Numerical Solution for 2D Steady Incompressible Navier Stokes Equations
MEEN 644
Finte Volume Techniques for Heat transfer and Fluid flow

Srikanth Sampathkumar

May 6, 2025

Abstract

A numerical solver for 2D steady incompressible flow over a backward facing step is presented. The pressure equation is solved using an alternate direction implicit (ADI) scheme and pressure-velocity coupling is achieved through fractional step method. The accuracy of the solution procedure is checked by observing the recirculation zones and the reattachment length for different Reynolds numbers. The values are compared with the data present in the literature. The results show that the proposed solution gives accurate results at low Reynolds numbers but under predicts reattachment length for high Reynolds number flows.

1 Introduction

The Navier-Stokes (NS) equations govern the fluid flow behaviour in the continuum regime. Due to their non-linear and non-local nature, closed form solution doesn't exist except for some special cases like Blassius flat plate boundary layer and Burgers' equation. Hence, NS equations are solved numerically. A numerical approach for solving steady incompressible 2D NS equations is presented here by applying it to the classical problem of flow through a backward facing step. This particular problem is chosen because of the extensive availability of experimental data [1, 2] and the rich features present in the flow.

Armaly et al[1] identified three distinct flow regimes based on Reynolds number (Re): laminar (Re $<$ 400), transitional (400 $<$ Re $<$ 6600), and turbulent (Re $>$ 6600). Using laser-Doppler anemometry, the authors revealed multiple recirculation zones-including primary, secondary (upper wall), and tertiary (lower wall) separation regions-previously unreported in literature. The authors also identified the critical role of expansion ratio and inlet velocity profiles in determining the length of the recircualtion zone.

Kim et al[2] applied fractional step method similar to that of Chorin [3] to solve the incompressible NS equations in three dimensions on a staggered grid. The authors showed that inconsistent boundary conditions for velocity field can lead to conservation issues and consequently lead to erroneous results. The results show good agreement with experimental results of recirculation zone length upto a Reynolds number of 500 beyond which the accuracy dropped. The authors attribute it to the third dimensional instability similar to Armaly [1].

Taking advantage of the works of Armaly et al.[1], Kim and Moin [2], this study aims to validate a newly developed incompressible Navier-Stokes solver by simulating 2D laminar flow over a backward-facing step for different Reynolds numbers from Re =100 to Re=400 in steps of 100.

2 Numerical Method

The steady incompressible NS equations are elliptic partial differential equations (PDEs) which comprise of, in 2D, two momentum equations and a continuity equation. They are given as follows:
Continuity equation (Non-dimensional form):

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

Momentum equations (Non-dimensional form):

$$u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}=-\frac{\partial p}{\partial x}+\frac{1}{Re}\frac{\partial^2 u}{\partial x^2} \tag{2}$$

$$u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}=-\frac{\partial p}{\partial y}+\frac{1}{Re}\frac{\partial^2 v}{\partial x^2} \tag{3}$$

where u, v are the non-dimensional x and y components of velocity, p is the non-dimensional kinematic pressure ($p=\frac{pressure}{density}$) and Re is the Reynolds number which is given by

$$Re=\frac{Uh}{\nu} \tag{4}$$

where U is the bulk velocity of the incoming fluid (m/s), h is the step height(m) and ν is the kinematic viscosity of the incoming fluid (m^2/s). All the components of the equations are non-dimensionalized to generalize the results. The quantities are non-dimensionalized as follows:

$$u=\frac{u_{dim}}{U}, v=\frac{v_{dim}}{U}, p=\frac{p_{dim}}{U^2}, x=\frac{x_{dim}}{h}, y=\frac{y_{dim}}{h} \tag{5}$$

where the quantities with a subscript "*dim*" denote their respective dimensional counterparts.

The incompressible NS equations have a decoupled continuity and momentum equations which means that the momentum equation could be advanced without the need of the continuity equations. As a consequence, the resulting velocity field might not necessarily be divergence free. The coupling of the momentum and continuity equations is achieved through explicit methods like Marker and Cell (MAC) method [4] and fractional-step method [3] and semi-implicit methods like SIMPLE [5] etc. Here, the explicit fractional step method is used due to its relatively simple implementation. The common feature among all of the coupling schemes is that they require solving a pressure poisson equation which enforces the continuity. In simple terms, the flow is conserved by pressure under incompressible conditions. The pressure poisson equation (PPE) is given by

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = f \tag{6}$$

where f is any function.

3 Problem Description

Flow through a backward facing step is studied here to test the accuracy of the numerical formulation due to the presence of features like recirculation zones which are very sensitive to the Reynolds number of the flow [2].The usual setup involves a straight channel which suddenly expands into a wider channel. This sudden expansion causes recirculation. In this work, the initial straight channel is removed and a fully developed velocity profile is provided at the inlet. The setup is shown in Fig 1. The fully developed velocity profile is given by

$$\frac{u_{inlet}}{U} = \frac{3}{2} - 6(y - \frac{3}{2})^2 \tag{7}$$

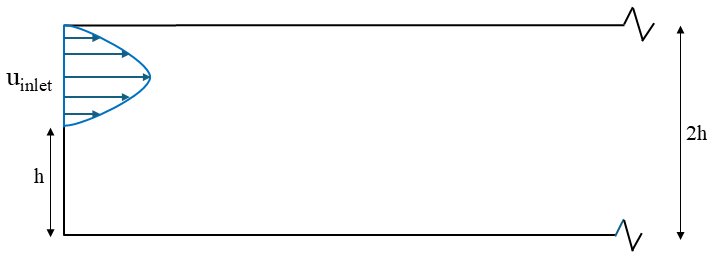

Figure 1:
Backward Facing step

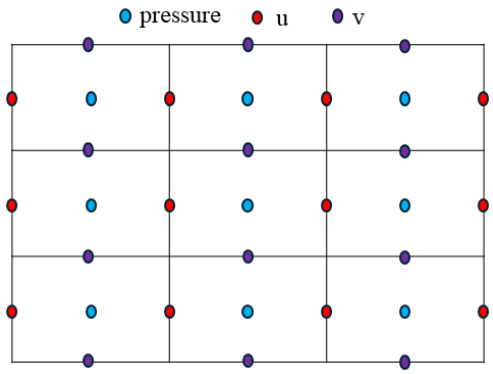

Figure 2:
Staggered grid

3.1 Discretization

The momentum equation is discretized using finite volume approach where the values at the faces are calculated using midpoint rule. To avoid odd-even decoupling in the pressure field, the staggered grid arrangement is used. In a staggered grid, the pressure is stored at the cell centres and the velocities are stored at the face centres. This arrangement allows for a compact stencil and eliminates checkerboard problem. The arrangement is shown in Fig. 2.

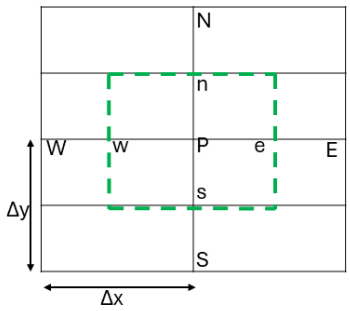

Figure 3: Control volume

Since velocity and pressure are stored at different locations, they require different control volumes. The generalized discretization for the momentum equations is given as follows:

$$R(u) = \frac{u_w u_w + u_e u_e}{\Delta x} + \frac{u_s v_s - u_n v_n}{\Delta y} + \frac{1}{Re}\left\{ \frac{\frac{\partial u}{\partial x}|_e - \frac{\partial u}{\partial x}|_w}{\Delta x} + \frac{\frac{\partial u}{\partial y}|_n - \frac{\partial u}{\partial y}|_s}{\Delta y} \right\} \tag{8}$$

$$R(v) = \frac{u_w v_w - u_e v_e}{\Delta x} + \frac{v_s v_s - v_n v_n}{\Delta y} + \frac{1}{Re}\left\{ \frac{\frac{\partial v}{\partial x}|_e - \frac{\partial v}{\partial x}|_w}{\Delta x} + \frac{\frac{\partial v}{\partial y}|_n - \frac{\partial v}{\partial y}|_s}{\Delta y} \right\} \tag{9}$$

where $R(u)$ and $R(v)$ are the discretized form of Eqns. 2 and 3 respectively without the pressure term. Δx and Δy are the grid size in the x and y directions respectively. The subscripts denote the location of the variable with respect to the location of interest P. The definition of the locations are shown in Fig. 3. The second order central scheme is used for discretization for all quantities except for the convective fluxes. The convective fluxes are discretized using upwind schemes and the direction of upwinding is obatined using a switching function based on the direction of flow at the point of interest. An example of the switching function for the x-momentum equation is shown here:

$$u_w u_w = \frac{|u_w| + u_w}{2} * u_W - \frac{|u_w| - u_w}{2} * u_P \tag{10}$$

When the velocity at w is positive, Eqn. 10 assigns the velocity at W to w whereas if the velocity is negative, it assign the velocity at P to w. The central scheme assigns equal weights to the right and left nodes when calculating the velocity at centre node which is not physical. This is because a quantity can be convected only is one direction at any point of time. Also, numerically, the central scheme doesn't allow for dissipation which results in the convective term becoming unstable.

When estimating the diffusive fluxes, the boundary nodes require special dicretization since there will not be a node on the other side of the boundary. For the boundary nodes, second order biased differencing is used. An example of this is shown here for bottom node:

$$\frac{9u_P - u_N}{3\Delta y} = \left.\frac{\partial u}{\partial y}\right|_s - O(\Delta y^2) \tag{11}$$

With the momentum equations complete, the attention falls onto the pressure poisson equation. When discretized, the PPE (ref Eqn 6) becomes

$$p_E + p_W - 2(1+\beta^2)p_P + \beta^2 p_N + \beta^2 p_S = \Delta x^2 * (f), \quad \beta = \frac{\Delta x}{\Delta y} \tag{12}$$

Equation 12 results in a penta-diagonal matrix which is computationally expensive to solve directly. Hence, Alternate Direction Implicit (ADI) is used to solve this penta-diagonal system. It involves a two step process with the solution space updated implicitly one direction in each step. The implementation is provided in Appendix B

3.2 Boundary Conditions

Since the steady incompressible NS equations form a system of elliptic PDEs, they require boundary conditions. The staggered grid arrangement eliminates the need for pressure boundary condition i.e. the pressure at the boundaries need not be required to advance the momentum equation. The boundary conditions used for this arrangement are as follows:

$$Left\ boundary:\ v(0,y)=0,\ u(0,y)=\begin{cases}u_{inlet}/U, & \text{for } y\geq 1 (see\ Eqn\ 7)\\ 0, & elsewhere\end{cases} \tag{13}$$

$$Top\ and\ bottom\ boundaries:\ u(x,0)=v(x,0)=u(x,2)=v(x,2)=0 \quad (No-slip) \tag{14}$$

$$Right\ boundary:\ \frac{\partial u}{\partial x}=\frac{\partial v}{\partial x}=0 \quad (Fully\ developed\ flow) \tag{15}$$

For solving the pressure poisson equation (Eqn. 12) at the boundary cells, the right hand side (RHS) has to be adjusted due to the requirement of pressure at a node outside the boundary. This can be achieved through having ghost nodes or directly accounting for those nodes using the boundary node itself. In this work, the boundary treatment is done using a zero gradient condition for pressure in the direction normal to the boundaries.

$$\text{At all boundaries,} \quad \frac{\partial p}{\partial n}=0 \tag{16}$$

where n denotes the normal direction. The use of Neumann boundary condition for pressure leads to infinite number of possible solution as long as the compatibility condition is satisfied.

3.3 Pressure-Velocity Coupling

The fractional step method [3] is used for coupling the momentum and continuity equations. This method involves calculating an intermediate velocity field by advancing the momentum equation without considering the pressure term and then correcting the velocity field using the pressure term obtained from the PPE. The steps are described as follows:

1. Solve for intermediate velocity:
$$\mathbf{u}^{*}=\mathbf{u}^{n}+\Delta t*(R(\mathbf{u}^{n}))$$
2. Solve pressure poisson equation:
$$div(grad(p^{n}))=\frac{1}{\Delta t}div(\mathbf{u}^{*})$$
3. Correct velocity:
$$\mathbf{u}^{n+1}=\mathbf{u}^{*}-\Delta t*grad(p^{n})$$

where R is shown in Eqns (2) & (3). div and $grad$ are the numerical divergence and gradient operators.Here, Δt is a pseudo-time step used for advancing the solution to steady state and n is the iteration number. The PPE in Step 2 is obtained by assuming divergence free nature of the velocity field. Hence, for the PPE to converge, the velocity field at $(n+1)^{th}$ level should be divergence free.

3.4 Convergence and Stability of the numerical schemes

The iterative nature of the numerical solution demands a convergence criterion which control when to stop the iterations. There are two loops in the solution method: the outer momentum loop and the inner pressure loop. The error associated with the pressure solution is computed using L2 norm given by:

$$\frac{\sum_{i=1}^{IM-2}\sum_{j=1}^{JM-2}|\left(p_{i,j}^{k+1}-p_{i,j}^{k}\right)|}{\sum_{i=1}^{IM-2}\sum_{j=1}^{JM-2}|\left(p_{i,j}^{k}\right)|}<\varepsilon \tag{17}$$

where ε is a small number. $(IM-2)$ and $(JM-2)$ are the total number of internal nodes in the x and y directions respectively. In this report, ε is assumed to be 10^{-3}. For the momentum equation, the solution is assumed to be converged if

$$max\left\{max(u^{n+1}-u^{n}),max(v^{n+1}-v^{n})\right\}<\delta \tag{18}$$

where δ is a small number. The value of δ is set to $1e-4$ by observing the effect of δ on the flow field.

The stability of an explicit solver for NS equations depend on the maximum Courant number of the flow which is given by:

$$Co_{max}=\frac{u_{max}*\Delta t}{min(\Delta x,\Delta y)} \tag{19}$$

The Courant number controls the extent of influence of changes in one cell on the neighboring cells. Courant number of 1 implies that any changes that occur in one cell affects only the first neighbors. In this work, the stability of the numerical scheme is ensured by restricting the maximum Courant number to 0.5 by choosing the pseudo-time step Δt appropriately.

3.5 Algorithm: Fractional Step Method

Finally, the algorithm for solving the NS equations using the fractional step method is provided here:

1. Generate computational grid.
2. Initialize the velocity and pressure fields.
3. Compute intermediate velocities u^* and v^*.
4. Adjust right boundary to enforce divergence-free condition.
5. Solve the pressure Poisson equation (PPE) to obtain p^n.
6. Correct the velocity field using u^*, v^*, and p^n.
7. Check for convergence: if not converged, return to Step 3; else, terminate.

4 Results and Discussions

4.1 Domain length study

The outlet or the right-side boundary of the domain is given a zero gradient boundary condition which implies that the flow is fully developed at that boundary. The length of the domain should be chosen such that this condition

is satisfied. The velocity profile is calculated the different points of the domain for the highest Re flow to estimate the domain length at which the flow becomes fully developed. The results are shown in Fig. 4

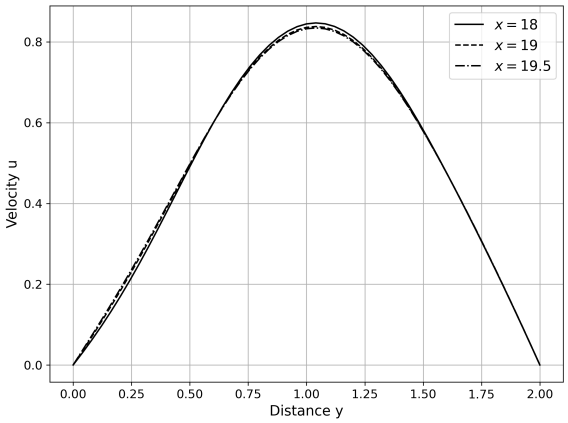

Figure 4: Domain length study: Re 400

The domain length study shows that after a length of 19 step heights, significant changes in the velocity field is not observed, Hence, the flow can be safely assumed to be fully developed beyond with a domain having a length equal to 20 step heights. All the simulations for different Re are carried out with a domain having a length of 20 step heights.

4.2 Grid sensitivity study

All numerical methods require grids which can affect the results of the simulation. Hence, a grid sensitivity study is done to choose the appropriate grid size that beyond which there will not be any significant changes in the solution field due to grid refinement. The study is conducted for the highest Re test case. The details of the study are shown in Table 1. The fully developed velocity profiles near the outlet boundary are compared. The results are shown in Fig. 5

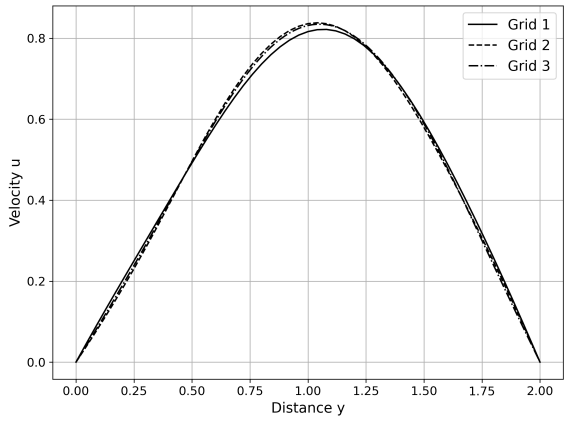

Figure 5: Grid sensitivity study: Re 400

It can be observed from Fig. 5 that the velocity profiles don't vary significantly between Grid 2 and Grid 3. Hence Grid 2 is chosen for simulating different Reynolds number flows.

Grid	Number of cells
Grid 1	2500
Grid 2	5000
Grid 3	10000

Table 1: Details of grids used for the sensitivity study

4.3 Flow visualization at different Reynolds numbers

The backward facing step introduces recirculation zones due to sudden expansion. The existence and the length of the recirculation zones depends on the Reynolds number of the flow [2]. Hence, the accuracy of the numerical solution can be observed by visualizing the streamlines of the flow and comparing the length of recirculation zones with those present in literature [1, 2]. The streamlines are shown in Fig. 6.

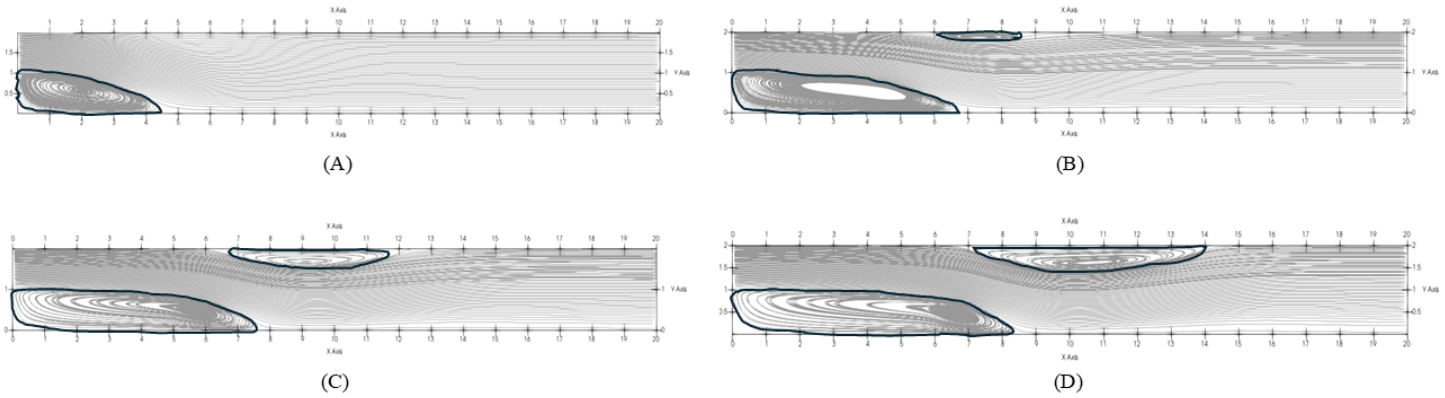

Figure 6: Streamlines at different Reynolds numbers.
(A) Re:100, (B) Re:200, (C) Re:300, (D) Re:400

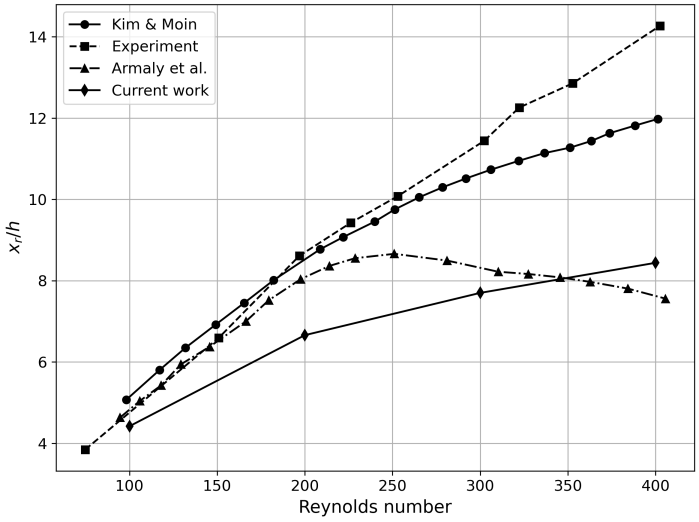

Figure 7: Recirculation zone length comparison
Kim & Moin[2], Experiment and Armaly et al[1]

The first observation that could be made from Fig. 6 is that the length of the recirculation zone increases with increase in Reynolds number. At low Reynolds number of 100, no recirculation is observed at the top wall which is consistent with literature [1, 2]. At Reynolds number of 200 a small recirculation zone is observed at the top wall. This is observed only in numerical calculations and not in experimental observation [1, 6]. Beyond Reynolds number of 300, there is a consistent recirculation zone at the top and bottom walls.

Figure 7 shows the comparison of the recirculation zone length between current work and previous works presented in literature. x_r is the length of the recirculation zone at the bottom wall in $metres$. The recirculation zone length obtained from current work compares well for low Reynolds numbers (Re$<$200). For higher Reynolds numbers, the current work under predicts the length. This may be attributed to the lower order convection scheme i.e. upwinding scheme used for discretizing the convecting fluxes. The upwind scheme is highly dissipative which leads to the loss of accuracy.

5 Summary and Conclusions

A numerical code to solve steady incompressible NS equations in two dimensions is presented in work by applying it to solve flow over a backward facing step. The solver captured all the features in the flow field like the recirculation zones. Though the solver under predicted the recirculation zone length at higher Reynolds numbers, it was able to capture the overall trend in the flow i.e. the appearance of the secondary recirculation zone at the top at higher Reynolds numbers and increase in the length of the recirculation zone with increase in Reynolds number.

References

[1] B. F. Armaly, F. Durst, J. Pereira, and B. Schönung, "Experimental and theoretical investigation of backward-facing step flow," *Journal of fluid Mechanics*, vol. 127, pp. 473–496, 1983.

[2] J. Kim and P. Moin, "Application of a fractional-step method to incompressible navier-stokes equations," *Journal of Computational Physics*, vol. 59, no. 2, pp. 308–323, 1985.

[3] A. J. Chorin, "Numerical solution of the navier-stokes equations," *Mathematics of computation*, vol. 22, no. 104, pp. 745–762, 1968.

[4] F. H. Harlow and J. E. Welch, "Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface," *The Physics of Fluids*, vol. 8, pp. 2182–2189, 12 1965.

[5] S. V. Patankar, "Numerical methods in heat transfer," in *Proceeding of International Heat Transfer Conference 7*, pp. 83–90, Begellhouse, 1982.

[6] M. A. Hossain, M. T. Rahman, and S. Ridwan, "Numerical investigation of fluid flow through a 2d backward facing step channel,"

[7] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–6, 2015.

A Parallelization

The complete parallelization of the code requires the use of PETSc library or mpi4py library of python. In this work, a simple acceleration procedure is used. The numba library [7] available in python is used for speeding up the simulation. The ”JIT (just-in-time)” decorator in numba library allows for compiling the python code into a machine code which bypasses the normal interpreter based functioning of python. This drastically increased the speed of the calculation. The JIT functionality is used in the ADI solver for the pressure poisson equation. The normal code without the JIT function took 10 hours to run 960 iterations while the code with the JIT function took approximately 20 minutes to calculate 4227 iterations. The solution speed can be further enhanced by parallelization.

B Numerical implementation

```
print("Inside file: loading packaages")

import numpy as np

from numba import njit,prange

import evtk

print("Packages imported")

# grid generation
delX = 0.2
delY = 0.04
x_max = 20
y_max = 2
x_grid = np.arange(0,x_max+delX,delX)
y_grid = np.arange(0,y_max+delY,delY)
Nx = len(x_grid)
Ny = len(y_grid)

grid_spacing = min(delX,delY)

beta = delX/delY
maxCo = 0.5
delT = maxCo * grid_spacing/1.5
Re = 400



def parabolic_vel_profile(y_tilda):
    temp = (y_tilda - 1.5)**2
    u_tilda = 1.5 - 6*temp
```

```
    return u_tilda


def func_uStar(u_old,v_old):
    u_Star = np.copy(u_old)
    for j in range(0,Ny-1): #All the nodal values need to be estimated
        for i in range(0,Nx-2): #Two boundary nodes are removed [control
                                #volume is written for uP = u[j,i+1]]
            uE = u_old[j,i+2]
            uP = u_old[j,i+1]
            uW = u_old[j,i]
            if(j!=0):
                uS = u_old[j-1,i+1]
            else:
                uS = 0 #vs will be zero due to no slip. So no need to
                       #calculate uS explicitly. dudy_s is sufficient
            if(j!=Ny-2):
                uN = u_old[j+1,i+1]
            else:
                uN=0 #vn will be zero due to no slip. Hence dudy_n is sufficient
            uw = 0.5*(uW+uP)
            ue = 0.5*(uP+uE)

            vn = 0.5*(v_old[j+1,i]+v_old[j+1,i+1])
            vs = 0.5*(v_old[j,i]+v_old[j,i+1])

            dudx_e = (uE-uP)/delX
            dudx_w = (uP-uW)/delX

            if(j!=0):
                dudy_s = (uP-uS)/delY
            else:
                dudy_s = (9*uP - uN)/(3*delY)

            if(j!=Ny-2):
                dudy_n = (uN-uP)/delY
            else:
                dudy_n = (uS-9*uP)/(3*delY)


            #Upwinding the convection term to make the scheme stable. CS is
            #not a stable scheme for convection
            uw_uw = 0.5*(np.abs(uw)+uw)*uW - 0.5*(np.abs(uw)-uw)*uP
            ue_ue = 0.5*(np.abs(ue)+ue)*uP - 0.5*(np.abs(ue)-ue)*uE

            us_vs = 0.5*(np.abs(vs)+vs)*uS - 0.5*(np.abs(vs)-vs)*uP
            un_vn = 0.5*(np.abs(vn)+vn)*uP - 0.5*(np.abs(vn)-vn)*uN
```

```
            #Assembling all together
            term_1 = (uw_uw-ue_ue)/delX
            term_2 = (us_vs - un_vn)/delY
            term_3 = (dudx_e-dudx_w)/delX
            term_4 = (dudy_n-dudy_s)/delY

            Ru = term_1 + term_2 + (term_3 + term_4)/Re
            u_Star[j,i+1] = u_old[j,i+1] + delT*Ru #delT is the relaxation
                                                   #factor for updation
    return u_Star


def func_vStar(u_old,v_old):
    v_Star = np.copy(v_old)
    for j in range(0,Ny-2):
        for i in range(0,Nx-1):
            vP = v_old[j+1,i]
            vN = v_old[j+2,i]
            vS = v_old[j,i]

            if(i!=0):
                vW = v_old[j+1,i-1]
            else: #uw will be zero and the flow at inlet is uni-directional.
                  #So vW is zero
                vW = 0

            if(i!=Nx-2):
                vE = v_old[j+1,i+1]
            else:
                vE = vP

            uw = 0.5*(u_old[j+1,i]+u_old[j,i])
            ue = 0.5*(u_old[j+1,i+1]+u_old[j,i+1])

            vn = 0.5*(vN+vP)
            vs = 0.5*(vP+vS)

            #upwinding for convective flux to achieve satbility
            uw_vw = 0.5*(np.abs(uw)+uw)*vW - 0.5*(np.abs(uw)-uw)*vP
            ue_ve = 0.5*(np.abs(ue)+ue)*vP - 0.5*(np.abs(ue)-ue)*vE

            vs_vs = 0.5*(np.abs(vs)+vs)*vS - 0.5*(np.abs(vs)-vs)*vP
            vn_vn = 0.5*(np.abs(vn)+vn)*vP - 0.5*(np.abs(vn)-vn)*vN

            #diffusive fluxes
            dvdx_e = (vE-vP)/delX
```

```
            dvdx_w = (vP-vW)/delX
            dvdy_n = (vN-vP)/delY
            dvdy_s = (vP-vS)/delY

            #putting all together
            term_1 = (uw_vw-ue_ve)/delX
            term_2 = (vs_vs-vn_vn)/delY
            term_3 = (dvdx_e-dvdx_w)/delX
            term_4 = (dvdy_n-dvdy_s)/delY

            Rv = term_1+term_2+(term_3+term_4)/Re
            v_Star[j+1,i] = v_old[j+1,i]+delT*Rv
    return v_Star

@njit
def func_tdma(a,b,c,d):
    n = len(d)
#     a, b, c, d = map(np.array, (a, b, c, d))
    a = np.copy(a)
    b = np.copy(b)
    c = np.copy(c)
    d = np.copy(d)
    for i in range(1,n):
        b[i] = b[i] - c[i-1]*(a[i]/b[i-1])
        d[i] = d[i] - d[i-1]*(a[i]/b[i-1])
    out = np.zeros_like(d)
    out[n-1] = d[n-1]/b[n-1]
    j=n-2
    while(j>=0):
        out[j] = (d[j] - c[j]*out[j+1])/b[j]
        j=j-1
    return out


#Convergence check for ADI
@njit(parallel=True)
def func_checkConvergence_ADI(pk,pk_1):
    yp = np.shape(pk)[0]#no. of rows. iterate along 'y' axis
    xp = np.shape(pk)[1]#no. of columns. iterate along 'x' axis
    error=0.0
    p_sum = 0.0
    for j in prange(yp): #only the interior nodes to be checked
        for i in range(xp):
            error+=np.abs(pk_1[j,i]-pk[j,i])
            p_sum +=np.abs(pk[j,i])
    rel_err = error/(p_sum+1e-10) #Added a very small number to avoid divide
                                  #by zero error
```

```
    return rel_err

@njit(parallel=True)
def func_RHS_ADI(u_star,v_star):
    rhs = np.zeros((Ny-1,Nx-1))
    for j in prange(Ny-1):
        for i in range(Nx-1):
            rhs[j,i] = (u_star[j,i+1]-u_star[j,i])/delX +
            (v_star[j+1,i]-v_star[j,i])/delY
    rhs = rhs * (delX**2)/delT
    return rhs



@njit(parallel=True)
def func_ADI(p_init,u_star,v_star):
    p_var = np.copy(p_init)
    rhs = func_RHS_ADI(u_star,v_star)

    converged_ADI = False
    iteration = 0
    b_x = np.zeros(Nx-1)
    a_x = np.zeros_like(b_x)
    c_x = np.zeros_like(b_x)

    a_x[:] = 1
    b_x[:] = -2*(1+beta**2)
    c_x[:] = 1

    b_y = np.zeros(Ny-1)
    a_y = np.zeros_like(b_y)
    c_y = np.zeros_like(b_y)

    a_y[:] = beta**2
    b_y[:] = -2*(1+beta**2)
    c_y[:] = beta**2

    while(not converged_ADI):
        #x-sweep
        p_var[0,0] = 0

        for row in prange(Ny-1):
            d_x = np.zeros_like(b_x)
            for col in range(0,Nx-1):
                if(row!=0): #For the bottom row, no nodes exist to the south
                    p_bottom = p_var[row-1,col]
                else: #gradient is zero
```

```
                p_bottom = p_var[0,col]
            if(row!=Ny-2): #Similarly no node at north for the top cell
                p_top = p_var[row+1,col]
            else:
                p_top = p_var[-1,col]
            d_x[col] = rhs[row,col] - np.power(beta,2)*(p_top+p_bottom)
        d_x[0] = d_x[0] - p_var[row,0]
        d_x[-1] = d_x[-1] - p_var[row,-1]
        result_x = func_tdma(a_x,b_x,c_x,d_x)
        p_var[row,:] = result_x

    #y-sweep

    for col in prange(Nx-1):
        d_y = np.zeros_like(b_y)
        for row in range(0,Ny-1):
            if(col!=0):
                p_left = p_var[row,col-1]
            else:
                p_left = p_var[row,0]
            if(col!=Nx-2):
                p_right = p_var[row,col+1]
            else:
                p_right = p_var[row,-1]
            d_y[row] = rhs[row,col] - (p_left + p_right)
        d_y[0] = d_y[0] - np.power(beta,2)*(p_var[0,col])
        d_y[-1] = d_y[-1] - np.power(beta,2)*(p_var[-1,col])
        result_y = func_tdma(a_y,b_y,c_y,d_y)
        p_var[:,col] = result_y

    #check convergence of ADI:
    error_ADI = func_checkConvergence_ADI(p_init,p_var)
    iteration+=1
    if(error_ADI<1e-3):
        converged_ADI = True
        print("ADI converged")
        print(iteration)
        print(error_ADI)
    else:
        p_init = np.copy(p_var)
return p_var


def func_u_v_update(uStar,vstar,pressure):
    u_np1 = np.copy(uStar)
    v_np1 = np.copy(vStar)
```

```
    for i in range(0,Nx-2):
        for j in range(0,Ny-1):
            u_np1[j,i+1] = uStar[j,i+1] - (delT/delX)*(pressure[j,i+1] -
                                                       pressure[j,i])

    for i in range(0,Nx-1):
        for j in range(0,Ny-2):
            v_np1[j+1,i] = vStar[j+1,i] - (delT/delY)*(pressure[j+1,i] -
                                                       pressure[j,i])

    return u_np1,v_np1



# Algorithm - NS equation

#Variable creation
u_init = np.zeros((Ny-1,Nx))#rows by columns
v_init = np.zeros((Ny,Nx-1))
p_init = np.zeros((Ny-1,Nx-1))

#assigning BC
loc_inlet = 0 # variable for identifying where the inlet begins.
for j in range(0,Ny-1):
    if(y_grid[j]>=1.0):
        u_init[j,0] = parabolic_vel_profile(y_grid[j]+delY/2)
        if(loc_inlet==0):
            loc_inlet = j #node number at which the inlet begins


converged_mom = False
iteration_mom = 0
while(not converged_mom):
    iteration_mom+=1

    #Calcualte uStar
    uStar = func_uStar(u_init,v_init)

    #Calculate vStar
    vStar = func_vStar(u_init,v_init)

    #Applying BC for u at the right face i.e. at the outflow boundary
    uStar[:,-1] = uStar[:,-2]

    #Adjusting for flow rate imbalance between inlet and outlet
    area_inlet = (Ny-1-loc_inlet)*delY
    flow_rate_inlet = np.sum(uStar[loc_inlet:,0])*delY #non-dimensional
```

```
    flow_rate_outlet = np.sum(uStar[:,-1])*delY

    flow_rate_imbalance = flow_rate_outlet - flow_rate_inlet

    u_corr = flow_rate_imbalance/((Ny-1)*delY)

    uStar[:,-1] = uStar[:,-1] - u_corr

    #Solve poisson-equation for pressure
    p_n = func_ADI(p_init,uStar,vStar)

    #Correct the velocities
    u_np1,v_np1 = func_u_v_update(uStar,vStar,p_n)

    du = np.abs(u_np1 - u_init).max()
    dv = np.abs(v_np1 - v_init).max()

    max_res = max(du,dv)
    if(max_res<1e-4):
        print(f"Momentum equation converged in {iteration_mom} iteration")
        converged_mom = True
    else:
        if(iteration_mom%20 == 0):
            print(f"Momentum equation: Iteration {iteration_mom},
            Residual: {max_res}")
        u_init = np.copy(u_np1)
        v_init = np.copy(v_np1)




def interpolate_to_Nodes(u_val,v_val):
    u_Node = np.zeros((Ny,Nx))
    v_Node = np.zeros_like(u_Node)

    for i in range(0,Nx-1):
        for j in range(1,Ny-1):
            u_Node[j,i] = 0.5*(u_val[j,i]+u_val[j-1,i])
    for i in range(1,Nx-1):
        for j in range(1,Ny-1):
            v_Node[j,i] = 0.5*(v_val[j,i]+v_val[j,i-1])

    #right boundary value based on fully developed flow assumption
    u_Node[:,-1] = u_Node[:,-2]
    v_Node[:,-1] = v_Node[:,-2]

    return u_Node, v_Node
```

```
u_node, v_node = interpolate_to_Nodes(u_np1,v_np1)


z_grid = np.array([0.0, 0.1])
u_vtk = np.zeros((Nx, Ny, 2))  # VTK ordering (x,y,z)
v_vtk = np.zeros((Nx, Ny, 2))

for k in range(2):  # Both z-layers get the same 2D data
    u_vtk[:, :, k] = u_node.T # The x and y axis are transposed between
                              #numpy and paraview
    v_vtk[:, :, k] = v_node.T


evtk.hl.gridToVTK("./output",x=x_grid,
                    y=y_grid,z=z_grid,pointData={"Ux":u_vtk,"Uy":v_vtk})
```