

Lid Driven Cavity Problem

Haobo Zhao

530.767 CFD-HW4

April 19, 2024

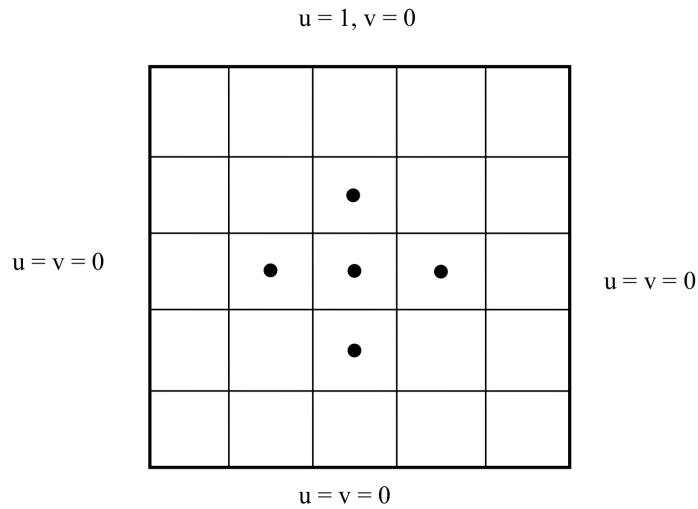
In this project, we developed a N-S solver for Lid Driven Cavity problem, which use 2nd order central difference scheme in space and a second order (AB2 for convection and CN2 for viscous) fractional step in time. By separate update of the face velocity (“semi-staggered” approach), we could approach zero divergence with the compact stencil for pressure. We also used ghost point method to obtain boundary condition. By compare our result with the reference result, we could say our result is robust.

Contents

1	Question Review	2
2	N-S Solver	3
2.1	Discretization of N-S Equation	3
2.1.1	Step (I):	3
2.1.2	Step (II):	4
2.1.3	Step (III):	5
2.1.4	LineSOR implementation	5
2.2	Setting	7
2.2.1	Solver field	7
2.2.2	Boundary condition and implementation	7
2.3	Solver Architecture	8
3	Result for Re=100	9
3.1	Stream lines at Re=100	9
3.1.1	Streamline comparison of different grid size result	10
3.2	Result comparison to the reference	10
3.2.1	Streamline comparison	10
3.2.2	Velocity at geometry center	10

4	Result for Re=1000	13
4.1	Stream lines at Re=1000	13
4.2	Result comparison to the reference	14
4.2.1	Velocity at geometry center	14
4.2.2	Streamline comparison	16
Appendix		18

1 Question Review



Consider the 1×1 cavity shown above. The cavity has a lid that moves to the right with unit velocity. This “driven cavity” problem is considered a benchmark in CFD and is often used to validate the fidelity and accuracy of Navier-Stokes solvers. Accurate solutions to this problem have been obtained by a number of researchers. One often quoted work in this area is that of Ghia et al. (*Journal of Computational Physics*, Vol. 48, page 387, 1982)[2]. Others include Botella, O. and Peyret, R., 1998. Benchmark spectral results on the lid-driven cavity flow. *Computers & Fluids*, 27(4), pp.421-433.[1]

With this HW, you will start developing the code that you will also use for the Final Project

1. Simulate the driven cavity flow with $Re = 100$ on uniform grids with 16, 32, 64, 128 points in each direction. Simulate the flow until a steady state is reached. Use a 2nd order central difference scheme in space and a second order (AB2 for convection and CN2 for viscous) fractional step in time. The basic (non-Van-Kan version of fractional step is acceptable). Use the separate update of the face velocity (“semi-staggered” approach) so that zero divergence can be satisfied with the compact stencil for pressure.
2. Compare your computed results (velocity profiles etc) against those from previous published studies for all grids and comment on the comparison.
3. Compute the flow at $Re = 1000$ and show that on a sufficiently fine mesh, your results match published results.

2 N-S Solver

2.1 Discretization of N-S Equation

The general steps solving the N-S equation is showing below:

- (I): Using the data from the current time step and the previous time step to get u^*, v^* .
Then use u^*, v^* to get U^*, V^* .
- (II): Using U^*, V^* , solving the pressure poisson equation (PPE), get p^{n+1} .
- (III): Using p^{n+1} , and u^*, v^* , get u^{n+1}, v^{n+1} .
Using p^{n+1} , and U^*, V^* , get U^{n+1}, V^{n+1} .

2.1.1 Step (I):

Using the the data we got from current time step and the previous time step get u^*, v^* .
Then use u^*, v^* get U^*, V^* .

Update u^*, v^* :

The first step is going to update u^*, v^* :

$$\frac{\vec{u}^* - \vec{u}^n}{\Delta t} = -[\frac{3}{2}\nabla \cdot (\vec{U}^n \vec{u}^n) - \frac{1}{2}\nabla \cdot (\vec{U}^{n-1} \vec{u}^{n-1})] + \frac{1}{Re} \nabla^2 \frac{\vec{u}^* + \vec{u}^n}{2} \quad (1)$$

which is:

$$\underbrace{[1 - \frac{\Delta t}{2Re} \nabla^2] \vec{u}^*}_{(1)} = -\frac{3}{2} \underbrace{[\Delta t \nabla \cdot (\vec{U}^n \vec{u}^n)]}_{(2)} + \frac{1}{2} \underbrace{[\Delta t \nabla \cdot (\vec{U}^{n-1} \vec{u}^{n-1})]}_{(3)} + \underbrace{[1 + \frac{\Delta t}{2Re} \nabla^2 \vec{u}]}_{(4)} \quad (2)$$

Could say, (2), (3) are convection term, and (4) is the diffusion term of the eqaution.
Could use this equation to update u^*, v^* .

Where,

$$\begin{aligned} (1) &= \left\{ 1 - \frac{\Delta t}{Re} \nabla^2 \right\} u^* = P^* - \left[\frac{E^* + W^* - 2P^*}{\Delta x^2} + \frac{N^* + S^* - 2P^*}{\Delta y^2} \right] \frac{\Delta t}{Re} \\ (2) &= \frac{\Delta t}{\Delta} \left((U_e^n \frac{E^n + P^n}{2} - U_w^n \frac{W^n + P^n}{2}) + (V_n^n \frac{N^n + P^n}{2} - V_s^n \frac{S^n + P^n}{2}) \right) \\ (3) &= \frac{\Delta t}{\Delta} \left((U_e^{n-1} \frac{E^{n-1} + P^{n-1}}{2} - U_w^{n-1} \frac{W^{n-1} + P^{n-1}}{2}) + (V_n^{n-1} \frac{N^{n-1} + P^{n-1}}{2} - V_s^{n-1} \frac{S^{n-1} + P^{n-1}}{2}) \right) \\ (4) &= P^n + \frac{\Delta t}{2Re\Delta^2} (E^n + W^n + N^n + S^n - 4P^n) \end{aligned}$$

Could found in this expression, the formula to calculate u^*, v^* is same, only input u or v in to the equation:

P means (i, j) , W means $(i - 1, j)$, E means $(i + 1, j)$, S means $(i, j - 1)$, N means $(i, j + 1)$.

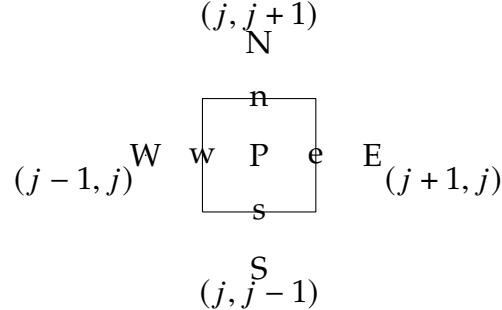


Figure 1: Illustration of a grid layout with directional points.

Put the result in, could get:

$$aW + bP + cE + eS + fN = d$$

The left hand side is the unknown characters, and the right hand side b is known from previous and current time step. Could see, this equation in our system means $Ax = d$, where A is consist of a, b, c, e, f , is a pentadiagonal matrix:

$$\begin{bmatrix} 1 & 0 & & 0 & & \\ a & b & c & & f & \\ & a & b & c & & f \\ & & a & b & c & 0 \\ 0 & & \ddots & \ddots & \ddots & \\ & e & & a & b & c \\ & e & & a & b & c \\ & 0 & & 0 & 1 & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ \vdots \\ d_{n-2} \\ d_{n-1} \\ d_{j-i} \end{bmatrix}$$

Solve this system, we could get u^* , and v^*

Update U^*, V^* :

$$U_e^* = \frac{u_p^* + u_E^*}{2}, V_n^* = \frac{v_p^* + v_N^*}{2}$$

2.1.2 Step (II):

Update p^{n+1} :

$$\nabla^2 p = \frac{\nabla \cdot \vec{u}^*}{\Delta t} = \frac{\frac{U_e^* - U_P^*}{\Delta x} + \frac{V_n^* - V_S^*}{\Delta y}}{\Delta t}$$

$$\frac{P_E + P_W - 2P_P}{\Delta x^2} + \frac{P_N + P_S - 2P_P}{\Delta y^2} = \frac{\frac{U_e^* - U_P^*}{\Delta x} + \frac{V_n^* - V_S^*}{\Delta y}}{\Delta t}$$

$$P_E + P_W + P_N + P_S - 4P_P = \frac{\Delta}{\Delta t} (U_e^* - U_P^* + V_n^* - V_s^*) = RHS$$

Similarly, this equation also give us a pentadiagonal system:

$$\begin{bmatrix} 1 & 0 & & 0 & & f & & \\ a & b & c & & & & f & \\ & a & b & c & & & & f \\ & & a & b & c & & & 0 \\ 0 & & & \ddots & \ddots & \ddots & & \\ e & & & & a & b & c & \\ & e & & & & a & b & c \\ & & 0 & & & 0 & 1 & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ \vdots \\ d_{n-2} \\ d_{n-1} \\ d_n \end{bmatrix}$$

2.1.3 Step (III):

Update u^{n+1}, v^{n+1} :

The equation to update the u^{n+1}, v^{n+1} is showing below:

$$\vec{u}^{n+1} = \vec{u}^* - \Delta t (\nabla p)_{cc}$$

where we could calculate the cell center pressure divergence $(\nabla p)_{cc}$, and:

$$\begin{cases} u^{n+1} = u^* - \Delta t \cdot \frac{P_E - P_W}{2\Delta x} \\ v^{n+1} = v^* - \Delta t \cdot \frac{P_N - P_S}{2\Delta y} \end{cases}$$

Update U^{n+1}, V^{n+1} :

The equation to update the U^{n+1}, V^{n+1} is showing below:

$$\vec{U}^{n+1} = \vec{U}^* - \Delta t (\nabla p)_{fc}$$

where we could calculate the face center pressure divergence $(\nabla p)_{fc}$, and:

$$\begin{cases} U_e^{n+1} = U_e^* - \Delta t \cdot \frac{P_E - P_P}{\Delta x} \\ V_n^{n+1} = V_n^* - \Delta t \cdot \frac{P_N - P_P}{\Delta y} \end{cases}$$

2.1.4 LineSOR implementation

In the discretization steps (I) and (II) of the Navier-Stokes equations, we obtain a pentadiagonal matrix system. However, our Tridiagonal Matrix Algorithm (TDMA) can only solve systems represented by tridiagonal matrices. To address this, we transfer the additional diagonal lines, e and f, to the right-hand side and continue iterating. This process is repeated until the results satisfy the original requirements of the pentadiagonal system.

The origional pentadiagonal system:

$$\begin{bmatrix} 1 & 0 & & 0 & & f & & \\ a & b & c & & & & f & \\ & a & b & c & & & & \\ & & a & b & c & & 0 & \\ 0 & & & \ddots & \ddots & \ddots & & \\ e & & & a & b & c & & \\ & e & & & a & b & c & \\ & & 0 & & 0 & 1 & & \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_{n-2} \\ u_{n-1} \\ u_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ \vdots \\ d_{n-2} \\ d_{n-1} \\ d_n \end{bmatrix}$$

For each line j, the system could be represented as:

$$[E \mid T \mid F] \begin{bmatrix} [u_{j-1}^{n+1}] \\ [u_j^{n+1}] \\ [u_{j+1}^{n+1}] \end{bmatrix} = [[d_j]]$$

Could say, E mans operating the previous line (j-1) to current line (j), and F means operating on the next line (j+1) to current line. Where $[E \mid T \mid F]$ is:

$$\left[\begin{array}{cccccc|cccccc|cccccc} 0 & 0 & \cdots & 0 & 0 & 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & e_2 & 0 & \cdots & 0 & 0 & a_2 & b_2 & c_2 & \ddots & \vdots & 0 & 0 & f_2 & 0 & \cdots & 0 & 0 \\ 0 & 0 & e_3 & \cdots & 0 & 0 & 0 & a_3 & \ddots & \ddots & 0 & 0 & 0 & 0 & f_3 & \cdots & 0 & 0 \\ 0 & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & c_2 & 0 & 0 & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & e_{n-1} & 0 & 0 & \ddots & \ddots & 0 & b_{n-1} & c_{n-1} & 0 & 0 & 0 & \cdots & f_{n-1} & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 \end{array} \right]$$

LineSOR:

For LineSOR, we move the two diagonal line (e and f) to the right hand side:

$$[0 \mid T \mid 0] \begin{bmatrix} [u_{j-1}^{k+1}] \\ [u_j^{k+1}] \\ [u_{j+1}^{k+1}] \end{bmatrix} = [[d_j]] - [E \mid 0 \mid F] \begin{bmatrix} [u_{j-1}^k] \\ [u_j^k] \\ [u_{j+1}^k] \end{bmatrix}$$

Which is:

$$[T] \begin{bmatrix} [u_j^{k+1}] \end{bmatrix} = [[d_j]] - [E] \begin{bmatrix} [u_{j-1}^k] \end{bmatrix} - [F] \begin{bmatrix} [u_{j+1}^k] \end{bmatrix}$$

Where we could use TDMA to solve this equation. As this result is different from directly solve the equation we previously got, we need it keep iterating, until the error small enough:

$$Error = [E \mid T \mid F] \begin{bmatrix} [u_{j-1}^{k+1}] \\ [u_j^{k+1}] \\ [u_{j+1}^{k+1}] \end{bmatrix} - [[d_j]] < 1e-6$$

2.2 Setting

2.2.1 Solver field

As in the discretization of N-S equation, we have u , v , p , and U , V fields. To handle boundary condition, we use ghost point. For convenient, we set all the fields in same size: $(N + 2) \cdot (N + 2)$, the u , v , p fields are:

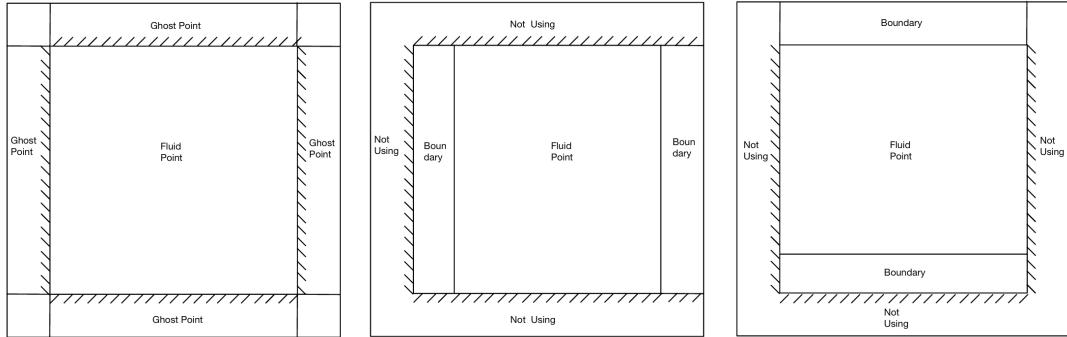


Figure 2: $u/v/p$, U , and V field

The left one is for u , v , and p field, where they are defined at cell center. It contains ghost point, which is in the solid part near to the boundary. U field is in the middle, which is 1 column, and 2 rows less than $u/v/p$ field. V field is the right one, where it is 1 row, and 2 column less than $u/v/p$ field.

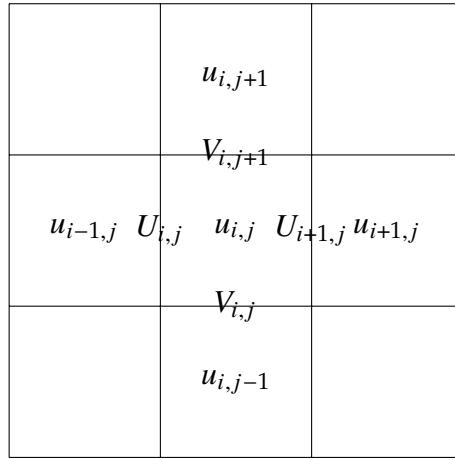


Figure 3: Illustration of a grid layout with $u/v/p$, U , and V .

2.2.2 Boundary condition and implementation

First, let us clarify boundary condition: Both 4 sides are in dirichlet boundary condition. Which means at the boundary point:

$$\frac{u_p + u_{gs}}{2} = u_B$$

2.3 Solver Architecture

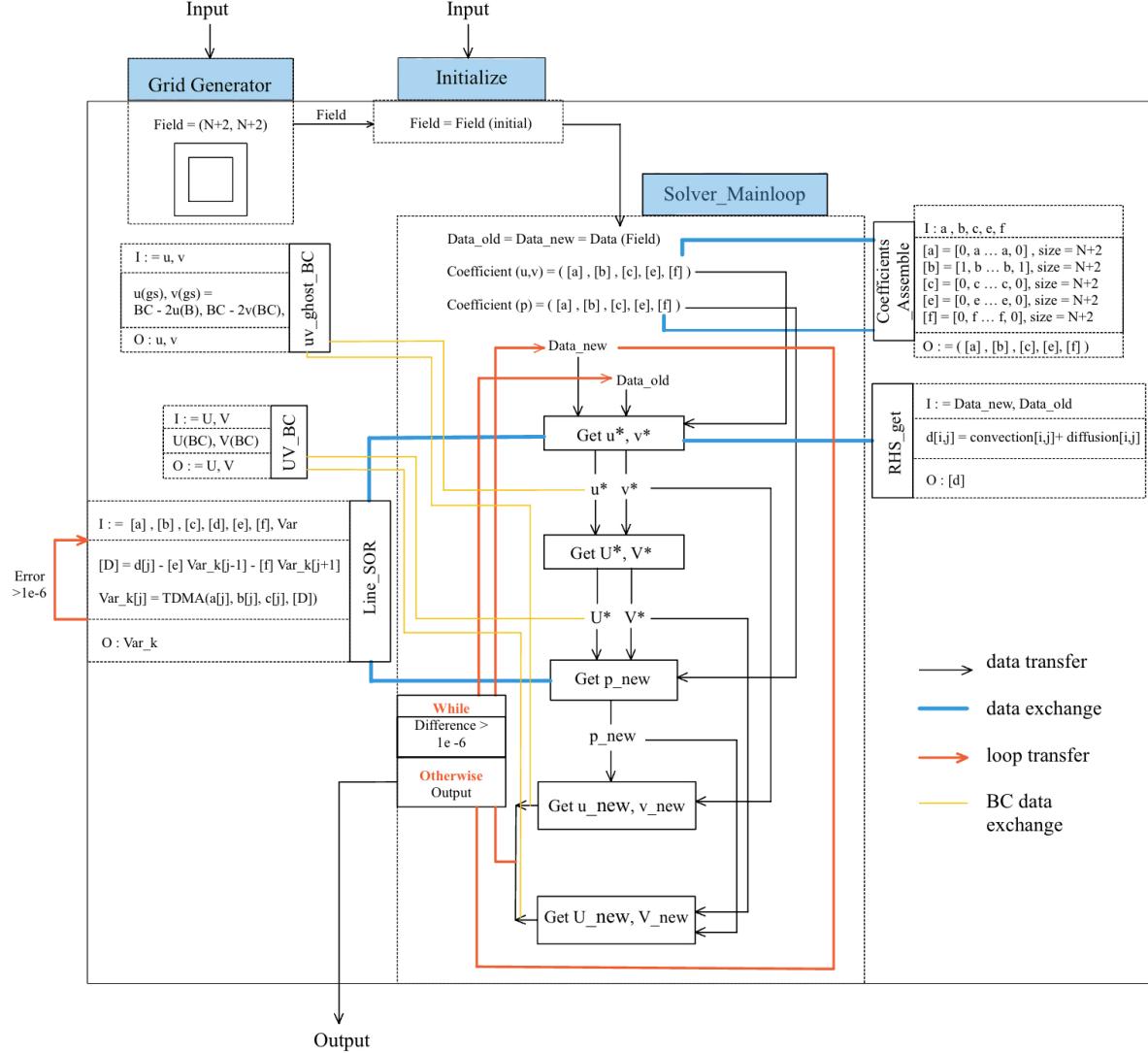


Figure 4: Solver Architecture

This diagram shows our Solver structure. By Using grid generator to generate fields for u, v, p , and U, V , transfer the data to initialize the initial value, then use the main loop to doing time iteration till the fields get to steady state, we could output the result.

- **I/O:** I means function input, and O means function output.
- **Data Transfer Lines:** The diagram uses different colored lines to indicate various types of data movement, while the lines with arrow means one-way data transfer, which means value assignment, and the lines without arrow means data exchange, where there is data input to function, but there is also data get back from function.

3 Result for Re=100

3.1 Stream lines at Re=100

For $Re=100$, we got the result after the flow reaches steady state, where we use the total difference of u , v , and p smaller than $1e-6$, we could get the streamlines results in different grid size showing below:

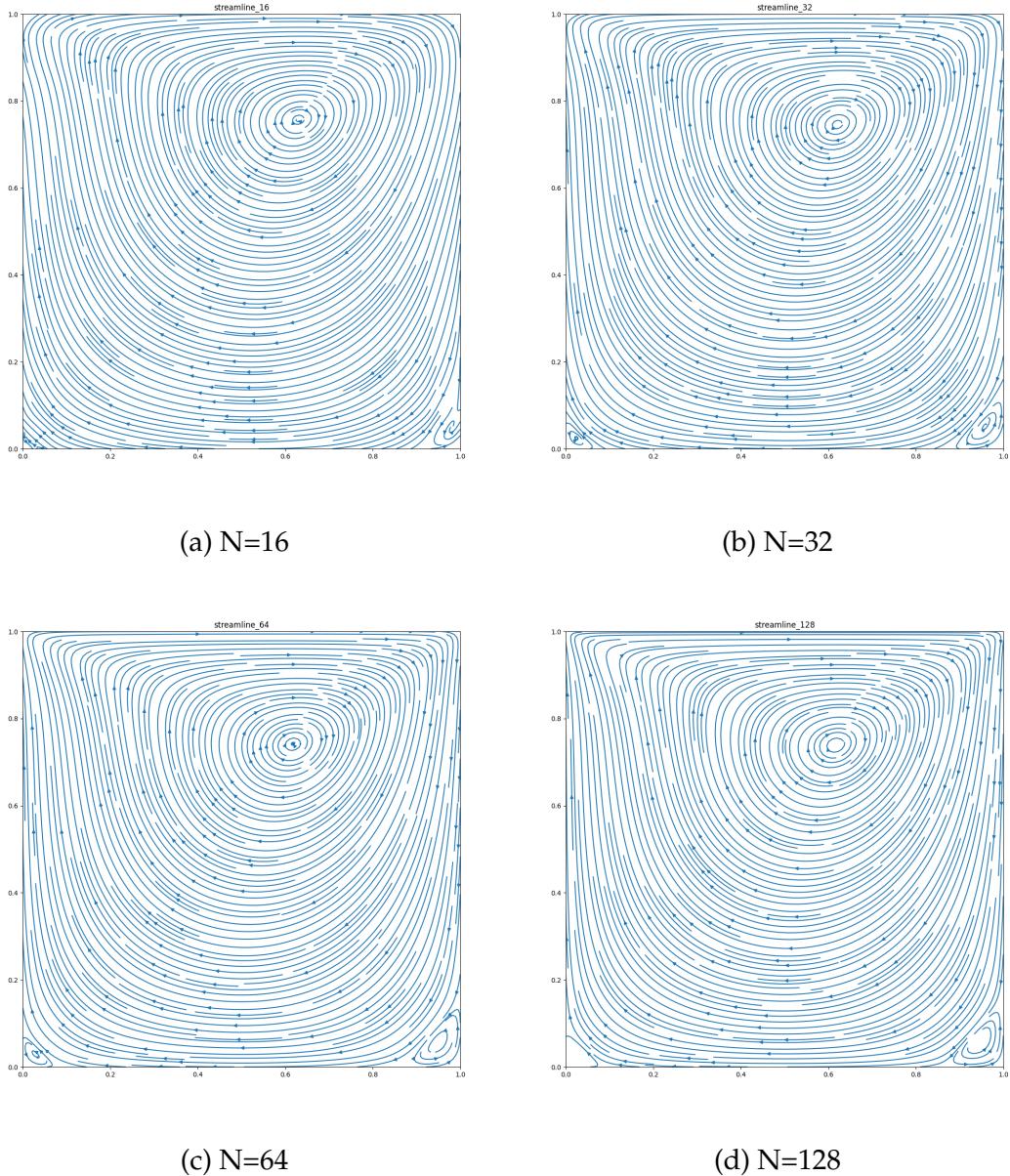


Figure 5: Stream line visualizations at different grid resolutions

For general view, we could see there is a big vortex appear at the biased center of the cavity, and the small vortex represent at the left bottom and a relatively larger vortex appear at the right bottom of the cavity.

3.1.1 Streamline comparison of different grid size result

Compare different grid size result, we could find that for N=16 grid, the vortex at left bottom is not showing, and the vortex on the right bottom is small. Where as the grid become finer, the vortex shows up and the vortex at the right bottom become larger.

3.2 Result comparison to the reference

To make our result is robust, we compare the result with the reference (Ghia et al, 1982)[2]. The results is showing below:

3.2.1 Streamline comparison

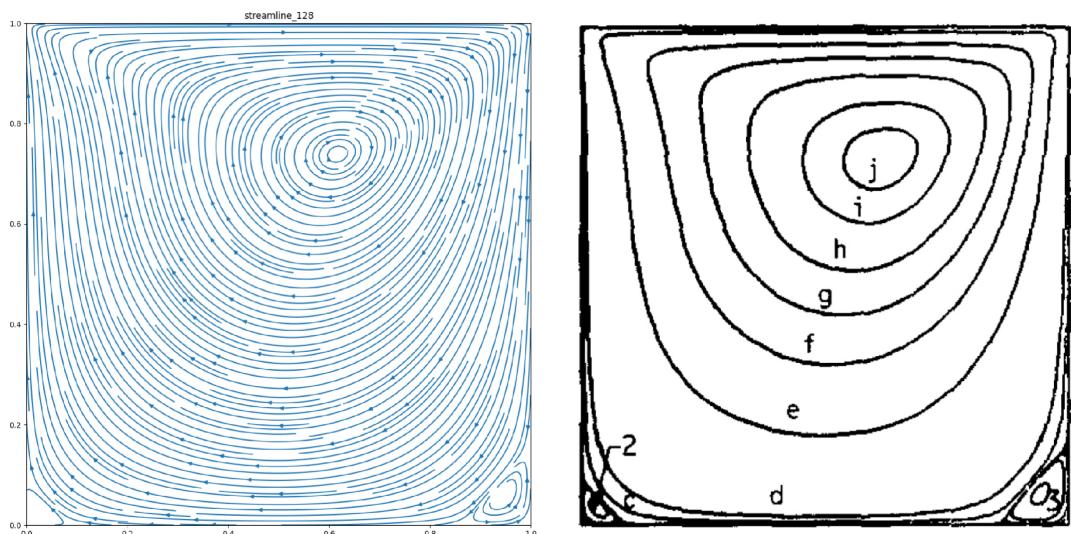


Figure 6: left: Result Streamline, right: Ghia(1982)[2]’s result

Compare out N=128 result to the Ghia (1982)[2] ’s result, we could find the results is similar, where the there shows three vortex, and the stream lines are similar.

3.2.2 Velocity at geometry center

Measure our result’s accuracy need us to compare the result with the reference (Ghia et al. 1982)[2]. By compare the result at geometry center, we could check the accuracy of our result.

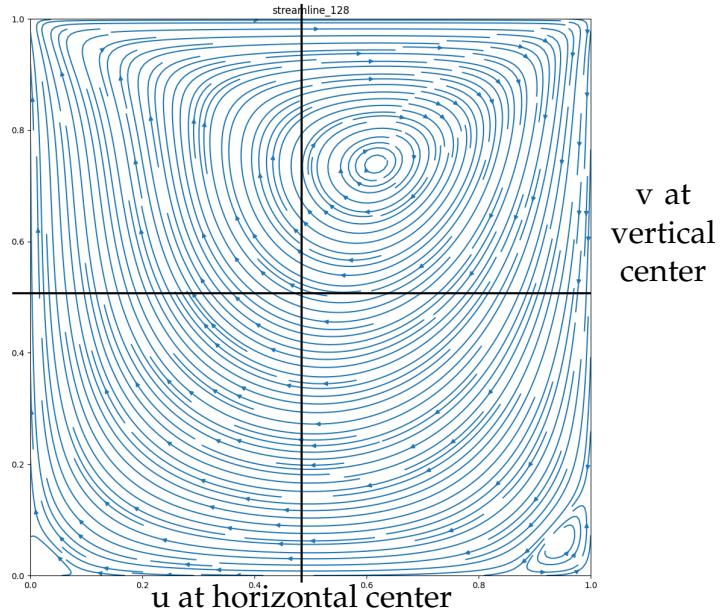


Figure 7: velocity at geometry center

We use the the horizontal velocity u at horizontal center (at same vertical line, middle at horizontal), and vertical velocity v at vertical center (at same horizontal line, middle at horizontal). The general result is showing below:

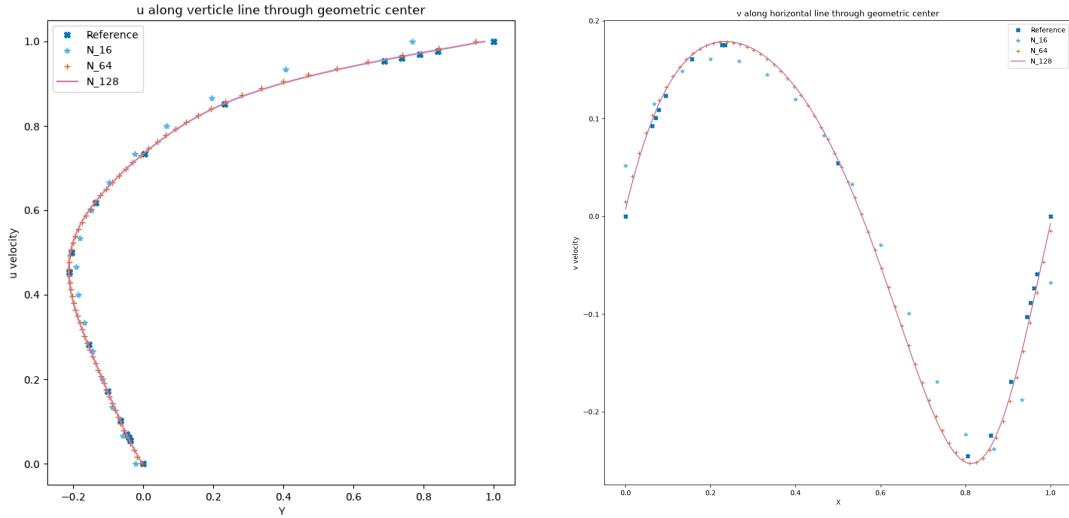


Figure 8: u along vertical line at middle of x (left), v along horizontal line at middle of y (right). Reference is Ghia (1982)[2]'s result. There is also enlarged figures showing down below.

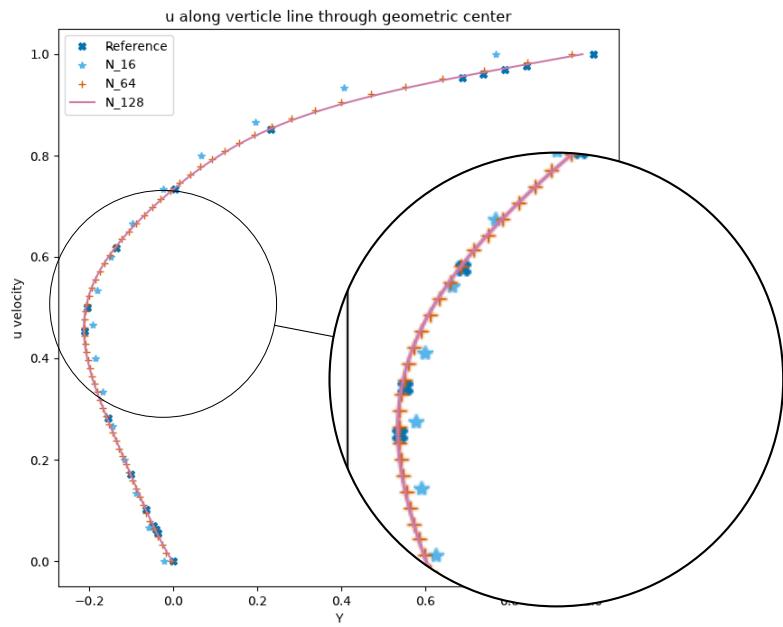


Figure 9: u along vertical line at middle of x (large)

This figure shows velocity u at the horizontal middle of the domain. Could say, in general the result is matching. Also compare different grid size, could find the coarsest grid, $N=16$, biased largest from reference result. Conversely, the finest grid, $N=128$, match pretty good.

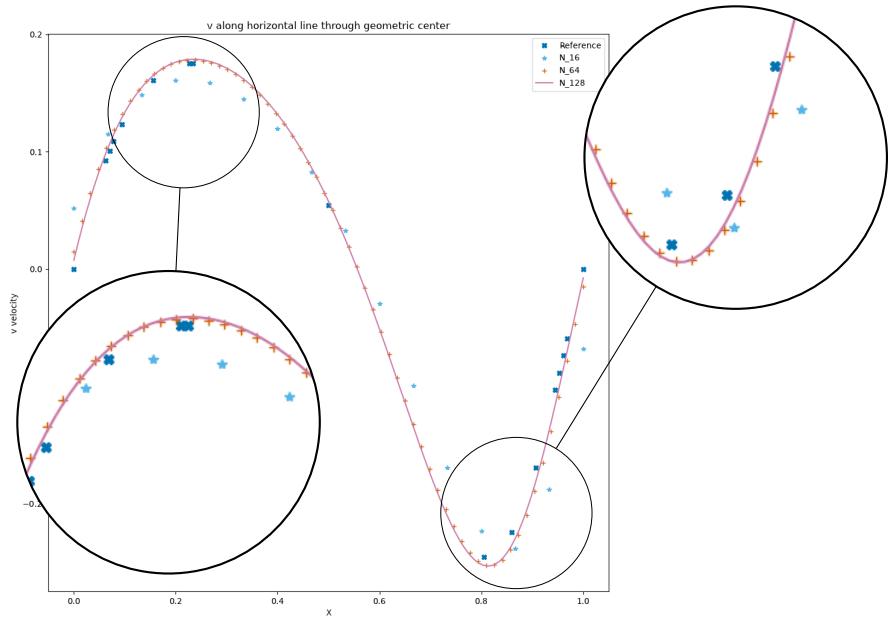


Figure 10: v along horizontal line at middle of y (large)

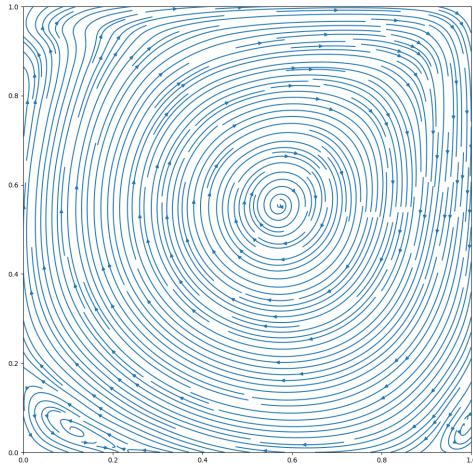
Could find in these solution, that the $N = 16$ grid is the most rough grid, where the difference with the paper result ($N=128$) is the largest. The finest grid ($N=128$) us the most accurate grid, where the result match the result from paper.

As the grid become finer, the result become more accurate.

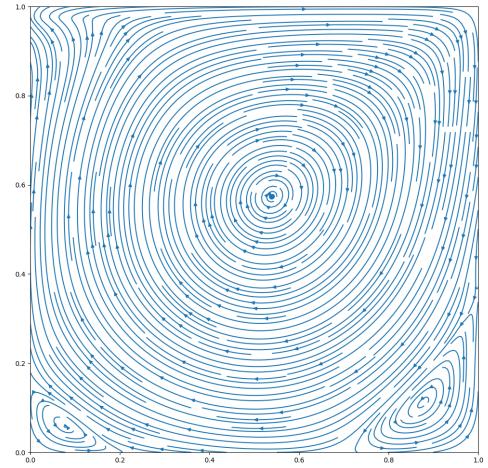
4 Result for $\text{Re}=1000$

4.1 Stream lines at $\text{Re}=1000$

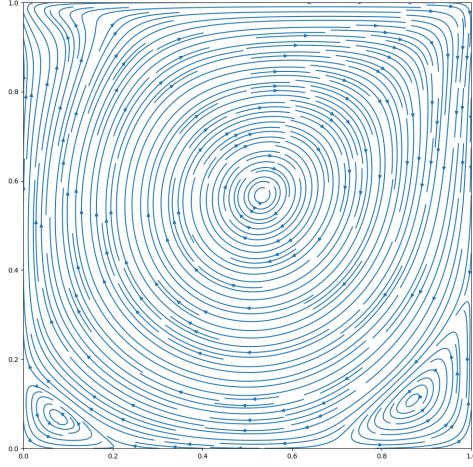
For $\text{Re}=1000$, the streamlines result is showing below:



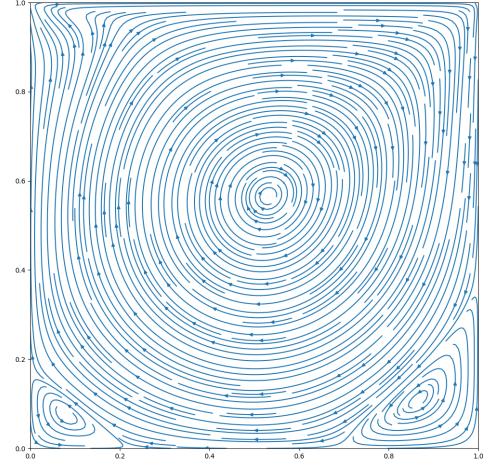
(a) $N=16$



(b) $N=32$



(c) $N=64$



(d) $N=128$

Figure 11: Stream line visualizations for $\text{Re}=1000$ at different grid resolutions

For $\text{Re}=1000$, could see the main vortex is still at the center, while the two small

voetex at the left bottom and right bottom is also present, but compare to the result at $Re=100$, could also find there is a small curvature at the top left, while it is showing more obvious at the finer grid.

4.2 Result comparison to the reference

4.2.1 Velocity at geometry center

Measure our result's accuracy need us to compare the result with the reference (Ghia et al. 1982)[2]. By compare the result at geometry center, we could check the accuracy of our result.

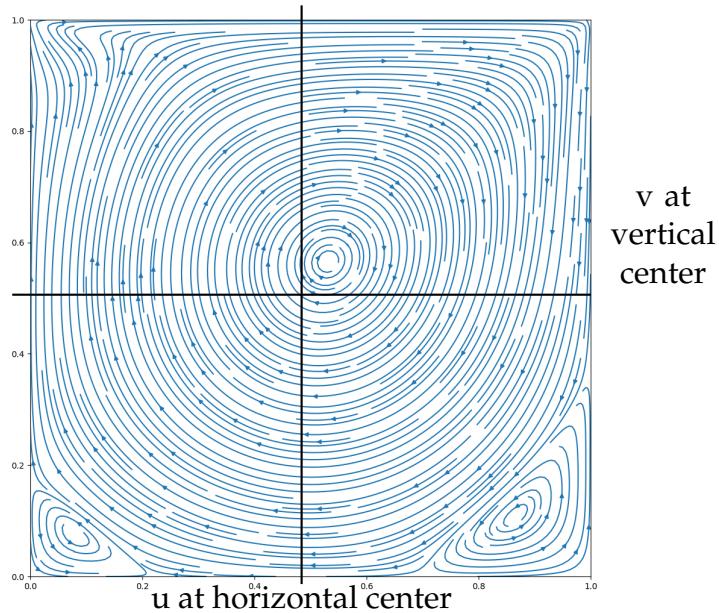


Figure 12: velocity at geometry center

At $Re=1000$, we use the the horizontal velocity u at horizontal center (at same vertical line, middle at horizontal), and vertical velocity v at vertical center (at same horizontal line, middle at horizontal). The general result is showing below:

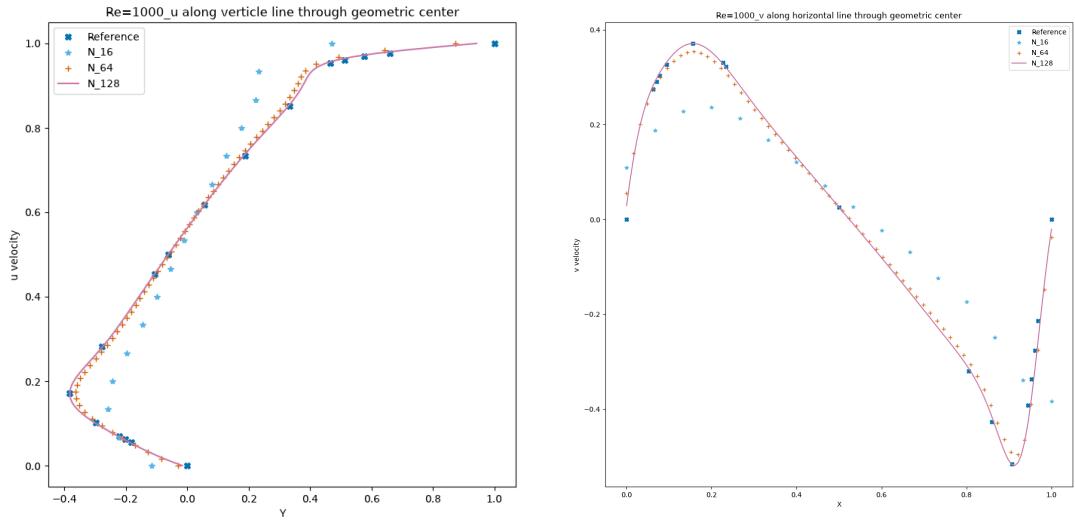


Figure 13: u along vertical line at middle of x (left), v along horizontal line at middle of y (right). There is also enlarged figures showing down below.

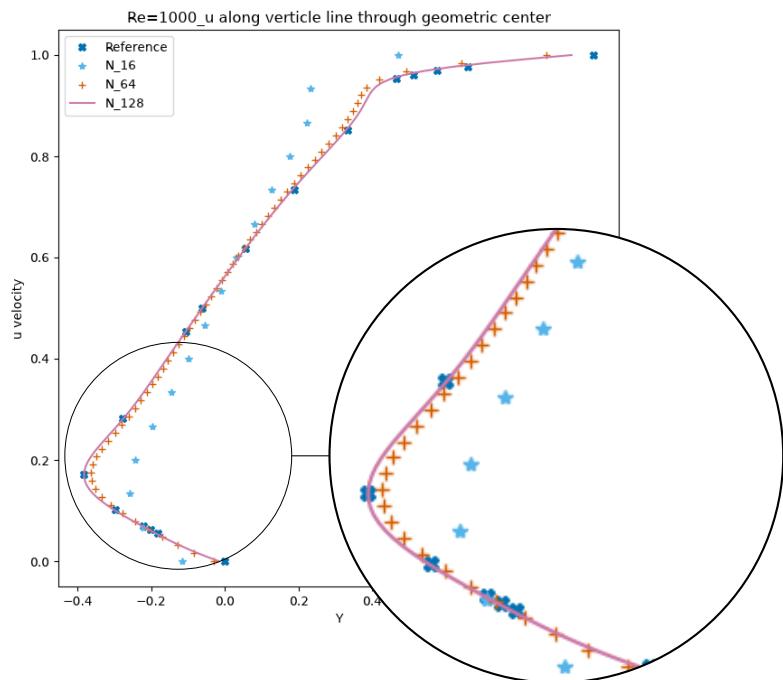


Figure 14: u along vertical line at middle of x (large)

This figure shows velocity u at the horizontal middle of the domain. Could say, in general the result is matching. Also compare different grid size, could find the coarse grid, $N=16$, biased largest from reference result. Conversely, the finest grid, $N=128$, match pretty good.

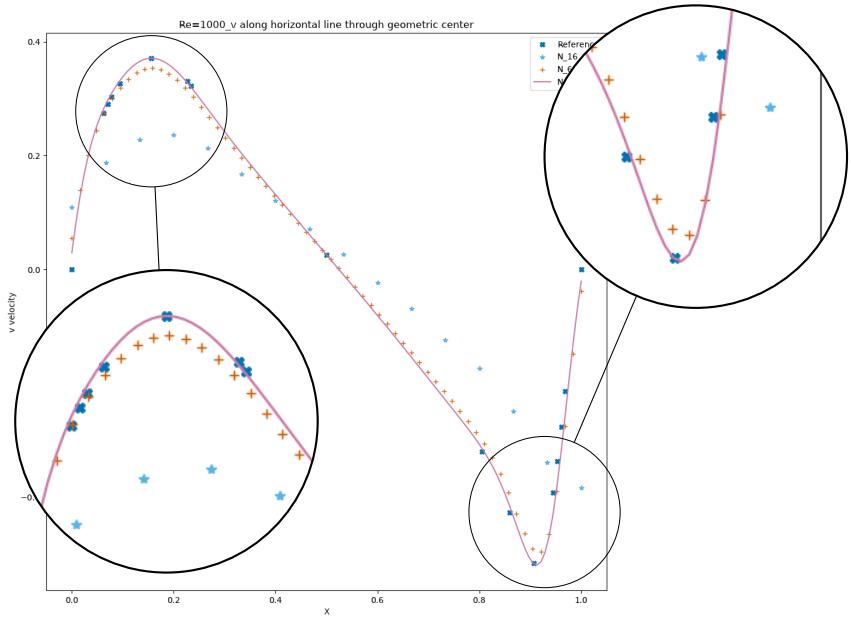


Figure 15: v along horizontal line at middle of y (large)

We also have very similar observation on the vertical velocity v on the vertical geometry center. Could find as the grid become finer, the result match better.

4.2.2 Streamline comparison

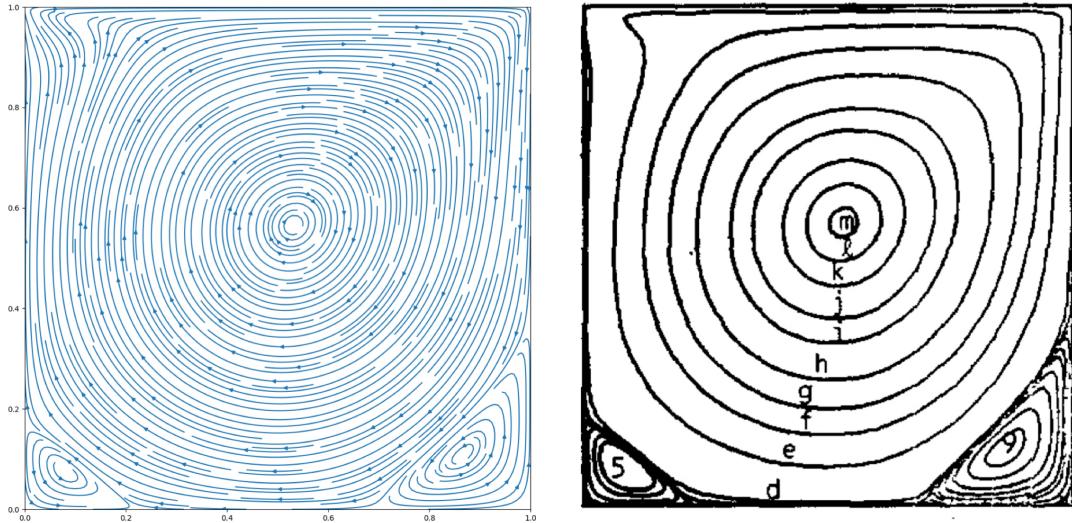


Figure 16: left: Result Streamline, right: Ghia(1982)[2]'s result

Could say the results are matching, where the vortex location and the shape are similar.

References

- [1] O. Botella and R. Peyret. Benchmark spectral results on the lid-driven cavity flow. *Computers Fluids*, 27(4):421–433, 1998.
- [2] U Ghia, K.N Ghia, and C.T 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.

Appendix

Our basic Solver code is showing below:

```
1 # This solver is a new attempt using same size of u, U... ,
2 # keep updating ghost point and dont need to change formula
3
4 import numpy as np
5 import math
6 import matplotlib.pyplot as plt
7
8
9 def TDMA(a, b, c, d): # TDMA Solver, input abcd in same length
10    # a[0], c[-1] not been used
11    do = d.copy()
12    ao = a.copy()
13    bo = b.copy()
14    co = c.copy()
15    N = len(d)
16    xo = np.zeros(N)
17    for rowi in range(1,N):
18        k = ao[rowi]/bo[rowi-1]
19        bo[rowi] -= co[rowi-1]*k
20        do[rowi] -= do[rowi-1]*k
21    xo[N-1] = do[N-1]/bo[N-1]
22    for rowi in range(N-2,-1,-1):
23        xo[rowi] = (do[rowi]-co[rowi]*xo[rowi+1])/bo[rowi]
24    return xo
25
26
27 def general_grid_generator(len_x, Nx, Ny):
28    # Creating same size of u, v, U, V
29    # where col 0, row 0, row -1, of U not used
30    # where row 0, col 0, col -1, of V not used
31    initial_Value = 0.01
32
33    dx = len_x/Nx
34    grid = np.full((Nx+2, Ny+2), initial_Value)
35    u,v,U,V = data_copy(grid, grid, grid, grid)
36    p = np.copy(grid)
37    return u, v, U, V, dx
38
39 def data_copy(u, v, U, V):
40    return np.copy(u), np.copy(v), np.copy(U), np.copy(V)
41
42 def Ghost_BC(u,v): # updating ghost point based on BC condition
43    # (u_indomain + u_ghost)/2 = Boundary_value
44    # u_ghost = 2 * Boundary_value - u_indomain
45    u[-1,:] = 2*u[-1,:] - u[-2,:]
46    v[-1,:] = -v[-2,:]
47
48    u[0, :] = -u[1,:]
49    v[0, :] = -v[1,:]
50
51    u[1:-1, 0] = -u[1:-1,1]
52    v[1:-1, 0] = -v[1:-1,1]
53
54    u[1:-1, -1] = -u[1:-1,-2]
```

```

55     v[1:-1, -1] = - v[1:-1,-2]
56     return u, v
57
58
59 def Ghost_BC_p(p): # updating ghost point for p based on BC condition
60     p[-1,:] = p[-2,:]
61     p[-1,:] = p[-2,:]
62
63     p[0, :] = p[1,:]
64     p[0, :] = p[1,:]
65
66     p[1:-1, 0] = p[1:-1,1]
67     p[1:-1, 0] = p[1:-1,1]
68
69     p[1:-1, -1] = p[1:-1,-2]
70     p[1:-1, -1] = p[1:-1,-2]
71     return p
72
73
74
75 def UV_BC(U, V):
76     U[1:-1,1] = 0
77     U[1:-1,-1] = 0
78     V[1,1:-1] = 0
79     V[-1,1:-1] = 0
80     return U,V
81
82
83
84 def coeff_assemble(rows_cols, a_e, b_e, c_e, e_e, f_e):
85     rows, cols = rows_cols[0], rows_cols[1]
86     # input elements of abcdef, get line matrix of abcdef
87     a = np.full((rows, cols), a_e) # abc in same length of Ncols of u
88     b = np.full((rows, cols), b_e)
89     c = np.full((rows, cols), c_e)
90
91     e = np.full((rows, cols), e_e) # ef in same length of Nrows of u
92     f = np.full((rows, cols), f_e)
93     a[:,0]= a[:, -1] = c[:,0] = c[:, -1] = 0
94     b[:,0] = b[:, -1] = 1
95     e[:, -1] = e[:, 0] = f[:, -1] = f[:, 0] = 0
96     return a,b,c, e,f
97
98
99
100 def RHS_uv(f, f_old, U, V, U_old, V_old, c0, r):
101     # Update Right Hand Side
102     f_P = f[1:-1,1:-1]
103     f_W = f[1:-1,:-2]
104     f_E = f[1:-1,2:]
105     f_S = f[:-2,1:-1]
106     f_N = f[2:,1:-1]
107
108     f_P_old = f_old[1:-1,1:-1]
109     f_W_old = f_old[1:-1,:-2]
110     f_E_old = f_old[1:-1,2:]
111     f_S_old = f_old[:-2,1:-1]
112     f_N_old = f_old[2:,1:-1]

```

```

113
114 convection_new = c0*( U[1:-1,2:] * (f_E + f_P)/2 -
115                         U[1:-1,1:-1] * (f_W + f_P)/2 +
116                         V[2:,1:-1] * (f_N + f_P)/2 -
117                         V[1:-1,1:-1] * (f_S + f_P)/2 )
118
119 convection_old = c0*( U_old[1:-1,2:] * (f_E_old + f_P_old)/2 -
120                         U_old[1:-1,1:-1] * (f_W_old + f_P_old)/2 +
121                         V_old[2:,1:-1] * (f_N_old + f_P_old)/2 -
122                         V_old[1:-1,1:-1] * (f_S_old + f_P_old)/2 )
123
124 diffusion = f_P + r*(f_E + f_W + f_N + f_S -4*f_P)
125 return -3/2*convection_new + 1/2*convection_old + diffusion
126
127 def Not_Use_UV_BC(U,V):
128     U[0,:] = U[-1,:] = U[:,0] = V[:,0] = V[:, -1] = V[0,:] = -10000
129     return U,V
130
131 def Residual(u, a, b, c, d, e, f):
132     Res = np.zeros_like(u)
133     Res[1:-1,1:-1] = (a[1:-1,1:-1] * u[1:-1,:-2] +
134                         b[1:-1,1:-1] * u[1:-1,1:-1] +
135                         c[1:-1,1:-1] * u[1:-1,2:] +
136                         e[1:-1,1:-1] * u[:-2,1:-1] +
137                         f[1:-1,1:-1] * u[2:,1:-1] -
138                         d[:, :])
139     # print(Res)
140     return Res
141
142
143 def Point_GS(a,b,c,d,e,f,u):
144     u_k = np.copy(u)
145     u_k_new = np.copy(u)
146     n_rows = np.size(u,0)
147     w = 1.9
148     D = np.zeros_like(a[1,:])
149     Res = 100
150     k = 0
151     while Res>1e-6:
152         u_k = np.copy(u_k_new)
153         u_k_new[1:-1,1:-1] = (-a[1:-1,1:-1] * u_k_new[1:-1,:-2] +
154                             -c[1:-1,1:-1] * u_k[1:-1,2:] +
155                             -e[1:-1,1:-1] * u_k_new[:-2,1:-1] +
156                             -f[1:-1,1:-1] * u_k[2:,1:-1] +
157                             d[:, :])/b[1:-1,1:-1]
158         u_k_new[1:-1] = u[1:-1]*(1-w) + u_k_new[1:-1]*w # SOR term
159
160         Res = np.linalg.norm(Residual(u_k_new, a, b, c, d, e, f))
161         # print("Res=")
162         # print(Res)
163         if Res > 1e+200:
164             print("NOT CONVERGE NOT CONVERGE NOT CONVERGE NOT CONVERGE")
165             print(Res)
166             quit()
167             u = np.copy(u_k)
168             k+=1
169             # print("finish iteration")
170

```

```

171     return u_k_new, k
172
173
174
175 def LineSOR(a,b,c,d,e,f, u):
176     # input abcdef, and u, get D
177     # then use TDMA get new u
178     # abc is in same size of N_rows of u, ef in same size of N_cols of u
179     # ! Input d is LIKE inner field of u, need to cut each line to use !
180     u_k = np.copy(u)
181     u_k_new = np.copy(u)
182     n_rows = np.size(u,0)
183     w = 1.9
184     D = np.zeros_like(a[1,:])
185     Res = 100
186     while Res>1e-6:
187         u_k = np.copy(u_k_new)
188         for j in range(1,n_rows-1): # the number is based on d, which is (N-1) x (N-1)
189             # D = np.zeros_like(a[1,:])
190             D[1:-1] = d[j-1,:] - e[j,1:-1]*u_k_new[j-1,1:-1] - f[j,1:-1]*u_k[j+1,1:-1]
191             D[0], D[-1] = u[j,0], u[j,-1] # ghost point = ghost point
192             u_k_new[j,:] = TDMA(a[j,:],b[j,:],c[j,:],D) # TDMA solve uk
each line
193             # u_k_new[j,:] = u[j,:]*(1-w) + TDMA(a,b,c,D)*w # SOR term
194
195     Res = np.linalg.norm(Residual(u_k_new, a, b, c, d, e, f))
196     # print("Res=")
197     # print(Res)
198     if Res > 1e+20:
199         print("NOT CONVERGE NOT CONVERGE NOT CONVERGE NOT CONVERGE")
200         quit()
201     u = np.copy(u_k)
202     # print("finish iteration")
203     return u_k_new
204
205
206
207
208 def Solver(u, v, U, V, p, t, dt, dx, Re):
209     r = dt/(2*Re*dx**2)
210     c0 = dt/(dx)
211
212     rows_cols = np.shape(u)
213     a_uv,b_uv,c_uv,e_uv,f_uv = coeff_assemble(rows_cols, -r, 1+4*r, -r, -r, -r)
214     a_p,b_p,c_p,e_p,f_p = coeff_assemble(rows_cols, 1, -4, 1, 1, 1)
215
216     U, V = UV_BC(U, V)
217     u, v = Ghost_BC(u,v)
218
219     U, V = Not_Use_UV_BC(U, V)
220
221     u_star, v_star, U_star, V_star = data_copy(u, v, U, V)
222     u_new, v_new, U_new, V_new = data_copy(u, v, U, V)
223     u_old, v_old, U_old, V_old = data_copy(u, v, U, V)
224

```

```

225
226 # n_max = int(t/dt)
227 Difference = 100
228
229 n=0
230 while Difference>1e-6:
231     U, V = UV_BC(U, V) # Setup BC
232
233     # Step 1: get u_star, v_star, U_star, V_star
234     u, v = Ghost_BC(u,v)
235     u_star, v_star = Ghost_BC(u_star,v_star)
236
237     # get u_star:
238     d = RHS_uv(u, u_old, U, V, U_old, V_old, c0, r)
239     u_star = LineSOR(a_uv,b_uv,c_uv, d , e_uv,f_uv, u)
240
241
242     u_star, v_star = Ghost_BC(u_star,v_star)
243
244     # get v_star:
245     d = RHS_uv(v, v_old, U, V, U_old, V_old, c0, r)
246     v_star = LineSOR(a_uv,b_uv,c_uv, d , e_uv,f_uv, v)
247
248     u_star, v_star = Ghost_BC(u_star,v_star)
249
250     # get U_star, V_star
251     U_star[1:-1,2:] = (u_star[1:-1,1:-1] + u_star[1:-1,2:])/2
252     V_star[2:,1:-1] = (v_star[2:,1:-1] + v_star[1:-1,1:-1])/2
253
254     U_star, V_star = UV_BC(U_star, V_star) # Setup BC
255
256
257     # Step 2: get New Pressure P
258     d = dx/dt * (
259         U_star[1:-1, 2:] - U_star[1:-1, 1:-1] +
260         V_star[2:, 1:-1] - V_star[1:-1, 1:-1]
261     )
262     p_new,k = Point_GS(a_p,b_p,c_p,d, e_p,f_p, p)
263     p_new = Ghost_BC_p(p_new)
264
265
266     # Step 3: get u_new, v_new, U_new , V_new
267     # update u_new, v_new
268     u_new[1:-1,1:-1] = u_star[1:-1, 1:-1] - dt/dx*(p_new[1:-1,2:] -
269     p_new[1:-1,:-2])/2
270     v_new[1:-1,1:-1] = v_star[1:-1, 1:-1] - dt/dx*(p_new[2:,1:-1] -
271     p_new[:2,1:-1])/2
272     u_new, v_new = Ghost_BC(u_new,v_new)
273     # update U_new , V_new
274     U_new[1:-1,2:] = U_star[1:-1,2:] - dt/dx*(p_new[1:-1,2:]-p_new
275     [1:-1,1:-1])
276     V_new[2:,1:-1] = V_star[2:,1:-1] - dt/dx*(p_new[2:,1:-1]-p_new
277     [1:-1,1:-1])
278     U_new, V_new = UV_BC(U_new, V_new) # Setup BC
279
280     # print('new done')
281
282     u_old, v_old, U_old, V_old = data_copy(u, v, U, V)

```

```

279     u, v, U, V = data_copy(u_new, v_new, U_new, V_new)
280     N = rows_cols[1]-2
281     Difference = np.sum(np.abs(u - u_old)+np.abs(v - v_old)+np.abs(
282     p_new - p))/N**2
283
284     p = np.copy(p_new)
285
286     # print(Big_Res)
287     n +=1
288     print("time=% .3f, Difference=% .9f, p iteration times=%d" % ((n+1)*dt,
289     Difference,k))
290
291     return u, v, p
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334 def save(u,v,p,label):

```

```

335     with open(f'output_N={label}.txt', 'w') as f:
336         f.write(f'For grid size N= {label}:\n')
337         f.write('Array u:\n')
338         f.write(','.join(map(str, u)) + '\n\n')
339         f.write('Array v:\n')
340         f.write(','.join(map(str, v)) + '\n\n')
341         f.write('Array p:\n')
342         f.write(','.join(map(str, p)) + '\n')
343
344
345
346 def main():
347     N = 32
348     t = 20
349
350
351
352     dt = 0.02
353
354     len_x = 1
355     len_y = 1
356     Nx = Ny = N
357     Re = 100
358
359     u, v, U, V, p, dx = general_grid_generator(len_x, Nx, Ny)
360
361     u,v, p =Solver(u, v, U, V, p, t, dt, dx, Re)
362
363     print(u)
364     print(v)
365     print(p)
366
367     save(u,v,p,N)
368
369
370
371     draw_streamline(u[1:-1,1:-1],v[1:-1,1:-1], len_x, len_y , N)
372     draw_velocity_contour(u[1:-1,1:-1],v[1:-1,1:-1], len_x, len_y , N)
373     draw_u_contour(u[1:-1,1:-1],v[1:-1,1:-1], len_x, len_y , N)
374
375
376
377
378 if __name__ == '__main__':
379     main()

```

Listing 1: Lid Driven Cavity Solver

```

1 import numpy as np
2 import re
3 import matplotlib.pyplot as plt
4
5 def load_2d_arrays_with_numpy(label):
6     filename = f'output_N={label}.txt'
7     try:
8         with open(filename, 'r') as file:
9             content = file.read()
10    except FileNotFoundError:
11        print("")

```

```

12         return None, None, None
13
14     #
15     def extract_2d_array(array_content):
16         array_data = []
17         #
18         matches = re.findall(r'\[(.*?)\]', array_content.replace('\n', ''),
19         re.DOTALL)
20         for match in matches:
21             #
22             formatted_match = ' '.join(match.split())
23             numbers = np.fromstring(formatted_match, sep=' ')
24             array_data.append(numbers)
25         return np.array(array_data)
26
27     #
28     u_data = re.search(r'Array u:\s*((?:\[.*?\]\[, \s]*)+)', content,
29     re.DOTALL)
29     v_data = re.search(r'Array v:\s*((?:\[.*?\]\[, \s]*)+)', content,
29     re.DOTALL)
30     p_data = re.search(r'Array p:\s*((?:\[.*?\]\[, \s]*)+)', content,
30     re.DOTALL)
31
32     if not (u_data and v_data and p_data):
33         print("          u          , v,
34                           p")"
35     return np.array([]), np.array([]), np.array([])
36
37     u = extract_2d_array(u_data.group(1))
38     v = extract_2d_array(v_data.group(1))
39     p = extract_2d_array(p_data.group(1))
40
41     return u, v, p
42
43 def smallluvp(u,v,p):
44     return u[1:-1,1:-1],v[1:-1,1:-1],p[1:-1,1:-1]
45
46 def Center(u,v,p, label):
47     u,v,p = smallluvp(u,v,p)
48
49     center = int(label/2)
50     u_x_center = (u[:,center] + u[:,center-1])/2
51     v_x_center = (v[:,center] + v[:,center-1])/2
52     p_x_center = (p[:,center] + p[:,center-1])/2
53
54     u_y_center = (u[center,:] + u[center-1,:])/2
55     v_y_center = (v[center,:] + v[center-1,:])/2
56     p_y_center = (p[center,:] + p[center-1,:])/2
57
58     return u_x_center, v_x_center, p_x_center\
59     ,u_y_center, v_y_center, p_y_center
60
61
62 def get_ue_x_c():
63     ye_0 = [1.00000, 0.9766, 0.9688, 0.9609, 0.9531, 0.8516, 0.7344,

```

```

0.6172, 0.5000, 0.4531, 0.2813, 0.1719, 0.1016, 0.0703, 0.0625, 0.0547,
0.0000]
65 ue_x_c_0 = [1.00000, 0.84123, 0.78871, 0.73722, 0.68717, 0.23151,
0.00332, -0.13641, -0.20581, -0.21090, -0.15662, -0.10150, -0.06434,
-0.04775, -0.04192, -0.03717, 0.00000]
66 return ye_0[::-1], ue_x_c_0[::-1]
67
68 def get_ue_y_c():
69     xe_0 = [1.0000, 0.9688, 0.9609, 0.9531, 0.9453, 0.9063, 0.8594, 0.8047,
0.5000, 0.2344, 0.2266, 0.1563, 0.0938, 0.0781, 0.0703, 0.0625, 0.0000]
70     xe_y_c_0 = [0.00000, -0.05906, -0.07391, -0.08864, -0.10313, -0.16914,
-0.22445, -0.24533, 0.05454, 0.17527, 0.17507, 0.16077, 0.12317,
0.10890, 0.10091, 0.09233, 0.00000]
71     return xe_0[::-1], xe_y_c_0[::-1]
72
73
74 # def drawing(u_x_c_16, u_x_c_64, u_x_c_128):
75 #     ye, ue_x_c = get_ue_x_c()
76 #     y = np.linspace(0,N*1/N, N)
77 #     plt.figure(figsize=(8, 8))
78 #     plt.plot(ye, ue_x_c, label='Paper Result')
79 #     plt.plot(y, u_x_c, label='Result')
80 #     plt.xlabel('X')
81 #     plt.ylabel('u velocity')
82 #     plt.legend()
83 #     # plt.title('u Contour')
84 #     # plt.savefig(f'u_contour_{label}.png')
85 #     plt.show()
86
87
88 def drawing_x_c(u_x_c_16, u_x_c_64, u_x_c_128):
89     ye, ue_x_c_vertical = get_ue_x_c()
90     y_16 = np.linspace(0,1, 16)
91     y_64 = np.linspace(0,1, 64)
92     y_128 = np.linspace(0,1, 128)
93     plt.figure(figsize=(8, 8))
94     plt.plot( ue_x_c_vertical, ye, '-x', label='Paper Result', color= 'red'
)
95     plt.plot( u_x_c_16,y_16, '-x', label='Result_16',color= 'blue')
96     plt.plot( u_x_c_64,y_64, '-x', label='Result_64',color= 'green')
97     plt.plot( u_x_c_128,y_128, '--', label='Result_128',color= 'navy')
98     plt.xlabel('Y')
99     plt.ylabel('u velocity')
100    plt.legend()
101    plt.title('u along verticle line through geometric center')
102    plt.savefig('u along verticle line through geometric center')
103    # plt.show()
104
105
106
107
108
109 def drawing_y_c(v_y_c_16, v_y_c_64, v_y_c_128):
110     xe, ue_y_c_vertical = get_ue_y_c()
111     plt.figure(figsize=(12,12))
112     x_16 = np.linspace(0,16*1/16, 16)
113     x_64 = np.linspace(0,64*1/64, 64)
114     x_128 = np.linspace(0,128*1/128, 128)

```

```

115
116 plt.plot(xe, ue_y_c_vertical, '-x', label='Paper Result', color= 'red')
117 plt.plot(x_16, v_y_c_16, '-x', label='Result_16',color= 'blue')
118 plt.plot(x_64, v_y_c_64, '-x', label='Result_64',color= 'green')
119 plt.plot(x_128, v_y_c_128, '--', label='Result_128',color= 'navy')
120 plt.xlabel('X')
121 plt.ylabel('v velocity')
122 plt.legend()
123 plt.title('v along horizontal line through geometric center')
124 plt.savefig('v along horizontal line through geometric center')
125 # plt.show()
126
127
128 def center_get(N):
129     u, v, p = load_2d_arrays_with_numpy(N)
130     return Center(u,v,p, N)
131
132
133
134
135
136
137
138
139 def main():
140     #
141     # N = 128
142     # u, v, p = load_2d_arrays_with_numpy(N)
143     # print(u[-2,:])
144     # u_x_c, v_x_c, p_x_c, u_y_c, v_y_c, p_y_c = Center(u,v,p, N)
145
146     u_x_c_128, v_x_c_128, p_x_c_128, u_y_c_128, v_y_c_128, p_y_c_128 =
147     center_get(128)
148     u_x_c_64, v_x_c_64, p_x_c_64, u_y_c_64, v_y_c_64, p_y_c_64 = center_get
149     (64)
150     u_x_c_16, v_x_c_16, p_x_c_16, u_y_c_16, v_y_c_16, p_y_c_16 = center_get
151     (16)
152
153
154
155 if __name__ == '__main__':
156     main()

```

Listing 2: Post Operator: