LID DRIVEN CAVITY SOLVED USING FRACTIONAL STEP METHOD

Cherith Lavisetty
Department of Aerospace and Ocean Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24060

The aim of this project is to solve the Incompressible Navier Stokes equations for a lid driven cavity problem. I have used Finite Volume method spatially to discretize the N S equations and have used 2nd Order Adams Bashforth Time integration to move forward in time. I have used three grid resolutions to solve the problem for whihc the results have been published in this report and also have performed convergnce studies and analysed the dependence of time step and spatial discretization on the convergence.

I. Introduction

Any fluid system that we think of are governed by a set of strongly coupled partial differential equations called as Navier Stokes (NS) Equations , Conservation of Mass and Conservation of Energy. This coupled nature of the NS equations makes it so much harder to solve them completely which makes us resort to numerical methods to solve these coupled partial differential equations. In this Project specifically I have used a Finite Volume method framework in which we treat the domain to be divided into finite number of cells and enforce conservation of physical properties at the cell level. We can discretize the NS equations by integrating them over the cell volume and use Gauss Divergence theorem to convert the Volume integrals to Surface integrals and can further evaluate that integral using proper approximations to obtain a set of coupled algebraic equations.

II. Problem Statement

The problem that is solved in this project is a 2D Lid driven Cavity of a square domain of which all the lengths of the boundaries are considered to be of unit length. The top part of the cavity is moving with a velocity of unit magnitude to the right direction , and the velocity is diffused into the fluid inside the cavity due to viscous diffusion. All the other three sides of the box are considered as rigid walls without any mass infusion and also considered to be non slip walls (velocity of the wall surfaces is zero).

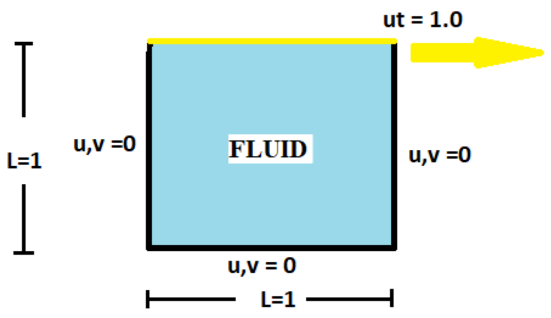


Fig. 1 The domain of the cavity withe the lid moving rightwards

III. Governing Equations

For the given problem in hand we are solving for velocity and Pressure , hence we don't need to solve the energy conservation equation which leaves us with a Continuity equation and two Momentum equations along x and y axes respectively. And given that the problem is set in in-compressible regime we can make further simplifications such as considering the density of the fluid to be constant since the Mach number is very low given the velocity of the fluid. Considering these simplifications the governing equations boil down to the set of equations listed below :

$$\frac{\delta u^*}{\delta x^*} + \frac{\delta v^*}{\delta y^*} = 0 \tag{1}$$

$$\frac{\delta u^*}{\delta t^*} + \nabla^*(u^*\vec{u^*}) = -\frac{\nabla^* P^*}{\rho^*} + \frac{\nabla^*(\mu^*\nabla^* u^*)}{\rho^*} \tag{2}$$

$$\frac{\delta v^*}{\delta t^*} + \nabla^*(v^*\vec{u^*}) = -\frac{\nabla^* P^*}{\rho^*} + \frac{\nabla^*(\mu^*\nabla^* v^*)}{\rho^*} \tag{3}$$

where the quantities $u^*, v^*, x^*, y^*, \nabla^*, \rho^*, t^*, P^*, \mu^*$ are the dimensional quantities.

A. Non-Denationalization of NS Equations

To non denationalise the above equations we can use the following scaling of parameters. For this specific problem we can consider the length of the side of the cavity to be our characteristic length and the velocity of the top most layer to be our characteristic velocity. The scaling of other parameters are listed below:

(x,y) = $(\frac{x^*}{L^*}, \frac{y^*}{L^*})$ (u,v) = $(\frac{u^*}{U^*}, \frac{v^*}{U^*})$ t = $\frac{t^* L^*}{U^*}$

$\rho = \frac{\rho^*}{\rho^*_{ref}}$ $P = \frac{P^*}{\rho^*_{ref}(U^*_{ref})^2}$ $\nabla = \nabla^* L^*$ $\mu = \frac{\mu*}{\mu^*_{ref}}$

Substituting the dimensional quantities in the 1, 2 and 3 we get :

$$\frac{\delta u}{\delta x} + \frac{\delta v}{\delta y} = 0 \tag{4}$$

$$\frac{\delta u}{\delta t} + \nabla(u\vec{u}) = -\frac{\delta P}{\delta x} + \frac{1}{Re}\nabla^2 v \tag{5}$$

$$\frac{\delta v}{\delta t} + \nabla(v\vec{u}) = -\frac{\delta P}{\delta y} + \frac{1}{Re}\nabla^2 v \tag{6}$$

where $Re = \frac{\rho^*_{ref} U^*_{ref} L^*_{ref}}{\mu^*_{ref}}$

IV. Numerical Discretization

To solve the above mentioned equations we need to discretize the continuous partial differential equations into discreet algebraic equations using some discretization methods. For this project I have used a Finite Volume Method to discretize the equations for which we have divide the whole computational domain into some finite number of cells and we perform a volume integral over these PDE's in the cells also called as control volumes. Lets first see how to discretize the partial differential equation over the control volume, lets consider a 2D square cell for our discrete control volume. To discretize the PDE we have to perform a volume integral on our governing equations over our control volume, and apply Gauss divergence theorem to convert volume integrals to surface integrals. For demonstration please look how discretization is done for x momentum momentum equation:

$$\iiint_V (\nabla(u\vec{u}))\ dV = -\iiint_V (\frac{\delta P}{\delta x})\ dV + \iint_V \frac{1}{Re}(\nabla^2 u)\ dV$$

$$\iint_S (uu)\ dy + \iint_S (uv)\cdot\ dx = -\iint_S (P)\ dy + \iint_S \frac{1}{Re}(\nabla u)\cdot\hat{i}\ dy$$

$$((uu)_e - (uu)_w)dy + ((uv)_n - (uv)_s)dx = -(P_e - P_w)dy + (\frac{\delta u}{\delta x}_e - \frac{\delta u}{\delta x}_w)dy$$

In order to obtain a higher order accuracy and to avoid checker boarding of pressure we use a staggered grid approach to discretize the domain , which essentially means that we situate the Pressure variables at the cell centers and u and v velocities on the x and y faces of the cells. Hence the control volumes for Pressure , horizontal velocity and vertical velocity are all different which can be seen in the figure below.

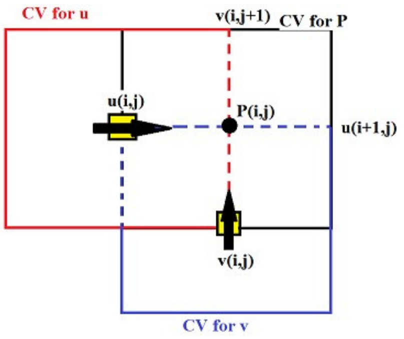


Fig. 2 Control Volumes and indexing of Staggered Grid

A. Discretization of X-Momentum equation:

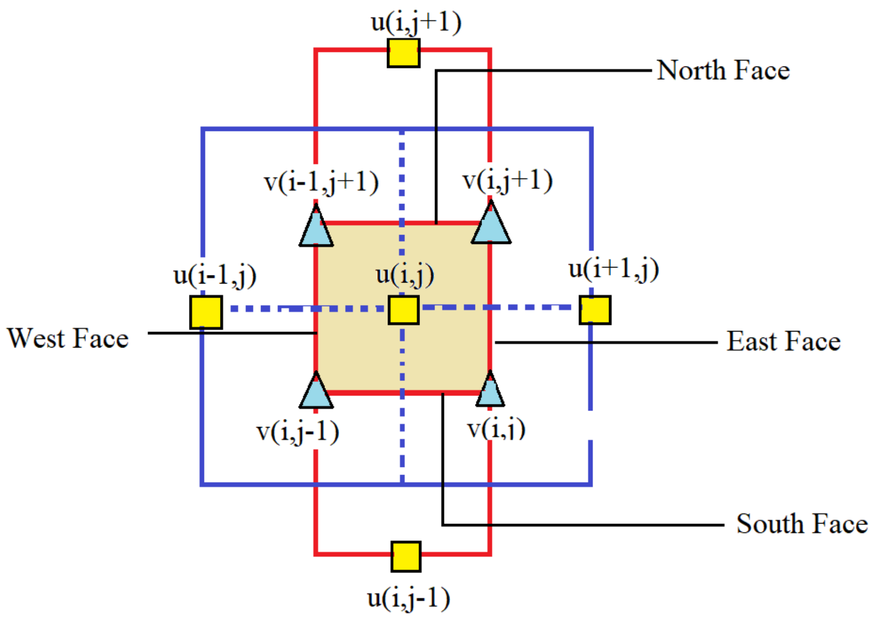


Fig. 3 X momentum control volume and its neighbouring Cells

Using the above staggered approach we can write x momentum equation as: using the red control volume

$$u_e = \frac{u(i,j)+u(i+1,j)}{2}, u_w = \frac{u(i-1,j)+u(i,j)}{2}, u_n = \frac{u(i,j)+u(i,j+1)}{2}, u_s = \frac{u(i,j-1)+u(i,j)}{2}$$

$$v_n = \frac{v(i,j+1)+u(i-1,j+1)}{2}, v_s = \frac{v(i,j)+u(i-1,j)}{2}$$

Diffusion terms:

$$Difterm_x = \frac{u(i+1,j)-2u(i,j)+u(i-1,j)}{dx}dy, Difterm_y = \frac{u(i,j+1)-2u(i,j)+u(i,j-1)}{dy}dx$$

we get :

$$(u_e u_e - u_w u_w)dy + (u_n v_n - u_s v_s)dx = -[P(i,j) - P(i-1,j)]dy + Difterm_x + Difterm_y$$

B. Discretization of Y-Momentum equation:

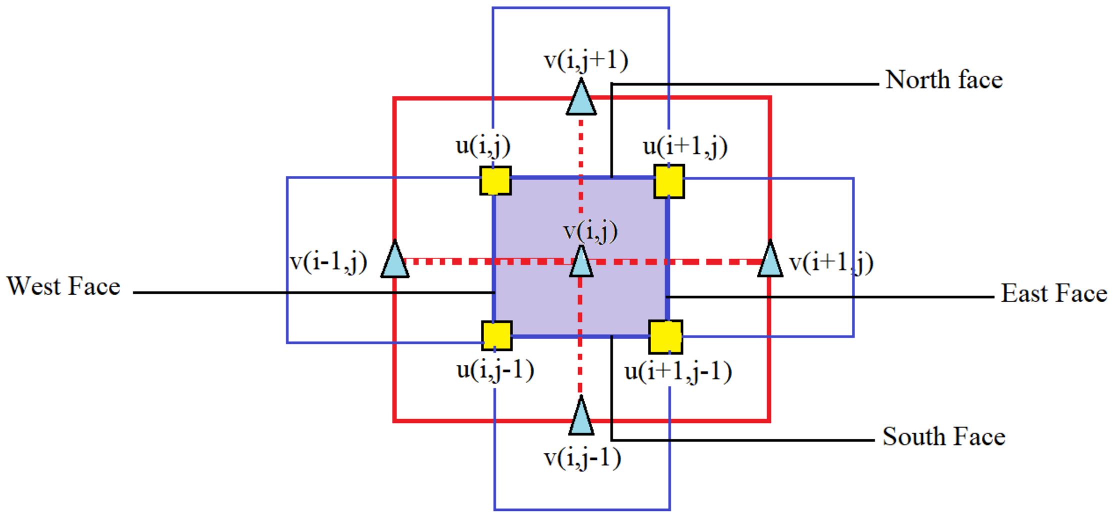


Fig. 4 Y momentum control volume and its neighbouring Cells

$$v_e = \frac{v(i,j)+v(i+1,j)}{2}, v_w = \frac{v(i-1,j)+v(i,j)}{2}, v_n = \frac{v(i,j)+v(i,j+1)}{2}, v_s = \frac{v(i,j-1)+v(i,j)}{2}$$

$$u_n = \frac{u(i+1,j)+u(i+1,j-1)}{2}, u_w = \frac{v(i,j)+u(i,j-1)}{2}$$

Diffusion terms:

$$Difterm_x = \frac{v(i+1,j)-2v(i,j)+v(i-1,j)}{dx}dy, Difterm_y = \frac{v(i,j+1)-2v(i,j)+v(i,j-1)}{dy}dx$$

we get :

$$(u_e v_e - u_w v_w)dy + (v_n v_n - v_s v_s)dx = -[P(i,j)-P(i,j-1)]dx + Difterm_x + Difterm_y$$

C. Discretization of Continuity equation:

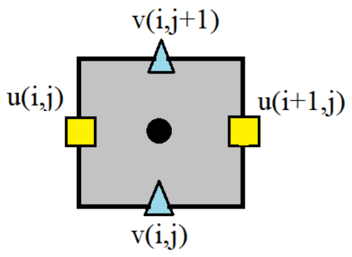


Fig. 5 Pressure Control Volume

The continuity equation is discretized as : (using the Pressure control volume)

$$[u(i+1,j)-u(i,j)]dy + [v(i,j+1)-v(i,j)]dx = 0$$

V. ALGORITHM AND SOLUTION PROCEDURE

For the specific problem in hand I am solving a unsteady 2D in-compressible Navier Stokes equations hence our discretization involves both Time as well as Spatial Discretization. Integrating the unsteady x momentum equation:

Let

$$H_i = -D_j(u_i u_j) + \frac{1}{Re} D_j(G_j u_i)$$

$$D_j = DivergenceOperator, G_j = Gradient$$

$$\iiint_V \left(\int_{t_n}^{t_{n+1}} \frac{\delta u}{\delta t} dt \right) dV = \int_{t_n}^{t_{n+1}} \iiint_V (-\nabla(u\vec{u}) - \frac{\delta P}{\delta x} + \frac{1}{Re}\nabla^2 v) dV dt$$

I am not considering the gradient of Pressure for time integration and Upon simplification we get:

$$\frac{\tilde{u_i} - u_i^n}{\delta t} = \frac{3}{2}H_i^n - \frac{1}{2}H_i^{n-1}$$

The intermediate velocity simplifies to:

$$\tilde{u_i} = u_i^n + (\frac{3}{2}H_i^n - \frac{1}{2}H_i^{n-1})\delta t \tag{7}$$

The actual final equation should be of the form:

$$u_i^{n+1} = u_i^n + (\frac{3}{2}H_i^n - \frac{1}{2}H_i^{n-1})\delta t - G_i P^{n+1} \tag{8}$$

Subtracting the (7) from (8) we get:

$$u_i^{n+1} = \tilde{u_i} - \delta t(G_i P^{n+1}) \tag{9}$$

Similarly we get the following equations on integrating y-momentum equations:

$$\tilde{u_i} = u_i^n + (\frac{3}{2}H_i^n - \frac{1}{2}H_i^{n-1})\delta t$$

$$v_i^{n+1} = \tilde{v_i} - \delta t(G_i P^{n+1}) \tag{10}$$

The velocity at the n+1 the time step has to satisfy the continuity equation that we have discussed earlier hence substitute the (9) and (10) into the discreet continuity equation to get the following Pressure Poisson equation:

$$\nabla^2 P^n + 1 = \frac{1}{\delta t}\vec{\nabla} \cdot (\vec{u}) \tag{11}$$

Hence after every time integration of velocities we try to solve for the Pressure Field and after we get convergence we can correct our velocities using equations (9) and (10) and check for steady state convergence.

VI. BOUNDARY CONDITIONS

The only thing left now to solve the problem is to decide how to deal with the boundary conditions which ultimately defines our problem. To put it simply the boundary conditions for the lid driven cavity problem is the non penetrating boundary condition across the three walls of the cavity, no slip condition on all the walls, No pressure difference across all the walls and also across the top layer and finally the velocity of the lid on the top of the cavity.

Now lets see how to execute these Boundary Conditions in terms of our discreet probelm:

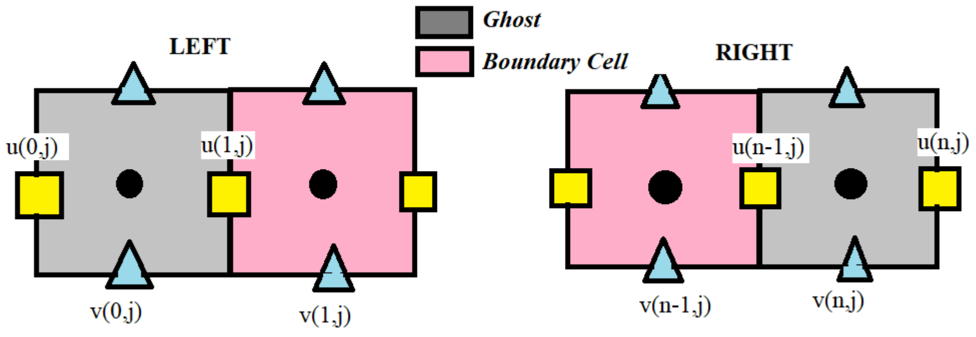


Fig. 6 Left and Right Boundaries

A. Velocity Boundary Conditions

For the Left Wall we can directly set u velocity to be zero and for setting no slip we can set the velocity at the wall to be zero by taking the average of two vertical velocities:

Left Wall:

$$u(1,j)=0,\ \frac{v(0,j)+v(1,j)}{2}=0$$

Right Wall:

$$u(n-1,j)=0,\ \frac{v(n-1,j)+v(n,j)}{2}=0$$

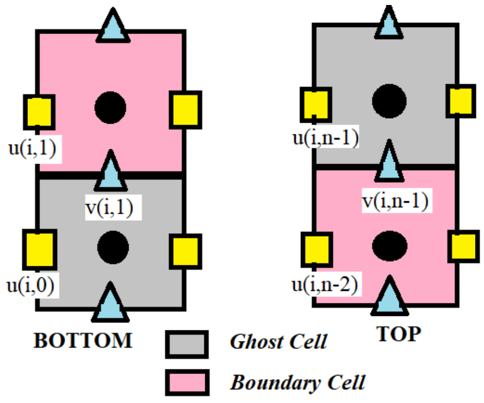


Fig. 7 Bottom and Top Boundaries

For Bottom and Top We can set the v velocities to be zero directly as they are available directly as variables in our problem on domain, and we can take the averages of u velocities on the east and west faces to 0 and 2 on bottom and top boundaries respectively:

Bottom Wall:

$$v(i,1)=0,\ \frac{u(i,0)+u(i,1)}{2}=0$$

Top Wall:

$$v(i,n-1)=0,\ \frac{u(i,n-2)+u(i,n-1)}{2}=1$$

B. PRESSURE BOUNDARY CONDITION

The Pressure Boundary Condition at all the walls is directly incorporated into the Pressure Poisson equation (Continuity equation) (11). After discretizing the Pressure Poisson equation using Finite Volume Method we get Pressure gradient surface fluxes on each of the four faces in the control volume, before making an approximation for the pressure gradient we can directly substitute 0 as the pressure gradient. The x Pressure Gradient on left and right walls can be directly set zero and in the same way the y Pressure Gradient on Top and Bottom walls can be set zero.

$$\frac{\delta P}{\delta x}_{left} = 0; \frac{\delta P}{\delta x}_{right} = 0$$

$$\frac{\delta P}{\delta y}_{bottom} = 0; \frac{\delta P}{\delta y}_{top} = 0$$

VII. LINEAR SOLVER for PRESSURE EQUATION

To obtain Pressure values at each time integration step we need to solve a set of Linear Equations . In our 2D Lid driven cavity Problem after discretizing the Poisson equation we get a penta diagonal system of Linear equations. We can solve this by using Successive Over Relaxation over Cell. But in this project I have Alternating Directional Implicit Solver (Thomas algorithm along x and y) which converts Penta Diagonal system into 2 Tri Diagonal systems.

VIII. STEADY STATE AND CONVERGENCE

In this project I am trying to obtain a steady state solution of the given lid driven cavity problem , which in a sense would mean that our final solution should reach a state of no change which is independent of time. Although absolute steadiness is not guaranteed I am solving here for a tolerance of 1E-8. So after every time integration step, solving the Pressure Poisson equation and updating the velocities finally I calculate the root mean square (L2 Norm) of the change in velocities from the previous time step to check the steadiness of the solution.

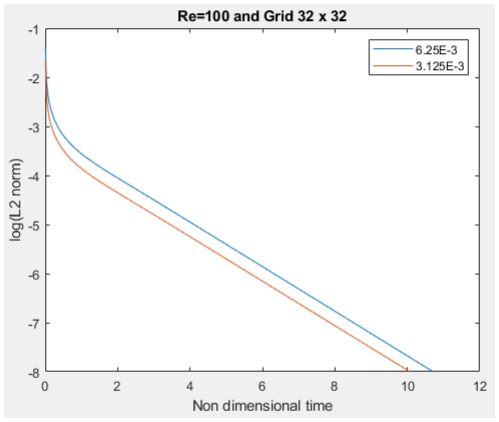
$$L2\, norm\, of\ u = \sqrt{\frac{\sum_{i=1}^{N} (u_i^{n+1} - u_i^{n})^2}{N}}$$

The convergence for the inner iterations of Pressure Poisson solver is evaluated by using the L2 norm for the residuals. For this project I have used a tolerance of 1E-5 as the termination condition for the convergence. One of the important thing to note is to decide whether we need to initialise the pressure matrix after every time iteration. If we are using the Pressure from a previous time step in the Momentum Time integration step (which would obviously be the Pressure correction values calculated from the previous time step) we need to initialise the Pressure Correction matrix every time we start to solve a Pressure Poisson equation. Else if we do not include any Pressure terms while the time integration is done we are solving for the Pressure itself and that need not be initialised after every time integration step, if not initialised it takes the same amount of work to get a converged value of pressure for every time iteration and the convergence time is massively affected.

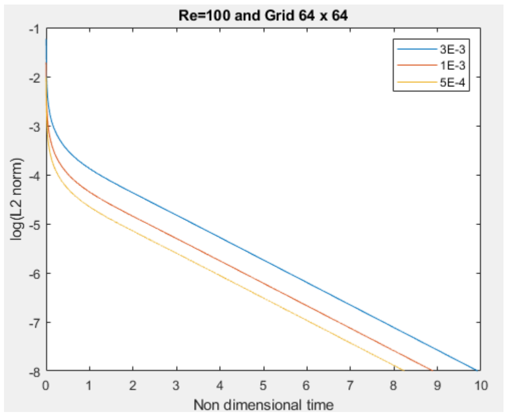
IX. RESULTS

A. Reynolds Number=100

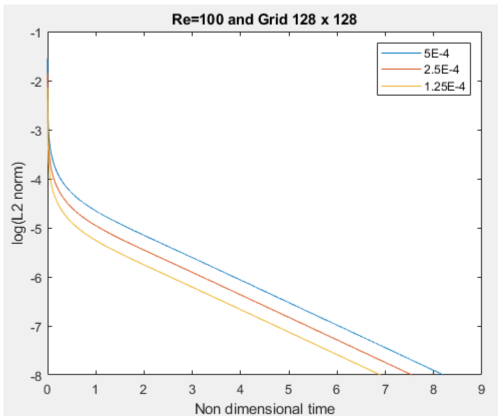
1. Evolution of L2 Norm for change in u



(a)



(b)



(c)

Fig. 8 L2 norm of change in u for different Grid Resolutions

I have plotted the L2 norm of change in u velocity as a function of non dimensional time units till we get certain amount of steadiness in our solution. One of the interesting thing found out was when I tried to use a time step of 0.02 on the coarsest grid resolution of the three (32 x 32) the solution diverged. The CFL condition and 2d Neumann Condition for this problem are:

$$CFL\ condition : \triangle t \leq \frac{\triangle x}{U}; \quad \triangle t \leq 0.03125$$

$$2d\ Neumanncondition : \triangle t \leq 0.25\frac{(\triangle x)^2}{\nu}; \quad \triangle t \leq 0.02441$$

Although the taken time step is within the bounds of the CFL and Neumann condition we still get see the instability because of the higher order explicit time integration and such methods are controlled even more strictly by a non linear CFL condition. For the 2nd order Adams Bashforth time integration the CFL condition is as follows[1] :

$$\triangle t \leq 2^{\frac{2}{3}} C^{\frac{1}{3}} (\frac{\triangle x}{U})^{\frac{4}{3}}$$

Considering C to be 1 to assume the worst case we get a maximum time step of 0.015625 which is less than 0.02 hence the divergence.

2. *CPU time for convergence*

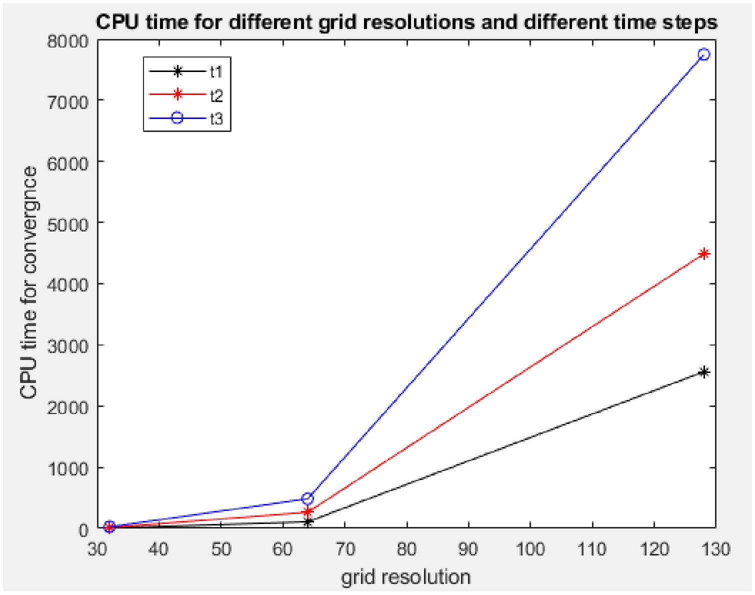


Fig. 9 CPU time for convergence

NOTE: The largest time step 0.02 was making the solution of 32x32 diverge hence I have used the 0.015s time step obtained from Non Linear CFL which is what used to plot the largest time step of the 32x32 grid in the above graph.

Approximating the CPU time dependence on grid resolution as a Power Law: By observing the above graph we can see that for the respective similar time scales it looks like the CPU convergence time and grid resolution is related in some way by a power law. So I have tried to fit the above data for each time scales t1,t2,t3 for the three grids using a curve fitter in matlab to showcase the relation in the form of :

$$CPU\_time = a(x)^b \quad where\ a\ and\ b\ are\ some\ constants, x = grid\ resoultion$$

Using MATLAB here are the results i have found out for the respective governing power laws:

Time step	a	b
t1(largest)	4.831e-07	4.615
t2(medium)	1e-05	4.106
t3(smallest)	2.728e-05	4.012

Although the a values are not exactly close, which could be because of the fact that the smallest , medium and the biggest time scales in the three resolutions are not exactly closely chosen, nevertheless the power of the resolution shows a similar trend indicating the existence of some power law relating the CPU time and the grid resolution.

3. Comparison with Ghia et al

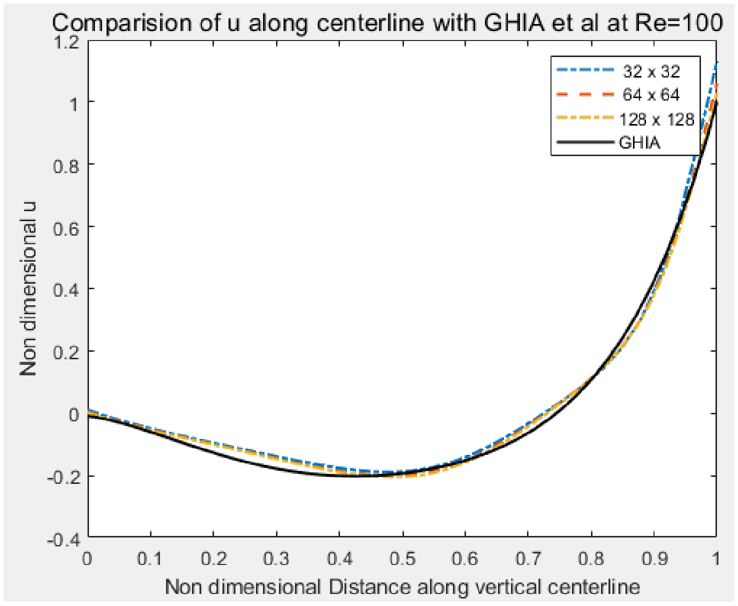


Fig. 10 u velocity along vertical center line passing through Geometric center

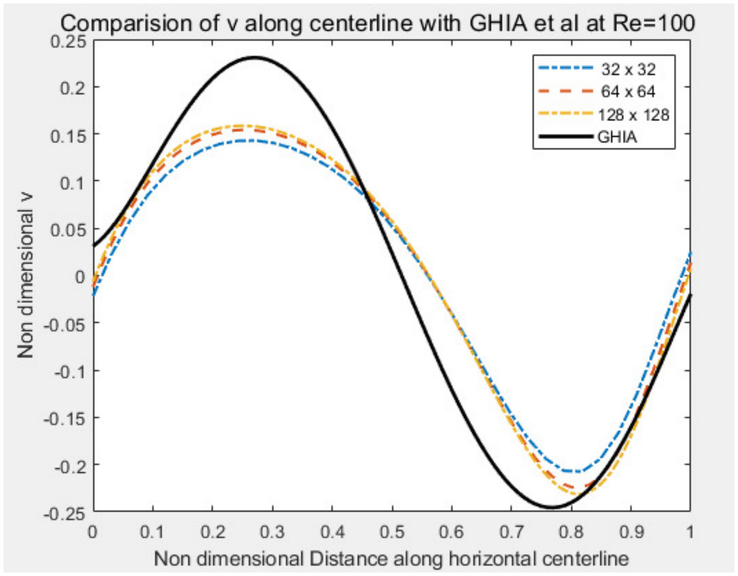


Fig. 11 v velocity along horizontal center line passing through Geometric center

We can see that the solution obtained from this project for Re=100 agreed pretty well with the benchmark solution of Ghia et al. 2. The u velocity solutions matches too well with the benchmark data where as there isn't that good an agreement with the v velocity solution although the solution approximately reflects the same nature of the solution as the benchmark data.

B. Reynolds Number = 1000

1. Restriction of time step:

For the Re=1000 and the grid resolution of 128x128 we have to evaluate all the necessary conditions to verify the most restrictive condition on our maximum time step. And i have compared the same for the Re=100 case to see what condition is most restrictive in either of the cases:

Stability Condition	Re=100	Re=1000
Linear CFL conditon: $\delta t \leq \frac{\delta x}{U}$	0.0078125	0.0078125
2d Neumann Condition: $\triangle t \leq 0.25\frac{(\triangle x)^2}{\nu}$	<u>0.001525</u>	0.01525
Non Linear CFL: $\triangle t \leq 2^{\frac{2}{3}}C^{\frac{1}{3}}(\frac{\triangle x}{U})^{\frac{4}{3}}$	0.0024607	<u>0.0024607</u>

From the above table we can clearly see that the most restrictive condition for the two cases are quite different. For the case of Re=100 which is a viscous dominated flow the most restrictive stability condition is the 2d neumann condition for the 128 x 128 case where as for the Re=1000 case in which viscous effects are significantly lower than a Re=100 flow the Non linear CFL condition is the most restrictive stability condition over the time step. This clearly reflects that the stability of a numerical scheme is directly controlled by the dynamics of the respective problem and the respective effects govern the most restrictive time step. Although I have mentioned that the maximum time step for Re=1000 we can use is 0.00246 my solution has diverged forcing me to restrict the time step even further to 0.002 .

2. CPU time for Convergence

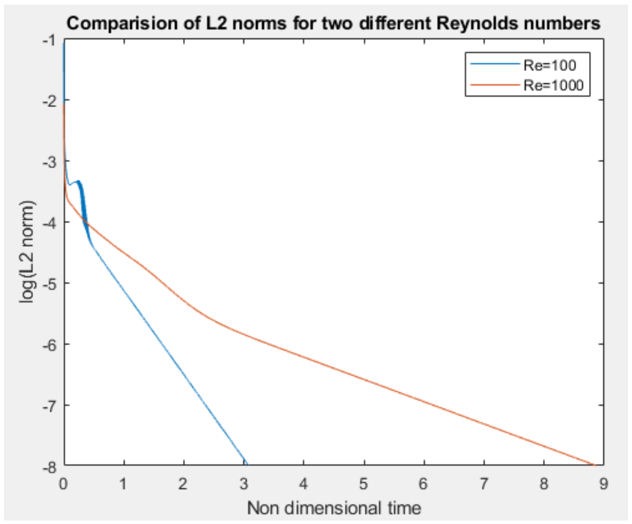


Fig. 12 Bottom and Top Boundaries

From the graph above we can clearly see that with Re =1000 it takes about 3 times more time to reach steady than with Re=100. This is because the low Reynolds number case is more stable because the viscous effects are more dominant and we can clearly see how the viscous terms gives more stability damping out the instabilities in the flow, where as in high Reynolds number flows the Convection effects are more dominating then Viscous effects hence looses that stabilising property.

3. Comparison with Ghia et al

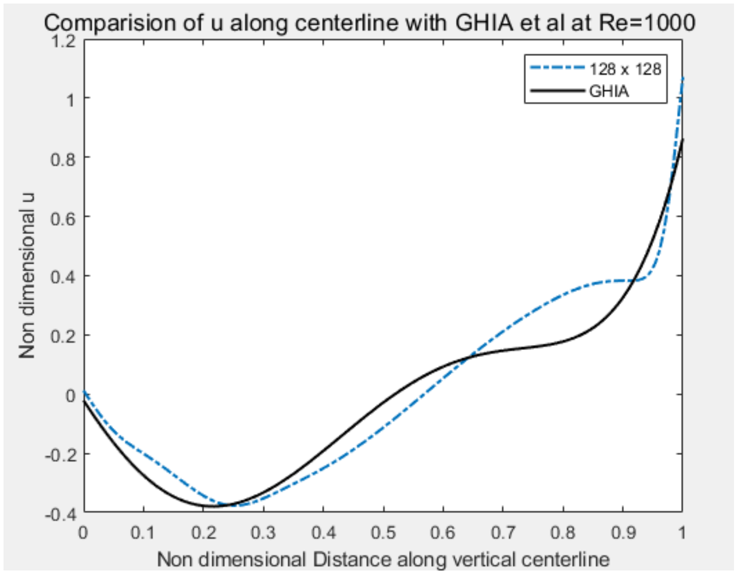


Fig. 13 Bottom and Top Boundaries

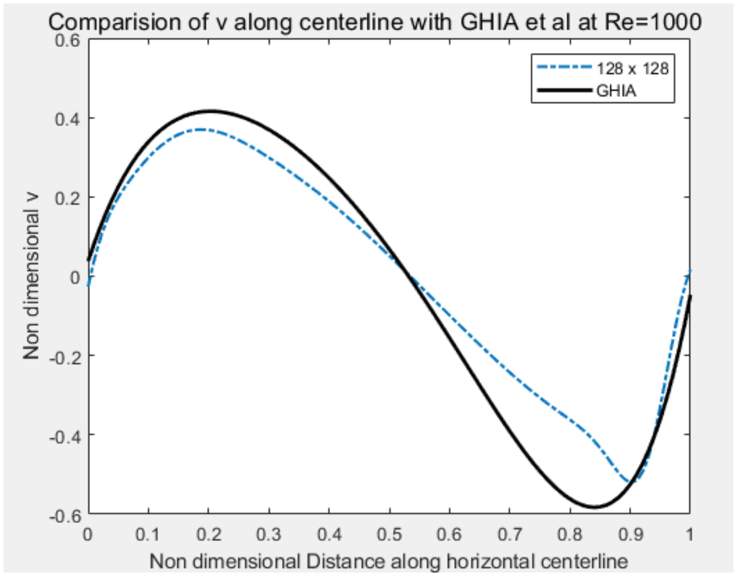


Fig. 14 Bottom and Top Boundaries

For Re=1000 , the steady state solution that I have obtained behaves almost the same as the Re=100 case. In fact the v velocity solution agrees better in this case, it exhibits the same nature as Ghia et al. 's solution and also is closer than the previous case. The u velocity distribution agrees very well just like the previous case.

C. Methods to Accelerate Convergence:

As we know that solving the NS equations to convergence is a heavy calculation and it takes quite a lot of time especially on the finer grids. Few techniques to accelerate the convergence is to :

- Use ADI instead of Plain Gauss Seidel which I have implemented
- Another way is to use Succesive Over Relaxation in Gauss Seidel and for the optimal relaxation factor to get the maximum convergence speed use the below relation[3]:

$$\omega_{opt} = \frac{2}{1 + \sin(\frac{\pi}{N+1})}$$

- Also you can use over relaxation on ADI scheme but the relaxation parameter cannot go beyond 1.32 . And there is no analytical formula to find an optimal paramter.
- And another promising approach is to write a MULTIGRID solver to resolve the low wave number errors faster, and this can be extended to multiple levels and the time reduces logarithamically.

The above mentioned techniques have been tested except the Multigrid method and the code will be presented in the appendix for your reference.

References

[1] Schneider, K., Kolomenskiy, D., and Deriaz, E., "Is the CFL condition sufficient? Some remarks," *The Courant–Friedrichs–Lewy (CFL) Condition: 80 Years After Its Discovery*, 2013, pp. 139–146.

[2] Ghia, U., Ghia, K. N., and Shin, C., "High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method," *Journal of computational physics*, Vol. 48, No. 3, 1982, pp. 387–411.

[3] Yang, S., and Gobbert, M. K., "The optimal relaxation parameter for the SOR method applied to the Poisson equation in any space dimensions," *Applied Mathematics Letters*, Vol. 22, No. 3, 2009, pp. 325–331.

Appendix

Please find the code used to solve the problem below:

```
    #include <iostream>
#include <cmath>
#include <vector>
#include <fstream>
#include <iomanip>
#include <fstream>
#include <chrono>
#include <utility>
#include <algorithm>
using namespace std;
using namespace std::chrono;
typedef vector<vector<double>> matrix;
auto start =high_resolution_clock::now();
/*------------------------- FUNCTIONS FOR ALGORITHM ------------------------------*/
void FV_Momentum();
void Pressure_Poisson();
void TDMA_X( int , matrix);
void TDMA_Y( int , matrix);
void Velocity_Correction();
void Velocity_L2Norm();
void Output_Solution();

/* FUNCTION FOR OPERATIONS */
std::tuple<matrix,matrix> Update_Boundaries( matrix , matrix );
void printmatrix(matrix);

/*------------------------PARAMETERS ------------------------------------------ */
int nx =128 , ny=nx;
double lx = 1 , ly=lx ; double dx = lx/nx , dy = ly/ny;
/* -----------------------PHYSICAL PARAMETERS ----------------------------------*/
double Re = 100 ;
double dt = 1.25E-4  ;

/*-------- PARAMETERS FOR CONVERGENCE AND REFERENCE----------------------------- */
int TIME_ITER =0 ; double res_avg = 1.0 ; double w= 1.128  ;
double err_u = 1 , err_v = err_u;

/*------------------ MATRICES FOR THE PROBLEM ----------------------------------*/
matrix    u_nm1( ny+2 , vector<double>( nx+2 , 0.0) );
matrix    v_nm1( ny+2 , vector<double>( nx+2 , 0.0) );
matrix      u_n( ny+2 , vector<double>( nx+2 , 0.0 ));
matrix      v_n( ny+2 , vector<double>( nx+2 , 0.0 ));
matrix  u_tilda( ny+2 , vector<double>( nx+2 , 0.0 ));
matrix  v_tilda( ny+2 , vector<double>( nx+2 , 0.0 ));

/* -----------MATRICES FOR THE PRESSURE POISSON PROBLEM ----------------------*/
matrix  P_corr( ny+2 , vector<double>( nx+2 , 0.0 ));
matrix      Ae( ny+2 , vector<double>( nx+2 , 1.0/pow(dx,2) ));
matrix      Aw( ny+2 , vector<double>( nx+2 , 1.0/pow(dx,2) ));
matrix      An( ny+2 , vector<double>( nx+2 , 1.0/pow(dy,2) ));
matrix      As( ny+2 , vector<double>( nx+2 , 1.0/pow(dy,2) ));

int main(){
    for( int i=1 ; i<ny+1 ; ++i){
        Aw[i][1]  = 0.0;
        Ae[i][nx] = 0.0;
    }
    for( int j=1 ; j<nx+1 ; ++j){
        An[ny][j] = 0.0;
        As[1][j] = 0.0;
    }
    std::tie( u_n , v_n ) = Update_Boundaries( u_n , v_n );
    u_nm1 = u_n ; v_nm1 = v_n;

std::ofstream fout;
fout.open("err_u_log_128_125_E4.txt");
/*.........................................................................................................................................*/
    double vel_tol = 1E-8 ;
    while( err_u > vel_tol || TIME_ITER < 1 ){
        ++TIME_ITER;
        FV_Momentum();
        Pressure_Poisson();
        Velocity_Correction();
        Velocity_L2Norm();
    cout<<" TIME ITER :"<<TIME_ITER<<endl<<'\t';
    cout<<" err_u : "<<err_u<<endl<<'\t';
    cout<<" err_v : "<<err_u<<endl;



/*--------------------------------- RECORDING EVOLUTION OF ERROR VS TIME---------------------*/
   // auto t2 = high_resolution_clock::now();
   // auto elapsed_time = duration_cast<milliseconds>( t2 - start );
    fout<<err_u<<'\t'<<TIME_ITER<<endl;

    }
fout.close();
/*......................................................................................................................................*/
    //printmatrix( v_n );
    cout<<endl<<endl;
    //printmatrix( u_n );
    Output_Solution();
/*--------------------------------CPU TIME  FOR THE WHOLE SIMULATION-----------------*/
    auto stop = high_resolution_clock::now();
            auto duration =duration_cast<milliseconds>(stop -start);
            cout<<'\n'<<"The time taken for convergence is : "<<duration.count() <<" milliseconds \n";

std::ofstream outfile;
  outfile.open("CPU_time_test_128.txt", std::ios_base::app); // append instead of overwrite
  outfile<<endl<<" 128 x 128 "<<"\t "<<dt<<'\t'<<duration.count() ;

}
```

```
/*------==========================UPDATING THE GHOST CELLS=========================================-------------*/
std::tuple<matrix,matrix> Update_Boundaries( matrix u , matrix v ){
    for( int i=1 ; i<ny+1 ; ++i){
        u[i][1]    = 0.0; v[i][0]    = 2*0.0 -  v[i][1]; //0
        u[i][nx+1] = 0.0; v[i][nx+1] = 2*0.0 - v[i][nx]; //0
    }
    for( int j=1 ; j<nx+1 ; ++j){
        u[0][j]    = 2*0.0 -  u[1][j]; v[0][j]    = 0.0; //0
        u[ny+1][j] = 2*1.0 - u[ny][j]; v[ny+1][j] = 0.0;
    }
    return std::tuple(u,v);
}

/*--------==================================TIME INTEGRATION OF VELOCITIES=====================------------------*/
void FV_Momentum(){
    /* X-MOMENTUM Time Integration */
    for( int i=1 ; i<ny+1 ; ++i ){
        for( int j=2 ; j<nx+1 ; ++j ){
            double ue_n = 0.5*( u_n[i][j] + u_n[i][j+1] );                  double ue_nm1 = 0.5*( u_nm1[i][j] + u_nm1[i][j+1] );
            double uw_n = 0.5*( u_n[i][j-1] + u_n[i][j] );                  double uw_nm1 = 0.5*( u_nm1[i][j-1] + u_nm1[i][j] );
            double un_n = 0.5*( u_n[i][j] + u_n[i+1][j] );                  double un_nm1 = 0.5*( u_nm1[i][j] + u_nm1[i+1][j] );
            double us_n = 0.5*( u_n[i-1][j] + u_n[i][j] );                  double us_nm1 = 0.5*( u_nm1[i-1][j] + u_nm1[i][j] );
            double vn_n = 0.5*( v_n[i+1][j-1] + v_n[i][j] );                double vn_nm1 = 0.5*( v_nm1[i+1][j-1] + v_nm1[i][j] );
            double vs_n = 0.5*( v_n[i][j-1] + v_n[i][j] );                  double vs_nm1 = 0.5*( v_nm1[i][j-1] + v_nm1[i][j] );

        double Conv_n = -( ue_n*ue_n - uw_n*uw_n )/dx -( un_n*vn_n - us_n*vs_n )/dy;
        double Conv_nm1 = -( ue_nm1*ue_nm1 - uw_nm1*uw_nm1 )/dx -( un_nm1*vn_nm1 - us_nm1*vs_nm1 )/dy ;
        double Diff_n = (1.0/Re)*( ( u_n[i][j+1] -2*u_n[i][j] + u_n[i][j-1] )/pow( dx ,2 ) + ( u_n[i+1][j] -2*u_n[i][j] + u_n[i-1][j] )/pow( dy ,2 ) ) ;
        double Diff_nm1 = (1.0/Re)*( ( u_nm1[i][j+1] -2*u_nm1[i][j] + u_nm1[i][j-1] )/pow( dx ,2 ) + ( u_nm1[i+1][j] -2*u_nm1[i][j] + u_nm1[i-1][j] )/pow( dy ,2 ) ) ;
        u_tilda[i][j] = u_n[i][j] + dt*( 1.5*( Conv_n + Diff_n ) + 0.5*( Conv_nm1 + Diff_nm1 ) );
        }
    }

    /* Y-MOMENTUM Time Integration */
    for( int i=2 ; i<ny+1 ; ++i ){
        for( int j=1 ; j<nx+1 ; ++j){
            double ve_n = 0.5*( v_n[i][j] + v_n[i][j+1] );                    double ve_nm1 = 0.5*( v_nm1[i][j] + v_nm1[i][j+1] );
            double vw_n = 0.5*( v_n[i][j-1] + v_n[i][j] );                    double vw_nm1 = 0.5*( v_nm1[i][j-1] + v_nm1[i][j] );
            double vn_n = 0.5*( v_n[i][j] + v_n[i+1][j] );                    double vn_nm1 = 0.5*( v_nm1[i][j] + v_nm1[i+1][j] );
            double vs_n = 0.5*( v_n[i-1][j] + v_n[i][j] );                    double vs_nm1 = 0.5*( v_nm1[i-1][j] + v_nm1[i][j] );
            double ue_n = 0.5*( u_n[i-1][j+1] + u_n[i][j+1] );                double ue_nm1 = 0.5*( u_nm1[i-1][j+1] + u_nm1[i][j+1] );
            double uw_n = 0.5*( u_n[i-1][j] + u_n[i][j] );                    double uw_nm1 = 0.5*( u_nm1[i-1][j] + u_nm1[i][j] );
        double Conv_n = -( ue_n*ve_n - uw_n*vw_n )/dx - ( vn_n*vn_n - vs_n*vs_n )/dy ;
        double Conv_nm1 = -( ue_nm1*ve_nm1 - uw_nm1*vw_nm1 )/dx - ( vn_nm1*vn_nm1 - vs_nm1*vs_nm1 )/dy ;
        double Diff_n = (1.0/Re)*( (v_n[i][j+1] - 2*v_n[i][j] + v_n[i][j-1])/pow( dx ,2 ) + ( v_n[i+1][j] -2*v_n[i][j] + v_n[i-1][j])/pow( dy , 2 ) );
        double Diff_nm1 = (1.0/Re)*( (v_nm1[i][j+1] - 2*v_nm1[i][j] + v_nm1[i][j-1])/pow( dx ,2 ) + ( v_nm1[i+1][j] -2*v_nm1[i][j] + v_nm1[i-1][j])/pow( dy , 2 ) ) ;
        v_tilda[i][j] = v_n[i][j] + dt*( 1.5*( Conv_n + Diff_n ) + 0.5*( Conv_nm1 + Diff_nm1 ) );
        }
    }
    u_nm1 = u_n ;
    v_nm1 = v_n ;
    std::tie( u_tilda , v_tilda ) = Update_Boundaries( u_tilda , v_tilda );
}


/*=======================================================PRESSURE EQUATION SOLVER==================================================================== */
void Pressure_Poisson(){
    /* PRESSURE Initialization */
    if( TIME_ITER==1){
        for( int i=1 ; i<ny+1 ; ++i ){
            for( int j=1 ; j<nx+1 ; ++j ){
                P_corr[i][j] = 0.0;
            }
         }
    }


    matrix  S( ny+2 , vector<double>( nx+2 , 0.0 ) );
    matrix res( ny+2, vector<double>( nx+2 , 0.0 ) );
    /* SOURCE TERMS */
    for( int i=1 ; i<ny+1 ; ++i ){
        for( int j=1 ; j<nx +1 ; ++j ){
            S[i][j] = (1.0/dt)*( ( u_tilda[i][j+1] - u_tilda[i][j] )/dx + ( v_tilda[i+1][j] - v_tilda[i][j] )/dy );
        }
    }
    /* Solving Discreet POISSON Equation */
double tol = 1E-5; int ITER=0; res_avg =1.0 ;
    while( res_avg>tol || ITER<10 ){
        ++ITER;

        for( int i=1 ; i<ny+1 ; ++i){
            TDMA_X( i , S );
            TDMA_Y( i , S );
        }

        res_avg = 0.0;
        for( int i=1 ; i<ny+1 ; ++i ){
            for( int j=1 ; j<nx+1 ; ++j ){
                double AE = Ae[i][j] , AW = Aw[i][j] , AN = An[i][j] , AS = As[i][j] ;
                double AP = AE + AW + AN + AS ;
                res[i][j]    =  AP*P_corr[i][j] -( AE*P_corr[i][j+1] + AW*P_corr[i][j-1] + AN*P_corr[i+1][j] + AS*P_corr[i-1][j] - S[i][j] );
                res_avg +=  pow( res[i][j] , 2 )/(nx*ny);
            }
        }

    }
    cout<<endl<<endl;
    cout<<" ITER "<<ITER<<'\t'; cout<<" res_avg "<<res_avg<<endl;


}
```

```
/*=====================================================TDMA FUNCIONS=================================================================================================================*/
void TDMA_X( int row , matrix Source){
    double AE[nx+2] , AW[nx+2] , AN[nx+2] , AS[nx+2] , AP[nx+2] ,  S[nx+2] ;
/*---------------- SETTING UP THE COEFFICIENT MATRIX AND SOURCE VECTOR--------------*/
    for( int j=1 ; j<nx+1 ; ++j){
        AE[j] = Ae[row][j] , AW[j] = Aw[row][j] , AN[j] = An[row][j] , AS[j] = As[row][j] ;
        AP[j] = -( AE[j] + AW[j] + AN[j] + AS[j] ) ;
        S[j]  = w*( Source[row][j] - AN[j]*P_corr[row+1][j] - AS[j]*P_corr[row-1][j] ) + AP[j]*(1-w)*P_corr[row][j] ;
        AW[j] = w*AW[j] , AE[j] = w*AE[j] ;
    }
/*--------------------------- FOWARD SUBSTITUTION---------------------- */
    for( int j=2 ; j<nx+1 ; ++j){
        AP[j] = AP[j] - ( AW[j]/AP[j-1] )*AE[j-1];
        S[j]  = S[j]  - ( AW[j]/AP[j-1] )*S[j-1] ;
    }
/*--------------------------- BACKWARD SUBSTITUTION---------------------- */
    for( int j=nx ; j>0 ; --j ){
        P_corr[row][j] = ( S[j] - AE[j]*P_corr[row][j+1] )/AP[j];
    }
}


void TDMA_Y( int col , matrix Source ){
    double AE[ny+2] , AW[ny+2] , AN[ny+2] , AS[ny+2] , AP[ny+2] , S[ny+2] ;
/* ------------- -SETTING UP COEFFICIENT MATRIX AND SOURCE VECTOR------------ */
    for( int i=1 ; i<ny+1 ; ++i){
        AE[i] = Ae[i][col] , AW[i] = Aw[i][col] , AN[i] = An[i][col] , AS[i] = As[i][col] ;
        AP[i] = -( AE[i] + AW[i] + AN[i] + AS[i] );
        S[i]  = w*( Source[i][col] - AE[i]*P_corr[i][col+1] - AW[i]*P_corr[i][col-1] ) + AP[i]*(1-w)*P_corr[i][col] ;
        AS[i] = w*AS[i] , AN[i] = w*AN[i] ;
    }
/*--------------------------- FOWARD SUBSTITUTION---------------------- */
    for( int i=2 ; i<ny+1 ; ++i){
        AP[i] = AP[i] - ( AS[i]/AP[i-1] )*AN[i-1];
        S[i]  = S[i]  - ( AS[i]/AP[i-1] )*S[i-1] ;
    }
/*--------------------------- BACKWARD SUBSTITUTION---------------------- */
    for( int i=ny ; i>0 ; --i){
        P_corr[i][col] = ( S[i] - AN[i]*P_corr[i+1][col] )/AP[i];
    }
}


/*=================================================VELOCITY CORRECTIONS AND ERRORS ========================================================================================*/
void Velocity_Correction(){
    for( int i=1 ; i<ny+1 ; ++i ){
        for( int j=2 ; j<nx+1 ; ++j ){
            u_n[i][j] = u_tilda[i][j] - dt*( P_corr[i][j] - P_corr[i][j-1]  )/dx ;
        }
    }
    for( int i=2 ; i<ny+1 ; ++i ){
        for( int j=1 ; j<nx+1 ; ++j ){
            v_n[i][j] = v_tilda[i][j] - dt*( P_corr[i][j] - P_corr[i-1][j] )/dy ;
        }
    }
    std::tie( u_n , v_n ) = Update_Boundaries( u_n , v_n );
}

void Velocity_L2Norm(){
    err_u =0 ; err_v =0;
    for( int i=1 ; i<ny+1 ; ++i ){
        for( int j=2 ; j<nx+1 ; ++j ){
            err_u += pow( (u_n[i][j] - u_nm1[i][j] ) , 2 )/((nx-1)*(ny-1));
        }
    }
    for( int i=2 ; i<ny+1 ; ++i ){
        for( int j=1 ; j<nx+1 ; ++j){
            err_v += pow( (v_n[i][j] - v_nm1[i][j] ) , 2 )/((nx-1)*(ny-1));
        }
    }
    err_u = pow( err_u , 0.5 ) ; err_v = pow( err_v , 0.5 );
}

/* ========================================================== OUTPUT ================================================================================================== */
void printmatrix( matrix mat ){
    int Rows = mat.size();
    int Cols = mat[0].size();
    for( int i=0 ; i<Rows ; ++i){
        for( int j=0 ; j<Cols ; ++j){
            cout<<mat[i][j]<<'\t';
            if( j==Cols-1 ) cout<<endl;
        }
    }
}

void Output_Solution(){
    ofstream fout("u_128.txt");
    for( int i=0 ; i<ny+2 ; ++i  ){
        for( int j=0 ; j<nx+2 ; ++j){
            fout<<u_n[i][j]<<'\t';
            if( j==ny+1 ) fout<<endl;
        }
    }
    fout.close();

    fout.open("v_128.txt");
    for( int i=0 ; i<ny+2 ; ++i  ){
        for( int j=0 ; j<nx+2 ; ++j){
            fout<<v_n[i][j]<<'\t';
            if( j==ny+1 ) fout<<endl;
        }
    }
    fout.close();
}
```

```
/*----------------------ADDITIONAL CODE SHOWING THE SOR SOLVER FOR POISSON EQUATION -----------------------*/
void Pressure_Poisson(){
    /* PRESSURE Initialization */
    if( TIME_ITER==1 ){
        for( int i=1 ; i<ny+1 ; ++i ){
            for( int j=1 ; j<nx+1 ; ++j ){
                P_corr[i][j] = 0.0;
            }
        }
    }


    matrix  S( ny+2 , vector<double>( nx+2 , 0.0 ) );
    matrix res( ny+2, vector<double>( nx+2 , 0.0 ) );
    /* SOURCE TERMS */
    for( int i=1 ; i<ny+1 ; ++i ){
        for( int j=1 ; j<nx +1 ; ++j ){
            S[i][j] = (1.0/dt)*( ( u_tilda[i][j+1] - u_tilda[i][j] )/dx + ( v_tilda[i+1][j] - v_tilda[i][j] )/dy );
        }
    }
    /* Solving Discreet POISSON Equation */
double tol = 1E-5; int ITER=0; res_avg =1.0 ;
    while( res_avg>tol ){
        ++ITER;
        //cout<<" PRESSURE ITER : "<<ITER<<endl;
        for( int i=1 ; i<ny+1 ; ++i ){
            for( int j=1 ; j<nx+1 ; ++j){
                double AE = Ae[i][j] , AW = Aw[i][j] , AN = An[i][j] , AS = As[i][j] ;
                double AP = AE + AW + AN + AS ;
                P_corr[i][j] = (1-w)*P_corr[i][j] + w*(1.0/AP)*( AE*P_corr[i][j+1] + AW*P_corr[i][j-1] + AN*P_corr[i+1][j] + AS*P_corr[i-1][j] - S[i][j] ) ;
            }
        }
        res_avg = 0.0;
        for( int i=1 ; i<ny+1 ; ++i ){
            for( int j=1 ; j<nx+1 ; ++j ){
                double AE = Ae[i][j] , AW = Aw[i][j] , AN = An[i][j] , AS = As[i][j] ;
                double AP = AE + AW + AN + AS ;
                res[i][j]    =  AP*P_corr[i][j] -( AE*P_corr[i][j+1] + AW*P_corr[i][j-1] + AN*P_corr[i+1][j] + AS*P_corr[i-1][j] - S[i][j] );
                res_avg +=  pow( res[i][j] , 2 )/(nx*ny);
            }
        }
        res_avg = pow( res_avg , 0.5 );

    }
    cout<<endl<<endl;
    cout<<" ITER "<<ITER<<'\t'; cout<<" res_avg "<<res_avg<<endl;

}

/*------------------------------------------------ CODE FOR EXPLICIT TIME INTEGRATION -----------------------------*/
void FV_Momentum(){
    /* X-MOMENTUM Time Integration */
    for( int i=1 ; i<ny+1 ; ++i ){
        for( int j=2 ; j<nx+1 ; ++j ){
            double ue = 0.5*( u_n[i][j] + u_n[i][j+1] );
            double uw = 0.5*( u_n[i][j-1] + u_n[i][j] );
            double un = 0.5*( u_n[i][j] + u_n[i+1][j] );
            double us = 0.5*( u_n[i-1][j] + u_n[i][j] );
            double vn = 0.5*( v_n[i+1][j-1] + v_n[i][j] );
            double vs = 0.5*( v_n[i][j-1] + v_n[i][j] );

        double Conv = -( ue*ue - uw*uw )/dx -( un*vn - us*vs )/dy ;
        double Diff = (1.0/Re)*( ( u_n[i][j+1] -2*u_n[i][j] + u_n[i][j-1] )/pow( dx ,2 ) + ( u_n[i+1][j] -2*u_n[i][j] + u_n[i-1][j] )/pow( dy ,2 ) ) ;
        u_n[i][j] = u_n[i][j] + dt*( Conv + Diff ) ;
        }
    }

    /* Y-MOMENTUM Time Integration */
    for( int i=2 ; i<ny+1 ; ++i ){
        for( int j=1 ; j<nx+1 ; ++j){
            double ve = 0.5*( v_n[i][j] + v_n[i][j+1] );
            double vw = 0.5*( v_n[i][j-1] + v_n[i][j] );
            double vn = 0.5*( v_n[i][j] + v_n[i+1][j] );
            double vs = 0.5*( v_n[i-1][j] + v_n[i][j] );
            double ue = 0.5*( u_n[i-1][j+1] + u_n[i][j+1] );
            double uw = 0.5*( u_n[i-1][j] + u_n[i][j] );

        double Conv = -( ue*ve - uw*vw )/dx - ( vn*vn - vs*vs )/dy ;
        double Diff = (1.0/Re)*( (v_n[i][j+1] - 2*v_n[i][j] + v_n[i][j-1])/pow( dx ,2 ) + ( v_n[i+1][j] -2*v_n[i][j] + v_n[i-1][j])/pow( dy , 2 ) );
        v_n[i][j] = v_n[i][j] + dt*( Conv + Diff ) ;
        }
    }

    std::tie( u_n , v_n ) = Update_Boundaries( u_n , v_n );

}
```