Numerically Solving the 2-D Navier-Stokes Equations

Ashvat Bansal
University of Illinois at Urbana-Champaign, IL, 61820, USA

In this report, a CFD solver to solve the 2-D incompressible Navier-Stokes Equations was developed. The code was then tested using the standard benchmark test of a flow driven cavity problem. The code implemented was able to simulate the flow accurately,and produce good results when plotted. The results were compared to a commercially available solver's outputs, and it was found that the velocity and pressure plots matched up well when compared to the commercial CFD solver.

I. Nomenclature

$\mathbf{u}$	=	Net Velocity [m/s]
u	=	x-component of Velocity [m/s]
v	=	y-component of Velocity [m/s]
t	=	Time [s]
ρ	=	Density [kg/m^2]
p	=	Pressure [Pa]
ν	=	kinematic Viscosity [m^2/s]
x	=	x Domain [m]
Δx	=	Length of Computational Cell [m]
y	=	y Domain [m]
Δy	=	Height of Computational Cell [m]
u^*	=	Intermediate x-Velocity [m/s]
v^*	=	Intermediate y-Velocity [m/s]
i	=	Index in x direction
j	=	Index in y Direction
$\mathbf{L}$	=	Laplacian Operator
$\mathbf{R}$	=	Right Side of Poisson Equation

II. Introduction

Navier-Stokes Equations, named after *Claude-Louis Navier* and *George Gabriel Stokes*, is a partial differential equation that describes the flow and motion of fluids. This equation is a generalization of the equation devised by the mathematician *Leonhard Euler* in the 18$^{\text{th}}$ century, where Euler's equation described the flow of incompressible and frictionless fluids. The element of viscosity was added by Navier in the year 1821 to make the problem more realistic. Throughout the 19th Century, British mathematician and physicist

Sir George Gabriel Stokes improved the work of Navier and Euler, though complete solutions were only obtained for simple two dimensional flow cases, while the flow can be approximated for three dimensions using numerical analysis methods.

The objective of this report is to develop a CFD solver that solves the Navier-Stokes Equations in 2 dimensions, which will later be tested. The development of the CFD solver is discussed in section III, and the results will be compared to analyzed and compared to existing solvers in section IV.

III. Developing the CFD Solver

A. Governing Equations

The Navier-Stokes Equation describe the motion and flow of fluid substances. These equations arise from applying Newton's Second Law to fluid motion, with the assumption that stress is the sum of a diffusing viscous term (proportional to the gradient of velocity) and a pressure term. The incompressible Navier-Stokes Equation for momentum can be written as

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \boldsymbol{u} \tag{1}$$

where $\boldsymbol{u} = [u, v]$ is the velocity vector in 2 dimensions, t is time, ρ is the density, p is the pressure and ν is the kinematic viscosity. The continuity equation for Navier-Stokes can be written as

$$\nabla \cdot \boldsymbol{u} = 0 \tag{2}$$

The continuity equation ensures that the rate of mass flow remains constant, ensuring incompressibility.

B. Computational Grid

The governing equations for the Navier-Stokes are solved using a computational grid/mesh. The grid used to solve the problem has cells of width Δx and height of Δy. The grid divides the computational space into $n_x \times n_y$ cells, where n_x are the number of cells in the horizontal direction, and n_y are the number of cells in the vertical direction.

A staggered grid was used to store the variables of pressures and velocities. The center of the grid cells stored the pressure values while the faces of the grid cells stored the velocity values. Due to its spatial relationship between pressure and velocities, this method was the preferred one.

Given below is a snippet of code that creates the desired mesh grid

```
% Index Extends
i_min    =   2;                          % Min index in i-direction
i_max    =   i_min + nx - 1;             % Max index in i-direction
j_min    =   2;                          % Min index in j-direction
j_max    =   j_min + ny - 1;             % Max index in j-direction

% Create Mesh Grid
x(i_min:i_max+1)  =   linspace(0, Lx, nx + 1);                    % Left faces
```

```
y(j_min:j_max+1)  =   linspace(0, Ly, ny + 1);                    % Bottom faces
xm(i_min:i_max)   =   0.5*(x(i_min:i_max) + x(i_min+1:i_max+1));  % i cell middle
ym(j_min:j_max)   =   0.5*(y(j_min:j_max) + y(j_min+1:j_max+1));  % j cell middle

% Create Mesh Sizes
dx      =   x(i_min+1) - x(i_min);          % Length of 1 cell
dy      =   y(j_min+1) - y(j_min);          % Height of 1 cell
dxi     =   1/dx;
dyi     =   1/dy;
```

Vectors (Arrays) are created to keep track of all the necessary locations on the grid surface. Standard index notation is used to refer to the x and y directions i.e, index i refers to x direction and index j refers to the y direction. The vector $x[]$ stores the left faces of the grid cells, while the vector $y[]$ refers to the bottom faces of respective grid cells. The location of the middle of the grid cells are stored in $x_m(i)$ and $y_m(j)$ vectors. It is important to note that since this is Matlab, the indexing starts from 1 and not 0, unlike other coding languages. Further, the index extends do not start at 1, since we need to add cells outside of the domain to enforce the boundary conditions. Lastly, the values $\frac{1}{\Delta x}$ and $\frac{1}{\Delta y}$ were pre-calculated, as divisions are computationally more expensive than multiplications, and pre-computing the said values enables us to use multiplication instead of division on later calculations, which in the end results in lower run times.

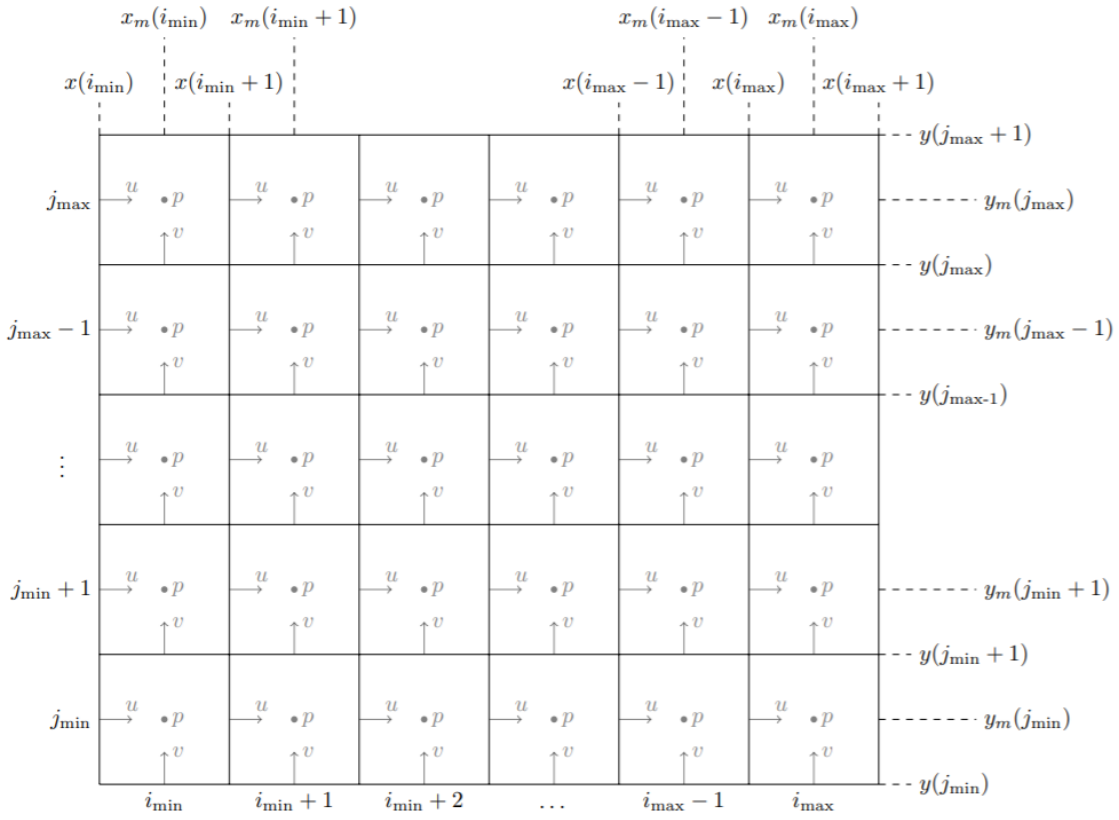

Fig. 1 Computational Grid showing the Location of Pressures and Velocities, along with the Spatial Locations $x_m(i), y_m(j), x(i), y(j)$ **[2]**

C. Temporal Discretization

Eq. (1) can be re-written as

$$\frac{\partial \boldsymbol{u}}{\partial t} = -\boldsymbol{u} \cdot \nabla \boldsymbol{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \boldsymbol{u} \tag{3}$$

The above equation can be descretized temporally by

$$\frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^n}{\Delta t} = -\frac{1}{\rho}\nabla p^{n+1} - \boldsymbol{u}^n \cdot \nabla \boldsymbol{u}^n + \nu \nabla^2 \boldsymbol{u}^n \tag{4}$$

where n refers to the current time step, and $n+1$ refers to the next time step. The time-step Δt should be chosen such that it satisfies $\frac{u\Delta t}{\Delta x} < 1$. This inequality is known as the CFL (Courant-Freidrichs-Lewy) Condition, and is a necessary condition for convergence.

To introduce continuity into the above equation, the equation is solved using the fractional-step or predictor-corrector method. This algorithm has an initial predictor step, which predicts the value from a given function, while the corrector step refines the predicted value using another method to the value at the same point.

The *predictor step* computes the predicted velocity u^* by solving the momentum equation without

including the pressure terms, i.e.,

$$\frac{\boldsymbol{u}^* - \boldsymbol{u}^n}{\Delta t} = -\boldsymbol{u}^n \cdot \nabla \boldsymbol{u}^n + \nu \nabla^2 \boldsymbol{u}^n \tag{5}$$

while the *corrector step* solves for velocity $\mathbf{u}^{n+1}$ by using the pressure term. i.e.,

$$\frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^*}{\Delta t} = -\frac{1}{\rho} \nabla p^{n+1} \tag{6}$$

Hence, it can be seen that Eq. (4) = Eq. (5) + Eq. (6)

The pressure is found such that u^{n+1} satisfies the continuity equation by solving

$$\nabla^2 p^{n+1} = -\frac{\rho}{\Delta t} \nabla \cdot \boldsymbol{u}^* \tag{7}$$

which can be derived by taking the divergence of Eq. (5) and enforcing $\nabla \cdot \boldsymbol{u}^{n+1} = 0$ This equation is referred to as the Pressure Poisson Equation.

D. Discretization of Momentum in x-Direction

The momentum equation (ommiting pressure terms) in the x direction can be discretized using finite difference as

$$\frac{u^* - u^n}{\Delta t} = \nu \left(\frac{\partial^2 u^n}{\partial x^2} + \frac{\partial^2 u^n}{\partial y^2} \right) - \left(u^n \frac{\partial u^n}{\partial x} + v^n \frac{\partial u^n}{\partial y} \right) \tag{8}$$

Hence, the predictor step for u velocity can be written as

$$u^* = u^n + \Delta t \left(\nu \left(\frac{\partial^2 u^n}{\partial x^2} + \frac{\partial^2 u^n}{\partial y^2} \right) - \left(u^n \frac{\partial u^n}{\partial x} + v^n \frac{\partial u^n}{\partial y} \right) \right) \tag{9}$$

where u^* is the predicted velocity in x direction. The viscous and convective terms are discretized using 1^{st} and 2^{nd} order central difference and spatial difference

$$\begin{aligned}
\frac{\partial^2 u}{\partial x^2} &= \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} \\
\frac{\partial^2 u}{\partial y^2} &= \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} \\
u \frac{\partial u}{\partial x} &= u_{i,j} \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} \\
v \frac{\partial u}{\partial y} &= \frac{1}{4}(v_{i-1,j} + v_{i,j} + v_{i-1,j+1} + v_{i,j+1}) \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y}
\end{aligned} \tag{10}$$

Fig. 2 shows the velocities used in the discretization.

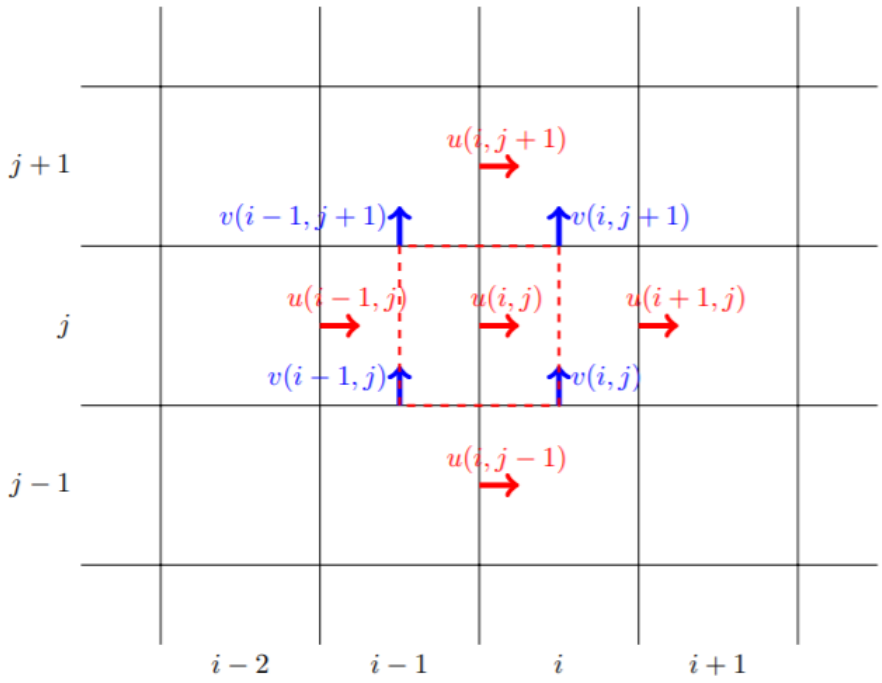

Fig. 2 Velocities and their locations on the grid [2]

Given below is the MATLAB code snippet that computes u^*

```
% u_star calc %
for j = j_min:j_max
    for i = i_min+1:i_max
        v_curr = 0.25*(v(i-1,j) + v(i-1,j+1) + v(i,j)+v(i,j+1));

        u_star(i,j) = u(i,j) + dt*(nu*(u(i-1,j) - 2*u(i,j)+u(i+1,j))*dxi^2 ...
            + nu*(u(i,j-1) - 2*u(i,j) + u(i,j+1))*dyi^2 ...
            - u(i,j)*(u(i+1,j) - u(i-1,j))*0.5*dxi...
            - v_curr*(u(i,j+1) - u(i,j-1))*0.5* dyi);
    end
end
```

E. Dicretization of Momentum in y-Direction

Similarly, the momentum equation (ommiting pressure terms) in the y direction can be discretized using finite difference as

$$\frac{v^* - v^n}{\Delta t} = \nu\left(\frac{\partial^2 v^n}{\partial x^2} + \frac{\partial^2 v^n}{\partial y^2}\right) - \left(u\frac{\partial v^n}{\partial x} + v^n\frac{\partial v^n}{\partial y}\right) \tag{11}$$

Hence, the predictor step for v velocity can be written as

$$v^* = v^n + \Delta t \left(\nu \left(\frac{\partial^2 v^n}{\partial x^2} + \frac{\partial^2 v^n}{\partial y^2} \right) - \left(u \frac{\partial v^n}{\partial x} + v^n \frac{\partial v^n}{\partial y} \right) \right) \tag{12}$$

where v^* is the predicted velocity in y direction. Again, the viscous and convective terms are discretized using 1^{st} and 2^{nd} order central difference and spatial difference

$$\begin{aligned}
\frac{\partial^2 v}{\partial x^2} &= \frac{v_{i-1,j} - 2v_{i,j} + v_{i+1,j}}{\Delta x^2} \\
\frac{\partial^2 v}{\partial y^2} &= \frac{v_{i,j-1} - 2v_{i,j} + v_{i,j+1}}{\Delta y^2} \\
u \frac{\partial v}{\partial x} &= \frac{1}{4}(u_{i,j-1} + u_{i,j} + u_{i+1,j-1} + u_{i+1,j}) \frac{v_{i+1,j} - v_{i-1,j}}{2\Delta x} \\
v \frac{\partial v}{\partial y} &= v_{i,j} \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y}
\end{aligned} \tag{13}$$

Fig. 3 shows the velocities used in the discretization.

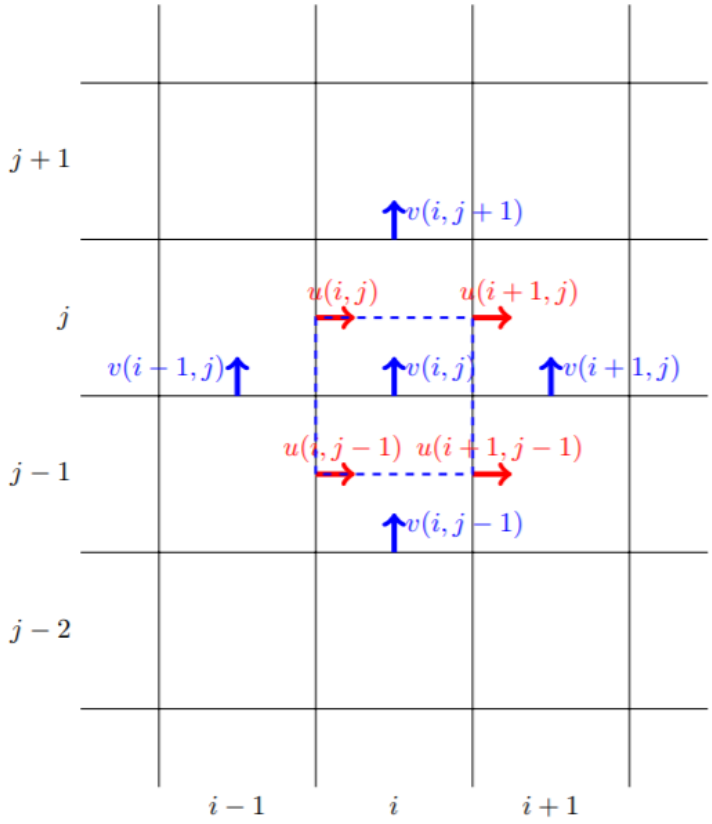

Fig. 3 Velocities and their locations on the grid [2]

Given below is the MATLAB code snippet that computes v^*

```
% v_star calc %
for j = j_min+1:j_max
   for i = i_min:i_max
```

```
        u_curr = 0.25*(u(i,j-1) + u(i,j) + u(i+1,j-1) + u(i+1,j));

        v_star(i,j) = v(i,j) + dt*(nu*(v(i-1,j) - 2*v(i,j) + v(i+1,j))*dxi^2 ...
            + nu*(v(i,j-1) - 2*v(i,j) + v(i,j+1))*dyi^2 ...
            - u_curr*(v(i+1,j) - v(i-1,j))*0.5*dxi...
            - v(i,j)*(v(i,j+1) - v(i,j-1))*0.5* dyi);
    end
end
```

F. Poisson Equation

The Pressure Poisson equation derived in section C (Eq. (6)) is the equation that satisfies the incomressibility and continuity conditions. This equation can be re-written as

$$\boldsymbol{L}\boldsymbol{p}^{n+1} = \boldsymbol{R} \tag{14}$$

where $\boldsymbol{L} = \nabla^2$ is the Laplacian operator, and $\boldsymbol{p}^{n+1}$ is the pressure in each computational grid cell. The right hand side of the equation $\left(-\frac{\rho}{\Delta t}\nabla \cdot \boldsymbol{u}^*\right)$ can be equated to a single matrix $\mathbf{R}$, giving us the equation above. Using the MATLAB Symbolic left division operator (\), the equation can be solved for p^{n+1} by $p^{n+1} = L\backslash R$.

Boundary Conditions for the Pressure Poisson equation are Neumann or zero derivative. Furthermore, all pressures computed are based of a previously set value in 1 computational cell.

The pressure equation is discretized as

$$\begin{aligned}
\nabla^2 p^{n+1} &= \frac{\partial^2 p^{n+1}}{\partial x^2} + \frac{\partial^2 p^{n+1}}{\partial y^2} \approx \frac{p^{n+1}_{i-1,j} - 2p^{n+1}_{i,j} + p^{n+1}_{i+1,j}}{\Delta x^2} + \frac{p^{n+1}_{i,j-1} - 2p^{n+1}_{i,j} + p^{n+1}_{i,j+1}}{\Delta y^2} \\
\nabla \cdot \boldsymbol{u}^* &= \frac{\partial u^*}{\partial x} + \frac{\partial v^*}{\partial y} \approx \frac{u^*_{i+1,j} - u^*_{i,j}}{\Delta x} + \frac{v^*_{i,j+1} - v^*_{i,j}}{\Delta y}
\end{aligned} \tag{15}$$

Fig. 4 provides a visual representation of the position of pressures and velocities on the computation grid

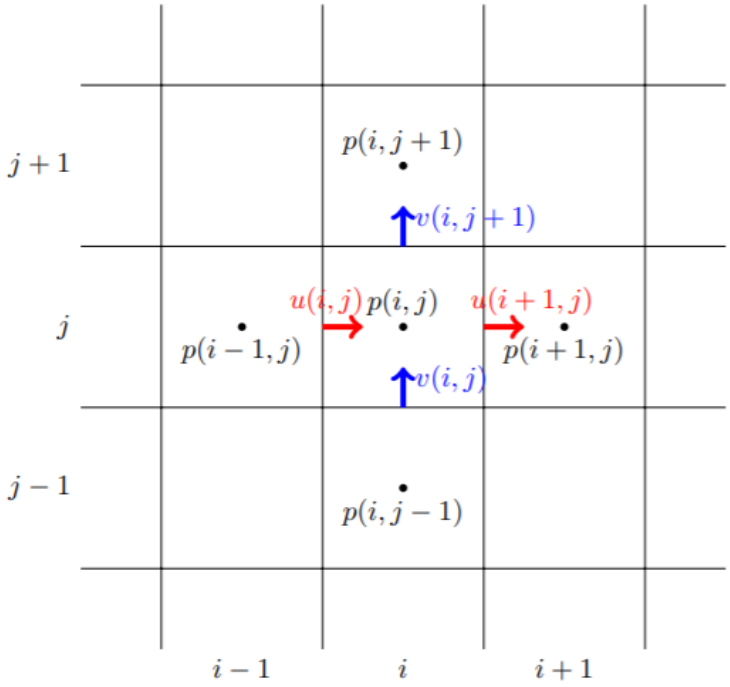

Fig. 4 Relative location of pressure and velocities on the grid [2]

Using this discretization we can write Eq. (14) as

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{-1}{\Delta x} & D_y & \frac{-1}{\Delta x} & 0 & -\frac{-1}{\Delta y} & 0 & 0 & 0 & 0 \\ 0 & \frac{-1}{\Delta x} & D_{xy} & 0 & 0 & -\frac{-1}{\Delta y} & 0 & 0 & 0 \\ -\frac{-1}{\Delta y} & 0 & 0 & D_x & \frac{-1}{\Delta x} & 0 & -\frac{-1}{\Delta y} & 0 & 0 \\ 0 & -\frac{-1}{\Delta y} & 0 & \frac{-1}{\Delta x} & D & \frac{-1}{\Delta x} & 0 & -\frac{-1}{\Delta y} & 0 \\ 0 & 0 & -\frac{-1}{\Delta y} & 0 & \frac{-1}{\Delta x} & D_x & 0 & 0 & -\frac{-1}{\Delta y} \\ 0 & 0 & 0 & -\frac{-1}{\Delta y} & 0 & 0 & D_{xy} & \frac{-1}{\Delta x} & 0 \\ 0 & 0 & 0 & 0 & -\frac{-1}{\Delta y} & 0 & \frac{-1}{\Delta x} & D_y & \frac{-1}{\Delta x} \\ 0 & 0 & 0 & 0 & 0 & -\frac{-1}{\Delta y} & 0 & \frac{-1}{\Delta x} & D_{xy} \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} p(1,1) \\ p(2,1) \\ p(3,1) \\ p(1,2) \\ p(2,2) \\ p(3,2) \\ p(1,3) \\ p(2,3) \\ p(3,3) \end{bmatrix}}_{\mathbf{p}^{n+1}} = \underbrace{\begin{bmatrix} R(1,1) \\ R(2,1) \\ R(3,1) \\ R(1,2) \\ R(2,2) \\ R(3,2) \\ R(1,3) \\ R(2,3) \\ R(3,3) \end{bmatrix}}_{R} \tag{16}$$

where $D = \frac{2}{\Delta x} + \frac{2}{\Delta y}$, $D_x = D - \frac{1}{\Delta x}$, $D_y = D - \frac{1}{\Delta y}$, and $D_{xy} = D - \frac{1}{\Delta x} - \frac{1}{\Delta y}$. Note that Eq. (16) is a case where $n_x \times n_y = 3 \times 3$

Since the Laplacian only depends on the computational mesh, it was computed at the beginning of the computation once.

Given below is the MATLAB code to compute the Laplacian Matrix

```
% Laplace Operator
L = zeros(nx*ny, nx*ny) ;

for j = 1:ny
    for i = 1:nx
        L(i+(j-1)*nx, i+(j-1)*nx) = 2*dxi^2 + 2*dyi^2;

        for ii = i-1: 2:i+1
            if(ii > 0 && ii <= nx) % Interior point(s)
                L(i+(j-1)*nx, ii+(j-1)*nx) = -dxi^2;
            else % Neumann conditions on boundary
                L(i+(j-1)*nx, i+(j-1)*nx) = L(i+(j-1)*nx, i+(j-1)*nx) - dxi^2;
            end
        end

        for jj = j-1: 2: j+1
            if(jj > 0 && jj <= ny) % Interior point(s)
                L(i+(j-1)*nx, i+(jj-1)*nx) = -dyi^2;
            else % Neumann conditions on boundary
                L(i+(j-1)*nx, i+(j-1)*nx) = L(i+(j-1)*nx, i+(j-1)*nx) - dyi^2;
            end
        end

    end
end

% Pressure in First Cell
L(1,:) = 0 ; L(1,1)= 1;
```

MATLAB code to compute the $\boldsymbol{R}$ Matrix is

```
% R calc
n=0;

for j = j_min:j_max
    for i = i_min:i_max
        n = n+1;
        R(n) = -(rho/dt)*((u_star(i+1,j) - u_star(i,j))*dxi + (v_star(i,j+1) - v_star(i,j))*
            dyi);
    end
end
```

The pressure is found by solving $\boldsymbol{L}\boldsymbol{p}^{n+1} = \boldsymbol{R}$ which in MATLAB can be done using

```
% Pressure
```

```
pv = L\(R');
n = 0;
p = zeros(i_max,j_max);

for j = j_min:j_max
    for i = i_min:i_max
        n = n+1;
        p(i,j) = pv(n);
    end
end
```

where pv is the vector representation of the pressure, while p is its representation as a mesh.

G. Corrector Step

After computing the pressure, it is used to update the velocities from $\boldsymbol{u}^*$ to u^{n+1} using Eq. (6). The pressure gradient can be computed using finite differences

$$\frac{\partial p}{\partial x} = \frac{p(i,j) - p(i-1,j)}{\Delta x}, \text{ and}$$
$$\frac{\partial p}{\partial y} = \frac{p(i,j) - p(i,j-1)}{\Delta y}$$

and is used to update the component velocities. MATLAB code to perform the corrector step is:

```
% Corrector Step
for j = j_min:j_max
    for i = i_min+1:i_max
        u(i,j) = u_star(i,j) - (dt/rho)*(p(i,j) - p(i-1,j))*dxi ;
        end
end
for j = j_min+1:j_max
    for i = i_min:i_max
        v(i,j) = v_star(i,j) - (dt/rho)*(p(i,j)-p(i,j-1))*dyi ;
    end
end
```

H. Boundary Conditions

Boundary conditions are needed for the velocity field and the pressure Poisson equation. Since a staggered grid is being used, where the velocities are represented at the faces, it becomes tricky to define the boundary conditions for the velocities. To combat this, fictitious velocity was created outside the domain such that the velocity at the edge of the domain the boundary conditions. The representation of this is shown in Fig. 6, where the velocity at the top boundary is equal to 0.

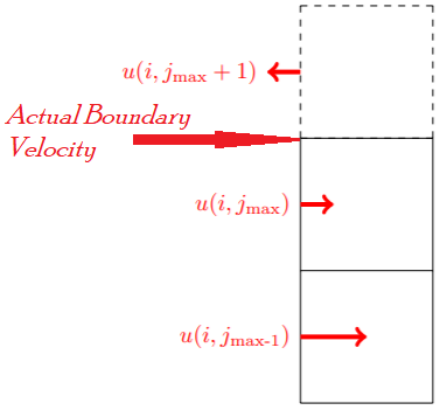

Fig. 5 Example of $u_{top} = 0$ boundary condition

Using the above idea, the boundary conditions were defined in MATLAB

```
% Boundary Conditions
u(:,j_min-1) = u(:,j_min) - 2*(u(:,j_min) - u_bottom);
u(:,j_max+1) = u(:,j_max) - 2*(u(:,j_max) - u_top);
v(i_min-1,:) = v(i_min,:) - 2*(v(i_min,:) - v_left);
v(i_max+1,:) = v(i_max,:) - 2*(v(i_max,:) - v_right);
```

IV. Results

The Solver created above needed to be tested. The condition used for testing was Cavity Flow problem, as it is one of the easiest CFD problems due to it having very simple initial and boundary conditions. The 2-D driven cavity flow problem is the problem of having a square domain with 3 walls having no slip condition and the 4th (top) wall being considered as free stream.

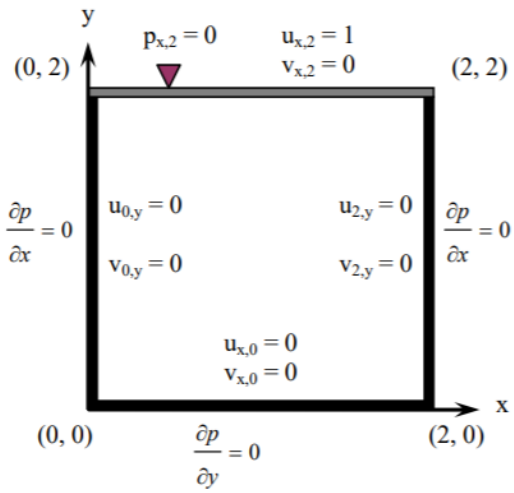

Fig. 6 Cavity Flow Problem [6]

The case used for the test is displayed above, and further mentioned below.

A. Input Parameters

Given below are the input parameters used to run the test

```
% Input Parameters
nx      =   50;                         % Number of cells in x-direction
ny      =   50;                         % Number of cells in y-direction
Lx      =   2;                          % Length in x-direction
Ly      =   2;                          % Length in y-direction
nu      =   0.1;                        % Kinematic Viscosity
rho     =   1;                          % Density
t_final =   5;                          % Time Duration
dt      =   0.0025;
```

B. Initial Conditions

$$u = 0; v = 0; p = 0; \text{ for all } (x, y)$$

C. Boundary Conditions

$$\begin{aligned}
&\text{Bottom Wall} && : u_{x,0} = 0, v_{x,0} = 0, \left.\frac{\partial p}{\partial y}\right|_{y=0} = 0 \\
&\text{Left Wall} && : u_{0,y} = 0, v_{0,y} = 0, \left.\frac{\partial p}{\partial x}\right|_{x=0} = 0 \\
&\text{Right Wall} && : u_{0,2} = 0, v_{0,2} = 0, \left.\frac{\partial p}{\partial x}\right|_{x=2} = 0 \\
&\text{Top Wall} && : u_{x,2} = 1, v_{x,2} = 0, p_{x,2} = 0
\end{aligned}$$

D. Plots

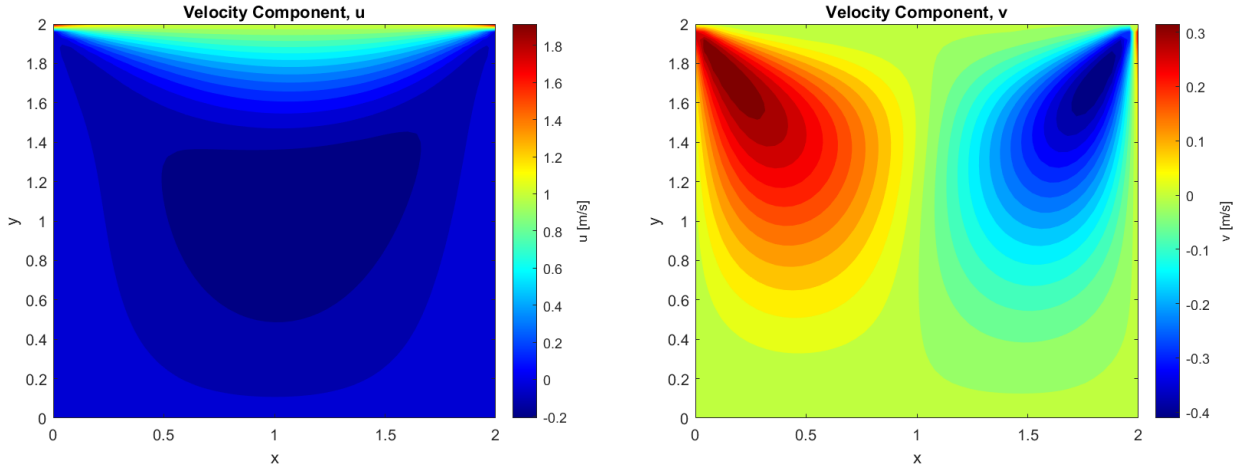

Fig. 7 x component (left) and y component (right) velocities

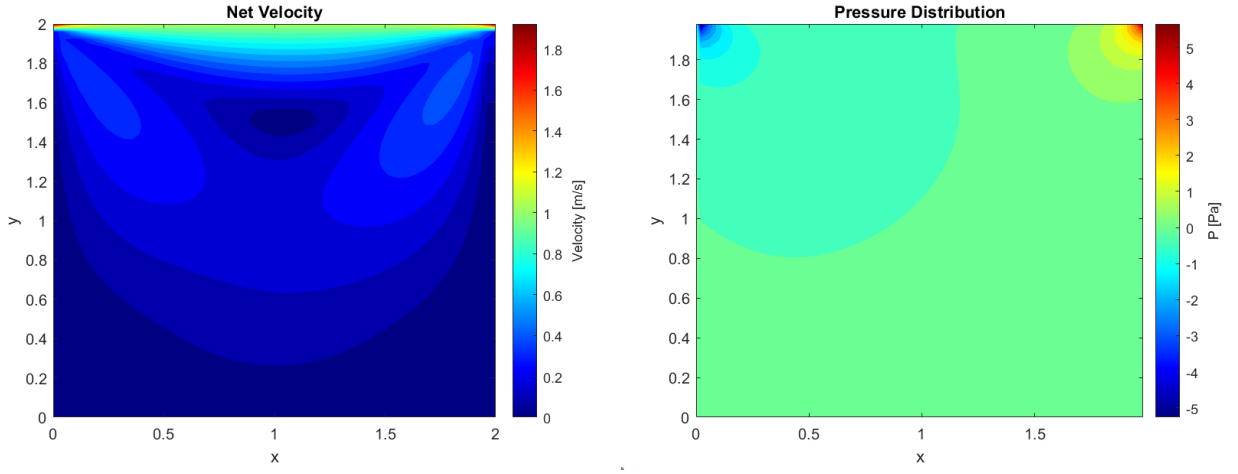

Fig. 8 Net Velocity (left) and Pressure Distribution (right) Plots

E. Discussion

The plots produced in the Results sections are outputted by the code that was discussed earlier. The plots in Fig. 7 display the component velocities of the fluid after simulation has ended, while the plots in Fig. 8 display the net velocity of the fluid body after simulation has ended, along with the pressure distribution in the fluid body. It is important to note that the maximum velocity on the contour plot for x-component is greater than the boundary condition. This is due to the the way the boundary conditions were implemented, which is discussed Section III. Hence the output area is slight larger than the domain area.

To test the validity, the above plots can be compared to a commercial CFD solver's plots. Given below are the plots produced OpenFOAM

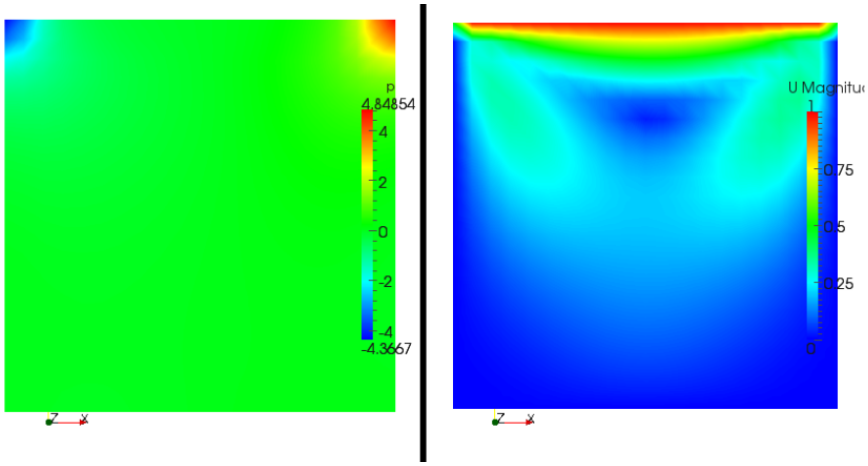

Fig. 9 Cavity Flow Problem OpenFOAM plots

OpenFOAM is a commercially CFD solver. Plots in Fig. 9 are the pressure and velocity outputs produced by the commercial CFD solver. Comparing the plots for Fig. 7 and Fig. 8 to Fig. 9, it can be seen that the plots match up very well. Hence, it can be concluded that the Solver developed was able to simulate the flow for lid driven cavity problem accurately

V. Conclusion

In this report, a CFD solver to solve the Navier-Stokes Equations was developed on Matlab using Finite Difference Method. The code developed was then tested using the lid driven cavity flow problem and the test produced satisfactory results. The plots outputted were then compared to plots from an existing CFD solver to validate its correctness. It was found that the the solution developed by the solver was a good solution, and thus satsifying the objective of this report.

Even though the solver developed produced good results, it can further be improved. The Numerical Method used in this instance was Finite Difference Method, though there are other methods available, such as the Finite Element or Finite Volume Methods, or using a different algorithm like SIMPLE, which might produce better results. Further, the simulation of the code was limited by hardware capabilities. To obtain a more accurate result, a more powerful machine is required, as increasing the number of computational grid cells or decreasing the Δt values was too taxing for the machine, and hence was taking a long time to compute. In future, this solver will be updated to incorporate a better approximation method.

VI. References

[1] Hosch, W. L., “Navier-Stokes equation”, *Encyclopaedia Britannica*, 2020.

[2] Owkes, M., "A guide to writing your first CFD solver", *Montana State University*, 2017.

[3] Goza, A., Lecture Notes, *University of Illinois Urbana Champaign*, 2019.

[4] Barba, L.A., "CFD Python: 12 steps to Navier-Stokes", *Lorena A. Barba Group*, 2013.

[5] Seibold, B., "A compact and fast Matlab code solving the incompressible Navier-Stokes equations on rectangular domains", *Massachusetts Institute of Technology* , 2008.

[6] Becker, N., "Solving Navier – Stokes Equation: Finite Difference Method (FDM)", Accessed 2020.

[7] Ertuk, E., "Discussions On Driven Cavity Flow", *International Journal for Numerical Methods in Fluids* , 2009.

[8] OpenCFD, “OpenFOAM- The Open Source Computational Fluid Dynamics (CFD) Toolbox”, Accessed 2020.