Numerical simulation of the Lid-driven Cavity Benchmark Problem

Jagannath Suresh and Shrikar Banagiri
Department of Mechanical Engineering, Virginia Tech.

The lid-driven cavity problem is a benchmark flow case in CFD due its simplicity and the unambiguous nature of its boundary conditions. Considerable work has been done since the 1980s in this area, particularly with the increase of mesh refinement and Reynolds numbers. In this report, a second-order accurate simple explicit discretization scheme was used to simulate the steady pressure and flow fields at various mesh sizes and Reynolds numbers. An order of accuracy test was conducted to verify if the observed order of accuracy matched the formal order. The mathematical correctness of the code was tested by using manufactured solutions. Results showed that the discretization scheme passed the order of accuracy test. For coarser meshes, numerical dissipation was observed in the streamlines, i.e., a discontinuity in the streamlines was observed on coarser meshes. However, this numerical dissipation diminished when the grid refinement factor was increased and/or the Reynolds number was increased. The code solutions were compared with those of ANSYS Fluent. This comparison showed excellent agreement between the two results.

I. INTRODUCTION

The lid-driven cavity problem was first posed by Kawaguti in the Journal of the Physical Society of Japan in 1961.[1] The problem statement is as follows. A 2-D rectangular (or square) cavity is filled with an incompressible fluid. The top edge is a lid which slides at a uniform velocity while the other three edges are stationary. Since Kawaguti's original paper, many studies have been published about this problem. Erturk et al.[2] performed numerical simulations on a grid size of 601x601 with Reynolds numbers as high as 21000. Marchi et al.[3] generated detailed numerical solutions on a 8192x8192 grid for a wide range of Reynolds numbers. Across all the studies, two important flow regimes were noted. One was the presence of primary and secondary vortices (around the center and lower corners of the domain respectively), and the occurrence of singularities at the corners of the domain.

In this study, we have used a simple explicit discretization scheme (Point Jacobi and Gauss-Siedel iterative methods) to solve the governing equations on mesh sizes of 33x33, 65x65, 95x95, 101x101, and 129x129. Code validation was done by comparing the results with ANSYS Fluent. An 81x81 grid was used for validation with Fluent. The Navier-Stokes equations were modified by introducing pressure-based time derivative pre-conditioning in the continuity equation in order to reduce the condition number of the system.

Artificial viscosity was also introduced to mitigate odd-even decoupling in the system. Since we are interested in the steady state solution, a first order accurate discretization scheme was used with respect to time.

II. THEORY

A. Governing Equations

The fluid in the 2D lid driven cavity is considered to be incompressible and hence the necessary modification is made in the governing equations. Since the primary objective of the study is to understand the pressure and velocity fields in the 2D lid driven cavity, the governing equations for the study are the equations of continuity and momentum conservation for a 2 dimensional steady incompressible flow along with time derivative pre-conditioning which are expressed below.

- **Continuity Equation**

$$\frac{1}{\beta^2}\frac{\partial p}{\partial t} + \rho[\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}] - S = f_{mass}(x, y) \tag{1}$$

- **X Momentum Equation**

$$\rho\frac{\partial u}{\partial t} + \rho[u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}] - \mu[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}] + \frac{\partial p}{\partial x} = f_{xmtm}(x, y) \tag{2}$$

- **Y Momentum Equation**

$$\rho\frac{\partial v}{\partial t} + \rho[u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}] - \mu[\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}] + \frac{\partial p}{\partial y} = f_{ymtm}(x, y) \tag{3}$$

where ρ, p, u and v are the density, pressure, X-velocity and Y Velocity respectively.

$f_{mass}(x, y)$, $f_{xmtm}(x, y)$ and $f_{ymtm}(x, y)$ are the manufactured solution source terms for the respective equations.

1. Time Derivative Preconditioning

Time derivative preconditioning is a method where a temporal derivative of pressure is added to the continuity equation as done in Equation (1). A factor of $1/\beta^2$ is multiplied with it to make the equation dimensionally consistent.We are allowed to do this as we are only interested in the steady state solutions and at steady state we recover the original equations.

- The value of β is chosen to be equal to max($u^2 + v^2$, K($u_\infty^2 + v_\infty^2$)) where u,v are the velocities at the respective nodes i,j and u_∞,v_∞ are the free stream velocities. The value of the constant K is chosen to be a small number. Choosing the maximum function for the value of β limits it from getting too small.

2. Artificial Viscocity

The source term S in the continuity equation (Equation (1)) is the artificial viscocity term. This term is added to the continuity equation to prevent the occurence of odd-even decoupling. Odd even decoupling occurs when a lower order scheme is used to discretize the equation and this results in a jump between 2 indices resulting in the decoupling of the even and odd cells in the whole domain. Adding of the artificial viscocity term to the continuity equation helps fix this problem.

The articial viscocity source term is given by:

$$S = - \mid \lambda_{xmax} \mid (\frac{C^{(4)}}{\beta^2})\Delta x^3(\frac{\partial^4 p}{\partial x^4}) - \mid \lambda_{ymax} \mid (\frac{C^{(4)}}{\beta^2})\Delta y^3(\frac{\partial^4 p}{\partial y^4}) \tag{4}$$

where $\mid \lambda_{xmax} \mid$ is the eigenvalue of (x,t) with the largest magnitude which is given by:

$$\mid \lambda_{xmax} \mid = \frac{|u| + \sqrt{u^2 + 4\beta^2}}{2} \tag{5}$$

and $C^{(4)}$ typically varies from $\frac{1}{128}$ to $\frac{1}{16}$

B. Boundary Conditions

A set of equations used along a domain boundary to obtain a particular solution to the problem is called a boundary condition. The equation is solved and it's known variables influence the boundary condition for a given physical condition. For example on a physical "Wall", the boundary condition for the momentum equation could be a slip or a no-slip condition, while for the energy equation it could be a specified flux, an imposed temperature, or a convection heat transfer condition. The general categories of boundary conditions are mentioned below:

- a Dirichlet condition, where the unknown variable is defined at the boundary;
- a von Neumann condition, where the flux expressed in the conservation equation is defined at the boundary face;
- a Robin-type condition, where the unknown variable and flux at the boundary are expressed via a constitutive relation

The boundary conditions we use in the present study are:

1. Bottom and side walls: u = v = 0 or the no slip boundary condition
2. Top Wall: u = U_{lid} and v = 0.
3. For the pressure at the wall, linear extrapolation is used from the interior nodes: for example - $P^n_{i,1} = 2P^n_{i,2} - P^n_{i,3}$. The other 3 walls are also handled similarly.

III. MATHEMATICAL MODELING

A. Discretization

The baseline discretization approach used in this study is the simple explicit method.The temporal derivatives are replaced with a forward difference in time and the spatial derivatives are replaced with central differences.

For the continuity equation, this results in:

$$P_{i,j}^{n+1} = P_{i,j}^{n} - \beta_{i,j}^{2}\Delta t[\rho\frac{u_{i+1,j}^{n} - u_{i-1,j}^{n}}{2\Delta x} + \rho\frac{v_{i,j+1}^{n} - v_{i,j-1}^{n}}{2\Delta y} - S_{i,j} - f_{mass}(x,y)] \quad (6)$$

For the x-momentum equation, this results in:

$$u_{i,j}^{n+1} = u_{i,j}^{n} - \frac{\Delta t}{\rho}[\rho u_{i,j}^{n}\frac{u_{i+1,j}^{n} - u_{i-1,j}^{n}}{2\Delta x} + \rho v_{i,j}^{n}\frac{u_{i,j+1}^{n} - u_{i,j-1}^{n}}{2\Delta y} + \frac{P_{i+1,j}^{n} - P_{i-1,j}^{n}}{2\Delta x} - \mu\frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^{2}} - \mu\frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^{2}} - f_{xmtm}(x,y)] \quad (7)$$

The y momentum equation is also discretized similarly and this results in:

$$v_{i,j}^{n+1} = v_{i,j}^{n} - \frac{\Delta t}{\rho}[\rho u_{i,j}^{n}\frac{v_{i+1,j}^{n} - v_{i-1,j}^{n}}{2\Delta x} + \rho v_{i,j}^{n}\frac{v_{i,j+1}^{n} - v_{i,j-1}^{n}}{2\Delta y} + \frac{P_{i+1,j}^{n} - P_{i-1,j}^{n}}{2\Delta y} - \mu\frac{v_{i+1,j}^{n} - 2v_{i,j}^{n} + v_{i-1,j}^{n}}{\Delta x^{2}} - \mu\frac{v_{i,j+1}^{n} - 2v_{i,j}^{n} + v_{i,j-1}^{n}}{\Delta y^{2}} - f_{ymtm}(x,y)] \quad (8)$$

B. Stability

The time step for the iterations should be chosen without compromising the stability of the method. Stability for this method comes from a combination of the convective stability limit and the diffusive limit.

The convective stability stability limit for the time step is:

$$\Delta t_{c} \leq \frac{min(\Delta x, \Delta y)}{\mid \lambda_{max} \mid} \quad (9)$$

Where $\mid \lambda_{max} \mid = \max(\lambda_{x}, \lambda_{y})$ and

$$\lambda_{x} = \frac{\mid u \mid + \sqrt{u^{2} + 4\beta^{2}}}{2} \quad (10)$$

$$\lambda_{y} = \frac{\mid v \mid + \sqrt{v^{2} + 4\beta^{2}}}{2} \quad (11)$$

The diffusive stability limit for the time step is:

$$\Delta t_d \leq \frac{\Delta x \Delta y}{4\nu}, \qquad \nu = \frac{\mu}{\rho} \tag{12}$$

The stability limit of the time step is obtained by combining the convective and diffusive time step limits and adding the CFL number:

$$\Delta t \leq min(\Delta t_c, \Delta t_d) = CFL * min(\Delta t_c, \Delta t_d) \tag{13}$$

In this study, the CFL values were varied between zero and one.

IV. RESULTS

A. Code Verification

The velocity profiles display an expected trend, i.e., the u-velocity steadily rises as the distance from the bottom wall increases and the v-velocity decreases.

1. Order of accuracy testing

Code verification was performed using the method of manufactured solutions. A series of sine and cosine functions were used as source terms for the manufactured solutions. The order of accuracy test, which is often called as the most stringent code verification test, was applied to the current CFD code. Three different discretization norms, i.e., L1 norm, L2 norm, and L infinity norm were computed for the following mesh sizes: 33x33, 65x65, 81x81, 95x95, 101x101, and 129x129. The following figure 12 shows the variation of the discretization error (DE) norms as a function of the grid spacing (h) on a log scale. Note: The grid spacing of the finest mesh (129x129) is taken to be 1.

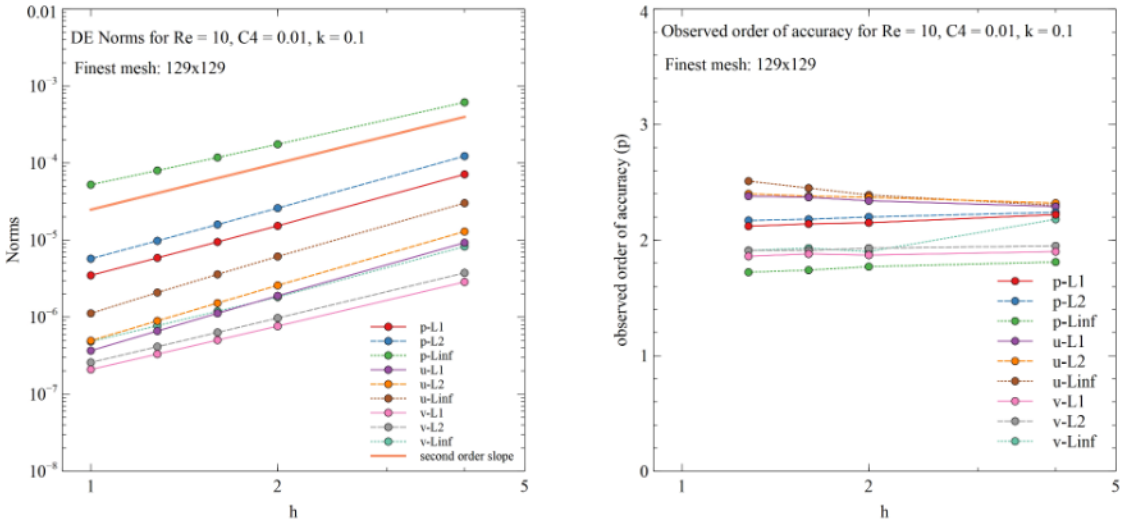
From the figure 12, it is quite clear that the norms have a slope similar to the second order line shown. Thus, it may be surmised that the observed order of accuracy of the scheme is close to 2, which matches the formal order of accuracy. For a better understanding of the observed order of accuracy of each norm, figure 12 (in the appendix) shows the observed order of accuracy of the norms as a function of the grid spacing. From this figure, it can be concluded that the observed order of accuracy approaches 2 as the mesh is refined. Furthermore, the observed order of accuracy for the L infinity pressure norm drops slightly below 2. This decrease can be attributed to the pressure singularities in the top left and top right corners of the domain. However, this singularity is not quite prevalent in figure 12 due to the moderately high Reynolds number. In order to capture the effect of the pressure singularities correctly, we ran the case files for Re = 10. Figure 1 shows the DE norms and observed order of accuracy for this case.The observed order of accuracy was calculated using

the following formula:

$$p = \frac{\ln \frac{||\epsilon_2||}{||\epsilon_1||}}{\ln r} \tag{14}$$

where $||\epsilon_2||$ and $||\epsilon_1||$ are the discretisation error norms of the coarse and fine mesh respectively and r is the grid refinement factor.

From the observed order of accuracy graph, it is quite clear that the Linf norm captures the singularities well (the order of accuracy for Linf norm of pressure is 1.7).



(a) DE Norms as a function of grid spacing (h) for Re = 10

(b) Observed order of accuracy as a function of grid spacing (h) for Re = 10

Figure 1: DE norms and observed order of accuracy for Re = 10

2. Effect of C4 constant

The code couples the solutions at odd and even nodes through the addition of an artificial viscosity parameter. The effective artificial viscosity is directly proportional to the C4 constant. Thus, odd-even decoupling would reduce by increasing the value of C4. Figure 2 shows the effect of C4 values on the discretization error in pressure. At the lowest C4 value (0.008), the odd-even decoupling is the highest whereas the peaks start to vanish when C4 is increased to 0.05 and the curve smoothens out at the maximum value of 0.0625.

3. Iterative Residual History and Effect of κ

The iterative residual history was plotted for two different mesh sizes; 65x65 and 129x129. The figure 3 shows the convergence history for the Point Jacobi and Symmetric Gauss-Siedel

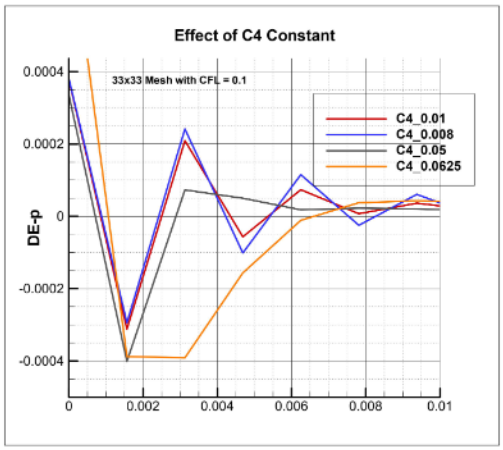


Figure 2: Effect of C4 constant on the Discretization Error of Pressure

schemes. For the 65x65 mesh, both the schemes converge at similar speeds. However, when the mesh is refined to 129x129, the Symmetric Gauss-Siedel scheme converges faster than the Point Jacobi scheme. The Symmetric Gauss-Siedel scheme converged 2000 iterations before the Point Jacobi scheme. Figure 4 shows the effect of the constant κ on the iterative convergence. Until the value of κ was increased to 0.5, its value had minimal effect on the iterative residuals. However, beyond the value of 0.5, the iterative convergence gets faster. In the figure, the fastest convergence occurs at $\kappa = 0.9$.

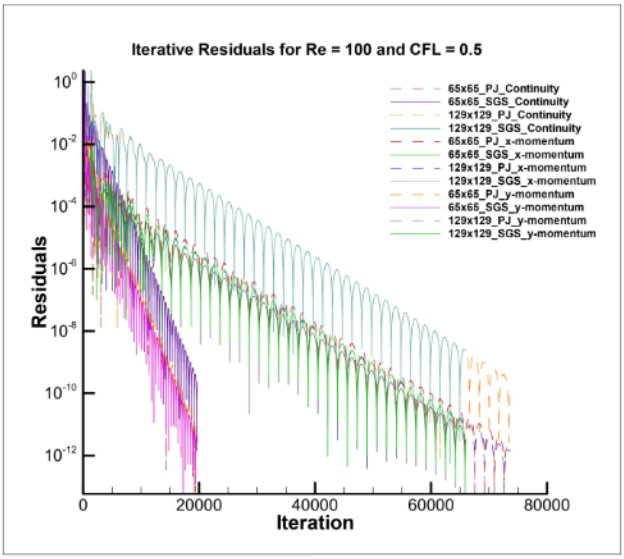
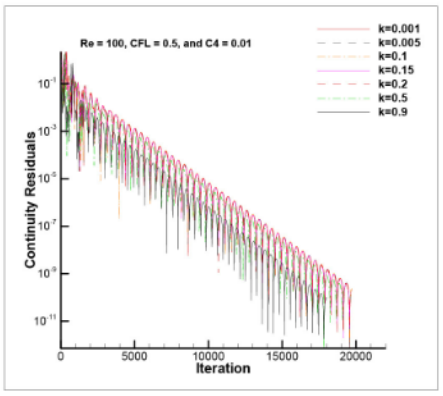
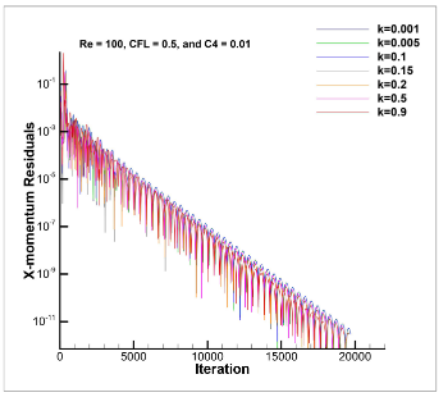


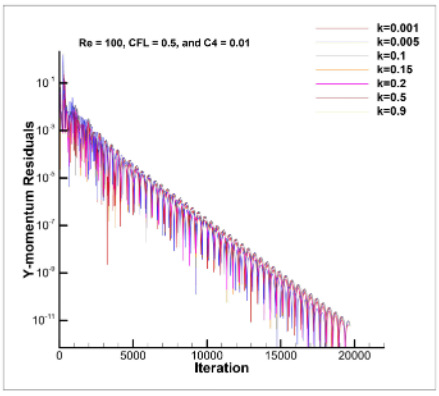
Figure 3: Iterative residual history for Point Jacobi and Symmetric Gauss-Siedel schemes



(a) Continuity residuals



(b) X-Momentum residuals



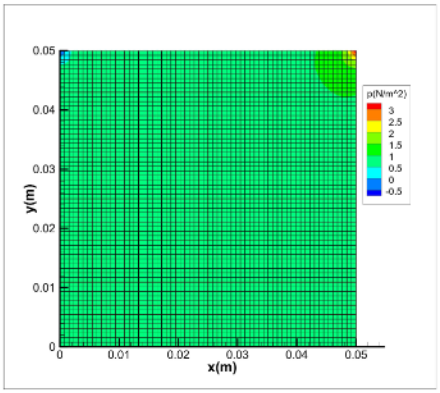
(c) Y-Momentum residuals

Figure 4: Effect of κ on the iterative residuals

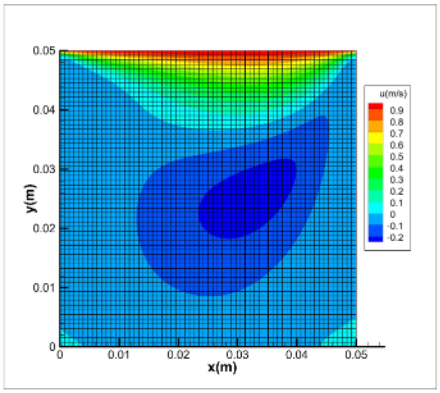
B. Flow-field Solutions

1. Comparison of numerical and MMS solutions

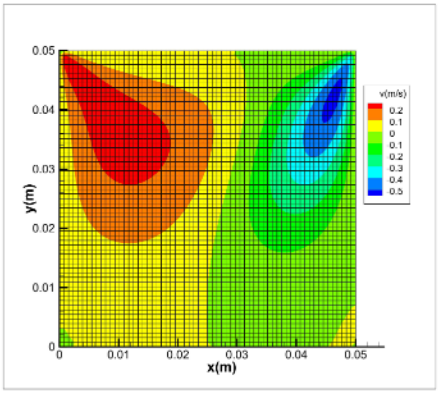
The figure 5shows the contour mesh profiles of pressure, u-velocity, and v-velocities computed on a 65x65 grid with a Reynolds number of 100. The u-velocity, v-velocity, and pressure profiles for the manufactured solution case are shown below in figure 10. The contours demonstrate the key differences between the manufactured and numerical solutions. Firstly, since the manufactured source terms were trigonometric in nature, the manufactured solutions were parts of sine or cos curves. On the contrary, the numerical solution admits vortices in the center of the domain and eddies in the corners of the domain/ Furthermore, there are strong pressure singularities in the top left and top right corners of the domain.



(a) pressure

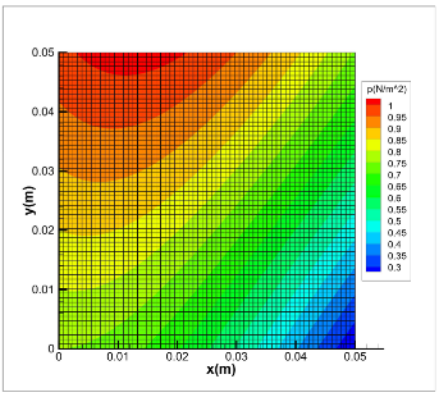


(b) u-velocity

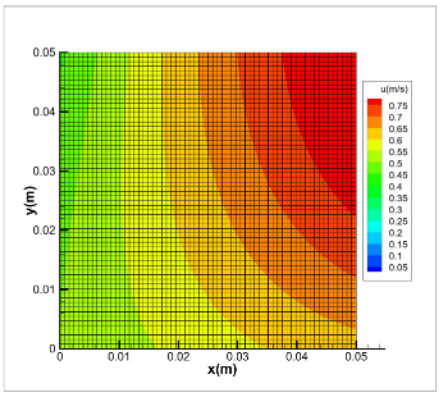


(c) v-velocity

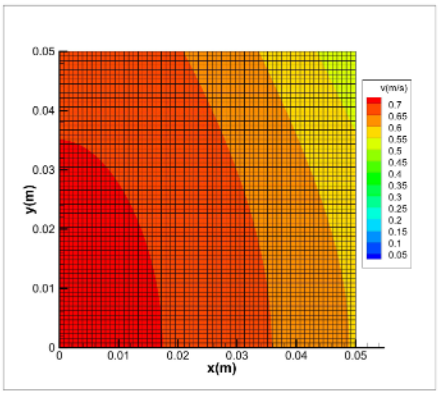
Figure 5: Pressure (Pascals), u-velocity (m/s), and v-velocity (m/s)



(a) pressure



(b) u-velocity

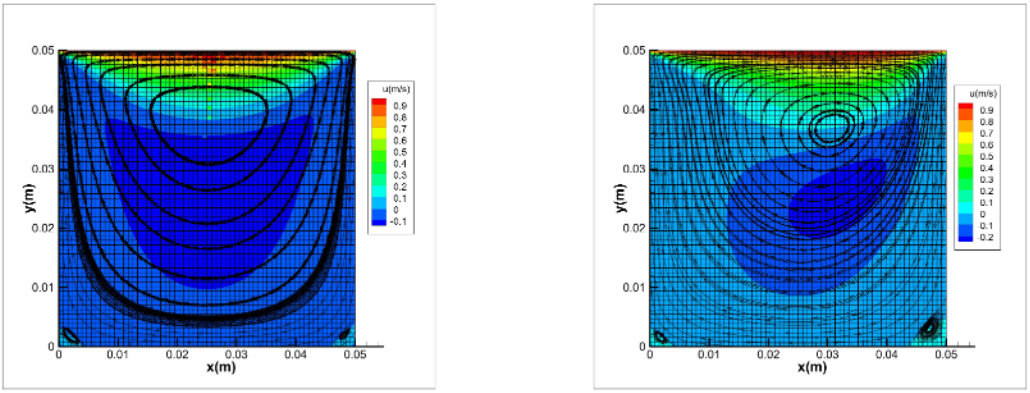


(c) v-velocity

Figure 6: MMS Pressure (Pascals), MMS u-velocity (m/s), and MMS v-velocity (m/s)

2. Effect of Reynolds number on streamlines

Figure 7 shows the effect of Reynolds number on a 65x65 mesh. The Reynolds numbers employed were 5, 100, 500, and 1000. The Reynolds number of 1000 was run on a finer mesh (129x129) since the discretization scheme was becoming unstable on the 65x65 grid. Generally, the streamlines became tighter as the Reynolds number increased. This tightness can be explained by a reduction in viscous dissipations. On a similar note, the circularity of the streamlines also increased. Ah higher Reynolds numbers, the vortices at the center of the domain are well-formed when compared to lower Reynolds numbers.[3]



(a) Re = 5

(b) Re = 100

(c) Re = 500

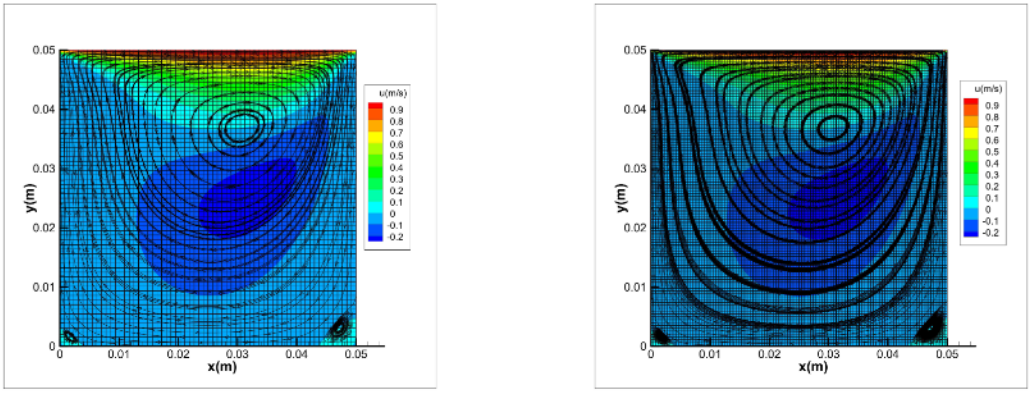
(d) Re = 1000

Figure 7: Effect of Reynolds Number on the streamlines

3. Effect of mesh size on streamlines

Figure 8 shows the effect of mesh size on the u-velocity streamlines for a Reynolds number of 100. The two grids used for this demonstration are 65x65 and 129x129. The figure shows

that a significant portion of streamlines loop back onto themselves in the 129x129 geometry when compared to the 65x65 grid.



(a) 65x65

(b) 129x129

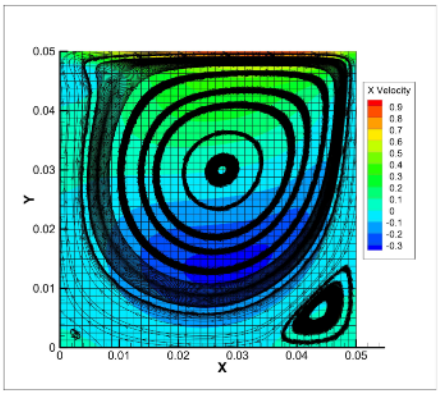
Figure 8: Effect of mesh refinement on the streamlines

V. VALIDATION

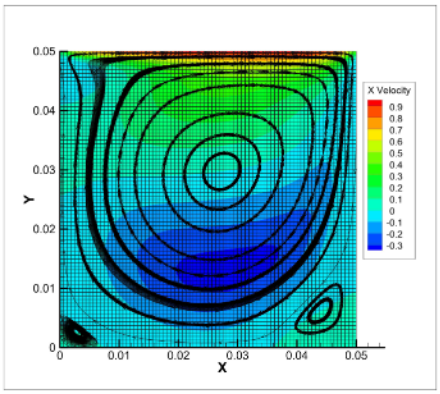
Simulations were run on ANSYS Fluent to do the solution validation of the results obtained from the code developed in MATLAB. Three different mesh sizes were used for simulation in Fluent which are namely:

- 41×41
- 81×81
- 161×161

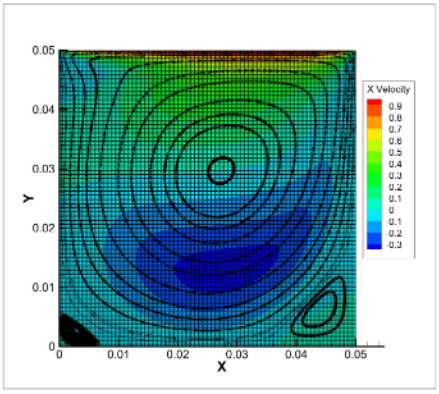
The boundary conditions as mentioned in sub-section B of section 2 were applied in Fluent and simulations were run with the SIMPLE algorithm for a case with Re = 100. The corresponding results obtained for the 3 grid sizes as mentioned above is depicted below:



(a) 41×41



(b) 81×81

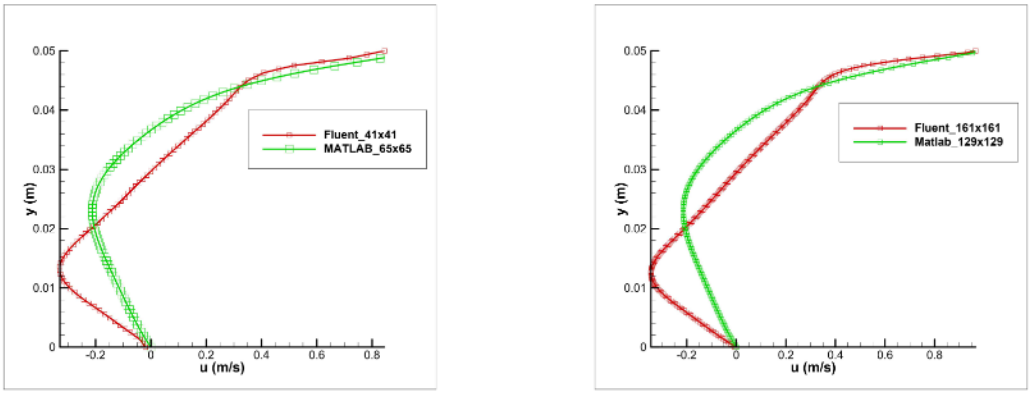


(c) 161×161

Figure 9: Contour plots of X-velocity for 3 different mesh sizes (ANSYS)

To get a clearer comparison, centerline velocities were extracted in the X-directions from

both ANSYS and MATLAB for 2 cases and the corresponding comparison is depicted below:



(a) (b)

Figure 10: Line graphs comparing centerline X velocities for 2 cases in ANSYS and MATLAB

The line graphs in ANSYS above is in close resemblance with that of MATLAB and hence it validates the code developed.

The results were also compared with those reported in literature. Due to computational limitations, running simulations for very high Reynolds numbers and finer meshes were difficult. Since most of the literature talks about high Reynold's number flows in fine meshes,the exact comparison of results could not be carried out. Nevertheless, the preliminary comparisons that we carried out has given promising results. On comparison with the streamline values for a Re=100 flow by Jignesh et al.[4] on a 128×128 grid, we found that the results we obtained for Re = 100 simulation on a 129×129 grid has a very close resemblance with it. Re = 100 and Re = 1000 cases were also compared with the research published by Tamer. A. AbdelMigid et al.[5] which validated the code even further in addition to the validation carried out with Fluent.

VI. CONCLUSION

Steady flow analysis of the 2D lid driven cavity shows the existence of secondary vortices at the corners of the cavity. Recirculation effects are also observed throughout the area.The overall center of the streamlines were observed to be shifting towards the centre of the cavity as Reynold's numbers were increased. The study had computational limitations and hence further research could not be done on finer meshes and in the future we would like to study this better by performing a mesh convergence study.

APPENDIX

A. Tortoise Git

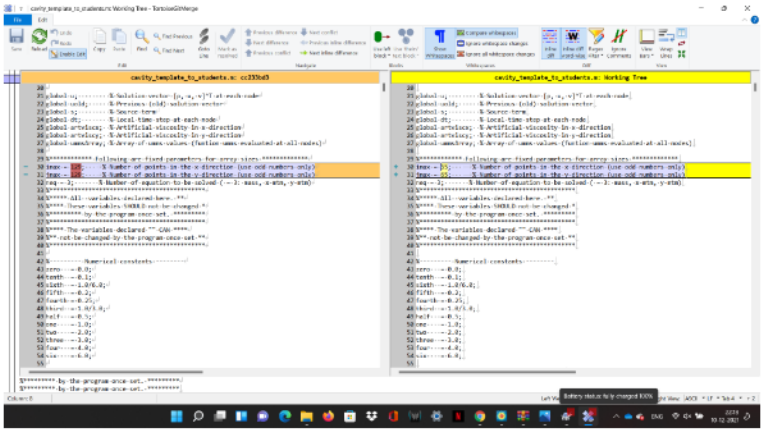
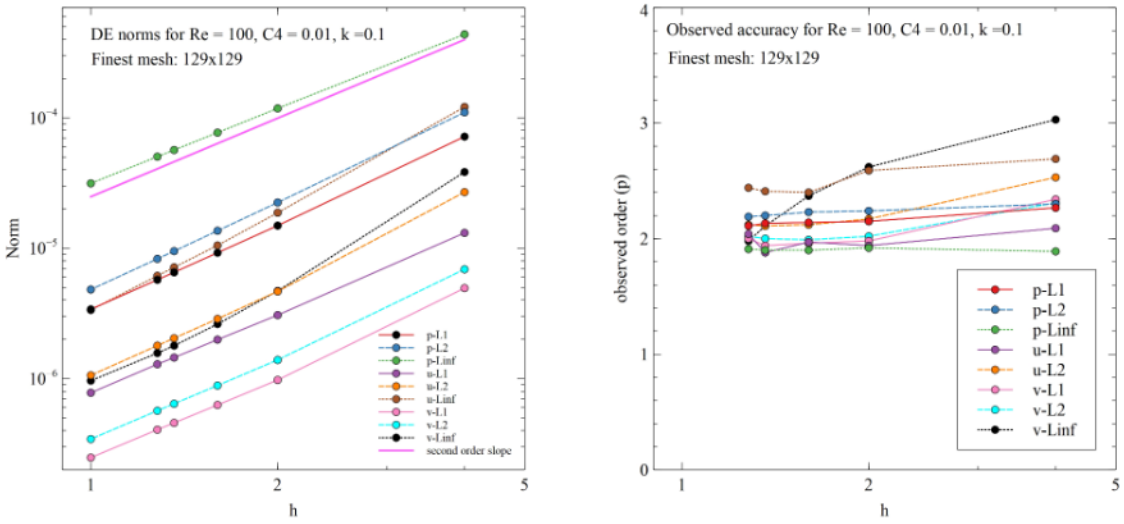


Figure 11: Diff in TortoiseGit

B. DE norms and order of accuracy for Re = 100

As the Reynolds number increased from 10 to 100, the decline in the observed order of accuracy of p-Linf has reduced.



(a) DE Norms as a function of grid spacing (h) for Re = 100

(b) Observed order of accuracy as a function of grid spacing (h) for Re = 100

Figure 12: DE norms and observed order of accuracy for Re = 100

C. Relating Iterative Residual and Iterative Error

A scaled relation between iterative residual and iterative error is shown in the following figure. For each iterative residual, the corresponding error in the pressure can be calculated by travelling down the constant iteration line till an intersection with the iterative error plot is reached. The plot shows iterative error for a 65x65 grid running at Re = 100 with CFL = 0.5.

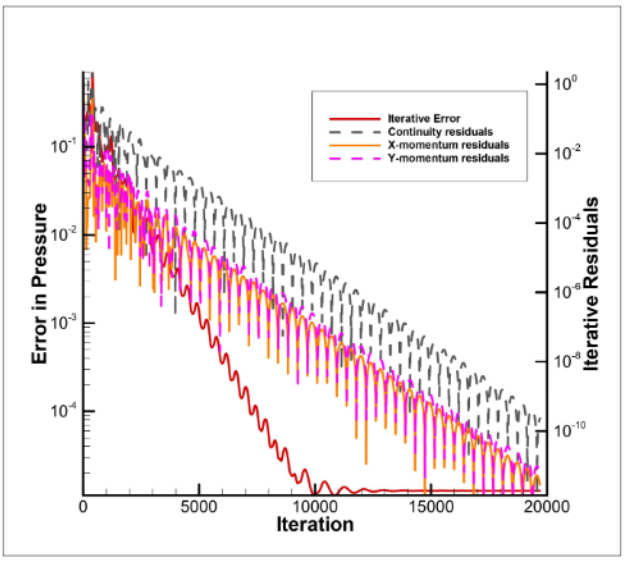


Figure 13: Scaled relationship between iterative residual and iterative error

References

[1] Kawaguti, M. *Numerical Solution of the Navier-Stokes Equations for the Flow in a Two-Dimensional Cavity*, Journal of the Physical Society of Japan, 1982.

[2] Erturk, E.,Corke, T.C.,Gokcol, C. *Numerical solutions of 2-D steady incompressible driven-cavity flow at high Reynolds numbers*, International Journal for Numerical Methods in Fluids, 2005.

[3] Marchi, C. H., Santiago, C. D., Carvalho, C. A. R., *Lid-Driven Square Cavity Flow:A Benchmark Solution With an 8192x8192 Grid*, Journal of Verification, Validation and Uncertainty Quantification, 2021.

[4] Jignesh P. Thaker, Jyotirmay Banerjee, *Numerical Simulation of Flow in Lid-driven Cavity using OpenFOAM* , INSTITUTE OF TECHNOLOGY, NIRMA UNIVERSITY, AHMEDABAD

[5] Tamer A. AbdelMigid, Khalid M. Saqr, Mohamed A. Kotb, Ahmed A. Aboelfarag, Revisiting the lid-driven cavity flow problem: Review and new steady state benchmarking results using GPU accelerated code, Alexandria Engineering Journal, Volume 56, Issue 1, 2017, Pages 123-135, ISSN 1110-0168,

```
function [PrsMatrix, uvelMatrix, vvelMatrix] = cavity_template_to_students(~)
tic   %begin timer function
%--- Variables for file handling ---
%--- All files are globally accessible ---

global fp1 % For output of iterative residual history
global fp2 % For output of field data (solution)
%   global fp3 % For writing the restart file
%   global fp4 % For reading the restart file
global fp5 % For output of final DE norms (only for MMS)
%$$$$$$   global fp6 % For debug: Uncomment for debugging.

global imax jmax neq nmax
global zero tenth sixth fifth fourth third half one two three four six
global iterout imms isgs irstr ipgorder lim cfl Cx Cy toler rkappa Re pinf uinf rho↙
rhoinv xmin xmax ymin ymax Cx2 Cy2 fsmall
global rlength rmu vel2ref dx dy rpi phi0 phix phiy phixy apx apy apxy fsinx fsiny↙
fsinxy

%**Use these variables cautiously as these are globally accessible from all↙
functions.**

global u;          % Solution vector [p, u, v]^T at each node
global uold;       % Previous (old) solution vector
global s;          % Source term
global dt;         % Local time step at each node
global artviscx;   % Artificial viscosity in x-direction
global artviscy;   % Artificial viscosity in y-direction
global ummsArray; % Array of umms values (funtion umms evaluated at all nodes)

%************* Following are fixed parameters for array sizes *************
imax = 129;      % Number of points in the x-direction (use odd numbers only)
jmax = 129;      % Number of points in the y-direction (use odd numbers only)
neq = 3;        % Number of equation to be solved ( = 3: mass, x-mtm, y-mtm)
%*****************************************
%***** All  variables declared here. **
%**** These variables SHOULD not be changed *
%********* by the program once set. *********
%*****************************************
%**** The variables declared "" CAN ****
%** not be changed by the program once set **
%*****************************************

%--------- Numerical constants --------
zero   = 0.0;
tenth  = 0.1;
sixth  = 1.0/6.0;
fifth  = 0.2;
fourth = 0.25;
third  = 1.0/3.0;
half   = 0.5;
one    = 1.0;
```

```
two    = 2.0;
three  = 3.0;
four   = 4.0;
six    = 6.0;

%--------- User sets inputs here  --------

nmax = 500000;         % Maximum number of iterations
iterout = 5000;        % Number of time steps between solution output
imms = 1;              % Manufactured solution flag: = 1 for manuf. sol., = 0 ↙
otherwise
isgs = 1;              % Symmetric Gauss-Seidel  flag: = 1 for SGS, = 0 for point ↙
Jacobi
irstr = 0;             % Restart flag: = 1 for restart (file 'restart.in', = 0 for ↙
initial run
ipgorder = 0;          % Order of pressure gradient: 0 = 2nd, 1 = 3rd (not needed)
lim = 1;               % variable to be used as the limiter sensor (= 1 for ↙
pressure)

cfl  = 0.1;       % CFL number used to determine time step
Cx = 0.01;       % Parameter for 4th order artificial viscosity in x
Cy = 0.01;           % Parameter for 4th order artificial viscosity in y
toler = 1.e-10;      % Tolerance for iterative residual convergence
rkappa = 0.1;        % Time derivative preconditioning constant
Re = 10.0;           % Reynolds number = rho*Uinf*L/rmu
pinf = 0.801333844662; % Initial pressure (N/m^2) -> from MMS value at cavity ↙
center
uinf = 1.0;       % Lid velocity (m/s)
rho = 1.0;        % Density (kg/m^3)
xmin = 0.0;       % Cavity dimensions...: minimum x location (m)
xmax = 0.05;     %                       maximum x location (m)
ymin = 0.0;       %                        maximum y location (m)
ymax = 0.05;     %                       maximum y location (m)
Cx2 = 0.0;        % Coefficient for 2nd order damping (not required)
Cy2 = 0.0;       % Coefficient for 2nd order damping (not required)
fsmall = 1.e-20; % small parameter

%-- Derived input quantities (set by function 'set_derived_inputs' called from ↙
main)----

rhoinv =  -99.9;     % Inverse density, 1/rho (m^3/kg)
rlength = -99.9;     % Characteristic length (m) [cavity width]
rmu = -99.9;         % Viscosity (N*s/m^2)
vel2ref = -99.9;     % Reference velocity squared (m^2/s^2)
dx = -99.9;          %    Delta x (m)
dy = -99.9;          % Delta y (m)
rpi = -99.9;         % Pi = 3.14159... (defined below)

%-- constants for manufactured solutions ----
phi0 = [0.25, 0.3, 0.2];          % MMS constant
phix = [0.5, 0.15, 1.0/6.0];      % MMS amplitude constant
phiy = [0.4, 0.2, 0.25];          % MMS amplitude constant
phixy = [1.0/3.0, 0.25, 0.1];     % MMS amplitude constant
```

```
apx = [0.5, 1.0/3.0, 7.0/17.0];     % MMS frequency constant
apy = [0.2, 0.25, 1.0/6.0];         % MMS frequency constant
apxy = [2.0/7.0, 0.4, 1.0/3.0];     % MMS frequency constant
fsinx = [0.0, 1.0, 0.0];            % MMS constant to determine sine vs. cosine
fsiny = [1.0, 0.0, 0.0];            % MMS constant to determine sine vs. cosine
fsinxy = [1.0, 1.0, 0.0];           % MMS constant to determine sine vs. cosine
% Note: fsin = 1 means the sine function
% Note: fsin = 0 means the cosine function
% Note: arrays here refer to the 3 variables

%**************************************************************************
%                            Main Function
%**************************************************************************
%----- Looping indices --------

i = 0;                          % i index (x direction)
j = 0;                          % j index (y direction)
k = 0;                          % k index (# of equations)
n = 0;                         % Iteration number index

conv = -99.9 ; % Minimum of iterative residual norms from three equations


%--------- Solution variables declaration --------

ninit = 0;          % Initial iteration number (used for restart file)

%$$$$$$    u(imax,jmax,neq);         % Solution vector (p, u, v)^T at each node
%$$$$$$    uold(imax,jmax,neq);      % Previous (old) solution vector
%$$$$$$    s(imax,jmax,neq);    % Source term
%$$$$$$    dt(imax,jmax);        % Local time step at each node
%$$$$$$    artviscx(imax,jmax);  % Artificial viscosity in x-direction
%$$$$$$    artviscy(imax,jmax);  % Artificial viscosity in y-direction
res = [0,0,0];              % Iterative residual for each equation
resinit = [0,0,0];          % Initial iterative residual for each equation (from
iteration 1)
rL1norm = [0,0,0];          % L1 norm of discretization error for each equation
rL2norm = [0,0,0];          % L2 norm of discretization error for each equation
rLinfnorm = [0,0,0];        % Linfinity norm of discretization error for each
equation
rtime = -99.9;         % Variable to estimate simulation time
dtmin = 1.0e99;        % Minimum time step for a given iteration (initialized
large)

x = -99.9;       % Temporary variable for x location
y = -99.9;       % Temporary variable for y location

% Solution variables initialization with dummy values
% for i=1:imax
%  for j=1:jmax
%      dt(i,j) = -99.9;
%      artviscx(i,j) = -99.9;
%      artviscy(i,j) = -99.9;
```

```
%    for k=1:neq
%      u(i,j,k) = -99.9;
%      uold(i,j,k) = -99.9;
%      s(i,j,k) = -99.9;
%      res(k) = -99.9;
%      resinit(k) = -99.9;
%      res(k) = -99.9;
%      rL1norm(k) = -99.9;
%      rL2norm(k) = -99.9;
%      rLinfnorm(k) = -99.9;
%    end
%  end
% end

dt = zeros(imax,jmax);
artviscx = zeros(imax,jmax);
artviscy = zeros(imax,jmax);
u = zeros(imax,jmax,neq);
uold = zeros(imax,jmax,neq);
ummsArray = zeros(imax,jmax,neq);
s = zeros(imax,jmax,neq);
res = zeros(neq,1);
resinit = zeros(neq,1);
rL1norm = zeros(neq,1);
rL2norm = zeros(neq,1);
rLinfnorm = zeros(neq,1);

dt(:,:) = -99.9;
artviscx(:,:) = -99.9;
artviscy(:,:) = -99.9;
u(:,:,:) = -99.9;
uold(:,:,:) = -99.9;
s(:,:,:) = -99.9;
res(:) = -99.9;
resinit(:) = -99.9;
rL1norm(:) = -99.9;
rL2norm(:) = -99.9;
rLinfnorm(:) = -99.9;


% Debug output: Uncomment and modify if debugging
%$$$$$$ fp6 = fopen("./Debug.dat","w");
%$$$$$$ fprintf(fp6,"TITLE = \"Debug Data Data\"\n");
%$$$$$$ fprintf(fp6,"variables=\"x(m)\"\"y(m)\"\"visc-x\"\"visc-y\"\n");
%$$$$$$ fprintf(fp6, "zone T=\"n=%d\"\n",n);
%$$$$$$ fprintf(fp6, "I= %d J= %d\n",imax, jmax);
%$$$$$$ fprintf(fp6, "DATAPACKING=POINT\n");

% Set derived input quantities
set_derived_inputs();

% Set up headers for output files
output_file_headers();
```

```
% Set Initial Profile for u vector
[ninit, rtime, resinit] = initial(ninit, rtime, resinit);

% Set Boundary Conditions for u
set_boundary_conditions();

% Write out inital conditions to solution file
write_output(ninit, resinit, rtime);

% Initialize Artificial Viscosity arrays to zero (note: artviscx(i,j) and artviscy↙
(i,j)
artviscx(:,:) = zero;
artviscy(:,:) = zero;

% Evaluate Source Terms Once at Beginning
%(only interior points; will be zero for standard cavity)
compute_source_terms();

%========== Main Loop ==========
isConverged = 0;

for n = ninit:nmax
    % Calculate time step
    dtmin = compute_time_step(dtmin);

    % Save u values at time level n (u and uold are 2D arrays)
    uold = u;

    if isgs==1 % ==Symmetric Gauss Seidel==

        % Artificial Viscosity
        Compute_Artificial_Viscosity();

        % Symmetric Gauss-Siedel: Forward Sweep
        SGS_forward_sweep();

        % Set Boundary Conditions for u
        set_boundary_conditions();

        % Artificial Viscosity
        Compute_Artificial_Viscosity();

        % Symmetric Gauss-Siedel: Backward Sweep
        SGS_backward_sweep();

        % Set Boundary Conditions for u
        set_boundary_conditions();
    else
        if isgs==0 % ==Point Jacobi==

            % Artificial Viscosity
            Compute_Artificial_Viscosity();
```

```
            % Point Jacobi: Forward Sweep
            point_Jacobi();

            % Set Boundary Conditions for u
            set_boundary_conditions();
        else
            fprintf('ERROR: isgs must equal 0 or 1!\n');
            return;
        end
    end

    % Pressure Rescaling (based on center point)
    pressure_rescaling();

    % Update the time
    rtime = rtime + dtmin;

    % Check iterative convergence using L2 norms of iterative residuals
    [res, resinit, conv] = check_iterative_convergence(n, res, resinit, ninit, ↙
rtime, dtmin);

    if(conv<toler)
        fprintf(fp1, '%d %e %e %e\n',n, res(1), res(2), res(3));
        isConverged = 1;
        break;
    end

    % Output solution and restart file every 'iterout' steps
    if( (mod(n,iterout)==0) )
        write_output(n, resinit, rtime);
    end

end  % ========== End Main Loop ==========

if isConverged == 0
    fprintf('Solution failed to converge in %d iterations!!!', nmax);
end

if isConverged == 1
    fprintf('Solution converged in %d iterations!!!', n);
end

% Calculate and Write Out Discretization Error Norms (will do this for MMS only)
Discretization_Error_Norms(rL1norm, rL2norm, rLinfnorm);

% Output solution and restart file
write_output(n, resinit, rtime);

% Close open files
fclose(fp1);
fclose(fp2);
fclose(fp5);
```

```
%$$$$$$   fclose(fp6); % Uncomment for debug output (

PrsMatrix = u(:,:,1);    %output arrays
uvelMatrix = u(:,:,2);
vvelMatrix = u(:,:,3);
%contour(PrsMatrix)
%x = linspace(0,0.05,jmax);
%y = linspace(0,0.05,jmax);
%[X,Y] = meshgrid(x,y);
%contourf(X,Y,uvelMatrix,10)
%colorbar('AxisLocation','in')
toc  %end timer function
end

%*********************************************************************/
%*                          All Other   Functions                    */
%*********************************************************************/

%*********************************************************************
%*********************************************************************
function set_derived_inputs(~)
global imax jmax
global one
global Re uinf rho rhoinv xmin xmax ymin ymax
global rlength rmu vel2ref dx dy rpi

rhoinv = one/rho;                          % Inverse density, 1/rho (m^3/kg) */
rlength = xmax - xmin;                     % Characteristic length (m) [cavity↙
width] */
rmu = rho*uinf*rlength/Re;                 % Viscosity (N*s/m^2) */
vel2ref = uinf*uinf;                       % Reference velocity squared (m^2/s^2)↙
*/
dx = (xmax - xmin)/(imax - 1);          % Delta x (m) */
dy = (ymax - ymin)/(jmax - 1);          % Delta y (m) */
rpi = acos(-one);                          % Pi = 3.14159... */
fprintf('rho,V,L,mu,Re: %f %f %f %f %f\n',rho,uinf,rlength,rmu,Re);
end

%*********************************************************************
function output_file_headers(~)

% Uses global variable(s): imms, fp1, fp2
% Note: The vector of primitive variables is:
%               u = [p, u, v]^T
% Set up output files (history and solution)

global imms fp1 fp2 fp5
fp1 = fopen('./history.dat','w');
fprintf(fp1,'TITLE = "Cavity Iterative Residual History"\n');
fprintf(fp1,'variables="Iteration""Res1""Res2""Res3"\n');

fp2 = fopen('./cavity.dat','w');
fprintf(fp2,'TITLE = "Cavity Field Data"\n');
```

```
if (imms==1)

    fprintf(fp2,'variables="x(m)""y(m)""p(N/m^2)""u(m/s)""v(m/s)"');
    fprintf(fp2,'"p-exact""u-exact""v-exact""DE-p""DE-u""DE-v"\n');

else

    if (imms==0)

        fprintf(fp2,'variables="x(m)""y(m)""p(N/m^2)""u(m/s)""v(m/s)"\n');

    else

        fprintf('ERROR! imms must equal 0 or 1!!!\n');
        return;
    end
end
fp5 = fopen('./DEnorms.dat','w');
fprintf(fp5,'Title = "Discretization Error Norms"\n');
fprintf(fp5,'variables="p-L1""u-L1""v-L1""p-L2""u-L2""v-L2""p-Linf""u-Linf""v-Linf"\n');

% Header for Screen Output
fprintf('Iter. Time (s)   dt (s)      Continuity    x-Momentum    y-Momentum\n');

end
%**********************************************************************
function [ninit, rtime, resinit] = initial(ninit, rtime, resinit)
%
%Uses global variable(s): zero, one, irstr, imax, jmax, neq, uinf, pinf
%To modify: ninit, rtime, resinit, u, s

% 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

% This subroutine sets inital conditions in the cavity
% Note: The vector of primitive variables is:
%              u = (p, u, v)^T

global zero one irstr imax jmax neq uinf pinf xmax xmin ymax ymin
global u s ummsArray

if (irstr==0)   % Starting run from scratch
    ninit = 1;          % set initial iteration to one
    rtime = zero;       % set initial time to zero
    for k = 1:neq
        resinit(k) = one;
    end
    for j = 1:jmax
        for i = 1:imax
```

```
            u(i,j,1) = pinf;
            u(i,j,2) = zero;
            u(i,j,3) = zero;
            s(i,j,1) = zero;
            s(i,j,2) = zero;
            s(i,j,3) = zero;
        end
        u(i,jmax,2) = uinf; % Initialize lid (top) to freestream velocity
    end
else
    if (irstr==1)  % Restarting from previous run (file 'restart.in')
        fp4 = fopen('./restart.in','r'); % Note: 'restart.in' must exist!
        if (fp4==NULL)
            fprintf('Error opening restart file. Stopping.\n');
            return;
        end
        fscanf(fp4, '%d %lf', ninit, rtime); % Need to known current iteration #↙
and time value
        fscanf(fp4, '%lf %lf %lf', resinit(0), resinit(1), resinit(2)); % Needs↙
initial iterative residuals for scaling
        for j=1:jmax
            for i=1:imax
                fscanf(fp4, '%lf %lf %lf %lf %lf', x, y, u(i,j,1), u(i,j,2), u(i,j,↙
3));
            end
        end
        ninit = ninit + 1;
        fprintf('Restarting at iteration %d\n', ninit);
        fclose(fp4);
    else
        printf('ERROR: irstr must equal 0 or 1!\n');
        return;
    end
end

%initialize the ummsArray with values computed with umms function
for j=1:jmax
    for i=1:imax
        for k=1:neq
            x = (xmax - xmin)*(i-1)/(imax - 1);
            y = (ymax - ymin)*(j-1)/(jmax - 1);
            ummsArray(i,j,k) = umms(x,y,k);
        end
    end
end
end
%**************************************************************************
function set_boundary_conditions(~)
%
%Uses global variable(s): imms
%To modify: u (via other functions: bndry() and bndrymms())

global imms
```

```
% This subroutine determines the appropriate BC routines to call
if (imms==0)
    bndry();
else
    if (imms==1)
        bndrymms();
    else
        printf('ERROR: imms must equal 0 or 1!\n');
        return;
    end
end
end
%*********************************************************************
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 imax jmax uinf
global u

% This applies the cavity boundary conditions


% !************************************************************ */
% !************ADD CODING HERE FOR INTRO CFD STUDENTS************ */
% !************************************************************ */

u(:,1,2) = zero; % x-velocity is zero at the bottom wall
u(:,1,3) = zero; %y-velocity is zero at the bottom wall
u(1,:,2) = zero; %x-velocity is zero at the left wall
u(1,:,3) = zero; %y-velocity is zero at the left wall
u(imax,:,2) = zero; %x-velocity is zero at the right wall
u(imax,:,3) = zero; %y-velocity is zero at the right wall
u(:,jmax,2) = uinf; %x-velocity is equal to the lid velocity at top wall
u(:,jmax,3) = zero; %y-velocity is zero at the top wall
u(:,1,1) = (two.*u(:,2,1))- u(:,3,1); %pressure at bottom wall
u(imax,:,1) = (two.*u(imax-1,:,1))-u(imax-2,:,1); %pressure at the right wall
u(1,:,1) = (two.*u(2,:,1))-u(3,:,1); %pressure at the left wall
u(:,jmax,1) = (two.*u(:,jmax-1,1))-u(:,jmax-2,1); % pressure at the top wall
end
%*********************************************************************
function bndrymms(~)
%
%Uses global variable(s): two, imax, jmax, neq, xmax, xmin, ymax, ymin, rlength
%To modify: u
% 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
% This applies the cavity boundary conditions for the manufactured solution

global two imax jmax neq
global u ummsArray

% Side Walls
for j = 2:jmax-1
    i = 1;
    for k = 1:neq
        u(i,j,k) = ummsArray(i,j,k);
    end
    u(1,j,1) = two*u(2,j,1) - u(3,j,1);     % 2nd Order BC
    %    u(1,j,1) = u(2,j,1);                  % 1st Order BC

    i=imax;
    for k = 1:neq
        u(i,j,k) = ummsArray(i,j,k);
    end
    u(imax,j,1) = two*u(imax-1,j,1) - u(imax-2,j,1);    % 2nd Order BC
    %   u(imax,j,1) = u(imax-1,j,1);                      % 1st Order BC
end

% Top/Bottom Walls
for i=1:imax
    j = 1;
    for k = 1:neq
        u(i,j,k) = ummsArray(i,j,k);
    end
    u(i,1,1) = two*u(i,2,1) - u(i,3,1);    % 2nd Order BC
    %$$$$$$     u(i,1,1) = u(i,2,1);            % 1st Order BC

    j = jmax;
    for k = 1:neq
        u(i,j,k) = ummsArray(i,j,k);
    end
    u(i,jmax,1) = two*u(i,jmax-1,1) - u(i,jmax-2,1);    % 2nd Order BC
    %$$$$$$     u(i,jmax,1) = u(i,jmax-1,1);              % 1st Order BC
end
end
%**********************************************************************
function [ummstmp] = umms( x, y, k)
%
%Uses global variable(s): one, rpi, rlength
%Inputs: x, y, k
%To modify: <none>
%Returns: umms

% ummstmp; % Define return value for umms as % precision

% termx       % Temp variable
% termy       % Temp variable
```

```
% termxy      % Temp variable
% argx        % Temp variable
% argy        % Temp variable
% argxy       % Temp variable

% This function returns the MMS exact solution

global one rpi rlength
global phi0 phix phiy phixy apx apy apxy fsinx fsiny fsinxy

argx = apx(k)*rpi*x/rlength;
argy = apy(k)*rpi*y/rlength;
argxy = apxy(k)*rpi*x*y/rlength/rlength;
termx = phix(k)*(fsinx(k)*sin(argx)+(one-fsinx(k))*cos(argx));
termy = phiy(k)*(fsiny(k)*sin(argy)+(one-fsiny(k))*cos(argy));
termxy = phixy(k)*(fsinxy(k)*sin(argxy)+(one-fsinxy(k))*cos(argxy));

ummstmp = phi0(k) + termx + termy + termxy;
end
%*********************************************************************
function write_output( n,  resinit,  rtime)
%
%Uses global variable(s): imax, jmax, new, xmax, xmin, ymax, ymin, rlength, imms
%Uses global variable(s): ninit, u, dt, resinit, rtime
%To modify: <none>
%Writes output and restart files.

% 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

global imax jmax xmax xmin ymax ymin imms
global u ummsArray
global fp2 fp3

% Field output
fprintf(fp2, 'zone T="n=%d"\n',n);
fprintf(fp2, 'I= %d J= %d\n',imax, jmax);
fprintf(fp2, 'DATAPACKING=POINT\n');

if (imms==1)
    for j=1:jmax
        for i=1:imax
            x = (xmax - xmin)*(i-1)/(imax - 1);
            y = (ymax - ymin)*(j-1)/(jmax - 1);
            fprintf(fp2,'%e %e %e %e %e %e %e %e %e %e %e\n', x, y, ...
                u(i,j,1), u(i,j,2), u(i,j,3), ummsArray(i,j,1), ummsArray(i,j,2), ↙
ummsArray(i,j,3), ...
                (u(i,j,1)-ummsArray(i,j,1)), (u(i,j,2)-ummsArray(i,j,2)), (u(i,j,3) ↙
-ummsArray(i,j,3)));
```

```
        end
    end
else
    if (imms==0)
        for j=1:jmax
            for i=1:imax
                x = (xmax - xmin)*(i-1)/(imax - 1);
                y = (ymax - ymin)*(j-1)/(jmax - 1);
                fprintf(fp2,'%e %e %e %e %e\n', x, y, ...
                    u(i,j,1), u(i,j,2), u(i,j,3));
            end
        end
    else
        fprintf('ERROR: imms must equal 0 or 1!\n');
        return;
    end
end

% Restart file: overwrites every 'iterout' iteration
fp3 = fopen('./restart.out','w');
fprintf(fp3,'%d %e\n', n, rtime);
fprintf(fp3,'%e %e %e\n', resinit(1), resinit(2), resinit(3));
for j=1:jmax
    for i=1:imax
        x = (xmax - xmin)*(i-1)/(imax - 1);
        y = (ymax - ymin)*(j-1)/(jmax - 1);
        fprintf(fp3,'%e %e %e %e %e\n', x, y, ...
            u(i,j,1), u(i,j,2), u(i,j,3));
    end
end
fclose(fp3);
end
%**********************************************************************
function compute_source_terms(~)
%
%Uses global variable(s): imax, jmax, imms, rlength, xmax, xmin, ymax, ymin
%To modify: s (source terms)

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

% x        % Temporary variable for x location
% y        % Temporary variable for y location

% Evaluate Source Terms Once at Beginning (only %erior po%s; will be zero for↙
standard cavity)

global imax jmax imms xmax xmin ymax ymin
global s

for j=2:jmax-1
    for i=2:imax-1
        x = (xmax - xmin)*(i-1)/(imax - 1);
```

```
        y = (ymax - ymin)*(j-1)/(jmax - 1);
        s(i,j,1) = (imms)*srcmms_mass(x,y);
        s(i,j,2) = (imms)*srcmms_xmtm(x,y);
        s(i,j,3) = (imms)*srcmms_ymtm(x,y);
    end
end
end
%**********************************************************************
function [srcmasstmp] = srcmms_mass( x, y)
%
%Uses global variable(s): rho, rpi, rlength
%Inputs: x, y
%To modify: <none>
%Returns: srcmms_mass
% srcmasstmp; % Define return value for srcmms_mass as % precision

% dudx;     % Temp variable: u velocity gradient in x direction
% dvdy;  % Temp variable: v velocity gradient in y direction

% This function returns the MMS mass source term

global rho rpi rlength
global phix phiy phixy apx apy apxy

dudx = phix(2)*apx(2)*rpi/rlength*cos(apx(2)*rpi*x/rlength)   ...
    + phixy(2)*apxy(2)*rpi*y/rlength/rlength  ...
    * cos(apxy(2)*rpi*x*y/rlength/rlength);

dvdy = -phiy(3)*apy(3)*rpi/rlength*sin(apy(3)*rpi*y/rlength)   ...
    - phixy(3)*apxy(3)*rpi*x/rlength/rlength  ...
    * sin(apxy(3)*rpi*x*y/rlength/rlength);

srcmasstmp = rho*dudx + rho*dvdy;
end
%**********************************************************************
function [srcxmtmtmp] = srcmms_xmtm( x,  y)
%
%Uses global variable(s): rho, rpi, rmu, rlength
%Inputs: x, y
%To modify: <none>
%Returns: srcmms_xmtm

% srcxmtmtmp; % Define return value for srcmms_xmtm as % precision

% dudx;     % Temp variable: u velocity gradient in x direction
% dudy;  % Temp variable: u velocity gradient in y direction
% termx;        % Temp variable
% termy;        % Temp variable
% termxy;       % Temp variable
% uvel;         % Temp variable: u velocity
% vvel;         % Temp variable: v velocity
% dpdx;         % Temp variable: pressure gradient in x direction
% d2udx2;       % Temp variable: 2nd derivative of u velocity in x direction
```

```
% d2udy2;       % Temp variable: 2nd derivative of u velocity in y direction

%This function returns the MMS x-momentum source term

global rho rpi rmu rlength
global phi0 phix phiy phixy apx apy apxy

termx = phix(2)*sin(apx(2)*rpi*x/rlength);
termy = phiy(2)*cos(apy(2)*rpi*y/rlength);
termxy = phixy(2)*sin(apxy(2)*rpi*x*y/rlength/rlength);
uvel = phi0(2) + termx + termy + termxy;

termx = phix(3)*cos(apx(3)*rpi*x/rlength);
termy = phiy(3)*cos(apy(3)*rpi*y/rlength);
termxy = phixy(3)*cos(apxy(3)*rpi*x*y/rlength/rlength);
vvel = phi0(3) + termx + termy + termxy;

dudx = phix(2)*apx(2)*rpi/rlength*cos(apx(2)*rpi*x/rlength) ...
    + phixy(2)*apxy(2)*rpi*y/rlength/rlength  ...
    * cos(apxy(2)*rpi*x*y/rlength/rlength);

dudy = -phiy(2)*apy(2)*rpi/rlength*sin(apy(2)*rpi*y/rlength)  ...
    + phixy(2)*apxy(2)*rpi*x/rlength/rlength  ...
    * cos(apxy(2)*rpi*x*y/rlength/rlength);

dpdx = -phix(1)*apx(1)*rpi/rlength*sin(apx(1)*rpi*x/rlength) ...
    + phixy(1)*apxy(1)*rpi*y/rlength/rlength  ...
    * cos(apxy(1)*rpi*x*y/rlength/rlength);

d2udx2 = -phix(2)*((apx(2)*rpi/rlength).^2)  ...
    * sin(apx(2)*rpi*x/rlength)  ...
    - phixy(2)*((apxy(2)*rpi*y/rlength/rlength).^2)  ...
    * sin(apxy(2)*rpi*x*y/rlength/rlength);

d2udy2 = -phiy(2)*((apy(2)*rpi/rlength).^2)  ...
    * cos(apy(2)*rpi*y/rlength)  ...
    - phixy(2)*((apxy(2)*rpi*x/rlength/rlength).^2)  ...
    * sin(apxy(2)*rpi*x*y/rlength/rlength);

srcxmtmtmp = rho*uvel*dudx + rho*vvel*dudy + dpdx  ...
    - rmu*( d2udx2 + d2udy2 );

end
%**********************************************************************
function [srcymtmtmp] = srcmms_ymtm( x, y)
%
%Uses global variable(s): rho, rpi, rmu, rlength
%Inputs: x, y
%To modify: <none>
%Returns: srcmms_ymtm

% srcymtmtmp; % Define return value for srcmms_ymtm as % precision
```

```
% dvdx;          % Temp variable: v velocity gradient in x direction
% dvdy;          % Temp variable: v velocity gradient in y direction
% termx;         % Temp variable
% termy;         % Temp variable
% termxy;        % Temp variable
% uvel;          % Temp variable: u velocity
% vvel;          % Temp variable: v velocity
% dpdy;          % Temp variable: pressure gradient in y direction
% d2vdx2;        % Temp variable: 2nd derivative of v velocity in x direction
% d2vdy2;        % Temp variable: 2nd derivative of v velocity in y direction

% This function returns the MMS y-momentum source term

global rho rpi rmu rlength
global phi0 phix phiy phixy apx apy apxy

termx = phix(2)*sin(apx(2)*rpi*x/rlength);
termy = phiy(2)*cos(apy(2)*rpi*y/rlength);
termxy = phixy(2)*sin(apxy(2)*rpi*x*y/rlength/rlength);
uvel = phi0(2) + termx + termy + termxy;

termx = phix(3)*cos(apx(3)*rpi*x/rlength);
termy = phiy(3)*cos(apy(3)*rpi*y/rlength);
termxy = phixy(3)*cos(apxy(3)*rpi*x*y/rlength/rlength);
vvel = phi0(3) + termx + termy + termxy;

dvdx = -phix(3)*apx(3)*rpi/rlength*sin(apx(3)*rpi*x/rlength)   ...
    - phixy(3)*apxy(3)*rpi*y/rlength/rlength   ...
    * sin(apxy(3)*rpi*x*y/rlength/rlength);

dvdy = -phiy(3)*apy(3)*rpi/rlength*sin(apy(3)*rpi*y/rlength)   ...
    - phixy(3)*apxy(3)*rpi*x/rlength/rlength   ...
    * sin(apxy(3)*rpi*x*y/rlength/rlength);

dpdy = phiy(1)*apy(1)*rpi/rlength*cos(apy(1)*rpi*y/rlength)   ...
    + phixy(1)*apxy(1)*rpi*x/rlength/rlength   ...
    * cos(apxy(1)*rpi*x*y/rlength/rlength);

d2vdx2 = -phix(3)*((apx(3)*rpi/rlength).^2)   ...
    * cos(apx(3)*rpi*x/rlength)   ...
    - phixy(3)*((apxy(3)*rpi*y/rlength/rlength).^2)   ...
    * cos(apxy(3)*rpi*x*y/rlength/rlength);

d2vdy2 = -phiy(3)*((apy(3)*rpi/rlength).^2)   ...
    * cos(apy(3)*rpi*y/rlength)   ...
    - phixy(3)*((apxy(3)*rpi*x/rlength/rlength).^2)   ...
    * cos(apxy(3)*rpi*x*y/rlength/rlength);

srcymtmtmp = rho*uvel*dvdx + rho*vvel*dvdy + dpdy   ...
    - rmu*( d2vdx2 + d2vdy2 );

end
%**********************************************************************
```

```
function [dt] = 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)

global four half fourth
global vel2ref rmu rho dx dy cfl rkappa
global imax jmax
global u dt


% !************************************************************* */
% !************ADD CODING HERE FOR INTRO CFD STUDENTS************ */
% !************************************************************* */
dtvisc = (fourth*dx*dy*rho)/rmu;       % Viscous time step stability criteria↙
(constant over domain)
uvel2 = zeros(imax-2,jmax-2);
beta2 = zeros(imax-2,jmax-2);
lambda_x = zeros(imax-2,jmax-2);
lambda_y = zeros(imax-2,jmax-2);
lambda_max = zeros(imax-2,jmax-2);
dtconv = zeros(imax-2,jmax-2);
%dtmin = zeros(imax-2,jmax-2);
for j = 2:jmax-1
    for i = 2:imax-1
        uvel2(i,j) = (u(i,j,2)).^2+(u(i,j,3)).^2;      % Local velocity squared
        beta2(i,j) = max(uvel2(i,j),rkappa.*vel2ref);       % Beta squared↙
parameter for time derivative preconditioning
        lambda_x(i,j) = half*(abs(u(i,j,2))+ sqrt((u(i,j,2).^2)+(four.*beta2(i,↙
j))));   % Max absolute value eigenvalue in (x,t)
        lambda_y(i,j) = half*(abs(u(i,j,3))+ sqrt((u(i,j,3).^2)+(four.*beta2(i,↙
j))));     % Max absolute value eigenvalue in (y,t)
        lambda_max(i,j) = max(lambda_x(i,j),lambda_y(i,j));  % Max absolute value↙
eigenvalue (used in convective time step computation)
        dtconv(i,j) = min(dx,dy)./lambda_max(i,j);       % Local convective time↙
step restriction
        dt(i,j) = cfl.*min(dtvisc,dtconv(i,j));
        %dtmin(i,j) = dt(i,j);
    end
end
dtmin = min(dtmin,min(min(dt)));
%dtmin = dt; % local time stepping
end
%*********************************************************************
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 [these are
% not used]

global two four six half
global imax jmax lim dx dy Cx Cy fsmall vel2ref rkappa
%global rho %not sure why this was used
global u
global artviscx artviscy


% !************************************************************* */
% !************ADD CODING HERE FOR INTRO CFD STUDENTS************ */
% !************************************************************* */

uvel2 = zeros(imax-2,jmax-2);
beta2 = zeros(imax-2,jmax-2);
lambda_x = zeros(imax-2,jmax-2);
lambda_y = zeros(imax-2,jmax-2);

for j = 2:jmax-1
    for i = 2:imax-1
        uvel2(i,j) = (u(i,j,2)).^2+(u(i,j,3)).^2;       % Local velocity squared
        beta2(i,j) = max(uvel2(i,j),rkappa.*vel2ref);        % Beta squared↙
parameter for time derivative preconditioning
        lambda_x(i,j) = half*(abs(u(i,j,2))+ sqrt((u(i,j,2).^2)+(four.*beta2(i,↙
j))));   % Max absolute value eigenvalue in (x,t)
        lambda_y(i,j) = half*(abs(u(i,j,3))+ sqrt((u(i,j,3).^2)+(four.*beta2(i,↙
j))));      % Max absolute value eigenvalue in (y,t)
    end
end

d4px4 = zeros(imax-2,jmax-2);
d4py4 = zeros(imax-2,jmax-2);
for j = two:jmax-1
    for i = two:imax-1
        if i == 2
```

```
            d4px4(i,j) = ((14*u(i+3,j,1))+(2*u(i-1,j,1))+(24*u(i+1,j,1))-(26*u(i+2,j,1))-(3*u(i+4,j,1))-(11*u(i,j,1))).*(4*(dx^4));
        elseif i == imax-1
            d4px4(i,j) = (((14*u(i-3,j,1))+(2*u(i+1,j,1))+(24*u(i-1,j,1))-(26*u(i-2,j,1))-(3*u(i-4,j,1))-(11*u(i,j,1)))).*(4*(dx^4));
        else
            d4px4(i,j) = (u(i+2,j,1)-(4*(u(i+1,j,1)+u(i-1,j,1)))+(six*u(i,j,1))+u(i-2,j,1)).*(dx^4);
        end
    end
end
for j = 2:imax-1
    for i = 2:jmax-1
        if i == 2
            d4py4(j,i) = ((14*u(j,i+3,1))+(2*u(j,i-1,1))+(24*u(j,i+1,1))-(26*u(j,i+2,1))-(3*u(j,i+4,1))-(11*u(j,i,1))).*(4*(dy^4));
        elseif i == jmax-1
            d4py4(j,i) = (((14*u(j,i-3,1))+(2*u(j,i+1,1))+(24*u(j,i-1,1))-(26*u(j,i-2,1))-(3*u(j,i-4,1))-(11*u(j,i,1))).*(4*(dy^4)));
        else
            d4py4(j,i) = (u(j,i+2,1)-(4*(u(j,i+1,1)+u(j,i-1,1)))+(six*u(j,i,1))+u(j,i-2,1)).*(dy^4);
        end
    end
end
for j = 2:jmax-1
    for i = 2:imax-1
        artviscx(i,j) = -((lambda_x(i,j).*Cx.*(dx^3))./beta2(i,j)).*d4px4(i,j).*lim;
        artviscy(i,j) = -((lambda_y(i,j).*Cy.*(dy^3))./beta2(i,j)).*d4py4(i,j).*lim;
    end
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

% Symmetric Gauss-Siedel: Forward Sweep

% !************************************************************* */
% !************ADD CODING HERE FOR INTRO CFD STUDENTS************ */
% !************************************************************* */
dpdx = zeros(imax-2,jmax-2);
dpdy = zeros(imax-2,jmax-2);
dudx = zeros(imax-2,jmax-2);
dudy = zeros(imax-2,jmax-2);
dvdx = zeros(imax-2,jmax-2);
dvdy = zeros(imax-2,jmax-2);
d2udx2 = zeros(imax-2,jmax-2);
d2vdx2 = zeros(imax-2,jmax-2);
d2udy2 = zeros(imax-2,jmax-2);
d2vdy2 = zeros(imax-2,jmax-2);
uvel2 = zeros(imax-2,jmax-2);       % Local velocity squared
beta2 = zeros(imax-2,jmax-2);
for j = 2:jmax-1
    for i = 2:imax-1
        uvel2(i,j) = (u(i,j,2)).^2+(u(i,j,3)).^2;       % Local velocity squared
        beta2(i,j) = max(uvel2(i,j),rkappa.*vel2ref);       % Beta squared↙
parameter for time derivative preconditioning
    end
end
for j = two:jmax-1
    for i = two:imax-1
         dpdx(i,j) = half*(u(i+1,j,1)-u(i-1,j,1))./(dx);
        dudx(i,j) = (u(i+1,j,2)-u(i-1,j,2))./(2*dx);
        dvdx(i,j) = (u(i+1,j,3)-u(i-1,j,3))./(2*dx);
        d2udx2(i,j) = (u(i+1,j,2)-(2*u(i,j,2))+u(i-1,j,2))./(dx^2);
        d2vdx2(i,j) = (u(i+1,j,3)-(2*u(i,j,3))+u(i-1,j,3))./(dx^2);
    end
end
for j = 2:jmax-1
    for i = 2:imax-1
        dpdy(i,j) = (u(i,j+1,1)-u(i,j-1,1))./(2*dy);
        dudy(i,j) = (u(i,j+1,2)-u(i,j-1,2))./(2*dy);
        dvdy(i,j) = (u(i,j+1,3)-u(i,j-1,3))./(2*dy);
        d2udy2(i,j) = (u(i,j+1,2)-(2*u(i,j,2))+u(i,j-1,2))./(dy^2);
        d2vdy2(i,j) = (u(i,j+1,3)-(2*u(i,j,3))+u(i,j-1,3))./(dy^2);
    end
```

```
end

for j = 2:jmax-1
    for i = 2:imax-1
        u(i,j,1) = u(i,j,1) - (beta2(i,j).*dt(i,j)).*((rho.*dudx(i,j))+(rho.*dvdy
(i,j))-s(i,j,1)-artviscx(i,j)-artviscy(i,j));
    end
end
for j = 2:jmax-1
    for i = 2:imax-1
         u(i,j,2) = u(i,j,2) - (dt(i,j)*rhoinv).*((rho*u(i,j,2)).*(dudx(i,j))+
(rho*u(i,j,3)).*(dudy(i,j))+dpdx(i,j)-(rmu*d2udx2(i,j))-(rmu*d2udy2(i,j))-s(i,j,
2));
    end
end
for j = 2:jmax-1
    for i = 2:imax-1
        u(i,j,3) = u(i,j,3) - (dt(i,j)*rhoinv).*((rho*u(i,j,2)).*(dvdx(i,j))+(rho*u
(i,j,3)).*(dvdy(i,j))+dpdy(i,j)-(rmu*d2vdx2(i,j))-(rmu*d2vdy2(i,j))-s(i,j,3));
    end
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

% Symmetric Gauss-Siedel: Backward Sweep

% !************************************************************ */
% !************ADD CODING HERE FOR INTRO CFD STUDENTS************ */
% !************************************************************ */
dpdx = zeros(imax-2,jmax-2);
dpdy = zeros(imax-2,jmax-2);
dudx = zeros(imax-2,jmax-2);
dudy = zeros(imax-2,jmax-2);
dvdx = zeros(imax-2,jmax-2);
dvdy = zeros(imax-2,jmax-2);
d2udx2 = zeros(imax-2,jmax-2);
d2vdx2 = zeros(imax-2,jmax-2);
d2udy2 = zeros(imax-2,jmax-2);
d2vdy2 = zeros(imax-2,jmax-2);
uvel2 = zeros(imax-2,jmax-2);      % Local velocity squared
beta2 = zeros(imax-2,jmax-2);
for j = 2:jmax-1
    for i = 2:imax-1
        uvel2(i,j) = (u(i,j,2)).^2+(u(i,j,3)).^2;      % Local velocity squared
        beta2(i,j) = max(uvel2(i,j),rkappa.*vel2ref);       % Beta squared↙
parameter for time derivative preconditioning
    end
end
for j = two:jmax-1
    for i = two:imax-1
         dpdx(i,j) = half*(u(i+1,j,1)-u(i-1,j,1))./(dx);
        dudx(i,j) = (u(i+1,j,2)-u(i-1,j,2))./(2*dx);
        dvdx(i,j) = (u(i+1,j,3)-u(i-1,j,3))./(2*dx);
        d2udx2(i,j) = (u(i+1,j,2)-(2*u(i,j,2))+u(i-1,j,2))./(dx^2);
        d2vdx2(i,j) = (u(i+1,j,3)-(2*u(i,j,3))+u(i-1,j,3))./(dx^2);
    end
end
for j = 2:jmax-1
    for i = 2:imax-1
        dpdy(i,j) = (u(i,j+1,1)-u(i,j-1,1))./(2*dy);
        dudy(i,j) = (u(i,j+1,2)-u(i,j-1,2))./(2*dy);
        dvdy(i,j) = (u(i,j+1,3)-u(i,j-1,3))./(2*dy);
        d2udy2(i,j) = (u(i,j+1,2)-(2*u(i,j,2))+u(i,j-1,2))./(dy^2);
        d2vdy2(i,j) = (u(i,j+1,3)-(2*u(i,j,3))+u(i,j-1,3))./(dy^2);
    end
end
for j = jmax-1:2
    for i = imax-1:2
        u(i,j,1) = u(i,j,1) - (beta2(i,j).*dt(i,j)).*((rho.*dudx(i,j))+(rho.*dvdy↙
(i,j))-s(i,j,1)-artviscx(i,j)-artviscy(i,j));
    end
end
for j = jmax-1:2
    for i = imax-1:2
         u(i,j,2) = u(i,j,2) - (dt(i,j)*rhoinv).*((rho*u(i,j,2)).*(dudx(i,j))+↙
```

```
(rho*u(i,j,3)).*(dudy(i,j))+dpdx(i,j)-(rmu*d2udx2(i,j))-(rmu*d2udy2(i,j))-s(i,j,↙
2));
    end
end
for j = jmax-1:2
    for i = imax-1:2
        u(i,j,3) = u(i,j,3) - (dt(i,j)*rhoinv).*((rho*u(i,j,2)).*(dvdx(i,j))+(rho*u↙
(i,j,3)).*(dvdy(i,j))+dpdy(i,j)-(rmu*d2vdx2(i,j))-(rmu*d2vdy2(i,j))-s(i,j,3));
    end
end




end
%**********************************************************************
function point_Jacobi(~)
%
%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: 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 artviscx artviscy dt s
global uold %not sure why this was used.

% Point Jacobi method

% !************************************************************ */
% !************ADD CODING HERE FOR INTRO CFD STUDENTS************ */
% !************************************************************ */
dpdx = zeros(imax-2,jmax-2);
dpdy = zeros(imax-2,jmax-2);
```

```
dudx = zeros(imax-2,jmax-2);
dudy = zeros(imax-2,jmax-2);
dvdx = zeros(imax-2,jmax-2);
dvdy = zeros(imax-2,jmax-2);
d2udx2 = zeros(imax-2,jmax-2);
d2vdx2 = zeros(imax-2,jmax-2);
d2udy2 = zeros(imax-2,jmax-2);
d2vdy2 = zeros(imax-2,jmax-2);
for i = 2:imax-1
    for j = 2:jmax-1
        dpdx(i,j) = (uold(i+1,j,1)-uold(i-1,j,1))./(2*dx);
        dudx(i,j) = (uold(i+1,j,2)-uold(i-1,j,2))./(2*dx);
        dvdx(i,j) = (uold(i+1,j,3)-uold(i-1,j,3))./(2*dx);
        d2udx2(i,j) = (uold(i+1,j,2)-(2*uold(i,j,2))+uold(i-1,j,2))./(dx^2);
        d2vdx2(i,j) = (uold(i+1,j,3)-(2*uold(i,j,3))+uold(i-1,j,3))./(dx^2);
    end
end
for j = 2:jmax-1
    for i = 2:imax-1
        dpdy(i,j) = (uold(i,j+1,1)-uold(i,j-1,1))./(2*dy);
        dudy(i,j) = (uold(i,j+1,2)-uold(i,j-1,2))./(2*dy);
        dvdy(i,j) = (uold(i,j+1,3)-uold(i,j-1,3))./(2*dy);
        d2udy2(i,j) = (uold(i,j+1,2)-(2*uold(i,j,2))+uold(i,j-1,2))./(dy^2);
        d2vdy2(i,j) = (uold(i,j+1,3)-(2*uold(i,j,3))+uold(i,j-1,3))./(dy^2);
    end
end
uvel2 = zeros(imax-2,jmax-2);      % Local velocity squared
beta2 = zeros(imax-2,jmax-2);
for j = 2:jmax-1
    for i = 2:imax-1
        uvel2(i,j) = (uold(i,j,2)).^2+(uold(i,j,3)).^2;      % Local velocity
squared
        beta2(i,j) = max(uvel2(i,j),rkappa.*vel2ref);       % Beta squared
parameter for time derivative preconditioning
    end
end
%pnew = zeros(imax,jmax);
%unew = zeros(imax,jmax);
%vnew = zeros(imax,jmax);
for j = 2:jmax-1
    for i = 2:imax-1
        u(i,j,1) = uold(i,j,1) - (beta2(i,j).*dt(i,j)).*((rho.*dudx(i,j))+(rho.
*dvdy(i,j))-s(i,j,1)-artviscx(i,j)-artviscy(i,j));
    end
end
for j = 2:jmax-1
    for i = 2:imax-1
         u(i,j,2) = uold(i,j,2) - (dt(i,j)*rhoinv).*((rho*uold(i,j,2)).*(dudx(i,j))
+(rho*uold(i,j,3)).*(dudy(i,j))+dpdx(i,j)-(rmu*d2udx2(i,j))-(rmu*d2udy2(i,j))-s(i,
j,2));
    end
end
for j = 2:jmax-1
```

```
    for i = 2:imax-1
        u(i,j,3) = uold(i,j,3) - (dt(i,j)*rhoinv).*((rho*uold(i,j,2)).*(dvdx(i,j))+↙
(rho*uold(i,j,3)).*(dvdy(i,j))+dpdy(i,j)-(rmu*d2vdx2(i,j))-(rmu*d2vdy2(i,j))-s(i,j,↙
3));
    end
end

end
%**********************************************************************
function pressure_rescaling(~)
%
%Uses global variable(s): imax, jmax, imms, xmax, xmin, ymax, ymin, rlength, pinf
%To Modify: u

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

% iref                       % i index location of pressure rescaling point
% jref                       % j index location of pressure rescaling point

% x        % Temporary variable for x location
% y        % Temporary variable for y location
% deltap   % delta_pressure for rescaling all values

global imax jmax imms xmax xmin ymax ymin pinf
global u

iref = (imax-1)/2+1;      % Set reference pressure to center of cavity
jref = (jmax-1)/2+1;
if (imms==1)
    x = (xmax - xmin)*(iref-1)/(imax - 1);
    y = (ymax - ymin)*(jref-1)/(jmax - 1);
    deltap = u(iref,jref,1) - umms(x,y,1); % Constant in MMS
else
    deltap = u(iref,jref,1) - pinf; % Reference pressure
end

j=1:jmax;
i=1:imax;
u(i,j,1) = u(i,j,1) - deltap;


end
%**********************************************************************
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
global u uold fp1
global dt
%global fsmall

% Compute iterative residuals to monitor iterative convergence

% !************************************************************ */
% !************ADD CODING HERE FOR INTRO CFD STUDENTS************ */
% !************************************************************ */
rsq = zeros(neq,1); % defining a variable for the square of local residuals
L2norm = zeros(neq,1); % defining a variable for L2 norm
for k = 1:neq
    for j = 2:jmax-1
        for i = 2:imax-1
            res(k) = abs(u(i,j,k)-uold(i,j,k))./dt(i,j);
            if n<=6
                resinit(k) = res(k);
            end
            res(k) = res(k)./resinit(k);
            rsq(k) = rsq(k) + (res(k)).^2;
        end
    end
    L2norm(k) = (rsq(k)/(imax*jmax)).^0.5;
end
%conv = min(L2norm);
conv = max(L2norm);


% Write iterative residuals every 10 iterations
if ( (mod(n,10)==0)||(n==ninit) )
    fprintf(fp1, '%d %e %e %e\n',n, res(1), res(2), res(3) );
    fprintf('%d  %e   %e   %e\n',n, res(1), res(2), res(3) );
    % Maybe a need to format this better
end



% Write header for iterative residuals every 200 iterations
if ( (mod(n,200)==0)||(n==ninit) )
    fprintf('Iter.     Continuity    x-Momentum    y-Momentum\n');
end

end

%*********************************************************************
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 fp5

if imms==1

% !************************************************************** */
% !************ADD CODING HERE FOR INTRO CFD STUDENTS************ */
% !************************************************************** */
x = zeros(imax,jmax);
y = zeros(imax,jmax);
r1 = zeros(neq,1);
r2 = zeros(neq,1);
r3 = zeros(imax,jmax);
for k = 1:neq
    for j = 1:jmax
        for i = 1:imax
            x(i,j) = (xmax - xmin)*(i-1)/(imax - 1);
            y(i,j) = (ymax - ymin)*(j-1)/(jmax - 1);
            r1(k) = r1(k) + abs(u(i,j,k)-ummsArray(i,j,k));
            r2(k) = r2(k) + (abs(u(i,j,k)-ummsArray(i,j,k)).^2);
            r3(i,j) = abs(u(i,j,k)-ummsArray(i,j,k));
        end
    end
    rL1norm(k) = r1(k)./(imax*jmax);
    rL2norm(k) = sqrt((r2(k)./(imax*jmax)));
    rLinfnorm(k) = max(max(r3));
end

end
fprintf(fp5, '%e %e %e %e %e %e %e %e %e\n',rL1norm(1),rL1norm(2),rL1norm(3),↙
rL2norm(1),rL2norm(2),rL2norm(3),rLinfnorm(1),rLinfnorm(2),rLinfnorm(3));
end
```