MAE 471 Computational Fluid Dynamics

Dr. Herrmann

Final Project: Lid Driven Cavity

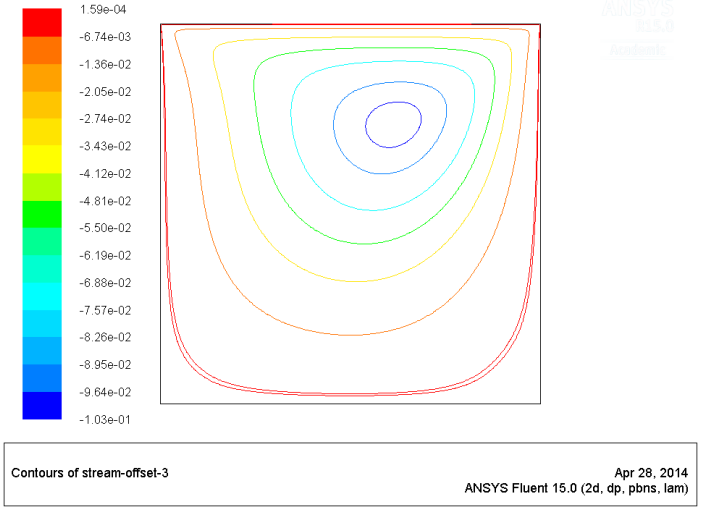
Due: May 7, 2014

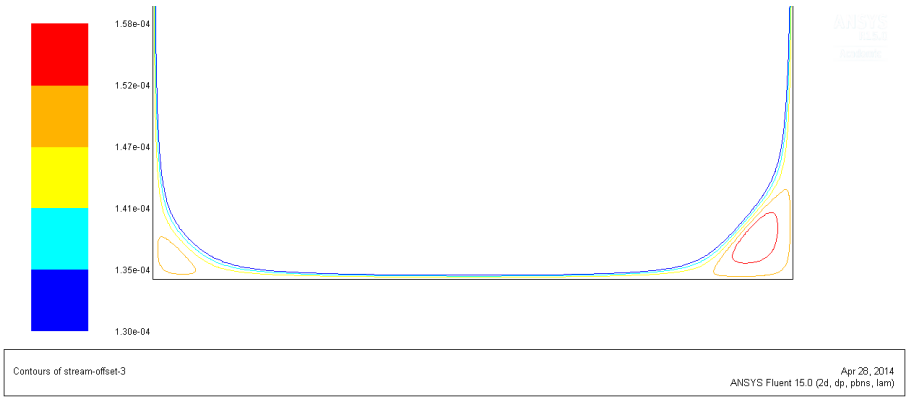
Josh Stout

Task 1:

Using fluent, I was able to solve the problem demonstrated by Ghia et al with results fairly consistent to theirs. A mesh resolution of 320x320 was used, with a Reynold's number of 100 designated by a side length of 1, fluid density of 1, viscosity of 0.01, and plate velocity of 1. The plate velocity was defined in the positive x direction as an absolute velocity, and all walls were considered to be non-permeable and non-slip. Solution methods were set as a coupled scheme with second order pressure and second order momentum terms, with a least squares cell based gradient. The accepted convergence criteria was set as 10^{-5} for continuity and 10^{-4} for both the x and y velocity terms.

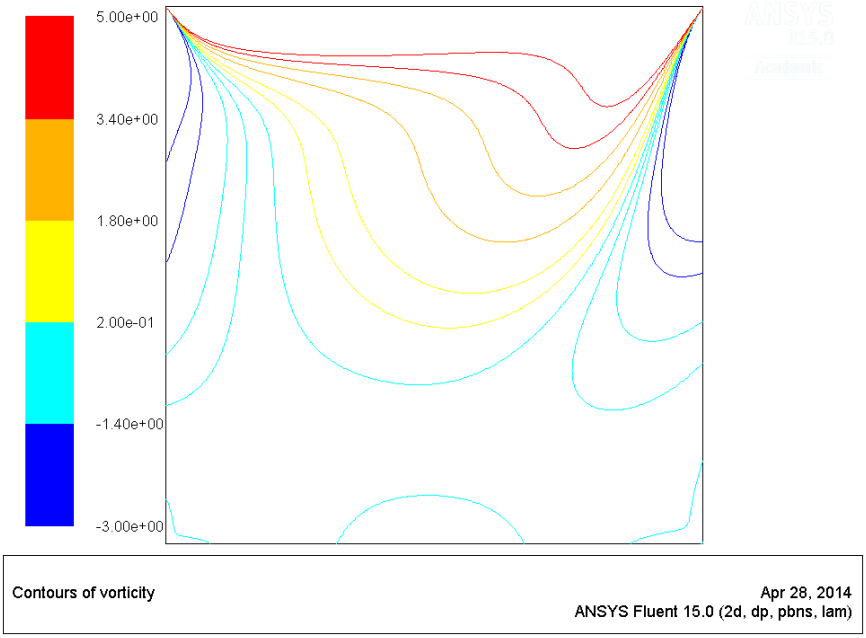
The stream function had to be offset according to the value found at the boundary in Fluent, resulting in the following plots:





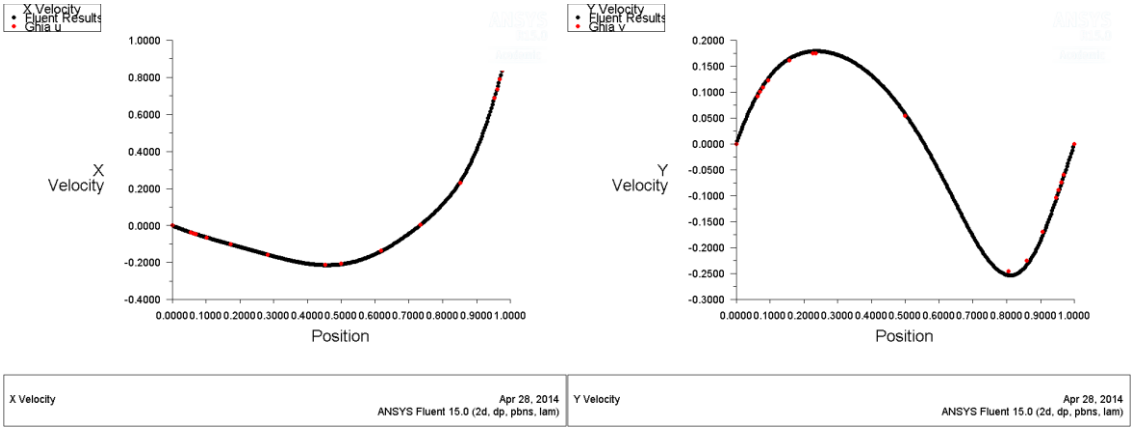
The corner vortices were not as easily visible as in Ghia's paper, so additional value refinement had to be used in order to bring them into light. Their contour values are very sensitive, but the vortices are most definitely present in the simulation, and closely resemble Ghia's in shape.

Using the same simulation results as above, a user defined function for vorticity was created in order to properly replicate Ghia's results. It was manually entered as $\omega = \frac{du}{dy} - \frac{dv}{dx}$, and creating a surface according to the values presented by Ghia resulted in the following plot:



This is even more similar to Ghia's results than the stream function, as all of the contours are fully represented.

Likewise, plots of the x direction velocity were created for the vertical mid-surface, as well as plots for the y direction velocity along the horizontal mid-surface. These also compare favorably with Ghia.

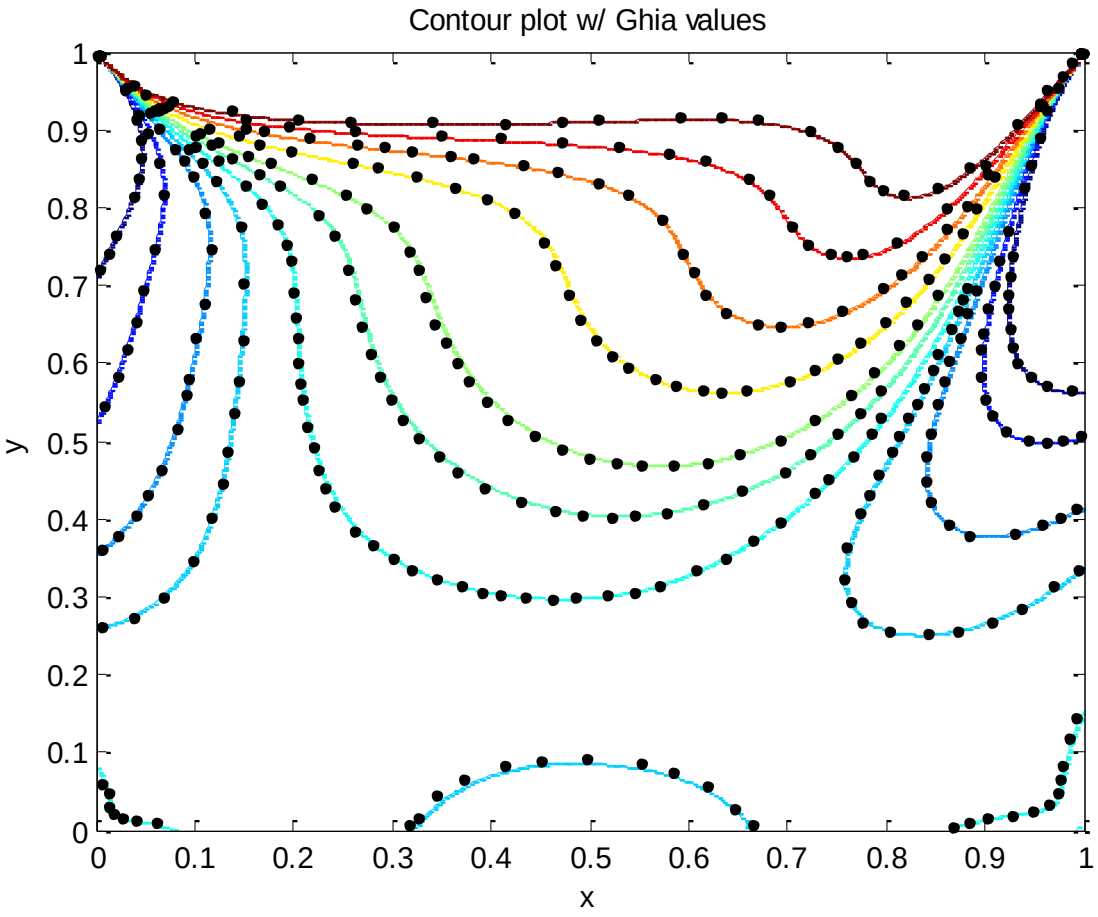


Task 2:

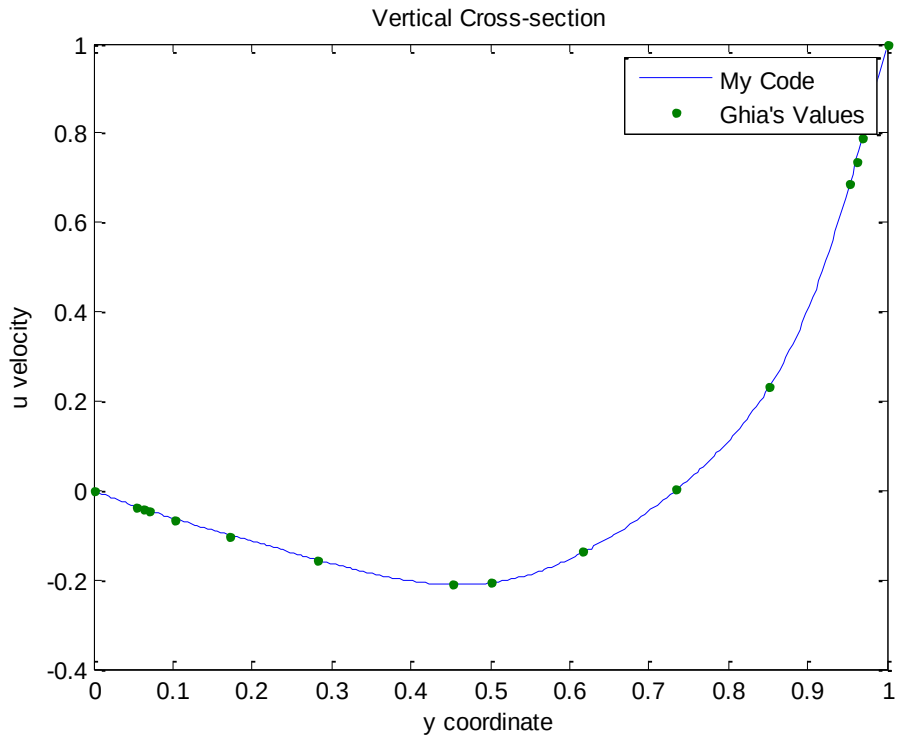
For my personally coded solution, I have chosen to employ an FTCS scheme within the fractional step method. For convergence criteria, I am using a constant 10^{-5} for each Gauss-Seidel function call, and a kinetic energy formulation for the overall convergence. The sum of squared velocities, normalized by M^2, is calculated at every point inside of the domain, and then stored for each iteration. The solution is considered converged when the difference between consecutive kinetic energy values is less than 10^{-7}.

The grid resolutions chosen are 60x60, 120x120, and 240x240, of which plots will only be shown from the 240x240 results.

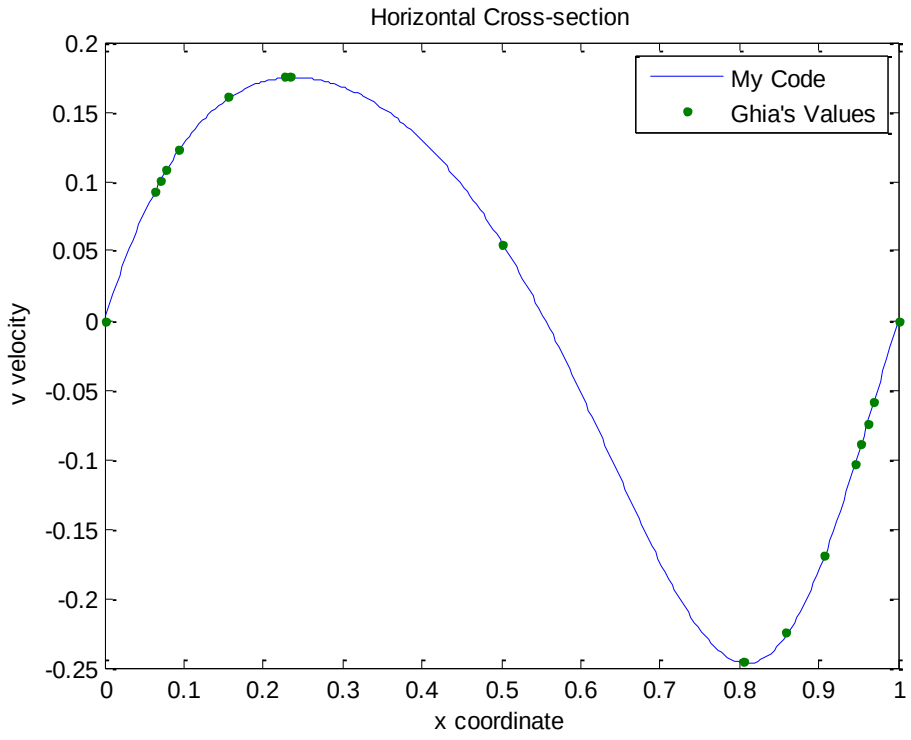
Vorticity Contour with Ghia comparison:



Vertical cross-section u velocity:



Horizontal cross section v velocity:



Visually, these values line up very well.

For solution verification, a GCI method was employed on the maximum vertical velocity along the horizontal cut.

Mesh Size	60	120	240
Velocity	0.1694	0.1744	0.1746

The relevant equations are as follows, with the subscript 1 indicating the finest mesh. The r value here is 2, as the mesh spacing was halved for each successive run.

$$p = \frac{\log(\frac{v3-v2}{v2-v1})}{\log(r)}, \varepsilon_{21} = \frac{u_1-u_2}{u_1}, \quad GCI_{21} = 1.25 * \frac{\varepsilon_{21}}{r^p-1}, \quad GCI_{32} = 1.25 * \frac{\varepsilon_{32}}{r^p-1}, \quad \frac{GCI_{21}}{GCI_{32}} * r^p$$

Plugging values in:

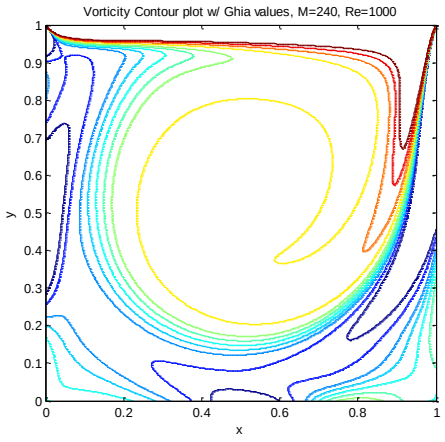
P = 4.4629 GCI_{21} = 7.7521e-5 GCI_{32} = 0.00171 $\frac{7.7521e-5}{0.0017} * 2^{4.4629} = 0.99869$

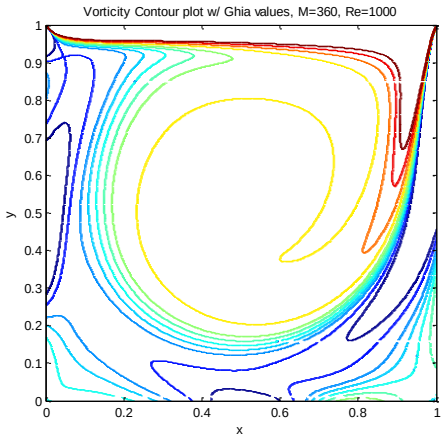
The asymptotic convergence check is very close to 1, providing evidence that the mesh is fine enough for the given convergence criteria.

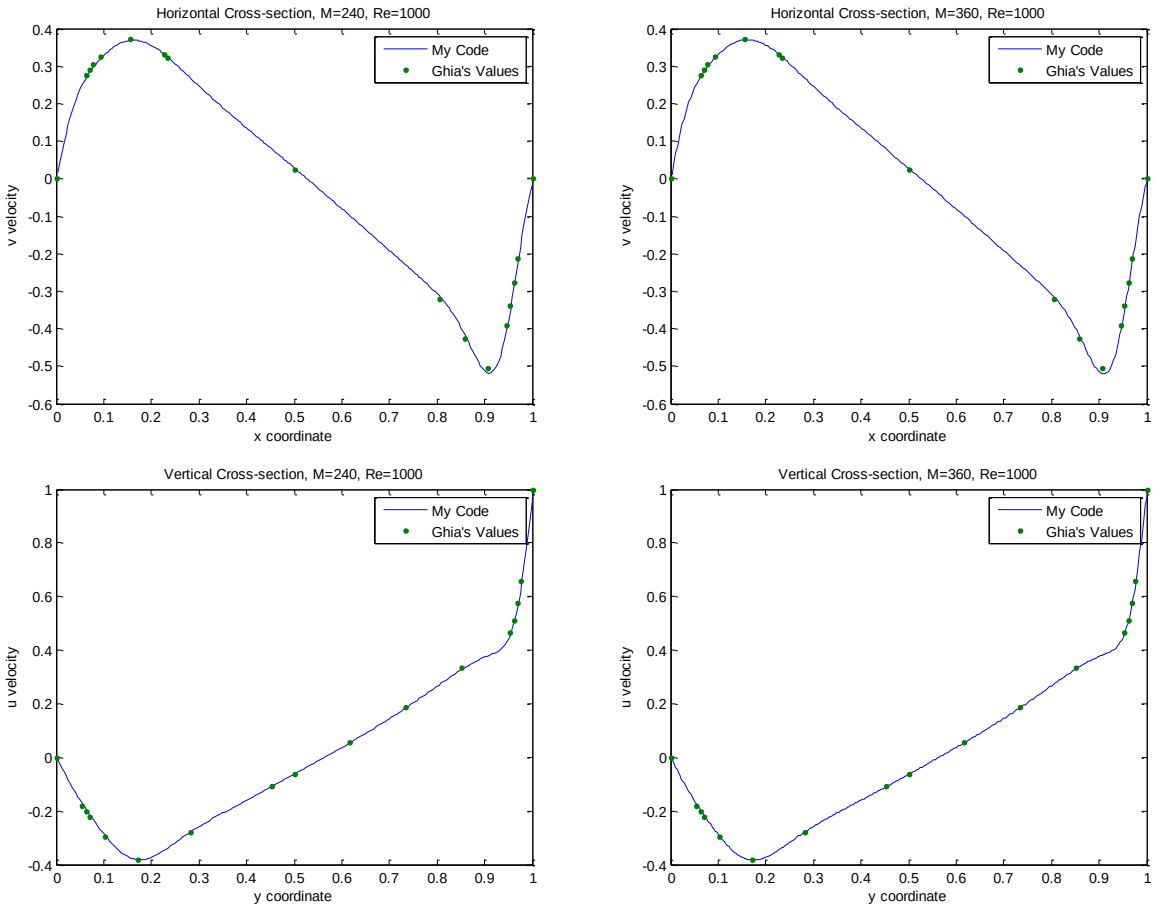
Note: the P value here is significantly larger than the second order convergence I would expect from the central difference approximations. This may be a result of the kinetic energy convergence term not quite being valued the same across mesh levels.

Bonus: Re = 1000

2 Mesh levels were chosen, all of the same settings were used except that the timestep safety factor was increased to 0.25, the upper limit on timesteps was increased, and the Reynold's number was changed to 1000.







These plots indicate that not only do the results match up quite well with Ghia's values, but they are also fairly independent of the mesh sizing.

C++ Code: Formatted in Microsoft Visual Studio 2013

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <fstream>
using namespace std;

// Initial ghost cell condition, based on top wall u velocity of 1
void IC(
	std::vector<std::vector<double>>& u,
	int M,
	std::vector<std::vector<double>>& v)
{
	for (int i = 0; i < M + 1; i++){
		u[i][M + 1] = 2 - u[i][M];
		u[i][0] = -u[i][1];
		v[0][i] = -v[1][i];
		v[M + 1][i] = -v[M][i];
	}
	for (int i = 0; i < M + 2; i++){
		u[0][i] = 0;
		u[M][i] = 0;
		v[i][0] = 0;
		v[i][M] = 0;
	}
}

// Calculate the maximum allowable timestep based on d<=1/2 and Re <= 2/C
double dtcalc(
	double dt1,
	std::vector<std::vector<double>> dt2,
	int M,
	std::vector<std::vector<double>>& u,
	std::vector<std::vector<double>>& v)
{
	double dt = dt1; //bring in mesh timestep constraint (d<=1/2) for comparison
	double dttemp;		//create a temporary value for storage
	//loop through grid to find maximum allowable timestep according to Re <= 2/C
	for (int j = 1; j < M + 1; j++) {
		for (int i = 1; i < M + 1; i++){
			dttemp = 8 * 0.01 / (pow(u[i][j + 1] + u[i][j], 2) + pow(v[i + 1][j] + v[i][j], 2));
			if (dttemp<dt)
				dt = dttemp;
		}
	}
	return dt;
}

// Predictor step for Fractional Step Method, using FTCS
void pred(
	double dt,
	double h,
	int M,
	double Re,
	std::vector<std::vector<double>>& ustar,
	std::vector<std::vector<double>>& vstar,
	std::vector<std::vector<double>>& u,
	std::vector<std::vector<double>>& v)
{
	// FTCS for ustar and vstar terms
	for (int j = 1; j < M + 1; j++){
		for (int i = 1; i < M; i++){
			ustar[i][j] = ((u[i - 1][j] + u[i + 1][j] - 4 * u[i][j] + u[i][j - 1] + u[i][j + 1]) / (Re*h*h)
```

```
                                        - ((u[i + 1][j] + u[i][j])*(u[i + 1][j] + u[i][j]) -  (u[i][j] + u[i - 1][j])*(u[i][j] + u[i - 1][j])) / (4 * h)
                                        - ((u[i][j] + u[i][j + 1])*(v[i][j] + v[i + 1][j]) -      (u[i][j - 1] + u[i][j])*(v[i][j - 1] + v[i + 1][j - 1])) /
                                        (4 * h))*dt + u[i][j];
                        }
                }

                for (int j = 1; j < M; j++){
                        for (int i = 1; i < M + 1; i++){
                                vstar[i][j] = ((v[i - 1][j] + v[i + 1][j] - 4 * v[i][j] + v[i][j - 1] + v[i][j + 1]) / (Re*h*h)
                                        - ((v[i][j + 1] + v[i][j])*(v[i][j + 1] + v[i][j]) - (v[i][j] + v[i][j - 1])*(v[i][j] + v[i][j - 1])) / (4 * h)
                                        - ((u[i][j] + u[i][j + 1])*(v[i][j] + v[i + 1][j]) - (u[i - 1][j] + u[i - 1][j + 1])*(v[i - 1][j] + v[i][j])) /
                                        (4 * h))*dt + v[i][j];
                        }
                }

                // Apply boundary conditions
                // Top and Bottom conditions for u, left and right for v
                for (int i = 0; i < M + 1; i++){
                        ustar[i][M + 1] = 2 - u[i][M];
                        ustar[i][0] = -u[i][1];
                        vstar[0][i] = -v[1][i];
                        vstar[M + 1][i] = -v[M][i];
                }
                for (int i = 0; i < M + 2; i++){
                        ustar[0][i] = 0;
                        ustar[M][i] = 0;
                        vstar[i][0] = 0;
                        vstar[i][M] = 0;
                }
}

// Gauss Seidel Iteration Solver for the Poisson Equation
int GaussSeidel(
                int M,
                double h,
                std::vector<std::vector<double>>& ustar,
                std::vector<std::vector<double>>& vstar,
                std::vector<std::vector<double>>& Phi,
                std::vector<std::vector<double>>& PhiResid,
                std::vector<std::vector<double>>& b,
                double dt)
{
                int GScount = 1; //begin a counter for GS iterations
                for (int j = 1; j < M + 1; j++){        //presolve the b value at each mesh point
                        for (int i = 1; i < M + 1; i++){
                                b[i][j] = (ustar[i][j] - ustar[i - 1][j] + vstar[i][j] - vstar[i][j - 1])*h / dt;
                        }
                }

                double MaxResid = 1;  // Initialize residual at 1 to begin looping
                // Continue looping Guass Seidel until max residual is < 1e-5
                while (MaxResid > 0.00001){
                        MaxResid = 0;           //reset max residual

                        //Gauss Seidel loop
                        for (int j = 1; j < M + 1; j++){
                                for (int i = 1; i < M + 1; i++){
                                        Phi[i][j] = 0.25*(Phi[i - 1][j] + Phi[i + 1][j] + Phi[i][j + 1] + Phi[i][j - 1]) - 0.25*b[i][j];
                                }
                        }
                        //ghost cell boundary conditions for dphi/dh = 0
                        for (int i = 1; i < M + 1; i++){
                                Phi[0][i] = Phi[1][i];                          //Left side
                                Phi[M + 1][i] = Phi[M][i];                      // Right side
                                Phi[i][0] = Phi[i][1];                          // Bottom
                                Phi[i][M + 1] = Phi[i][M];                      // Top
```

```
                }
                //Loop through mesh and calculate the current residual
                for (int j = 1; j < M + 1; j++){
                        for (int i = 1; i < M + 1; i++){
                                PhiResid[i][j] = 4 * Phi[i][j] - (Phi[i - 1][j] + Phi[i + 1][j] + Phi[i][j + 1] + Phi[i][j - 1]) + b[i][j];
                        }
                }

                // Calculate the maximum of residuals present for convergence test
                double resid = 0;       //initialize residual value
                for (int j = 1; j < M + 1; j++) {
                        for (int i = 1; i < M + 1; i++){
                                if (abs(PhiResid[i][j])>resid)
                                        resid = abs(PhiResid[i][j]);
                        }
                }
                MaxResid = resid; //apply new max residual to MaxResid for loop condition
                GScount++;
        }
        return GScount - 1;
}

//Corrector step function
double Corrector(
        double dt,
        int M,
        double h,
        std::vector<std::vector<double>>& ustar,
        std::vector<std::vector<double>>& vstar,
        std::vector<std::vector<double>>& u,
        std::vector<std::vector<double>>& v,
        std::vector<std::vector<double>>& Phi)
{       //Calculate the new u
        for (int j = 1; j < M + 1; j++){
                for (int i = 1; i < M; i++){
                        u[i][j] = ustar[i][j] - dt / h*(Phi[i + 1][j] - Phi[i][j]);
                }
        }
        //Calculate the new v
        for (int j = 1; j < M; j++){
                for (int i = 1; i < M + 1; i++){
                        v[i][j] = vstar[i][j] - dt / h*(Phi[i][j + 1] - Phi[i][j]);
                }
        }
        //Apply non-zero boundary conditions
        for (int i = 1; i < M; i++){
                u[i][0] = -u[i][1];
                u[i][M + 1] = 2 - u[i][M];
                v[0][i] = -v[1][i];
                v[M + 1][i] = -v[M][i];
        }
        //Calculate the current kinetic energy of all points within non ghost mesh
        double Kval = 0;
        for (int j = 0; j < M; j++){
                for (int i = 0; i < M + 1; i++){
                        Kval = Kval + (u[i][j + 1] * u[i][j + 1] + v[j + 1][i] * v[j + 1][i]);
                }
        }
        Kval = Kval / (M*M); //normalize energy according to grid size
        //output kinetic energy term for convergence check
        return Kval;
}

//Vorticity Solver
void VortCalc(
        int M,
```

```
		double h,
		std::vector<std::vector<double>>& u,
		std::vector<std::vector<double>>& v,
		std::vector<std::vector<double>>& vorticity)
{
		for (int j = 0; j < M+1; j++){
			for (int i = 0; i < M+1; i++){
				vorticity[i][j] = -(v[i + 1][j] - v[i][j] - u[i][j + 1] + u[i][j]) / h;
			}
		}
}

int main(){
		std::cout.precision(10);
		const double Re = 100;				//Fix Reynold's number
		double M;
		//store value M from user input
		std::cout << "Input desired elements in x and y direction:" << std::endl;
		std::cin >> M;
		const double h = 1 / M;
		double dt1 = Re*h*h / 4;
		int GScount;
		std::vector<std::vector<double>> u;
		std::vector<std::vector<double>> v;
		std::vector<std::vector<double>> ustar;
		std::vector<std::vector<double>> vstar;
		std::vector<std::vector<double>> Phi;
		std::vector<std::vector<double>> PhiResid;
		std::vector<std::vector<double>> b;
		std::vector<std::vector<double>> dt2;
		std::vector<std::vector<double>> vorticity;
		std::vector<double> Kresid;
		std::vector<double> Kval;
		u.resize(M + 1, std::vector<double>(M + 2, 0));
		v.resize(M + 2, std::vector<double>(M + 1, 0));
		ustar = u; vstar = v;
		Phi.resize(M + 2, std::vector<double>(M + 2, 0));
		PhiResid = Phi;
		b.resize(M + 2, std::vector<double>(M + 2, 0));
		dt2.resize(M, std::vector<double>(M, 0));
		vorticity.resize(M + 1, std::vector<double>(M + 1, 0));
		Kresid.resize(30000, 1);		//allocate enough space for 1 per iteration
		Kval = Kresid;


		//Begin solution phase
		IC(u, M, v); //initial conditions function
		int k = 0;	//initialize loop counter
		double velocityresid = 1;		// initialize velocity field residual for looping

		//Begin looping of solution functions
		while (Kresid[k] > 0.0000001){
			double dt = dtcalc(dt1, dt2, M, u, v);
			pred(dt, h, M, Re, ustar, vstar, u, v);
			GScount = GaussSeidel(M, h, ustar, vstar, Phi, PhiResid, b, dt);
			Kval[k] = Corrector(dt, M, h, ustar, vstar, u, v, Phi);
			//Output monitor of GS steps
			if (k % 100 == 0){
				std::cout << "This iteration required " << GScount << " Gauss Seidel step(s)." << std::endl;
			}
			//Calculate a kinetic residual for convergence criteria
			if (k > 0){
				Kresid[k + 1] = Kval[k] - Kval[k - 1];
			}
			//Exit condition if not converged within 30000 loops
			if (k == 29998){
```

```
                        std::cout << "The solution has not converged within the allocated loop space." << endl;
                                break;
                }
                k++;
        }
        std::cout << "Solution converged kinetically after " << k << " iterations" << endl;

        //Calculate vorticity based on final solution
        VortCalc(M, h, u, v, vorticity);

        //Create vectors for the velocities
        std::vector<double> x;
        x.resize(M + 2);
        //first and last points in ghost cells
        for (int i = -1; i < M + 1; i++){
                x[i + 1] = (2 * i + 1)*(h / 2);
        }

        //write file for velocity profiles
        std::ofstream velocity;
        velocity.open("velocity.txt");
        velocity << std::setw(20) << "x" << std::setw(20) << "u" <<
                std::setw(20) << "y" << std::setw(20) << "v" << endl;
        for (int i = 0; i < M + 2; i++){
                velocity << std::setw(20) << x[i] << std::setw(20) << u[M / 2 - 1][i] <<
                        std::setw(20) << x[i] << std::setw(20) << v[i][M / 2 - 1] << std::endl;
        }
        velocity.close();

        //Create vectors for vorticity
        std::vector<double> xvort;
        xvort.resize(M + 1);
        for (int i = 0; i < M + 1; i++){
                xvort[i] = i*h;
        }

        //write file for vorticity
        std::ofstream Vorticity;
        std::fixed;
        Vorticity.precision(6);
        Vorticity.open("Vorticity.txt");
        Vorticity << std::setw(12) << " x/y ";
        for (int i = 0; i < M + 1; i++){
                Vorticity << std::setw(12) << xvort[i];
        }
        Vorticity << endl;
        for (int j = 0; j < M + 1; j++){
                Vorticity << std::setw(12) << xvort[j];
                for (int i = 0; i < M + 1; i++){
                        Vorticity << std::setw(12) << vorticity[i][j];
                }
                Vorticity << endl;
        }
        Vorticity.close();

        int preventclose;
        std::cout << "The simulation is now complete.";
        std::cin >> preventclose;
}
```