Indian Institute of Science
Department of Mechanical Engineering

Validation of Shear and Pressure-Driven Flows

Aswin Jayaprakash Bhat
from Indian Institute of Technology, Palakkad

Supervisor: Dr. Shubhdeep Mandal

A report submitted in fulfilment of the requirements of
Internship at Indian Institute of Science
in *Mechanical Engineering*

August 3, 2025

Declaration

I, Aswin Jayaprakash Bhat, of the Department of Mechanical Engineering, IIT Palakkad, confirm that this is my own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

I give consent to a copy of my report being shared with future students as an exemplar.

Aswin Jayaprakash Bhat
August 3, 2025

Contents

List of Figures

List of Abbreviations

CFD	Computational Fluid Dynamics
FVM	Finite Volume Method
IB	Immersed Boundary
IBM	Immersed Boundary Method
Re_p	Particle Reynolds Number
BCs	Boundary Condition(s)

Chapter 1

Introduction

Computational Fluid Dynamics (CFD) is a branch of fluid mechanics that makes use of numerical methods and algorithms to simulate fluid flow under various conditions and situations. CFD allows people to simulate and study complex fluid behavior by forcing the fluid particles to follow certain physical rules like Navier Stokes Equations. In this report I will explain how I wrote code for simulating various flows using FVM and later IBM to ultimately verify findings of the paper " Inertial bifurcation of the equilibrium position of a neutrally-buoyant circular cylinder in shear flow between parallel walls " by Andrew J. Fox, James W. Schneider, and Aditya S. Khair. According to the research paper, a neutrally bouyant cylinder of density same as that of fluid will shift its equilibrium position as the Particle Reynold number of flow changes. We had to simulate this verify the plots we get.

1.1 Basics of a CFD code

A CFD code is based upon solving Navier Stokes Equations to mimic natural flows under predefined constraints. These equations are not possible to solve analytically. Numerical methods are used to approximate these equations and simulate near perfect flow hen compared to its physical counterpart. The main aim when writing code for simulating fluid flows is to ensure physical laws are followed all the time and error is minimal.

1.2 Finite Volume Method

One of the most widely used numerical methods to solve partial differential equations in fluid dynamics and heat transfer problems. FVM is based on integral form of conservation laws. With the help of FVM one can arrive at a satisfactory solution to 2D Navier-Stokes.
FVM is done in certain order:

- Choose the governing equations.
- The domain is firstly divided into smaller control volumes or cells.
- The governing PDEs are integrated over each control volume. These PDEs could be Navier-Stokes or Heat equations
- Approximate the terms to discretise each term. Interpolation techniques are used for this.
- Apply boundary conditions on the domain

- Solve system of equations after assembling equations of all control volumes.
- perform convergence check to see rate of convergence.

1.3 Immersed Boundary Method

The Immersed Boundary Method (IBM) is a numerical technique used to simulate fluid–structure interaction problems, where flexible or rigid bodies interact with a fluid. It was originally developed by Charles Peskin and has helped significantly in field of CFD.
With the help of IBM, we no longer need to change the eulerian grid according to shape of the body, as was the case for FVM.
Steps in IBM:

- Initialize the fluid velocity field and grid and particle boundary markers.
- Interpolate fluid velocities to boundary points of particle using delta function.
- Calculate boundary force on the particle depending upon boundary conditions applied.
- Spread the force to fluid grid. This adds a body force term to Navier-Stokes equation also.
- Solve the Navier-stokes equations now with help of projection method.
- Move each lagrangian point with the interpolated fluid velocity.

Since its discovery, Peskin's original IBM now has been applied with various techniques like IBM with Lattice Boltzmann Method or IBM with FVM.

1.4 Tasks alloted

We were first supposed to simulate following cases and compare our numerical results to analyical equations available:

1. Steady-state heat conduction in 2D – Plot the temperature distribution and compare with the analytical solution.
2. Simple shear flow between parallel plates – Plot the velocity profile at different time steps and compare it with the analytical solution.
3. Plane Poiseuille flow – Plot the velocity profile over time and validate with the analytical result.
4. Flow over a cylinder – Compute the drag and lift forces, and compare your results with available reference data.
5. Oscillating shear flow – Model the oscillating shear flow and analyze the velocity profile variation over time.
6. Pulsating flow – Simulate pulsating flow conditions and compare the results with theoretical predictions.

   We were supposed to simulate the above cases in FVM and later in IBM also.

   Finally we were supposed to reproduce the conditions in given research paper to simulate and verify the results of it.

1.5 Research Paper validation

After verifying our code for above cases with analytical solutions, we were supposed to use our code to confirm the findings of paper 'Inertial bifurcation of the equilibrium position of a neutrally-buoyant circular cylinder in shear flow between parallel walls'.
We had to confirm the lift forces and motion of cylinder from the initial equilibrium position with increase in critical Re_p. The paper provides a graph comparing lift force with respect to transverse position of the cylinder. Along with this it also compares motion of cylinder under various Re_p

Chapter 2

Literature Review

2.1 CFD Video Lectures by Sandip Mazumder

According to the video lecture series by Prof. Sandip Mazumder on Computational Fluid Dynamics (CFD), we learnt about solving fluid flow problems using the Finite Volume Method (FVM). This method works by dividing the domain into small control volumes and applying conservation of mass, momentum, and energy to each one. One of the main highlight of his lectures is the use of a collocated grid system, where pressure and velocity are stored at the same grid point. This makes implementation simpler, but also creates problems like checkerboarding, where pressure oscillates in an unphysical way. Prof. Mazumder explains how to fix this using Rhie-Chow interpolation, which helps couple pressure and velocity correctly. He also covers topics like time discretization, upwind and central schemes for convection, and the SIMPLE algorithm for pressure correction. These lectures are useful for understanding how to build a CFD solver from the scratch.

2.2 CFD Video Lectures by Tony Saad

In the CFD video lectures by Prof. Tony Saad, we explore more practical ways to set up and solve CFD problems. His lectures focus on building the solution step-by-step, starting from writing the differential equations and turning them into algebraic equations using finite difference or finite volume methods. He explains how to implement iterative solvers like Jacobi and Gauss-Seidel and introduces concepts like residuals and convergence checking. A unique part of Prof. Saad's lectures is his use of Coding to demonstrate how to implement the methods. This makes it easier to follow and test the concepts in real simulations. He also introduces the method of manufactured solutions, which helps verify if a code is working correctly. These lectures are useful for both understanding theory and applying it in code.

2.3 Immersed Boundary Method by Charles S. Peskin

We also study the Immersed Boundary Method (IBM) through the lectures by Prof. Charles S. Peskin, who originally developed this method to model blood flow in the heart. In IBM, the fluid is solved on a regular Eulerian grid, while the object (like a particle, boundary, or elastic membrane) is represented by Lagrangian points. These Lagrangian points are used to apply forces to the fluid and also move with the fluid velocity. This is done using discrete delta functions, which help transfer information between the Eulerian grid and Lagrangian points. IBM allows us to simulate flow around complex or moving objects without needing to generate a special mesh around the object. This makes it very useful for problems like flow around a swimming fish, heart valve movement, or suspended particles. Prof. Peskin's lectures explain the basic idea, mathematical formulation, and how to apply it in simulations. His work is the foundation for many modern fluid–structure interaction methods.

2.4 CFD-Video Lectures by Fluid Mechanics 101

There is also a helpful YouTube lecture series titled "CFD Video Lectures - YouTube" on the channel Fluid Mechanics 101. These videos cover many important CFD concepts including the SIMPLE and PISO algorithms, collocated grid (Rhie-Chow interpolation), and detailed explanation of Finite Volume Method (FVM). These lectures provide helpful visualizations and code demonstrations to support the theoretical concepts. The combination of solver algorithms and practical tips makes this lecture series a great supplement to formal CFD courses.

Chapter 3

Methodology

3.1 Important equations

In order to simulate these flows, we first had to ensure required governing equations were satisfied.

3.1.1 Heat Conduction

Steady-state 2D heat conduction with no internal heat generation is governed by the Laplace equation:

$$\delta^2 T/\delta x^2 + \delta^2 T/\delta y^2 = 0$$

3.1.2 Incompressible Flow

For most of the case we use Navier-Stokes Equation in 1D or 2D:

$$\delta u/\delta t + u.\nabla u = -\nabla p + \nu \nabla^2 u$$
$$\nabla . u = 0$$

3.2 Steps in coding

3.2.1 Discretisation and Implementation Schemes

In this section we will discuss how grids were initialised for the domain and how time steps were calculated for the cases mentioned before.

1. Diffusion Term
   The diffusion term was discretised using a second-order central difference scheme. This was applied to velocity components during momentum update step.

   $$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1,j} - 2 * u_{i,j} + u_{i-1,j}}{\Delta x^2}$$

2. Advection Term
   For advection term, first order upwind scheme was used in order to ensure numerical stability for high Re_p cases. Example of upwind scheme:

   $$\frac{\partial u^2}{\partial x} \approx \frac{u_{i,j}^2 - u_{i-1,j}^2}{\Delta x}$$

3. Projection Method
   A pressure projection method was used to enforce incompressibility. The steps involved are:

   (a) Firstly we calcuate the intermediate velocities.

   (b) Then we use these velocities in pressure poission equation:

   $\nabla^2 p = \frac{\rho}{\Delta t} \nabla . u_n$
   This equation was solved with either SOR or Multigrid method.

   (c) finally we get the corrected velocity field from:

   $u^{n+1} = u^n - \frac{\Delta t}{\rho} \frac{\partial p}{\partial x}$

4. Collocated Grid
   While defining the collocated grid, all variables like pressure and velocity were stored at the center of cells. This makes the implementation easier than a staggered grid but leads to artificial pressure oscillations or checkerboarding. To deal with it we used Rhie-Chow interpolation during the pressure correction step to couple velocity and pressure correctly and prevent decoupling. In Rhie-Chow interpolation, velocities at cell faces are adjusted by term of pressure gradient and time step.

5. IB scheme
   For applying the Immersed Boundary scheme, Peskin's IBM framework was applied. The IB points were distributed along a line or cylinder depending upon the case. Then fluid velocities were interpolated from collocated grid to the IB points using 4-point discrete delta kernel. After applying the restoring force, force was spread back to the grid again using the 4-point discrete delta kernel. While solving the Navier-Stokes equation, force term was added finally.

3.2.2 FVM

For all FVM codes, we first defined a domain and discretised it into smaller cells. We utilised collocated grid and thus stored all values at cell centers. To prevent Checkerboard oscillations we are also using Rhie-Chow Interpolation. Afterwards we applied Navier-stokes (or Heat equation for case 1) in each cell. Boundary conditions were implemented accordingly:

1. Steady-state heat conduction in 2D : Dirichlet BCs were applied on all four walls of rectangular system.

2. Simple shear flow between parallel plates: Top wall moves at user given velocity(10 for now) with bottom wall also moving with user defined velocity (stationary here) while Neumann boundary condition is imposed on left and right walls.

3. Plane Poiseuille flow: Neumann BC is applied on left and right wall along flow direction.

4. Oscillating shear flow: Bottom wall is again stationary here but top wall is constantly changing direction.

5. Pulsating flow: no slip BCs are applied.

Iterative solvers like Successive Over-Relaxation(SOR) or Multigrid method (with V-Cycle and SOR or Gauss-Siedel smoothers) are used to compute numerical solution and error analysis is done by comparing numerical and analytical values. Relative error is plotted, showing convergence. Time step is calculated based on stability criteria (diffusive stability limit and convective limit).

3.2.3 IBM

Here we simulate flow using the IBM and a pressure-corrected projection method. We also rely on multigrid method (V-cycle) for solving the pressure Poisson equation and Navier-stokes equation. Boundary conditions similar to FVM code are applied here.

Except for 'Flow over a cylinder' code, in all cases we will be using a line of immersed boundary markers in center of stream. For this case we implement the immersed boundary conditions on cylinder's boundary.

We initialise with applying BCs according to case and interpolate to immersed boundary. Afterwards spreading of forces is done and divergence is computed. Then we implement iterative solver like Multigrid method and then enforce $\nabla . u = 0$. We finally plot relative L2 error after comparing with analytical solution.

3.3 Verifying Research Paper

According to the paper, we should observe a lift force on the neutrally buoyant cylinder for Re_p more than critical Reynold number. To cross-check this, we firstly implemented IBM on boundary of fixed cylinder to compute lift forces acting on it. Later we also allowed free motion of cylinder to observe the migration.

Uniform grid with equal spacing as considered. The cylinder considered has a confinement ratio κ of 0.125. IB points were uniformly spread on circle while couette flow was initialised in the background. Fluid velocities were interpolated to IB points. Then force imposed for motion was calculated and spread to grid. With the force now available, we updated the velocity and repeat until steady state. We compute net lift force at the end of convergence and non-dimentionalise it as done in paper.

Chapter 4

Results

4.1 FVM code results

The following section shows the plotted results of cases mentioned previously.

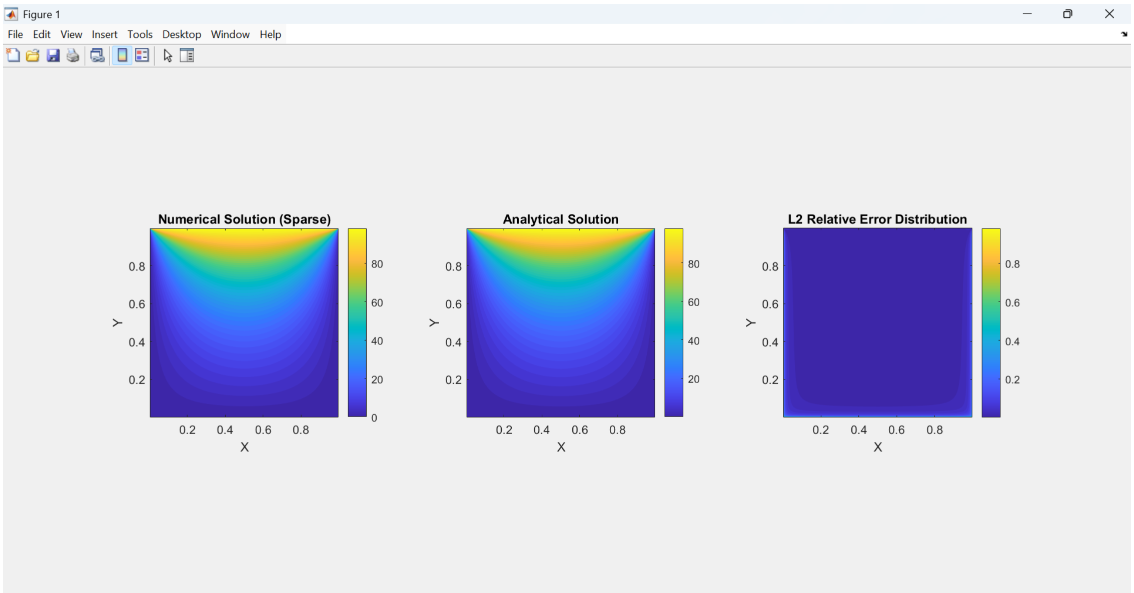

Figure 4.1: Steady-state heat conduction in 2D

As you can see from the figure, there is very minimal difference between the simulated solution with sparse solver and analytical solution.

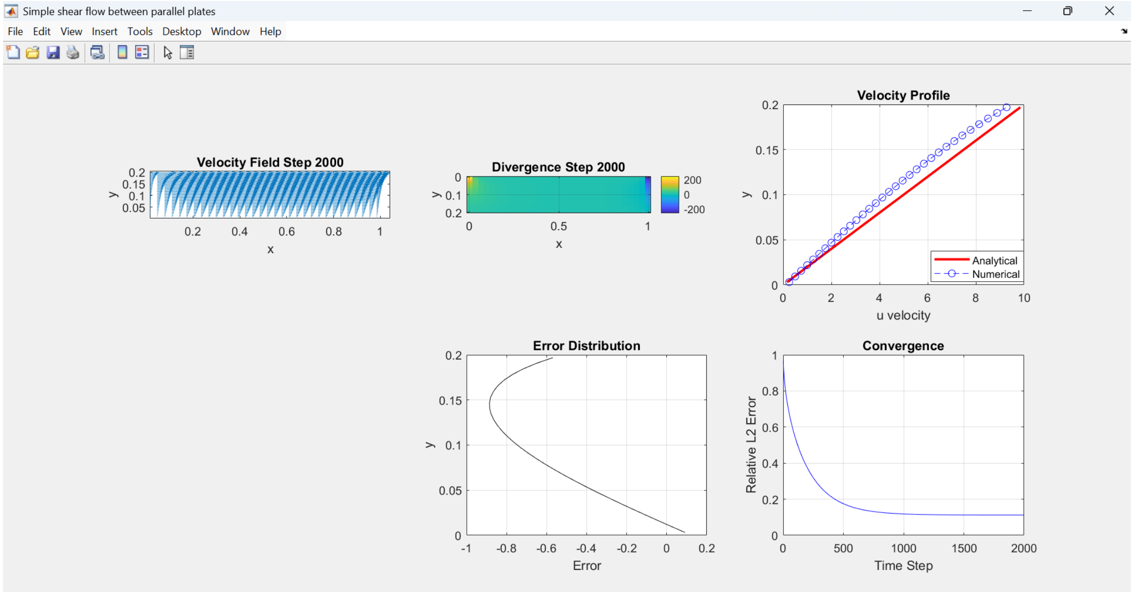

Figure 4.2: Simple shear flow between parallel plates

From here onwards we have also plotted Divergence of velocity field in order to check mass conservation laws being followed thoroughly. We can see how the solution converges and the error profile at certain time step also. Velocity profile and Velocity field are also plotted for easy visualisation.

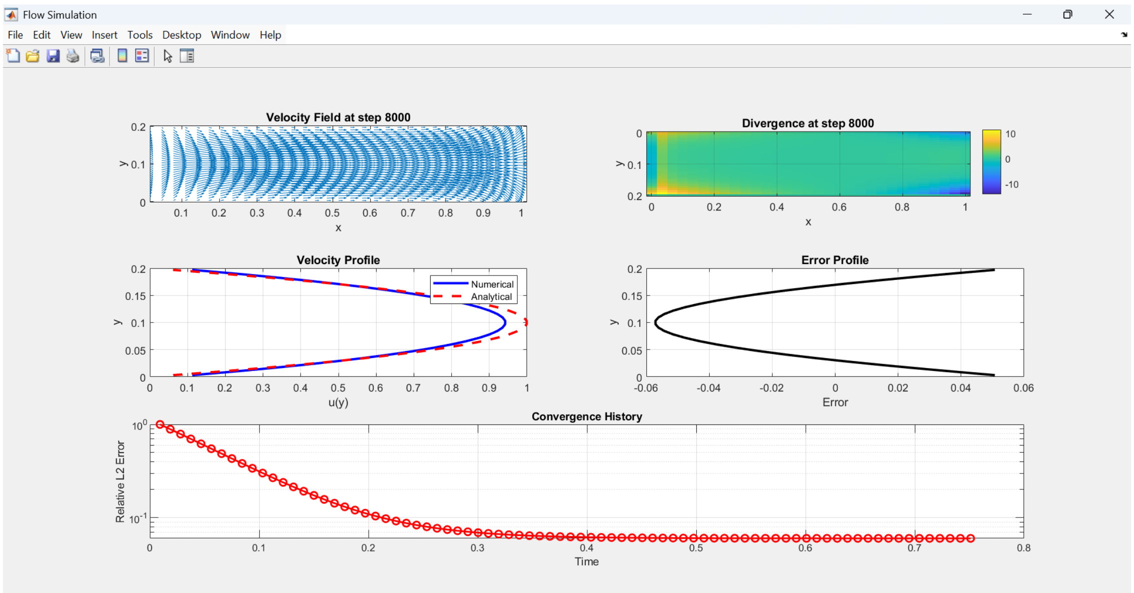

Figure 4.3: Plane Poiseuille flow

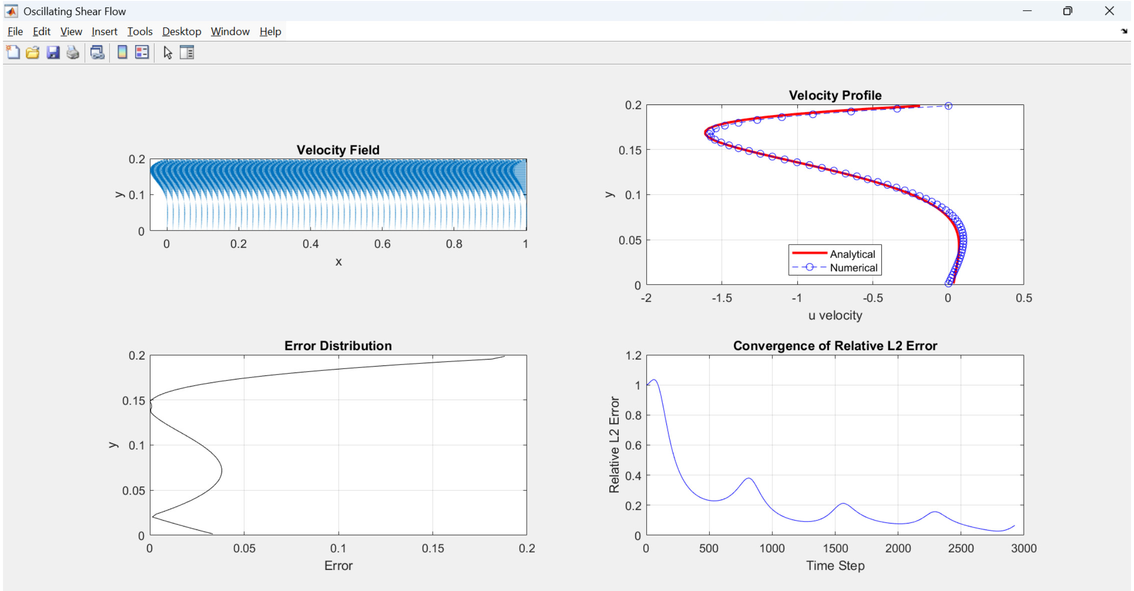

Figure 4.4: Oscillating shear flow

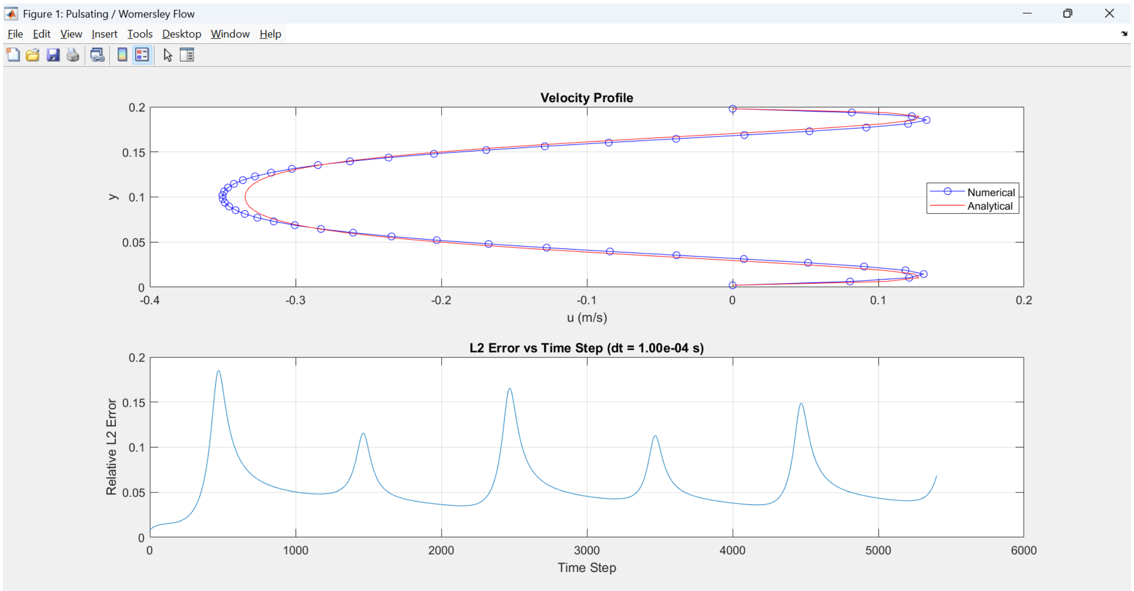

Figure 4.5: Pulsating flow

4.2 IBM code results

The same cases where simulated using IBM. This section shows results of implementing IBM in simulation. We can compare with FVM results for same cases and notice easily that IBM shows better results within less time steps.

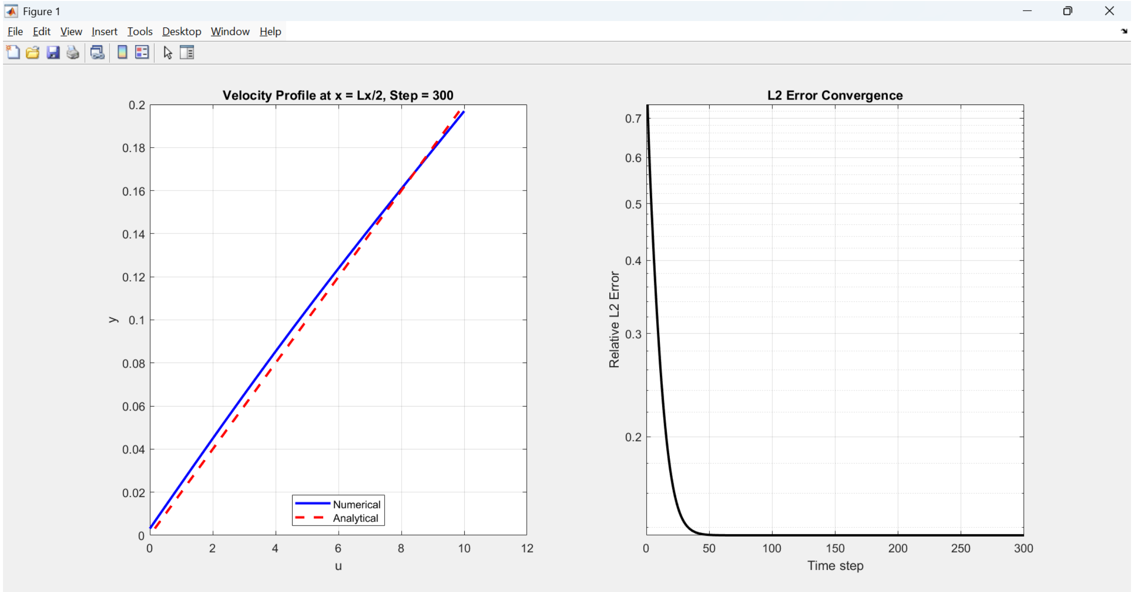

Figure 4.6: Simple shear flow between parallel plates using IBM

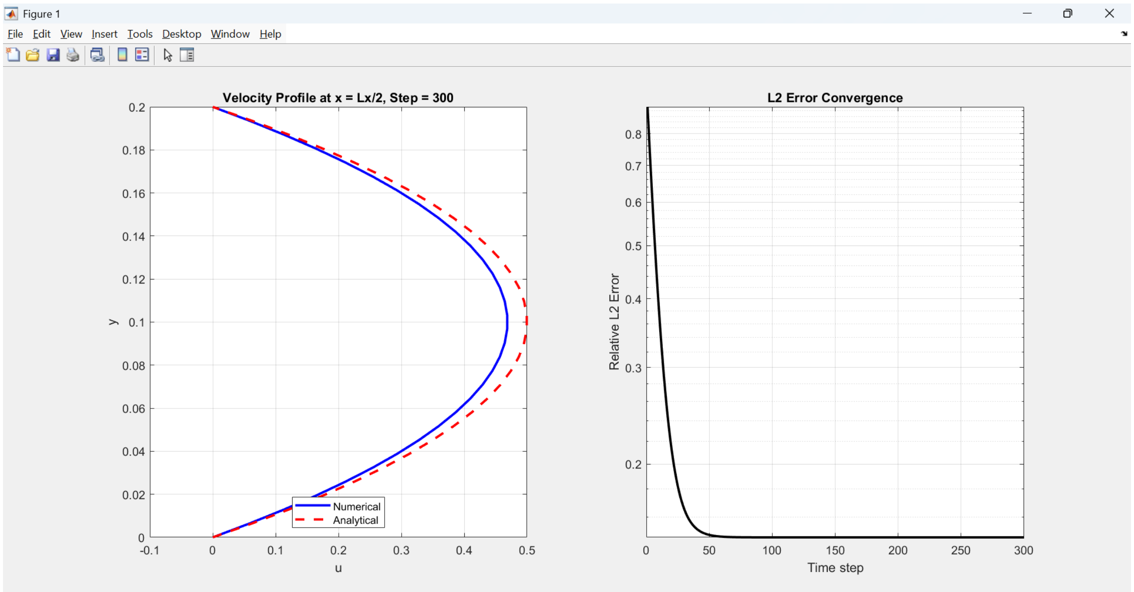

Figure 4.7: Plane Poiseuille flow in IBM

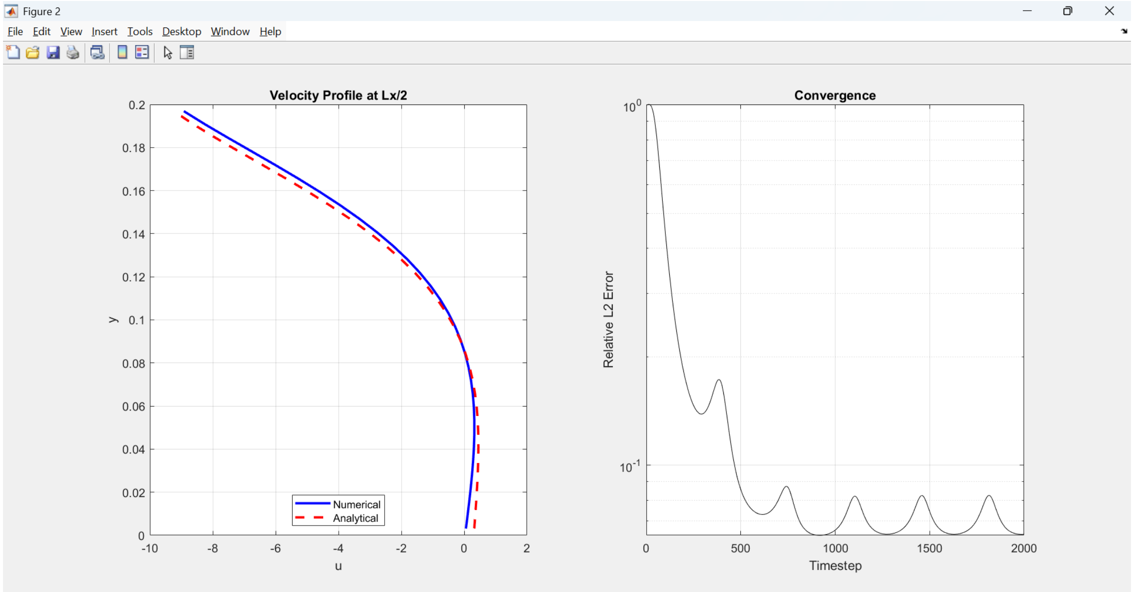

Figure 4.8: Oscillating shear flow in IBM

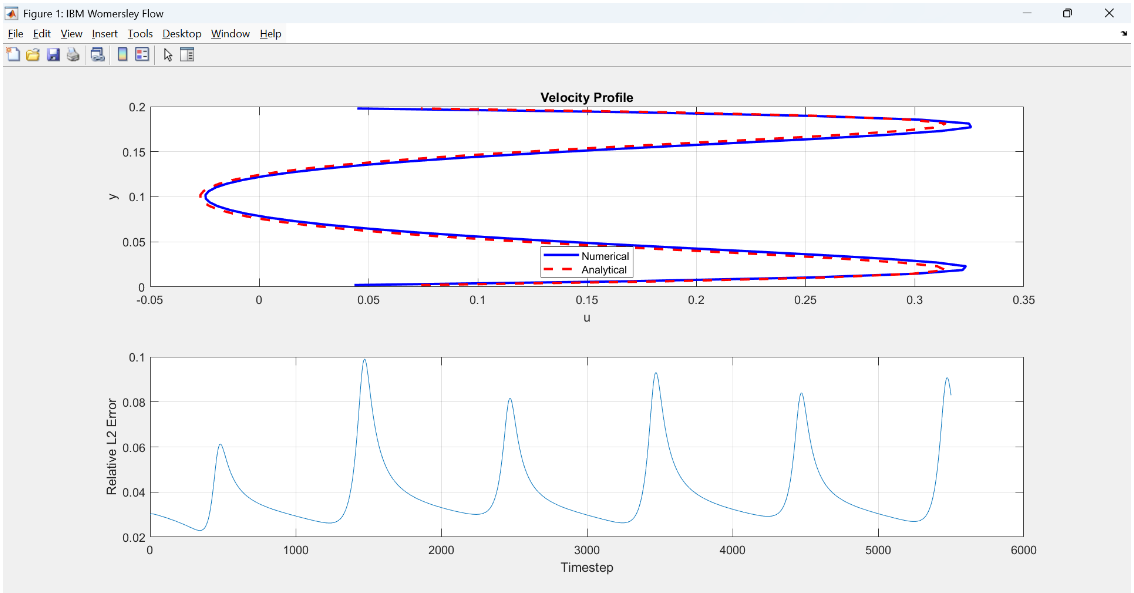

Figure 4.9: Pulsating flow in IBM

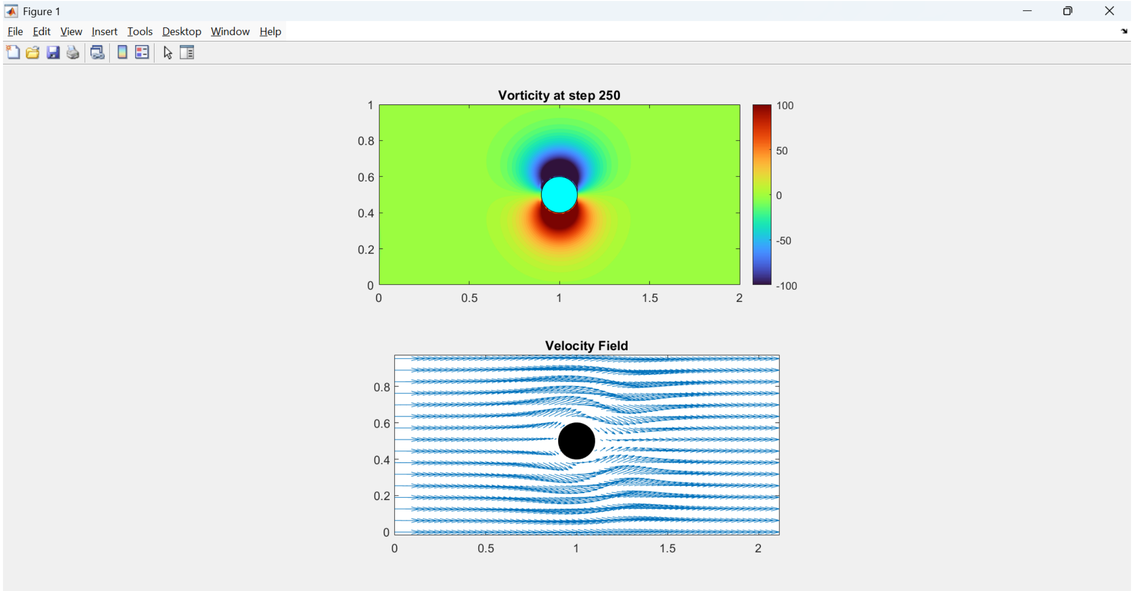

Figure 4.10: Flow over a cylinder

// Here velocity field and vorticity of flow around cylinder. These plots are for Re_p lesser than critical value.

4.3 Verifying Paper

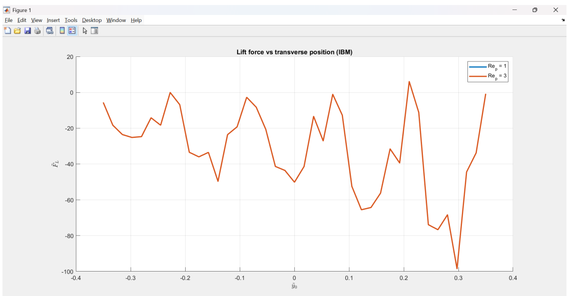

Figure 4.11: Lift force of cylinder v/s transverse position

Compared to the plots in the research paper 'Inertial bifurcation of the equilibrium position of a neutrally-buoyant circular cylinder in shear flow between parallel walls', we receive significant deviation in plot. This implies that our code is not correct and needs more improvement.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this study, we successfully implemented a 2D incompressible Navier-Stokes solver using FVM, and later enhanced it with IBM to simulate flow–structure interactions. Successfully validated the solver for a variety of canonical flows:

- Simple shear flow between parallel plates
- Poiseuille pressure-driven flow
- Oscillatory shear flow
- Pulsatile flow

We verified that both the FVM and IBM frameworks reproduced analytical solutions in these cases with satisfactory tolerances.

We also simulated flow around a stationary immersed cylinder and visualised the velocity field and vorticity patterns.

We attempted to reproduce the results of paper "Inertial bifurcation of the equilibrium position of a neutrally-buoyant circular cylinder in shear flow between parallel walls" (Phys. Rev. Research 2, 013009 (2020)) but despite using IBM solver we were unable to do so.

References

- CFD Video Lectures by Sandip Mazumder
  https://www.youtube.com/@sandipmazumder171/playlists
- CFD Video Lectures by Tony Saad
  https://www.youtube.com/playlist?list=PLEaL16Sf-KICvBLrYFwt5h_LgedJyN59n
- CFD Video Lectures by Fluid Mechanics 101 https://www.youtube.com/@fluidmechanics101/playlists
- A guide to writing your first CFD solver
  https://www.montana.edu/mowkes/research/source-codes/GuideToCFD.pdf
- Immersed Boundary Method by Charles S. Peskin
  https://math.nyu.edu/~peskin/ib_lecture_notes/index.html
- Inertial bifurcation of the equilibrium position of a neutrally-buoyant circular cylinder in shear flow between parallel walls
  https://journals.aps.org/prresearch/abstract/10.1103/PhysRevResearch.2.013009

Appendix A

Code

A.1 FVM codes

A.1.1 Steady-state heat conduction in 2D

```
clc; clear; close all;
% Note:Code works only for temperatures applied to top or
   bottom wall. we have to change analytical solution if
   different BCs is applied

% Domain and Grid parameters
Lx = 1;
Ly = 1;
Nx = 400;
Ny = 400;
dx = Lx/Nx;
dy = Ly/Ny;

x = linspace(dx/2, Lx-dx/2, Nx);
y = linspace(dy/2, Ly-dy/2, Ny);
[X, Y] = meshgrid(x, y);

% Boundary Conditions (change according to user input)
BC.T_top = 100;
BC.T_bottom = 0;
BC.T_left = 0;
BC.T_right = 0;

% Source Term (Poisson Equation)
% Set f = 0 for Laplace
f = zeros(Ny, Nx);

% Solver Options
solver_type = 'Sparse';    % SOR or Sparse_solver

% Solver Parameters
omega = 2 / (1 + sin(pi/max(Nx, Ny))); % Optimal omega
```

```
max_iter = 10000;
tol = 1e-12;

% Solve
if strcmpi(solver_type, 'SOR')
    [numerical_temp, iteration_count, error_history] =
   SOR_solver(f, BC, dx, dy, omega, max_iter, tol);
elseif strcmpi(solver_type, 'Sparse')
    numerical_temp = Sparse_solver(f, BC, dx, dy);
    error_history = [];
    iteration_count = 1;
else
    error('Unknown solver type. Choose "SOR" or "Sparse".');
end


% Analytical Solution
analytical_temp = zeros(Ny, Nx);
N_terms = 100; % Number of sine terms

for n = 1:2:(2*N_terms-1) % Only odd terms
    lambda= n*pi;
    term = (4*BC.T_top)/(n*pi) *sinh(n*pi*Y) ./ sinh(n*pi*Ly)
    .*sin(n*pi*X);
    analytical_temp =analytical_temp  + term;
end

%Error Calc

error_abs= abs(numerical_temp- analytical_temp);
L2_error =sqrt(sum(error_abs(:).^2)) /sqrt(sum(
   analytical_temp(:).^2));

fprintf('L2 Relative Error = %.6e\n', L2_error);

% Plotting (Includes error plot)

figure('Position',[100 100 1200 400]);

subplot(1,3,1);
contourf(X, Y, numerical_temp, 50, 'LineColor','none');
colorbar;
title(['Numerical Solution (' solver_type ')']);
xlabel('X');
ylabel('Y');
axis equal tight;

subplot(1,3,2);
contourf(X, Y, analytical_temp, 50, 'LineColor','none');
```

```
colorbar;
title('Analytical Solution');
xlabel('X');
ylabel('Y');
axis equal tight;

subplot(1,3,3);
contourf(X, Y, error_abs ./ abs(analytical_temp + eps), 50, '
    LineColor','none');
%+eps to avoid divide by zero
colorbar;
title('L2 Relative Error Distribution');
xlabel('X');
ylabel('Y');
axis equal tight;


% Plot Convergence History
if strcmpi(solver_type, 'SOR')
    figure;
    semilogy(error_history,'-o');
    grid on;
    xlabel('Iteration');
    ylabel('Max Error');
    title('SOR Convergence History');
end



% Helper Functions

% Dirichlet BCs (for neumann different analytical solution)
function T = apply_BC(T, BC)
    T(1,:)   = BC.T_top;
    T(end,:) = BC.T_bottom;
    T(:,1)   = BC.T_left;
    T(:,end) = BC.T_right;
end

% SOR Solver
function [T, iter, error_history] = SOR_solver(f, BC, dx, dy,
    omega, max_iter, tol)
    [Ny, Nx] = size(f);
    T = zeros(Ny, Nx);
    T = apply_BC(T, BC);

    dx2 = dx^2;
    dy2 = dy^2;
    coeff = 1/(2*(dx2 + dy2));
```

```

    error_history = [];

    for iter = 1:max_iter
        T_old = T;
        T(2:end-1,2:end-1) = (1-omega)*T(2:end-1,2:end-1) + omega *coeff *((T(2:end-1,3:end)+ T(2:end-1,1:end-2))*dy2 +(T(3:end,2:end-1)+T(1:end-2,2:end-1))*dx2 -f(2:end-1,2:end-1)*dx2*dy2 );
        T = apply_BC(T, BC);

        % Error Check
        err = max(max(abs(T - T_old)));
        error_history = [error_history; err];

        if err < tol
            fprintf('SOR converged in %d iterations with error %.3e\n', iter, err);
            break
        end
    end

    if iter == max_iter
        fprintf('SOR reached max iterations (%d) with error %.3e\n', iter, err);
    end
end

%Sparse Matrix Solver
function T = Sparse_solver(f, BC, dx, dy)
    [Ny, Nx] = size(f);
    N = Ny * Nx;
    dx2 = dx^2;
    dy2 = dy^2;

    % Sparse Matrix Assembly
    main_diag = -2*(1/dx2 + 1/dy2) * ones(N,1);
    off_diag_x = 1/dx2 * ones(N,1);
    off_diag_y = 1/dy2 * ones(N,1);

    A = spdiags([off_diag_y, off_diag_x, main_diag, off_diag_x, off_diag_y], [-Nx,-1,0,1,Nx], N, N);
    b = -reshape(f,[],1);

    % applying Dirichlet BCs
    top_idx    = (Ny-1)*Nx + (1:Nx);
    bottom_idx = (0)*Nx + (1:Nx);
    left_idx   = ((Ny-1):-1:0)*Nx + 1;
    right_idx  = ((Ny-1):-1:0)*Nx + Nx;
```

```
    A(top_idx,:)    = 0; A(sub2ind(size(A), top_idx, top_idx)
   ) = 1; b(top_idx) = BC.T_top;
    A(bottom_idx,:) = 0; A(sub2ind(size(A), bottom_idx,
   bottom_idx)) = 1; b(bottom_idx) = BC.T_bottom;
    A(left_idx,:)   = 0; A(sub2ind(size(A), left_idx,
   left_idx)) = 1; b(left_idx) = BC.T_left;
    A(right_idx,:)  = 0; A(sub2ind(size(A), right_idx,
   right_idx)) = 1; b(right_idx) = BC.T_right;

    T_vec = A\b;

    % Reshape to 2D
    T = reshape(T_vec, [Nx, Ny])';
end
```

A.1.2 Simple shear flow between parallel plates

```
clc; clear;close all;

% Domain and grid setup
x= 32;
y = 32;
Lx =1;
Ly = 0.2;
dx = Lx/x;
dy =Ly / y;
visc = 0.1;

% Boundary conditions
Ut = 10; % Top wall velocity
Ub = 0;  % Bottom wall velocity

% Time step based on stability
CFL =0.3;
u_max = max(abs([Ut, Ub]));
dt1 =1e6;
dt2 = CFL * min(dx, dy) / u_max;
dt = min(dt1, dt2)
% dt = 0.000025; % chosen small for stability

% Preallocation
p  = zeros(y+2, x+2);
u = zeros(y+2, x+2);
v  =zeros(y+2, x+2);
ut =zeros(y+2, x+2);
vt= zeros(y+2, x+2);
divut =zeros(y+2, x+2);

```

```
% Analytical solution for simple shear flow (linear profile)
ya = linspace(dy/2,Ly -dy/2,y);
ua = (Ut -Ub)/Ly* ya +Ub;


%% Mesh for plotting
[X, Y] = meshgrid(dx/2:dx:Lx - dx/2, dy/2:dy:Ly - dy/2);

%% Setup figure with subplots
figure('Name', 'Simple shear flow between parallel plates', '
    NumberTitle', 'off');

subplot(2,3,1);
hQuiver = quiver(X, Y, zeros(size(X)), zeros(size(Y)), '
    AutoScaleFactor', 3);
title('Velocity Field');
xlabel('x'); ylabel('y'); axis equal tight;

subplot(2,3,2);
hIm = imagesc(linspace(0,Lx,x), linspace(0,Ly,y), zeros(y,x))
    ;
colorbar;
title('Divergence');
xlabel('x'); ylabel('y');
axis equal tight; grid on;

subplot(2,3,3);
hProfile = plot(ua, ya, 'r-', 'LineWidth', 2); hold on;
hNumerical = plot(ua*0, ya, 'bo--');
title('Velocity Profile');
xlabel('u velocity'); ylabel('y');
legend('Analytical','Numerical');
grid on;

subplot(2,3,5);
hError = plot(zeros(y,1), ya, 'k');
title('Error Distribution');
xlabel('Error'); ylabel('y');
grid on;

subplot(2,3,6);
hConv = plot(0, 0, 'b');
title('Convergence of Relative L2 Error');
xlabel('Time Step'); ylabel('Relative L2 Error');
grid on;
```

```
% %% Setup video writer
% k = VideoWriter('CouetteFlowSimulation.mp4', 'MPEG-4'); % Name of video file
% k.FrameRate = 10; % Frames per second
% open(k);



%% Time-stepping loop
tsteps =2000;
L2_Err_hist = zeros(tsteps, 1);

for n = 1:tsteps

    [u, v,~] =apply_bcs(u, v, Ut, Ub,p);

    %  X momentum
    for i = 3:x+1
        for j = 2:y+1
            ue = 0.5 * (u(j, i+1) + u(j, i));
            uw = 0.5 * (u(j, i) + u(j, i-1));
            un = 0.5 * (u(j+1, i) + u(j, i));
            us = 0.5 * (u(j, i) + u(j-1, i));
            vn = 0.5 * (v(j+1, i-1) + v(j+1, i));
            vs = 0.5 * (v(j, i-1) + v(j, i));

            if n > 50
    convection = 0;
else
    convection = -(ue^2 - uw^2)/dx - (un*vn - us*vs)/dy;
end

            diffusion = visc * ((u(j, i-1) - 2*u(j, i) + u(j, i+1))/dx^2 + ...
                                (u(j-1, i) - 2*u(j, i) + u(j+1, i))/dy^2);

            ut(j, i) = u(j, i) + dt * (convection + diffusion);
        end
    end

    %  Y momentum
    for i = 2:x+1
        for j = 3:y+1
            ve = 0.5 * (v(j, i+1) + v(j, i));
            vw = 0.5 * (v(j, i) + v(j, i-1));
            ue = 0.5 * (u(j, i+1) + u(j-1, i+1));
            uw = 0.5 * (u(j, i) + u(j-1, i));
```

```
            vn = 0.5 * (v(j+1, i) + v(j, i));
            vs = 0.5 * (v(j, i) + v(j-1, i));

            if n > 50
    convection = 0;
else
    convection = -(ue*ve - uw*vw)/dx - (vn^2 - vs^2)/dy;
            end
            diffusion = visc * ((v(j, i+1) - 2*v(j, i) + v(j, i-1))/dx^2 + ...
                                (v(j+1, i) - 2*v(j, i) + v(j-1, i))/dy^2);
            vt(j, i) =v(j, i) + dt *(convection +diffusion);
        end
    end

    % pressre correction
    rho = 1;
    divut(2:end-1, 2:end-1) = (ut(2:end-1,3:end) - ut(2:end-1,2:end-1))/dx + ...
                              (vt(3:end,2:end-1) - vt(2:end-1,2:end-1))/dy;

    rhs = rho * divut / dt;


    [p, ~] = fmg_solver(rhs, Lx, Ly, x, y);


    [~,~,p]=apply_bcs(u, v, Ut, Ub,p);

    %corner smoothining
    u(end,end) = mean([u(end-1,end), u(end,end-1)]);
    v(end,end) = mean([v(end-1,end), v(end,end-1)]);

    % Bottom-right corner
    u(1,end) = mean([u(2,end), u(1,end-1)]);
    v(1,end) = mean([v(2,end), v(1,end-1)]);

    % Top-left corner
    u(end,1) = mean([u(end-1,1), u(end,2)]);
    v(end,1) = mean([v(end-1,1), v(end,2)]);

    % Bottom-left corner
    u(1,1) = mean([u(2,1), u(1,2)]);
    v(1,1) = mean([v(2,1), v(1,2)]);

    p(2,2)=0; % Set reference pressure point to zero to prevent accidental existence of pressure gradients by only
```

```
    enforcing neumann equations

 %velocity correction
    [u, v] = rhie_chow_correction(ut, vt, p, dx, dy, dt);

    % --- Under-relaxation ---
    alpha = 0.4;


    u= alpha* u  +(1- alpha) *ut;
    v = alpha*v + (1 -alpha)* vt;


    % to display the velocity at the geometric center of the
   cell

    uc = 0.5*(u(2:end-1,2:end-1)+ u(2:end-1,  3:end));
    vc = 0.5 *(v(2:end-1,2:end-1) +v(3:end,2:end-1));

    u_profile =mean(uc, 2); % average across x-direction

    % Error
    error_profile =u_profile -ua';
    L2_error= sqrt(sum(error_profile.^2) /length(
   error_profile)) / ...
        sqrt(sum(ua.^2) /length(ua));
    L2_Err_hist(n) = L2_error;



    if mod(n,10) == 0
        set(hQuiver, 'UData', uc, 'VData', vc);
        set(hIm, 'CData', flipud(divut(2:end-1,2:end-1)));

        set(hNumerical, 'XData', u_profile, 'YData', ya);
        set(hError, 'XData', error_profile, 'YData', ya);
        set(hConv, 'XData', 1:n, 'YData', L2_Err_hist(1:n));

        subplot(2,3,1); title(['Velocity Field Step ',
   num2str(n)]);
        subplot(2,3,2); title(['Divergence Step ', num2str(n)
   ]);
        subplot(2,3,3); title('Velocity Profile');
        subplot(2,3,5); title('Error Distribution');
        subplot(2,3,6); title('Convergence');

        drawnow;

        % %  Capture frame for video
```

```
        % frame = getframe(gcf);
        % writeVideo(k, frame);
    end

end


% %% Close video file
% close(k);
% disp('Video saved successfully as CouetteFlowSimulation.mp4');

fprintf('Max divergence: %.2e\n',max(abs(divut(:))));


function [u, v,p] = apply_bcs(u, v,Ut,Ub,p)
    % Left wall
    u(:,1) = u(:,2);
    v(:,1) = 0;

    % Right wall
    u(:,end) = u(:,end-1);
    v(:,end) = 0;

    % Top wall (Moving wall)
    u(end,:) = Ut;
    v(end,:) = 0;

    % Bottom wall (Fixed wall)
    u(1,:) = Ub;
    v(1,:) = 0;


  u(:,end) = u(:,end-1);
    v(:,end) = v(:,end-1); % convective (zero-gradient) outflow BCs on velocity

    % Apply Neumann BCs on pressure
    p(:,1)   = p(:,2);
    p(:,end) = p(:,end-1);
    p(1,:)   = p(2,:);
    p(end,:) = p(end-1,:);

end


function [p,err]=sor_solver(p, S,Lx,Ly,x, y)
    dx=Lx/x ;
    dy=Ly/y ;
```

```
    Ae = ones(y+2, x+2) / dx^2;
    Aw = ones(y+2, x+2) / dx^2;
    An = ones(y+2, x+2) / dy^2;
    As = ones(y+2, x+2) / dy^2;
    Ap=-(Ae+Aw+An+As);

    it = 0;
    err = 1e10;
    tol = 1e-8;
    maxit=1000;
    B = 1.9; % between 1 and 2

    while err > tol && it < maxit
        pk = p;
        for i =2:x+1
            for j =2:y+1
                ap = Ap(j,i); ae = Ae(j,i); aw = Aw(j,i); an = An(j,i); as = As(j,i);

                pe = p(j,i+1); pw = p(j,i-1); pn = p(j+1,i); ps = p(j-1,i);

                res = S(j,i) - (ae*pe + aw*pw + an*pn + as*ps);
                p(j,i) = B * res / ap + (1-B) * pk(j,i);
            end
        end
        u  = zeros(y+2, x+2);
        v  = zeros(y+2, x+2);
        Ut=10;
        Ub=0;
        [~,~,p]=apply_bcs(u, v, Ut, Ub,p);  % applying pressure BCs to prevent pressure drift in Neumann BCs problems
        err = norm(p(:) - pk(:), 2);
        it = it+1;
    end
end



function [p, err] = multigrid_solver(p, rhs, Lx, Ly, Nx, Ny)
    max_iter = 100;
    tol = 1e-8;
    err = 1e10;

    for iter = 1:max_iter
        p_old = p;
        p = V_cycle(p, rhs, Lx, Ly, Nx, Ny);
```

```
        res = res_computing(p, rhs, Lx, Ly, Nx, Ny);
        err = norm(res(:), 2);

        if err < tol
            break;
        end
    end
end

% V-Cycle Function
function p = V_cycle(p, rhs, Lx, Ly, Nx, Ny)
    if Nx <= 6 || Ny <= 6
        p = sor_solver_local(p, rhs, Lx, Ly, Nx, Ny, 100);
        return;
    end
    p = sor_solver_local(p, rhs, Lx, Ly, Nx, Ny, 10);
    res = res_computing(p, rhs, Lx, Ly, Nx, Ny);
    res_coarse = restrict(res);

    Nc_x = size(res_coarse,2) - 2;
    Nc_y = size(res_coarse,1) - 2;
    e_coarse = zeros(Nc_y + 2, Nc_x + 2);

    e_coarse = V_cycle(e_coarse, res_coarse, Lx, Ly, Nc_x,
   Nc_y);

    e_fine = prolong(e_coarse, Nx, Ny);
    p = p + e_fine;

    % Post-smoothing
    p = sor_solver_local(p, rhs, Lx, Ly, Nx, Ny, 7);
end

function p = sor_solver_local(p, rhs, Lx, Ly, Nx, Ny, Niter)
    if Nx < 2 || Ny < 2
        warning('SOR skipped due to small grid');
        return;
    end

    dx = Lx / Nx;
    dy = Ly / Ny;
    B = 1.9;

    for iter = 1:Niter
        p_old = p;
        for i = 2:Nx+1
            for j = 2:Ny+1
                p(j,i) = (1-B)*p(j,i) +B*0.5*((dy^2*(p(j,i+1)
    +p(j,i-1)) + dx^2*(p(j+1,i)+ p(j-1,i)) -dx^2*dy^2 * rhs(j
```

```
,i))/ (2*(dx^2 + dy^2)));
            end
        end

        % Neumann BCs
        p(:,1)   = p(:,2);
        p(:,end) = p(:,end-1);
        p(1,:)   = p(2,:);
        p(end,:) = p(end-1,:);

        if norm(p(:) - p_old(:), 2) < 1e-8
            break;
        end
    end
end




function res = res_computing(p, rhs, Lx, Ly, Nx, Ny)
    dx = Lx / Nx;
    dy = Ly / Ny;

    res = zeros(Ny+2, Nx+2);
    for i = 2:Nx+1
        for j = 2:Ny+1
            laplace = (p(j,i+1) - 2*p(j,i) + p(j,i-1)) / dx^2
    + ...
                      (p(j+1,i) - 2*p(j,i) + p(j-1,i)) / dy
    ^2;
            res(j,i) = rhs(j,i) - laplace;
        end
    end
end


function coarse = restrict(fine)
    [Nyf, Nxf] = size(fine);
    Nxc = ceil((Nxf - 2)/2);
    Nyc = ceil((Nyf - 2)/2);

    coarse = zeros(Nyc+2, Nxc+2);
    for i = 2:Nxc+1
        for j = 2:Nyc+1
            i_f = 2*(i-1);
            j_f = 2*(j-1);

            neighbors = fine(j_f-1:j_f+1, i_f-1:i_f+1);
            weights = [1 2 1; 2 4 2; 1 2 1];
```

```

            % Handle edges
            valid = ~isnan(neighbors);
            w_sum = sum(weights(valid));

            coarse(j,i) = sum(neighbors(valid) .* weights(valid), 'all') / w_sum;
        end
    end
end

% Prolongation
function fine = prolong(coarse, Nxf, Nyf)
    Nxc = size(coarse, 2) - 2;
    Nyc = size(coarse, 1) - 2;

    fine = zeros(Nyf+2, Nxf+2);

    for i = 2:Nxc+1
        for j = 2:Nyc+1
            i_f = 2 * (i - 1);
            j_f = 2 * (j - 1);

            % Safely assign values to fine grid
            if j_f <= Nyf && i_f <= Nxf
                fine(j_f,   i_f)   = fine(j_f,   i_f)   + coarse(j, i);
            end
            if j_f + 1 <= Nyf && i_f <= Nxf
                fine(j_f+1, i_f)   = fine(j_f+1, i_f)   + coarse(j, i);
            end
            if j_f <= Nyf && i_f + 1 <= Nxf
                fine(j_f,   i_f+1) = fine(j_f,   i_f+1) + coarse(j, i);
            end
            if j_f + 1 <= Nyf && i_f + 1 <= Nxf
                fine(j_f+1, i_f+1) = fine(j_f+1, i_f+1) + coarse(j, i);
            end
        end
    end

    % average overlapping contributions
    fine(2:end-1, 2:end-1) = fine(2:end-1, 2:end-1) / 4;
end


function [u_corr, v_corr] = rhie_chow_correction(ut, vt, p,
```

```
    dx, dy, dt)
    % appliyng Rhie-Chow interpolation based velocity
    correction
    [Ny, Nx] = size(p);
    u_corr = ut;
    v_corr = vt;

    %  u  correction
    u_corr(2:end-1,3:end-1) = ut(2:end-1,3:end-1) - ...
        dt * (p(2:end-1,3:end-1) - p(2:end-1,2:end-2)) / dx;

    %  vcorrection
    v_corr(3:end-1,2:end-1) = vt(3:end-1,2:end-1) - ...
        dt * (p(3:end-1,2:end-1) - p(2:end-2,2:end-1)) / dy;
end

% FMG Solver
function [p, err] = fmg_solver(rhs, Lx, Ly, Nx, Ny)
    levels = floor(log2(min(Nx, Ny))) - 1;
    levels = min(levels, 5);  % Optional hard cap to avoid
    too deep levels
    if levels < 1
        warning('FMG: Too few grid levels, using regular
    multigrid.');
        [p, err] = multigrid_solver(zeros(size(rhs)), rhs, Lx
    , Ly, Nx, Ny);
        return;
    end

    % Coarsest grid size
    Nc_x = floor(Nx / 2^(levels - 1));
    Nc_y = floor(Ny / 2^(levels - 1));

    if Nc_x < 3 || Nc_y < 3
        [p, err] = multigrid_solver(zeros(size(rhs)), rhs, Lx
    , Ly, Nx, Ny);
        return;
    end

    % Construct RHS at coarsest level
    rhs_c = rhs;
    for l = 1:(levels - 1)
        rhs_c = restrict(rhs_c);
    end
    p_c = zeros(size(rhs_c));
    [Nc_y_full, Nc_x_full] = size(rhs_c);
    p_c = sor_solver_local(p_c, rhs_c, Lx, Ly, Nc_x_full - 2,
    Nc_y_full - 2, 100);

```

```
    for l = (levels - 1):-1:0

        Nxf = floor(Nx / 2^l);
        Nyf = floor(Ny / 2^l);
        p_f = prolong(p_c, Nxf, Nyf);


        rhs_f = rhs;
        for li = 1:l
            rhs_f = restrict(rhs_f);
        end

        p_c =V_cycle(p_f, rhs_f, Lx, Ly, Nxf, Nyf);
    end

    p = p_c;

    res =res_computing(p, rhs, Lx, Ly, Nx, Ny);
    err =norm(res(:), 2);
end
```

A.1.3 Plane Poiseuille flow

```
clc; clear; close all

% grid and domain properties while using staggered grid
x=32; y=32;
Lx=1; Ly=0.2;
dx=Lx/x; dy=Ly/y;
visc=0.05;

%top and Bottom wall fixed
Ut=0; Ub=0;

% Time step selection based on stability criteria
dt1 = 0.5/(visc*(1/(dx^2) + 1/(dy^2)));
CFL = 0.5;
u_max = max(abs([Ut, Ub]));
if u_max == 0
    dt2 = Inf;
else
    dt2 = CFL * min(dx/u_max, dy/u_max);
end
dt = min(dt1, dt2);
dt = 0.25 * dt; % safety margin
dpx = 0.1; % pressure gradient

% Preallocate fields
p = zeros(y+2, x+2);
```

```
u = zeros(y+2, x+2);
v = zeros(y+2, x+2);
ut = zeros(y+2, x+2);
vt = zeros(y+2, x+2);
divut = zeros(y+2, x+2);

% Cell center velocities and meshgrid
uc= 0.5*(u(2:end-1, 2:end-1) +u(2:end-1,3:end));
vc =0.5*(v(2:end-1,2:end-1)+ v(3:end,2:end-1));
[X,Y] = meshgrid(dx/2:dx:Lx-dx/2, dy/2:dy:Ly-dy/2);

fig = figure('Name','Flow Simulation','NumberTitle','off');

subplot(3,2,1);
hQuiver = quiver(X,Y,uc,vc,'AutoScaleFactor',3);
title('Velocity Field'); xlabel('x'); ylabel('y'); axis equal tight;

% Initial divergence
for i=2:x+1
    for j=2:y+1
        divu(j,i) =(u(j,i)-u(j,i-1))/dx   +(v(j,i)-v(j-1,i))/dy;
    end
end
subplot(3,2,2);
hIm = imagesc(linspace(0,Lx,x), linspace(0,Ly,y), flipud(divu(2:end-1,2:end-1)));
colorbar; title('Divergence of Velocity'); xlabel('x'); ylabel('y'); axis equal tight; grid on;

t = 0; tsteps = 8000;
err_hist = []; t_hist = [];
% k = VideoWriter('poiselle_flow_vid.mp4', 'MPEG-4'); k.FrameRate = 20; open(k);
err_initial = NaN;

for n = 1:tsteps
    [u,v,~] = apply_boundary_conditions(u,v,p,dpx,dx);

    % X-momentum
    for i = 3:x+1
        for j = 2:y+1
            ue = 0.5*(u(j,i+1)+u(j,i));
            uw = 0.5*(u(j,i)  +u(j,i-1));
            un = 0.5*(u(j+1,i)+u(j,i));
            us = 0.5*(u(j,i)  +u(j-1,i));
            vn = 0.5*(v(j+1,i-1)+v(j+1,i));
            vs = 0.5*(v(j,i-1)  +v(j,i));
```

```
            convection = -(ue^2- uw^2)/dx-(un*vn- us*vs)/dy;
            diffusion = visc* ((u(j,i-1)-2*u(j,i)+u(j,i+1))/dx^2+ (u(j-1,i)-2*u(j,i)  +u(j+1,i))/dy^2);
            pressure_source = dpx;
            rho=0.01;
            ut(j,i) = u(j,i)+ dt*(convection+ diffusion+  (pressure_source/ rho));
        end
    end

    % Y-momentum
    for i = 2:x+1
        for j = 3:y+1
            ve = 0.5*(v(j,i+1) +v(j,i));
            vw = 0.5*(v(j,i)   +v(j,i-1));
            ue = 0.5*(u(j,i+1) +u(j-1,i+1));
            uw = 0.5*(u(j,i)   +u(j-1,i));
            vn = 0.5*(v(j+1,i) +v(j,i));
            vs = 0.5*(v(j,i)   +v(j-1,i));
            convection = - (ue*ve-uw*vw)/dx -(vn^2-vs^2)/dy;
            diffusion = visc* ((v(j,i+1)-2*v(j,i)+v(j,i-1))/dx^2+   (v(j+1,i)-2*v(j,i)+ v(j-1,i))/dy^2);
            vt(j,i) = v(j,i) + dt*(convection+diffusion);
        end
    end


    divut(2:end-1,2:end-1) = (ut(2:end-1,3:end)-ut(2:end-1,2:end-1))/dx + (vt(3:end,2:end-1)-vt(2:end-1,2:end-1))/dy;
    rhs = rho * divut / dt;

        [p,~] = sor_solver(p,rhs,Lx,Ly,x,y);


    [~,~,p] = apply_boundary_conditions(u,v,p,dpx,dx);

    % Velocity correction
    u(2:end-1,3:end-1) =ut(2:end-1,3:end-1) -dt*(p(2:end-1,3:end-1)-p(2:end-1,2:end-2))/dx;
    v(3:end-1,2:end-1)=vt(3:end-1,2:end-1) - dt*(p(3:end-1,2:end-1) -p(2:end-2,2:end-1))/dy;

    for i=2:x+1
        for j=2:y+1
            divu(j,i) =(u(j,i)-u(j,i-1))/dx +(v(j,i)-v(j-1,i))/dy;
        end
    end

```

```
    uc = 0.5*  (u(2:end-1,2:end-1) +u(2:end-1,3:end));
    vc= 0.5*(v(2:end-1,2:end-1) +v(3:end,2:end-1));

    if mod(n,100)==0 || n==0
        subplot(3,2,1);
        set(hQuiver,'UData',uc,'VData',vc);
        title(['Velocity Field at step ',num2str(n)]);

        subplot(3,2,2);
        set(hIm,'CData',flipud(divu(2:end-1,2:end-1)));
        title(['Divergence at step ',num2str(n)]);

        ya = dy/2 : dy: Ly-dy/2;
        ua = (dpx/(2*visc)) *ya .*(Ly - ya)/rho;
        x_phys = linspace(dx/2,Lx-dx/2, x);
        [~, mid_idx] = min(abs(x_phys - Lx/2));
        u_num = uc(:,mid_idx);

        subplot(3,2,3);
        plot(u_num', ya, 'b-', ua, ya, 'r--', 'LineWidth',2);
        ylabel('y'); xlabel('u(y)'); legend('Numerical','Analytical');
        title('Velocity Profile'); grid on;

        err_profile = u_num - ua';
        subplot(3,2,4);
        plot(err_profile, ya, 'k-', 'LineWidth', 2);
        ylabel('y'); xlabel('Error'); title('Error Profile'); grid on;

        L2_err = norm(err_profile, 2);
        if isnan(err_initial)
            err_initial =L2_err;
        end
        err_rel = L2_err /err_initial;
        err_hist(end+1) = err_rel;
        t_hist(end+1) = t;

        subplot(3,2,[5 6]);
        semilogy(t_hist, err_hist, 'r-o', 'LineWidth', 1.5);
        xlabel('Time');
        ylabel('Relative L2 Error');
        title('Convergence History'); grid on;

        drawnow;
        % frame = getframe(gcf);
        % writeVideo(k, frame);
    end
    t = t + dt;
```

```
end

% close(k);
% disp('Video saved successfully');

figure('Name','Final Convergence','NumberTitle','off');
semilogy(t_hist, err_hist, 'r-o', 'LineWidth', 1.5);
xlabel('Time'); ylabel('Relative L2 Norm of Error');
title('Final Convergence of Numerical Solution'); grid on;

fprintf('Max divergence: %.2e\n', max(abs(divu(:))));

function [u, v, p] = apply_boundary_conditions(u, v, p,dpx,dx
)
    % Left wall
    u(:,1) =u(:,2);
    v(:,1) =0;

    % Right wall
    u(:,end) =u(:,end-1);
    v(:,end) =0;

    % Top wall
    u(end,:) =0;
    v(end,:) =0;

    % Bottom wall
    u(1,:) =0;
    v(1,:) =0;

    % non zero-gradient boundary condition for pressure at
    boundaries
    p(:,1)      = p(:,2)-dpx*dx;         % left
    p(:,end)    = p(:,end-1)+dpx*dx;     % right

end



% poission solver SOR
function [p,err]=sor_solver(p, S,Lx,Ly,x, y)
    dx=Lx/x ;
    dy=Ly/y ;
    Ae = ones(y+2, x+2) / dx^2;
    Aw = ones(y+2, x+2) / dx^2;
    An = ones(y+2, x+2) / dy^2;
    As = ones(y+2, x+2) / dy^2;
    Ap=-(Ae+Aw+An+As);

```

```
    it = 0;
    err = 1e10;
    tol = 1e-12;
    maxit=20000;
    B = 1.9; % between 1 and 2

    while err > tol && it < maxit
        pk = p;
        for i =2:x+1
            for j =2:y+1
                ap = Ap(j,i); ae = Ae(j,i); aw = Aw(j,i); an = An(j,i); as = As(j,i);
                pe = p(j,i+1); pw = p(j,i-1); pn = p(j+1,i); ps = p(j-1,i);
                res = S(j,i) - (ae*pe + aw*pw + an*pn + as*ps);
                p(j,i) = B * res/ap + (1-B)*pk(j,i);
            end
        end
        err = norm(p(:)-pk(:),2);
        it = it+1;
    end
end
```

A.1.4 Oscillating shear flow

```
% Oscillating Shear Flow Simulation with MG, Rhie-Chow
clc; clear; close all;

%% Parameters
Nx =64;
Ny=64;
Lx = 1;
Ly =.2;
dx = Lx/Nx;
dy = Ly/Ny;
x =linspace(0,Lx, Nx);
y =linspace(0,Ly, Ny);
[X, Y] =meshgrid(x, y);

nu = 0.05;              % Kinematic viscosity
U0 = 5;                % Wall velocity amplitude
f = 10;
omega = 2*pi*f;
rho =1;

% Time settings
CFL = 0.35;
dt = CFL*min(dx, dy)^2/nu;
```

```
Tf= 2/f;
Nt = ceil(Tf/dt);
dt = Tf/Nt;
time = linspace(0,Tf,Nt);


u = zeros(Ny,Nx);
v = zeros(Ny,Nx);
p = zeros(Ny,Nx);
relL2 = zeros(1, Nt);

alpha = sqrt(omega/(2*nu));
yc = linspace(dy/2,Ly -dy/2,Ny)';
% Figure
figure('Name', 'Oscillating Shear Flow', 'NumberTitle', 'off');

subplot(2,2,1);
hQuiver = quiver(X, Y, zeros(size(X)), zeros(size(Y)), 'AutoScaleFactor', 3);
title('Velocity Field'); xlabel('x'); ylabel('y'); axis equal tight;


subplot(2,2,2);
hAnalytical = plot(zeros(Ny,1), yc, 'r-', 'LineWidth', 2); hold on;
hNumerical = plot(zeros(Ny,1), yc, 'bo--');
title('Velocity Profile'); xlabel('u velocity'); ylabel('y'); legend('Analytical', 'Numerical','Location','south'); grid on;

subplot(2,2,3);
hError = plot(zeros(Ny,1), yc, 'k');
title('Error Distribution'); xlabel('Error'); ylabel('y'); grid on;

subplot(2,2,4);
hConv = plot(0, 0, 'b');
title('Convergence of Relative L2 Error'); xlabel('Time Step'); ylabel('Relative L2 Error'); grid on;



% %% Setup video writer
% k = VideoWriter('CouetteFlowSimulation.mp4', 'MPEG-4'); % Name of video file
% k.FrameRate = 10; % Frames per second
% open(k);
```

```
%% Time loop
for n = 1:Nt
    t = time(n);


    u(1,:) = 0;
    u(end,:) = U0 * sin(omega * t);
    v([1 end], :) = 0;
    v(:, [1 end]) = 0;

    [u_star, v_star] = explicit_predictor(u, v, p, rho, nu, dt, dx, dy);

    rhs = divergence(u_star, v_star, dx, dy) / dt;


    p_corr = poisson_solver(rhs, dx, dy);

    [u, v] = rhie_chow_projection(u_star, v_star, p_corr, rho, dt, dx, dy);
    p = p + p_corr;


    u_analytical = U0 * exp(-alpha * flip(yc'))' .* sin(omega*t - alpha * flip(yc'))';
    u_center = u(:, round(Nx/2));
    err = abs(u_center - u_analytical);
    relL2(n) = norm(err) / norm(u_analytical);

    if mod(n, 10) == 0 || n == 1 || n == Nt
        set(hQuiver, 'UData', u, 'VData', v);
        set(hAnalytical, 'YData', yc, 'XData', u_analytical);
        set(hNumerical, 'YData', yc, 'XData', u_center);
        set(hError, 'XData', err, 'YData', yc);
        set(hConv, 'XData', 1:n, 'YData', relL2(1:n));
        drawnow;

        % % - Capture frame for video -
        % frame = getframe(gcf);
        % writeVideo(k, frame);
    end
end



% %% Close video file
% close(k);
% disp('Video saved successfully as CouetteFlowSimulation.mp4
```

```
    ');

% Helper Functions
function [u_corr, v_corr] = rhie_chow_projection(u_star,
    v_star, p_corr, rho, dt, dx, dy)
     [Ny, Nx] = size(u_star);
     u_corr = u_star;
     v_corr = v_star;

     dpdx = zeros(Ny, Nx);
     dpdy = zeros(Ny, Nx);

     dpdx(:,2:Nx-1) = (p_corr(:,3:Nx) - p_corr(:,1:Nx-2)) /
    (2*dx);
     dpdy(2:Ny-1,:) = (p_corr(3:Ny,:) - p_corr(1:Ny-2,:)) /
    (2*dy);

     u_corr = u_star - dt / rho * dpdx;
     v_corr = v_star - dt / rho * dpdy;
end

function p = poisson_solver(rhs, dx, dy)
     % V-cycle solver with gs smoothing
     maxLevel = floor(log2(min(size(rhs)))) - 1;
     p = fmg_vcycle(rhs, dx, dy, maxLevel);
end

function p = fmg_vcycle(rhs, dx, dy, level)
     if level == 0
         p = zeros(size(rhs));
         p = gauss_seidel(p, rhs, dx, dy, 150);
     else
         coarse_rhs = restrict(rhs);
         coarse_p = fmg_vcycle(coarse_rhs, 2*dx, 2*dy, level -
     1);
         fine_p = prolong(coarse_p);
         fine_p = gauss_seidel(fine_p, rhs, dx, dy, 150);
         p = fine_p;
     end
end

function out = gauss_seidel(p, rhs, dx, dy, iterations)
     [Ny, Nx] = size(p);
     dx2 = dx^2; dy2 = dy^2;
     denom = 2*(dx2 + dy2);
     for iter = 1:iterations
         for j = 2:Ny-1
             for i = 2:Nx-1
                 p(j,i) = ((p(j,i+1) + p(j,i-1))*dy2 + (p(j+1,
```

```
i) + p(j-1,i))*dx2 - rhs(j,i)*dx2*dy2) / denom;
            end
        end
    end
    out = p;
end

function coarse = restrict(fine)
    coarse = fine(1:2:end, 1:2:end);
end

function fine = prolong(coarse)
    [Ny, Nx] = size(coarse);
    fine = zeros(2*Ny, 2*Nx);
    fine(1:2:end, 1:2:end) = coarse;
    fine(2:2:end, 1:2:end) = coarse;
    fine(1:2:end, 2:2:end) = coarse;
    fine(2:2:end, 2:2:end) = coarse;
end
function div = divergence(u, v, dx, dy)
    div = (u(:,[2:end end]) - u(:,[1 1:end-1])) / (2*dx) + ...
          (v([2:end end],:) - v([1 1:end-1],:)) / (2*dy);
end

function [u_star, v_star] = explicit_predictor(u, v, p, rho, nu, dt, dx, dy)
    [Ny, Nx] = size(u);
    u_star = u;
    v_star = v;

    %Laplacians
    uxx = zeros(Ny, Nx); uyy = zeros(Ny, Nx);
    vxx = zeros(Ny, Nx); vyy = zeros(Ny, Nx);

    uxx(:,2:Nx-1) = (u(:,3:Nx) - 2*u(:,2:Nx-1) + u(:,1:Nx-2)) / dx^2;
    uyy(2:Ny-1,:) = (u(3:Ny,:) - 2*u(2:Ny-1,:) + u(1:Ny-2,:)) / dy^2;

    vxx(:,2:Nx-1) = (v(:,3:Nx) - 2*v(:,2:Nx-1) + v(:,1:Nx-2)) / dx^2;
    vyy(2:Ny-1,:) = (v(3:Ny,:) - 2*v(2:Ny-1,:) + v(1:Ny-2,:)) / dy^2;

    %Pressure grads
    px = zeros(Ny, Nx); py = zeros(Ny, Nx);
    px(:,2:Nx-1) = (p(:,3:Nx) - p(:,1:Nx-2)) / (2*dx);
    py(2:Ny-1,:) = (p(3:Ny,:) - p(1:Ny-2,:)) / (2*dy);
```

```

    u_star = u + dt * (-px/rho + nu * (uxx + uyy));
    v_star = v + dt * (-py/rho + nu * (vxx + vyy));
end
```

A.1.5 Pulsating flow

```
clc; clear; close all;

y=48;
Ly=0.2;
dy=Ly/y;

visc=0.01;
rho=1;

% Womersleyflow parameters
U0=1;
f=5;
omega=2*pi*f;
T=2*pi/omega;
dt=T/2000;% 2000 points per cycle

dpdx_amp=dpx_calc(U0, omega, visc, rho, Ly);  % analytical dp
    /dx amplitude
tsteps=5400;

yc=linspace(dy/2, Ly - dy/2, y)';
u=womer_vel(yc, 0, dpdx_amp, omega, visc, rho, Ly);  % t=0

% Initial acceleration consistency to reduce startup error
du_dt0=womersley_acceleration(yc, 0, dpdx_amp, omega, visc,
    rho, Ly);
u=u+0.5*dt*du_dt0;
ut=u;

err_l2_hist=zeros(tsteps,1);

% Plot Setup
figure('Name','Pulsating/Womersley Flow');
subplot(2,1,1);
hProfile=plot(u, yc, 'bo-', 'DisplayName', 'Numerical'); hold
     on;
hExact=plot(u, yc, 'r-', 'DisplayName', 'Analytical');
xlabel('u (m/s)'); ylabel('y'); grid on;
legend; title('Velocity Profile');

subplot(2,1,2);
```

```
hError=plot(0,0); xlabel('Time Step'); ylabel('Relative L2
    Error'); grid on;
title(sprintf('L2 Error vs Time Step (dt=%.2e s)', dt));

% Time-stepping Loop
old_dpx=dpdx_amp;
for n=1:tsteps
    t=n*dt;
    dpdx=dpdx_amp*sin(omega*t);
    smooth_dpx=0.5*(dpdx+old_dpx);
    old_dpx=dpdx;

    %  C r a n k Nicolson  time integration (semi-implicit)
    for j=2:y-1
        diffu_n=visc*(u(j+1) - 2*u(j)+u(j-1))/dy^2;
        frocin_n=old_dpx/rho;
         diffu_np1=visc*(ut(j+1) - 2*ut(j)+ut(j-1))/dy^2;
        frocin_np1=dpdx/rho;
        ut(j)=u(j)+dt/2*(diffu_n+frocin_n+diffu_np1+
    frocin_np1);
    end

    % No-slip BCs
    ut(1)=0;
    ut(end)=0;
    u=ut;


    u_exact=womer_vel(yc, t, dpdx_amp, omega, visc, rho, Ly);
    u_exact(1)=0; u_exact(end)=0;


    err=u - u_exact;
    err_l2=norm(err,2);
    norm_ref=norm(u_exact,2)+1e-10;
    err_l2_hist(n)=err_l2/norm_ref;

    % Plot every few steps
    if mod(n, 20) == 0 || n == 1
        set(hProfile, 'XData', u, 'YData', yc);
        set(hExact, 'XData', u_exact, 'YData', yc);
        set(hError, 'XData', 1:n, 'YData', err_l2_hist(1:n));
        drawnow;
    end
end


fprintf('Final Relative L2 Error: %.2e\n', err_l2_hist(end));

```

```
%helper functions
function dpdx_amp=dpx_calc(U0, omega, visc, rho, Ly)
    i=1i;
    lambda=sqrt(i*omega/visc);
    spatial_factor=abs(1 - cosh(lambda*0)/cosh(lambda*Ly/2));
    dpdx_amp=-U0*omega*rho/spatial_factor;
end

function u=womer_vel(y, t, dpdx_amp, omega, visc, rho, Ly)
    i=1i;
    lambda=sqrt(i*omega/visc);
    y_shifted=y - Ly/2;
    denom=cosh(lambda*Ly/2);
    spatial_part=1 - cosh(lambda*y_shifted)/denom;
    time_factor=exp(i*(omega*t - pi/2));
    prefactor=dpdx_amp/(i*omega*rho);
    u_complex=prefactor*spatial_part*time_factor;
    u=real(u_complex);
end

function du_dt=womersley_acceleration(y, t, dpdx_amp, omega,
   visc, rho, Ly)
    i=1i;
    lambda=sqrt(i*omega/visc);
    y_shifted=y - Ly/2;
    denom=cosh(lambda*Ly/2);
    spatial_part=1 - cosh(lambda*y_shifted)/denom;
    time_factor=omega*exp(i*(omega*t+pi/2));
    prefactor=dpdx_amp/rho;
    du_dt_complex=prefactor*spatial_part*time_factor;
    du_dt=real(du_dt_complex);
end
```

A.2 IBM code

A.2.1 simple shear flow

```
clc; clear; close all

% Domain and grid setup
x=32;
y=32;
Lx=1; Ly=0.2;
dx=Lx/x; dy=Ly/y;

visc=0.01;
Ut=10; Ub=0;
dt=min(1, 0.5*dx^2/visc);

```

```
[X, Y]=meshgrid(dx/2:dx:Lx-dx/2, dy/2:dy:Ly-dy/2);

u=zeros(y+2, x+2); v=zeros(y+2, x+2);
ut=u; vt=v; p=zeros(y+2, x+2);

% Analytical velocity profile
ya=linspace(dy/2, Ly-dy/2, y);
ua=(Ut-Ub)/Ly*ya+Ub;

Nb=100; s=linspace(0, 1, Nb);
Xb=Lx*s; Yb=Ly/2*ones(1, Nb);
tsteps=300;
err_l2_hist=zeros(tsteps, 1);
FxL_old=zeros(1, Nb);
FyL_old=zeros(1, Nb);
alpha=0.5;

for n=1:tsteps
    [u,v, ~]=applybcs(u,v, Ut,Ub, p);
    [uL, vL]=vel_interpolate(u,v, Xb,Yb, dx,dy);
    epsilon=1;
    FxL=epsilon*(0-uL);
    FyL=epsilon*(0-vL);

    FxL=alpha*FxL+(1-alpha)*FxL_old;
    FyL=alpha*FyL+(1-alpha)*FyL_old;
    FxL_old=FxL; FyL_old=FyL;

    forceField=spread_force(FxL,FyL, Xb,Yb, dx,dy,x,y);
    fx=forceField(:,:,1);
    fy=forceField(:,:,2);

    u_center=0.5*(u(2:end-1, 2:end-1)+u(2:end-1, 3:end));
    fx_center=0.5*(fx(2:end-1,2:end-1)+fx(2:end-1,3:end));
    u_center_new=implicit_diffusion_u(u_center+dt*fx_center,
   Lx, Ly, x, y, dt, visc);
    u(2:end-1, 2:end-1)=u_center_new;
    u(2:end-1, 3:end)=u_center_new;
    ut=u;

    v_center=v(2:end-1, 2:end-1);
    fy_center=fy(2:end-1, 2:end-1);
    v_center_new=implicit_diffusion_v(v_center+dt*fy_center,
   Lx, Ly, x, y, dt, visc);
    v(2:end-1, 2:end-1)=v_center_new;
    vt=v;

    divut=(ut(2:end-1,3:end)-ut(2:end-1,2:end-1))/dx+   (vt
   (3:end,2:end-1)-vt(2:end-1,2:end-1))/dy;
```

```

    rhs_full=zeros(y+2, x+2);
    rhs_full(2:end-1, 2:end-1)=divut/dt;

    for cycle=1:20
        p_old=p;
        p=V_cycle(p, rhs_full, Lx, Ly, x, y);
        if norm(p(:)-p_old(:), 2)/norm(p_old(:), 2) < 1e-6
            break;
        end
    end

    [u, v]=rhie_chow_correction(ut, vt, p, dx, dy, dt);

    uc=0.5*(u(2:end-1,2:end-1)+u(2:end-1,3:end));
    u_profile=mean(uc, 2);
    error_profile=u_profile-ua';
    L2_error=sqrt(sum(error_profile.^2)/length(error_profile))/sqrt(sum(ua.^2)/length(ua));
    err_l2_hist(n)=L2_error;

    if mod(n,10) == 0
        fprintf("Step %d, L2 Error=%.2e\n", n, L2_error);

        x_index=round(x/2)+1;
        uc_current=0.5*(u(2:end-1,2:end-1)+u(2:end-1,3:end));

        figure(1); clf;
        subplot(1,2,1);
        plot(uc_current(:, x_index), ya, 'b-', 'LineWidth', 2); hold on;
        plot(ua, ya, 'r--', 'LineWidth', 2);
        xlabel('u'); ylabel('y'); title(['Velocity Profile at x=Lx/2, Step=', num2str(n)]);
        legend('Numerical', 'Analytical','Location','south'); grid on;

        subplot(1,2,2);
        semilogy(1:n, err_l2_hist(1:n), 'k-', 'LineWidth', 2);
        xlabel('Time step'); ylabel('Relative L2 Error');
        title('L2 Error Convergence'); grid on;
        drawnow;
    end
end

function [u, v, p]=applybcs(u, v, Ut, Ub, p)
    u(:,1)=u(:,2);
    u(:,end)=u(:,end-1);
```

```

    v(:,1)=0;
    v(:,end)=0;

    u(end,:)=Ut;
    u(1,:)=Ub;

    v([1 end],:)=0;

    p(:,1)=p(:,2);
    p(:,end)=p(:,end-1);
    p(1,:)=p(2,:);
    p(end,:)=p(end-1,:);
end

function [uL, vL]=vel_interpolate(u, v, Xb, Yb, dx, dy)
    Nb=length(Xb);
    uL=zeros(1,Nb);
    vL=zeros(1,Nb);
    for k=1:Nb
        xk=Xb(k); yk=Yb(k);
        i0=floor(xk/dx)+1; j0=floor(yk/dy)+1;
        for i=i0-1:i0+2
            for j=j0-1:j0+2
                if i >= 1 && i <= size(u,2) && j >= 1 && j <= size(u,1)
                    xi=(i-1)*dx; yj=(j-1)*dy;
                    phi=delta_kernel((xk-xi)/dx)*delta_kernel((yk-yj)/dy);
                    uL(k)=uL(k)+u(j,i)*phi;
                    vL(k)=vL(k)+v(j,i)*phi;
                end
            end
        end
    end
end

function f=spread_force(FxL, FyL, Xb, Yb, dx, dy, Nx, Ny)
    f=zeros(Ny+2, Nx+2, 2); Nb=length(Xb);
    for k=1:Nb
        xk=Xb(k); yk=Yb(k);
        i0=floor(xk/dx)+1; j0=floor(yk/dy)+1;
        for i=i0-1:i0+2
            for j=j0-1:j0+2
                xi=(i-1)*dx; yj=(j-1)*dy;
                phi=delta_kernel((xk-xi)/dx)*delta_kernel((yk-yj)/dy);
                if i >= 1 && i <= Nx+2 && j >= 1 && j <= Ny+2
                    f(j,i,1)=f(j,i,1)+FxL(k)*phi*dx*dy;
```

```
                    f(j,i,2)=f(j,i,2)+FyL(k)*phi*dx*dy;
                end
            end
        end
    end
end
% 4-point kernel (Peskins standard)
function val=delta_kernel(r)
    r=abs(r);
    if r < 1
        val=0.125*(3-2*r+sqrt(1+4*r-4*r^2));
    elseif r < 2
        val=0.125*(5-2*r-sqrt(-7+12*r-4*r^2));
    else
        val=0;
    end
end



function u_new=implicit_diffusion_u(u_old, Lx, Ly, Nx, Ny, dt, nu)
    dx=Lx/Nx;
    dy=Ly/Ny;
    N=Nx*Ny;
    A=sparse(N, N);
    b=zeros(N, 1);
    coeff_center=1+2*dt*nu*(1/dx^2+1/dy^2);
    coeff_x=-dt*nu/dx^2;
    coeff_y=-dt*nu/dy^2;
    index=@(i,j) (j-1)*Nx+i;
    for j=1:Ny
        for i=1:Nx
            n=index(i,j);
            % Top and bottom wlls    Dirichilet BC
            if j == 1
                A(n,:)=0;
                A(n,n)=1;
                b(n)=0;       % Bottom wall velocity Ub
                continue;
            elseif j == Ny
                A(n,:)=0;
                A(n,n)=1;
                b(n)=10;      % Top wall velocity Ut
                continue;
            end
            % Interior and Left/Right
            A(n,n)=coeff_center;
            %Westside
```

```
            if i > 1
                A(n, index(i-1,j))=coeff_x;
            else
                A(n,n)=A(n,n)-coeff_x;  % Neumann
            end
            %Eastside
            if i < Nx
                A(n, index(i+1,j))=coeff_x;
            else
                A(n,n)=A(n,n)-coeff_x;  % Neumann
            end
            A(n, index(i,j-1))=coeff_y;
            A(n, index(i,j+1))=coeff_y;
            b(n)=u_old(j,i);
        end
    end

    u_vec=A \ b;
    u_new=reshape(u_vec, [Nx, Ny])';
end

% V-Cycle Function
function p=V_cycle(p, rhs, Lx, Ly, Nx, Ny)
    if Nx <= 4 || Ny <= 4
        p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, 100);
        return;
    end

    p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, 10);
    res=compute_residual(p, rhs, Lx, Ly, Nx, Ny);
    res_coarse=restrict(res);
    Nc_x=size(res_coarse,2)-2;
    Nc_y=size(res_coarse,1)-2;
    e_coarse=zeros(Nc_y+2, Nc_x+2);
    e_coarse=V_cycle(e_coarse, res_coarse, Lx, Ly, Nc_x, Nc_y
   );
    e_fine=prolong(e_coarse, Nx, Ny);
    p=p+e_fine;

    p=sor_solve(p,rhs, Lx, Ly, Nx, Ny, 7);
end

% SOR Smoother (Local)
function p=sor_solve(p,rhs, Lx, Ly, Nx, Ny, Niter)
    dx=Lx/Nx;
    dy=Ly/Ny;
    B=1.9;
    for iter=1:Niter
        p_old=p;
```

```
        for i=2:Nx+1
            for j=2:Ny+1
                p(j,i)=(1-B)*p(j,i)+B*0.5*((dy^2*(p(j,i+1)+p(
    j,i-1))+  dx^2*(p(j+1,i)+p(j-1,i))-   dx^2*dy^2*rhs(j,i))
    /(2*(dx^2+dy^2)));
            end
        end
        % Neumann boundary conditions
        p(:,1)  =p(:,2);
        p(:,end)=p(:,end-1);
        p(1,:)  =p(2,:);
        p(end,:)=p(end-1);

        if norm(p(:)-p_old(:), 2) < 1e-8
            break;
        end
    end
end
function res=compute_residual(p, rhs, Lx, Ly, Nx, Ny)
    dx=Lx/Nx;
    dy=Ly/Ny;

    res=zeros(Ny+2, Nx+2);
    for i=2:Nx+1
        for j=2:Ny+1
            laplace=(p(j,i+1)-2*p(j,i)+p(j,i-1))/dx^2+  (p(j
    +1,i)-2*p(j,i)+p(j-1,i))/dy^2;
            res(j,i)=rhs(j,i)-laplace;
        end
    end
end
function coarse=restrict(fine)
    [Nyf, Nxf]=size(fine);
    Nxc=ceil((Nxf-2)/2);
    Nyc=ceil((Nyf-2)/2);

    coarse=zeros(Nyc+2, Nxc+2);
    for i=2:Nxc+1
        for j=2:Nyc+1
            i_f=2*(i-1);
            j_f=2*(j-1);
            neighbors=fine(j_f-1:j_f+1, i_f-1:i_f+1);
            weights=[1 2 1; 2 4 2; 1 2 1];
            % Handle edges
            valid=~isnan(neighbors);
            w_sum=sum(weights(valid));
            coarse(j,i)=sum(neighbors(valid) .* weights(valid
    ), 'all')/w_sum;
        end
```

```
    end
end

% Prolongation bilinear
function fine=prolong(coarse, Nxf, Nyf)
    Nxc=size(coarse,2)-2;
    Nyc=size(coarse,1)-2;
    fine=zeros(Nyf+2, Nxf+2);
    for i=2:Nxc+1
        for j=2:Nyc+1
            i_f=2*(i-1);
            j_f=2*(j-1);
            fine(j_f,i_f)  =fine(j_f, i_f)  +coarse(j,i);
            fine(j_f+1,i_f)  =fine(j_f+1, i_f)  +coarse(j,i);
            fine(j_f,i_f+1)=fine(j_f, i_f+1)+coarse(j,i);
            fine(j_f+1,i_f+1)=fine(j_f+1,i_f+1)+coarse(j,i);
        end
    end
    fine(2:end-1,2:end-1)=fine(2:end-1,2:end-1)/4;
end


function [u_corr, v_corr]=rhie_chow_correction(ut, vt, p, dx,
    dy, dt)
    [Ny, Nx]=size(p);
    u_corr=ut;
    v_corr=vt;
    %  u correction
    u_corr(2:end-1,3:end-1)=ut(2:end-1,3:end-1)-  dt*(p(2:end
   -1,3:end-1)-p(2:end-1,2:end-2))/ dx;
    %  v correction
    v_corr(3:end-1,2:end-1)=vt(3:end-1,2:end-1)- dt*(p(3:end
   -1,2:end-1)-p(2:end-2,2:end-1)) /dy;
end


function v_new=implicit_diffusion_v(v_old, Lx, Ly, Nx, Ny, dt
   , nu)
    dx=Lx/Nx;
    dy=Ly/Ny;
    N=Nx*Ny;
    A=sparse(N,N);
    b=zeros(N, 1);
    coeff_center=1+2*dt*nu*(1/dx^2+1/dy^2);
    coeff_x=-dt*nu/dx^2;
    coeff_y=-dt*nu/dy^2;
    index=@(i,j) (j-1)*Nx+i;

    for j=1:Ny
```

```
        for i=1:Nx
            n=index(i,j);
            %top and bottom: dirichilet (v=0)
            if j == 1 || j == Ny
                A(n,:)=0;
                A(n,n)=1;
                b(n)=0;
                continue;
            end
            A(n,n)=coeff_center;
            if i > 1
                A(n, index(i-1,j))=coeff_x;
            else
                A(n,n)=A(n,n)-coeff_x;
            end
            if i < Nx
                A(n, index(i+1,j))=coeff_x;
            else
                A(n,n)=A(n,n)-coeff_x;
            end
            A(n, index(i,j-1))=coeff_y;
            A(n, index(i,j+1))=coeff_y;

            b(n)=v_old(j,i);
        end
    end

    v_vec=A \ b;
    v_new=reshape(v_vec, [Nx, Ny])';
end
```

A.2.2 Plane Poiseuille flow

```
clc; clear; close all

% Domain and grid setup
x=32; y=32;
Lx=1; Ly=0.2;
dx=Lx/x; dy=Ly/y;
visc=0.01;
Ut=0; Ub=0;
dt=min(1, 0.5*dx^2/visc);

rho=1;                  % density
dpx=-1;                 % pressure gradient(dp/dx -ve then flow
   in +ve)
[X, Y]=meshgrid(dx/2:dx:Lx-dx/2, dy/2:dy:Ly-dy/2);
u=zeros(y+2, x+2); v=zeros(y+2, x+2);
ut=u; vt=v; p=zeros(y+2, x+2);
```

```
ya=linspace(0, Ly, y);

ua=( -dpx/(2*visc) )*ya .* (Ly - ya);
Nb=100; s=linspace(0, 1, Nb);

% Time loop
tsteps=300; L2_error_history=zeros(tsteps, 1);
FxL_old=zeros(1, Nb); FyL_old=zeros(1, Nb);
alpha=0.5;

for n=1:tsteps
    [u, v, ~]=applybcs(u,v, Ut, Ub, p);

    fx=ones(y+2, x+2)*(-dpx)/rho;  % uniform body force in x
    fy=zeros(y+2, x+2);            % no vertical force

Nx_u=x+1;    % for u: faces in x
Ny_u=y;

u_staggered=u(2:end-1, 2:end);      % 32 33
fx_staggered=fx(2:end-1, 2:end);    % 32 33

u_new=implicit_diffusion_u(u_staggered+dt*fx_staggered, Lx,
    Ly, Nx_u, Ny_u, dt, visc);
u(2:end-1, 2:end)=u_new;
ut=u;
    v_center=v(2:end-1, 2:end-1);
    fy_center=fy(2:end-1, 2:end-1);
    v_center_new=implicit_diffusion_v(v_center+dt*fy_center,
    Lx, Ly, x, y, dt, visc);
    v(2:end-1, 2:end-1)=v_center_new;
    vt=v;

    divut=(ut(2:end-1,3:end) - ut(2:end-1,2:end-1))/dx+...
            (vt(3:end,2:end-1) - vt(2:end-1,2:end-1))/dy;

    rhs_full=zeros(y+2, x+2);
    rhs_full(2:end-1, 2:end-1)=divut/dt;

    for cycle=1:30
        p_old=p;
        p=V_cycle(p, rhs_full, Lx, Ly, x, y);
        if norm(p(:) - p_old(:), 2)/norm(p_old(:), 2) < 1e-8
            break;
        end
    end

```

```
    [u, v]=rhie_chow_correction(ut, vt, p, dx, dy, dt);

    uc=0.5*(u(2:end-1,2:end-1)+u(2:end-1,3:end));
    u_profile=mean(uc, 2);
    error_profile=u_profile - ua';
    L2_error=sqrt(sum(error_profile.^2)/length(error_profile))/sqrt(sum(ua.^2)/length(ua));
    L2_error_history(n)=L2_error;

    if mod(n,10) == 0
        fprintf("Step %d, L2 Error=%.2e\n", n, L2_error);

        x_index=round(x/2)+1;
        uc_current=0.5*(u(2:end-1,2:end-1)+u(2:end-1,3:end));

        figure(1); clf;
        subplot(1,2,1);
        plot(uc_current(:, x_index), ya, 'b-', 'LineWidth', 2); hold on;
        plot(ua, ya, 'r--', 'LineWidth', 2);
        xlabel('u'); ylabel('y'); title(['Velocity Profile at x=Lx/2, Step=', num2str(n)]);
        legend('Numerical', 'Analytical','Location','south'); grid on;

        subplot(1,2,2);
        semilogy(1:n, L2_error_history(1:n), 'k-', 'LineWidth', 2);
        xlabel('Time step'); ylabel('Relative L2 Error');
        title('L2 Error Convergence'); grid on;
        drawnow;
    end
end

function [u, v, p]=applybcs(u, v, Ut, Ub, p)
    u(:,1)=u(:,2);
    u(:,end)=u(:,end-1);

    v(:,1)=0;
    v(:,end)=0;

    u(end,:)=Ut;
    u(1,:)=Ub;

    v([1 end],:)=0;

    p(:,1)=p(:,2);
    p(:,end)=p(:,end-1);
    p(1,:)=p(2,:);
```

```
    p(end,:)=p(end-1,:);
end

function [uL, vL]=vel_interpolate(u, v, Xb, Yb, dx, dy)
    Nb=length(Xb);
    uL=zeros(1,Nb);
    vL=zeros(1,Nb);
    for k=1:Nb
        xk=Xb(k); yk=Yb(k);
        i0=floor(xk/dx)+1; j0=floor(yk/dy)+1;
        for i=i0-1:i0+2
            for j=j0-1:j0+2
                if i >= 1 && i <= size(u,2) && j >= 1 && j <= size(u,1)
                    xi=(i-1)*dx; yj=(j-1)*dy;
                    phi=delta_kernel((xk - xi)/dx)*delta_kernel((yk - yj)/dy);
                    uL(k)=uL(k)+u(j,i)*phi;
                    vL(k)=vL(k)+v(j,i)*phi;
                end
            end
        end
    end
end

function f=spread_force(FxL, FyL, Xb, Yb, dx, dy, Nx, Ny)
    f=zeros(Ny+2, Nx+2, 2); Nb=length(Xb);
    for k=1:Nb
        xk=Xb(k); yk=Yb(k);
        i0=floor(xk/dx)+1; j0=floor(yk/dy)+1;
        for i=i0-1:i0+2
            for j=j0-1:j0+2
                xi=(i-1)*dx; yj=(j-1)*dy;
                phi=delta_kernel((xk - xi)/dx)*delta_kernel((yk - yj)/dy);
                if i >= 1 && i <= Nx+2 && j >= 1 && j <= Ny+2
                    f(j,i,1)=f(j,i,1)+FxL(k)*phi*dx*dy;
                    f(j,i,2)=f(j,i,2)+FyL(k)*phi*dx*dy;
                end
            end
        end
    end
end

% 4-point kernel (Peskins standard)
function val=delta_kernel(r)
    r=abs(r);
    if r < 1
        val=0.125*(3 - 2*r+sqrt(1+4*r - 4*r^2));
```

```
    elseif r < 2
        val=0.125*(5 - 2*r - sqrt(-7+12*r - 4*r^2));
    else
        val=0;
    end
end



function u_new=implicit_diffusion_u(u_old, Lx, Ly, Nx, Ny, dt, nu)
    dx=Lx/Nx;
    dy=Ly/Ny;
    N=Nx*Ny;
    A=sparse(N, N);
    b=zeros(N, 1);
    coeff_center=1+2*dt*nu*(1/dx^2+1/dy^2);
    coeff_x=-dt*nu/dx^2;
    coeff_y=-dt*nu/dy^2;
    index=@(i,j) (j-1)*Nx+i;

    for j=1:Ny
        for i=1:Nx
            n=index(i,j);
            % top and bottom walls  Dirichilet BC
            if j == 1
                A(n,:)=0;
                A(n,n)=1;
                b(n)=0;        % Bottom wall velocity Ub=0
                continue;
            elseif j == Ny
                A(n,:)=0;
                A(n,n)=1;
                b(n)=0;       % Top wall velocity Ut=0
                continue;
            end
            A(n,n)=coeff_center;
            if i > 1
                A(n, index(i-1,j))=coeff_x;
            else
                A(n,n)=A(n,n) - coeff_x;  % Neumann
            end
            if i < Nx
                A(n, index(i+1,j))=coeff_x;
            else
                A(n,n)=A(n,n) - coeff_x;  % Neumann
            end
            A(n, index(i,j-1))=coeff_y;
            A(n, index(i,j+1))=coeff_y;
```

```
                b(n)=u_old(j,i);
            end
        end
        u_vec=A \b;
        u_new=reshape(u_vec, [Nx, Ny])';
end

% V-Cycle Function
function p=V_cycle(p, rhs, Lx, Ly, Nx, Ny)
    if Nx <= 4 || Ny <= 4
        p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, 100);
        return;
    end

    p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, 15);
    res=residualcalc(p, rhs, Lx, Ly, Nx, Ny);
    res_coarse=restrict(res);


    Nc_x=size(res_coarse,2) - 2;
    Nc_y=size(res_coarse,1) - 2;
    e_coarse=zeros(Nc_y+2, Nc_x+2);
    e_coarse=V_cycle(e_coarse, res_coarse, Lx, Ly, Nc_x, Nc_y);
    e_fine=prolong(e_coarse, Nx, Ny);
    p=p+e_fine;


    p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, 10);
end


function p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, Niter)
    dx=Lx/Nx;
    dy=Ly/Ny;
    B=1.9;

    for iter=1:Niter
        p_old=p;
        for i=2:Nx+1
            for j=2:Ny+1
                p(j,i)=(1-B)*p(j,i)+B*0.5*((dy^2*(p(j,i+1)+p(j,i-1))+ dx^2*(p(j+1,i)+p(j-1,i)) -dx^2*dy^2*rhs(j,i))/(2*(dx^2+dy^2)));
            end
        end

        % Neumann boundary conditions
        p(:,1)  =p(:,2);
```

```
        p(:,end)=p(:,end-1);
        p(1,:)  =p(2,:);
        p(end,:)=p(end-1);

        if norm(p(:) -p_old(:), 2) < 1e-8
            break;
        end
    end
end

function res=residualcalc(p,rhs, Lx, Ly, Nx, Ny)
    dx=Lx/Nx;
    dy=Ly/Ny;
    res=zeros(Ny+2, Nx+2);
    for i=2:Nx+1
        for j=2:Ny+1
            res(j,i)=rhs(j,i) -(p(j,i+1) - 2*p(j,i)+p(j,i-1))/dx^2+ (p(j+1,i) - 2*p(j,i)+p(j-1,i))/dy^2;
        end
    end
end
function coarse=restrict(fine)
    [Nyf, Nxf]=size(fine);
    Nxc=ceil((Nxf - 2)/2);
    Nyc=ceil((Nyf - 2)/2);
    coarse=zeros(Nyc+2, Nxc+2);
    for i=2:Nxc+1
        for j=2:Nyc+1
            i_f=2*(i-1);
            j_f=2*(j-1);
            neighbors=fine(j_f-1:j_f+1, i_f-1:i_f+1);
            weights=[1 2 1; 2 4 2; 1 2 1];
            valid=~isnan(neighbors);
            w_sum=sum(weights(valid));
            coarse(j,i)=sum(neighbors(valid) .* weights(valid), 'all')/w_sum;
        end
    end
end

% Prolongation bilinear
function fine=prolong(coarse, Nxf, Nyf)
    Nxc=size(coarse,2)- 2;
    Nyc=size(coarse,1)- 2;

    fine=zeros(Nyf+2, Nxf+2);

    for i=2:Nxc+1
        for j=2:Nyc+1
```

```
            i_f=2*(i-1);
            j_f=2*(j-1);

            fine(j_f, i_f)  =fine(j_f, i_f)  +coarse(j,i);
            fine(j_f+1, i_f)  =fine(j_f+1, i_f)  +coarse(j,i);
            fine(j_f,i_f+1)=fine(j_f, i_f+1)+coarse(j,i);
            fine(j_f+1, i_f+1)=fine(j_f+1, i_f+1)+coarse(j,i);
        end
    end
    fine(2:end-1,2:end-1)=fine(2:end-1,2:end-1)/4;
end


function [u_corr, v_corr]=rhie_chow_correction(ut, vt, p, dx, dy, dt)
    [Ny, Nx]=size(p);
    u_corr=ut;
    v_corr=vt;
    u_corr(2:end-1,3:end-1)=ut(2:end-1,3:end-1) -dt*(p(2:end-1,3:end-1) - p(2:end-1,2:end-2))/dx;
    v_corr(3:end-1,2:end-1)=vt(3:end-1,2:end-1) -dt*(p(3:end-1,2:end-1) - p(2:end-2,2:end-1))/dy;
end


function v_new=implicit_diffusion_v(v_old, Lx, Ly, Nx, Ny, dt, nu)
    dx=Lx/Nx;
    dy=Ly/Ny;
    N=Nx*Ny;
    A=sparse(N, N);
    b=zeros(N, 1);
    coeff_center=1+2*dt*nu*(1/dx^2+1/dy^2);
    coeff_x=-dt*nu/dx^2;
    coeff_y=-dt*nu/dy^2;
    index=@(i,j) (j-1)*Nx+i;

    for j=1:Ny
        for i=1:Nx
            n=index(i,j);
            %top and bottom: Dirichilet(v=0)
            if j == 1 || j == Ny
                A(n,:)=0;
                A(n,n)=1;
                b(n)=0;
                continue;
            end
```

```
            A(n,n)=coeff_center;
            if i > 1
                A(n, index(i-1,j))=coeff_x;
            else
                A(n,n)=A(n,n) - coeff_x;
            end
            if i < Nx
                A(n, index(i+1,j))=coeff_x;
            else
                A(n,n)=A(n,n) - coeff_x;
            end
            A(n, index(i,j-1))=coeff_y;
            A(n, index(i,j+1))=coeff_y;
            b(n)=v_old(j,i);
        end
    end
    v_vec=A \ b;
    v_new=reshape(v_vec, [Nx, Ny])';
end
```

A.2.3 Oscillating shear flow

```
clc; clear;

x=32;
y=32;
Lx=1;
Ly=.2;
dx=Lx/x;
dy=Ly/y;

visc=.05;
U0=10;                    % Amplitude
f=5 ;                   % Frequency (Hz)
omega=2*pi*f;       % Angular frequency

Ub=0;                     % Bottom wall velocity (stationary)

CFL=0.5;
dt1=1e6;
dt2=CFL*min(dx, dy)/abs(U0);
dt=0.9*min(dt1, dt2)


[X, Y]=meshgrid(dx/2:dx:Lx-dx/2, dy/2:dy:Ly-dy/2);

y_ib=Ly*ones(1, x);
x_ib=linspace(dx/2, Lx-dx/2, x);

```

```
u =zeros(y+2, x+2);
v =zeros(y+2, x+2);
ut=zeros(y+2, x+2);
vt=zeros(y+2, x+2);
p =zeros(y+2, x+2);

fx=zeros(y+2, x+2);
fy=zeros(y+2, x+2);

% Plot setup
figure;
subplot(1,2,1);
prof_line=plot(zeros(y,1), linspace(dy/2,Ly-dy/2,y), 'b-', '
    LineWidth', 2); hold on;
anal_line=plot(zeros(y,1), linspace(dy/2,Ly-dy/2,y), 'r--', '
    LineWidth', 2);
legend('Numerical','Analytical','Location','south');
xlabel('u'); ylabel('y'); title('Velocity Profile at Lx/2');
    grid on;

subplot(1,2,2);
err_plot=semilogy(0,0,'k');
xlabel('Timestep'); ylabel('Relative L2 Error'); title('
    Convergence'); grid on;
tsteps=2000;
err_l2_hist=zeros(tsteps,1);


%
% %Video setup
% k=VideoWriter('IBM_OscShearFlow.mp4', 'MPEG-4');
% k.FrameRate=10;
% open(k);

%Time loop
for n=1:tsteps
    time=n*dt;
    Ut=U0*sin(omega*time);
    u_desired=Ut*ones(size(x_ib));
    u_ib=interpolate(u, x_ib, y_ib, dx, dy);
    f_ib=(u_desired-u_ib)/dt;
    [fx, fy]=spreadf(f_ib, zeros(size(f_ib)), x_ib, y_ib,
   size(u), dx, dy);

    ut=u+dt*(visc*laplacian(u, dx, dy)+fx);
    vt=v+dt*(visc*laplacian(v, dx, dy)+fy);

    divut=(ut(2:end-1,3:end)-ut(2:end-1,2:end-1))/dx+(vt(3:
   end,2:end-1)-vt(2:end-1,2:end-1))/dy;
```

```
    rhs=divut/dt;
    [p,~]=sor_solver(p, rhs, Lx, Ly, x, y);

    u(2:end-1,2:end-1)=ut(2:end-1,2:end-1)-dt*(p(2:end-1,3:
end)-p(2:end-1,2:end-1))/dx;
    v(2:end-1,2:end-1)=vt(2:end-1,2:end-1)-dt*(p(3:end,2:end
-1)-p(2:end-1,2:end-1))/dy;

    u(end,:)=Ut;    % Top wall velocity
    u(1,:)  =Ub;    % Bottom wall stationary

    % Error analysis
    yc=linspace(dy/2, Ly-dy/2, y);
    uc=0.5*(u(2:end-1,2:end-1)+u(2:end-1,3:end));
    u_profile=mean(uc,2);
    ua=vel_anal(yc, time, U0, omega, visc);

    err=sqrt(sum((u_profile-ua').^2)/length(ua))/sqrt(sum(ua
.^2)/length(ua));
    err_l2_hist(n)=err;

    % Plot
    if mod(n,10) == 0
        set(prof_line, 'XData', u_profile, 'YData', yc);
        set(anal_line, 'XData', ua, 'YData', yc);
        set(err_plot, 'XData', 1:n, 'YData', err_l2_hist(1:n)
);
        drawnow;


        %         % Save frame to video
        % frame=getframe(gcf);
        % writeVideo(k, frame);
    end
end


% %Close Video
% close(k);
% disp('Video saved as IBM_OscShearFlow.mp4');


function u_analytical=vel_anal(y, t, U0, omega, visc)
        alpha=sqrt(omega/(2*visc));
        Ly=y(end);
        y_from_top=Ly-y;
        u_analytical=U0*exp(-alpha*y_from_top) .* sin(omega*t
-alpha*y_from_top);
end
```

```
function L=laplacian(f, dx, dy)
    L=zeros(size(f));
    L(2:end-1,2:end-1)=(f(2:end-1,3:end)-2*f(2:end-1,2:end-1)
   +f(2:end-1,1:end-2))/dx^2+ (f(3:end,2:end-1)-2*f(2:end
   -1,2:end-1)+f(1:end-2,2:end-1))/dy^2;
end

function u_ib=interpolate(u, x_ib, y_ib, dx, dy)
    u_ib=zeros(size(x_ib));
    for k=1:length(x_ib)
        i=floor(x_ib(k)/dx)+1;
        j=floor(y_ib(k)/dy)+1;
        wx=(x_ib(k)-(i-1)*dx)/dx;
        wy=(y_ib(k)-(j-1)*dy)/dy;
        u_ib(k)=(1-wx)*(1-wy)*u(j,i)+wx*(1-wy)*u(j,i+1)+(1-wx
   )*wy*u(j+1,i)+wx*wy*u(j+1,i+1);
    end
end

function [fx, fy]=spreadf(fx_ib, fy_ib, x_ib, y_ib, size_u,
   dx, dy)
    fx=zeros(size_u);
    fy=zeros(size_u);
    for k=1:length(x_ib)
        i=floor(x_ib(k)/dx)+1;
        j=floor(y_ib(k)/dy)+1;
        wx=(x_ib(k)-(i-1)*dx)/dx;
        wy=(y_ib(k)-(j-1)*dy)/dy;
        fx(j,i) =fx(j,i) +(1-wx)*(1-wy)*fx_ib(k);
        fx(j,i+1)=fx(j,i+1) +wx*(1-wy)*fx_ib(k);
        fx(j+1,i) =fx(j+1,i) +(1-wx)*wy*fx_ib(k);
        fx(j+1,i+1)=fx(j+1,i+1) +wx*wy*fx_ib(k);

        fy(j,i)=fy(j,i) +(1-wx)*(1-wy)*fy_ib(k);
        fy(j,i+1)=fy(j,i+1)+wx*(1-wy)*fy_ib(k);
        fy(j+1,i)=fy(j+1,i)+(1-wx)*wy*fy_ib(k);
        fy(j+1,i+1)  =fy(j+1,i+1)  +wx*wy*fy_ib(k);
    end
end

function [p, err]=sor_solver(p, rhs, Lx, Ly, Nx, Ny)
    dx=Lx/Nx;
    dy=Ly/Ny;
    B=1.9;
    tol=1e-10;
    maxit=8000;
    err=1e10;
```

```
    it=0;

    while err > tol && it < maxit
        p_old=p;
        for i=2:Nx+1
            for j=2:Ny+1
                if i < Nx+1 && j < Ny+1
                    p(j,i)=(1-B)*p(j,i)+B*0.5*((dy^2*(p(j,i+1)+p(j,i-1))+ dx^2*(p(j+1,i)+p(j-1,i))-dx^2*dy^2*rhs(j,i))/(2*(dx^2+dy^2)));
                end
            end
        end
        err=norm(p(:)-p_old(:), 2);
        it=it+1;
    end
end
```

A.2.4 Pulsating flow

```
clc; clear; close all;

% Parameters
Ny=48;
Ly=0.2;
dy=Ly/Ny;
visc=0.01;
rho=1;

U0=1;
f=5;
omega=2*pi*f;
T=2*pi/omega;
dt=T/2000;

tsteps=5500;
dpx_amp=dpx_calc(U0, omega, visc, rho, Ly);


y=linspace(-dy/2, Ly+dy/2, Ny+2)';
yc=y(2:end-1);
u=zeros(Ny+2, 1);  % including ghost nodes
ut=zeros(Ny+2, 1);

y_ib=[0, Ly];
x_ib=ones(size(y_ib));  % dummy x since it's 1D

% Plot setup
figure('Name','IBM Womersley Flow');
```

```
subplot(2,1,1);
hProf=plot(u(2:end-1), yc, 'b-', 'LineWidth', 2); hold on;
hAnal=plot(u(2:end-1), yc, 'r--', 'LineWidth', 2);
xlabel('u'); ylabel('y'); title('Velocity Profile'); grid on; legend('Numerical','Analytical','Location','south');

subplot(2,1,2);
hErr=plot(0,0); xlabel('Timestep'); ylabel('Relative L2 Error'); grid on;
err_l2_hist=zeros(tsteps,1);


u(2:end-1)=vel_womers(yc, 0, dpx_amp, omega, visc, rho, Ly);

% Time loop
for n=1:tsteps
    t=n * dt;
    dpdx=dpx_amp * sin(omega * t);
    u_ib_desired=[0; 0];
    u_ib=interpolation(u, y_ib, y, dy);
    f_ib_old=zeros(size(u_ib));
    alpha=0.3;

    f_ib_raw=(u_ib_desired-u_ib)/dt;
    f_ib=alpha * f_ib_old+(1-alpha) * f_ib_raw;
    f_ib_old=f_ib;
    F=spreadf(f_ib, y_ib, y, dy);

    for j=2:Ny+1
        diffu_n=visc * (u(j+1)-2*u(j)+u(j-1))/dy^2;
        diffu_np1=visc * (ut(j+1)-2*ut(j)+ut(j-1))/dy^2;
        f_avg=dpdx/rho+F(j);
        ut(j)=u(j)+dt * (0.5 * (diffu_n+diffu_np1)+f_avg);
    end
    u=ut;

    %analytical vel
    u_anal=vel_womers(yc, t, dpx_amp, omega, visc, rho, Ly);

    % Error
    err=u(2:end-1)-u_anal;
    err_l2_hist(n)=norm(err)/norm(u_anal);

    if mod(n,20)==0 || n==1
        set(hProf,'XData',u(2:end-1), 'YData', yc);
        set(hAnal,'XData',u_anal, 'YData', yc);
        set(hErr, 'XData', 1:n, 'YData', err_l2_hist(1:n));
        drawnow;
    end
```

```
end

fprintf('Final Relative L2 Error: %.2e\n', err_l2_hist(end));

% IBM helper funcs
function u_ib=interpolation(u, y_ib, y, dy)
    u_ib=zeros(size(y_ib));
    for k=1:length(y_ib)
        j=floor(y_ib(k)/dy)+1;
        wy=(y_ib(k)-y(j))/dy;
        u_ib(k)=(1-wy)*u(j)+wy*u(j+1);
    end
end

function F=spreadf(f_ib, y_ib, y, dy)
    F=zeros(size(y));
    for k=1:length(y_ib)
        j=floor(y_ib(k)/dy)+1;
        wy=(y_ib(k)-y(j))/dy;
        F(j)  =F(j)  +(1-wy) * f_ib(k);
        F(j+1)=F(j+1)+wy * f_ib(k);
    end
end

function dpx_amp=dpx_calc(U0, omega, visc, rho, Ly)
    i=1i;
    spatial_factor=abs(1-cosh(sqrt(i * omega/visc)*0)/cosh(
   sqrt(i * omega/visc)* Ly/2));
    dpx_amp=-U0 * omega * rho/spatial_factor;
end

function u=vel_womers(y, t, dpx_amp, omega, visc, rho, Ly)
    i=1i;
    y_shifted=y-Ly/2;
    denom=cosh(sqrt(i * omega/visc) * Ly/2);
    spatial_part=1-cosh(sqrt(i * omega/visc) * y_shifted)/
   denom;
    time_factor=exp(i * (omega * t-pi/2));
    prefactor=dpx_amp/(i * omega * rho);
    u_complex=prefactor * spatial_part * time_factor;
    u=real(u_complex);
end
```

A.2.5 Flow over a cylinder

```
clc; clear; close all;

Nx=256;
Ny=64;
```

```
Lx=2.0;
Ly=1.0;
dx=Lx/Nx;
dy=Ly/Ny;
x=linspace(0, Lx, Nx);
y=linspace(0, Ly, Ny);
[X, Y]=meshgrid(x, y);

rho=1.0;
nu=0.001;
U_inf=10;
dt=0.001;
steps=3000;
beta=1.5;  % SOR over-relaxation factor

cx=1.0;
cy=0.5;
R=0.1;
Nb=100;
theta=linspace(0, 2*pi, Nb);
Xb=cx+R*cos(theta);
Yb=cy+R*sin(theta);

% Field includes ghost cells
u=U_inf * ones(Ny+2, Nx+2);
v=zeros(Ny+2, Nx+2);
p=zeros(Ny+2, Nx+2);
ut=u; vt=v;

FxL_old=zeros(1, Nb);
FyL_old=zeros(1, Nb);
alpha=0.5;
Fx_total=zeros(1, steps);
Fy_total=zeros(1, steps);

%Time loop
for n=1:steps
    [u, v, p]=apply_bc(u, v, p, U_inf);
    ut=GS_diffuse(u, nu, dt, dx, dy);
    vt=GS_diffuse(v, nu, dt, dx, dy);

    [uL, vL]=vel_interpol(ut, vt, Xb, Yb, dx, dy);
    epsilon=1000;
    FxL=-epsilon * uL;
    FyL=-epsilon * vL;
    FxL=alpha *FxL+(1-alpha) *FxL_old;
    FyL=alpha *FyL+(1-alpha) *FyL_old;
    FxL_old=FxL;
    FyL_old=FyL;
```

```
    Fx_total(n)=sum(FxL);  %drag_force
    Fy_total(n)=sum(FyL);  %lift force

    [fx, fy]= spreadf(FxL, FyL, Xb, Yb, Nx, Ny, dx, dy);
    ut=ut+dt * fx/rho;
    vt=vt+dt * fy/rho;
    div=((ut(2:end-1,3:end)  -ut(2:end-1,2:end-1))/dx+(vt(3:end,2:end-1) -vt(2:end-1,2:end-1))/dy);
    rhs=zeros(Ny+2, Nx+2);
    rhs(2:end-1,2:end-1)=div/dt;
    p=solve_poisson(p, rhs, dx, dy, beta);

    % Velocity correction
    u(2:end-1,2:end-1)= ut(2:end-1,2:end-1)-dt *(p(2:end-1,3:end)-p(2:end-1,2:end-1))/dx;
    v(2:end-1,2:end-1) =vt(2:end-1,2:end-1)-dt* (p(3:end,2:end-1)-p(2:end-1,2:end-1))/dy;

    % Plot every 200 steps
    if mod(n,10) == 0
        uc=0.5 *(u(2:end-1,2:end-1) +u(2:end-1 ,3:end));
        vc=  0.5 * (v(2:end-1,2:end-1)+  v(3:end,2:end-1));
        omega=(v(2:end-1,3:end) -v(2:end-1,1:end- 2 ))/ (2*dx)-  (u(3:end,2:end-1) -u(1:end-2,2:end-1))/(2*dy);

        figure(1); clf;
        subplot(2,1,1);
        contourf(X, Y, omega, 100, 'LineColor', 'none'); colorbar();
        colormap(turbo);
        caxis([-100 100]);
        hold on; fill(cx+R*cos(theta), cy+R*sin(theta), 'c');
        title(['Vorticity at step ', num2str(n)]);
        axis equal tight;

        subplot(2,1,2);
        quiver(X(1:4:end,1:4:end), Y(1:4:end,1:4:end), uc(1:4:end,1:4:end), vc(1:4:end,1:4:end), 3);
        hold on; fill(cx+R*cos(theta), cy+R*sin(theta), 'k');
        title('Velocity Field'); axis equal tight;

        time=dt * (1:steps);
        drawnow;
    end
end

function [u,v,p]=apply_bc(u,v,p,Uinf)
    u(:,1)=Uinf; u(:,end)=u(:,end-1);
    v(:,1)=0;      v(:,end)=0;
```

```
    u(1,:)=u(2,:); u(end,:)=u(end-1,:);
    v(1,:)=0;      v(end,:)=0;
    p(:,1)=p(:,2); p(:,end)=p(:,end-1);
    p(1,:)=p(2,:); p(end,:)=p(end-1,:);
end
function u=GS_diffuse(u, nu, dt, dx, dy)
    [Ny,Nx]=size(u);
    for iter=1:50
        u_old=u;
        for j=2:Ny-1
            for i=2:Nx-1
                u(j,i)=(u_old(j,i)+dt*nu*( (u(j+1,i)+u(j-1,i)
   -2*u(j,i))/dy^2+(u(j,i+1)+u(j,i-1)-2*u(j,i))/dx^2));
            end
        end
    end
end
function [uL, vL]=vel_interpol(u, v, Xb, Yb, dx, dy)
    Nb=length(Xb);
    uL=zeros(1,Nb); vL=zeros(1,Nb);
    for k=1:Nb
        i=floor(Xb(k)/dx)+2; j=floor(Yb(k)/dy)+2;
        uL(k)=u(j,i);
        vL(k)=v(j,i);
    end
end


function [fx, fy]=spreadf(FxL, FyL, Xb, Yb, Nx, Ny, dx, dy)
    fx=zeros(Ny+2, Nx+2); fy=zeros(Ny+2, Nx+2);
    for k=1:length(Xb)
        i=floor(Xb(k)/dx)+2; j=floor(Yb(k)/dy)+2;
        fx(j,i)=fx(j,i)+FxL(k);
        fy(j,i)=fy(j,i)+FyL(k);
    end
end


function p=solve_poisson(p, rhs, dx, dy, beta)
    [Ny,Nx]=size(p);
    for it=1:200
        p_old=p;
        for j=2:Ny-1
            for i=2:Nx-1
                p(j,i)=(1-beta)*p(j,i)+beta*0.25 * (p(j+1,i)+
   p(j-1,i)+p(j,i+1)+p(j,i-1)-dx^2*rhs(j,i));
            end
        end
        if max(abs(p(:)-p_old(:))) < 1e-6
```

```
                break;
            end
        end
end
```

A.3 Verifying results of Paper

```
clc; clear;

Nx=256; Ny=64;
Lx=4; Ly=1;
dx=Lx/Nx; dy=Ly/Ny;
x=linspace(0, Lx-dx, Nx);
y=linspace(0, Ly-dy, Ny);
[X, Y]=meshgrid(x, y);
AR=Lx/Ly;

kappa=0.125;
a=kappa*Ly;
Reps=[1, 3];
Uw=0.1;
G=2*Uw/Ly;
rho=1;

theta=linspace(0, 2*pi, 100);
Nb=length(theta);
Xb0=a*cos(theta);
Yb0=a*sin(theta);


ytilda_values=linspace(-0.35, 0.35, 41); % same as Fig. 3
lift_vs_y=zeros(length(Reps), length(ytilda_values));

for r=1:length(Reps)
    Rep=Reps(r);
    nu=G*a^2/Rep;
    dt=0.1;  % LBM-compatible value
    mu=rho*nu;

    for j=1:length(ytilda_values)
        y0=(ytilda_values(j)+0.5)*Ly;
        x0=2; % center in x
        ux=zeros(Ny, Nx); uy=zeros(Ny, Nx);
        fx=zeros(Ny, Nx); fy=zeros(Ny, Nx);


        Xb=x0+Xb0;
        Yb=y0+Yb0;

        for i=1:Ny
            uy(i,:)=0;
            ux(i,:)=Uw*(y(i)-0)/Ly;
        end


        for iter=1:1000

            Ub=interpolate_vel(ux, uy, Xb, Yb, x, y, dx, dy);
```

```
            Up=mean(Ub,1);
            Omega=mean((Ub(:,1).*Yb0(:)-Ub(:,2).*Xb0(:))/a^2);

            Up_local=[Up(1)+Omega*(-Yb0(:)), Up(2)+Omega*( Xb0(:))];

            alpha=1000;  % spring stiffness
            Fb=alpha*(Up_local-Ub);

            fx(:)=0; fy(:)=0;
            [fx, fy]=spreadf(fx, fy, Fb, Xb, Yb, x, y, dx, dy);
            ux=ux+dt*fx/rho;
            uy=uy+dt*fy/rho;
        end

        lift_vs_y(r,j)=sum(Fb(:,2));
    end
end


Fl_dimless=lift_vs_y ./(rho*Uw^2*a*kappa^2); % equation taken from paper

figure;
hold on
colors=lines(length(Reps));
for i=1:length(Reps)
    plot(ytilda_values, Fl_dimless(i,:), 'LineWidth', 2, 'Color', colors(i
    ,:));
end
xlabel('$\tilde{y}_0$', 'Interpreter', 'latex')
ylabel('$\tilde{F}_L$', 'Interpreter', 'latex')
legend(arrayfun(@(r) sprintf('Re_p=%d', r), Reps, 'UniformOutput', false))
title('Lift force vs transverse position (IBM)')
grid on
function Ub=interpolate_vel(ux, uy, Xb, Yb, x, y, dx, dy)
    [Ny, Nx]=size(ux);
    Nb=length(Xb); Ub=zeros(Nb,2);
    for k=1:Nb
        i0=floor(Xb(k)/dx); j0=floor(Yb(k)/dy);
        for ii=-1:2
            for jj=-1:2
                i=mod(i0+ii-1, Nx)+1;
                j=min(max(j0+jj,1), Ny); % non-periodic in y
                phi=delta((Xb(k)-x(i))/dx)*delta((Yb(k)-y(j))/dy);
                Ub(k,1)=Ub(k,1)+ux(j,i)*phi;
                Ub(k,2)=Ub(k,2)+uy(j,i)*phi;
            end
        end
    end
end


function [fx, fy]=spreadf(fx, fy, Fb, Xb, Yb, x, y, dx, dy)
    [Ny, Nx]=size(fx);
    Nb=length(Xb);
    for k=1:Nb
        i0=floor(Xb(k)/dx); j0=floor(Yb(k)/dy);
        for ii=-1:2
            for jj=-1:2
                i=mod(i0+ii,Nx)+1;
                j=mod(j0+jj,Ny)+1;
                phi=delta((Xb(k)-x(i))/dx)*delta((Yb(k)-y(j))/dy);
```

```
                fx(j,i)=fx(j,i)+Fb(k,1)*phi*dx*dy;
                fy(j,i)=fy(j,i)+Fb(k,2)*phi*dx*dy;
            end
        end
    end
end


function val=delta(r)
    r=abs(r);
    if r < 1
        val=0.125*(3-2*r+sqrt(1+4*r-4*r^2));
    elseif r < 2
        val=0.125*(5-2*r-sqrt(-7+12*r-4*r^2));
    else
        val=0;
    end
end
```