A FINITE VOLUME METHOD ALGORITHM ON A RECTANGULAR DRIVEN CAVITY SIMULATION FOR A STEADY INCOMPRESSIBLE LAMINAR FLOW

Submitted by: TANAY MILIND DESHPANDE

UNDERGRADUATE COMPUTATION ORIENTED RESEARCH
PROJECT REPORT

Contents

1. Introduction 3
2. Acknowledgements 4
3. Discretization of the governing equations 5
4. Modified Pressure correction and under-relaxation 8
5. Subroutine descriptions 11
6. Results and analysis 13

Appendix I- C++ Implementation 15

References 20

INTRODUCTION

The Finite Volume Method (FVM) is commercially the most widely used CFD technique to convert the Navier Stokes partial differential equations (PDEs) that govern fluid flow into a linear algebraic system of equations. This report formulates an algorithm to solve steady-state, laminar and incompressible fluid flow equations with a structured collocated grid on a rectangular domain and describes its implementation using C++.

This new technique uses the error terms from previous u,v/ p iterations to pass information to the next one, thus achieving faster convergence for the Semi Implicit Method for Pressure Linked Equations (SIMPLE) algorithm (Patankar, 1972) for low Reynolds number problems in regular geometries. After setting boundary conditions for pressure and X and Y direction velocities, the Central Differencing scheme is employed for calculating velocities at cell faces. By invoking the divergence theorem, the Navier-Stokes equations are discretized at each cell in the grid into a system of coefficients of neighbouring cell velocities. The algebraic equations representing each line in the grid are then fed into a solver subroutine that implements the Tridiagonal Matrix Algorithm (Thomas, 1949). Then, a new technique is modelled which constructs a different set of pressure corrector equations which use the velocity errors due to neighbouring cells that were neglected in SIMPLE (this omission being the essence of the SIMPLE algorithm). After two Alternating Direction Implicit traversals for velocities, the continuity equation is discretized in a similar manner for evaluating pressure correction terms which update the pressure and velocities at every node for the next iteration.

This report demonstrates that the presented algorithm is faster at achieving convergence (it = 50-70) vs. standard FVM techniques (it = 90-130). The incremental computational effort of the correct under-relaxation steps is meanwhile 2 more calculations per iteration, so it is merely O(it).

ACKNOWLEDGEMENTS

I thank my project guide, Dr. Shibu Clement, for his constant and ceaseless encouragement for my work as well as for guiding me towards the right approach for learning and knowing how to apply Computational Fluid Dynamics. His rigorous mathematical foundations for fluid mechanics would always continue to be a source of great inspiration for my research.

DISCRETIZATION OF THE GOVERNING EQUATIONS

The generalized transport theorem can be expressed as-

$$\rho \frac{D\phi}{Dt} = del \cdot (del\phi) + S_\phi$$

When this equation is used for the balance of momentum, we get the Navier-Stokes equations.

$$\rho \frac{Du}{Dt} = -(del.p)_x + \mu\,(del^2 u)$$

The convection term in the material derivative of u can be rewritten as

$$\rho \frac{\partial u}{\partial t} + \left(\vec{V} \cdot del\right)u = \rho \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + u\frac{\partial u}{\partial x} + u\frac{\partial v}{\partial y}$$

since the last 2 terms represent the divergence of velocity, which is 0 for incompressible flows.

$$= \rho \frac{\partial u}{\partial t} + del \cdot (\vec{V} u\rho)$$

As outlined in [1], by invoking the divergence theorem the integral form of the Navier-Stokes equation is

$$\int_t^{t+del(t)} \int_{CV} \frac{\partial(\rho u)}{\partial t}\, dV dt + \int_t^{t+del(t)} \oint_{CS} \hat{n} \cdot (\vec{V} u\rho)\, dA dt = \int_t^{t+del(t)} \int_{CV} -\frac{\partial p}{\partial x}\, dV dt + \int_t^{t+del(t)} \oint_{CS} \hat{n} \cdot (\mu del u) dA dt$$

Discretizing as per the Finite Volume Method,

$$\rho \frac{u - u^o}{del(t)}\, del(t) delV + \rho\,(u_e^2 - u_w^2 + u_n v_n - u_s v_s)\, A del(t) = \frac{p_W - p_E}{2dx}\, delV del(t) + \mu \left[\frac{u_E - u}{dx} - \frac{u - u_W}{dx} + \frac{u_N - u}{dy} - \frac{u - u_S}{dy}\right]$$

is the x direction momentum transport equation where the small subscripts denote various quantities at the cell faces while the capital subscripts denote them at the nodes (cell-centres). The superscript o indicates that we are using the value of the variable stored in the previous time-step, which is time-wise backward by del(t). Linearizing is carried out by the Central Differencing scheme with equal grid spacing in the x & y directions.

$$A = h, V = h^2, dx = dy = h$$

$$u_e^2 = u_e^o \frac{u_E + u}{2}$$

$$u_w^2 = u_w^o \frac{u_W + u}{2}$$

Similarly, v at the north & south faces is also calculated by Central Differenced averages from the previous time-step.

In particular, if the objective is to solve for steady flows, the transient term is omitted from the transport equation. We then get,

$$4\mu u_p + \left(\frac{\rho h u_e}{2} - \mu\right)u_E + \left(\frac{-\rho h u_w}{2} - \mu\right)u_W + \left(\frac{\rho h v_n}{2} - \mu\right)u_N + \left(\frac{-\rho h v_s}{2} - \mu\right)u_S = \frac{p_W - p_E}{2}h$$

and

$$4\mu v_p + \left(\frac{\rho h u_e}{2} - \mu\right)v_E + \left(\frac{-\rho h u_w}{2} - \mu\right)v_W + \left(\frac{\rho h v_n}{2} - \mu\right)v_N + \left(\frac{-\rho h v_s}{2} - \mu\right)v_S = \frac{p_S - p_N}{2}h$$

as the x & y direction momentum transport equations. As a further development of this study, we can derive a transient non-dimensional form of the Navier-Stokes equation in a discretized manner. Noting that

$$u^* = \frac{u}{U}, v^* = \frac{v}{U}, x^* = \frac{x}{L}, v^* = \frac{v}{U}, p^* = \frac{p}{\rho U^2}, t^* = \frac{t}{del(t)}$$

where U is the characteristic speed of the flow (in driven-cavity flow, it might be taken as the driving speed of the lid), L is the characteristic length scale (edge of the rectangular domain), del(t) is the time-step of the transient solution, we see that the x direction Navier-Stokes equation looks like

$$\frac{L}{Udel(t)}\frac{\partial u^*}{\partial t^*} + u^*\frac{\partial u^*}{\partial x^*} + v^*\frac{\partial u^*}{\partial y^*} = -\frac{\partial p^*}{\partial x^*} + \frac{\mu}{\rho UL}\left(\frac{\partial^2 u^*}{\partial x^{*2}} + \frac{\partial^2 u^*}{\partial y^{*2}}\right)$$

Integrating over a square faced finite volume,

$$\frac{h^2}{ULdel(t)}\left(u^* - u^{*o}\right) + \frac{h}{L}\left(u_e^{*2} - u_w^{*2} + u_n^* v_n^* - u_s^* v_s^*\right) = \frac{p_W^* - p_E^*}{2}\frac{h}{L} + \frac{1}{Re}\left(u_E^* + u_W^* + u_N^* + u_S^* - 4u^*\right)$$

or in a discretized format (with h*N=L)

$$u^*\left(\frac{h}{NUdel(t)} + \frac{4}{Re}\right) + u^{*o}\left(\frac{-h}{NUdel(t)}\right) + u_E^*\left(\frac{u_e^{*o}}{2N} - \frac{1}{Re}\right) + u_W^*\left(\frac{-u_w^{*o}}{2N} - \frac{1}{Re}\right) + u_N^*\left(\frac{v_n^{*o}}{2N} - \frac{1}{Re}\right) + u_S^*\left(\frac{-v_s^{*o}}{2N} - \frac{1}{Re}\right) = \frac{p_W^* - p_E^*}{2N}$$

becomes the x direction momentum transport equation. Similarly, the y direction Navier-Stokes equation can be expressed as

$$v^*\left(\frac{h}{NUdel(t)}+\frac{4}{Re}\right)+v^{*o}\left(\frac{-h}{NUdel(t)}\right)+v_E^*\left(\frac{u_e^{*o}}{2N}-\frac{1}{Re}\right)+v_W^*\left(\frac{-u_w^{*o}}{2N}-\frac{1}{Re}\right)$$
$$+v_N^*\left(\frac{v_n^{*o}}{2N}-\frac{1}{Re}\right)+v_S^*\left(\frac{-v_s^{*o}}{2N}-\frac{1}{Re}\right)=\frac{p_S^*-p_N^*}{2N}$$

Solving the Navier-Stokes equations is the first step of a complete discrete solution of any flow problem. Here, we make a note of the characteristic features of this Central Differenced solution. The velocity at a node in the grid depends upon the velocities in the same direction of its 4 immediate neighbours and its velocity as well as the control volume face velocities one time-step earlier.

MODIFIED PRESSURE CORRECTION AND UNDER-RELAXATION

As per [3], the continuity equation manifests itself as the pressure correction equation Patankar's SIMPLE algorithm. The essence of the SIMPLE method is that the errors of the momentum transport equations are linearly expressed and for computing velocity errors at a point in the grid, the corresponding velocity errors of its neighbours are considered to be zero. This crucial assumption of the SIMPLE method is not assumed in the algorithm presented here but instead, we follow the procedure below. The mass conservation principle for an incompressible flow is

$$\iiint_{CV} del \cdot \vec{V}\, dV = \iint_{CS} \hat{n} \cdot \vec{V}\, dA = 0$$

For equal grid spacing in x & y directions,

$$u_e - u_w + v_n - v_s = 0$$

and with central differencing,

$$u_E - u_W + v_N - v_S = 0$$

By taking linear errors from the discretized momentum transport equations, we achieve the following 3 results-

Dimensional form without assuming steady-state

$$u'\left(\rho h^2 + 4\mu del(t)\right) + u^{o\prime}\left(-\rho h^2\right) + u'_E\left(\frac{\rho h del(t) u_e^o}{2} - \mu del(t)\right) + u'_W\left(-\frac{\rho h del(t) u_w^o}{2} - \mu del(t)\right) + u'_N\left(\frac{\rho h del(t) v_n^o}{2} - \mu del(t)\right) + u'_S\left(-\frac{\rho h del(t) v_s^o}{2} - \mu del(t)\right) = \frac{p'_W - p'_E}{2} h del(t)$$

where the primes signify the errors associated with that particular iteration.

Now, for all the neighbouring velocities (i.e.- uE, uW, uN and uS), we apply this error equation and we observe that for each control volume only the RHS of the equation changes as per the local pressure gradients. So,

$$u'_E\left(\rho h^2 + 4\mu del(t)\right) = (p'_P - p'_E) h del(t)$$

and rearranging,

$$u'_E = \frac{(p'_P - p'_E) h del(t)}{\left(\rho h^2 + 4\mu del(t)\right)}$$

and for other neighbouring velocities (from the y direction error-in-momentum-transport equation),

$$u'_W = \frac{(p'_W - p'_P)hdel(t)}{\left(\rho h^2 + 4\mu del(t)\right)} \qquad v'_S = \frac{(p'_S - p'_P)hdel(t)}{\left(\rho h^2 + 4\mu del(t)\right)}$$

$$v'_N = \frac{(p'_P - p'_N)hdel(t)}{\left(\rho h^2 + 4\mu del(t)\right)}$$

So, the continuity equation for the next iteration becomes

$$u_E - u_W + v_N - v_S + u'_E - u'_W + v'_N - v'_S = 0$$

And substituting,

$$u_E - u_W + v_N - v_S + \frac{(p'_P - p'_E)hdel(t)}{\left(\rho h^2 + 4\mu del(t)\right)} - \frac{(p'_W - p'_P)hdel(t)}{\left(\rho h^2 + 4\mu del(t)\right)} + \frac{(p'_P - p'_N)hdel(t)}{\left(\rho h^2 + 4\mu del(t)\right)} - \frac{(p'_S - p'_P)hdel(t)}{\left(\rho h^2 + 4\mu del(t)\right)} = 0$$

Solving for error in pressure terms,

$$4p'_P - p'_E - p'_E - p'_N - p'_S = \frac{-(u_E - u_W + v_N - v_S)\left(\rho h^2 + 4\mu del(t)\right)}{hdel(t)}$$

The corresponding velocity corrector equations for every point in the grid are

$$u' = \frac{(p'_W - p'_E)hdel(t)}{2\left(\rho h^2 + 4\mu del(t)\right)}$$

$$v' = \frac{(p'_S - p'_N)hdel(t)}{2\left(\rho h^2 + 4\mu del(t)\right)}$$

Dimensional form by assuming steady-state conditions

$$4p'_P - p'_E - p'_E - p'_N - p'_S = \frac{-(u_E - u_W + v_N - v_S)\mu}{h}$$

$$u' = \frac{(p'_W - p'_E)h}{4\mu}$$

$$v' = \frac{(p'_S - p'_N)h}{4\mu}$$

Non-dimensional form for transient solutions

$$u'_E = \frac{\frac{p^{*\prime} - p^{*\prime}_E}{N}}{\frac{h}{NUdel(t)} + \frac{4}{Re}}$$

$$u'_W = \frac{\frac{-p^{*'} + p^{*'}_W}{N}}{\frac{h}{NUdel(t)} + \frac{4}{Re}} \qquad v'_S = \frac{\frac{-p^{*'} - p^{*'}_S}{N}}{\frac{h}{NUdel(t)} + \frac{4}{Re}}$$

$$v'_N = \frac{\frac{p^{*'} - p^{*'}_N}{N}}{\frac{h}{NUdel(t)} + \frac{4}{Re}}$$

The modified continuity equation appears as

$$4p'_P - p'_E - p'_E - p'_N - p'_S = -N(u_E - u_W + v_N - v_S)\left(\frac{h}{NUdel(t)} + \frac{4}{Re}\right)$$

with the velocity correctors as

$$u' = \frac{\frac{p^{*'}_W - p^{*'}_E}{N}}{2\left(\frac{h}{NUdel(t)} + \frac{4}{Re}\right)}$$

$$v' = \frac{\frac{p^{*'}_S - p^{*'}_N}{N}}{2\left(\frac{h}{NUdel(t)} + \frac{4}{Re}\right)}$$

The common pattern being followed here in this algorithm is that while SIMPLE neglected the errors of neighbouring terms while calculating for the continuity/pressure-corrector equation, we are in fact doing exactly the opposite. We are using the error in momentum transport equations of the neighbouring cells and using these velocity errors in the continuity equation. The following step given in [1], [2] is clearly ***not*** being followed in this algorithm

$$a_p u'_p = \sum a_{neighbors} u'_{neighbors} + a_{pressures}\frac{p'_E - p'_W}{2}$$

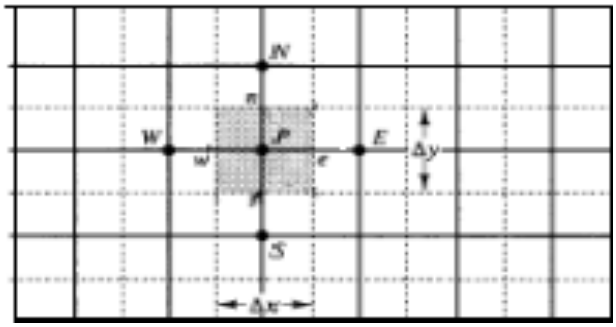
$$a_p v'_p = \sum a_{neighbors} \phi'_{neighbors} + a_{pressures}\frac{p'_N - p'_S}{2}$$

The errors calculated for the pressure and the velocity terms are then under-relaxed and incremented to the existing estimates. Under-relaxation is necessary for solution stability and to protect against divergence even in high viscosity cases.

$$u, v_{next\ iteration} = u, v_{old} + relax * u, v'_{error\ of\ old\ iteration}$$

$$p_{next\ iteration} = p_{old} + relax * p'_{error\ of\ old\ iteration}$$

SUBROUTINE DESCRIPTIONS



The above diagram shows a representation of how a rectangular domain is discretized into finite volumes with the terminology being used for neighbouring cells and cell faces. For a complete solver, we first construct a square grid of side L with N cells of uniform x & y grid spacing of h respectively. After setting the boundary conditions (which are Dirichlet in the driven-cavity problem with the only non-zero condition being the lid velocity at the north edge) and setting the initial guesses of u, v & p = 0 for all interior nodes of the domain, we can begin iterating. a subroutine node** creategrid (void) constructs the grid, sets the boundary conditions, allows the user to define fluid properties and U and returns the array of nodes in a double referenced format to the main function. Next, for each call of the subroutine node** traversal (node** grid), the traversal function performs the following operations to the grid. First, for every point in the interior, the x & y direction Navier-Stokes equations are written. Thus, we have 7 coefficients per point- of itself, its 4 neighbouring velocities, the RHS of the x direction equation and the RHS of the y direction equation.

Now, we use an Alternating Direction Implicit version of these equations so that we first take one horizontal line of the grid at a time, assume the north and south neighbouring velocities to be known from the previous iterations' values and take them over to the RHS. Hence, for every horizontal line, we obtain a tridiagonal matrix of coefficients. In the next sweep, which is eastward instead of southward, we take one vertical line at a time, assume to east and west values to be known from the previous iteration and again acquire a tridiagonal system of coefficient (wherein the equation at a node has just 3 unknowns- its own velocity and that of its two neighbours- either E&W or N&S). For each line of the grid, we solve this linear system of algebraic equations with the Tridiagonal Matrix Algorithm [5] as per the C++ algorithm outlined at the end of this section. So, after two southward and eastward sweeps, we approximately solve the entire set of discretized equations for the grid.

The two subroutines float* xNSeqn (node n) & float* yNSeqn (node n) accept a given node of the grid and return a vector array of numbers which are the coefficients of the Navier-Stokes equations for the neighbouring cells. 3 of these are fed into the TDMA solver subroutine float* tridiagmatrix (float **A) which accepts a matrix and returns a vector array of solutions. Finally, the pressure corrector equations are solved in a similar fashion for each cell and these errors are incremented to the existing values in an under-relaxed form.

```
using namespace std;
float* TridiagMatrix (float **A)
{
//Solves a tridiagonal matrix for N-1 equations & N-1 unknowns, so it can be readily used for traversal provided the matrices passed are of sizes equal to the TridiagMatrix declaration.
int p, q;
float *soln = new float [N];
soln[0]=0;
//Forming sparse matrix
for (p=0; p<N; p++)
{A[p][0]=0;}
for (q=0; q<5; q++)
{A[0][q]=0;}
A[1][3]=0;
A[N-1][1]=0;
//Converting to upper triangular matrix
//1st line of matrix
A[2][2]= A[2][2]+ (A[1][2]*(-A[2][1])/A[1][1]);
A[2][4]= A[2][4]+ (A[1][4]*(-A[2][1])/A[1][1]);
A[2][1]= A[2][1]+ (A[1][1]*(-A[2][1])/A[1][1]);
for (p=3; p<(N-1); p++)
{
        A[p][2]= A[p][2]+ (A[p-1][3]*(-A[p][1])/A[p-1][2]);
        A[p][4]= A[p][4]+ (A[p-1][4]*(-A[p][1])/A[p-1][2]);
        A[p][1]= A[p][1]+ (A[p-1][2]*(-A[p][1])/A[p-1][2]);
}
//N-1 line of matrix
A[N-1][3]= A[N-1][3]+ (A[N-2][3]*(-A[N-1][2])/A[N-2][2]);
A[N-1][4]= A[N-1][4]+ (A[N-2][4]*(-A[N-1][2])/A[N-2][2]);
A[N-1][2]= A[N-1][2]+ (A[N-2][2]*(-A[N-1][2])/A[N-2][2]);
//Solving linear equations
soln[N-1]= A[N-1][4]/A[N-1][3];
for (p=(N-2); p>1; p--)
{
        soln[p]= (A[p][4] - (A[p][3] * soln[p+1])) / A[p][2];
}
soln[1]= (A[1][4] - A[1][2]*soln[2])/A[1][1];
return soln;
}
```

RESULTS AND ANALYSIS

For a driven-cavity flow, the following results were obtained for a 10*10 grid with north boundary velocity U=2m/s, grid spacing h=0.001m, density rho=1216kg/m^3 and dynamic viscosity mu=1.2Pas (for glycerol, a suitably viscous fluid)

u values-

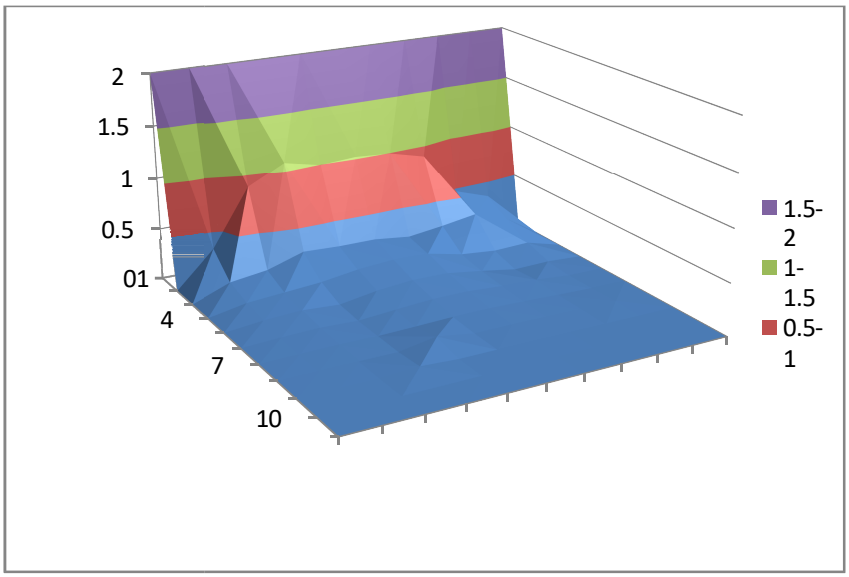
2	2	2	2	2	2		2	2	2	2
0	0.321627	0.890446	1.07252	0.997579	1.0196	0.99258	0.90669	0.465087	0.358013	0
0	0.0325	0.153907	0.22613	0.223347	0.185481	0.10278	0.14437	0.25487	0.197274	0
0	0.009034	0.020734	0.054214	0.070335	0.076859	0.075757	0.066937	0.046232	0.0219499	0
0	-0.0125	-0.01158	-0.00313	-0.00381	-0.00812	-0.0101	-0.0103	-0.00887	-0.0083654	0
0	-0.00837	-0.01288	-0.01283	-0.01126	-0.00944	-0.00761	-0.00551	-0.02731	-0.00139	0
0	-0.00464	-0.00844	-0.01006	-0.01033	-0.00983	-0.00867	-0.00705	-0.00449	0	0
0	-0.00234	-0.00463	-0.00593	-0.00652	-0.00065	-0.00597	-0.00495	-0.00336	-0.011658	0
0	-0.00103	-0.00209	-0.00278	-0.00312	-0.00317	-0.00296	-0.0025	-0.00172	0	0
0	-0.0005	-0.00105	-0.00142	-0.0015	-0.00151	-0.00153	-0.00126	-0.00081	0	0
0	0	0	0	0	0	0	0	0	0	0

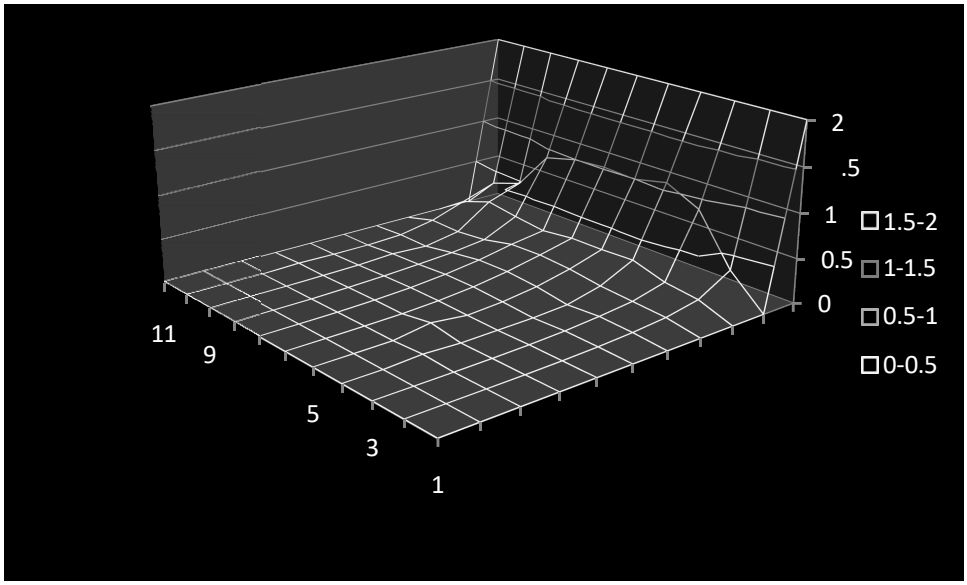
v values-

0	0	0	0	0	0	0	0	0	0	0
0	0.13142	0.113977	-0.08285	-0.01731	-0.03881	-0.08736	-0.03721	-0.08836	-0.00265	0
0	0.135443	0.108075	-0.19666	-0.17577	-0.21449	-0.2196	-0.24798	-0.22808	-0.1072	0
0	0.068355	0.093729	-0.1102	-0.15835	-0.18308	-0.17258	-0.08913	-0.06331	-0.08829	0
0	0.034192	0.05516	0.034146	-0.09089	-0.12118	-0.11594	-0.06589	-0.00374	-0.01603	0
0	0.01496	0.02475	-0.0073	-0.04106	-0.06425	-0.06574	-0.04293	-0.01424	0	0
0	0.0054	0.009458	-0.00163	-0.01661	-0.02898	-0.03171	-0.02308	-0.01066	-0.00327	0
0	0.001291	0.002937	0	-0.06644	-0.01186	-0.01349	-0.01044	-0.00541	-0.00205	0
0	0	0	-0.02371	-0.00422	-0.00473	-0.00356	-0.001	0	0	0
0	0	0	0	-0.0021	-0.00236	-0.00128	0	0	0	0

Total velocity vector magnitudes for graphical analysis-

10	2	2	2	2	2	2	2	2	2	2	2
9	0	0.347441	0.897711	1.075715	0.997729	1.020338	0.996417	0.907453	0.473405	0.358023	0
8	0	0.139288	0.188063	0.299682	0.284216	0.283565	0.242464	0.286944	0.342025	0.224519	0
7	0	0.068949	0.095995	0.122812	0.173263	0.198559	0.188477	0.111466	0.078394	0.090981	0
6	0	0.036404	0.056362	0.034289	0.09097	0.121452	0.116379	0.066689	0.009624	0.018084	0
5	0	0.017142	0.027899	0.014764	0.042576	0.064936	0.066179	0.043284	0.0308	0.00139	0
4	0	0.007118	0.012676	0.010191	0.01956	0.030602	0.032878	0.024132	0.011563	0.003274	0
3	0	0.002673	0.005483	0.00593	0.066759	0.01188	0.014754	0.011557	0.006366	0.011837	0
2	0	0.00103	0.00209	0.023875	0.005248	0.005696	0.004632	0.002695	0.001717	0	0
1	0	0.0005	0.00105	0.00142	0.002581	0.002802	0.001995	0.001256	0.000808	0	0
0	0	0	0	0	0	0	0	0	0	0	0
y/x	0	1	2	3	4	5	6	7	8	9	10





These results show that the velocity profile obtained is fairly accurate with the published results of driven-cavity flow.

Modifying the pressure correctors leads to a faster convergence (it = 50-70) with S edge flow producing a sufficient amount of fluid swirling. The drawbacks are relatively high computational memory storage (2 calculation steps per iteration) and instability at Re > 1500.

Further explorations could be undertaken for transient flow analysis using semi-implicit form.

APPENDIX I- C++ IMPLEMENTATION

CREATEGRID.CPP

```
#include <stdlib.h>
using namespace std;
int h=0.001;
int N= 10;
float mu, rho;
class node
{
 private:
 float u, v, p;
 public:
 node ()
 {
  setu (0);
  setv (0);
  setp (0);
 }
 float getp ()
 {return p;}
 void setp (float pval)
 {if (pval>=0.0000001 || pval<=-0.0000001){p = pval;}
  else p=0;}
 float getu ()
 {return u;}
 float getv ()
 {return v;}
 void setu (float uval)
 {if (uval>= 0.001 || uval<=-0.001) {u = uval;}
  else u=0;}
 void setv (float vval)
 {if (vval>= 0.001 || vval<=-0.001) {v = vval;}
  else v=0;}
 node *loceast, *locwest, *locnorth, *locsouth;
};
node** creategrid ()
{
 cout << "Enter viscosity\n";
 cin >> mu;
 cout << "Enter density\n";
 cin >> rho;
 int i,j;
 //Dynamic Memory Allocation for the grid
 node **grid;
 grid = new node* [N+1];
 for (i=0; i<=N; i++)
 {
  *(grid + i) = new node [N+1];
 }
 //Laying connections for inner nodes
 for (i=1; i<N; i++)
 {
  for (j=1; j<N; j++)
  {
            grid[i][j].loceast= &grid[i+1][j];
            grid[i][j].locwest= &grid[i-1][j];
            grid[i][j].locnorth= &grid[i][j+1];
            grid[i][j].locsouth= &grid[i][j-1];
  }
 }
 cout << "Grid created and grid connections made.\n";
float uBCwest=0, vBCwest=0, uBCsouth=0, vBCsouth=0,
uBCeast=0, vBCeast=0, uBCnorth=0, vBCnorth=0;
float pBCwest=0, pBCsouth=0, pBCeast=0, pBCnorth=0;
cout << "Enter North Boundary Condition x direction
velocity\n";
cin >> uBCnorth;
//North BC
for (i=0; i<=N; i++)
{
 grid[i][N].setu(uBCnorth);
 grid[i][N].setv(vBCnorth);
 grid[i][N].setp(pBCnorth);
}
//South BC
for (i=0; i<=N; i++)
{
 grid[i][0].setu(uBCsouth);
 grid[i][0].setv(vBCsouth);
 grid[i][0].setp(pBCsouth);
}
//West BC
for (j=1; j<=(N-1); j++)
{
 grid[0][j].setu(uBCwest);
 grid[0][j].setv(vBCwest);
 grid[0][j].setp(pBCwest);
}
//East BC
for (j=1; j<=(N-1); j++)
{
 grid[N][j].setu(uBCeast);
 grid[N][j].setv(vBCeast);
 grid[N][j].setp(pBCeast);
}
cout << "Boundary Conditions taken. North-\n";
for (i=0; i<=N; i++)
{
            cout<< grid[i][N].getu()<<" ";
}
cout<<"\n";
for (i=0; i<=N; i++)
{
            cout<< grid[i][N].getv()<<" ";
}
cout<<"\n";
for (i=0; i<=N; i++)
{
            cout<< grid[i][N].getp()<<" ";
}
cout<<"\nSouth-\n";
for (i=0; i<=N; i++)
{
            cout<< grid[i][0].getu()<<" ";
}
cout<<"\n";
for (i=0; i<=N; i++)
{
            cout<< grid[i][0].getv()<<" ";
}
cout<<"\n";
for (i=0; i<=N; i++)
{
            cout<< grid[i][0].getp()<<" ";
}
cout<<"\nEast-\n";
for (j=1; j<=(N-1); j++)
{
            cout<< grid[N][j].getu()<<" ";
}
cout<<"\n";
for (j=1; j<=(N-1); j++)
{
            cout<< grid[N][j].getv()<<" ";
}
```

```
cout<<"\n";
for (j=1; j<=(N-1); j++)
{
		cout<< grid[N][j].getp()<<" ";
}
cout<<"\nWest-\n";
for (j=1; j<=(N-1); j++)
{
		cout<< grid[0][j].getu()<<" ";
}
cout<<"\n";
for (j=1; j<=(N-1); j++)
{
		cout<< grid[0][j].getv()<<" ";
}
cout<<"\n";
for (j=1; j<=(N-1); j++)
{
		cout<< grid[0][j].getp()<<" ";
}
cout<<"\n";
return grid;
}
```

XNSEQN.CPP

```
using namespace std;
float* xNSeqn (node n)
{
//Remember (ue-uw+vn-vs)=0 by continuity; we use
incompressibility here too
float uat, uE, uW, ue, uw, vn, vs, vat, vN, vS, pE, pW, uN, uS;
float coeffat, coeffE, coeffW, coeffN, coeffS, srcterm;
float* xeqn;
xeqn= new float[6];
uat= n.getu();
vat= n.getv();
pE= n.loceast->getp();
pW= n.locwest->getp();
uE= n.loceast->getu();
uN= n.locnorth->getu();
uS= n.locsouth->getu();
uW= n.locwest->getu();
vN= n.locnorth->getv();
vS= n.locsouth->getv();
//Using Central Differencing
ue= (uE+uat)/2;
uw= (uW+uat)/2;
vn= (vat+vN)/2;
vs= (vat+vS)/2;
coeffat= 4*mu;
coeffE= -mu+rho*h*ue/2;
coeffW= -mu-rho*h*uw/2;
coeffN= -mu+rho*h*vn/2;
coeffS= -mu-rho*h*vs/2;
srcterm= (pW-pE)*h/2;
//Order of arranging in equation- at,E,W,N,S,src
*xeqn= coeffat;
*(xeqn+1)= coeffE;
*(xeqn+2)= coeffW;
*(xeqn+3)= coeffN;
*(xeqn+4)= coeffS;
*(xeqn+5)= srcterm;
return xeqn;
}
```

YNSEQN.CPP

```
using namespace std;
float* yNSeqn (node n)
{
//Remember (ue-uw+vn-vs)=0 by continuity; we use
incompressibility here too
float uat, uE, uW, ue, uw, vn, vs, vat, vN, vS, pN, pS, vE, vW;
float coeffat, coeffE, coeffW, coeffN, coeffS, srcterm;
float* yeqn;
yeqn= new float[6];
uat= n.getu();
vat= n.getv();
pN= n.locnorth->getp();
pS= n.locsouth->getp();
uE= n.loceast->getu();
uW= n.locwest->getu();
vN= n.locnorth->getv();
vS= n.locsouth->getv();
vE= n.loceast->getv();
vW= n.locwest->getv();
//Using Central Differencing
ue= (uE+uat)/2;
uw= (uW+uat)/2;
vn= (vat+vN)/2;
vs= (vat+vS)/2;
coeffat= 4*mu;
coeffE= -mu+rho*h*ue/2;
coeffW= -mu-rho*h*uw/2;
coeffN= -mu+rho*h*vn/2;
coeffS= -mu-rho*h*vs/2;
srcterm= (pS-pN)*h/2;
//Order of arranging in equation- at,E,W,N,S,src
*yeqn= coeffat;
*(yeqn+1)= coeffE;
*(yeqn+2)= coeffW;
*(yeqn+3)= coeffN;
*(yeqn+4)= coeffS;
*(yeqn+5)= srcterm;
return yeqn;
}
```

CONTINUITYEQN.CPP

```
using namespace std;
float* continuityeqn (node n)
{
float uE, uW, vN, vS;
float coeffat, coeffE, coeffW, coeffN, coeffS, srcterm;
float* conteqn = new float[6];
uE= n.loceast->getu();
uW= n.locwest->getu();
vN= n.locnorth->getv();
vS= n.locsouth->getv();
srcterm= (-uE+uW-vN+vS)*mu/0.001;
coeffat= 1;
coeffE=coeffW=coeffN=coeffS= -0.25;
//Order of arranging in equation- at,E,W,N,S,src
*conteqn= coeffat;
*(conteqn+1)= coeffE;
*(conteqn+2)= coeffW;
*(conteqn+3)= coeffN;
*(conteqn+4)= coeffS;
*(conteqn+5)= srcterm;
return conteqn;
}
```

TRAVERSAL.CPP

```
using namespace std;
node** traversal (node **grid)
{
int i, j, k, p, q;
float *xeqn, *yeqn, *conteqn;
float **mat;
float errp, prevp, prevu, prevv;
//Dynamic Memory Allocation for sparse matrix
```

```
mat= new float* [N];
for (k=0; k<N; k++)
{
 mat[k]= new float[5];
}
//Forming sparse matrix
for (p=0; p<N; p++)
{mat[p][0]=0;}
for (q=0; q<5; q++)
{mat[0][q]=0;}
mat[1][3]=0;
mat[N-1][1]=0;
float* matsoln;
matsoln= new float [N];
matsoln[0]=0;
//-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------
//Row wise u iterations, marching direction- South
for (j=(N-1); j>0; j--)
{
 cout<<"Row no. (y coordinate)- "<<j<<"\n";
 //For a single row of the grid (constant y), we form the
tridiagonal matrix.
 //1st line of matrix
 xeqn= xNSeqn(grid[1][j]);
 mat[1][1]= xeqn[0];
 mat[1][2]= xeqn[1];
 mat[1][4]= xeqn[5] - xeqn[3]*(grid[1][j].locnorth)->getu() -
xeqn[4]*(grid[1][j].locsouth)->getu() -
xeqn[2]*(grid[1][j].locwest)->getu();
 //Filling up 2nd to N-2 lines of matrix
 for (i=2, p=2; i<=(N-2), p<=(N-2); i++, p++)
 {
  xeqn= xNSeqn(grid[i][j]);
  mat[p][1]= xeqn[2];
  mat[p][2]= xeqn[0];
  mat[p][3]= xeqn[1];
  mat[p][4]= xeqn[5] - xeqn[3]*(grid[i][j].locnorth)->getu() -
xeqn[4]*(grid[i][j].locsouth)->getu();
 }
 //N-1 line of matrix
 xeqn= xNSeqn(grid[N-1][j]);
 mat[N-1][2]= xeqn[2];
 mat[N-1][3]= xeqn[0];
 mat[N-1][4]= xeqn[5] - xeqn[3]*(grid[N-1][j].locnorth)->getu()
- xeqn[4]*(grid[N-1][j].locsouth)->getu() - xeqn[1]*(grid[N-
1][j].loceast)->getu();
 cout<<"First sweep for u- coefficient matrix check\n";
 for (p=1; p<N; p++)
 {
            for (q=1; q<5; q++)
            {
                        cout<<mat[p][q]<<" ";
            }
            cout<<"\n";
 }
 matsoln= TridiagMatrix(mat);
  for (i=1, k=1; i<N, k<N; i++, k++)
 {
  grid[i][j].setu(matsoln[k]);
 }
 cout<<"First sweep for u- solution assignment check\n";
 for (i=1; i<N; i++)
 {cout<<grid[i][j].getu()<<" ";}
 cout<<"\n";
 //Now we iterate for the next row of the grid downwards.
}
//Column wise u iterations, marching direction- East
for (i=1; i<N; i++)
{
 cout<< "Column no. (x coordinate) "<<i<<"\n";
 //For a single column of the grid (constant x), we form the
tridiagonal matrix.
 //1st line of matrix
 xeqn= xNSeqn(grid[i][N-1]);
 mat[1][1]= xeqn[0];
 mat[1][2]= xeqn[4];
 mat[1][4]= xeqn[5] - xeqn[1]*grid[i][N-1].loceast->getu() -
xeqn[2]*grid[i][N-1].locwest->getu() - xeqn[3]*grid[i][N-
1].locnorth->getu();
 //Filling up 2nd to N-2 lines of matrix
 for (j=(N-2), p=2; j>=2, p<=(N-2); j--, p++)
 {
            xeqn= xNSeqn(grid[i][j]);
            mat[p][1]= xeqn[3];
  mat[p][2]= xeqn[0];
  mat[p][3]= xeqn[4];
  mat[p][4]= xeqn[5] - xeqn[1]*grid[i][j].loceast->getu() -
xeqn[2]*grid[i][j].locwest->getu();
 }
 //N-1 line of matrix
 xeqn= xNSeqn(grid[i][1]);
 mat[N-1][2]= xeqn[3];
 mat[N-1][3]= xeqn[0];
 mat[N-1][4]= xeqn[5] - xeqn[1]*grid[i][1].loceast->getu() -
xeqn[2]*grid[i][1].locwest->getu() - xeqn[4]*grid[i][1].locsouth-
>getu();
 cout<<"Second sweep for u- coefficient matrix check\n";
 for (p=1; p<N; p++)
 {
            for (q=1; q<5; q++)
            {
                        cout<<mat[p][q]<<" ";
            }
            cout<<"\n";
 }
 matsoln= TridiagMatrix(mat);
  for (j=(N-1), k=1; j>0, k<N; j--, k++)
 {
            grid[i][j].setu(matsoln[k]);
 }
 cout<<"Second sweep for u- solution assignment check\n";
 for (j=(N-1); j>=1; j--)
 {cout<<grid[i][j].getu()<<" ";}
 cout<<"\n";
 //Now we iterate for the next column of the grid eastward.
}
//-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------
//Row wise errp iterations, marching direction- South
for (j=(N-1); j>0; j--)
{
 cout<<"Row no. (y coordinate)- "<<j<<"\n";
 //1st line of matrix
 conteqn= continuityeqn(grid[1][j]);
 mat[1][1]= conteqn[0];
 mat[1][2]= conteqn[1];
 //Error in pressure is assumed to be 0 for calculation from
previous iteration.
 mat[1][4]= conteqn[5];
 //Filling up 2nd to N-2 lines of matrix
 for (i=2, p=2; i<=(N-2), p<=(N-2); i++, p++)
 {
  conteqn= continuityeqn(grid[i][j]);
  mat[p][1]= conteqn[2];
  mat[p][2]= conteqn[0];
  mat[p][3]= conteqn[1];
  mat[p][4]= conteqn[5];
 }
```

```
 //N-1 line of matrix
 conteqn= continuityeqn(grid[N-1][j]);
 mat[N-1][2]= conteqn[2];
 mat[N-1][3]= conteqn[0];
 mat[N-1][4]= conteqn[5];
  cout<<"First sweep for errp- coefficient matrix check\n";
 for (p=1; p<N; p++)
 {
                for (q=1; q<5; q++)
                {
                        cout<<mat[p][q]<<" ";
                }
                cout<<"\n";
 }
 matsoln= TridiagMatrix(mat);

 cout<<"\nIntermediate errp check-\n";
 for (k=1; k<=(N-1); k++)
 {
                cout<<matsoln[k]<<" ";
 }
//p Corrector eqn-
 for (i=1, k=1; i<N, k<N; i++, k++)
 {
               errp=matsoln[k];
               prevp= grid[i][j].getp();
   grid[i][j].setp(prevp+errp);
 }
 cout<<"First sweep for p- solution assignment check\n";
 for (i=1; i<N; i++)
 {cout<<grid[i][j].getp()<<" ";}
 cout<<"\n";
//u Corrector eqns-
 prevu= grid[1][j].getu();
 grid[1][j].setu(prevu+matsoln[2]*0.001/(-8*mu));
 prevu= grid[N-1][j].getu();
 grid[N-1][j].setu(prevu+matsoln[N-2]*0.001/(8*mu));
 for (i=2, k=2; i<=(N-2), k<=(N-2); i++, k++)
 {
               prevu=grid[i][j].getu();
   grid[i][j].setu(prevu+ (matsoln[k+1]-matsoln[k-1])*0.001/(-
8*mu));
 }
 cout<<"u with increment- check\n";
 for (i=1; i<N; i++)
 {cout<<grid[i][j].getu()<<" ";}
 cout<<"\n";
}
//Column wise errp iterations, marching direction-East
for (i=1; i<N; i++)
{
 cout<< "Column no. (x coordinate) "<<i<<"\n";
 //1st line of matrix
 conteqn= continuityeqn(grid[i][N-1]);
 mat[1][1]= conteqn[0];
 mat[1][2]= conteqn[4];
 mat[1][4]= conteqn[5];
 //Filling up 2nd to N-2 lines of matrix
 for (j=(N-2), p=2; j>=2, p<=(N-2); j--, p++)
 {
  conteqn= continuityeqn(grid[i][j]);
  mat[p][1]= conteqn[3];
  mat[p][2]= conteqn[0];
  mat[p][3]= conteqn[4];
  mat[p][4]= conteqn[5];
 }
 //N-1 line of matrix
 conteqn= continuityeqn(grid[i][1]);
 mat[N-1][2]= conteqn[3];
 mat[N-1][3]= conteqn[0];
 mat[N-1][4]= conteqn[5];
 cout<<"Second sweep for errp- coefficient matrix check\n";
 for (p=1; p<N; p++)
 {
                for (q=1; q<5; q++)
                {
                        cout<<mat[p][q]<<" ";
                }
                cout<<"\n";
 }
 matsoln= TridiagMatrix(mat);
  cout<<"\nIntermediate errp check-\n";
 for (k=1; k<=(N-1); k++)
 {
                cout<<matsoln[k]<<" ";
 }
//p Corrector eqn-
 for (j=(N-1), k=1; j>0, k<N; j--, k++)
 {
               errp= matsoln[k];
               prevp= grid[i][j].getp();
  grid[i][j].setp(prevp+errp);
 }
  cout<<"Second sweep for p- solution assignment check\n";
 for (j=(N-1); j>=1; j--)
 {cout<<grid[i][j].getp()<<" ";}
 cout<<"\n";
//v Corrector eqns-
 prevv= grid[i][N-1].getv();
 grid[i][N-1].setv(prevv+matsoln[2]*0.001/(8*mu));
 prevv= grid[i][1].getv();
 grid[i][1].setv(prevv+matsoln[N-2]*0.001/(-8*mu));
 for (j=(N-2), k=2; j>=2, k<=(N-2); j--, k++)
 {
               prevv= grid[i][j].getv();
  grid[i][j].setv(prevv+ (matsoln[k-1]-matsoln[k+1])*0.001/(-
8*mu));
 }
 cout<<"v with increment- check\n";
 for (j=(N-1); j>=1; j--)
 {cout<<grid[i][j].getv()<<" ";}
 cout<<"\n";
}
//-------------------------------------------------------------------------
-------------------------------------------------------------------------
--------
//Row wise v iterations, marching direction- East
for (j=(N-1); j>0; j--)
{
 cout<<"Row no. (y coordinate)- "<<j<<"\n";
  //For a single row of the grid (constant y), we form the
tridiagonal matrix.
  //1st line of matrix
  yeqn= yNSeqn(grid[1][j]);
  mat[1][1]= yeqn[0];
  mat[1][2]= yeqn[1];
  mat[1][4]= yeqn[5] - yeqn[3]*grid[1][j].locnorth->getv() -
yeqn[4]*grid[1][j].locsouth->getv() - yeqn[2]*grid[1][j].locwest-
>getv();
  //Filling up 2nd to N-2 lines of matrix
  for (i=2, p=2; i<=(N-2), p<=(N-2); i++, p++)
  {
   yeqn= yNSeqn(grid[i][j]);
   mat[p][1]= yeqn[2];
   mat[p][2]= yeqn[0];
   mat[p][3]= yeqn[1];
   mat[p][4]= yeqn[5] - yeqn[3]*grid[i][j].locnorth->getv() -
yeqn[4]*grid[i][j].locsouth->getv();
  }
 //N-1 line of matrix
  yeqn= yNSeqn(grid[N-1][j]);
  mat[N-1][2]= yeqn[2];
```

```
 mat[N-1][3]= yeqn[0];
 mat[N-1][4]= yeqn[5] - yeqn[3]*grid[N-1][j].locnorth->getv() -
yeqn[4]*grid[N-1][j].locsouth->getv() - yeqn[1]*grid[N-
1][j].loceast->getv();
 cout<<"First sweep for v- coefficient matrix check\n";
 for (p=1; p<N; p++)
 {
                for (q=1; q<5; q++)
                {
                        cout<<mat[p][q]<<" ";
                }
                cout<<"\n";
 }
 matsoln= TridiagMatrix(mat);
 for (i=1, k=1; i<N, k<N; i++, k++)
 {
  grid[i][j].setv(matsoln[k]);
 }
  cout<<"First sweep for v- solution assignment check\n";
 for (i=1; i<N; i++)
 {cout<<grid[i][j].getv()<<" ";}
 cout<<"\n";
 //Now we iterate for the next row of the grid downwards.
}
//Column wise v iterations, marching direction- South
//For a single column wise (constant y) of the grid, we form the
tridiagonal matrix
for (i=1; i<N; i++)
{
 cout<< "Column no. (x coordinate) "<<i<<"\n";
 //1st line of matrix
 yeqn= yNSeqn(grid[i][N-1]);
 mat[1][1]= yeqn[0];
 mat[1][2]= yeqn[4];
 mat[1][4]= yeqn[5] - yeqn[1]*grid[i][N-1].loceast->getv() -
yeqn[2]*grid[i][N-1].locwest->getv() - yeqn[3]*grid[i][N-
1].locnorth->getv();
 //Filling up 2nd to N-2 lines of matrix
 for (j=(N-2), p=2; j>=2, p<=(N-2); j--, p++)
 {
                yeqn= yNSeqn(grid[i][j]);
                mat[p][1]= yeqn[3];
   mat[p][2]= yeqn[0];
   mat[p][3]= yeqn[4];
   mat[p][4]= yeqn[5] - yeqn[1]*grid[i][j].loceast->getv() -
yeqn[2]*grid[i][j].locwest->getv();
 }
 //N-1 line of matrix
 yeqn= yNSeqn(grid[i][1]);
 mat[N-1][2]= yeqn[3];
 mat[N-1][3]= yeqn[0];
 mat[N-1][4]= yeqn[5] - yeqn[1]*grid[i][1].loceast->getv() -
yeqn[2]*grid[i][1].locwest->getv() - yeqn[4]*grid[i][1].locsouth-
>getv();
 cout<<"Second sweep for v- coefficient matrix check\n";
 for (p=1; p<N; p++)
 {
                for (q=1; q<5; q++)
                {
                        cout<<mat[p][q]<<" ";
                }
                cout<<"\n";
 }
 matsoln= TridiagMatrix(mat);

 for (j=(N-1), k=1; j>0, k<N; j--, k++)
 {
                grid[i][j].setv(matsoln[k]);
 }
 cout<<"Second sweep for v- solution assignment check\n";
 for (j=(N-1); j>=1; j--)
 {cout<<grid[i][j].getv()<<" ";}
 cout<<"\n";
}
return grid;
}
```

TANAY'SSOLVER.CPP

```
#include<iostream>
#include<cmath>
using namespace std;
#include "creategrid.cpp"
#include "xNSeqn.cpp"
#include "yNSeqn.cpp"
#include "continuityeqn.cpp"
#include "TridiagMatrix.cpp"
#include "Traversal.cpp"
int main()
{
int n, it;
cout << "Enter no. of iterations to be processed\n";
cin >> it;
node** grid;
grid= creategrid();
for (n=0; n<it; n++)
{
cout<< "Iteration no.- "<<n<<"\n";
grid= traversal(grid);
}
return 0;
}
```

REFERENCES

1. Versteeg H., Malalasekera W., 'Introduction to Computational Fluid Dynamics: The Finite Volume Method', 1995, Longman Scientific & Technical
2. Anderson J., 'Computational Fluid Dynamics: The Basics with Applications', 1995, McGraw-Hill
3. Patankar S., 'Numerical Fluid Flow and Heat Transfer', Hemisphere Publishing Corporation, 1980
4. Ferziger J., Peric M., 'Computational Methods for Fluid Dynamics, Springer-Verlag, 2001
5. The SIMPLE Algorithm, Patakar S., Spalding B., 1972 & The Tridiagonal Matrix Algorithm, Thomas, 1949