MAE-561 Computational Fluid Dynamics

Jose Parra

I. Abstract

In this computational study, we examine fluid flow and scalar transport in a two-dimensional mixing chamber, driven by rotating disks using finite difference methods such as the 2D viscous Burger's equation. The simulation utilizes a comprehensive set of governing equations, including continuity, momentum, and mass fraction equations, to capture the behavior of flow using Navier Stokes. Boundary conditions for multiple inlets with predefined velocity and scalar profiles, along with no-slip wall conditions and a zero-gradient outlet. The numerical approach is validated through Grid Convergence Index (GCI) analysis across several mesh refinements, ensuring the accuracy and stability of the simulation under a Courant-Friedrichs-Lewy (CFL) condition of 0.9, Reynolds number of 100, and Sc number of 2.

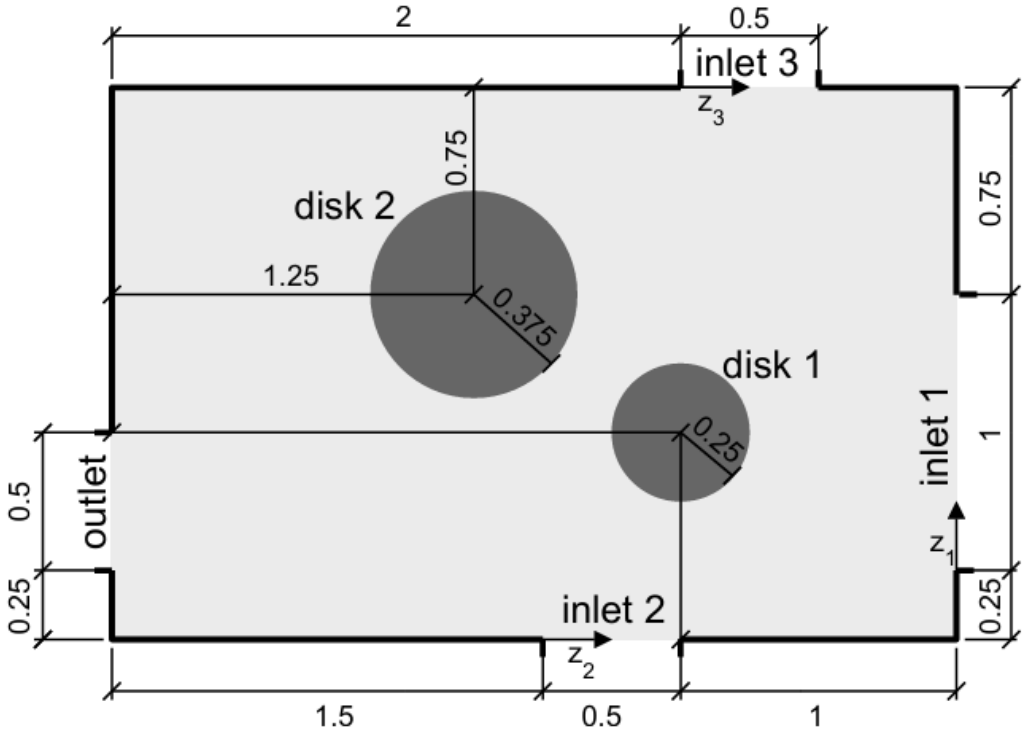

Fig. 1 Mixing Chamber Geometry

II. Governing Equations

Continuity equation

$$\frac{\partial u}{\partial t}+\frac{\partial v}{\partial y}=0$$

x-momentum equation

$$\frac{\partial u}{\partial t}+\frac{\partial uu}{\partial x}+\frac{\partial uv}{\partial y}=-\frac{1}{\rho}\frac{\partial p}{dx}+v\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)$$

y-momentum equation

$$\frac{\partial v}{\partial t}+\frac{\partial uv}{\partial x}+\frac{\partial vv}{\partial y}=-\frac{1}{\rho}\frac{\partial p}{dy}+v\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right)$$

mass fraction equation.

$$\frac{\partial Y}{\partial t}+\frac{\partial uY}{\partial x}+\frac{\partial vY}{\partial y}=-\frac{1}{\rho}\frac{\partial p}{dy}+v\left(\frac{\partial^2 Y}{\partial x^2}+\frac{\partial^2 Y}{\partial y^2}\right)$$

inlet boundary conditions

$$u_{in,1}(z_1)=U_1\cos(\alpha_1)\,f(z_1),\quad v_{in,1}(z_1)=U_1(\alpha_1)f(z_1),\quad Y_{in,1}=1$$

$$u_{in,2}(z_2)=U_2\cos(\alpha_2)\,f(z_2),\quad v_{in,2}(z_2)=U_2(\alpha_2)f(z_2),\quad Y_{in,2}=0$$

$$u_{in,3}(z_3)=U_3\cos(\alpha_3)\,f(z_3),\quad v_{in,3}(z_3)=U_3(\alpha_3)f(z_3),\quad Y_{in,3}=0$$

$$U_1=\frac{3}{4},U_2=2,U_3=3,\alpha_1=\pi,\alpha_2=\frac{\pi}{3},\alpha_3=\frac{-\pi}{6}$$

outlet boundary conditions

While z_i is a non-dimensional coordinate along the inlet as indicated in Fig. 1, i.e., $z_i=0$ at one edge of the inlet and $z_i=1$ at the other edge, e.g., $z_1=(y-0.25)/1$, $z_2=\frac{x-1.5}{0.5}$

$$f(z)=6z(1-z)$$

wall boundary conditions.

$$outlet:\quad \frac{\partial u}{\partial x}|_{out}=0,\qquad \frac{\partial v}{dx}|_{out}=0,\qquad \frac{\partial Y}{\partial x}|_{out}=0$$

k(t) equation

$$k(t)=\int_0^{L_y}\int_0^{L_x}\frac{1}{2}(\hat{u}^2+\hat{v}^2)\,dxdy$$

S(t) equation

$$S(t)=\int_0^{L_y}\int_0^{L_x}\frac{1}{2}(\hat{u}^2+\hat{v}^2)\,dxdy$$

Kbar equation

$$\overline{K} = \frac{1}{10}\int_0^{10} k(t)dt$$

R equation

$$\bar{R} = \frac{1}{10}\int_0^{10} S(t)dt$$

III. Numerical Section

Table 1. Numerical Analysis

$$H^n_{v_{i,j+\frac{1}{2}}} =$$

$$\frac{-\frac{1}{2}\left(u^n_{i+\frac{1}{2},j} + u^n_{i+\frac{1}{2},j+1}\right) * \frac{1}{2}\left(v^n_{i,j+\frac{1}{2}} + v^n_{i+1,j+\frac{1}{2}}\right) - \frac{1}{2}\left(u^n_{i-\frac{1}{2},j} + u^n_{i-\frac{1}{2},j+1}\right) * \frac{1}{2}\left(v^n_{i,j+\frac{1}{2}} + v^n_{i-1,j+\frac{1}{2}}\right)}{h}$$

$$- \frac{\left(\frac{1}{2}\left(v^n_{i,j+\frac{3}{2}} + v^n_{i,j+\frac{1}{2}}\right)\right)^2 - \left(\frac{1}{2}\left(v^n_{i,j+\frac{1}{2}} + v^n_{i,j-\frac{1}{2}}\right)\right)^2}{h}$$

$$H^n_{u_{i+\frac{1}{2},j}} =$$

$$\frac{\frac{1}{2}\left(u^n_{i+\frac{3}{2},j}+u^n_{i+\frac{1}{2},j+1}\right)*\frac{1}{2}\left(v^n_{i,j+\frac{1}{2}}+v^n_{i+1,j+\frac{1}{2}}\right)-\frac{1}{2}\left(u^n_{i-\frac{1}{2},j}+u^n_{i-\frac{1}{2},j+1}\right)*\frac{1}{2}\left(v^n_{i,j+\frac{1}{2}}+v^n_{i-1,j+\frac{1}{2}}\right)}{h}$$
$$-\frac{\left(\frac{1}{2}\left(v^n_{i,j+\frac{3}{2}}+v^n_{i,j+\frac{1}{2}}\right)\right)^2-\left(\frac{1}{2}\left(v^n_{i,j+\frac{1}{2}}+v^n_{i,j-\frac{1}{2}}\right)\right)^2}{h}$$

IV. Simulation Parameters

Parameters	
M	384
N	256
CFL	0.9
Poisson Criteria	10^{-4}

V. Solution Verification

	M	N	h	Kbar	p	f_h	GCI21	GCI32	% check
1	48	32	0.0625	9.6846					
2	96	64	0.03125	9.8544					
3	192	128	0.015625	9.9049	1.749481	9.926277	0.269775	0.911735	0.994902
4	384	256	0.007813	9.9326	0.866397	9.966253	0.423518	0.774277	0.997211

Table 2. $\bar{K}$ Solution Verification

Final answer fh=9.966253 +- 0.42518%

	M	N	h	Rbar	p	fh	GCI21	GCI32	% check
1	48	32	0.0625	0.0531					
2	96	64	0.03125	0.0546					
3	192	128	0.015625	0.0579	-1.1375	0.05185	-13.0613	-6.29579	0.943005
4	384	256	0.007813	0.0587	2.044394	0.058956	0.545145	2.279793	0.986371

Table 3. $\bar{R}$ Solution Verification

Final answer fh=0.058956 +- 0.545145%

VI. Results Figure

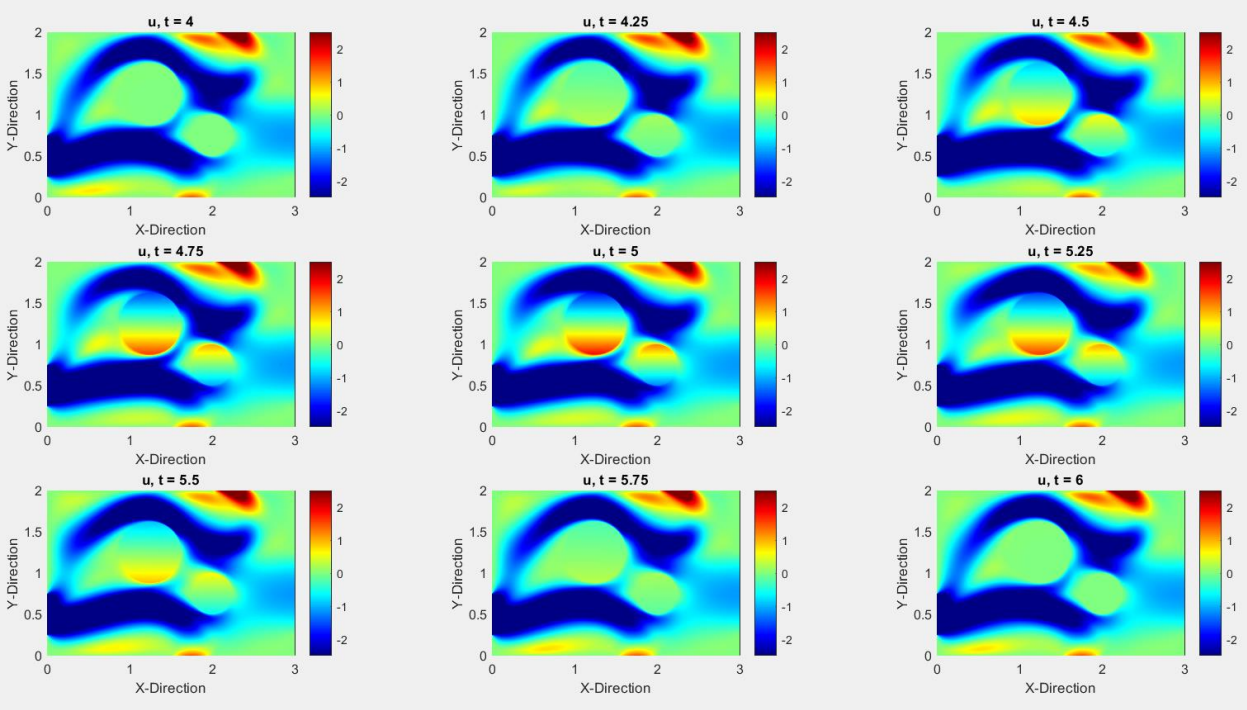

Figure 2. U Velocity Profile Resolution N=256 M=384

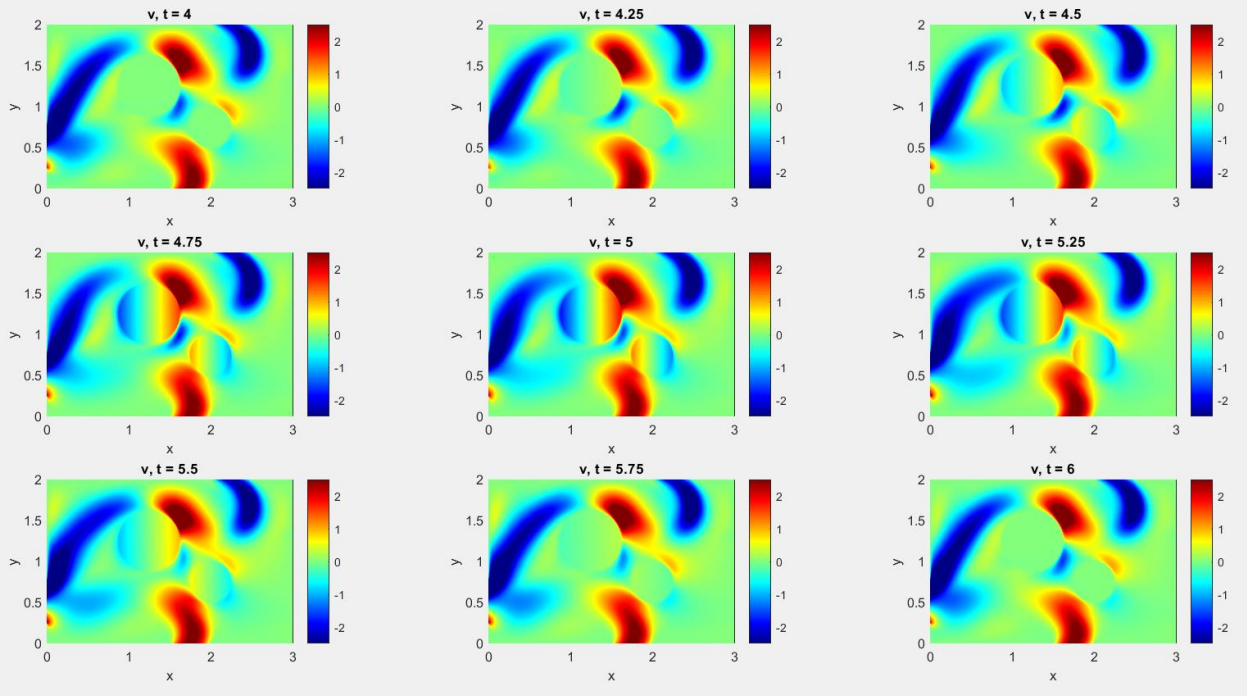

Figure 3. V Velocity Profile Resolution N=256 M=384

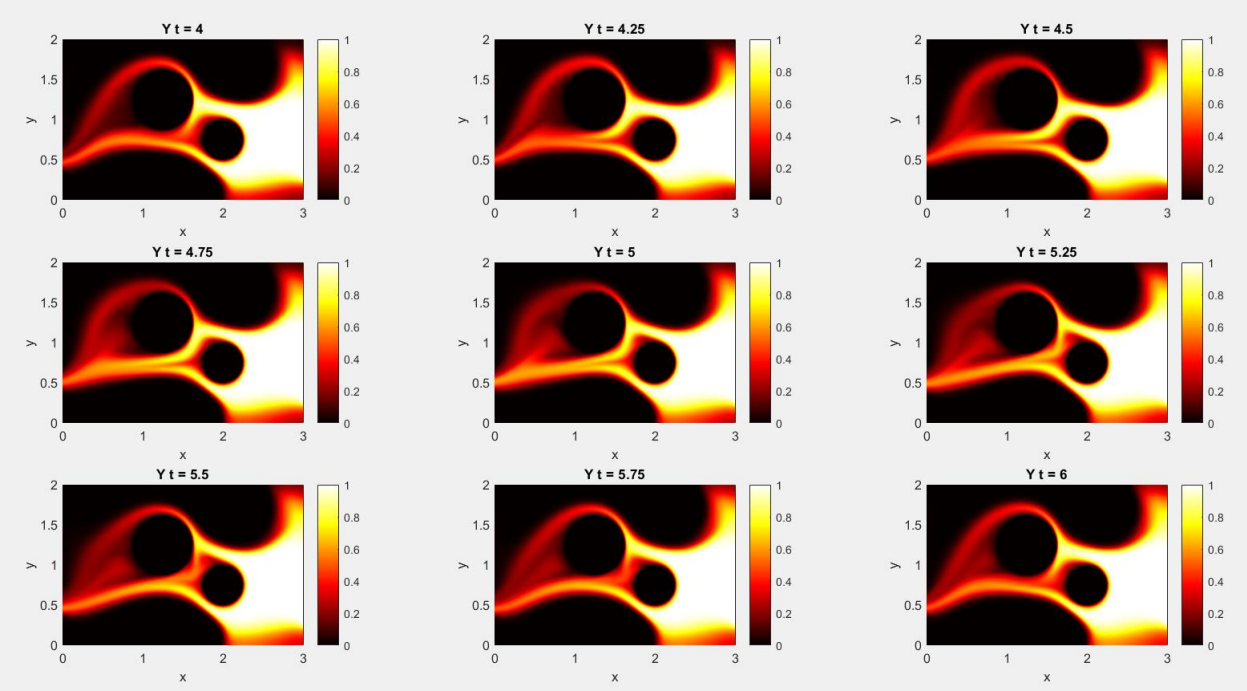

Figure 4. Mass Fraction Profile Resolution N=256 M=384

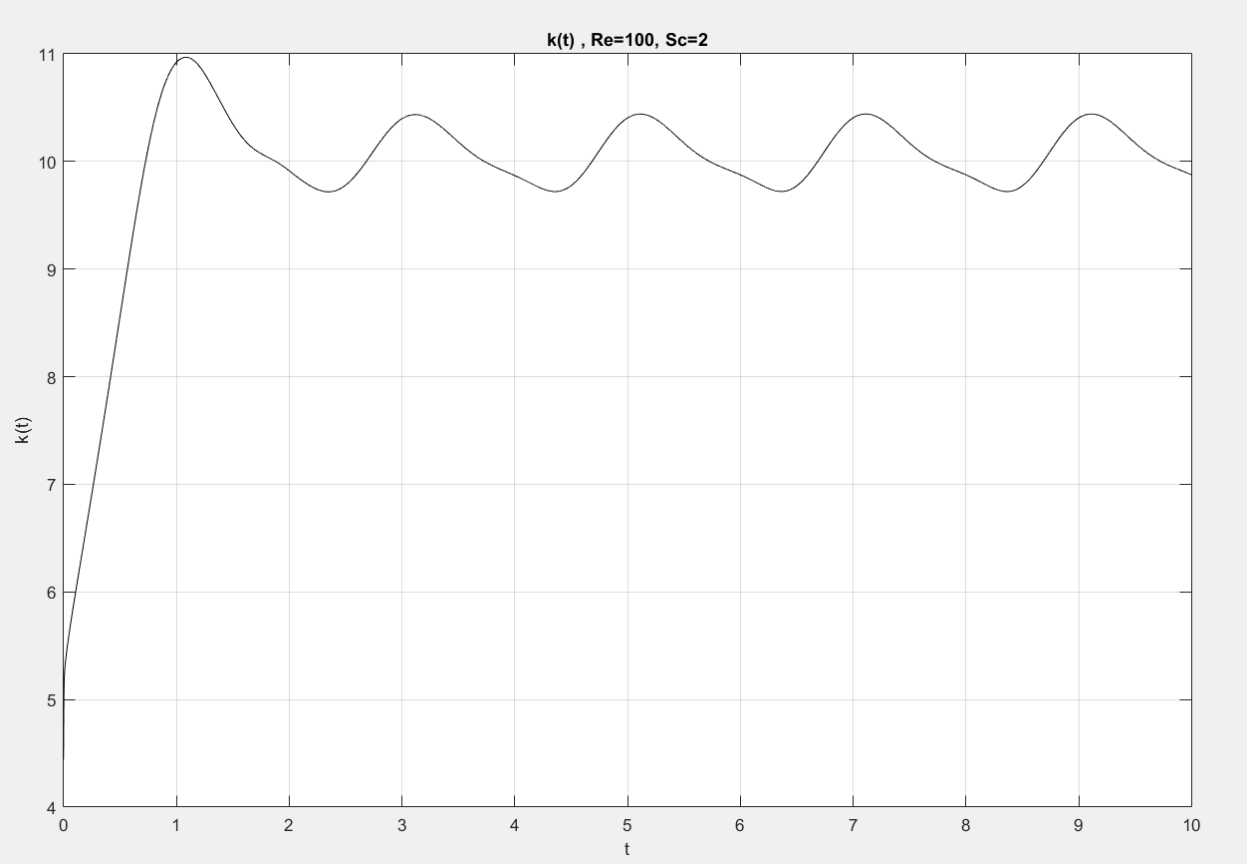

Figure 5. Kinetic Energy Resolution N=256 M=384

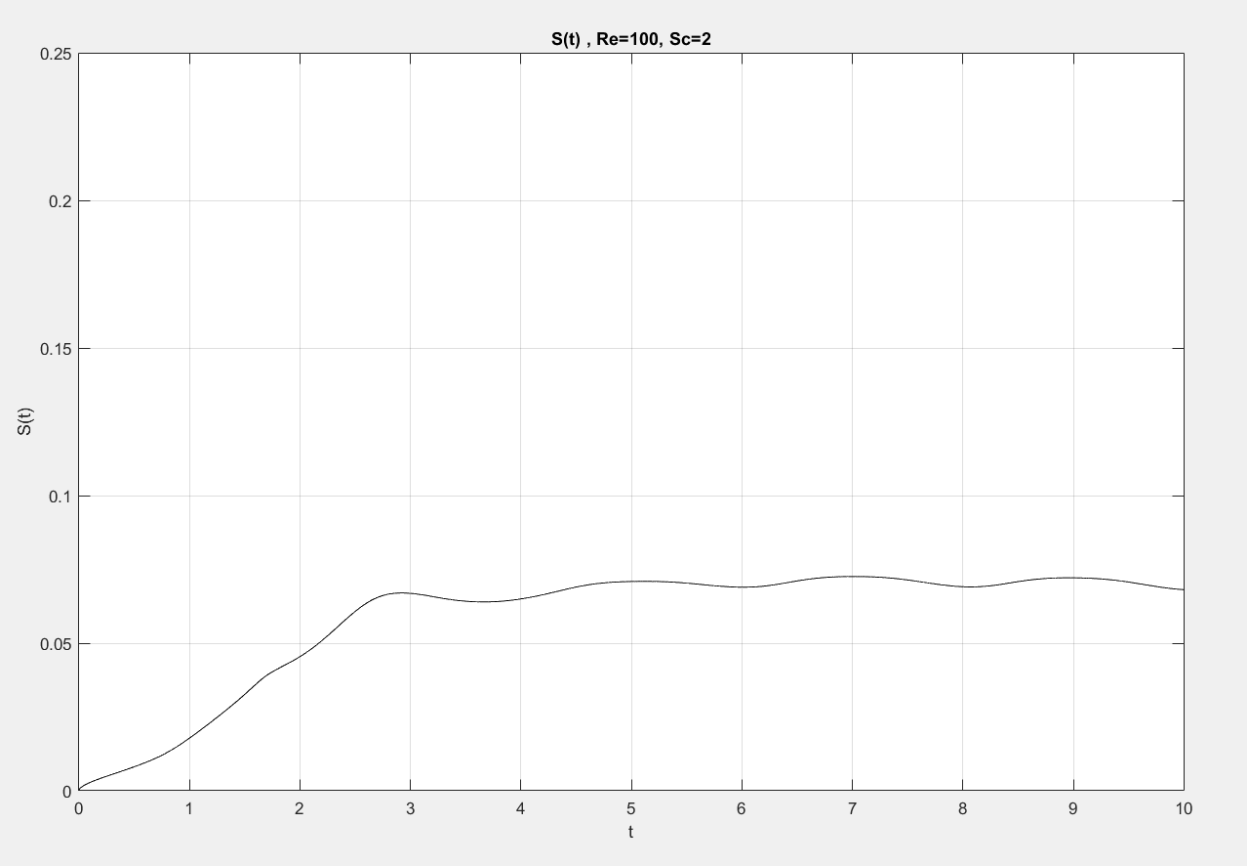

Figure 6. Scalar Mixture Resolution N=256 M=384

VII. Results Discussion

Velocity Component u

The first series of contour plots represent the horizontal velocity component u. We observe the formation and movement of vortices, which suggest complex flow behavior that develops with time. The U is affected by the rotating circle as shown by figure 1 we can see that the circle top and bottom portion change velocities by the change of their color shown by the color bar. Blue being moving left and red being moving right. We can see the cylinders having an affect on the flow when oscillating.

Velocity Component v

The first series of contour plots represent the Vertical velocity component v. We observe the formation and movement of vortices, which suggest complex flow

behavior that develops with time. The v is affected by the rotating circle as shown by figure 1 we can see that the circle top and bottom portion change velocities by the change of their color shown by the color bar. Blue being moving downwards and red being moving upwards. We can see the cylinders having an affect on the flow when oscillating.
vertical component displays regions of concentrated activity which align with the horizontal component's vortical structures.

Mass Fraction Y

The mass fraction contour plots indicate the distribution of a particular scalar within the fluid, potentially representing the concentration of a dissolved substance or temperature anomalies. The time started at 4 to let the mass fill in and go into some sort of steady state. The lighter regions being where the mass is concentrated.

Kinetic Energy k(t)

The kinetic energy plot as a function of time shows a non-linear relationship, indicating a dynamic flow regime. The periodic peaks in kinetic energy may correspond to the shedding of vortices or the pulsing of flow structures within the domain as well they can also by do to the hyperbolic equations used for the numerical methods. The kinetic energy oscillates around 10. The rise of kinetic energy is due to the mass coming into the system and the oscillation are due to the circles adding and taking away kinetic energy.

Scalar Mixing parameter s(t)

The Scaler mixing parameter plot as a function of time shows a non-linear relationship, indicating a dynamic flow regime. The periodic peaks in kinetic energy may correspond to the shedding of vortices or the pulsing of flow structures within the domain to show how well the mixing is done at a certain time around 3 seconds, the mixture starts to oscillate around 0.08

Mixing chambers

The calsourceIBfinal helped mixing and

Solution Verification result

The simulation was ran several times, each time using a grid with a finer mesh size. By running GCI ($\bar{K}$ and $\bar{R}$) stabilize as the grid gets more detailed and with the bigger and finer mesh. Based on our the GCI32 we can see the mesh converging. The check shows the accuracy to see how close we are getting to the true solution and as shown we get closer.

Findings

The data from our simulations show that as we used more refined meshes, the values of $\bar{K}$ and $\bar{R}$ began to change less and less with each step of refinement. This is a good sign—it suggests that our results are 'converging,' which in non-expert terms means that our simulation is becoming more accurate and is less affected by how finely we divide the grid.

Movie Discussion

The movie uploaded was at a lower resolution of N=80 due to time constraints but the N size at which the simulation converges would be a mesh size of N=256 as shown in the GCI analysis

VIII. Conclusion

In this report, we modeled a two-dimensional mixing chamber with rotating disks, applying finite difference methods to solve the continuity, momentum, and mass fraction equations. The simulations were conducted with specific boundary conditions and verified through Grid Convergence Index (GCI) for various mesh refinements. The results, obtained under a CFL condition of 0.9, a Reynolds number of 100, and a Schmidt number of 2, demonstrated numerical stability and accuracy, providing confidence in the simulation data for potential engineering applications.

IX. Appendix

```
% Clear all variables from the workspace and clear the command window, start a timer
clear all;
clc;


% Declare global variables to be accessible across functions
global xc yf yc xf xc3 yc3 Re Sc h CFL Lx Ly;


% Initialization of simulation parameters
M = 48*8; % Number of grid points in x-direction
N = 32 *8 ; % Number of grid points in y-direction


h = 3/M; % Grid spacing based on domain size and number of points
t = 0; % Initial time

CFL = 0.9;
Re = 100; % Reynolds number
Sc = 2; % Schmidt number

Lx = 3; % Length of domain in x-direction
Ly = 2; % Length of domain in y-direction
xf = linspace(0, Lx, M+1)'; % Face-centered grid points in x
xc = linspace(-h/2, Lx+h/2, M+2)'; % Cell-centered grid points in x
yf = linspace(0, Ly, N+1)'; % Face-centered grid points in y
yc = linspace(-h/2, Ly+h/2, N+2)'; % Cell-centered grid points in y
xc3 = linspace(-5*h/2, Lx+5*h/2, M+6)'; % Extended grid for boundary treatment in x
yc3 = linspace(-5*(1/N), Ly+5*1/N, N+6)'; % Extended grid for boundary treatment in y

% Set initial conditions and boundary conditions for velocity and scalar fields
u = zeros(M+1, N+2); % Initialize horizontal velocity component
v = zeros(M+2, N+1); % Initialize vertical velocity component
Y = zeros(M+6, N+6); % Initialize scalar field Y
u = bc_u(u, t); % Apply boundary conditions to u
v = bc_v(v, t); % Apply boundary conditions to v
Y = bc_Y3(Y, t); % Apply boundary conditions to Y
[u, v] = correctOutlet(u, v); % Correct outlet velocities
```

```
% Setup and solve the Poisson equation for pressure correction
dt = 1; % Time step
phi = zeros(M+2, N+2); % Initialize pressure field
phi = bcGS(phi); % Apply boundary conditions for Gauss-Seidel solver
divergenceV = calcDivV(u, v); % Calculate divergence of velocity field
f = (1/dt) * divergenceV; % Source term for Poisson equation
check = sum(f, 'all'); % Check if source term is nearly zero
nIterMax = 30; % Maximum number of iterations for Poisson solver
ep = 1*10^(-4); % Convergence criterion
phi = myPoisson(phi, f, h, nIterMax, ep); % Solve Poisson equation
[u, v] = projectV(u, v, phi, dt); % Correct velocities to ensure divergence-free
condition
u = bcGhost_u(u, t); % Update ghost cell boundaries for u
v = bcGhost_v(v, t); % Update ghost cell boundaries for v

% Main simulation loop setup
timeplot = [4, 4.25, 4.5, 4.75, 5, 5.25, 5.5, 5.75, 6, 10]; % Times at which to plot
results
outputTime = [4, 4.25, 4.5, 4.75, 5, 5.25, 5.5, 5.75, 6, 10] ; % Times at which to
output results
ifig = 1; % Index for plotting figures
counterSK = 1; % Step counter for kinetics
Timeroutputt = 1; % Counter for output times

% Pre-calculation of hyperbolic terms for velocity updates
[Hu, Hv] = hyperbolic_uv_2D(u, v); % Calculate hyperbolic terms for u and v

% Initial conditions for iterative solutions
Hu0 = Hu; % Store initial hyperbolic term for u
Hv0 = Hv; % Store initial hyperbolic term for v
u0 = u; % Store initial condition for u
v0 = v; % Store initial condition for v

% Initial calculation of source terms and kinetics
S(1) = calcS3(Y); % Calculate initial source term S
k(1) = calck(u, v); % Calculate initial kinetic term k

% Time stepping loop
while t <= 10
    [dt, outputFlag] = calcDtBurgers561(t, outputTime(Timeroutputt), u, v); %
Calculate stable time step
    [Hu, Hv] = hyperbolic_uv_2D(u, v); % Update hyperbolic terms for u and v

    % Calculation of source terms for heat and mass transfer
    [Qu, Qv, QY1] = calcSourceIBFinal(u, v, Y, t, dt); % Calculate initial source
terms
    u = parabolic_CN1_2D_u(u, 1.5*Hu - 0.5*Hu0 + Qu, dt); % Update u using CN and
Adams-Bashforth methods
    v = parabolic_CN1_2D_v(v, 1.5*Hv - 0.5*Hv0 + Qv, dt); % Update v using CN and
Adams-Bashforth methods
    t = t + dt/2; % Update time by half step for stability

    [Qu, Qv, QY2] = calcSourceIBFinal(u, v, Y, t, dt); % Recalculate source terms for
heat and mass transfer
    u = parabolic_CN2_2D_u(u, 1.5*Hu - 0.5*Hu0 + Qu, dt); % Second update for u
```

```
    v = parabolic_CN2_2D_v(v, 1.5*Hv - 0.5*Hv0 + Qv, dt); % Second update for v

    HY = hyperbolic_Y_WENO_2D(Y, u0, v0, u, v, dt); % Calculate hyperbolic terms for
scalar field Y
    Y = parabolic_CN1_2D_Y3(Y, QY1 + HY, dt); % First update for Y using CN and
Adams-Bashforth methods
    Y = parabolic_CN2_2D_Y3(Y, QY2 + HY, dt); % Second update for Y
    t = t + dt/2; % Complete time step update

    % Boundary condition application and correction at each time step
    u = bcGhost_u(u, t); % Update ghost cells for u
    v = bcGhost_v(v, t); % Update ghost cells for v
    u = bc_u(u, t); % Reapply boundary conditions for u
    v = bc_v(v, t); % Reapply boundary conditions for v
    [u, v] = correctOutlet(u, v); % Correct outlet velocities

    % Solve Poisson equation for pressure correction at new time step
    divergenceV = calcDivV(u, v);
    f = (1/dt) * divergenceV; % Source term for Poisson equation
    check = sum(f, 'all'); % Check if source term is close to zero
    phi = myPoisson(phi, f, h, nIterMax, ep); % Solve Poisson equation
    [u, v] = projectV(u, v, phi, dt); % Project velocity field to be divergence-free

    % Update velocity fields and hyperbolic terms for next iteration
    u0 = u; % Update stored initial condition for u
    v0 = v; % Update stored initial condition for v
    Hu0 = Hu; % Update stored hyperbolic term for u
    Hv0 = Hv; % Update stored hyperbolic term for v

    counterSK = counterSK + 1; % Increment counter for kinetics
    S(counterSK) = calcS3(Y); % Calculate new source term S
    k(counterSK) = calck(u, v); % Calculate new kinetic term k
    tcounter1(counterSK) = t; % Update time vector for plotting

    % Check if output time is reached and plot if necessary
    if t <= 6
        if outputFlag == 1
            Timeroutputt = Timeroutputt + 1; % Increment output time counter

            % Plotting results for u, v, and Y at specified times
            examFig1 = figure(1);
            subplot(3,3,ifig);
            pcolor(xf, yc, u');
            shading interp;
            colormap('jet');
            colorbar;
            xlim([0 3]);
            ylim([0 2]);
            clim([-2.5 2.5]);
            xlabel('X-Direction');
            ylabel('Y-Direction');
           title(['u, t = ', num2str(timeplot(ifig))]);

            pbaspect([3 2 1]);
```

```
            examFig2 = figure(2);
            subplot(3,3,ifig);
            pcolor(xc, yf, v');
            shading interp;
            colormap('jet');
            colorbar;
            xlim([0 3]);
            ylim([0 2]);
            clim([-2.5 2.5]);
            xlabel('x');
            ylabel('y');
            title(['v, t = ' num2str(timeplot(ifig))]);
            pbaspect([3 2 1]);

            examFig3 = figure(3);
            subplot(3,3,ifig);
            pcolor(xc3, yc3, Y');
            shading interp;
            colormap('hot');
            colorbar;
            xlim([0 3]);
            ylim([0 2]);
            clim([0 1]);
            xlabel('x');
            ylabel('y');
            title(['Y t = ' num2str(timeplot(ifig))])
            pbaspect([3 2 1]);

            ifig = ifig + 1; % Increment figure index for next plot
        end
    end

    % Terminate simulation when time reaches 10 seconds
    if t == 10
        break;
    end
end

% Calculate and display averaged kinetic and scalar quantities over the simulation
period
K_bar = myTrapezoidal(tcounter1', k) / 10 % Calculate average kinetic energy
R_bar = myTrapezoidal(tcounter1', S) / 10 % Calculate average scalar quantity

% Plot kinetic and scalar functions over time
examFig4 = figure(4);
plot(tcounter1, k, 'k');
grid on;
title(['k(t) , Re=', num2str(Re), ', Sc=', num2str(Sc)]);
xlabel('t');
ylabel('k(t)');
xlim([0 10]);
pbaspect([3 2 1]);

examFig5 = figure(5);
plot(tcounter1, S, 'k');
```

```
grid on;
title(['S(t) , Re=', num2str(Re), ', Sc=', num2str(Sc)]);
xlabel('t');
ylabel('S(t)');
xlim([0 10]);
ylim([0 0.25]);
pbaspect([3 2 1]);
```

```
   function u = bc_u(u, t)
    global xf yc;

     M = size(u,1)-1;
     N = size(u,2)-2;

z1 =@(yc) (yc - 0.25)/1;
z2 =@(xf) (xf -1.5)/0.5;
z3 =@(xf) (xf - 2)/0.5;

% left boundary cell centered with outlet = 0
for j =1:length(yc)
    if yc(j) >= 0.25 && yc(j) <= 0.75
        u(1,j) = (4*u(2,j)-u(3,j))/3; % nuemann with the derivative cancelling to
zero cell centered
    else
        u(1,j) = 0;
    end
end

% right boundary cell centered
for j =1:length(yc)
    i = M+1;
    if yc(j) >= 0.25 && yc(j) <= 1.25
        u(i,j) =  9/2*cos(pi).*z1(yc(j)).*(1-z1(yc(j))) ; %dirichlet b.c cell
centered
    else
        u(i,j) = 0;
    end
end


% bottom boundary right on node
for i = 1:M
    j = 1;
    if xf(i) >= 1.5 && xf(i) <= 2
        u(i,j) = 2*2 * cos(pi/3)*6.*z2(xf(i)).*(1-z2(xf(i))) - u(i,j+1);   %
dirichlet node based
    else
```

```
        u(i,j) = -u(i,j+1);
    end
end

%% top boundary right on node
for i = 1:length(xf)
    j = N+2;
    if xf(i) >= 2 && xf(i) <= 2.5
        u(i,j) = 2*3*cos(-pi/6) * 6*z3(xf(i))*(1-z3(xf(i)))-u(i,j-1);  % dirichlet
node based
    else
        u(i,j) = -u(i,j-1);
    end
end

end
function [v] = bc_v(v, t)
    global xc yf;
    %Cell Center mesh**Node Based Mesh
    M = size(xc,1); N = size(yf,1);
    %Fz function
    %Set values
    alpha1=pi;alpha2=pi/3;alpha3=-pi/6; U1=3/4;U2=2;U3=3;

z1 =@(yf) (yf - 0.25)/1; z2 =@(xc) (xc -1.5)/0.5; z3 =@(xc) (xc - 2)/0.5;
%%%%%%%%%%%%%% Boundaries %%%%%%%%%%%%
% Top boundary
for i = 1:M
    j = N;
    if (xc(i) >= 2) && (xc(i) <= 2.5)
        v(i,j) = U3*sin(alpha3) * 6*z3(xc(i))*(1-z3(xc(i)));
    else
        v(i,j) = 0;
    end
end


% Bottom boundary
for i = 1:M
    j = 1;
    if (xc(i) >= 1.5) && (xc(i) <= 2)
        v(i,j) = U2*sin(alpha2)*6*z2(xc(i))*(1-z2(xc(i)));   % dirichlet node based
    else
        v(i,j) = 0;
    end
end
% Right boundary
for j =1:N
    i = M;
    if (yf(j) >= 0.25) && (yf(j) <= 1.25)
        v(i,j) = 2* U1* sin(pi)*z1(yf(j))*(1-z1(yf(j))) - v(i-1,j);
    else
        v(i,j) = -v(i-1,j);
    end
end
```

```
% left boundary
for j =1:N
    i = 1;
    if (yf(j) >= 0.25) && (yf(j) <= 0.75)
        v(i,j) = v(i+1,j);
    else
        v(i,j) = - v(i+1,j);
    end
end


end
function Y = bc_Y3(Y,t)
    global xc3 yc3;
    % Cell Center mesh**Node Based Mesh
    M = size(xc3, 1);
    N = size(yc3, 1);

    % Set values
    alpha1 = pi; alpha2 = pi / 3; alpha3 = -pi / 6; U1 = 3 / 4; U2 = 2; U3 = 3;

    % Define your z-functions based on the problem statement
    z1 = @(yf) (yf - 0.25);
    z2 = @(xc) (xc - 1.5) / 0.5;
    z3 = @(xc) (xc - 2) / 0.5;

    %%%%%%%%%%%%%%% Boundaries %%%%%%%%%%%%
    % Left boundary

    for i = 1:3
        for j = 4:N-3
            if (yc3(j) >= 0.25) && (yc3(j) <= 0.75)
                Y(i, j) = Y(7 - i, j);  % Mirror values across a specified line
            else
                Y(i, j) = Y(7 - i, j);  % Mirror and negate values
            end
        end
    end

 %bottom boundary ghost cells

      for j = 1:3
        for i = 4:M-3
            if (xc3(i) >= 1.5) && (xc3(i) <= 2)
                Y(i, j) = -Y(i,7- j);  % Mirror values across a specified line
            else
                Y(i, j) = Y( i, 7-j);  % Mirror and negate values
            end
        end
      end

      %right boundary
      for j = 1:N
        for i = 1:M
```

```
            if i==M-2
                if yc3(j)>= 0.25 && yc3(j)<= 1.25
                    Y(i,j)=2-Y(i-1,j);
                else
                    Y(i,j)=Y(i-1,j);
                end
            elseif i==M-1
                if yc3(j)>= 0.25 && yc3(j)<= 1.25
                     Y(i,j)=2-Y(i-3,j);
                else
                    Y(i,j)=Y(i-3,j);
                end
            elseif i==M
                if yc3(j)>= 0.25 && yc3(j)<= 1.25
                     Y(i,j)=2-Y(i-5,j);
                else
                    Y(i,j)=Y(i-5,j);
                end
            end
        end
      end

    %TOP

      for j = 1:N
        for i = 1:M
            if j==N-2
                if xc3(i)> 2 && xc3(i)< 2.5
                    Y(i,j)=-Y(i,j-1);
                else
                    Y(i,j)=Y(i,j-1);
                end
            elseif j==N-1
                if xc3(i)> 2 && xc3(i)< 2.5
                     Y(i,j)=-Y(i,j-3);
                else
                    Y(i,j)=Y(i,j-3);
                end
            elseif j==N
                if xc3(i)> 2 && xc3(i)< 2.5
                     Y(i,j)=-Y(i,j-5);
                else
                    Y(i,j)=Y(i,j-5);
                end
            end
        end
      end


end
function psi = psiWENO(a,b,c,d)


epsilon = 10^(-6);
```

```
IS0 = 13*(a-b)^2 + 3*(a-3*b)^2;
IS1 = 13*(b-c)^2 + 3*(b+c)^2;
IS2 = 13*(c-d)^2 + 3*(3*c-d)^2;
%solving the alphas
alpha0 = 1/((epsilon+IS0)^2);
alpha1 = 6/((epsilon+IS1)^2);
alpha2 = 3/((epsilon+IS2)^2);

omega0 = alpha0/(alpha0 + alpha1 + alpha2);
omega2 = alpha2/(alpha0 + alpha1 + alpha2);
%Now solving the final psi
psi = (1/3)*omega0*(a-2*b+c) + (1/6)*(omega2 - 0.5)*(b - 2*c +d);

end
function [u, v] = correctOutlet(u, v)
    global h yc ucorr
% Dimentions of the mesh
    M = size(v, 1) - 2; N = size(u, 2) - 2;
    L = 0.5;


    q =   sum(v(2:end-1,1) * h) - sum(v(2:end-1,end) * h)+sum(u(1,2:end-1) * h) -
sum(u(end,2:end-1) * h);
    ucorr = q / L  ;

    for j = 2:N+1

            if (yc(j) >= 0.25) && (yc(j) <= 0.75)
            u(1,j) = u(1,j) - ucorr;
            end
    end
end
function phi = bcGS(phi)

s = size(phi);
M = s(1)-2;
N = s(2)-2;


%elvaluate the LHS and RHS
for j = 1:N+1
    phi(1,j) = phi(2,j); %LHS
    phi(M+2,j) = phi(M+1,j); %RHS
end

%evaluate the lower and upper boundary
for i = 1:M+1
    phi(i,1) = phi(i,2); %Lower
    phi(i,N+2) = phi(i,N+1); %upper
end

for j = 1:N+1
    phi(1,j) = phi(2,j); %LHS
    phi(M+2,j) = phi(M+1,j); %RHS
end
```

```
for i = 1:M+1
    phi(i,1) = phi(i,2); %Lower
    phi(i,N+2) = phi(i,N+1); %Upper
end

end
function divV = calcDivV(u, v)
    global h


    M = size(v,1) - 2;  N = size(v,2) - 1;

    divV = zeros(M+2,N+2);

    for i = 2:M+1
        for j = 2:N+1

            divV(i,j) = (u(i,j) - u(i-1,j)) / (h)+(v(i,j) - v(i,j-1)) / (h);
        end
    end
end
function [phi, Linf, iter] = myPoisson(phi, f, h, nIterMax, epsilon)

    % Calculate the size of the problem domain by excluding the boundary layers.
    M = size(phi,1)-2; % The number of interior points in the x-direction.
    N = size(phi,2)-2; % The number of interior points in the y-direction.

    % Initialize the iteration counter.
    iter=0;

    % Begin the iterative process for solving the equation.
    for i=1:nIterMax
        % Apply a multigrid method to smooth the solution 'phi' given the right-hand
side 'f' and grid spacing 'h'.
        phi = myMultigrid(phi, f, h); % Initial smoothing

        % Compute the norm of the relative residual to assess convergence.
        % This involves calculating the difference between the left and right-hand
sides of the equation,
        % normalized by the magnitude of 'f' to gauge the accuracy of the current
solution 'phi'.
        norm(i,:) = myRelResNorm(phi, f, h);

        % Determine the maximum value of the norm across all grid points,
        % representing the largest error in the current iteration.
        Linf = max(abs(norm(i,:)));

        % Check if the current solution's accuracy meets the convergence criterion.
        if Linf < epsilon
            break % Exit the loop if the solution is within the desired tolerance.
        end

        % Increment the iteration counter.
        iter = iter + 1;
```

```
    end
end

% Defines the function 'myMultigrid' that takes an initial guess 'phi', the right-
hand side 'f',
% and the grid spacing 'h' as inputs. The function returns an updated 'phi' that is
closer to the
% true solution of the PDE being solved.
function [phi] = myMultigrid(phi, f, h)

% Calculate the effective dimensions of the grid by subtracting the boundary cells.
M = size(phi, 1)-2; N = size(phi, 2)-2;

phi = myGaussSeidel(phi, f, h, 1);

% Check if both the grid dimensions are even, which is necessary for the
% multigrid coarsening process to proceed correctly.
if mod(M, 2) == 0 && mod(N, 2) == 0
    % Calculate the residual 'rh' on the current grid. The residual measures
    % the difference between the left-hand side and the right-hand side of the
    % PDE given the current guess 'phi'.
    rh = myResidual(phi, f, h);

    % Restrict the residual to a coarser grid, halving the grid resolution.
    % This operation aggregates information from the fine grid to a coarser one,
    % effectively focusing on lower-frequency errors.
    r2h = myRestrict(rh);

    % Initialize the error correction 'e2h' on the coarser grid with zeros.
    e2h = zeros(M/2 + 2, N/2 + 2); % Note: '+2' maintains boundary cells.


    e2h = myMultigrid(e2h, r2h, 2*h);

    % Prolongate (interpolate) the error correction back to the finer grid.
    eh = myProlong(e2h);

    % Update the solution 'phi' on the fine grid by adding the error correction.
    phi = phi + eh;

    % Apply boundary conditions to 'phi' using the function 'bcGS'.
    phi = bcGS(phi);

    % Perform another Gauss-Seidel smoothing step after adding the error correction.
    phi = myGaussSeidel(phi, f, h, 1);
else

    phi = myGaussSeidel(phi, f, h, 6);
end
end


function phi = myGaussSeidel(phi, f, h, niter)
    global omega; % Declare omega as a global variable so it can be used across
different functions or workspace
```

```
    % Calculate the number of interior points in each dimension
    M = size(phi, 1) - 2; % Subtracting 2 accounts for the boundary points
    N = size(phi, 2) - 2; % Subtracting 2 accounts for the boundary points


    % Initialize the iteration counter to start the iterative process
    k = 1;

    % Begin the iteration loop, which will run 'niter' times
    while k <= niter

        for j = 2:N+1 % Loop over interior points (excluding boundaries)
            for i = 2:M+1 % Loop over interior points (excluding boundaries)
                % Update the solution phi using the SOR formula
                % The update takes the average of the neighboring points, adjusts
with the right-hand side f, and scales by omega
                phi(i,j) =  (phi(i,j-1) + phi(i-1,j) + phi(i+1,j) + phi(i,j+1)) / 4 -
h^2 * f(i,j) / 4;
            end
        end

        k = k + 1; % Increment the iteration counter
        phi = bcGS(phi); % Apply boundary conditions using the bcGS function after
updating the solution
    end

end % End of the function
```

```
function r = myResidual(phi, f, h)
    % Compute the size of the computational domain, excluding ghost cells
    M = size(phi, 1) - 2;
    N = size(phi, 2) - 2;
    r = zeros(size(phi)); % Initialize the residual matrix




    for j = 2:N+1
        for i = 2:M+1
            % Compute residuals for the inner (non-ghost) cells
            r(i,j) = f(i,j) - ((phi(i+1,j) - 2*phi(i,j) + phi(i-1,j))/(h^2) +
(phi(i,j+1) - 2*phi(i,j) + phi(i,j-1))/(h^2));
        end
    end


    % The outermost cells (ghost cells) of r remain zero by initialization
end
```

```
function r2h = myRestrict(rh)
    % Calculate the dimensions of the finer grid excluding boundary layers.
    M = size(rh,1) - 2;
    N = size(rh,2) - 2;
    % Compute the dimensions of the coarser grid by halving the dimensions of the
finer grid.
    Mc = M / 2;
    Nc = N / 2;

    % Initialize the coarser grid 'r2h' with zeros, including space for boundaries.
    r2h = zeros(Mc + 2, Nc + 2);

    % Iterate through each cell in the coarser grid.
    for i = 2:Mc + 1
        for j = 2:Nc + 1
            % Compute the average of four corresponding cells in the finer grid 'rh'
to
            % obtain a single value in the coarser grid 'r2h'. This operation
effectively
            % reduces the resolution of the residual information, preparing it for
further
            % processing at the coarser level.
            % The averaging process helps in preserving the overall error
characteristics
            % while transitioning between grid resolutions.
            r2h(i,j) = 0.25 * (rh(2*i - 2, 2*j - 2) + rh(2*i - 1, 2*j - 2) + ...
                               rh(2*i - 1, 2*j - 1) + rh(2*i - 2, 2*j - 1));
        end
    end
end
function eh = myProlong(e2h)
    % Calculate the dimensions of the coarser grid excluding boundary layers.
    M = size(e2h, 1) - 2;
    N = size(e2h, 2) - 2;
    % Double the dimensions for the finer grid, since it's a step up in resolution.
    Mc = M * 2;
    Nc = N * 2;

    % Initialize the finer grid 'eh' with zeros, including space for boundaries.
    eh = zeros(Mc + 2, Nc + 2); % Initialize output matrix

    % Iterate through each cell in the coarser grid.
    for i = 2:M+1
        for j = 2:N+1

            % Assign the value from the coarser grid to the corresponding position in
the
            % finer grid. Each cell in the coarser grid maps to four cells in the
finer grid,
            % essentially copying the coarse-grid value to multiple cells in the
finer grid
            % without performing any averaging or interpolation.
```

```
            eh(2*i-2, 2*j-2) = e2h(i,j); % Bottom-left corner of the target cell in
the finer grid
            eh(2*i-1, 2*j-2) = e2h(i,j); % Top-left corner
            eh(2*i-2, 2*j-1) = e2h(i,j); % Bottom-right corner
            eh(2*i-1, 2*j-1) = e2h(i,j); % Top-right corner

        end
    end

    eh = bcGS(eh);
end


function rr = myRelResNorm(phi, f, h)
    % Compute the size of the computational domain, excluding ghost cells
    M = size(phi, 1);
    N = size(phi, 2) ;
    r = myResidual(phi,f,h);


    rr=(max(max(r)))/((max(max(f))));
end



function [u, v] = projectV(u, v, phi, dt)
    global t h;

    M = size(u, 1);N = size(u, 2); K = size(v,1);Y = size(v,2);
    for i = 2:K-1
        for j = 2:Y-1
            gphi = (phi(i,j+1)-phi(i,j))/h;
            v(i,j) = v(i,j) - dt*gphi;
        end
    end

    for i = 2:M-1
        for j = 2:N-1
            ggphi = (phi(i+1,j)-phi(i,j))/h;
            u(i,j) = u(i,j) - dt*ggphi;
        end
    end
    u = bcGhost_u(u,t+dt);
    v = bcGhost_v(v,t+dt);

end


function u = bcGhost_u(u,t)
global xf;

    M = size(u,1); N = size(u,2);
    z2 =@(xf) (xf -1.5)/0.5;  z3 =@(xf) (xf - 2)/0.5;
%Bottom
   for i = 1:M
```

```
        j = 1;
        if (xf(i) >= 1.5) && (xf(i) <= 2)
            u(i,j) =2 * 2* cos(pi/3)*6*z2(xf(i))*(1-z2(xf(i))) - u(i,j+1);
        else
            u(i,j) = -u(i,j+1);
        end
    end

%top
    for i = 1: M
        j = N;
        if (xf(i) >= 2) && (xf(i) <= 2.5)
        u(i,j) = 2*3*cos(-pi/6)* 6*z3(xf(i))*(1-z3(xf(i)))-u(i,j-1);
        else
        u(i,j) = -u(i,j-1);
    end
end
function v = bcGhost_v(v,t)
global xc yf;

    M = size(v,1);
    N = size(v,2);

    z1 = @(yf)(yf-0.25)/1;

    % right
    for j =1:N
             i = M;
         if (yf(j) >= 0.25) && (yf(j) <= 1.25)
             v(i,j) = 2* 9/2 *sin(pi)*z1(yf(j))*(1-z1(yf(j))) - v(i-1,j);
         else
             v(i,j) = -v(i-1,j);
        end
    end

   %left
    for j =1:N
         i = 1;
        if (yf(j) >= 0.25) && (yf(j) <= 0.75)
         v(i,j) = v(i+1,j);
        else
         v(i,j) = - v(i+1,j);
        end
     end



end

function [Hu, Hv] = hyperbolic_uv_2D(u, v)
    global h;


    Hu = zeros(size(u));
    Hv = zeros(size(v));
```

```
    % Calculate hyperbolic terms for u-velocity
    for j = 2:size(u,2)-1

        for i = 2:size(u,1)-1

        Hu(i,j) = -((0.5*(u(i+1,j)+u(i,j)))^2 - (0.5*(u(i,j)+u(i-1,j)))^2)/h -...
           (0.25*(u(i,j)+u(i,j+1))*(v(i,j)+v(i+1,j)) - 0.25*(u(i,j-1)+u(i,j))*(v(i,j-
1)+v(i+1,j-1)))/h;


        end
    end

    % Calculate hyperbolic terms for v-velocity
    for i = 2:size(v,1)-1
        for j = 2:size(v,2)-1

           Hv(i,j) = -((0.5*(v(i,j+1)+v(i,j)))^2 - (0.5*(v(i,j)+v(i,j-1)))^2)/h -...
           (0.25*(u(i,j)+u(i,j+1))*(v(i,j)+v(i+1,j)) - 0.25*(u(i-1,j+1)+u(i-
1,j))*(v(i,j)+v(i-1,j)))/h;

        end
    end

end
function S = calcS3(Y)


    global Lx Ly h;


    M = size(Y, 1) - 6; % Subtracting 6 to account for 3 ghost layers on each side
    N = size(Y, 2) - 6; % Subtracting 6 to account for 3 ghost layers on each side

    % Initialize S
    S = 0;

    % Calculate the integral using the composite 2D midpoint rule
    for i = 4:(M+3) % Starting and ending indices account for ghost cells
        for j = 4:(N+3) % Starting and ending indices account for ghost cells
            % Midpoint value for the current cell
            Ymid = Y(i,j);
            % Increment S using the midpoint rule for the integral
            S = S + Ymid * (1 - Ymid);
        end
    end

    % Multiply by the area element (h^2) and normalize by Lx*Ly
    S = (h^2 / (Lx * Ly)) * S;
end

function k = calck(u,v)

    global Lx Ly h;
```

```
    % Calculate the midpoints for u and v
    umid = (u(1:end-1, 2:end-1) + u(2:end, 2:end-1)) / 2;
    vmid = (v(2:end-1, 1:end-1) + v(2:end-1, 2:end)) / 2;

    % Calculate the kinetic energy using the midpoint rule for integration
    k = 0.5 * h^2 * sum(sum(umid.^2 + vmid.^2));
end
function [dt, outputFlag] = calcDtBurgers561(t, outputTime, u, v)
    global h CFL

    % Calculate the maximum absolute velocities for u and v
    maxAbsU = max(abs(u(:)));
    maxAbsV = max(abs(v(:)));


        Delta_t_u = CFL * h / ( maxAbsU + maxAbsV);

        Delta_t_v = CFL * h / (maxAbsU +  maxAbsV);



    dt = min(Delta_t_u, Delta_t_v);
     % Initialize the output flag to 0
    outputFlag = 0;

    % Adjust dt to reach outputTime without exceeding it
    if (t + dt > outputTime)
        dt = outputTime - t;
        outputFlag = 1;
    end

end

function [Hu, Hv] = hyperbolic_uv_2D(u, v)
    global h;


    Hu = zeros(size(u));
    Hv = zeros(size(v));

    % Calculate hyperbolic terms for u-velocity
    for j = 2:size(u,2)-1

        for i = 2:size(u,1)-1

        Hu(i,j) = -((0.5*(u(i+1,j)+u(i,j)))^2 - (0.5*(u(i,j)+u(i-1,j)))^2)/h -...
           (0.25*(u(i,j)+u(i,j+1))*(v(i,j)+v(i+1,j)) - 0.25*(u(i,j-1)+u(i,j))*(v(i,j-
1)+v(i+1,j-1)))/h;


        end
    end
```

```
    % Calculate hyperbolic terms for v-velocity
    for i = 2:size(v,1)-1
        for j = 2:size(v,2)-1

           Hv(i,j) = -((0.5*(v(i,j+1)+v(i,j)))^2 - (0.5*(v(i,j)+v(i,j-1)))^2)/h -...
           (0.25*(u(i,j)+u(i,j+1))*(v(i,j)+v(i+1,j)) - 0.25*(u(i-1,j+1)+u(i-
1,j))*(v(i,j)+v(i-1,j)))/h;

        end
    end

end
function u = parabolic_CN1_2D_u(u, Qu, dt)
global Re t h xc yf;
global a b c d;

M = size(u,1) - 1 ;
N = size(u,2) - 1 ;

% Initiate the variable terms
a = zeros(size(u));
b = zeros(size(u));
c = zeros(size(u));
d = zeros(size(u));


    for j = 2: N
      a(2:M+1,j) = - dt/(2*Re*h^2);         % a term
      b(2:M+1,j) = 1+ (dt/ (Re*h^2));       % b term
      c(2:M+1,j) = - dt/ (2*Re*h^2);        % c term
      d(2:M+1,j) = u(2:M+1,j) + ((dt/(2*Re*h^2))*(u(2:M+1,j+1)-
2*u(2:M+1,j)+u(2:M+1,j-1))) +  (dt/2) * Qu(2:M+1,j);     % d term
    end


    [a b c d] = bcCN1_u(a,b,c,d,t + dt); % apply boundary conditions
    for j =2:N
    u(2:M+1,j) = mySolveTriDiag(a(2:M+1,j),b(2:M+1,j),c(2:M+1,j),d(2:M+1,j));       %
call function to solve tridiagonal
    end

    %Apply boundary conditions bc_v
    u = bc_u(u, t + dt/2);  % apply boundary conditions
end

function v = parabolic_CN1_2D_v(v, Qv, dt)
    global Re t h xc yf a b c d ; % Use global variables
    tt=0.1; % Seems to be an unused variable

    % Determine the dimensions of the velocity field minus 1 to account for ghost
cells or boundaries
    M = size(v,1) - 1;
    N = size(v,2) - 1;

    % Initialize the coefficient matrices for the tridiagonal system solver
```

```
    a = zeros(size(v)); b = zeros(size(v)); c = zeros(size(v)); d = zeros(size(v));

    % Fill in the coefficient matrices based on the Crank-Nicolson scheme
    for j = 2:N
        a(2:M+1,j) = -dt / (2*Re*h^2);
        b(2:M+1,j) = 1 + (dt / (Re*h^2));
        c(2:M+1,j) = -dt / (2*Re*h^2);

        % Update the right-hand side 'd' based on the previous timestep and the
source term
        d(2:M+1,j) = v(2:M+1,j) + ((dt/(2*Re*h^2)) * (v(2:M+1,j+1) - 2*v(2:M+1,j) +
v(2:M+1,j-1))) + (dt/2) * Qv(2:M+1,j);
    end


    % Apply boundary conditions before solving the linear system
    [a, b, c, d] = bcCN1_v(a, b, c, d, t+dt);

    % Solve the tridiagonal system for each column
    for j = 2:N
        v(2:M+1,j) = mySolveTriDiag(a(2:M+1,j), b(2:M+1,j), c(2:M+1,j), d(2:M+1,j));
    end

    % Apply boundary conditions to the velocity field after the update
    v = bc_v(v, t+dt/2);
end

function [a, b, c, d] = bcCN1_v(a, b, c, d, t)
    % Access global variables: yf (the face-centered grid points in y-direction) and
M (not used within this function but likely relevant in the broader context)
    global yf M;

    % Determine the size of yf, which defines the grid's dimension in y-direction
    N = size(yf,1);

    % Define constants used in boundary conditions, including angles (alpha1, alpha2,
alpha3) and velocities (U1, U2, U3)
    alpha1 = pi; alpha2 = pi/3; alpha3 = -pi/6;
    U1 = 3/4; U2 = 2; U3 = 3;

    % Define functions z1, z2, z3 as transformations of coordinates yf and xc,
adjusting their scales/positions
    z1 =@(yf) (yf - 0.25)/1;
    z2 =@(xc) (xc - 1.5)/0.5;
    z3 =@(xc) (xc - 2)/0.5;

    % Apply boundary conditions on the left side of the domain
    for j = 2:N
        i = 2; % Fixed column index for the left boundary
        if yf(j) >= 0.25 && yf(j) <= 0.75
            b(i,j) = b(i,j) + a(i,j); % Modify b based on a for specified yf range
        else
            b(i,j) = b(i,j) - a(i,j); % Default modification of b outside the
specified yf range
        end
```

```
    end

    % Apply boundary conditions on the right side of the domain
    for j = 2:N
        % Conditional modifications based on yf's position
        if yf(j) >= 0.25 && yf(j) <= 1.25
            % Specific adjustment for b and d at the domain's right edge, involving
calculated terms with U1, z1, and alpha1
            d(end-1,j) = -2* U1 * z1(yf(j)) * (1 - z1(yf(j))) * c(end-1,j) *
sin(alpha1) + d(end-1,j);
            b(end-1,j) = b(end-1,j) - c(end-1,j);
        else
            % Default modification of b at the right boundary
            b(end-1,j) = b(end-1,j) - c(end-1,j);
        end
    end
end

   function u = bc_u(u, t)
    global xf yc;

     M = size(u,1)-1;
     N = size(u,2)-2;

z1 =@(yc) (yc - 0.25)/1;
z2 =@(xf) (xf -1.5)/0.5;
z3 =@(xf) (xf - 2)/0.5;

% left boundary cell centered with outlet = 0
for j =1:length(yc)
    if yc(j) >= 0.25 && yc(j) <= 0.75
        u(1,j) = (4*u(2,j)-u(3,j))/3; % nuemann with the derivative cancelling to
zero cell centered
    else
        u(1,j) = 0;
    end
end

% right boundary cell centered
for j =1:length(yc)
    i = M+1;
    if yc(j) >= 0.25 && yc(j) <= 1.25
        u(i,j) =  9/2*cos(pi).*z1(yc(j)).*(1-z1(yc(j))) ; %dirichlet b.c cell
centered
    else
        u(i,j) = 0;
    end
end


% bottom boundary right on node
for i = 1:M
    j = 1;
    if xf(i) >= 1.5 && xf(i) <= 2
```

```
        u(i,j) = 2*2 * cos(pi/3)*6.*z2(xf(i)).*(1-z2(xf(i))) - u(i,j+1);   %
dirichlet node based
    else
        u(i,j) = -u(i,j+1);
    end
end

% top boundary right on node
for i = 1:length(xf)
    j = N+2;
    if xf(i) >= 2 && xf(i) <= 2.5
        u(i,j) = 2*3*cos(-pi/6) * 6*z3(xf(i))*(1-z3(xf(i)))-u(i,j-1);  % dirichlet
node based
    else
        u(i,j) = -u(i,j-1);
    end
end

end
function u  =parabolic_CN2_2D_u(u,Qu,dt)
global Re t h a b c d;

sz = size(u);
M = sz(1)-1;
N = sz(2)-2;

a = zeros(sz);
b = zeros(sz);
c = zeros(sz);
d = zeros(sz);

dx = (1/Re).*dt/h^2;
d1 = dx/2;
d2 = d1;

for i = 2: M
    a(i,2:N+1) = -d2;
    b(i,2:N+1) = 1+2.*d2;
    c(i,2:N+1) = -d2;
    d(i,2:N+1) = d1.*u(i+1,2:N+1)+(1-2.*d1).*u(i,2:N+1)+d1.*u(i-1,2:N+1) +
Qu(i,2:N+1).*(dt/2);
end

[a,b,c,d] = bcCN2_u(a,b,c,d,t+dt);

for i = 2:M
    u(i,2:N+1) = mySolveTriDiag(a(i,2:N+1),b(i,2:N+1),c(i,2:N+1),d(i,2:N+1));
end

u = bc_u(u,t+dt);
end

function v = parabolic_CN2_2D_v(v, Qv, dt)
    % Utilize global variables including Re (Reynolds number), t (current time),
```

```
    % h (grid spacing), xc, yf (grid coordinates), and a, b, c, d (coefficients for
tridiagonal system)
    global Re t h xc yf a b c d;

    % Initialize the dimensions M and N based on the size of the input velocity field
'v'
    M = size(v, 1) - 1; % Subtract 1 to account for the boundary
    N = size(v, 2) - 1; % Subtract 1 to account for the boundary

    % Initialize the coefficient matrices a, b, c, d to zeros with the same size as
'v'
    a = zeros(size(v));
    b = zeros(size(v));
    c = zeros(size(v));
    d = zeros(size(v));

    % Loop to populate the a, b, c, d matrices based on the Crank-Nicolson
discretization
    for i = 2:M
        a(i, 2:N) = -dt / (2 * Re * h^2);
        b(i, 2:N) = 1 + (dt / (Re * h^2));
        c(i, 2:N) = -dt / (2 * Re * h^2);

        % Update d using the previous time step velocity 'v', the Laplacian of 'v',
and the source term 'Qv'
        d(i, 2:N) = v(i, 2:N) + ((dt / (2 * Re * h^2)) * (v(i+1, 2:N) - 2 * v(i, 2:N)
+ v(i-1, 2:N))) + (dt / 2) * Qv(i, 2:N);
    end

    % Apply boundary conditions with function bcCN2_v before solving
    [a, b, c, d] = bcCN2_v(a, b, c, d, t + dt);

    % Solve the tridiagonal system for each row
    for i = 2:M
        v(i, 2:N) = mySolveTriDiag(a(i, 2:N), b(i, 2:N), c(i, 2:N), d(i, 2:N));
    end

    % Apply velocity boundary conditions with function bc_v after the update
    v = bc_v(v, t + dt);
end

function HY = hyperbolic_Y_WENO_2D(Y,u,v,u1,v1,dt)
global t xc3 yc3;

dim = size(Y); % find M and N from size of Y array
M = dim(1) - 6;
N = dim(2) - 6;

% alpha10 = 1; % known quantities from TVD-RK3 method
alpha20 = -3/4;
alpha21 = 1/4;
alpha30 = -1/12;
alpha31 = -1/12;
alpha32 = 2/3;
```

```
a0 = zeros(M+6,N+6); % initialize the mesh for a & b terms accounting for ghost cells
b0 = zeros(M+6,N+6);
a1 = zeros(M+6,N+6);
b1 = zeros(M+6,N+6);
a2 = zeros(M+6,N+6);
b2 = zeros(M+6,N+6);

for j = 4:N+3 % x direction to cell centered
    for i = 4:M+3
        a0(i,j) = (u(i-2,j-2)+u(i-3,j-2))/2;
        a1(i,j) = (u1(i-2,j-2)+u1(i-3,j-2))/2;
        a2(i,j) = (a0(i,j)+a1(i,j))/2;
    end
end

for j = 4:N+3 % y direction to cell centered
    for i = 4:M+3
        b0(i,j) = (v(i-2,j-2)+v(i-2,j-3))/2;
        b1(i,j) = (v1(i-2,j-2)+v1(i-2,j-3))/2;
        b2(i,j) = (b0(i,j)+b1(i,j))/2;
    end
end

Y1 = Y - (calc_adYdx_WENO_2D(Y,a0) + calc_bdYdy_WENO_2D(Y,b0))*dt; % step 1 of TVD-
RK3 time integration
Y1 = bc_Y3(Y1,t+dt); % update bc per formula for TVD-RK3

Y2 = Y1 - alpha20*(calc_adYdx_WENO_2D(Y,a0) + calc_bdYdy_WENO_2D(Y,b0))*dt -
alpha21*(calc_adYdx_WENO_2D(Y1,a1) + calc_bdYdy_WENO_2D(Y1,b1))*dt; % step 2 of TVD-
RK3 time integration
Y2 = bc_Y3(Y2,t+dt/2); % update bc per formula for TVD-RK3

Ys = Y2 - alpha30*(calc_adYdx_WENO_2D(Y,a0) + calc_bdYdy_WENO_2D(Y,b0))*dt -
alpha31*(calc_adYdx_WENO_2D(Y1,a1) + calc_bdYdy_WENO_2D(Y1,b1))*dt -
alpha32*(calc_adYdx_WENO_2D(Y2,a2) + calc_bdYdy_WENO_2D(Y2,b2))*dt; % step 3 of TVD-
RK3 time integration
Ys = bc_Y3(Ys,t+dt); % update bc per formula for TVD-RK3

HY = (Ys - Y)/dt; % hyperbolic terms from eq. 3

end
function Y = parabolic_CN1_2D_Y3(Y,QY,dt)
global Re Sc t h xc3 yf3 a b c d ;

%initializing M and N
M=size(Y,1)-3;
N=size(Y,2)-3;
a = zeros(size(Y));b = zeros(size(Y));c = zeros(size(Y));d = zeros(size(Y));


 for j = 4: N
      a(4:M+3,j) = - dt/(2*Re*Sc*h^2);
      b(4:M+3,j) = 1+ (dt/ (Re*Sc*h^2));
      c(4:M+3,j) = - dt/ (2*Re*Sc*h^2);
```

```
       d(4:M+3,j) = Y(4:M+3,j) + ((dt/(2*Re*Sc*h^2))*(Y(4:M+3,j+1)-
2*Y(4:M+3,j)+Y(4:M+3,j-1))) +  (dt/2)* QY(4:M+3,j);


 end

 [a,b,c,d] = bcCN1_Y3(a,b,c,d,t+dt/2);
 for j =4:N
    Y(4:M+3,j) = mySolveTriDiag(a(4:M+3,j),b(4:M+3,j),c(4:M+3,j),d(4:M+3,j));
 end

Y = bc_Y3(Y, t+dt/2);
end
function divV = calcDivV(u, v)
    global h


    M = size(v,1) - 2;  N = size(v,2) - 1;

    divV = zeros(M+2,N+2);

    for i = 2:M+1
        for j = 2:N+1

            divV(i,j) = (u(i,j) - u(i-1,j)) / (h)+(v(i,j) - v(i,j-1)) / (h);
        end
    end
end

function I = myTrapezoidal(x, y)

    % Size for the loop
    n = size(x,1);


    I = 0;

   %starting the loop
    for i = 1:n-1


        h = x(i+1) - x(i);

        I = I + (y(i) + y(i+1)) * h / 2;
    end
end


function phi = mySOR(phi, f, h, niter)
    global omega; % Declare omega as a global variable so it can be used across
different functions or workspace


    M = size(phi, 1) - 2; % Subtracting 2 accounts for the boundary points
    N = size(phi, 2) - 2; % Subtracting 2 accounts for the boundary points
```

```
    % Compute the spectral radius for the SOR method
    rho = 0.5 * (cos(pi/M) + cos(pi/N)); % The spectral radius helps determine the
optimal omega

    % Compute the relaxation factor omega for the SOR method
    omega = 2 / (1 + sqrt(1 - rho^2)); % This is the formula for over-relaxation
factor that can accelerate convergence

    % Initialize the iteration counter to start the iterative process
    k = 1;

    % Begin the iteration loop, which will run 'niter' times
    while k <= niter
        phinew = phi; % Copy the current solution to phinew to prepare for updated
values
        for j = 2:N+1 % Loop over interior points (excluding boundaries)
            for i = 2:M+1 % Loop over interior points (excluding boundaries)
                % Update the solution phi using the SOR formula
                % The update takes the average of the neighboring points, adjusts
with the right-hand side f, and scales by omega
                phi(i,j) = phi(i,j) + omega * ((phi(i,j-1) + phi(i-1,j) + phi(i+1,j)
+ phi(i,j+1)) / 4 - h^2 * f(i,j) / 4 - phi(i,j));
            end
        end

        k = k + 1; % Increment the iteration counter
        phi = bcGS(phi); % Apply boundary conditions using the bcGS function after
updating the solution
    end

end % End of the function

function HY = hyperbolic_Y_WENO_2D(Y,u,v,u1,v1,dt)
global t xc3 yc3;

dim = size(Y); % find M and N from size of Y array
M = dim(1) - 6;
N = dim(2) - 6;

% alpha10 = 1; % known quantities from TVD-RK3 method
alpha20 = -3/4;
alpha21 = 1/4;
alpha30 = -1/12;
alpha31 = -1/12;
alpha32 = 2/3;

a0 = zeros(M+6,N+6); % initialize the mesh for a & b terms accounting for ghost cells
b0 = zeros(M+6,N+6);
a1 = zeros(M+6,N+6);
b1 = zeros(M+6,N+6);
a2 = zeros(M+6,N+6);
b2 = zeros(M+6,N+6);

for j = 4:N+3 % x direction to cell centered
```

```
    for i = 4:M+3
        a0(i,j) = (u(i-2,j-2)+u(i-3,j-2))/2;
        a1(i,j) = (u1(i-2,j-2)+u1(i-3,j-2))/2;
        a2(i,j) = (a0(i,j)+a1(i,j))/2;
    end
end

for j = 4:N+3 % y direction to cell centered
    for i = 4:M+3
        b0(i,j) = (v(i-2,j-2)+v(i-2,j-3))/2;
        b1(i,j) = (v1(i-2,j-2)+v1(i-2,j-3))/2;
        b2(i,j) = (b0(i,j)+b1(i,j))/2;
    end
end

Y1 = Y - (calc_adYdx_WENO_2D(Y,a0) + calc_bdYdy_WENO_2D(Y,b0))*dt; % step 1 of TVD-
RK3 time integration
Y1 = bc_Y3(Y1,t+dt); % update bc per formula for TVD-RK3

Y2 = Y1 - alpha20*(calc_adYdx_WENO_2D(Y,a0) + calc_bdYdy_WENO_2D(Y,b0))*dt -
alpha21*(calc_adYdx_WENO_2D(Y1,a1) + calc_bdYdy_WENO_2D(Y1,b1))*dt; % step 2 of TVD-
RK3 time integration
Y2 = bc_Y3(Y2,t+dt/2); % update bc per formula for TVD-RK3

Ys = Y2 - alpha30*(calc_adYdx_WENO_2D(Y,a0) + calc_bdYdy_WENO_2D(Y,b0))*dt -
alpha31*(calc_adYdx_WENO_2D(Y1,a1) + calc_bdYdy_WENO_2D(Y1,b1))*dt -
alpha32*(calc_adYdx_WENO_2D(Y2,a2) + calc_bdYdy_WENO_2D(Y2,b2))*dt; % step 3 of TVD-
RK3 time integration
Ys = bc_Y3(Ys,t+dt); % update bc per formula for TVD-RK3

HY = (Ys - Y)/dt; % hyperbolic terms from eq. 3

end
function Y_next = hyperbolic_WENO_1D(Y, a0, a1, a2, dt)
    global h t Y1 Y2;

    % Constants for the TVD RK-3 scheme
    alpha1_0 = 1;
    alpha2_0 = 3/4; alpha2_1 = 1/4;
    alpha3_0 = 1/3; alpha3_1 = 1/3; alpha3_2 = 1/3;

    % Stage 1
    Y1 = Y - alpha1_0 * dt* calc_adYdx_WENO(Y, a0);
    Y1 = bc3(Y1, t + dt); % Apply boundary conditions

    % Stage 2
    Y2 = Y1 + alpha2_0 * dt * calc_adYdx_WENO(Y, a0) - alpha2_1 * dt *
calc_adYdx_WENO(Y1, a1);
    Y2 = bc3(Y2, t + dt/2); % Apply boundary conditions

    % Stage 3
    Y_next = Y2 + (1/12) * dt * calc_adYdx_WENO(Y, a0) + (1/12) * dt *
calc_adYdx_WENO(Y1, a1) - (2/3) * dt * calc_adYdx_WENO(Y2, a2);
    Y_next = bc3(Y_next, t + dt); % Apply boundary conditions
```

```
    % Update time
    t = t + dt;
end

function [f] = calc_bdYdy_WENO_2D(Y, b)
global h dp dmdp;
[rows, cols] = size(Y);
f = zeros(rows, cols);
dp = zeros(rows, cols);
dmdp = zeros(rows, cols);

% Calculate dp and dmdp
for i = 1:rows
    for j = 1:cols-1
        dp(i, j) = (Y(i, j+1) - Y(i, j));
    end
    for j = 2:cols-1
        dmdp(i, j) = (Y(i, j+1) - 2*Y(i, j) + Y(i, j-1));
    end
end




% Calculate flux f using WENO
for i = 1:rows
    for j = 4:cols-3
        plus = psiWENO(dmdp(i, j-2)/h, dmdp(i, j-1)/h, dmdp(i, j)/h, dmdp(i, j+1)/h);
        minus = psiWENO(dmdp(i, j+2)/h, dmdp(i, j+1)/h, dmdp(i, j)/h, dmdp(i, j-
1)/h);
        if b(i, j) >= 0
            f(i, j) = b(i, j) * ((1/(12*h)) * (-dp(i, j-2) + 7*dp(i, j-1) + 7*dp(i,
j) - dp(i, j+1)) - plus);
        else
            f(i, j) = b(i, j) * ((1/(12*h)) * (-dp(i, j-2) + 7*dp(i, j-1) + 7*dp(i,
j) - dp(i, j+1)) + minus);
        end
    end
end


end

function [f] = calc_adYdx_WENO_2D(Y,a)
global h dp dmdp;
    [rows, cols] = size(Y);
    f = zeros(rows, cols);
    dp = zeros(rows, cols);
    dmdp = zeros(rows, cols);

    % Calculate dp and dmdp
    for j = 1:cols
        for i = 1:rows-1
            dp(i, j) = (Y(i+1, j) - Y(i, j));
        end
```

```
        for i = 2:rows-1
            dmdp(i, j) = (Y(i+1, j) - 2*Y(i, j) + Y(i-1, j));
        end
    end

    % Calculate flux f using WENO
    for j = 1:cols
        for i = 4:rows-3
            if a(i, j) >= 0
                f(i, j) = a(i, j) * ((1/(12*h)) * (-dp(i-2, j) + 7*dp(i-1, j) +
7*dp(i, j) - dp(i+1, j)) - psiWENO(dmdp(i-2, j)/h, dmdp(i-1, j)/h, dmdp(i, j)/h,
dmdp(i+1, j)/h));
            elseif a(i, j) < 0
                f(i, j) = a(i, j) * ((1/(12*h)) * (-dp(i-2, j) + 7*dp(i-1, j) +
7*dp(i, j) - dp(i+1, j)) + psiWENO(dmdp(i+2, j)/h, dmdp(i+1, j)/h, dmdp(i, j)/h,
dmdp(i-1, j)/h));
            end
        end
    end
end

function [f] = calc_adYdx_WENO(Y, a)
global h dp dmdp;
    [rows, cols] = size(Y);
    f = zeros(rows, cols);
    dp = zeros(rows, cols);
    dmdp = zeros(rows, cols);

    % Calculate dp and dmdp
    for j = 1:cols
        for i = 1:rows-1
            dp(i, j) = (Y(i+1, j) - Y(i, j));
        end
        for i = 2:rows-1
            dmdp(i, j) = (Y(i+1, j) - 2*Y(i, j) + Y(i-1, j));
        end
    end

    % Calculate flux f using WENO
    for j = 1:cols
        for i = 4:rows-3
            if a(i, j) >= 0
                f(i, j) = a(i, j) * ((1/(12*h)) * (-dp(i-2, j) + 7*dp(i-1, j) +
7*dp(i, j) - dp(i+1, j)) - psiWENO(dmdp(i-2, j)/h, dmdp(i-1, j)/h, dmdp(i, j)/h,
dmdp(i+1, j)/h));
            elseif a(i, j) < 0
                f(i, j) = a(i, j) * ((1/(12*h)) * (-dp(i-2, j) + 7*dp(i-1, j) +
7*dp(i, j) - dp(i+1, j)) + psiWENO(dmdp(i+2, j)/h, dmdp(i+1, j)/h, dmdp(i, j)/h,
dmdp(i-1, j)/h));
            end
        end
    end
end

function [a, b, c, d] = bcNodeCN(a, b, c, d, t)
```

```
    global h;
    Tp = -33/7*cos(pi/2*t^(5/3)) + 3*pi/8*sin(5*pi/24*(t^3 - 4*t^2));
    T0 =  3/2+2/7*cos(pi/2*t^(5/3));
    % Apply the Neumann boundary condition at the left boundary (i=2)

    d(2)=d(2)-a(2)*T0;


    % Assuming the right boundary also has a Neumann condition
    % Apply the Neumann boundary condition at the right boundary (end of the vector)
    a(end-1)=a(end-1)-(1/3)*c(end-1);
    b(end-1)=b(end-1)+(4/3)*c(end-1);
    d(end-1)=d(end-1)-Tp*2*h*c(end-1)/3;


end

function [T] = bcNode(T, t)
    global h;

    % Calculate the boundary values
    T0 = 3/2 + 2/7 * cos(pi/2 * t^(5/3));
    Tp = -33/7 * cos(pi/2 * t^(5/3)) + 3*pi/8 * sin(5*pi/24 * (t^3 - 4*t^2));

    % Apply the boundary conditions
    T(1) =  T0(1); % Dirichlet boundary condition at the first cell
    T(end) = (2*h*Tp+4*T(end-1)-T(end-2))/3; % Neumann boundary condition at the last
cell
end

function phi = bcGS(phi)

s = size(phi);
M = s(1)-2;
N = s(2)-2;


%elvaluate the LHS and RHS
for j = 1:N+1
    phi(1,j) = phi(2,j); %LHS
    phi(M+2,j) = phi(M+1,j); %RHS
end

%evaluate the lower and upper boundary
for i = 1:M+1
    phi(i,1) = phi(i,2); %Lower
    phi(i,N+2) = phi(i,N+1); %upper
end

for j = 1:N+1
    phi(1,j) = phi(2,j); %LHS
    phi(M+2,j) = phi(M+1,j); %RHS
end

for i = 1:M+1
```

```
    phi(i,1) = phi(i,2); %Lower
    phi(i,N+2) = phi(i,N+1); %Upper
end

end
function [a,b,c,d]  = bcCN2_Y3(a,b,c,d,t)
global xc3 yc3;

M = size(d,1)-6;
N = size(d,2)-6;


%Top boundary
for i = 4:M+3
    j=N+3;
    if xc3(i) >= 2 && xc3(i) <= 2.5
        b(i,j) = b(i,j)-c(i,j);
    else
        b(i,j) = b(i,j)+c(i,j);
    end
end
%Bottom
for i = 4:M+3
    j=4;
    if xc3(i) >= 1.5 && xc3(i) <= 2
        b(i,j) = b(i,j)-a(i,j);
    else
        b(i,j) = b(i,j)+a(i,j);
    end
end
end
```