary conditions to the west and east faces
351     # west
352     v[Ihalo:Ihalo + NI, Jhalo] = -v[Ihalo:Ihalo + NI, Jhalo + 1]
353     # east
354     v[Ihalo:NI + Ihalo, NJ + Jhalo+2] = +v[Ihalo:NI + Ihalo, NJ + Jhalo
    +1]
355
356     # calculate residuals
357     res_v = 0.0
358     for i in range(NI + Ihalo - 1, Ihalo, -1):
359         for j in range(Jhalo + 1, NJ + Jhalo + 1):
360             BV[i,j] = (AEV[i,j] * v[i, j + 1] + AWV[i,j] * v[i, j - 1] +
    ANV[i,j] * v[i - 1, j] + ASV[i,j] * v[i + 1, j]) \
361                     + (AEEV[i,j] * v[i, j + 2] + AWWV[i,j] * v[i, j - 2] +
     ANNV[i,j] * v[i - 2, j] + ASSV[i,j] * v[i + 2, j]) \
362                     + (p[i, j] - p[i - 1, j]) * dx - APV[i,j] * v[i, j]
363             res_v += abs(BV[i,j])
364
365     # inner iterations for the v equation
366     for inner_iter in range(1, maxit_inner_v + 1):
367         for i in range(NI + Ihalo - 1, Ihalo, -1):
368             for j in range(Jhalo + 1, NJ + Jhalo+1):
369                 v[i,j] =  relaxation_factor_u * ((AEV[i,j] * v[i, j + 1]
    + AWV[i,j] * v[i, j - 1] + ANV[i,j] * v[i - 1, j] + ASV[i,j] * v[i +
    1, j]) \
370                     + (AEEV[i,j] * v[i, j + 2] + AWWV[i,j] * v[i, j - 2] +
     ANNV[i,j] * v[i - 2, j] + ASSV[i,j] * v[i + 2, j]) \
371                     + (p[i, j] - p[i-1, j]) * dy )/APV[i,j] + (1 -
    relaxation_factor_u) * v[i, j]
372
373     return v, BV, res_v, APV
374
375 @njit
376 def calculate_p(u, v, p, NI, NJ, Ihalo, Jhalo, relaxation_factor_u,APU,
    APV):
```

```
377
378     AEP, AWP, ANP, ASP, AEEP, AWWP, ANNP, ASSP, APP = init_coef_arrays(NI
    , NJ, Ihalo, Jhalo)
379     BP = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
380     p_correct = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
381     # # Swift direction: south to north
382     for i in range(NI + Ihalo - 1, Ihalo - 1, -1):
383         for j in range(Jhalo + 1, NJ + Jhalo + 1):
384             AEP[i,j] = dy ** 2 / APU[i,j]
385             AWP[i,j] = dy ** 2 / APU[i,j-1]
386             ANP[i,j] = dx ** 2 / APV[i,j]
387             ASP[i,j] = dx ** 2 / APV[i+1 , j]
388
389     # west boundary condition
390     if j == Jhalo + 1:
391         AWP[:,j] = 0
392     # east boundary condition
393     if j == NJ + Jhalo:
394         AEP[:,j] = 0
395     # South boundary condition
396     if i == NI + Ihalo - 1:
397         ASP[i,:] = 0
398     # north boundary condition
399     if i == Ihalo:
400         ANP[i,:] = 0
401
402     # calculate residuals
403     res_p = 0.0
404     for i in range(NI + Ihalo - 1, Ihalo - 1, -1):
405         for j in range(Jhalo + 1, NJ + Jhalo + 1):
406             APP[i,j] = AEP[i,j] + AWP[i,j] + ANP[i,j] + ASP[i,j]
407             BP[i,j] = -dx*(-v[i + 1, j] + v[i, j]) - dy*(-u[i , j-1] + u[
    i, j])
408             res_p += abs(BP[i,j])
409
410     # inner iterations for p'-equation
411     for N in range(1, maxit_inner_p + 1):
```

```
412         for i in range(NI + Ihalo - 1, Ihalo - 1, -1):
413             for j in range(Jhalo + 1, NJ + Jhalo + 1):
414                 p_correct[i,j] = BP[i,j]/APP[i,j]
415                 p_correct[i, j] += (AEP[i,j] * p_correct[i,j+1] + AWP[i,j
    ] * p_correct[i,j-1] + ASP[i,j] * p_correct[i+1,j] + ANP[i,j] *
    p_correct[i-1,j])/APP[i,j]
416     # correct u-velocity
417     for i in range(NI + Ihalo - 1, Ihalo - 1, -1):
418         for j in range(Jhalo + 1, NJ + Jhalo):
419             u_correct = dy* (p_correct[i,j] - p_correct[i,j+1])/APU[i,j]
420             u[i,j] += relaxation_factor_u * u_correct
421             pass
422 
423     # correct v-velocity
424     for i in range(NI + Ihalo - 1, Ihalo, -1):
425         for j in range(Jhalo + 1, NJ + Jhalo + 1):
426             v_correct = dy * (p_correct[i, j] - p_correct[i - 1, j]) /
    APV[i, j]
427             v[i, j] += relaxation_factor_u * v_correct
428 
429     # correct p (the pressure)
430     for i in range(NI + Ihalo - 1, Ihalo - 1, -1):
431         for j in range(Jhalo + 1, NJ + Jhalo + 1):
432             p[i,j] += relaxation_factor_p * p_correct[i,j]
433 
434     return p, BP, res_p
435 
436 @njit
437 def calculate_c(c, u, v, p, nu, NI, NJ, Ihalo, Jhalo, relaxation_factor_u
    , convection_scheme, diffusion_order):
438     AEC, AWC, ANC, ASC, AEEC, AWWC, ANNC, ASSC, APC = init_coef_arrays(NI
    , NJ, Ihalo, Jhalo)
439     BC = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
440     # # Swift direction: south to north
441     for i in range(NI + Ihalo - 1, Ihalo - 1, -1):
442         for j in range(Jhalo + 1, NJ + Jhalo + 1):
443             # define coefficients for u-equation
```

```
444             # linear interpolation to find variables on the CV boundary
445             U_1_ue = u[i,j]
446             U_1_uw = u[i,j-1]
447             U_1_vn = v[i,j]
448             U_1_vs = v[i+1,j]
449             if Hybrid_switch == False:
450                 # calculate the coefficients for u_i,j
451                 [AEC[i, j], AWC[i, j], ANC[i, j], ASC[i, j], AEEC[i, j],
    AWWC[i, j], ANNC[i, j], ASSC[i, j],
452                  APC[i, j]] = \
453                     get_convective_coefficients(U_1_ue, U_1_uw, U_1_vn,
    U_1_vs, convection_scheme, i, j) \
454                     + get_diffusive_coefficients(diffusivity,
    diffusion_order,i,j)
455             # if Hybrid_switch == True, then do hybrid scheme
456             else:
457                 [AEC[i, j], AWC[i, j], ANC[i, j], ASC[i, j], AEEC[i, j],
    AWWC[i, j], ANNC[i, j], ASSC[i, j], APC[i, j]] \
458                     = Hybrid(U_1_ue, U_1_uw, U_1_vn, U_1_vs, diffusivity,
     convection_scheme, diffusion_order,i,j)
459
460     # Apply BCs
461     # south
462     c[-Ihalo-1, Jhalo + 1:NJ + Jhalo + 1] = + c[-Ihalo - 2, Jhalo + 1:NJ
    + Jhalo + 1]
463     # north
464     c[Ihalo - 1, Jhalo + 1:NJ + Jhalo + 1] =  + c[Ihalo, Jhalo + 1:NJ +
    Jhalo + 1]
465     # west
466     c[Ihalo:Ihalo + NI, Jhalo] = 2 - c[Ihalo:Ihalo + NI, Jhalo + 1]
467     # east
468     c[Ihalo:NI + Ihalo, NJ + Jhalo + 1] =  +c[Ihalo:NI + Ihalo, NJ +
    Jhalo]
469
470
471     # calculate residuals
472     res_c = 0.0
```

```
    for i in range(NI + Ihalo - 1, Ihalo - 1, -1):
        for j in range(Jhalo + 1, NJ + Jhalo + 1):
            S = - c[i, j] * reaction_rate_constant * dx * dy
            BC[i, j] = (AEC[i, j] * c[i, j + 1] + AWC[i, j] * c[i, j - 1] + ANC[i, j] * c[i - 1, j] + ASC[i, j] * c[
                i + 1, j]) \
                       + (AEEC[i, j] * c[i, j + 2] + AWWC[i, j] * c[i, j - 2] + ANNC[i, j] * c[i - 2, j] + ASSC[i, j] *
                          c[i + 2, j]) \
                        - (APC[i, j]) * c[i, j] + S
            res_c += abs(BC[i, j])

    # inner iterations for the u equation
    for inner_iter in range(1, maxit_inner_u + 1):
        for i in range(NI + Ihalo - 1, Ihalo - 1, -1):
            for j in range(Jhalo + 1, NJ + Jhalo + 1):
                S = - c[i, j] * reaction_rate_constant * dx * dy
                c[i, j] = relaxation_factor_u * ((AEC[i, j] * c[i, j + 1] + AWC[i, j] * c[i, j - 1] + ANC[i, j] * c[
                    i - 1, j] + ASC[i, j] * c[i + 1, j]) \
                                                 + (AEEC[i, j] * c[i, j + 2] + AWWC[i, j] * c[i, j - 2] + ANNC[i, j] *
                                                    c[i - 2, j] + ASSC[i, j] * c[i + 2, j]) + S) / (APC[i, j]) + (
                                       1 - relaxation_factor_u) * c[i, j]

    return c, BC, res_c


@njit
def calculate_c2(c,c_other, u, v, p, nu, NI, NJ, Ihalo, Jhalo, relaxation_factor_u, convection_scheme, diffusion_order):
    AEC, AWC, ANC, ASC, AEEC, AWWC, ANNC, ASSC, APC = init_coef_arrays(NI, NJ, Ihalo, Jhalo)
    BC = np.zeros((NI + 2 * Ihalo + 1, NJ + 2 * Jhalo + 1))
    # # Swift direction: south to north
    for i in range(NI + Ihalo - 1, Ihalo - 1, -1):
```

```
503         for j in range(Jhalo + 1, NJ + Jhalo + 1):
504             # define coefficients for u-equation
505             # linear interpolation to find variables on the CV boundary
506             U_1_ue = u[i,j]
507             U_1_uw = u[i,j-1]
508             U_1_vn = v[i,j]
509             U_1_vs = v[i+1,j]
510             if Hybrid_switch == False:
511                 # calculate the coefficients for u_i,j
512                 [AEC[i, j], AWC[i, j], ANC[i, j], ASC[i, j], AEEC[i, j],
    AWWC[i, j], ANNC[i, j], ASSC[i, j],
513                  APC[i, j]] = \
514                     get_convective_coefficients(U_1_ue, U_1_uw, U_1_vn,
    U_1_vs, convection_scheme, i, j) \
515                     + get_diffusive_coefficients(diffusivity,
    diffusion_order,i,j)
516             # if Hybrid_switch == True, then do hybrid scheme
517             else:
518                 [AEC[i, j], AWC[i, j], ANC[i, j], ASC[i, j], AEEC[i, j],
    AWWC[i, j], ANNC[i, j], ASSC[i, j], APC[i, j]] \
519                     = Hybrid(U_1_ue, U_1_uw, U_1_vn, U_1_vs, diffusivity,
     convection_scheme, diffusion_order,i,j)
520
521     # Apply BCs
522     # south
523     c[-Ihalo-1, Jhalo + 1:NJ + Jhalo + 1] = + c[-Ihalo - 2, Jhalo + 1:NJ
    + Jhalo + 1]
524     # north
525     c[Ihalo - 1, Jhalo + 1:NJ + Jhalo + 1] = + c[Ihalo, Jhalo + 1:NJ +
    Jhalo + 1]
526     # west
527     c[Ihalo:Ihalo + NI, Jhalo] = - c[Ihalo:Ihalo + NI, Jhalo + 1]
528     # east
529     c[Ihalo:NI + Ihalo, NJ + Jhalo + 1] =  + c[Ihalo:NI + Ihalo, NJ +
    Jhalo]
530
531
```

```
    # calculate residuals
    res_c = 0.0
    for i in range(NI + Ihalo - 1, Ihalo - 1, -1):
        for j in range(Jhalo + 1, NJ + Jhalo + 1):
            S = + c_other[i, j] * reaction_rate_constant * dx * dy
            BC[i, j] = (AEC[i, j] * c[i, j + 1] + AWC[i, j] * c[i, j - 1]
    + ANC[i, j] * c[i - 1, j] + ASC[i, j] * c[
                i + 1, j]) \
                        + (AEEC[i, j] * c[i, j + 2] + AWWC[i, j] * c[i, j
   - 2] + ANNC[i, j] * c[i - 2, j] + ASSC[i, j] *
                           c[i + 2, j]) \
                         - (APC[i, j]) * c[i, j] + S
            res_c += abs(BC[i, j])

    # inner iterations for the u equation
    for inner_iter in range(1, maxit_inner_u + 1):
        for i in range(NI + Ihalo - 1, Ihalo - 1, -1):
            for j in range(Jhalo + 1, NJ + Jhalo + 1):
                S = + c_other[i, j] * reaction_rate_constant * dx * dy
                c[i, j] = relaxation_factor_u * ((AEC[i, j] * c[i, j + 1]
    + AWC[i, j] * c[i, j - 1] + ANC[i, j] * c[
                    i - 1, j] + ASC[i, j] * c[i + 1, j]) \
                                                + (AEEC[i, j] * c[i, j +
    2] + AWWC[i, j] * c[i, j - 2] + ANNC[i, j] *
                                                    c[i - 2, j] + ASSC[i,
    j] * c[i + 2, j])+S) / (APC[i, j]) + (
                                        1 - relaxation_factor_u) * c[i, j]

    return c, BC, res_c
```

.3 user_setting.py

```
# Mesh dimensions
NI = 40
NJ = 80
# Grid spacing (uniform mesh)
dx = 1/NI
```

```
dy = 1/NJ
# Reynolds number
Re = 400

#
maxit_inner_u = 4
maxit_inner_v = 4
maxit_inner_p = 4

# error tolerance
error_tolerance = 0.0001
# max number of iterations.
maxit = 3000
# specify number of "halo-data" layers
Ihalo = 2
Jhalo = 2
# under-relaxation factor for u and v equations
relaxation_factor_u = 0.5
relaxation_factor_v = relaxation_factor_u
relaxation_factor_p = 0.3
# choose a convection scheme
convection_scheme = "UDS"
# choose the order of approximation for the diffusive term.
diffusion_order = 2
Hybrid_switch = False

reaction_rate_constant = 0.5
diffusivity = 0.001
```