AE625 - Computational Fluid Dynamics
Assignment - 05
Numerical simulation of Lid Driven Cavity Flow Problem

Ramkumar S. *

SC22M007, M.Tech. Aerospace - Aerodynamics and Flight Mechanics

The numerical computation of 2D lid driven cavity flow problem was performed with the domain of size 2X2 meters with top wall being moved at 1m/s velocity. Air is chosen as the working fluid with incompressibility assumption. The simulation was performed using finite difference method on a staggered grid using pressure correction technique and SIMPLE algorithm. Solutions obtained were post-processed for a range of timesteps and the contours of pressure, velocity magnitude and voricity were made. the streamlines were also plotted and the animations showing the change in flow field were also generated.

I. Introduction

The development of numerical solver that solves the lid driven cavity flow problem on a square domain of side 2 meters was performed in the present work. Air under standard atmospheric conditions was chosen as the working fluid and the computation was perfomed using finite-difference method.

The analysis was supposed to be performed on a colocated grid with fractional step method, but the method was found to be stiff and requiring quite more time to fix, hence the present work was done using the staggered-grid pressure correction technique and SIMPLE algorithm. [1] is used as the reference for the computation procedure.

II. Computation procedure

The 2D incompressible Navier-Stokes equations were solved in the present work in the partial differential form as given in Equations (1) to (3).

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

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{2}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \tag{3}$$

The computation was performed on a *forward staggered grid*, i.e. the staggered grid with pressure nodes covering the boundaries and velocity nodes present between them at equal spaces. The schematic of the domain is shown in Figure 1.

Then the equations are discretized with finite difference technique using central difference for all the spacial derivatives and forward difference for the temporal derivative. The discretized and rearranged momentum equations are given in Equations (4) and (5)

*SC22M007, M.Tech., Aerospace AFM

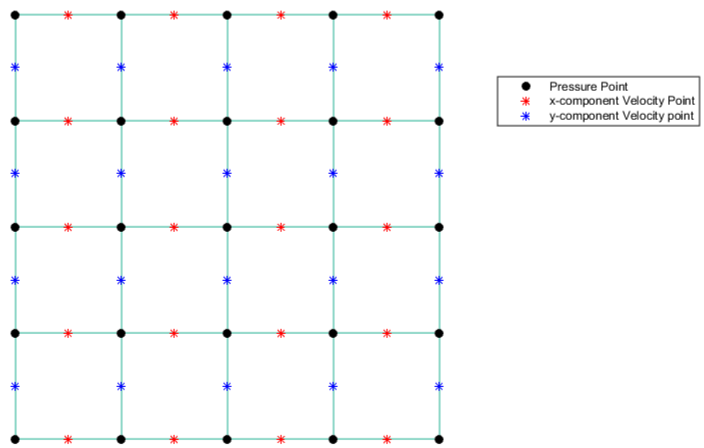


Fig. 1 forward staggered grid arrangement

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \Delta t\left(-F_{term} + D_{term} - P_{term}\right) \tag{4}$$

where,

$$F_{term} = \frac{u_{i+1,j}^2 - u_{i-1,j}^2}{2dx} + \frac{uv_{i,j+1} - uv_{i,j-1}}{2dy}$$

$$D_{term} = \nu\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{dx^2} + \frac{u_{i,j+1} - 2ui,j + ui,j-1}{dy^2}\right)$$

$$P_{term} = \frac{1}{\rho}\frac{p_{i+1,j} - p_{i-1,j}}{2dx}$$

$$v_{i,j}^{n+1} = v_{i,j}^{n} + \Delta t\left(-F_{term} + D_{term} - P_{term}\right) \tag{5}$$

where,

$$F_{term} = \frac{uv_{i+1,j} - uv_{i-1,j}}{2dx} + \frac{v_{i,j+1}^2 - v_{i,j-1}^2}{2dy}$$

$$D_{term} = \nu\left(\frac{v_{i+1,j} - 2v_{i,j} + v_{i-1,j}}{dx^2} + \frac{v_{i,j+1} - 2vi,j + vi,j-1}{dy^2}\right)$$

$$P_{term} = \frac{1}{\rho}\frac{p_{i,j+1} - p_{i,j-1}}{2dy}$$

The pressure correction equation was derived as given in [1] and is given as Equation (6)

$$ap'_{i,j} + b(p'_{i+1,j} + p'_{i-1,j}) + c(p'_{i,j+1} + p'_{i,j-1}) + d = 0 \tag{6}$$

where, the coefficients are as given below.

$$
\begin{aligned}
a &= \left(\frac{2dt}{dx^2} + \frac{2dt}{dy^2}\right) \\
b &= -\frac{dt}{dx^2} \\
c &= -\frac{dt}{dy^2} \\
d_{i,j} &= \frac{1}{dx}\left(\rho u^*_{i,j} - \rho u^*_{i-1,j}\right) + \frac{1}{dy}\left(\rho v^*_{i,j} - \rho v^*_{i,j-1}\right)
\end{aligned}
$$

further the computed pressure correction field p' is used to correct the pressure and velocity as given in Equations (7) to (9).

$$p_{i,j} = p^*_{i,j} + \alpha p'_{i,j} \tag{7}$$

$$u_{i,j} = u^*_{i,j} - \alpha \left.\frac{\partial p'}{\partial x}\right|_{i,j} \tag{8}$$

$$v_{i,j} = v^*_{i,j} - \alpha \left.\frac{\partial p'}{\partial y}\right|_{i,j} \tag{9}$$

where, $\alpha = 0.1$ is the under-relaxation factor used in the present computation work.

It is to be noted that there will be three overlapping grids, each for each flow field variable *u,v,p*, hence in the code, the indexing was taken care such that the velocity at a given point is driven by the pressure nodes surrounding it. This has to be taken care on doing coding.

The SIMPLE (Semi-Implicit Method for Pressure Linked Equations) algorithm was implemented for the computation of solution fields. The steps followed are as follows.

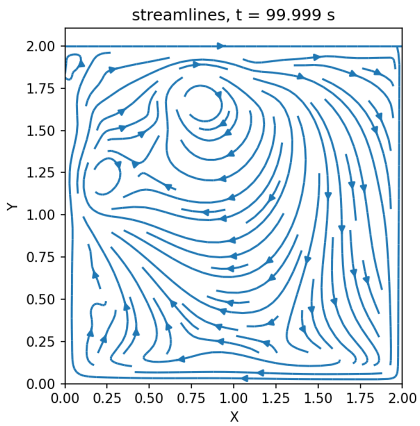
1) Initial pressure field was assumed and Equations (4) and (5) were solved for an intermediate velocity field which is called as u^*, v^*.
2) Then the mass imbalance term d in Equation (6) was computed then the pressure correction field p' was computed.
3) The computed pressure correction field was then used to update the pressure and velocity field using Equations (7) to (9).
4) The above procedure is repeated till the solution converges, i.e. the mass imbalance term $d \rightarrow 0$.

Since, this is a driven cavity flow problem, the generation of flow vorticity will be significant and it would be better to visualize the results in terms of the vorticity contour. Hence the Equation (10) is used to compute the vorticity term during post-processing.

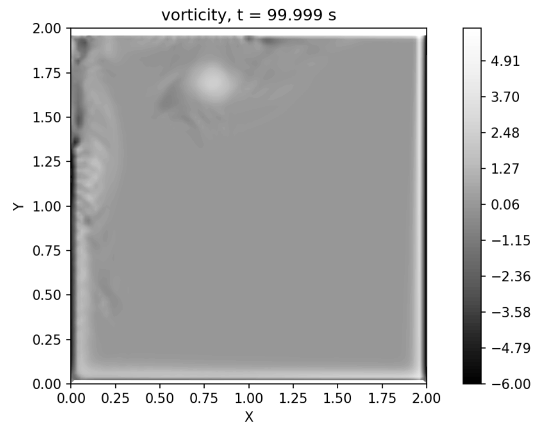
$$\omega = \frac{\partial u}{\partial y} - \frac{\partial v}{\partial x} \tag{10}$$

III. Results

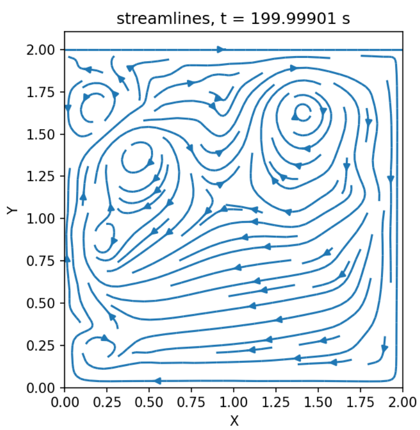
The simulation was carried out for 300 seconds to study how the flow pattern emerges with time. The intermediate timestep solution fields were saved to csv files and were used in post-processing to generate animations of solution fields with time. The several contours obtained at different time steps and are shown in Figures 2 and 3.



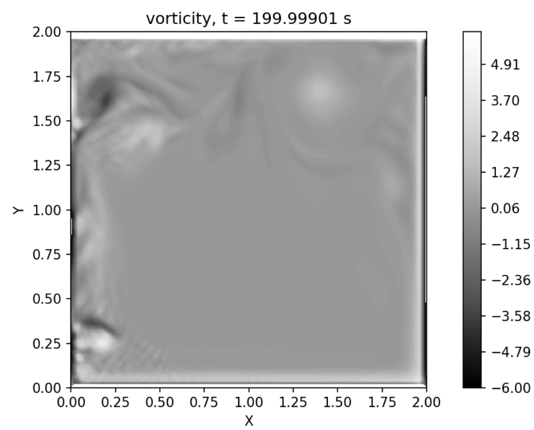
(a) streamline pattern, t = 100 s



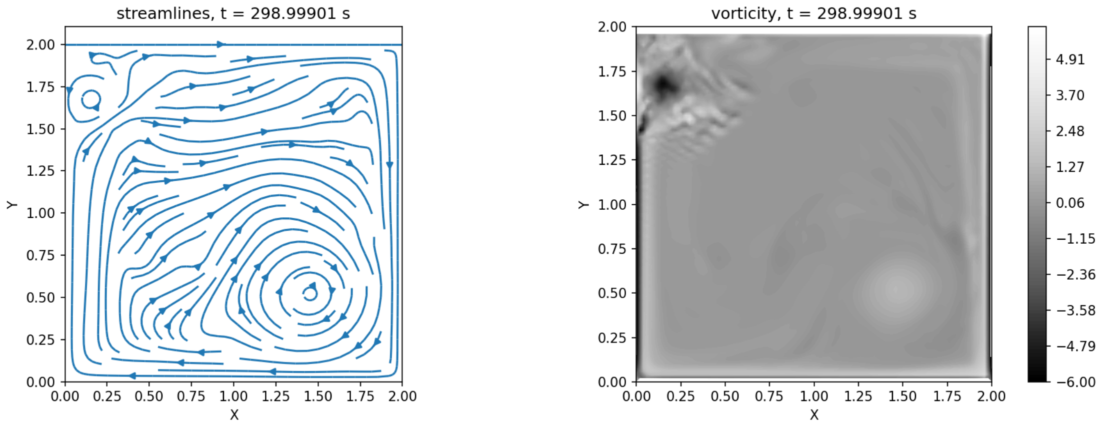
(b) vorticity, t = 100 s



(c) streamline pattern, t = 200 s



(d) vorticity, t = 200 s



(e) streamline pattern, t = 299 s **(f) vorticity, t = 299 s**

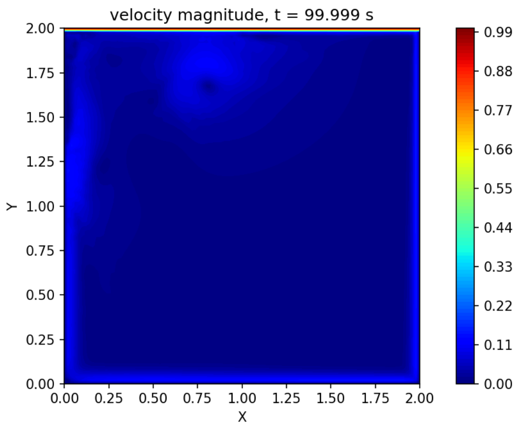
Fig. 2 vorticity and streamlines patterns at different solution timesteps

IV. Conclusion and further works

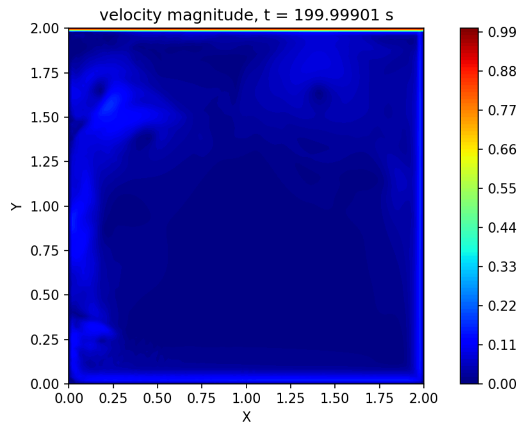
The numerical computation of lid driven cavity flow problem for the given configuration was successfully performed using staggered-grid finite difference method and pressure correction technique, SIMPLE algorithm. The further work will be based on the investigation on why the fractional step method did not work and the development of the code based on fractional step method that solves the same problem. The codes developed for this assignment are given in Section A.

References

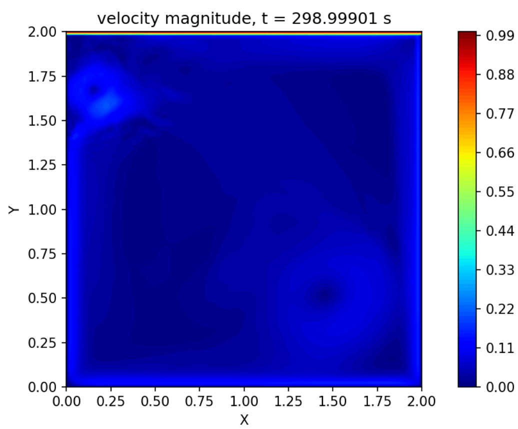
[1] McLay, A. "Computational Fluid Dynamics: the Basics with Applications, JD Anderson, McGraw-Hill Book Company Europe, McGraw-Hill House, Shoppenhangers Road, Maidenhead, Berkshire SL6 2QL. 1995. 547pp. Illustrated.£ 23.95." The Aeronautical Journal 100.998 (1996): 365-365.



(a) velocity magnitude, t = 100s



(b) velocity magnitude, t = 200s



(c) velocity magnitude, t = 299s

Fig. 3 velocity magnitude contours at different simulation times

A. Appendix - FORTRAN and Python codes

This section contains the FORTRAN and Python source codes used in solving the lid driven cavity flow problem.

Main solver FORTRAN code

```
!--------------------------------------------------------------------
! Navier Stokes Solver using FDM
! Lid driven cavity flow problem
!
! Main solver code
!--------------------------------------------------------------------

program main

    ! importing needed modules
    use modelVars

    ! initializing flow field variables
    call initializer()

    ! begin main loop computation
    mainloop: do itr = 1,N_timestep
    ! mainloop: do itr = 1,10

        ! solving momentum equation
        call solve_mmtmEqn()

        ! correcting pressure
        call solve_pEqn()

        ! update velocity
        call update_velocity()

        ! updating on screen
        print *,"iteration : ",itr,"/",N_timestep,"; time = ",time

        if (mod(itr,1000) == 0) then
            ! interpolate fields
            call interpolate_field()
            ! writing output to file
            call write_csv()
        endif

        ! updating timestep
        time = time + dt;

    end do mainloop

    ! ! interpolate fields
    ! call interpolate_field()
    !
    ! ! writing output to file
    ! call write_csv()

end program main
```

Parameters module FORTRAN code

```
!--------------------------------------------------------------------
! Navier Stokes Solver using FDM
! Lid driven cavity flow problem
!
! parameters definition file
!--------------------------------------------------------------------

! parameters module
module parameters

    ! overriding pre-allocation of datatypes
```

```
    implicit none

    ! defining kind of variables
    integer, parameter :: ikd = selected_int_kind(8)
    integer, parameter :: rkd = selected_real_kind(8,8)

    ! length of domain in x and y directions
    real(kind=rkd), parameter :: Lx = 2.0, Ly = 2.0

    ! fluid parameters
    real(kind=rkd), parameter :: rho = 1.225, nu = 1.4604e-5

    ! number of grid points in x and y direction
    integer(kind=ikd), parameter :: Nx = 101, Ny = 101

    ! plate velocity definition
    real(kind=rkd), parameter :: Uplate = 1.0

    ! simulation time and timestep
    real(kind=rkd), parameter :: endTime = 300.0, dt = 1e-3

    ! initializing grid points
    integer(kind=rkd), parameter :: npx = nx + 1, npy = ny + 1
    integer(kind=rkd), parameter :: nux = npx - 1, nuy = npy
    integer(kind=rkd), parameter :: nvx = npx, nvy = npy - 1

end module parameters

! model variables module
module modelVars

    ! importing needed modules
    use parameters

    ! scalar variables
    real(kind=rkd) :: time, dx, dy

    ! label variables
    integer(kind=ikd) :: itr, i,j, N_timestep = 0, filecount = 1

    ! scalar arrays
    real(kind=rkd),dimension(Nx,Ny) :: X,Y,Ui,Vi,Pi
    real(kind=rkd), dimension(nuy,nux) :: U,Us
    real(kind=rkd), dimension(nvy,nvx) :: V,Vs
    real(kind=rkd), dimension(npy,npx) :: P,PP,d,PPs

    ! declaring character strings for filename
    character(len=50) :: filename


end module modelVars
```

Subroutines Module FORTRAN code

```
!------------------------------------------------------------------------
! Navier Stokes Solver using FDM
! Lid driven cavity flow problem
!
! Subroutines definition file
!------------------------------------------------------------------------

! initializer
! subroutine-------------------------------------------------------------
subroutine initializer()

    ! importing needed modules
    use modelVars

```

```fortran
    implicit none

    ! initializing grid
    dx = Lx/float(Nx-1)
    dy = Ly/float(Ny-1)
    do i = 1,Nx
        do j = 1,Ny
            X(j,i) = float(i-1)*dx
            Y(j,i) = float(j-1)*dy
        end do
    end do

    ! initializing flow field
    U = 0; U(nuy,:) = Uplate; Us = U
    V = 0; Vs = V; P = 0

    ! computing number of timesteps
    N_timestep = int(endTime/dt)

    print *,"initialization done"


end subroutine initializer

! momentum equation solver
! subroutine-----------------------------------------------------------------------
subroutine solve_mmtmEqn()

    ! importing needed modules
    use modelVars

    implicit none

    ! declaring some local variables
    real(kind = rkd) :: F_term, D_term, P_term, va,vb,uc,ud

    ! solving x-momentum equation
    do i = 2,nux-1
        do j = 2,nuy-1
            va = 0.5*(v(j,i) + v(j,i+1))
            vb = 0.5*(v(j-1,i) + v(j-1,i+1))

            ! convection term
            F_term = (u(j,i+1)**2 - u(j,i-1)**2)/dx/2.0 + &
                (u(j+1,i)*va - u(j-1,i)*vb)/dy/2.0

            ! diffusion term
            D_term = nu*(u(j,i+1) - 2.0*u(j,i) + u(j,i-1))/dx**2 + &
                nu*(u(j+1,i) - 2.0*u(j,i) + u(j-1,i))/dy**2

            ! pressure term
            P_term = 1.0/rho*(p(j,i+1) - p(j,i))/dx

            us(j,i) = u(j,i) + dt*(-F_term + D_term - P_term)
        end do
    end do

    ! solving y-momentum equation
    do i = 2,nvx-1
        do j = 2,nvy-1
            uc = 0.5*(u(j,i-1) + u(j+1,i-1))
            ud = 0.5*(u(j,i) + u(j+1,i))

            ! convection term
            F_term = (ud*v(j,i+1) - uc*v(j,i-1))/dx/2.0 + &
                    (v(j+1,i)**2 - v(j-1,i)**2)/dy/2.0

            ! diffusion term
```

```
            D_term = nu*(v(j,i+1) - 2.0*v(j,i) + v(j,i-1))/dx**2 + &
                nu*(v(j+1,i) - 2.0*v(j,i) + v(j-1,i))/dy**2

            ! pressure term
            P_term = 1.0/rho*(p(j+1,i) - p(j,i))/dy

            vs(j,i) = v(j,i) + dt*(-F_term + D_term - P_term)
        end do
    end do


end subroutine solve_mmtmEqn

! pressure equation solver
! subroutine-----------------------------------------------------------------
subroutine solve_pEqn()

    ! importing needed modules
    use modelVars

    implicit none

    ! declaring some local variables
    real(kind=rkd) :: a,b,c,err
    integer(kind=ikd) :: p_itr

    ! computing coefficients
    a = 2.0*(dt/dx**2 + dt/dy**2)
    b = - dt/dx**2
    c = - dt/dy**2
    do i = 2,npx-1
        do j = 2,npy-1
            d(j,i) = rho/dx*(Us(j,i)-Us(j,i-1))+rho/dy*(Vs(j,i)-Vs(j-1,i))
        end do
    end do

    ! solving pressure correction equation
    pp = 0.0
    do p_itr = 1,100
        ! solving equation
        do i = 2,npx-1
            do j = 2,npy-1
                pp(j,i) = -1.0/a*(b*pp(j,i+1) + b*pp(j,i-1) + c*pp(j+1,i) + c*pp(j-1,i) + d(j,i))
            end do
        end do

        ! extrapolating pressure correction
        pp(:,1) = 2.0*pp(:,2) - pp(:,3)
        pp(:,npx) = 2.0*pp(:,npx-1) - pp(:,npx-2)
        pp(1,:) = 2.0*pp(2,:) - pp(3,:)
        pp(npy,:) = 2.0*pp(npy-1,:) - pp(npy-2,:)

        ! checking convergence
        err = maxval(abs(pp - pps))
        pps = pp

        if (err < 1e-4) then
            exit
        end if

    end do

    print *,"pressure converged in ",p_itr, err

    P = P + 0.1*pp

end subroutine solve_pEqn

```

```fortran
! update velocity subroutine-----------------------------------------------
subroutine update_velocity()

    ! importing needed modules
    use modelVars

    implicit none

    ! updating x-velocity
    do i = 2,nux-1
        do j = 2,nuy-1
            U(j,i) = Us(j,i) - 0.1*dt/dx*(pp(j,i+1) - pp(j,i))
        end do
    end do

    do i = 2,nvx-1
        do j = 2,nvy-1
            V(j,i) = Vs(j,i) - 0.1*dt/dy*(pp(j+1,i) - pp(j,i))
        end do
    end do

    print *,"updated velocity"

end subroutine update_velocity

! interpolation subroutine
! definition------------------------------------------------------------
subroutine interpolate_field()

    ! importing needed modules
    use modelVars

    implicit none

    ! interpolating fields
    do i = 1,Nx
        do j = 1,Ny
            Ui(j,i) = 0.5*(u(j,i)+u(j+1,i))
            Vi(j,i) = 0.5*(v(j,i)+v(j,i+1))
            Pi(j,i) = 0.25*(p(j,i)+p(j,i+1)+p(j+1,i)+p(j+1,i+1))
        end do
    end do

    ! correcting boundary values
    Ui(:,1) = 0.0
    Ui(:,Nx) = 0.0
    Ui(1,:) = 0.0
    Ui(Ny,:) = Uplate

    Vi(:,1) = 0.0
    Vi(:,Nx) = 0.0
    Vi(1,:) = 0.0
    Vi(Ny,:) = 0.0

    ! print *,"interpolation done"

end subroutine interpolate_field

! write csv subroutine
! definition------------------------------------------------------------
subroutine write_csv()

    ! importing needed modules
    use modelVars

    implicit none

    ! preparing filename to write solution data
```

```
    write(filename, "(A,I0.10,A)") "solution_data_",filecount,".csv"

    ! opening file with filename
    open(unit = 1, file = "solution_data/"//filename)

    ! writing header
    write(unit = 1, fmt = "(A)") "X Y Z U V P Time"

    ! looping to write data to file
    do i = 1,Nx
        do j = 1,Ny
            write(unit = 1, fmt = *) X(j,i), Y(j,i), 0.0, Ui(j,i), Vi(j,i), Pi(j,i),time
        end do
    end do

    ! closing file
    close(unit = 1)

    ! print *,"solution data writen to csv file ", filename

    filecount = filecount + 1

end subroutine write_csv
```

The Python script for postProcessing the data

```
#!/bin/python3

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import os,glob

# preparing directories to store contours
os.system("rm -rf contours")
os.system("mkdir -p contours/pressure")
os.system("mkdir -p contours/x-velocity")
os.system("mkdir -p contours/y-velocity")
os.system("mkdir -p contours/velocity-magnitude")
os.system("mkdir -p contours/vorticity")
os.system("mkdir -p contours/streamlines")

# reading files
fnames = sorted(glob.glob1(os.getcwd()+"/solution_data/", "*.csv"))

Nx = 101
Ny = 101

# transforming data into matrix
X = np.zeros([Ny,Nx])
Y = np.zeros([Ny,Nx])
U = np.zeros([Ny,Nx])
V = np.zeros([Ny,Nx])
P = np.zeros([Ny,Nx])


# looping through the files
for name in fnames:
    # reading data
    fid = pd.read_csv("solution_data/"+name, delim_whitespace=True)

    # getting time
    time = fid['Time'].iloc[0]

    # getting filecount
    filecount = name.split("_")[2].split(".csv")[0]

    count = 0
```

```
    for i in range(Nx):
        for j in range(Ny):
            X[j,i] = fid['X'].iloc[count]
            Y[j,i] = fid['Y'].iloc[count]
            U[j,i] = fid['U'].iloc[count]
            V[j,i] = fid['V'].iloc[count]
            P[j,i] = fid['P'].iloc[count]
            count += 1

    # computing vorticity
    dx = X[0,1]-X[0,0] # step size
    dy = Y[1,0]-Y[0,0] # step size
    omega = np.zeros([Ny,Nx])
    for i in range(1,Nx-1):
        for j in range(1,Ny-1):
            omega[j,i] = (U[j+1,i]-U[j-1,i])/dy/2.0 - (V[j,i+1]-V[j,i-1])/dx/2.0
    for i in range(1,Nx-1):
        j = 0
        omega[0,i] = (U[j+1,i]-U[j,i])/dy - (V[j,i+1]-V[j,i-1])/dx/2.0
        j = Ny-1
        omega[0,i] = (U[j,i]-U[j-1,i])/dy - (V[j,i+1]-V[j,i-1])/dx/2.0
    for j in range(1,Ny-1):
        i = 0
        omega[j,i] = (U[j+1,i]-U[j-1,i])/dy/2.0 - (V[j,i+1]-V[j,i])/dx
        i = Nx-1
        omega[j,i] = (U[j+1,i]-U[j-1,i])/dy/2.0 - (V[j,i]-V[j,i-1])/dx

    i = 0; j = 0
    omega[j,i] = (U[j+1,i]-U[j,i])/dy - (V[j,i+1]-V[j,i])/dx
    i = Nx-1; j = 0
    omega[j,i] = (U[j+1,i]-U[j,i])/dy - (V[j,i]-V[j,i-1])/dx
    i = Nx-1; j = Ny-1
    omega[j,i] = (U[j,i]-U[j-1,i])/dy - (V[j,i]-V[j,i-1])/dx
    i = 0; j = Ny-1
    omega[j,i] = (U[j,i]-U[j-1,i])/dy - (V[j,i+1]-V[j,i])/dx

    # plotting contours
    Umag = np.sqrt(U**2+V**2)

    # magnitude contour
    plt.figure()
    plt.contourf(X,Y,Umag,100,cmap = 'jet')
    plt.colorbar()
    plt.axis('image')
    plt.title("velocity magnitude, t = "+str(np.round(time,5))+" s")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.savefig("contours/velocity-magnitude/velocityMagnitude_"+filecount+".png", dpi = 150)

    # x-velocity contour
    plt.figure()
    plt.contourf(X,Y,U,100,cmap = 'jet')
    plt.colorbar()
    plt.axis('image')
    plt.title("x-velocity, t = "+str(np.round(time,5))+" s")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.savefig("contours/x-velocity/x-velocity_"+filecount+".png", dpi = 150)

    # y-velocity contour
    plt.figure()
    plt.contourf(X,Y,V,100,cmap = 'jet')
    plt.colorbar()
    plt.axis('image')
    plt.title("y-velocity, t = "+str(np.round(time,5))+" s")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.savefig("contours/y-velocity/y-velocity_"+filecount+".png", dpi = 150)
```

```
    # pressure contour
    plt.figure()
    plt.contourf(X,Y,P,100,cmap = 'jet')
    plt.colorbar()
    plt.axis('image')
    plt.title("pressure, t = "+str(np.round(time,5))+" s")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.savefig("contours/pressure/pressure_"+filecount+".png", dpi = 150)

    # streamlines
    plt.figure()
    plt.streamplot(X,Y,U,V)
    plt.axis('image')
    plt.title("streamlines, t = "+str(np.round(time,5))+" s")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.savefig("contours/streamlines/streamlines_"+filecount+".png", dpi = 150)

    # vorticity contour
    cmap = plt.cm.get_cmap('Greys').reversed()
    plt.figure()
    plt.contourf(X,Y,omega,100,cmap = cmap,levels = np.linspace(-6,6,100))
    plt.colorbar()
    plt.axis('image')
    plt.title("vorticity, t = "+str(np.round(time,5))+" s")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.savefig("contours/vorticity/vorticity_"+filecount+".png", dpi = 150)

    # plt.show()
    plt.close("all")

    print("processed ",name)
```

****