AME 50532 – Computational Fluid Dynamics

Lid-Driven Cavity Flow

Joseph Barbaro

1. Introduction and Theory

The steady, incompressible Naiver Stokes equations were solved for the viscous flow in a square, 0.05*m* x 0.05*m* two-dimensional lid driven cavity. Relaxation techniques were used to solve for the steady state solution, and the fluid was assumed to be incompressible. The scheme uses time-derivative preconditioning and artificial viscosity techniques. The pressure only appears as a gradient in the governing equations, so its value is rescaled with a centered reference point at each time step to prevent additional numerical errors. The baseline conditions were Re=100, ρ = 1.0kg/m^3 and U_{lid} = *1.0m/s.*

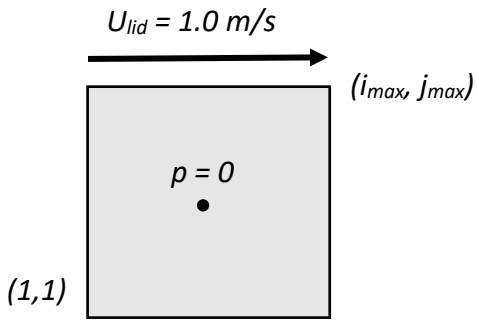

Figure 1. Lid-driven cavity.

1.1 Governing Equations

The governing equations for the incompressible flow are given in equations 1 through 3. The energy equation is decoupled from these equations. The mass continuity equation (eqn. 1) can be solved for the pressure component and the momentum equations can be solved for the velocity components (eqn. 2 & 3).

$$\frac{1}{\beta^2}\frac{\partial p}{\partial t} + \rho\frac{\partial u}{\partial x} + \rho\frac{\partial v}{\partial y} = S \tag{1}$$

$$\rho\frac{\partial u}{\partial t} + \rho u\frac{\partial u}{\partial x} + \rho v\frac{\partial u}{\partial y} + \frac{\partial P}{\partial x} = \mu\frac{\partial^2 u}{\partial x^2} + \mu\frac{\partial^2 u}{\partial y^2} \tag{2}$$

$$\rho\frac{\partial v}{\partial t} + \rho u\frac{\partial v}{\partial x} + \rho v\frac{\partial v}{\partial y} + \frac{\partial P}{\partial y} = \mu\frac{\partial^2 v}{\partial x^2} + \mu\frac{\partial^2 v}{\partial y^2} \tag{3}$$

1.2 Boundary Conditions

Boundary conditions are calculated in the subfunction given in Appendix A. The y-component of velocity is set equal to zero for each wall, and the x-component of velocity is set equal to zero for each wall except the lid, which has a velocity of 1.0m/s. A simple linear extrapolation was used to calculate the pressure boundary conditions at each wall. The artificial viscosity terms (discussed below) also required boundary conditions so that the discretizations did not exceed the bounds of the mesh. For the outermost layer, the artificial viscosity was set to zero. For the second outermost layer, vertical walls were calculated in the y-direction and the x-direction was set equal to the third layer. The horizontal walls and corners followed a similar logic.

1.3 Discretization

The simple explicit method was used to discretize the governing equations. Temporal derivatives used a forward difference in time and spatial derivatives used centered differences. The resulting continuity equation with artificial compressibility, artificial viscosity, and mms terms is

$$p_{i,j}^{n+1} = p_{i,j}^{n} - \beta_{i,j}^{2}\Delta t\left[\rho\frac{u_{i+1,j}^{n} - u_{i-1,j}^{n}}{2\Delta x} + \rho\frac{v_{i,j+1}^{n} - v_{i,j-1}^{n}}{2\Delta y} - S_{i,j} - f_{mass}(x,y)\right] \tag{4}$$

and the x-momentum and y-momentum equations use the following format.

$$\begin{aligned} u_{i,j}^{n+1} = u_{i,j}^{n} - \frac{\Delta t}{\rho}\Bigg[&\rho u_{i,j}^{n}\frac{u_{i+1,j}^{n} - u_{i-1,j}^{n}}{2\Delta x} + \rho v_{i,j}^{n}\frac{u_{i,j+1}^{n} - u_{i,j-1}^{n}}{2\Delta y} + \frac{p_{i+1,j}^{n} - p_{i-1,j}^{n}}{2\Delta x} \\ &- \mu\frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^{2}} - \mu\frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^{2}} - f_{xmtm}(x,y)\Bigg] \end{aligned} \tag{5}$$

1.4 Stability

Stability implies that errors do not grow as the program proceeds in the marching direction. This stability can be influenced by parameters such as mesh size, time step, and Reynolds number. The stability for the system originates from the convective stability limit and the diffusive stability limits (Appendix B). The convective stability limit for the time step is defined as

$$\Delta t_c \le \frac{\min(\Delta x, \Delta y)}{|\lambda|_{\max}} \tag{6}$$

where the eigenvalues are given by

$$\lambda_x = \frac{1}{2}\left(|u| + \sqrt{u^2 + 4\beta^2}\right) \tag{7}$$

$$\lambda_y = \frac{1}{2}\left(|v| + \sqrt{v^2 + 4\beta^2}\right) \tag{8}$$

The diffusive stability limit is defined as

$$\Delta t_d \le \frac{\Delta x \Delta y}{4\nu} \tag{9}$$

Therefore the time step is given the following constraint

$$\Delta t \le CFL \cdot \min(\Delta t_c, \Delta t_d) \tag{10}$$

where the CFL is a number that can vary between 0 and 1.

1.5 Artificial Viscosity

To address the problem of odd-even decoupling, an artificial viscosity term, *S*, was added to the governing equations.

$$S = -\frac{|\lambda_x|_{max} C^{(4)} \Delta x^3}{\beta^2} \frac{\partial^4 p}{\partial x^4} - \frac{|\lambda_y|_{max} C^{(4)} \Delta y^3}{\beta^2} \frac{\partial^4 p}{\partial y^4} \tag{11}$$

At each node, β^2 was selected based as the max of the local velocity and a specified fraction (.001-.9) of the freestream velocity. $C^{(4)}$ is a constant in the range

$$\frac{1}{128} \le C^{(4)} \le \frac{1}{16}. \tag{12}$$

and $|\lambda_x|_{max}$ and $|\lambda_y|_{max}$ correspond to the magnitude of the largest eigenvalues in (x, t) space and (y, t) space, respectively. Artificial viscosity calculations can be found in Appendix 2.

1.6 Artificial Compressibility

This project is only concerned with steady state solutions, so an artificial compressibility term can be added to the continuity equation. This term contains a temporal derivative of pressure with a leading $1/\beta^2$ term. The time derivative preconditioning term can be defined as

$$\beta^2 = \max(u^2 + v^2, \kappa U_{lid}^2) \tag{13}$$

where κ is a constant that can range from 0.001 to 0.9.

1.7 Point Jacobi and SGS Algorithms

Point relaxation techniques were implemented for this project. The Point Jacobi method evaluates the node for the current iteration level using unknowns that are all from the previous iteration level. The Gauss-Seidel algorithm is similar to the Point Jacobi except some adjacent unknowns can come from the current iteration level (Appendix C).

A Gauss-Seidel algorithm is more efficient and transferring information forward with each iteration because the most recent information is propagating forward. A backward sweep can be added to avoid this preferential treatment. The Symmetric Gauss-Seidel method uses a forward sweep using information at *i-1* locations and a backward sweep using information at *i+1* locations (Appendix D).

2. Iterative convergence analysis

Figure 2 shows the iterative convergence history for the Gauss-Seidel and Point Jacobi methods. Residuals are calculated using the following equation.

$$-R_{i,j}^{n} = (u_{i,j}^{n+1} - u_{i,j}^{n})/\Delta t \tag{14}$$

A ratio of iterative residuals to the initial iterative residuals was used as the convergence criteria (Appendix E). It can be observed in figure 2 that the Point-Jacobi method required more iterations to converge than the Symmetric Gauss-Seidel method. The SGS method contains a backwards sweep that, although it requires more calculations per iteration, is more efficient at propagating information and therefore converges using fewer iterations. When the mesh size was increased from 65x65 to 129x129 for the SGS method, the number of iterations required increased significantly as expected.

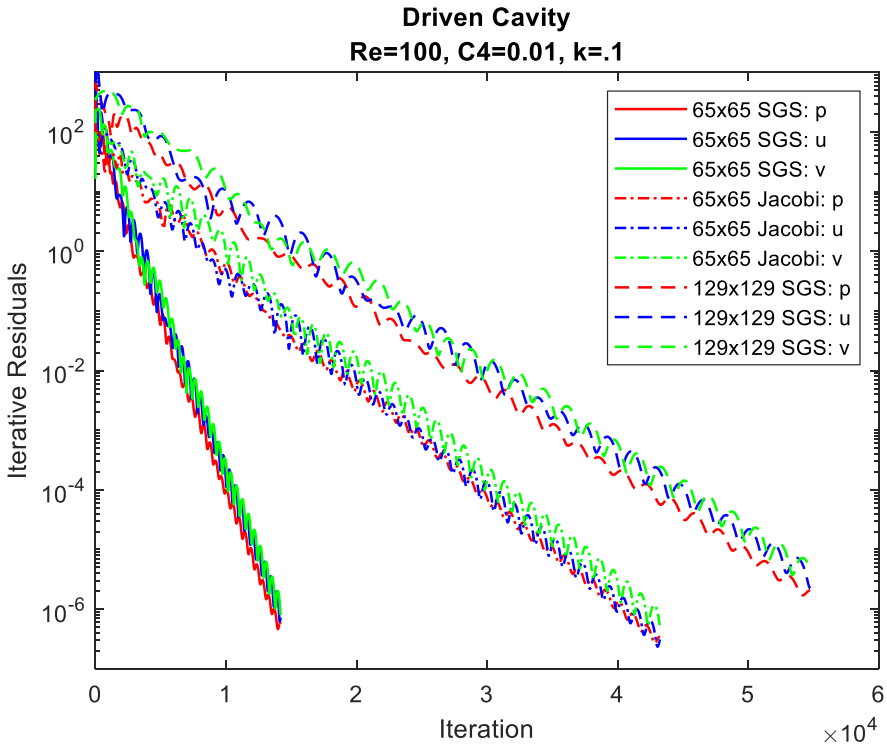

Figure 2. Iterative Convergence History

3. Code Verification

For the following code verification study, the method of manufactured solutions was used to generate an exact solution containing smooth, differentiable analytical functions. The manufactured solution was substituted into the governing equation to derive the source terms that could be added to the original PDE. This modified PDE can be used to evaluate the order of accuracy of the code. The MMS was applied to the uniformly spaced meshes noted in table 1.

Table 1. Mesh sizes used for code verification.

Mesh Name	Mesh Nodes	Grid Spacing, h
Mesh 1	129 * 129	1
Mesh 2	65 * 65	2
Mesh 3	33 * 33	4
Mesh 4	17 * 17	8
Mesh 5	9 * 9	16

3.1 Discretization Error Estimation

The discretization error can be estimated using the equation

$$DE_k = f_k - f_{exact} \tag{15}$$

where k is the local mesh refinement and f_{exact} is the manufactured solution. In figure 3, various error norms were calculated and plotted against the h value noted in table 1 (Appendix F). For a second-order accurate algorithm it is expected that a slope of two is observed in the logarithmic plot (figure 3).

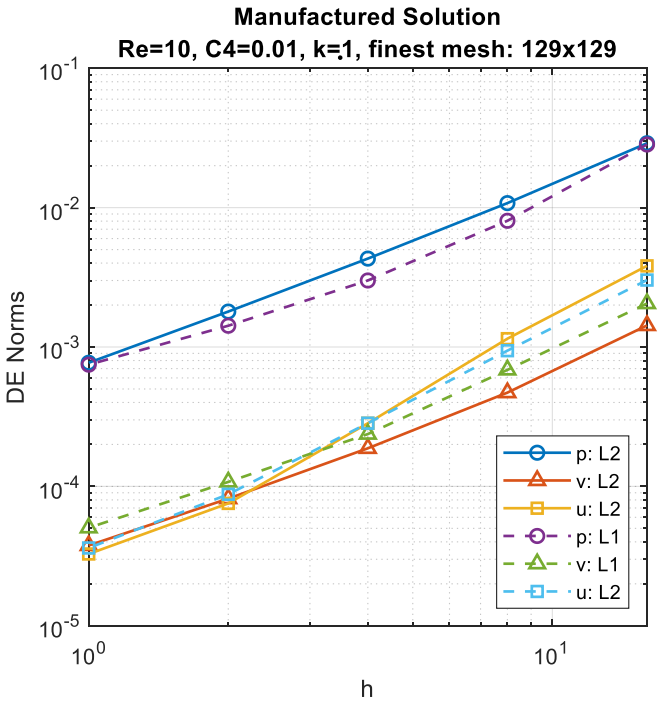

Figure 3. Discretization Error Estimation.

3.2 Observed Order of Accuracy

The observed order of accuracy for the manufactured solution is given by the equation

$$\hat{p} = \frac{ln\left(\frac{L_{k+1}}{L_k}\right)}{ln(r)} \tag{16}$$

where L_k is the DE norm for mesh level, k, and r is the grid refinement factor. The order of accuracies plotted in figure 5 increase for finer mesh sizes. The errors are generally below the expected formal order of accuracy of two, indicating that errors may be present in the algorithm.

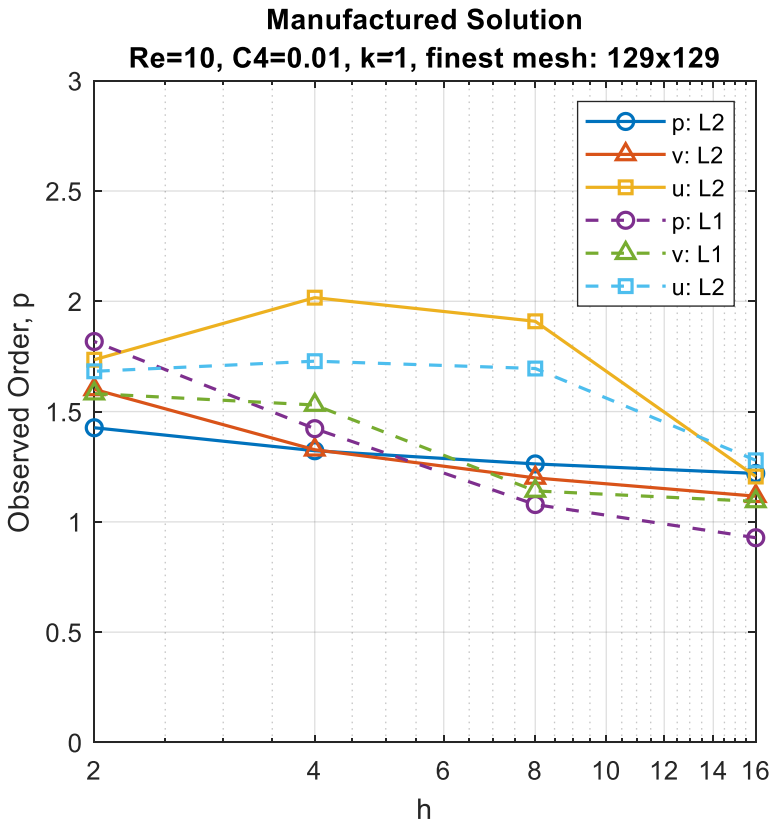

Figure 4. Observed order of accuracy.

4. Flow Analysis

Figures 5 through 11 show the contour plots for the lid-driven cavity at varying Reynolds numbers. The contour plots demonstrate how a clockwise rotation in the flow begins to develop as well as areas of high and low pressure at the upper-right and upper-left corners, respectively. Given that the velocity of the lid remains constant for this analysis, increasing the Reynolds number implies that kinematic viscosity decreases. Physically, this can be visualized in the contour plots as a deeper rotation of the fluid for higher Reynolds numbers. There are also vortices that begin to appear at the bottom corners for higher Reynolds numbers. These properties become the most defined for Re=500, Re=1000, and Re=5000 (Figures 11-13). These higher Reynolds number numerical solutions required larger mesh sizes to calculate and were found in given literature.

4.1 Re=10

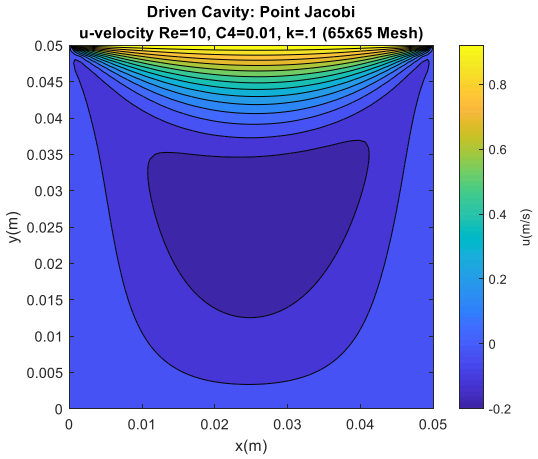

Figure 5. Re=10 u-velocity contour plot.

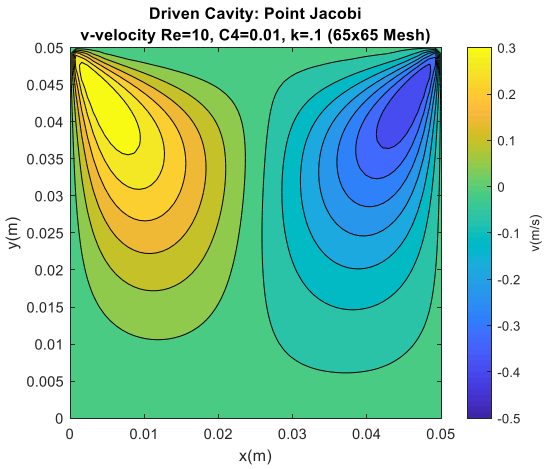

Figure 6. Re=10 v-velocity contour plot.

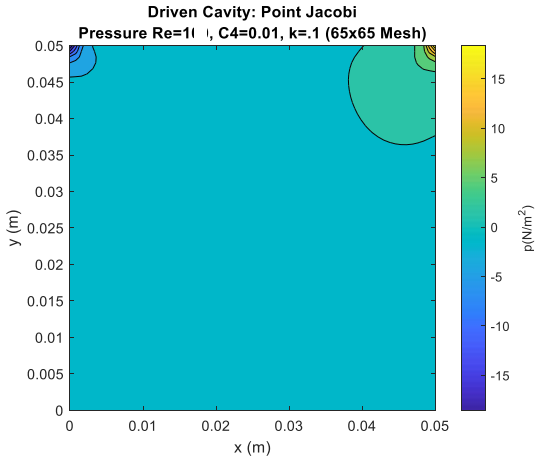

Figure 10. Re=10 pressure contour plot.

4.2 Re=100

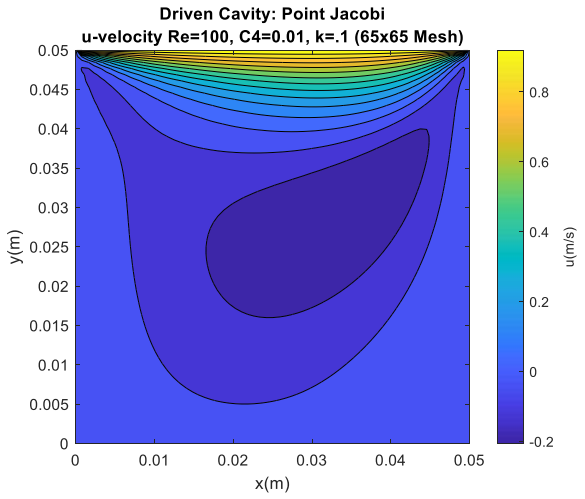

Figure 8. Re=100 u-velocity contour plot.

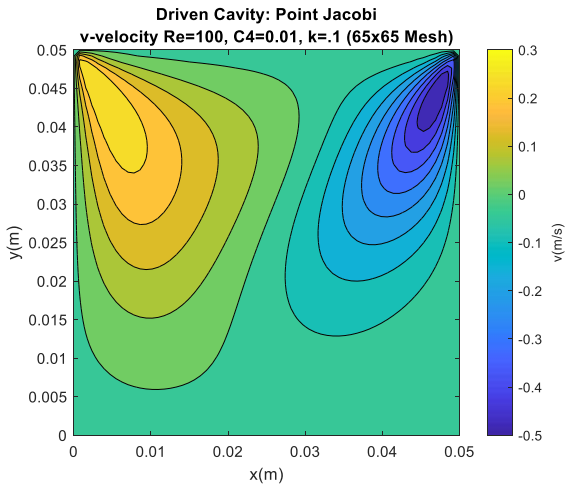

Figure 9. Re=100 v-velocity contour plot.

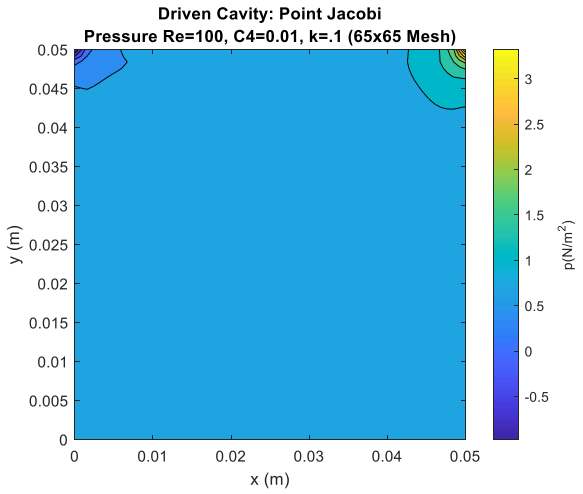

Figure 10. Re=100 pressure contour plot.

4.3 Given Numerical Solutions for Re=500 and Re=1000

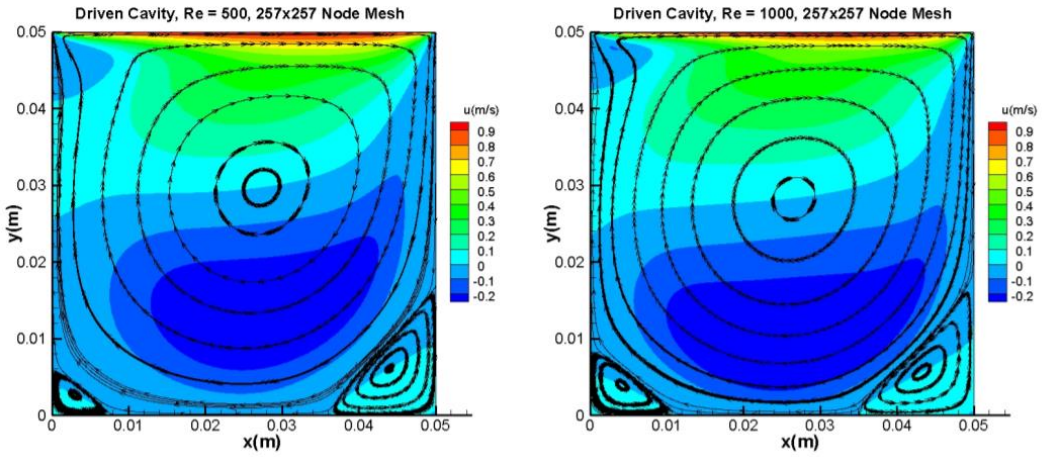

Figure 11. u-velocity contour plots for Re=500 and Re=1000, 257x257.

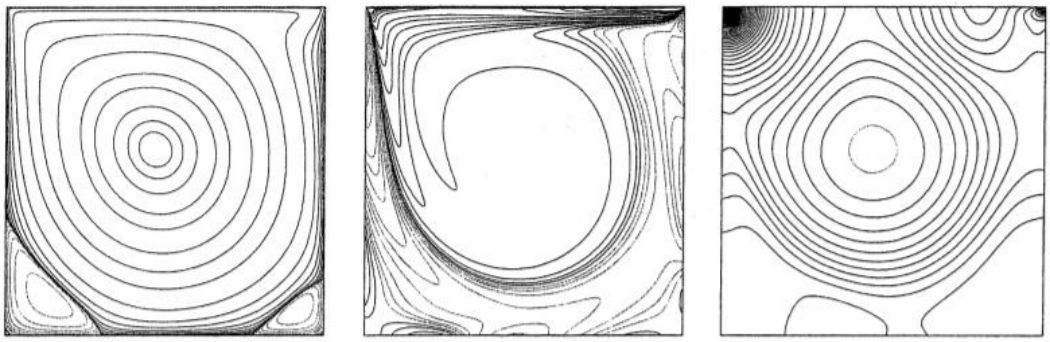

Figure 12. Re=1000, 1024x1024 stream function, vorticity, and pressure field[(1)].

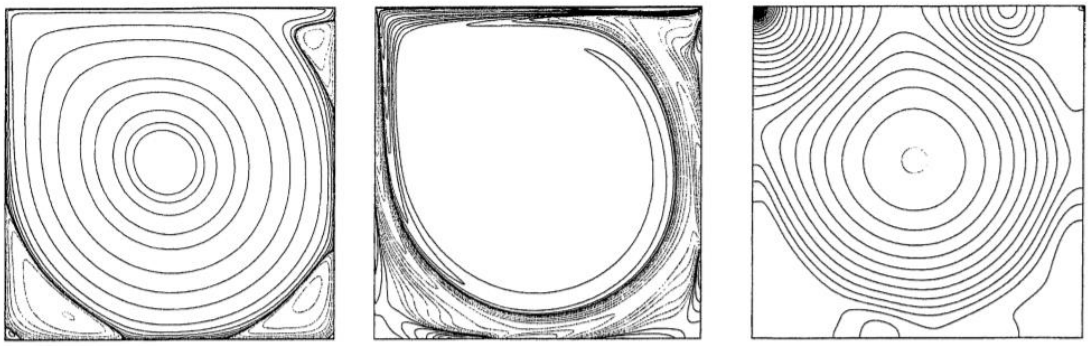

Figure 13. Re=5000, 2048x2048 stream function, vorticity, and pressure field[(1)].

5. Parameter Study

5.1 Effect of κ

Varying κ affects the time-derivative precondition term, which can influence the artificial compressibility, artificial viscosity, and local time step. Figure 14 shows how a change in κ affects the number of iterations required for convergence. For small κ, smaller β^2 terms are allowed (Eqn. 8), which can result in smaller time steps. For larger κ, more iterations were required.

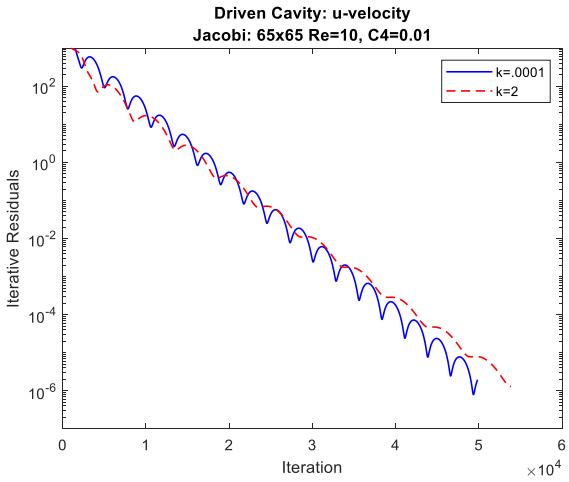

Figure 14. Iterative convergence history for u-velocity for different κ.

5.2 Effect of C(4)

$C^{(4)}$ is a constant that appears in the artificial viscosity terms. Figure 15 shows how the discretization error begins to oscillate as $C^{(4)}$ becomes small enough to begin to cancel the artificial viscosity terms. For smaller artificial viscosities, odd-even decoupling occurs because there is no longer a diffusive term.

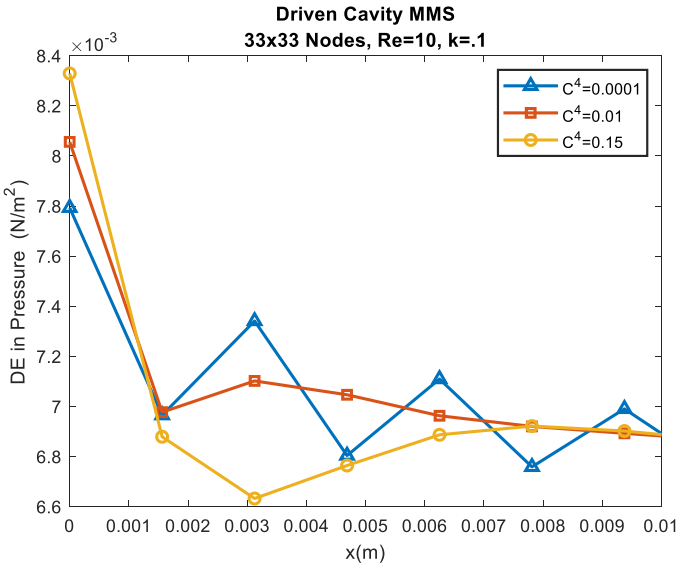

Figure 15. DE for Pressure for various values of $C^{(4)}$.

5.3 Effect of Mesh Size

For finer mesh sizes, the discretization error decreases. This was demonstrated figure 3, which plots the several grid spacings that are defined in table 1. For a uniform grid refinement factor of 2, it can be observed that the slope of the log-log plot in figure corresponds to a second order accurate system.

6. Comparison to Commercial Software Results

The cavity results can be validated against results from commercial software. Given in figures 16 through 18 are results from ANSYS Fluent for the lid-driven cavity at the baseline parameters.

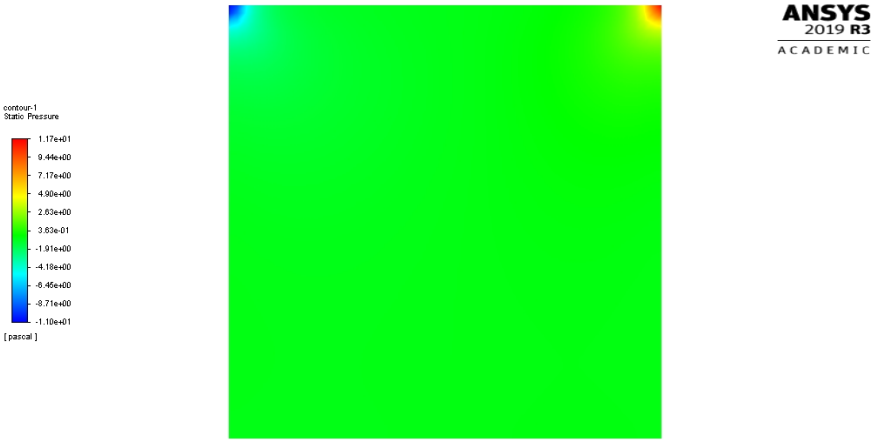

Figure 16. Fluent results for pressure.

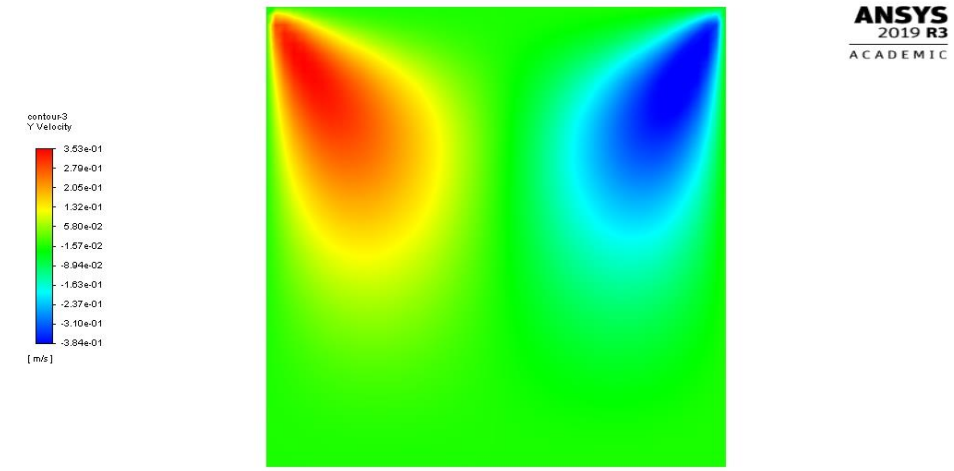

Figure 17. Fluent results for y-velocity.

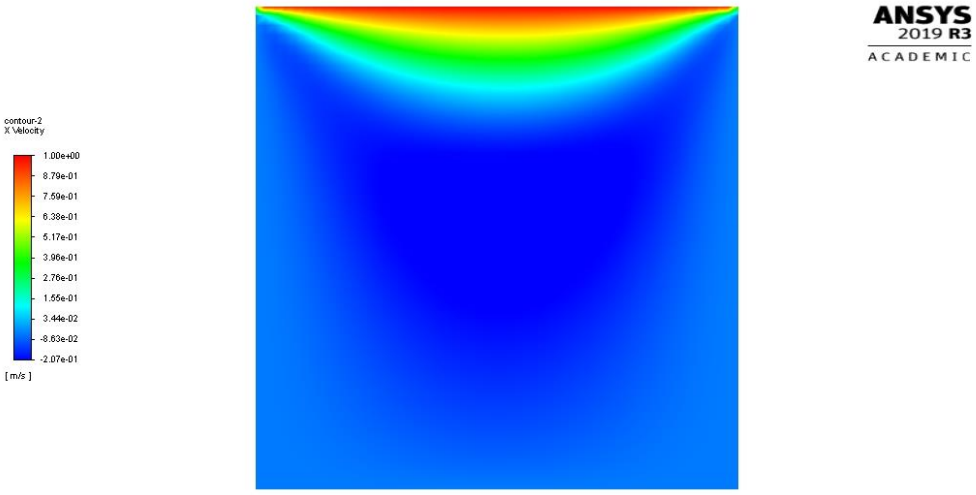

Figure 16. Fluent results for x-velocity.

Works Cited

(1) Bruneau, Charles-Henri and Mazen Saad. “The 2D lid-driven Cavity Problem Revisited” *Computers and Fluids*, no. 35, 2006, pp. 326-348.

Appendix A

```
function bndry(~)
%
%Uses global variable(s): zero, one, two, half, imax, jmax, uinf
%To modify: u
 % i                          % i index (x direction)
% j                          % j index (y direction)

global zero two half imax jmax uinf
global u

% This applies the cavity boundary conditions
%bottom
for i=1:imax
            j=1;
            u(i,j,1) = 2*u(i,j+1,1)-u(i,j+2,1);
            u(i,j,2) = zero;
            u(i,j,3) =  zero;
end
 %Left
for j=1:jmax
            i=1;
            u(i,j,1) = 2*u(i+1,j,1)-u(i+2,j,1);
           u(i,j,2) = zero;
            u(i,j,3) =  zero;
end

%Right
for j=1:jmax
            i=imax;
            u(i,j,1) = 2*u(i-1,j,1)-u(i-2,j,1);
            u(i,j,2) = zero;
            u(i,j,3) =  zero;
end
%Top
for i=2:imax-1
            j=jmax;
            u(i,j,1) = 2*u(i,j-1,1)-u(i,j-2,1);
            u(i,j,2) = uinf;
            u(i,j,3) =  zero;
end
% !************************************************************* */

%Artificial Viscosity boundary conditions

%{
for i=1:imax
    j=1;
    artviscx(i,j)=0;
    artviscy(i,j)=0;
end
```

```
for i=1:imax
    j=jmax;
    artviscx(i,j)=0;
    artviscy(i,j)=0;
end

for j=1:jmax
    i=1;
     artviscx(i,j)=0;
    artviscy(i,j)=0;
end
for j=1:jmax
    i=imax;
    artviscx(i,j)=0;
    artviscy(i,j)=0;
end

%2nd outermost layers
for i=3:imax-2
    j=2;
    artviscy(i,2)=artviscy(i,3);

    uvel2=u(i,j,2)^2+u(i,j,3)^2;
    beta2=max(uvel2, rkappa*vel2ref);
   d4pdx4=(u(i+2,j,1)-4*u(i+1,j,1)+6*u(i,j,1)-4*u(i-1,j,1)+u(i-
2,j,1))/dx^4;
    lambda_x= .5*(abs(u(i,j,2))+sqrt(u(i,j,2)^2+4*beta2));
    artviscx(i,j)=-(lambda_x*Cx*dx^3/beta2)*d4pdx4;
end

for i=3:imax-2
    j=jmax-1;
    artviscy(i,j)=artviscy(i,j-1);

    uvel2=u(i,j,2)^2+u(i,j,3)^2;
    beta2=max(uvel2, rkappa*vel2ref);
    d4pdx4=(u(i+2,j,1)-4*u(i+1,j,1)+6*u(i,j,1)-4*u(i-1,j,1)+u(i-
2,j,1))/dx^4;
    lambda_x= .5*(abs(u(i,j,2))+sqrt(u(i,j,2)^2+4*beta2));
    artviscx(i,j)=-(lambda_x*Cx*dx^3/beta2)*d4pdx4;
end

for j=3:jmax-2
    i=2;
    artviscx(i,j)=artviscx(i+1,j);
     uvel2=u(i,j,2)^2+u(i,j,3)^2;
    beta2=max(uvel2, rkappa*vel2ref);
    lambda_y= .5*(abs(u(i,j,3))+sqrt(u(i,j,3)^2+4*beta2));
    artviscy(i,j)=-(lambda_y*Cy*dy^3/beta2)*d4pdy4;
end
```

```
for j=3:jmax-2
    i=imax-1;
    artviscx(i,j)=artviscx(i-1,j);
    uvel2=u(i,j,2)^2+u(i,j,3)^2;
    beta2=max(uvel2, rkappa*vel2ref);
    lambda_y= .5*(abs(u(i,j,3))+sqrt(u(i,j,3)^2+4*beta2));
    artviscy(i,j)=-(lambda_y*Cy*dy^3/beta2)*d4pdy4;
end


%%Corners
artviscx(2,2)=artviscx(3,2);
artviscx(imax-1,jmax-1)=artviscx(imax-2,jmax-1);
artviscx(2,jmax-1)=artviscx(3,jmax-1);
artviscx(imax-1,2)=artviscx(imax-2,2);
artviscy(2,2)=artviscy(2,3);
artviscy(imax-1,jmax-1)=artviscy(imax-1,jmax-2);
artviscy(2,jmax-1)=artviscy(2,jmax-2);
artviscy(imax-1,2)=artviscy(imax-1,3);
%}


end

End
```

Appendix B

```
function [dtmin] = compute_time_step(dtmin)
%
%Uses global variable(s): one, two, four, half, fourth
%Uses global variable(s): vel2ref, rmu, rho, dx, dy, cfl, rkappa,
imax, jmax
%Uses: u
%To Modify: dt, dtmin

% i                         % i index (x direction)
% j                         % j index (y direction)

% dtvisc       % Viscous time step stability criteria (constant over
domain)
% uvel2        % Local velocity squared
% beta2        % Beta squared paramete for time derivative
preconditioning
% lambda_x     % Max absolute value eigenvalue in (x,t)
% lambda_y     % Max absolute value eigenvalue in (y,t)
% lambda_max   % Max absolute value eigenvalue (used in convective
time step computation)
% dtconv       % Local convective time step restriction

global four half fourth
global vel2ref rmu rho dx dy cfl rkappa imax jmax
global u dt
dtvisc=dx*dy*fourth/(rmu/rho);

for i = 2:(imax-1)
    for j=2:(jmax-1)

            uvel2=u(i,j,2)^2+u(i,j,3)^2;
            beta2=max(uvel2, rkappa*1);
            lambda_x=
half*(abs(u(i,j,2))+(u(i,j,2)^2+four*beta2)^half);
            lambda_y=
half*(abs(u(i,j,3))+(u(i,j,3)^2+four*beta2)^half);
            lambda_max= max(lambda_x,lambda_y);
            dtconv=min(dx,dy)/lambda_max;

            %Local Time Step
            dt(i,j)=cfl*min(dtconv, dtvisc);

                   %Global Time Step
            dtmin=min(dt(i,j),dtmin);

    end
end

  end
```

Appendix C

```
function Compute_Artificial_Viscosity(~)
%
%Uses global variable(s): zero, one, two, four, six, half, fourth
%Uses global variable(s): imax, jmax, lim, rho, dx, dy, Cx, Cy, Cx2,
Cy2, fsmall, vel2ref, rkappa
%Uses: u
%To Modify: artviscx, artviscy
 % i                          % i index (x direction)
% j                         % j index (y direction)
 % uvel2         % Local velocity squared
% beta2         % Beta squared parameter for time derivative
preconditioning
% lambda_x      % Max absolute value e-value in (x,t)
% lambda_y      % Max absolute value e-value in (y,t)
% d4pdx4        % 4th derivative of pressure w.r.t. x
% d4pdy4        % 4th derivative of pressure w.r.t. y
% % d2pdx2        % 2nd derivative of pressure w.r.t. x [these are not
used]
% % d2pdy2        % 2nd derivative of pressure w.r.t. y [these are not
used]
% % pfunct1       % Temporary variable for 2nd derivative damping
[these are
% not used]
% % pfunct2       % Temporary variable for 2nd derivative damping

global two four six half
global imax jmax lim rho dx dy Cx Cy Cx2 Cy2 fsmall vel2ref rkappa
global u
global artviscx artviscy

 % !********************************************************** */
for j=3:jmax-2
       for i=3:imax-2

    uvel2=u(i,j,2)^2+u(i,j,3)^2;
    d4pdy4=(u(i,j+2,1)-4*u(i,j+1,1)+6*u(i,j,1)-4*u(i,j-1,1)+…
u(i,j- 2,1))/dy^4;
    d4pdx4=(u(i+2,j,1)-4*u(i+1,j,1)+6*u(i,j,1)-4*u(i-1,j,1)+…
u(i-2,j,1))/dx^4;
    beta2=max(uvel2, rkappa*vel2ref);
    lambda_x= .5*(abs(u(i,j,2))+sqrt(u(i,j,2)^2+4*beta2));
    lambda_y= .5*(abs(u(i,j,3))+sqrt(u(i,j,3)^2+4*beta2));

    artviscx(i,j)=-(lambda_x*Cx*dx^3/beta2)*d4pdx4;
    artviscy(i,j)=-(lambda_y*Cy*dy^3/beta2)*d4pdy4;

       end
end
```

```
function SGS_forward_sweep(~)
%
%Uses global variable(s): two, three, six, half
%Uses global variable(s): imax, imax, jmax, ipgorder, rho, rhoinv, dx,
dy, rkappa, ...
%                        xmax, xmin, ymax, ymin, rmu, vel2ref
%Uses: artviscx, artviscy, dt, s
%To Modify: u

% i                          % i index (x direction)
% j                          % j index (y direction)

% dpdx          % First derivative of pressure w.r.t. x
% dudx          % First derivative of x velocity w.r.t. x
% dvdx          % First derivative of y velocity w.r.t. x
% dpdy          % First derivative of pressure w.r.t. y
% dudy          % First derivative of x velocity w.r.t. y
% dvdy          % First derivative of y velocity w.r.t. y
% d2udx2        % Second derivative of x velocity w.r.t. x
% d2vdx2        % Second derivative of y velocity w.r.t. x
% d2udy2        % Second derivative of x velocity w.r.t. y
% d2vdy2        % Second derivative of y velocity w.r.t. y
% beta2         % Beta squared parameter for time derivative
preconditioning
% uvel2         % Velocity squared

global two half
global imax jmax rho rhoinv dx dy rkappa rmu vel2ref
global artviscx artviscy dt s u uold

% Symmetric Gauss-Siedel: Forward Sweep
for j=2:jmax-1
       for i=2:imax-1


uvel2=u(i,j,2)^2+u(i,j,3)^2;
beta2=max(uvel2, rkappa*vel2ref);


 dpdx    =  (uold(i+1,j,1)-u(i-1,j,1))*half/dx;
 dudx    =  (uold(i+1,j,2)-u(i-1,j,2))*half/dx;
 dvdx    =  (uold(i+1,j,3)-u(i-1,j,3))*half/dx;
 dpdy    =  (uold(i,j+1,1)-u(i,j-1,1))*half/dy;
 dudy    =  (uold(i,j+1,2)-u(i,j-1,2))*half/dy;
 dvdy    =  (uold(i,j+1,3)-u(i,j-1,3))*half/dy;
 d2udx2  =  (uold(i+1,j,2)-2*u(i,j,2)+u(i-1,j,2))/dx^2;
 d2vdx2  =  (uold(i+1,j,3)-2*u(i,j,3)+u(i-1,j,3))/dx^2;
 d2udy2  =  (uold(i,j+1,2)-2*u(i,j,2)+u(i,j-1,2))/dy^2;
 d2vdy2  =  (uold(i,j+1,3)-2*u(i,j,3)+u(i,j-1,3))/dy^2;

u(i,j,1)=u(i,j,1)-beta2*dt(i,j)*(rho*dudx+rho*dvdy-…
     artviscx(i,j)-artviscy(i,j)-s(i,j,1));
```

```
u(i,j,2)=u(i,j,2)-dt(i,j)/rho*(rho*u(i,j,2)*dudx+rho*…
      u(i,j,3)*dudy+dpdx-rmu*d2udx2-rmu*d2udy2-s(i,j,2));
u(i,j,3)=u(i,j,3)-dt(i,j)/rho*(rho*u(i,j,3)*dvdx+rho*u(i,j,2)…
      *dvdy+dpdy-rmu*d2vdx2-rmu*d2vdy2-s(i,j,3));

       end
end


function SGS_backward_sweep(~)
%
%Uses global variable(s): two, three, six, half
%Uses global variable(s): imax, imax, jmax, ipgorder, rho, rhoinv, dx,
dy, rkappa, ...
%                        xmax, xmin, ymax, ymin, rmu, vel2ref
%Uses: artviscx, artviscy, dt, s
%To Modify: u

% i                          % i index (x direction)
% j                          % j index (y direction)

% dpdx          % First derivative of pressure w.r.t. x
% dudx          % First derivative of x velocity w.r.t. x
% dvdx          % First derivative of y velocity w.r.t. x
% dpdy          % First derivative of pressure w.r.t. y
% dudy          % First derivative of x velocity w.r.t. y
% dvdy          % First derivative of y velocity w.r.t. y
% d2udx2        % Second derivative of x velocity w.r.t. x
% d2vdx2        % Second derivative of y velocity w.r.t. x
% d2udy2        % Second derivative of x velocity w.r.t. y
% d2vdy2        % Second derivative of y velocity w.r.t. y
% beta2         % Beta squared parameter for time derivative
preconditioning
% uvel2         % Velocity squared

global two half
global imax jmax rho rhoinv dx dy rkappa rmu vel2ref
global artviscx artviscy dt s u uold

% Symmetric Gauss-Siedel: Backward Sweep
for i=(imax-1):-1:2
       for j=(jmax-1):-1:2


uvel2=u(i,j,2)^2+u(i,j,3)^2;
beta2=max(uvel2, rkappa*vel2ref);


 dpdx    =  (u(i+1,j,1)-u(i-1,j,1))*half/dx;
 dudx    =  (u(i+1,j,2)-u(i-1,j,2))*half/dx;
 dvdx    =  (u(i+1,j,3)-u(i-1,j,3))*half/dx;
 dpdy    =  (u(i,j+1,1)-u(i,j-1,1))*half/dy;
```

```
 dudy    =  (u(i,j+1,2)-u(i,j-1,2))*half/dy;
 dvdy    =  (u(i,j+1,3)-u(i,j-1,3))*half/dy;
 d2udx2  =  (u(i+1,j,2)-2*u(i,j,2)+u(i-1,j,2))/dx^2;
 d2vdx2  =  (u(i+1,j,3)-2*u(i,j,3)+u(i-1,j,3))/dx^2;
 d2udy2  =  (u(i,j+1,2)-2*u(i,j,2)+u(i,j-1,2))/dy^2;
 d2vdy2  =  (u(i,j+1,3)-2*u(i,j,3)+u(i,j-1,3))/dy^2;


u(i,j,1)=u(i,j,1)-beta2*dt(i,j)*(rho*dudx+rho*dvdy-…
      artviscx(i,j)-artviscy(i,j)-s(i,j,1));
u(i,j,2)=u(i,j,2)-dt(i,j)/rho*(rho*u(i,j,2)*dudx+rho*u(i,j,3)*dudy+…
      dpdx-rmu*d2udx2-rmu*d2udy2-s(i,j,2));
u(i,j,3)=u(i,j,3)dt(i,j)/rho*(rho*u(i,j,3)*dvdx+rho*u(i,j,2)*dvdy
      +dpdy-rmu*d2vdx2-rmu*d2vdy2-s(i,j,3));

        end
end
% !************************************************************** */
```

Appendix D

```
function point_Jacobi(~)
%Uses global variable(s): two, three, six, half
%Uses global variable(s): imax, imax, jmax, ipgorder, rho, rhoinv, dx,
dy, rkappa,max, xmin, ymax, ymin, rmu, vel2ref
%Uses: uold, artviscx, artviscy, dt, s
%To Modify: u
% i                         % i index (x direction)
% j                         % j index (y direction)
% dpdx          % First derivative of pressure w.r.t. x
% dudx          % First derivative of x velocity w.r.t. x
% dvdx          % First derivative of y velocity w.r.t. x
% dpdy          % First derivative of pressure w.r.t. y
% dudy          % First derivative of x velocity w.r.t. y
% dvdy          % First derivative of y velocity w.r.t. y
% d2udx2        % Second derivative of x velocity w.r.t. x
% d2vdx2        % Second derivative of y velocity w.r.t. x
% d2udy2        % Second derivative of x velocity w.r.t. y
% d2vdy2        % Second derivative of y velocity w.r.t. y
% beta2         % Beta squared parameter for time derivative
preconditioning
% uvel2         % Velocity squared
global two half
global imax jmax rho rhoinv dx dy rkappa rmu vel2ref
global u uold artviscx artviscy dt s

% Point Jacobi method
for i=2:imax-1
       for j=2:jmax-1
 uvel2   =   uold(i,j,2)^2+uold(i,j,3)^2;
 beta2   =   max(uvel2, rkappa*vel2ref);
 dpdx    =  (uold(i+1,j,1)-uold(i-1,j,1))*half/dx;
 dudx    =  (uold(i+1,j,2)-uold(i-1,j,2))*half/dx;
 dvdx    =  (uold(i+1,j,3)-uold(i-1,j,3))*half/dx;
 dpdy    =  (uold(i,j+1,1)-uold(i,j-1,1))*half/dy;
 dudy    =  (uold(i,j+1,2)-uold(i,j-1,2))*half/dy;
 dvdy    =  (uold(i,j+1,3)-uold(i,j-1,3))*half/dy;
 d2udx2  =  (uold(i+1,j,2)-2*uold(i,j,2)+uold(i-1,j,2))/dx^2;
 d2vdx2  =  (uold(i+1,j,3)-2*uold(i,j,3)+uold(i-1,j,3))/dx^2;
 d2udy2  =  (uold(i,j+1,2)-2*uold(i,j,2)+uold(i,j-1,2))/dy^2;
 d2vdy2  =  (uold(i,j+1,3)-2*uold(i,j,3)+uold(i,j-1,3))/dy^2;
  u(i,j,1)=uold(i,j,1)-beta2*dt(i,j)*(rho*dudx+rho*dvdy-…
     artviscx(i,j)-artviscy(i,j)-s(i,j,1));
 u(i,j,2)=uold(i,j,2)-dt(i,j)/rho*(rho*uold(i,j,2)*dudx+rho…
     *uold(i,j,3)*dudy+dpdx-rmu*d2udx2-rmu*d2udy2-s(i,j,2));
 u(i,j,3)=uold(i,j,3)-dt(i,j)/rho*(rho*uold(i,j,2)*dvdx+rho…
     *uold(i,j,3)*dudy+dpdy-rmu*d2vdx2-rmu*d2vdy2-s(i,j,3));

       end
end
```

Appendix E

```
 function [res, resinit, conv] = check_iterative_convergence...
    (n, res, resinit, ninit, rtime, dtmin)
%
%Uses global variable(s): zero
%Uses global variable(s): imax, jmax, neq, fsmall
%Uses: n, u, uold, dt, res, resinit, ninit, rtime, dtmin
%To modify: conv

% i                          % i index (x direction)
% j                          % j index (y direction)
% k                          % k index (# of equations)

global zero
global imax jmax neq fsmall
global u uold dt fp1
 % Compute iterative residuals to monitor iterative convergence
 %Residual Matrix
for j=1:jmax
    for i=1:imax
        for k=1:3
           resM(i,j,k)=abs((u(i,j,k)-uold(i,j,k)))/dt(i,j);
         end
    end
end
res(1)=norm(resM(:,:,1));
res(2)=norm(resM(:,:,2));
res(3)=norm(resM(:,:,3));
 %Initial Residual
if n>1 && n<5
     resinit(1)=max(resinit(1),res(1));
     resinit(2)=max(resinit(2),res(2));
     resinit(3)=max(resinit(3),res(3));
end
conv=max(res/resinit);

 % Write iterative residuals every 10 iterations
if ( (mod(n,10)==0)||(n==ninit) )
    fprintf(fp1, '%d   %e   %e   %e   %e\n',  n, rtime, res(1),
res(2), res(3));
end
 % Write header for iterative residuals every 200 iterations
if ( (mod(n,200)==0)||(n==ninit) )
    fprintf('Iter.    Time (s)       dt (s)      Continuity      x-
Momentum       y-Momentum\n');
    fprintf('%d   %e   %e   %e   %e   %e\n',  n, rtime, dtmin, res(1),
res(2), res(3));
end

end
```

Appendix F

```
function Discretization_Error_Norms(rL1norm, rL2norm, rLinfnorm)
%
%Uses global variable(s): zero
%Uses global variable(s): imax, jmax, neq, imms, xmax, xmin, ymax,
ymin, rlength
%Uses: u
%To modify: rL1norm, rL2norm, rLinfnorm
% i                          % i index (x direction)
% j                          % j index (y direction)
% k                          % k index (# of equations)

% x        % Temporary variable for x location
% y        % Temporary variable for y location
% DE    % Discretization error (absolute value)

global zero imax jmax neq imms xmax xmin ymax ymin u ummsArray

if imms==1
for k=1:neq
    for i=1:imax
        for j=1:jmax
          DE(i,j,k)=abs(u(i,j,k)-ummsArray(i,j,k));
         end
    end
end
  rL1norm(1)=norm(DE(:,:,1),1);
rL2norm(1)=norm(DE(:,:,1),2);
rLinfnorm(1)=norm(DE(:,:,1) ,Inf);
 rL1norm(2)=norm(DE(:,:,2)    ,1);
rL2norm(2)=norm(DE(:,:,2),2);
rLinfnorm(2)=norm( DE(:,:,2) ,Inf);
 rL1norm(3)=norm(DE(:,:,3) , 1);
rL2norm(3)=norm(DE(:,:,3) ,2)
rLinfnorm(3)=norm(DE(:,:,3),Inf);
fp5= fopen('DEnorms.out','w')

fprintf(fp5, '"Mesh""pL1""pL2""pInf""uL1""uL2""uInf""vL1""vL2""vInf"\n');
fprintf(fp5,' %e %e %e %e %e %e %e %e %e %e\n', imax,
rL1norm(1),rL2norm(1),rLinfnorm(1),rL1norm(2), ...
           rL2norm(2),rLinfnorm(2),rL1norm(3),rL2norm(3),rLinfnorm(3));
fclose(fp5);


fp5= fopen('DEnormsL2.out','w')

fprintf(fp5, '"Mesh""pL1""pL2""pInf""uL1""uL2""uInf""vL1""vL2""vInf"\n');
fprintf(fp5,' %e %e %e %e %e %e %e %e %e %e\n', imax,
rL1norm(1),rL2norm(1),rLinfnorm(1),rL1norm(2), ...
           rL2norm(2),rLinfnorm(2),rL1norm(3),rL2norm(3),rLinfnorm(3));
fclose(fp5);
end
```