



Indian Institute of Science
Department of Mechanical Engineering

Validation of Shear and Pressure-Driven Flows

Aswin Jayaprakash Bhat
from Indian Institute of Technology, Palakkad



IIT 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	4
1 Introduction	6
1.1 Basics of a CFD code	6
1.2 Finite Volume Method	6
1.3 Immersed Boundary Method	7
1.4 Tasks allotted	7
1.5 Research Paper validation	8
2 Literature Review	9
2.1 CFD Video Lectures by Sandip Mazumder	9
2.2 CFD Video Lectures by Tony Saad	9
2.3 Immersed Boundary Method by Charles S. Peskin	9
2.4 CFD-Video Lectures by Fluid Mechanics 101	10
3 Methodology	11
3.1 Important equations	11
3.1.1 Heat Conduction	11
3.1.2 Incompressible Flow	11
3.2 Steps in coding	11
3.2.1 Discretisation and Implementation Schemes	11
3.2.2 FVM	12
3.2.3 IBM	13
3.3 Verifying Research Paper	13
4 Results	14
4.1 FVM code results	14
4.2 IBM code results	16
4.3 Verifying Paper	19
5 Conclusions and Future Work	20
5.1 Conclusions	20
References	21
Appendices	22

A	Code	22
A.1	FVM codes	22
A.1.1	Steady-state heat conduction in 2D	22
A.1.2	Simple shear flow between parallel plates	26
A.1.3	Plane Poiseuille flow	37
A.1.4	Oscillating shear flow	42
A.1.5	Pulsating flow	47
A.2	IBM code	49
A.2.1	simple shear flow	49
A.2.2	Plane Poiseuille flow	57
A.2.3	Oscillating shear flow	65
A.2.4	Pulsating flow	69
A.2.5	Flow over a cylinder	71
A.3	Verifying results of Paper	75

List of Figures

4.1	Steady-state heat conduction in 2D	14
4.2	Simple shear flow between parallel plates	15
4.3	Plane Poiseuille flow	15
4.4	Oscillating shear flow	16
4.5	Pulsating flow	16
4.6	Simple shear flow between parallel plates using IBM	17
4.7	Plane Poiseuille flow in IBM	17
4.8	Oscillating shear flow in IBM	18
4.9	Pulsating flow in IBM	18
4.10	Flow over a cylinder	19
4.11	Lift force of cylinder v/s transverse position	19

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 pre-defined 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 allotted

We were first supposed to simulate following cases and compare our numerical results to analytical 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:

$$\begin{aligned} \delta u / \delta t + u \cdot \nabla u &= -\nabla p + \nu \nabla^2 u \\ \nabla \cdot u &= 0 \end{aligned}$$

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.

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

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 calculate the intermediate velocities.
- (b) Then we use these velocities in pressure poission equation:

$$\nabla^2 p = \frac{\rho}{\Delta t} \nabla \cdot u_n$$

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 $\nabla \cdot u = 0$. 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.

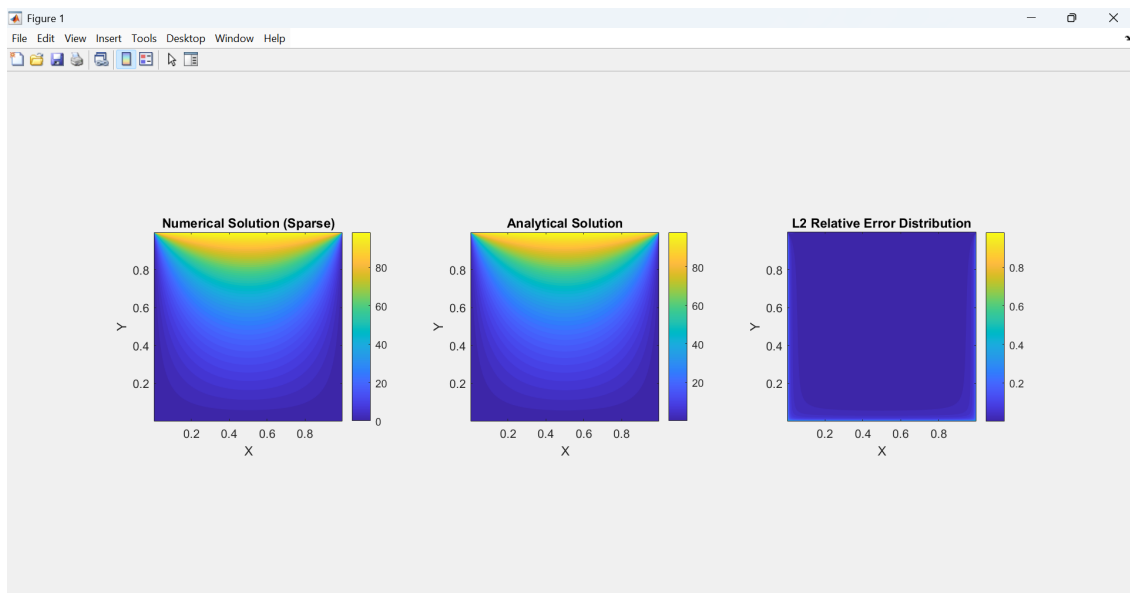


Figure 4.1: Steady-state heat conduction in 2D

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

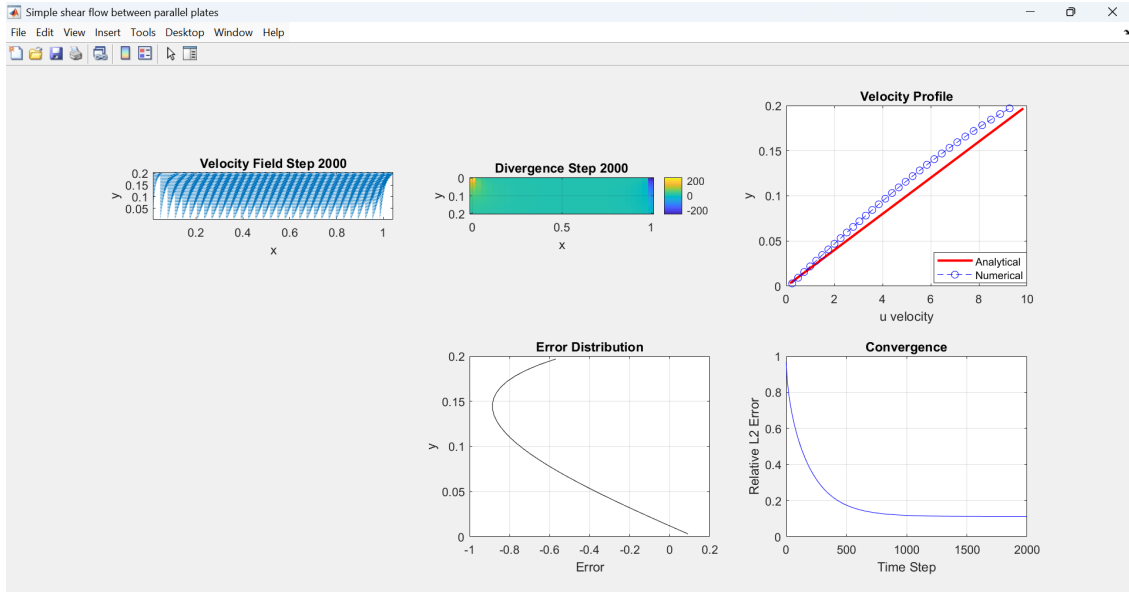


Figure 4.2: Simple shear flow between parallel plates

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.

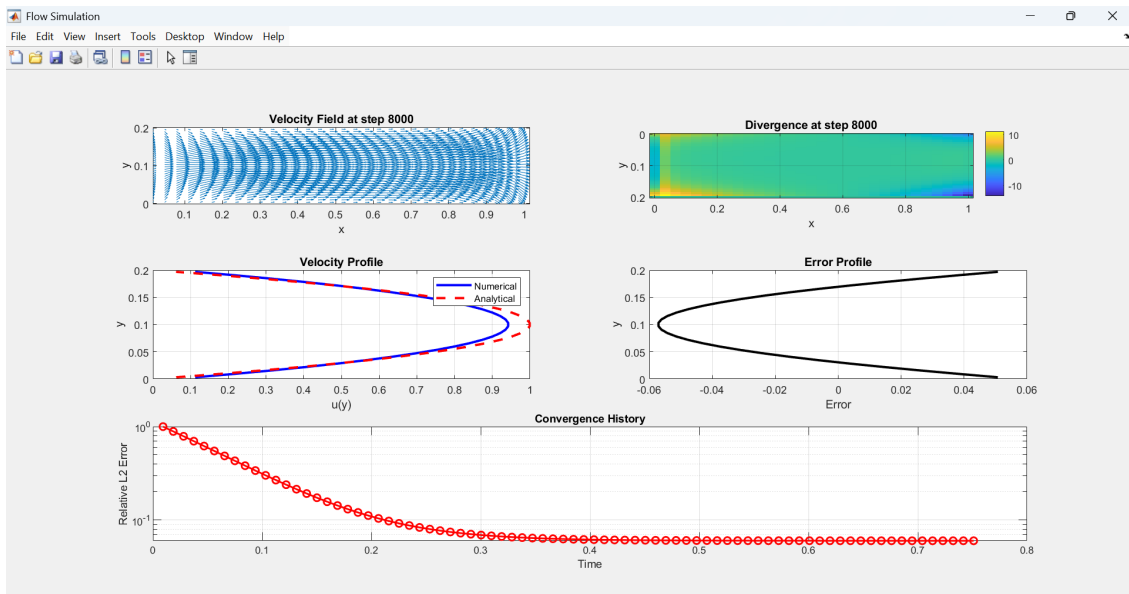


Figure 4.3: Plane Poiseuille flow

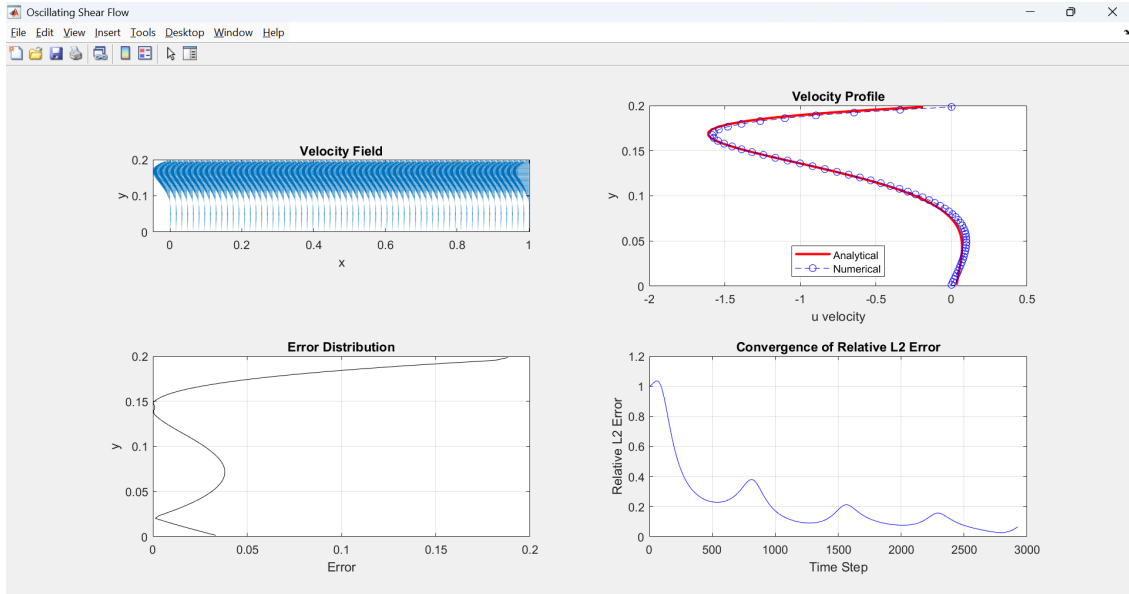


Figure 4.4: Oscillating shear flow

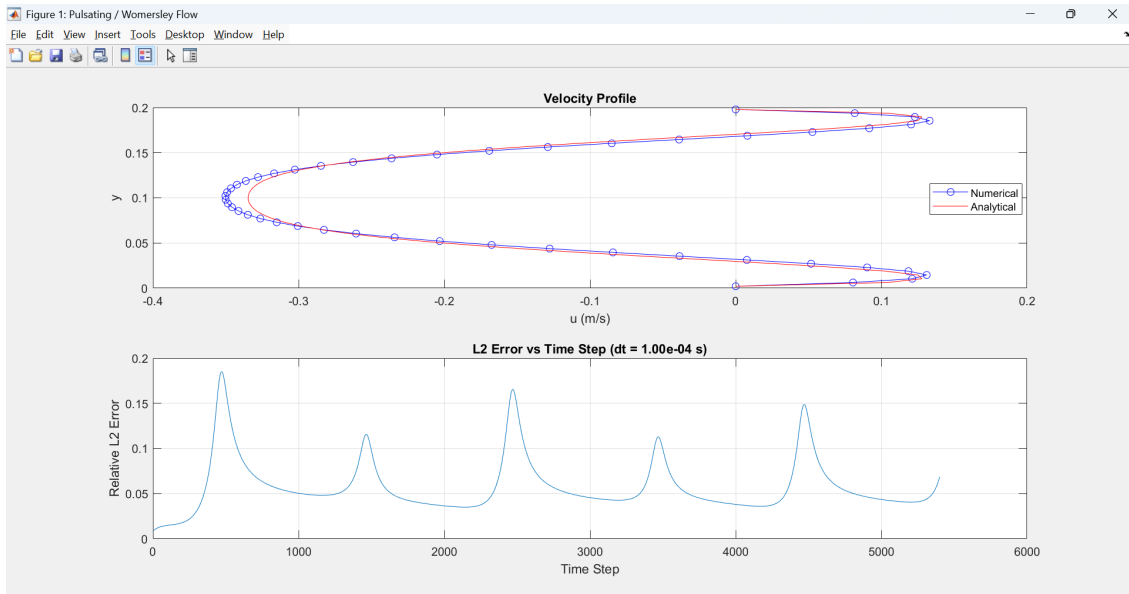


Figure 4.5: Pulsating flow

4.2 IBM code results

The same cases were 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.

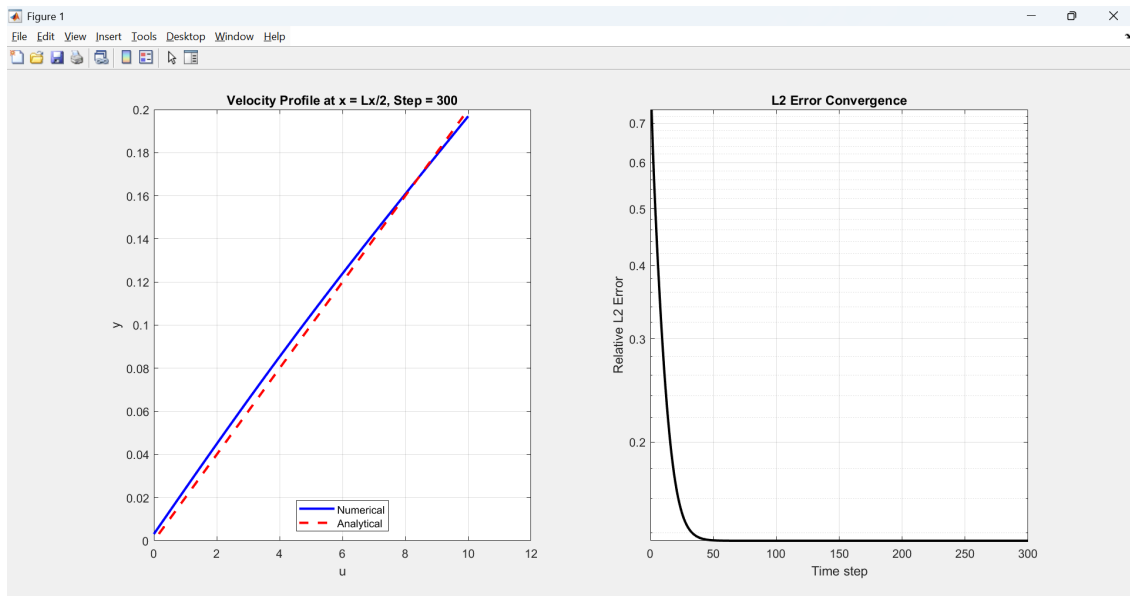


Figure 4.6: Simple shear flow between parallel plates using IBM

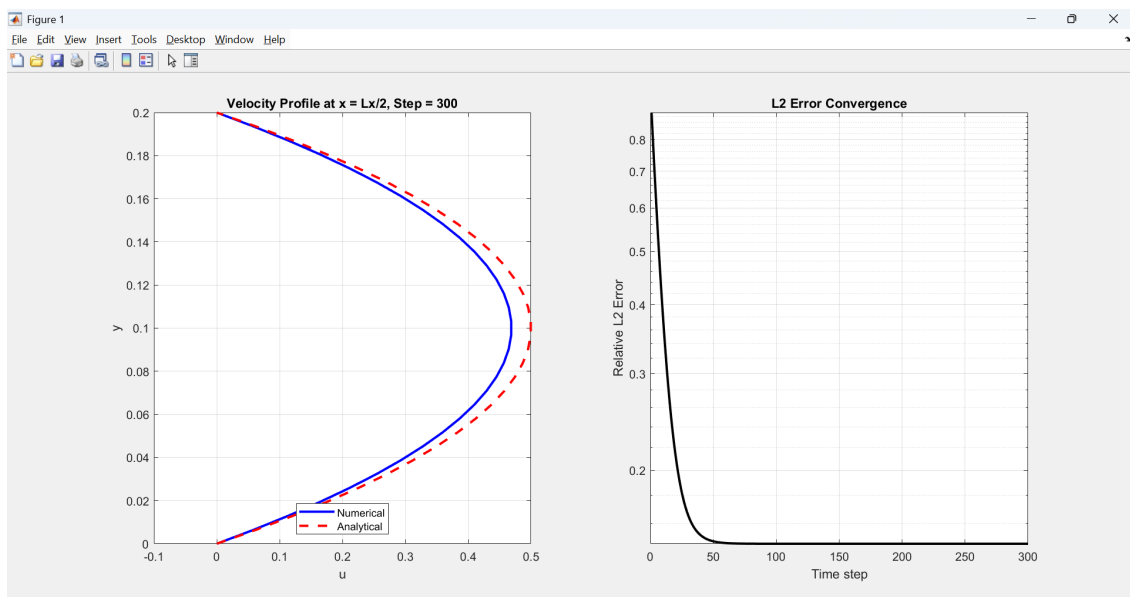


Figure 4.7: Plane Poiseuille flow in IBM

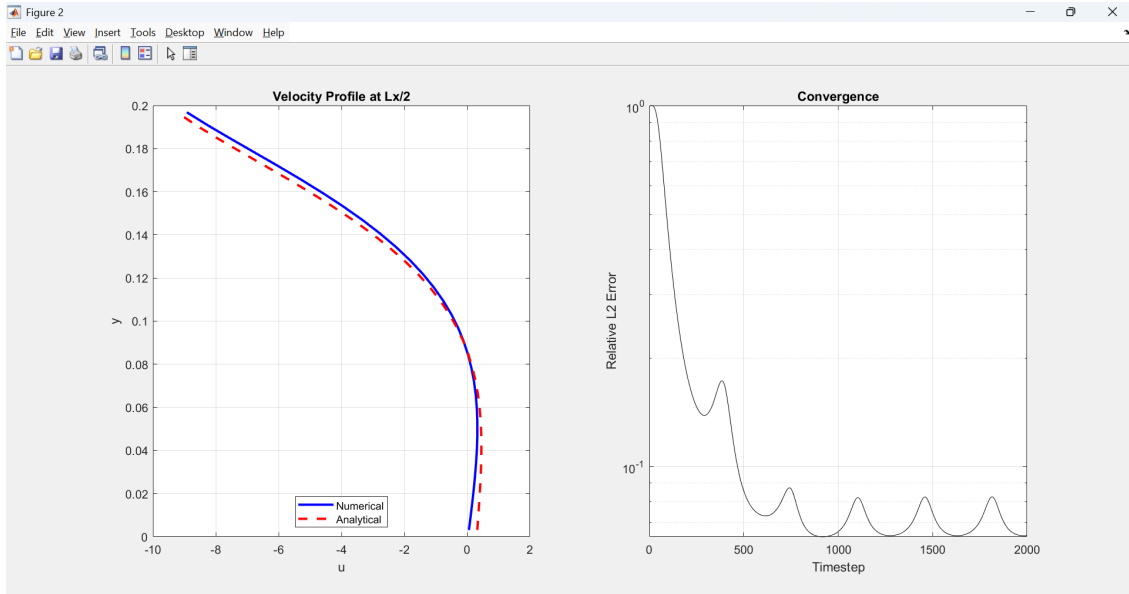


Figure 4.8: Oscillating shear flow in IBM

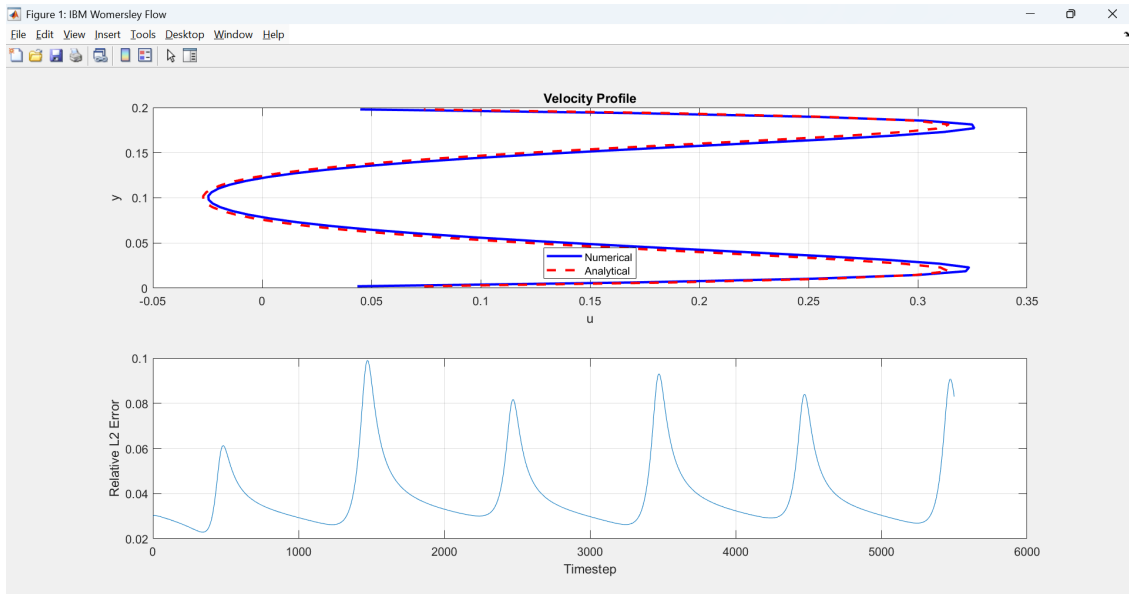


Figure 4.9: Pulsating flow in IBM

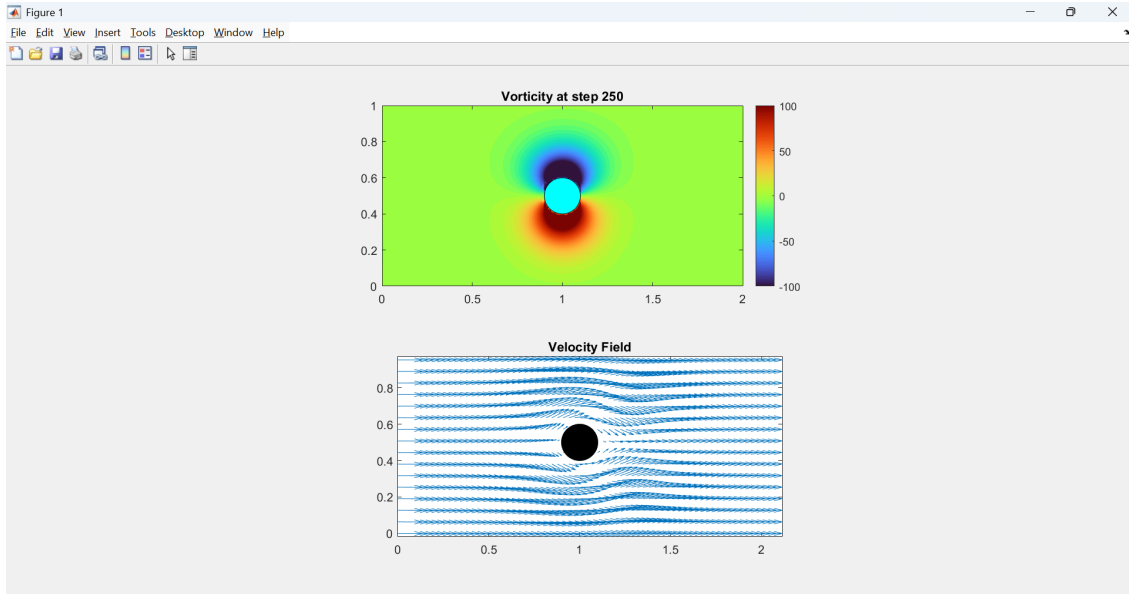


Figure 4.10: Flow over a cylinder

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

4.3 Verifying Paper

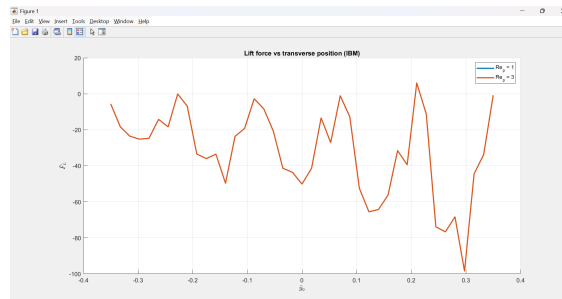


Figure 4.11: Lift force of cylinder v/s transverse position

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=PLEaLl6Sf-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

```
1 clc; clear; close all;
2 % Note:Code works only for temperatures applied to top or
   bottom wall. we have to change analytical solution if
   different BCs is applied
3
4 % Domain and Grid parameters
5 Lx = 1;
6 Ly = 1;
7 Nx = 400;
8 Ny = 400;
9 dx = Lx/Nx;
10 dy = Ly/Ny;
11
12 x = linspace(dx/2, Lx-dx/2, Nx);
13 y = linspace(dy/2, Ly-dy/2, Ny);
14 [X, Y] = meshgrid(x, y);
15
16 % Boundary Conditions (change according to user input)
17 BC.T_top = 100;
18 BC.T_bottom = 0;
19 BC.T_left = 0;
20 BC.T_right = 0;
21
22 % Source Term (Poisson Equation)
23 % Set f = 0 for Laplace
24 f = zeros(Ny, Nx);
25
26 % Solver Options
27 solver_type = 'Sparse'; % SOR or Sparse_solver
28
29 % Solver Parameters
30 omega = 2 / (1 + sin(pi/max(Nx, Ny))); % Optimal omega
```

```

31 max_iter = 10000;
32 tol = 1e-12;
33
34 % Solve
35 if strcmpi(solver_type, 'SOR')
36     [numerical_temp, iteration_count, error_history] =
        SOR_solver(f, BC, dx, dy, omega, max_iter, tol);
37 elseif strcmpi(solver_type, 'Sparse')
38     numerical_temp = Sparse_solver(f, BC, dx, dy);
39     error_history = [];
40     iteration_count = 1;
41 else
42     error('Unknown solver type. Choose "SOR" or "Sparse".');
43 end
44
45
46 % Analytical Solution
47 analytical_temp = zeros(Ny, Nx);
48 N_terms = 100; % Number of sine terms
49
50 for n = 1:2:(2*N_terms-1) % Only odd terms
51     lambda = n*pi;
52     term = (4*BC.T_top)/(n*pi) *sinh(n*pi*Y) ./ sinh(n*pi*Ly)
        .*sin(n*pi*X);
53     analytical_temp = analytical_temp + term;
54 end
55
56 %Error Calc
57
58 error_abs = abs(numerical_temp - analytical_temp);
59 L2_error = sqrt(sum(error_abs(:).^2)) / sqrt(sum(
        analytical_temp(:).^2));
60
61 fprintf('L2 Relative Error = %.6e\n', L2_error);
62
63 % Plotting (Includes error plot)
64
65 figure('Position',[100 100 1200 400]);
66
67 subplot(1,3,1);
68 contourf(X, Y, numerical_temp, 50, 'LineColor','none');
69 colorbar;
70 title(['Numerical Solution (' solver_type ')']);
71 xlabel('X');
72 ylabel('Y');
73 axis equal tight;
74
75 subplot(1,3,2);
76 contourf(X, Y, analytical_temp, 50, 'LineColor','none');

```



```

77 colorbar;
78 title('Analytical Solution');
79 xlabel('X');
80 ylabel('Y');
81 axis equal tight;
82
83 subplot(1,3,3);
84 contourf(X, Y, error_abs ./ abs(analytical_temp + eps), 50, '
    LineColor','none');
85 %+eps to avoid divide by zero
86 colorbar;
87 title('L2 Relative Error Distribution');
88 xlabel('X');
89 ylabel('Y');
90 axis equal tight;
91
92
93 % Plot Convergence History
94 if strcmpi(solver_type, 'SOR')
95     figure;
96     semilogy(error_history, '-o');
97     grid on;
98     xlabel('Iteration');
99     ylabel('Max Error');
100    title('SOR Convergence History');
101 end
102
103
104
105 % Helper Functions
106
107 % Dirichlet BCs (for neumann different analytical solution)
108 function T = apply_BC(T, BC)
109     T(1,:) = BC.T_top;
110     T(end,:) = BC.T_bottom;
111     T(:,1) = BC.T_left;
112     T(:,end) = BC.T_right;
113 end
114
115 % SOR Solver
116 function [T, iter, error_history] = SOR_solver(f, BC, dx, dy,
    omega, max_iter, tol)
117     [Ny, Nx] = size(f);
118     T = zeros(Ny, Nx);
119     T = apply_BC(T, BC);
120
121     dx2 = dx^2;
122     dy2 = dy^2;
123     coeff = 1/(2*(dx2 + dy2));

```

```

124
125     error_history = [];
126
127     for iter = 1:max_iter
128         T_old = T;
129         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 );
130         T = apply_BC(T, BC);
131
132         % Error Check
133         err = max(max(abs(T - T_old)));
134         error_history = [error_history; err];
135
136         if err < tol
137             fprintf('SOR converged in %d iterations with
error %.3e\n', iter, err);
138             break
139         end
140     end
141
142     if iter == max_iter
143         fprintf('SOR reached max iterations (%d) with error
%.3e\n', iter, err);
144     end
145 end
146
147 %Sparse Matrix Solver
148 function T = Sparse_solver(f, BC, dx, dy)
149     [Ny, Nx] = size(f);
150     N = Ny * Nx;
151     dx2 = dx^2;
152     dy2 = dy^2;
153
154     % Sparse Matrix Assembly
155     main_diag = -2*(1/dx2 + 1/dy2) * ones(N,1);
156     off_diag_x = 1/dx2 * ones(N,1);
157     off_diag_y = 1/dy2 * ones(N,1);
158
159     A = spdiags([off_diag_y, off_diag_x, main_diag,
off_diag_x, off_diag_y], [-Nx,-1,0,1,Nx], N, N);
160     b = -reshape(f,[],1);
161
162     % applying Dirichlet BCs
163     top_idx = (Ny-1)*Nx + (1:Nx);
164     bottom_idx = (0)*Nx + (1:Nx);
165     left_idx = ((Ny-1):-1:0)*Nx + 1;
166     right_idx = ((Ny-1):-1:0)*Nx + Nx;

```

```

167
168     A(top_idx,:)      = 0; A(sub2ind(size(A), top_idx, top_idx)
    ) = 1; b(top_idx) = BC.T_top;
169     A(bottom_idx,:)  = 0; A(sub2ind(size(A), bottom_idx,
    bottom_idx)) = 1; b(bottom_idx) = BC.T_bottom;
170     A(left_idx,:)    = 0; A(sub2ind(size(A), left_idx,
    left_idx)) = 1; b(left_idx) = BC.T_left;
171     A(right_idx,:)   = 0; A(sub2ind(size(A), right_idx,
    right_idx)) = 1; b(right_idx) = BC.T_right;
172
173     T_vec = A\b;
174
175     % Reshape to 2D
176     T = reshape(T_vec, [Nx, Ny])';
177 end

```

A.1.2 Simple shear flow between parallel plates

```

1 clc; clear; close all;
2
3 % Domain and grid setup
4 x= 32;
5 y = 32;
6 Lx =1;
7 Ly = 0.2;
8 dx = Lx/x;
9 dy =Ly / y;
10 visc = 0.1;
11
12 % Boundary conditions
13 Ut = 10; % Top wall velocity
14 Ub = 0; % Bottom wall velocity
15
16 % Time step based on stability
17 CFL =0.3;
18 u_max = max(abs([Ut, Ub]));
19 dt1 =1e6;
20 dt2 = CFL * min(dx, dy) / u_max;
21 dt = min(dt1, dt2)
22 % dt = 0.000025; % chosen small for stability
23
24 % Preallocation
25 p = zeros(y+2, x+2);
26 u = zeros(y+2, x+2);
27 v =zeros(y+2, x+2);
28 ut =zeros(y+2, x+2);
29 vt= zeros(y+2, x+2);
30 divut =zeros(y+2, x+2);
31

```

```

32 % Analytical solution for simple shear flow (linear profile)
33 ya = linspace(dy/2,Ly -dy/2,y);
34 ua = (Ut -Ub)/Ly* ya +Ub;
35
36
37 %% Mesh for plotting
38 [X, Y] = meshgrid(dx/2:dx:Lx - dx/2, dy/2:dy:Ly - dy/2);
39
40 %% Setup figure with subplots
41 figure('Name', 'Simple shear flow between parallel plates', '
    NumberTitle', 'off');
42
43 subplot(2,3,1);
44 hQuiver = quiver(X, Y, zeros(size(X)), zeros(size(Y)), '
    AutoScaleFactor', 3);
45 title('Velocity Field');
46 xlabel('x'); ylabel('y'); axis equal tight;
47
48 subplot(2,3,2);
49 hIm = imagesc(linspace(0,Lx,x), linspace(0,Ly,y), zeros(y,x))
    ;
50 colorbar;
51 title('Divergence');
52 xlabel('x'); ylabel('y');
53 axis equal tight; grid on;
54
55 subplot(2,3,3);
56 hProfile = plot(ua, ya, 'r-', 'LineWidth', 2); hold on;
57 hNumerical = plot(ua*0, ya, 'bo--');
58 title('Velocity Profile');
59 xlabel('u velocity'); ylabel('y');
60 legend('Analytical','Numerical');
61 grid on;
62
63 subplot(2,3,5);
64 hError = plot(zeros(y,1), ya, 'k');
65 title('Error Distribution');
66 xlabel('Error'); ylabel('y');
67 grid on;
68
69 subplot(2,3,6);
70 hConv = plot(0, 0, 'b');
71 title('Convergence of Relative L2 Error');
72 xlabel('Time Step'); ylabel('Relative L2 Error');
73 grid on;
74
75
76
77

```

```

78 %% %% Setup video writer
79 % k = VideoWriter('CouetteFlowSimulation.mp4', 'MPEG-4'); %
    Name of video file
80 % k.FrameRate = 10; % Frames per second
81 % open(k);
82
83
84
85 %% Time-stepping loop
86 tsteps = 2000;
87 L2_Err_hist = zeros(tsteps, 1);
88
89 for n = 1:tsteps
90
91     [u, v, ~] = apply_bcs(u, v, Ut, Ub, p);
92
93     % X momentum
94     for i = 3:x+1
95         for j = 2:y+1
96             ue = 0.5 * (u(j, i+1) + u(j, i));
97             uw = 0.5 * (u(j, i) + u(j, i-1));
98             un = 0.5 * (u(j+1, i) + u(j, i));
99             us = 0.5 * (u(j, i) + u(j-1, i));
100             vn = 0.5 * (v(j+1, i-1) + v(j+1, i));
101             vs = 0.5 * (v(j, i-1) + v(j, i));
102
103             if n > 50
104                 convection = 0;
105             else
106                 convection = -(ue^2 - uw^2)/dx - (un*vn - us*vs)/dy;
107             end
108
109             diffusion = visc * ((u(j, i-1) - 2*u(j, i) + u(j,
110                 i+1))/dx^2 + ...
111                 (u(j-1, i) - 2*u(j, i) + u(j
112                 +1, i))/dy^2);
113
114             ut(j, i) = u(j, i) + dt * (convection + diffusion
115             );
116         end
117     end
118
119     % Y momentum
120     for i = 2:x+1
121         for j = 3:y+1
122             ve = 0.5 * (v(j, i+1) + v(j, i));
123             vw = 0.5 * (v(j, i) + v(j, i-1));
124             ue = 0.5 * (u(j, i+1) + u(j-1, i+1));
125             uw = 0.5 * (u(j, i) + u(j-1, i));

```

```

123         vn = 0.5 * (v(j+1, i) + v(j, i));
124         vs = 0.5 * (v(j, i) + v(j-1, i));
125
126         if n > 50
127             convection = 0;
128     else
129         convection = -(ue*ve - uw*vw)/dx - (vn^2 - vs^2)/dy;
130         end
131         diffusion = visc * ((v(j, i+1) - 2*v(j, i) + v(j,
132             i-1))/dx^2 + ...
133             (v(j+1, i) - 2*v(j, i) + v(j
134             -1, i))/dy^2);
135         vt(j, i) =v(j, i) + dt *(convection +diffusion);
136     end
137     end
138     % pressre correction
139     rho = 1;
140     divut(2:end-1, 2:end-1) = (ut(2:end-1,3:end) - ut(2:end
141     -1,2:end-1))/dx + ...
142     (vt(3:end,2:end-1) - vt(2:end
143     -1,2:end-1))/dy;
144
145     rhs = rho * divut / dt;
146
147     [p, ~] = fmg_solver(rhs, Lx, Ly, x, y);
148
149     [~,~,p]=apply_bcs(u, v, Ut, Ub,p);
150
151     %corner smoothing
152     u(end,end) = mean([u(end-1,end), u(end,end-1)]);
153     v(end,end) = mean([v(end-1,end), v(end,end-1)]);
154
155     % Bottom-right corner
156     u(1,end) = mean([u(2,end), u(1,end-1)]);
157     v(1,end) = mean([v(2,end), v(1,end-1)]);
158
159     % Top-left corner
160     u(end,1) = mean([u(end-1,1), u(end,2)]);
161     v(end,1) = mean([v(end-1,1), v(end,2)]);
162
163     % Bottom-left corner
164     u(1,1) = mean([u(2,1), u(1,2)]);
165     v(1,1) = mean([v(2,1), v(1,2)]);
166
167     p(2,2)=0; % Set reference pressure point to zero to
168     prevent accidental existence of pressure gradients by only

```

```

    enforcing neumann equations
167
168 %velocity correction
169 [u, v] = rhie_chow_correction(ut, vt, p, dx, dy, dt);
170
171 % --- Under-relaxation ---
172 alpha = 0.4;
173
174
175 u= alpha* u +(1- alpha) *ut;
176 v = alpha*v + (1 -alpha)* vt;
177
178
179 % to display the velocity at the geometric center of the
    cell
180
181 uc = 0.5*(u(2:end-1,2:end-1)+ u(2:end-1, 3:end));
182 vc = 0.5 *(v(2:end-1,2:end-1) +v(3:end,2:end-1));
183
184 u_profile =mean(uc, 2); % average across x-direction
185
186 % Error
187 error_profile =u_profile -ua';
188 L2_error= sqrt(sum(error_profile.^2) /length(
error_profile)) / ...
189     sqrt(sum(ua.^2) /length(ua));
190 L2_Err_hist(n) = L2_error;
191
192
193
194 if mod(n,10) == 0
195     set(hQuiver, 'UData', uc, 'VData', vc);
196     set(hIm, 'CData', flipud(divut(2:end-1,2:end-1)));
197
198     set(hNumerical, 'XData', u_profile, 'YData', ya);
199     set(hError, 'XData', error_profile, 'YData', ya);
200     set(hConv, 'XData', 1:n, 'YData', L2_Err_hist(1:n));
201
202     subplot(2,3,1); title(['Velocity Field Step ',
num2str(n)]);
203     subplot(2,3,2); title(['Divergence Step ', num2str(n)
]);
204     subplot(2,3,3); title('Velocity Profile');
205     subplot(2,3,5); title('Error Distribution');
206     subplot(2,3,6); title('Convergence');
207
208     drawnow;
209
210     % % Capture frame for video

```

```

211         % frame = getframe(gcf);
212         % writeVideo(k, frame);
213     end
214
215 end
216
217
218 % %% Close video file
219 % close(k);
220 % disp('Video saved successfully as CouetteFlowSimulation.mp4
    ');
221
222 fprintf('Max divergence: %.2e\n',max(abs(divut(:)))));
223
224
225 function [u, v,p] = apply_bcs(u, v,Ut,Ub,p)
226     % Left wall
227     u(:,1) = u(:,2);
228     v(:,1) = 0;
229
230     % Right wall
231     u(:,end) = u(:,end-1);
232     v(:,end) = 0;
233
234     % Top wall (Moving wall)
235     u(end,:) = Ut;
236     v(end,:) = 0;
237
238     % Bottom wall (Fixed wall)
239     u(1,:) = Ub;
240     v(1,:) = 0;
241
242
243     u(:,end) = u(:,end-1);
244     v(:,end) = v(:,end-1); % convective (zero-gradient)
    outflow BCs on velocity
245
246     % Apply Neumann BCs on pressure
247     p(:,1) = p(:,2);
248     p(:,end) = p(:,end-1);
249     p(1,:) = p(2,:);
250     p(end,:) = p(end-1,:);
251
252 end
253
254
255 function [p,err]=sor_solver(p, S,Lx,Ly,x, y)
256     dx=Lx/x ;
257     dy=Ly/y ;

```



```

258     Ae = ones(y+2, x+2) / dx^2;
259     Aw = ones(y+2, x+2) / dx^2;
260     An = ones(y+2, x+2) / dy^2;
261     As = ones(y+2, x+2) / dy^2;
262     Ap=-(Ae+Aw+An+As);
263
264     it = 0;
265     err = 1e10;
266     tol = 1e-8;
267     maxit=1000;
268     B = 1.9; % between 1 and 2
269
270     while err > tol && it < maxit
271         pk = p;
272         for i =2:x+1
273             for j =2:y+1
274                 ap = Ap(j,i); ae = Ae(j,i); aw = Aw(j,i); an =
An(j,i); as = As(j,i);
275
276                 pe = p(j,i+1); pw = p(j,i-1); pn = p(j+1,i);
ps = p(j-1,i);
277
278                 res = S(j,i) - (ae*pe + aw*pw + an*pn + as*ps)
;
279                 p(j,i) = B * res / ap + (1-B) * pk(j,i);
280             end
281         end
282         u = zeros(y+2, x+2);
283         v = zeros(y+2, x+2);
284         Ut=10;
285         Ub=0;
286         [~,~,p]=apply_bcs(u, v, Ut, Ub,p); % applying
pressure BCs to prevent pressure drift in Neumann BCs
problems
287         err = norm(p(:) - pk(:), 2);
288         it = it+1;
289     end
290 end
291
292
293
294 function [p, err] = multigrid_solver(p, rhs, Lx, Ly, Nx, Ny)
295     max_iter = 100;
296     tol = 1e-8;
297     err = 1e10;
298
299     for iter = 1:max_iter
300         p_old = p;
301         p = V_cycle(p, rhs, Lx, Ly, Nx, Ny);

```

```

302         res = res_computing(p, rhs, Lx, Ly, Nx, Ny);
303         err = norm(res(:), 2);
304
305         if err < tol
306             break;
307         end
308     end
309 end
310
311 % V-Cycle Function
312 function p = V_cycle(p, rhs, Lx, Ly, Nx, Ny)
313     if Nx <= 6 || Ny <= 6
314         p = sor_solver_local(p, rhs, Lx, Ly, Nx, Ny, 100);
315         return;
316     end
317     p = sor_solver_local(p, rhs, Lx, Ly, Nx, Ny, 10);
318     res = res_computing(p, rhs, Lx, Ly, Nx, Ny);
319     res_coarse = restrict(res);
320
321     Nc_x = size(res_coarse,2) - 2;
322     Nc_y = size(res_coarse,1) - 2;
323     e_coarse = zeros(Nc_y + 2, Nc_x + 2);
324
325     e_coarse = V_cycle(e_coarse, res_coarse, Lx, Ly, Nc_x,
Nc_y);
326
327     e_fine = prolong(e_coarse, Nx, Ny);
328     p = p + e_fine;
329
330     % Post-smoothing
331     p = sor_solver_local(p, rhs, Lx, Ly, Nx, Ny, 7);
332 end
333
334 function p = sor_solver_local(p, rhs, Lx, Ly, Nx, Ny, Niter)
335     if Nx < 2 || Ny < 2
336         warning('SOR skipped due to small grid');
337         return;
338     end
339
340     dx = Lx / Nx;
341     dy = Ly / Ny;
342     B = 1.9;
343
344     for iter = 1:Niter
345         p_old = p;
346         for i = 2:Nx+1
347             for j = 2:Ny+1
348                 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)));
349         end
350     end
351
352     % Neumann BCs
353     p(:,1) = p(:,2);
354     p(:,end) = p(:,end-1);
355     p(1,:) = p(2,:);
356     p(end,:) = p(end-1,:);
357
358     if norm(p(:) - p_old(:), 2) < 1e-8
359         break;
360     end
361 end
362 end
363
364
365
366
367 function res = res_computing(p, rhs, Lx, Ly, Nx, Ny)
368     dx = Lx / Nx;
369     dy = Ly / Ny;
370
371     res = zeros(Ny+2, Nx+2);
372     for i = 2:Nx+1
373         for j = 2:Ny+1
374             laplace = (p(j,i+1) - 2*p(j,i) + p(j,i-1)) / dx^2
375             + ...
376                 (p(j+1,i) - 2*p(j,i) + p(j-1,i)) / dy
377             ^2;
378             res(j,i) = rhs(j,i) - laplace;
379         end
380     end
381 end
382
383 function coarse = restrict(fine)
384     [Nyf, Nxf] = size(fine);
385     Nxc = ceil((Nxf - 2)/2);
386     Nyc = ceil((Nyf - 2)/2);
387
388     coarse = zeros(Nyc+2, Nxc+2);
389     for i = 2:Nxc+1
390         for j = 2:Nyc+1
391             i_f = 2*(i-1);
392             j_f = 2*(j-1);
393
394             neighbors = fine(j_f-1:j_f+1, i_f-1:i_f+1);
395             weights = [1 2 1; 2 4 2; 1 2 1];

```

```

395
396         % Handle edges
397         valid = ~isnan(neighbors);
398         w_sum = sum(weights(valid));
399
400         coarse(j,i) = sum(neighbors(valid) .* weights(
valid), 'all') / w_sum;
401     end
402 end
403 end
404
405 % Prolongation
406 function fine = prolong(coarse, Nxf, Nyf)
407     Nxc = size(coarse, 2) - 2;
408     Nyc = size(coarse, 1) - 2;
409
410     fine = zeros(Nyf+2, Nxf+2);
411
412     for i = 2:Nxc+1
413         for j = 2:Nyc+1
414             i_f = 2 * (i - 1);
415             j_f = 2 * (j - 1);
416
417             % Safely assign values to fine grid
418             if j_f <= Nyf && i_f <= Nxf
419                 fine(j_f, i_f) = fine(j_f, i_f) +
coarse(j, i);
420             end
421             if j_f + 1 <= Nyf && i_f <= Nxf
422                 fine(j_f+1, i_f) = fine(j_f+1, i_f) +
coarse(j, i);
423             end
424             if j_f <= Nyf && i_f + 1 <= Nxf
425                 fine(j_f, i_f+1) = fine(j_f, i_f+1) +
coarse(j, i);
426             end
427             if j_f + 1 <= Nyf && i_f + 1 <= Nxf
428                 fine(j_f+1, i_f+1) = fine(j_f+1, i_f+1) +
coarse(j, i);
429             end
430         end
431     end
432
433     % average overlapping contributions
434     fine(2:end-1, 2:end-1) = fine(2:end-1, 2:end-1) / 4;
435 end
436
437
438 function [u_corr, v_corr] = rhie_chow_correction(ut, vt, p,

```

```

dx, dy, dt)
439 % applyng Rhie-Chow interpolation based velocity
correction
440 [Ny, Nx] = size(p);
441 u_corr = ut;
442 v_corr = vt;
443
444 % u correction
445 u_corr(2:end-1,3:end-1) = ut(2:end-1,3:end-1) - ...
446 dt * (p(2:end-1,3:end-1) - p(2:end-1,2:end-2)) / dx;
447
448 % vcorrection
449 v_corr(3:end-1,2:end-1) = vt(3:end-1,2:end-1) - ...
450 dt * (p(3:end-1,2:end-1) - p(2:end-2,2:end-1)) / dy;
451 end
452
453 % FMG Solver
454 function [p, err] = fmg_solver(rhs, Lx, Ly, Nx, Ny)
455 levels = floor(log2(min(Nx, Ny))) - 1;
456 levels = min(levels, 5); % Optional hard cap to avoid
too deep levels
457 if levels < 1
458 warning('FMG: Too few grid levels, using regular
multigrid.');
```

```

459 [p, err] = multigrid_solver(zeros(size(rhs)), rhs, Lx
, Ly, Nx, Ny);
460 return;
461 end
462
463 % Coarsest grid size
464 Nc_x = floor(Nx / 2^(levels - 1));
465 Nc_y = floor(Ny / 2^(levels - 1));
466
467 if Nc_x < 3 || Nc_y < 3
468 [p, err] = multigrid_solver(zeros(size(rhs)), rhs, Lx
, Ly, Nx, Ny);
469 return;
470 end
471
472 % Construct RHS at coarsest level
473 rhs_c = rhs;
474 for l = 1:(levels - 1)
475 rhs_c = restrict(rhs_c);
476 end
477 p_c = zeros(size(rhs_c));
478 [Nc_y_full, Nc_x_full] = size(rhs_c);
479 p_c = sor_solver_local(p_c, rhs_c, Lx, Ly, Nc_x_full - 2,
Nc_y_full - 2, 100);
480

```

```

481     for l = (levels - 1):-1:0
482
483         Nxf = floor(Nx / 2^l);
484         Nyf = floor(Ny / 2^l);
485         p_f = prolong(p_c, Nxf, Nyf);
486
487
488         rhs_f = rhs;
489         for li = 1:l
490             rhs_f = restrict(rhs_f);
491         end
492
493         p_c = V_cycle(p_f, rhs_f, Lx, Ly, Nxf, Nyf);
494     end
495
496     p = p_c;
497
498     res = res_computing(p, rhs, Lx, Ly, Nx, Ny);
499     err = norm(res(:), 2);
500 end

```

A.1.3 Plane Poiseuille flow

```

1  clc; clear; close all
2
3  % grid and domain properties while using staggered grid
4  x=32; y=32;
5  Lx=1; Ly=0.2;
6  dx=Lx/x; dy=Ly/y;
7  visc=0.05;
8
9  %top and Bottom wall fixed
10 Ut=0; Ub=0;
11
12 % Time step selection based on stability criteria
13 dt1 = 0.5/(visc*(1/(dx^2) + 1/(dy^2)));
14 CFL = 0.5;
15 u_max = max(abs([Ut, Ub]));
16 if u_max == 0
17     dt2 = Inf;
18 else
19     dt2 = CFL * min(dx/u_max, dy/u_max);
20 end
21 dt = min(dt1, dt2);
22 dt = 0.25 * dt; % safety margin
23 dp_x = 0.1; % pressure gradient
24
25 % Preallocate fields
26 p = zeros(y+2, x+2);

```

```

27 u = zeros(y+2, x+2);
28 v = zeros(y+2, x+2);
29 ut = zeros(y+2, x+2);
30 vt = zeros(y+2, x+2);
31 divut = zeros(y+2, x+2);
32
33 % Cell center velocities and meshgrid
34 uc= 0.5*(u(2:end-1, 2:end-1) +u(2:end-1,3:end));
35 vc =0.5*(v(2:end-1,2:end-1)+ v(3:end,2:end-1));
36 [X,Y] = meshgrid(dx/2:dx:Lx-dx/2, dy/2:dy:Ly-dy/2);
37
38 fig = figure('Name','Flow Simulation','NumberTitle','off');
39
40 subplot(3,2,1);
41 hQuiver = quiver(X,Y,uc,vc,'AutoScaleFactor',3);
42 title('Velocity Field'); xlabel('x'); ylabel('y'); axis equal
    tight;
43
44 % Initial divergence
45 for i=2:x+1
46     for j=2:y+1
47         divu(j,i) =(u(j,i)-u(j,i-1))/dx    +(v(j,i)-v(j-1,i))/
            dy;
48     end
49 end
50 subplot(3,2,2);
51 hIm = imagesc(linspace(0,Lx,x), linspace(0,Ly,y), flipud(divu
    (2:end-1,2:end-1)));
52 colorbar; title('Divergence of Velocity'); xlabel('x');
    ylabel('y'); axis equal tight; grid on;
53
54 t = 0; tsteps = 8000;
55 err_hist = []; t_hist = [];
56 % k = VideoWriter('poiselle_flow_vid.mp4', 'MPEG-4'); k.
    FrameRate = 20; open(k);
57 err_initial = NaN;
58
59 for n = 1:tsteps
60     [u,v,~] = apply_boundary_conditions(u,v,p,dpx,dx);
61
62     % X-momentum
63     for i = 3:x+1
64         for j = 2:y+1
65             ue = 0.5*(u(j,i+1)+u(j,i));
66             uw = 0.5*(u(j,i) +u(j,i-1));
67             un = 0.5*(u(j+1,i)+u(j,i));
68             us = 0.5*(u(j,i) +u(j-1,i));
69             vn = 0.5*(v(j+1,i-1)+v(j+1,i));
70             vs = 0.5*(v(j,i-1) +v(j,i));

```

```

71         convection = -(ue^2- uw^2)/dx-(un*vn- us*vs)/dy;
72         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);
73         pressure_source = dpx;
74         rho=0.01;
75         ut(j,i) = u(j,i)+ dt*(convection+ diffusion+ (
pressure_source/ rho));
76     end
77 end
78
79 % Y-momentum
80 for i = 2:x+1
81     for j = 3:y+1
82         ve = 0.5*(v(j,i+1) +v(j,i));
83         vw = 0.5*(v(j,i)  +v(j,i-1));
84         ue = 0.5*(u(j,i+1) +u(j-1,i+1));
85         uw = 0.5*(u(j,i)  +u(j-1,i));
86         vn = 0.5*(v(j+1,i) +v(j,i));
87         vs = 0.5*(v(j,i)  +v(j-1,i));
88         convection = - (ue*ve-uw*vw)/dx -(vn^2-vs^2)/dy;
89         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);
90         vt(j,i) = v(j,i) + dt*(convection+diffusion);
91     end
92 end
93
94
95     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;
96     rhs = rho * divut / dt;
97
98     [p,~] = sor_solver(p,rhs,Lx,Ly,x,y);
99
100
101     [~,~,p] = apply_boundary_conditions(u,v,p,dpx,dx);
102
103     % Velocity correction
104     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;
105     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;
106
107     for i=2:x+1
108         for j=2:y+1
109             divu(j,i) =(u(j,i)-u(j,i-1))/dx +(v(j,i)-v(j-1,i)
)/dy;
110         end
111     end
112

```



```

113     uc = 0.5* (u(2:end-1,2:end-1) +u(2:end-1,3:end));
114     vc= 0.5*(v(2:end-1,2:end-1) +v(3:end,2:end-1));
115
116     if mod(n,100)==0 || n==0
117         subplot(3,2,1);
118         set(hQuiver,'UData',uc,'VData',vc);
119         title(['Velocity Field at step ',num2str(n)]);
120
121         subplot(3,2,2);
122         set(hIm,'CData',flipud(divu(2:end-1,2:end-1)));
123         title(['Divergence at step ',num2str(n)]);
124
125         ya = dy/2 : dy: Ly-dy/2;
126         ua = (dpx/(2*visc)) *ya .*(Ly - ya)/rho;
127         x_phys = linspace(dx/2,Lx-dx/2, x);
128         [~, mid_idx] = min(abs(x_phys - Lx/2));
129         u_num = uc(:,mid_idx);
130
131         subplot(3,2,3);
132         plot(u_num', ya, 'b-', ua, ya, 'r--', 'LineWidth',2);
133         ylabel('y'); xlabel('u(y)'); legend('Numerical','
Analytical');
134         title('Velocity Profile'); grid on;
135
136         err_profile = u_num - ua';
137         subplot(3,2,4);
138         plot(err_profile, ya, 'k-', 'LineWidth', 2);
139         ylabel('y'); xlabel('Error'); title('Error Profile');
grid on;
140
141         L2_err = norm(err_profile, 2);
142         if isnan(err_initial)
143             err_initial =L2_err;
144         end
145         err_rel = L2_err /err_initial;
146         err_hist(end+1) = err_rel;
147         t_hist(end+1) = t;
148
149         subplot(3,2,[5 6]);
150         semilogy(t_hist, err_hist, 'r-o', 'LineWidth', 1.5);
151         xlabel('Time');
152         ylabel('Relative L2 Error');
153         title('Convergence History'); grid on;
154
155         drawnow;
156         % frame = getframe(gcf);
157         % writeVideo(k, frame);
158     end
159     t = t + dt;

```

```

160 end
161
162 % close(k);
163 % disp('Video saved successfully');
164
165 figure('Name','Final Convergence','NumberTitle','off');
166 semilogy(t_hist, err_hist, 'r-o', 'LineWidth', 1.5);
167 xlabel('Time'); ylabel('Relative L2 Norm of Error');
168 title('Final Convergence of Numerical Solution'); grid on;
169
170 fprintf('Max divergence: %.2e\n', max(abs(divu(:)))));
171
172 function [u, v, p] = apply_boundary_conditions(u, v, p,dpx,dx
    )
173     % Left wall
174     u(:,1) =u(:,2);
175     v(:,1) =0;
176
177     % Right wall
178     u(:,end) =u(:,end-1);
179     v(:,end) =0;
180
181     % Top wall
182     u(end,:) =0;
183     v(end,:) =0;
184
185     % Bottom wall
186     u(1,:) =0;
187     v(1,:) =0;
188
189     % non zero-gradient boundary condition for pressure at
    boundaries
190     p(:,1)      = p(:,2)-dpx*dx;          % left
191     p(:,end)    = p(:,end-1)+dpx*dx;      % right
192
193 end
194
195
196
197 % poission solver SOR
198 function [p,err]=sor_solver(p, S,Lx,Ly,x, y)
199     dx=Lx/x ;
200     dy=Ly/y ;
201     Ae = ones(y+2, x+2) / dx^2;
202     Aw = ones(y+2, x+2) / dx^2;
203     An = ones(y+2, x+2) / dy^2;
204     As = ones(y+2, x+2) / dy^2;
205     Ap=-(Ae+Aw+An+As);
206

```

```

207     it = 0;
208     err = 1e10;
209     tol = 1e-12;
210     maxit=20000;
211     B = 1.9; % between 1 and 2
212
213     while err > tol && it < maxit
214         pk = p;
215         for i =2:x+1
216             for j =2:y+1
217                 ap = Ap(j,i); ae = Ae(j,i); aw = Aw(j,i); an =
218                 An(j,i); as = As(j,i);
219                 pe = p(j,i+1); pw = p(j,i-1); pn = p(j+1,i);
220                 ps = p(j-1,i);
221                 res = S(j,i) - (ae*pe + aw*pw + an*pn + as*ps)
222             ;
223                 p(j,i) = B * res/ap + (1-B)*pk(j,i);
224             end
225         end
226     end
227     err = norm(p(:)-pk(:),2);
228     it = it+1;
229 end

```

A.1.4 Oscillating shear flow

```

1 % Oscillating Shear Flow Simulation with MG, Rhie-Chow
2 clc; clear; close all;
3
4 %% Parameters
5 Nx =64;
6 Ny=64;
7 Lx = 1;
8 Ly =.2;
9 dx = Lx/Nx;
10 dy = Ly/Ny;
11 x =linspace(0,Lx, Nx);
12 y =linspace(0,Ly, Ny);
13 [X, Y] =meshgrid(x, y);
14
15 nu = 0.05; % Kinematic viscosity
16 U0 = 5; % Wall velocity amplitude
17 f = 10;
18 omega = 2*pi*f;
19 rho =1;
20
21 % Time settings
22 CFL = 0.35;
23 dt = CFL*min(dx, dy)^2/nu;

```

```

24 Tf= 2/f;
25 Nt = ceil(Tf/dt);
26 dt = Tf/Nt;
27 time = linspace(0,Tf,Nt);
28
29
30 u = zeros(Ny,Nx);
31 v = zeros(Ny,Nx);
32 p = zeros(Ny,Nx);
33 relL2 = zeros(1, Nt);
34
35 alpha = sqrt(omega/(2*nu));
36 yc = linspace(dy/2,Ly -dy/2,Ny)';
37 % Figure
38 figure('Name', 'Oscillating Shear Flow', 'NumberTitle', 'off'
        );
39
40 subplot(2,2,1);
41 hQuiver = quiver(X, Y, zeros(size(X)), zeros(size(Y)), '
        AutoScaleFactor', 3);
42 title('Velocity Field'); xlabel('x'); ylabel('y'); axis equal
        tight;
43
44
45 subplot(2,2,2);
46 hAnalytical = plot(zeros(Ny,1), yc, 'r-', 'LineWidth', 2);
        hold on;
47 hNumerical = plot(zeros(Ny,1), yc, 'bo--');
48 title('Velocity Profile'); xlabel('u velocity'); ylabel('y');
        legend('Analytical', 'Numerical','Location','south');
        grid on;
49
50 subplot(2,2,3);
51 hError = plot(zeros(Ny,1), yc, 'k');
52 title('Error Distribution'); xlabel('Error'); ylabel('y');
        grid on;
53
54 subplot(2,2,4);
55 hConv = plot(0, 0, 'b');
56 title('Convergence of Relative L2 Error'); xlabel('Time Step'
        ); ylabel('Relative L2 Error'); grid on;
57
58
59
60 %% %% Setup video writer
61 % k = VideoWriter('CouetteFlowSimulation.mp4', 'MPEG-4'); %
        Name of video file
62 % k.FrameRate = 10; % Frames per second
63 % open(k);

```

```

64
65 %% Time loop
66 for n = 1:Nt
67     t = time(n);
68
69
70     u(1,:) = 0;
71     u(end,:) = U0 * sin(omega * t);
72     v([1 end], :) = 0;
73     v(:, [1 end]) = 0;
74
75     [u_star, v_star] = explicit_predictor(u, v, p, rho, nu,
dt, dx, dy);
76
77     rhs = divergence(u_star, v_star, dx, dy) / dt;
78
79
80     p_corr = poisson_solver(rhs, dx, dy);
81
82     [u, v] = rhie_chow_projection(u_star, v_star, p_corr, rho
, dt, dx, dy);
83     p = p + p_corr;
84
85
86     u_analytical = U0 * exp(-alpha * flip(yc'))' .* sin(omega
*t - alpha * flip(yc'))';
87     u_center = u(:, round(Nx/2));
88     err = abs(u_center - u_analytical);
89     relL2(n) = norm(err) / norm(u_analytical);
90
91     if mod(n, 10) == 0 || n == 1 || n == Nt
92         set(hQuiver, 'UData', u, 'VData', v);
93         set(hAnalytical, 'YData', yc, 'XData', u_analytical);
94         set(hNumerical, 'YData', yc, 'XData', u_center);
95         set(hError, 'XData', err, 'YData', yc);
96         set(hConv, 'XData', 1:n, 'YData', relL2(1:n));
97         drawnow;
98
99         %% - Capture frame for video -
100         % frame = getframe(gcf);
101         % writeVideo(k, frame);
102     end
103 end
104
105
106
107 % %% Close video file
108 % close(k);
109 % disp('Video saved successfully as CouetteFlowSimulation.mp4

```

```

    ');
110
111 % Helper Functions
112 function [u_corr, v_corr] = rhie_chow_projection(u_star,
    v_star, p_corr, rho, dt, dx, dy)
113     [Ny, Nx] = size(u_star);
114     u_corr = u_star;
115     v_corr = v_star;
116
117     dpdx = zeros(Ny, Nx);
118     dpdy = zeros(Ny, Nx);
119
120     dpdx(:,2:Nx-1) = (p_corr(:,3:Nx) - p_corr(:,1:Nx-2)) /
    (2*dx);
121     dpdy(2:Ny-1,:) = (p_corr(3:Ny,:) - p_corr(1:Ny-2,:)) /
    (2*dy);
122
123     u_corr = u_star - dt / rho * dpdx;
124     v_corr = v_star - dt / rho * dpdy;
125 end
126
127 function p = poisson_solver(rhs, dx, dy)
128     % V-cycle solver with gs smoothing
129     maxLevel = floor(log2(min(size(rhs)))) - 1;
130     p = fmg_vcycle(rhs, dx, dy, maxLevel);
131 end
132
133 function p = fmg_vcycle(rhs, dx, dy, level)
134     if level == 0
135         p = zeros(size(rhs));
136         p = gauss_seidel(p, rhs, dx, dy, 150);
137     else
138         coarse_rhs = restrict(rhs);
139         coarse_p = fmg_vcycle(coarse_rhs, 2*dx, 2*dy, level -
    1);
140         fine_p = prolong(coarse_p);
141         fine_p = gauss_seidel(fine_p, rhs, dx, dy, 150);
142         p = fine_p;
143     end
144 end
145
146 function out = gauss_seidel(p, rhs, dx, dy, iterations)
147     [Ny, Nx] = size(p);
148     dx2 = dx^2; dy2 = dy^2;
149     denom = 2*(dx2 + dy2);
150     for iter = 1:iterations
151         for j = 2:Ny-1
152             for i = 2:Nx-1
153                 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;
154         end
155     end
156 end
157 out = p;
158 end
159
160 function coarse = restrict(fine)
161     coarse = fine(1:2:end, 1:2:end);
162 end
163
164 function fine = prolong(coarse)
165     [Ny, Nx] = size(coarse);
166     fine = zeros(2*Ny, 2*Nx);
167     fine(1:2:end, 1:2:end) = coarse;
168     fine(2:2:end, 1:2:end) = coarse;
169     fine(1:2:end, 2:2:end) = coarse;
170     fine(2:2:end, 2:2:end) = coarse;
171 end
172 function div = divergence(u, v, dx, dy)
173     div = (u(:, [2:end end]) - u(:, [1 1:end-1])) / (2*dx) +
        ...
174         (v([2:end end], :) - v([1 1:end-1], :)) / (2*dy);
175 end
176
177 function [u_star, v_star] = explicit_predictor(u, v, p, rho,
        nu, dt, dx, dy)
178     [Ny, Nx] = size(u);
179     u_star = u;
180     v_star = v;
181
182     %Laplacians
183     uxx = zeros(Ny, Nx); uyy = zeros(Ny, Nx);
184     vxx = zeros(Ny, Nx); vyy = zeros(Ny, Nx);
185
186     uxx(:, 2:Nx-1) = (u(:, 3:Nx) - 2*u(:, 2:Nx-1) + u(:, 1:Nx-2))
        / dx^2;
187     uyy(2:Ny-1, :) = (u(3:Ny, :) - 2*u(2:Ny-1, :) + u(1:Ny-2, :))
        / dy^2;
188
189     vxx(:, 2:Nx-1) = (v(:, 3:Nx) - 2*v(:, 2:Nx-1) + v(:, 1:Nx-2))
        / dx^2;
190     vyy(2:Ny-1, :) = (v(3:Ny, :) - 2*v(2:Ny-1, :) + v(1:Ny-2, :))
        / dy^2;
191
192     %Pressure grads
193     px = zeros(Ny, Nx); py = zeros(Ny, Nx);
194     px(:, 2:Nx-1) = (p(:, 3:Nx) - p(:, 1:Nx-2)) / (2*dx);
195     py(2:Ny-1, :) = (p(3:Ny, :) - p(1:Ny-2, :)) / (2*dy);

```

```

196
197     u_star = u + dt * (-px/rho + nu * (uxx + uyy));
198     v_star = v + dt * (-py/rho + nu * (vxx + vyy));
199 end

```

A.1.5 Pulsating flow

```

1 clc; clear; close all;
2
3 y=48;
4 Ly=0.2;
5 dy=Ly/y;
6
7 visc=0.01;
8 rho=1;
9
10 % Womersleyflow parameters
11 U0=1;
12 f=5;
13 omega=2*pi*f;
14 T=2*pi/omega;
15 dt=T/2000;% 2000 points per cycle
16
17 dpdx_amp=dpdx_calc(U0, omega, visc, rho, Ly); % analytical dp
    /dx amplitude
18 tsteps=5400;
19
20 yc=linspace(dy/2, Ly - dy/2, y)';
21 u=womer_vel(yc, 0, dpdx_amp, omega, visc, rho, Ly); % t=0
22
23 % Initial acceleration consistency to reduce startup error
24 du_dt0=womersley_acceleration(yc, 0, dpdx_amp, omega, visc,
    rho, Ly);
25 u=u+0.5*dt*du_dt0;
26 ut=u;
27
28 err_l2_hist=zeros(tsteps,1);
29
30 % Plot Setup
31 figure('Name','Pulsating/Womersley Flow');
32 subplot(2,1,1);
33 hProfile=plot(u, yc, 'bo-', 'DisplayName', 'Numerical'); hold
    on;
34 hExact=plot(u, yc, 'r-', 'DisplayName', 'Analytical');
35 xlabel('u (m/s)'); ylabel('y'); grid on;
36 legend; title('Velocity Profile');
37
38 subplot(2,1,2);

```



```

39 hError=plot(0,0); xlabel('Time Step'); ylabel('Relative L2
    Error'); grid on;
40 title(sprintf('L2 Error vs Time Step (dt=%.2e s)', dt));
41
42 % Time-stepping Loop
43 old_dpx=dpdx_amp;
44 for n=1:tsteps
45     t=n*dt;
46     dpdx=dpdx_amp*sin(omega*t);
47     smooth_dpx=0.5*(dpdx+old_dpx);
48     old_dpx=dpdx;
49
50     % C r a n k Nicolson time integration (semi-implicit)
51     for j=2:y-1
52         diffu_n=visc*(u(j+1) - 2*u(j)+u(j-1))/dy^2;
53         frocin_n=old_dpx/rho;
54         diffu_np1=visc*(ut(j+1) - 2*ut(j)+ut(j-1))/dy^2;
55         frocin_np1=dpdx/rho;
56         ut(j)=u(j)+dt/2*(diffu_n+frocin_n+diffu_np1+
frocin_np1);
57     end
58
59     % No-slip BCs
60     ut(1)=0;
61     ut(end)=0;
62     u=ut;
63
64
65     u_exact=womer_vel(yc, t, dpdx_amp, omega, visc, rho, Ly);
66     u_exact(1)=0; u_exact(end)=0;
67
68
69     err=u - u_exact;
70     err_l2=norm(err,2);
71     norm_ref=norm(u_exact,2)+1e-10;
72     err_l2_hist(n)=err_l2/norm_ref;
73
74     % Plot every few steps
75     if mod(n, 20) == 0 || n == 1
76         set(hProfile, 'XData', u, 'YData', yc);
77         set(hExact, 'XData', u_exact, 'YData', yc);
78         set(hError, 'XData', 1:n, 'YData', err_l2_hist(1:n));
79         drawnow;
80     end
81 end
82
83
84 fprintf('Final Relative L2 Error: %.2e\n', err_l2_hist(end));
85

```

```

86 %helper functions
87 function dpdx_amp=dpx_calc(U0, omega, visc, rho, Ly)
88     i=1i;
89     lambda=sqrt(i*omega/visc);
90     spatial_factor=abs(1 - cosh(lambda*0)/cosh(lambda*Ly/2));
91     dpdx_amp=-U0*omega*rho/spatial_factor;
92 end
93
94 function u=womer_vel(y, t, dpdx_amp, omega, visc, rho, Ly)
95     i=1i;
96     lambda=sqrt(i*omega/visc);
97     y_shifted=y - Ly/2;
98     denom=cosh(lambda*Ly/2);
99     spatial_part=1 - cosh(lambda*y_shifted)/denom;
100    time_factor=exp(i*(omega*t - pi/2));
101    prefactor=dpdx_amp/(i*omega*rho);
102    u_complex=prefactor*spatial_part*time_factor;
103    u=real(u_complex);
104 end
105
106 function du_dt=womersley_acceleration(y, t, dpdx_amp, omega,
    visc, rho, Ly)
107     i=1i;
108     lambda=sqrt(i*omega/visc);
109     y_shifted=y - Ly/2;
110     denom=cosh(lambda*Ly/2);
111     spatial_part=1 - cosh(lambda*y_shifted)/denom;
112     time_factor=omega*exp(i*(omega*t+pi/2));
113     prefactor=dpdx_amp/rho;
114     du_dt_complex=prefactor*spatial_part*time_factor;
115     du_dt=real(du_dt_complex);
116 end

```

A.2 IBM code

A.2.1 simple shear flow

```

1 clc; clear; close all
2
3 % Domain and grid setup
4 x=32;
5 y=32;
6 Lx=1; Ly=0.2;
7 dx=Lx/x; dy=Ly/y;
8
9 visc=0.01;
10 Ut=10; Ub=0;
11 dt=min(1, 0.5*dx^2/visc);
12

```

```

13 [X, Y]=meshgrid(dx/2:dx:Lx-dx/2, dy/2:dy:Ly-dy/2);
14
15 u=zeros(y+2, x+2); v=zeros(y+2, x+2);
16 ut=u; vt=v; p=zeros(y+2, x+2);
17
18 % Analytical velocity profile
19 ya=linspace(dy/2, Ly-dy/2, y);
20 ua=(Ut-Ub)/Ly*ya+Ub;
21
22 Nb=100; s=linspace(0, 1, Nb);
23 Xb=Lx*s; Yb=Ly/2*ones(1, Nb);
24 tsteps=300;
25 err_l2_hist=zeros(tsteps, 1);
26 FxL_old=zeros(1, Nb);
27 FyL_old=zeros(1, Nb);
28 alpha=0.5;
29
30 for n=1:tsteps
31     [u,v, ~]=applybcs(u,v, Ut,Ub, p);
32     [uL, vL]=vel_interpolate(u,v, Xb,Yb, dx,dy);
33     epsilon=1;
34     FxL=epsilon*(0-uL);
35     FyL=epsilon*(0-vL);
36
37     FxL=alpha*FxL+(1-alpha)*FxL_old;
38     FyL=alpha*FyL+(1-alpha)*FyL_old;
39     FxL_old=FxL; FyL_old=FyL;
40
41     forceField=spread_force(FxL,FyL, Xb,Yb, dx,dy,x,y);
42     fx=forceField(:, :, 1);
43     fy=forceField(:, :, 2);
44
45     u_center=0.5*(u(2:end-1, 2:end-1)+u(2:end-1, 3:end));
46     fx_center=0.5*(fx(2:end-1, 2:end-1)+fx(2:end-1, 3:end));
47     u_center_new=implicit_diffusion_u(u_center+dt*fx_center,
Lx, Ly, x, y, dt, visc);
48     u(2:end-1, 2:end-1)=u_center_new;
49     u(2:end-1, 3:end)=u_center_new;
50     ut=u;
51
52     v_center=v(2:end-1, 2:end-1);
53     fy_center=fy(2:end-1, 2:end-1);
54     v_center_new=implicit_diffusion_v(v_center+dt*fy_center,
Lx, Ly, x, y, dt, visc);
55     v(2:end-1, 2:end-1)=v_center_new;
56     vt=v;
57
58     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;

```

```

59
60     rhs_full=zeros(y+2, x+2);
61     rhs_full(2:end-1, 2:end-1)=divut/dt;
62
63     for cycle=1:20
64         p_old=p;
65         p=V_cycle(p, rhs_full, Lx, Ly, x, y);
66         if norm(p(:)-p_old(:), 2)/norm(p_old(:), 2) < 1e-6
67             break;
68         end
69     end
70
71     [u, v]=rhie_chow_correction(ut, vt, p, dx, dy, dt);
72
73     uc=0.5*(u(2:end-1,2:end-1)+u(2:end-1,3:end));
74     u_profile=mean(uc, 2);
75     error_profile=u_profile-ua';
76     L2_error=sqrt(sum(error_profile.^2)/length(error_profile)
77 )/sqrt(sum(ua.^2)/length(ua));
78     err_l2_hist(n)=L2_error;
79
80     if mod(n,10) == 0
81         fprintf("Step %d, L2 Error=%.2e\n", n, L2_error);
82
83         x_index=round(x/2)+1;
84         uc_current=0.5*(u(2:end-1,2:end-1)+u(2:end-1,3:end));
85
86         figure(1); clf;
87         subplot(1,2,1);
88         plot(uc_current(:, x_index), ya, 'b-', 'LineWidth',
89 2); hold on;
90         plot(ua, ya, 'r--', 'LineWidth', 2);
91         xlabel('u'); ylabel('y'); title(['Velocity Profile at
92 x=Lx/2, Step=', num2str(n)]);
93         legend('Numerical', 'Analytical', 'Location', 'south');
94         grid on;
95
96         subplot(1,2,2);
97         semilogy(1:n, err_l2_hist(1:n), 'k-', 'LineWidth', 2)
98 ;
99         xlabel('Time step'); ylabel('Relative L2 Error');
100        title('L2 Error Convergence'); grid on;
101        drawnow;
102    end
103 end
104
105 function [u, v, p]=applybcs(u, v, Ut, Ub, p)
106     u(:,1)=u(:,2);
107     u(:,end)=u(:,end-1);

```

```

103
104     v(:,1)=0;
105     v(:,end)=0;
106
107     u(end,:)=Ut;
108     u(1,:)=Ub;
109
110     v([1 end],:)=0;
111
112     p(:,1)=p(:,2);
113     p(:,end)=p(:,end-1);
114     p(1,:)=p(2,:);
115     p(end,:)=p(end-1,:);
116 end
117
118 function [uL, vL]=vel_interpolate(u, v, Xb, Yb, dx, dy)
119     Nb=length(Xb);
120     uL=zeros(1,Nb);
121     vL=zeros(1,Nb);
122     for k=1:Nb
123         xk=Xb(k); yk=Yb(k);
124         i0=floor(xk/dx)+1; j0=floor(yk/dy)+1;
125         for i=i0-1:i0+2
126             for j=j0-1:j0+2
127                 if i >= 1 && i <= size(u,2) && j >= 1 && j <=
size(u,1)
128                     xi=(i-1)*dx; yj=(j-1)*dy;
129                     phi=delta_kernel((xk-xi)/dx)*delta_kernel
((yk-yj)/dy);
130                     uL(k)=uL(k)+u(j,i)*phi;
131                     vL(k)=vL(k)+v(j,i)*phi;
132                 end
133             end
134         end
135     end
136 end
137
138 function f=spread_force(FxL, FyL, Xb, Yb, dx, dy, Nx, Ny)
139     f=zeros(Ny+2, Nx+2, 2); Nb=length(Xb);
140     for k=1:Nb
141         xk=Xb(k); yk=Yb(k);
142         i0=floor(xk/dx)+1; j0=floor(yk/dy)+1;
143         for i=i0-1:i0+2
144             for j=j0-1:j0+2
145                 xi=(i-1)*dx; yj=(j-1)*dy;
146                 phi=delta_kernel((xk-xi)/dx)*delta_kernel((yk
-yj)/dy);
147                 if i >= 1 && i <= Nx+2 && j >= 1 && j <= Ny+2
148                     f(j,i,1)=f(j,i,1)+FxL(k)*phi*dx*dy;

```

```

149             f(j,i,2)=f(j,i,2)+FyL(k)*phi*dx*dy;
150         end
151     end
152 end
153 end
154 end
155 % 4-point kernel (Peskins standard)
156 function val=delta_kernel(r)
157     r=abs(r);
158     if r < 1
159         val=0.125*(3-2*r+sqrt(1+4*r-4*r^2));
160     elseif r < 2
161         val=0.125*(5-2*r-sqrt(-7+12*r-4*r^2));
162     else
163         val=0;
164     end
165 end
166
167
168
169 function u_new=implicit_diffusion_u(u_old, Lx, Ly, Nx, Ny, dt
    , nu)
170     dx=Lx/Nx;
171     dy=Ly/Ny;
172     N=Nx*Ny;
173     A=sparse(N, N);
174     b=zeros(N, 1);
175     coeff_center=1+2*dt*nu*(1/dx^2+1/dy^2);
176     coeff_x=-dt*nu/dx^2;
177     coeff_y=-dt*nu/dy^2;
178     index=@(i,j) (j-1)*Nx+i;
179     for j=1:Ny
180         for i=1:Nx
181             n=index(i,j);
182             % Top and bottom walls      Dirichilet BC
183             if j == 1
184                 A(n,:)=0;
185                 A(n,n)=1;
186                 b(n)=0;           % Bottom wall velocity Ub
187                 continue;
188             elseif j == Ny
189                 A(n,:)=0;
190                 A(n,n)=1;
191                 b(n)=10;          % Top wall velocity Ut
192                 continue;
193             end
194             % Interior and Left/Right
195             A(n,n)=coeff_center;
196             %Westside

```

```

197         if i > 1
198             A(n, index(i-1,j))=coeff_x;
199         else
200             A(n,n)=A(n,n)-coeff_x;    % Neumann
201         end
202         %Eastside
203         if i < Nx
204             A(n, index(i+1,j))=coeff_x;
205         else
206             A(n,n)=A(n,n)-coeff_x;    % Neumann
207         end
208         A(n, index(i,j-1))=coeff_y;
209         A(n, index(i,j+1))=coeff_y;
210         b(n)=u_old(j,i);
211     end
212 end
213
214 u_vec=A \ b;
215 u_new=reshape(u_vec, [Nx, Ny])';
216 end
217
218 % V-Cycle Function
219 function p=V_cycle(p, rhs, Lx, Ly, Nx, Ny)
220     if Nx <= 4 || Ny <= 4
221         p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, 100);
222         return;
223     end
224
225     p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, 10);
226     res=compute_residual(p, rhs, Lx, Ly, Nx, Ny);
227     res_coarse=restrict(res);
228     Nc_x=size(res_coarse,2)-2;
229     Nc_y=size(res_coarse,1)-2;
230     e_coarse=zeros(Nc_y+2, Nc_x+2);
231     e_coarse=V_cycle(e_coarse, res_coarse, Lx, Ly, Nc_x, Nc_y
232 );
233     e_fine=prolong(e_coarse, Nx, Ny);
234     p=p+e_fine;
235
236     p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, 7);
237 end
238
239 % SOR Smoother (Local)
240 function p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, Niter)
241     dx=Lx/Nx;
242     dy=Ly/Ny;
243     B=1.9;
244     for iter=1:Niter
245         p_old=p;

```

```

245         for i=2:Nx+1
246             for j=2:Ny+1
247                 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)));
248             end
249         end
250         % Neumann boundary conditions
251         p(:,1) =p(:,2);
252         p(:,end)=p(:,end-1);
253         p(1,:) =p(2,:);
254         p(end,:)=p(end-1);
255
256         if norm(p(:)-p_old(:), 2) < 1e-8
257             break;
258         end
259     end
260 end
261 function res=compute_residual(p, rhs, Lx, Ly, Nx, Ny)
262     dx=Lx/Nx;
263     dy=Ly/Ny;
264
265     res=zeros(Ny+2, Nx+2);
266     for i=2:Nx+1
267         for j=2:Ny+1
268             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;
269             res(j,i)=rhs(j,i)-laplace;
270         end
271     end
272 end
273 function coarse=restrict(fine)
274     [Nyf, Nxf]=size(fine);
275     Nxc=ceil((Nxf-2)/2);
276     Nyc=ceil((Nyf-2)/2);
277
278     coarse=zeros(Nyc+2, Nxc+2);
279     for i=2:Nxc+1
280         for j=2:Nyc+1
281             i_f=2*(i-1);
282             j_f=2*(j-1);
283             neighbors=fine(j_f-1:j_f+1, i_f-1:i_f+1);
284             weights=[1 2 1; 2 4 2; 1 2 1];
285             % Handle edges
286             valid=~isnan(neighbors);
287             w_sum=sum(weights(valid));
288             coarse(j,i)=sum(neighbors(valid) .* weights(valid
), 'all')/w_sum;
289         end

```



```

290     end
291 end
292
293 % Prolongation bilinear
294 function fine=prolong(coarse, Nxf, Nyf)
295     Nxc=size(coarse,2)-2;
296     Nyc=size(coarse,1)-2;
297     fine=zeros(Nyf+2, Nxf+2);
298     for i=2:Nxc+1
299         for j=2:Nyc+1
300             i_f=2*(i-1);
301             j_f=2*(j-1);
302             fine(j_f,i_f) =fine(j_f, i_f) +coarse(j,i);
303             fine(j_f+1,i_f) =fine(j_f+1, i_f) +coarse(j,i);
304             fine(j_f,i_f+1)=fine(j_f, i_f+1)+coarse(j,i);
305             fine(j_f+1,i_f+1)=fine(j_f+1,i_f+1)+coarse(j,i);
306         end
307     end
308     fine(2:end-1,2:end-1)=fine(2:end-1,2:end-1)/4;
309 end
310
311
312 function [u_corr, v_corr]=rhie_chow_correction(ut, vt, p, dx,
    dy, dt)
313     [Ny, Nx]=size(p);
314     u_corr=ut;
315     v_corr=vt;
316     % u correction
317     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;
318     % v correction
319     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;
320 end
321
322
323 function v_new=implicit_diffusion_v(v_old, Lx, Ly, Nx, Ny, dt
    , nu)
324     dx=Lx/Nx;
325     dy=Ly/Ny;
326     N=Nx*Ny;
327     A=sparse(N,N);
328     b=zeros(N, 1);
329     coeff_center=1+2*dt*nu*(1/dx^2+1/dy^2);
330     coeff_x=-dt*nu/dx^2;
331     coeff_y=-dt*nu/dy^2;
332     index=@(i,j) (j-1)*Nx+i;
333
334     for j=1:Ny

```

```

335     for i=1:Nx
336         n=index(i,j);
337         %top and bottom: dirichilet (v=0)
338         if j == 1 || j == Ny
339             A(n,:)=0;
340             A(n,n)=1;
341             b(n)=0;
342             continue;
343         end
344         A(n,n)=coeff_center;
345         if i > 1
346             A(n, index(i-1,j))=coeff_x;
347         else
348             A(n,n)=A(n,n)-coeff_x;
349         end
350         if i < Nx
351             A(n, index(i+1,j))=coeff_x;
352         else
353             A(n,n)=A(n,n)-coeff_x;
354         end
355         A(n, index(i,j-1))=coeff_y;
356         A(n, index(i,j+1))=coeff_y;
357
358         b(n)=v_old(j,i);
359     end
360 end
361
362 v_vec=A \ b;
363 v_new=reshape(v_vec, [Nx, Ny])';
364 end

```

A.2.2 Plane Poiseuille flow

```

1  clc; clear; close all
2
3  % Domain and grid setup
4  x=32; y=32;
5  Lx=1; Ly=0.2;
6  dx=Lx/x; dy=Ly/y;
7  visc=0.01;
8  Ut=0; Ub=0;
9  dt=min(1, 0.5*dx^2/visc);
10
11 rho=1; % density
12 dpx=-1; % pressure gradient(dp/dx -ve then flow
    in +ve)
13 [X, Y]=meshgrid(dx/2:dx:Lx-dx/2, dy/2:dy:Ly-dy/2);
14 u=zeros(y+2, x+2); v=zeros(y+2, x+2);
15 ut=u; vt=v; p=zeros(y+2, x+2);

```

```

16
17
18 ya=linspace(0, Ly, y);
19
20 ua=( -dpx/(2*visc) )*ya .* (Ly - ya);
21 Nb=100; s=linspace(0, 1, Nb);
22
23 % Time loop
24 tsteps=300; L2_error_history=zeros(tsteps, 1);
25 FxL_old=zeros(1, Nb); FyL_old=zeros(1, Nb);
26 alpha=0.5;
27
28 for n=1:tsteps
29     [u, v, ~]=applybcs(u,v, Ut, Ub, p);
30
31     fx=ones(y+2, x+2)*(-dpx)/rho; % uniform body force in x
32     fy=zeros(y+2, x+2);           % no vertical force
33
34     Nx_u=x+1; % for u: faces in x
35     Ny_u=y;
36
37     u_staggered=u(2:end-1, 2:end); % 32 33
38     fx_staggered=fx(2:end-1, 2:end); % 32 33
39
40     u_new=implicit_diffusion_u(u_staggered+dt*fx_staggered, Lx,
41                               Ly, Nx_u, Ny_u, dt, visc);
42     u(2:end-1, 2:end)=u_new;
43     ut=u;
44     v_center=v(2:end-1, 2:end-1);
45     fy_center=fy(2:end-1, 2:end-1);
46     v_center_new=implicit_diffusion_v(v_center+dt*fy_center,
47                                       Lx, Ly, x, y, dt, visc);
48     v(2:end-1, 2:end-1)=v_center_new;
49     vt=v;
50
51     divut=(ut(2:end-1,3:end) - ut(2:end-1,2:end-1))/dx+...
52            (vt(3:end,2:end-1) - vt(2:end-1,2:end-1))/dy;
53
54     rhs_full=zeros(y+2, x+2);
55     rhs_full(2:end-1, 2:end-1)=divut/dt;
56
57     for cycle=1:30
58         p_old=p;
59         p=V_cycle(p, rhs_full, Lx, Ly, x, y);
60         if norm(p(:) - p_old(:), 2)/norm(p_old(:), 2) < 1e-8
61             break;
62         end
63     end
64 end
65

```

```

63     [u, v]=rhie_chow_correction(ut, vt, p, dx, dy, dt);
64
65     uc=0.5*(u(2:end-1,2:end-1)+u(2:end-1,3:end));
66     u_profile=mean(uc, 2);
67     error_profile=u_profile - ua';
68     L2_error=sqrt(sum(error_profile.^2)/length(error_profile)
69 )/sqrt(sum(ua.^2)/length(ua));
70     L2_error_history(n)=L2_error;
71
72     if mod(n,10) == 0
73         fprintf("Step %d, L2 Error=%.2e\n", n, L2_error);
74
75         x_index=round(x/2)+1;
76         uc_current=0.5*(u(2:end-1,2:end-1)+u(2:end-1,3:end));
77
78         figure(1); clf;
79         subplot(1,2,1);
80         plot(uc_current(:, x_index), ya, 'b-', 'LineWidth',
81 2); hold on;
82         plot(ua, ya, 'r--', 'LineWidth', 2);
83         xlabel('u'); ylabel('y'); title(['Velocity Profile at
84 x=Lx/2, Step=', num2str(n)]);
85         legend('Numerical', 'Analytical', 'Location', 'south');
86         grid on;
87
88         subplot(1,2,2);
89         semilogy(1:n, L2_error_history(1:n), 'k-', 'LineWidth
90 ', 2);
91         xlabel('Time step'); ylabel('Relative L2 Error');
92         title('L2 Error Convergence'); grid on;
93         drawnow;
94     end
95 end
96
97 function [u, v, p]=applybcs(u, v, Ut, Ub, p)
98     u(:,1)=u(:,2);
99     u(:,end)=u(:,end-1);
100
101     v(:,1)=0;
102     v(:,end)=0;
103
104     u(end,:)=Ut;
105     u(1,:)=Ub;
106
107     v([1 end],:)=0;
108
109     p(:,1)=p(:,2);
110     p(:,end)=p(:,end-1);
111     p(1,:)=p(2,:);

```

```

107     p(end,:)=p(end-1,:);
108 end
109
110 function [uL, vL]=vel_interpolate(u, v, Xb, Yb, dx, dy)
111     Nb=length(Xb);
112     uL=zeros(1,Nb);
113     vL=zeros(1,Nb);
114     for k=1:Nb
115         xk=Xb(k); yk=Yb(k);
116         i0=floor(xk/dx)+1; j0=floor(yk/dy)+1;
117         for i=i0-1:i0+2
118             for j=j0-1:j0+2
119                 if i >= 1 && i <= size(u,2) && j >= 1 && j <=
size(u,1)
120                     xi=(i-1)*dx; yj=(j-1)*dy;
121                     phi=delta_kernel((xk - xi)/dx)*
delta_kernel((yk - yj)/dy);
122                     uL(k)=uL(k)+u(j,i)*phi;
123                     vL(k)=vL(k)+v(j,i)*phi;
124                 end
125             end
126         end
127     end
128 end
129
130 function f=spread_force(FxL, FyL, Xb, Yb, dx, dy, Nx, Ny)
131     f=zeros(Ny+2, Nx+2, 2); Nb=length(Xb);
132     for k=1:Nb
133         xk=Xb(k); yk=Yb(k);
134         i0=floor(xk/dx)+1; j0=floor(yk/dy)+1;
135         for i=i0-1:i0+2
136             for j=j0-1:j0+2
137                 xi=(i-1)*dx; yj=(j-1)*dy;
138                 phi=delta_kernel((xk - xi)/dx)*delta_kernel((
yk - yj)/dy);
139                 if i >= 1 && i <= Nx+2 && j >= 1 && j <= Ny+2
140                     f(j,i,1)=f(j,i,1)+FxL(k)*phi*dx*dy;
141                     f(j,i,2)=f(j,i,2)+FyL(k)*phi*dx*dy;
142                 end
143             end
144         end
145     end
146 end
147
148 % 4-point kernel (Peskins standard)
149 function val=delta_kernel(r)
150     r=abs(r);
151     if r < 1
152         val=0.125*(3 - 2*r+sqrt(1+4*r - 4*r^2));

```

```

153     elseif r < 2
154         val=0.125*(5 - 2*r - sqrt(-7+12*r - 4*r^2));
155     else
156         val=0;
157     end
158 end
159
160
161
162 function u_new=implicit_diffusion_u(u_old, Lx, Ly, Nx, Ny, dt
    , nu)
163     dx=Lx/Nx;
164     dy=Ly/Ny;
165     N=Nx*Ny;
166     A=sparse(N, N);
167     b=zeros(N, 1);
168     coeff_center=1+2*dt*nu*(1/dx^2+1/dy^2);
169     coeff_x=-dt*nu/dx^2;
170     coeff_y=-dt*nu/dy^2;
171     index=@(i,j) (j-1)*Nx+i;
172
173     for j=1:Ny
174         for i=1:Nx
175             n=index(i,j);
176             % top and bottom walls Dirichilet BC
177             if j == 1
178                 A(n,:)=0;
179                 A(n,n)=1;
180                 b(n)=0; % Bottom wall velocity Ub=0
181                 continue;
182             elseif j == Ny
183                 A(n,:)=0;
184                 A(n,n)=1;
185                 b(n)=0; % Top wall velocity Ut=0
186                 continue;
187             end
188             A(n,n)=coeff_center;
189             if i > 1
190                 A(n, index(i-1,j))=coeff_x;
191             else
192                 A(n,n)=A(n,n) - coeff_x; % Neumann
193             end
194             if i < Nx
195                 A(n, index(i+1,j))=coeff_x;
196             else
197                 A(n,n)=A(n,n) - coeff_x; % Neumann
198             end
199             A(n, index(i,j-1))=coeff_y;
200             A(n, index(i,j+1))=coeff_y;

```

```

201         b(n)=u_old(j,i);
202     end
203 end
204 u_vec=A \b;
205 u_new=reshape(u_vec, [Nx, Ny])';
206 end
207
208 % V-Cycle Function
209 function p=V_cycle(p, rhs, Lx, Ly, Nx, Ny)
210     if Nx <= 4 || Ny <= 4
211         p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, 100);
212         return;
213     end
214
215     p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, 15);
216     res=residualcalc(p, rhs, Lx, Ly, Nx, Ny);
217     res_coarse=restrict(res);
218
219     Nc_x=size(res_coarse,2) - 2;
220     Nc_y=size(res_coarse,1) - 2;
221     e_coarse=zeros(Nc_y+2, Nc_x+2);
222     e_coarse=V_cycle(e_coarse, res_coarse, Lx, Ly, Nc_x, Nc_y
223 );
224     e_fine=prolong(e_coarse, Nx, Ny);
225     p=p+e_fine;
226
227
228     p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, 10);
229 end
230
231
232 function p=sor_solve(p, rhs, Lx, Ly, Nx, Ny, Niter)
233     dx=Lx/Nx;
234     dy=Ly/Ny;
235     B=1.9;
236
237     for iter=1:Niter
238         p_old=p;
239         for i=2:Nx+1
240             for j=2:Ny+1
241                 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)));
242             end
243         end
244
245         % Neumann boundary conditions
246         p(:,1) =p(:,2);

```

```

247         p(:,end)=p(:,end-1);
248         p(1,:) =p(2,:);
249         p(end,:)=p(end-1);
250
251         if norm(p(:) -p_old(:), 2) < 1e-8
252             break;
253         end
254     end
255 end
256
257 function res=residualcalc(p,rhs, Lx, Ly, Nx, Ny)
258     dx=Lx/Nx;
259     dy=Ly/Ny;
260     res=zeros(Ny+2, Nx+2);
261     for i=2:Nx+1
262         for j=2:Ny+1
263             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;
264         end
265     end
266 end
267 function coarse=restrict(fine)
268     [Nyf, Nxf]=size(fine);
269     Nxc=ceil((Nxf - 2)/2);
270     Nyc=ceil((Nyf - 2)/2);
271     coarse=zeros(Nyc+2, Nxc+2);
272     for i=2:Nxc+1
273         for j=2:Nyc+1
274             i_f=2*(i-1);
275             j_f=2*(j-1);
276             neighbors=fine(j_f-1:j_f+1, i_f-1:i_f+1);
277             weights=[1 2 1; 2 4 2; 1 2 1];
278             valid=~isnan(neighbors);
279             w_sum=sum(weights(valid));
280             coarse(j,i)=sum(neighbors(valid) .* weights(valid
), 'all')/w_sum;
281         end
282     end
283 end
284
285 % Prolongation bilinear
286 function fine=prolong(coarse, Nxf, Nyf)
287     Nxc=size(coarse,2)- 2;
288     Nyc=size(coarse,1)- 2;
289
290     fine=zeros(Nyf+2, Nxf+2);
291
292     for i=2:Nxc+1
293         for j=2:Nyc+1

```



```

294         i_f=2*(i-1);
295         j_f=2*(j-1);
296
297         fine(j_f, i_f) =fine(j_f, i_f) +coarse(j,i);
298         fine(j_f+1, i_f) =fine(j_f+1, i_f) +coarse(j,i)
299     ;
300         fine(j_f, i_f+1)=fine(j_f, i_f+1)+coarse(j,i);
301         fine(j_f+1, i_f+1)=fine(j_f+1, i_f+1)+coarse(j,i)
302     ;
303     end
304 end
305
306
307 function [u_corr, v_corr]=rhie_chow_correction(ut, vt, p, dx,
    dy, dt)
308     [Ny, Nx]=size(p);
309     u_corr=ut;
310     v_corr=vt;
311     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;
312     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;
313 end
314
315
316 function v_new=implicit_diffusion_v(v_old, Lx, Ly, Nx, Ny, dt
    , nu)
317     dx=Lx/Nx;
318     dy=Ly/Ny;
319     N=Nx*Ny;
320     A=sparse(N, N);
321     b=zeros(N, 1);
322     coeff_center=1+2*dt*nu*(1/dx^2+1/dy^2);
323     coeff_x=-dt*nu/dx^2;
324     coeff_y=-dt*nu/dy^2;
325     index=@(i,j) (j-1)*Nx+i;
326
327     for j=1:Ny
328         for i=1:Nx
329             n=index(i,j);
330             %top and bottom: Dirichilet(v=0)
331             if j == 1 || j == Ny
332                 A(n,:)=0;
333                 A(n,n)=1;
334                 b(n)=0;
335                 continue;
336             end

```

```

337         A(n,n)=coeff_center;
338         if i > 1
339             A(n, index(i-1,j))=coeff_x;
340         else
341             A(n,n)=A(n,n) - coeff_x;
342         end
343         if i < Nx
344             A(n, index(i+1,j))=coeff_x;
345         else
346             A(n,n)=A(n,n) - coeff_x;
347         end
348         A(n, index(i,j-1))=coeff_y;
349         A(n, index(i,j+1))=coeff_y;
350         b(n)=v_old(j,i);
351     end
352 end
353 v_vec=A \ b;
354 v_new=reshape(v_vec, [Nx, Ny])';
355 end

```

A.2.3 Oscillating shear flow

```

1  clc; clear;
2
3  x=32;
4  y=32;
5  Lx=1;
6  Ly=.2;
7  dx=Lx/x;
8  dy=Ly/y;
9
10 visc=.05;
11 U0=10;                % Amplitude
12 f=5 ;                 % Frequency (Hz)
13 omega=2*pi*f;         % Angular frequency
14
15 Ub=0;                 % Bottom wall velocity (stationary)
16
17 CFL=0.5;
18 dt1=1e6;
19 dt2=CFL*min(dx, dy)/abs(U0);
20 dt=0.9*min(dt1, dt2)
21
22
23 [X, Y]=meshgrid(dx/2:dx:Lx-dx/2, dy/2:dy:Ly-dy/2);
24
25 y_ib=Ly*ones(1, x);
26 x_ib=linspace(dx/2, Lx-dx/2, x);
27

```

```

28 u =zeros(y+2, x+2);
29 v =zeros(y+2, x+2);
30 ut=zeros(y+2, x+2);
31 vt=zeros(y+2, x+2);
32 p =zeros(y+2, x+2);
33
34 fx=zeros(y+2, x+2);
35 fy=zeros(y+2, x+2);
36
37 % Plot setup
38 figure;
39 subplot(1,2,1);
40 prof_line=plot(zeros(y,1), linspace(dy/2,Ly-dy/2,y), 'b-', '
    LineWidth', 2); hold on;
41 anal_line=plot(zeros(y,1), linspace(dy/2,Ly-dy/2,y), 'r--', '
    LineWidth', 2);
42 legend('Numerical','Analytical','Location','south');
43 xlabel('u'); ylabel('y'); title('Velocity Profile at Lx/2');
    grid on;
44
45 subplot(1,2,2);
46 err_plot=semilogy(0,0,'k');
47 xlabel('Timestep'); ylabel('Relative L2 Error'); title('
    Convergence'); grid on;
48 tsteps=2000;
49 err_l2_hist=zeros(tsteps,1);
50
51
52 %
53 % %Video setup
54 % k=VideoWriter('IBM_OscShearFlow.mp4', 'MPEG-4');
55 % k.FrameRate=10;
56 % open(k);
57
58 %Time loop
59 for n=1:tsteps
60     time=n*dt;
61     Ut=U0*sin(omega*time);
62     u_desired=Ut*ones(size(x_ib));
63     u_ib=interpolate(u, x_ib, y_ib, dx, dy);
64     f_ib=(u_desired-u_ib)/dt;
65     [fx, fy]=spreadf(f_ib, zeros(size(f_ib)), x_ib, y_ib,
        size(u), dx, dy);
66
67     ut=u+dt*(visc*laplacian(u, dx, dy)+fx);
68     vt=v+dt*(visc*laplacian(v, dx, dy)+fy);
69
70     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;

```

```

71     rhs=divut/dt;
72     [p,~]=sor_solver(p, rhs, Lx, Ly, x, y);
73
74     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;
75     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;
76
77     u(end,:)=Ut;      % Top wall velocity
78     u(1,:)=Ub;      % Bottom wall stationary
79
80     % Error analysis
81     yc=linspace(dy/2, Ly-dy/2, y);
82     uc=0.5*(u(2:end-1,2:end-1)+u(2:end-1,3:end));
83     u_profile=mean(uc,2);
84     ua=vel_anal(yc, time, U0, omega, visc);
85
86     err=sqrt(sum((u_profile-ua').^2)/length(ua))/sqrt(sum(ua
.^2)/length(ua));
87     err_l2_hist(n)=err;
88
89     % Plot
90     if mod(n,10) == 0
91         set(prof_line, 'XData', u_profile, 'YData', yc);
92         set(anal_line, 'XData', ua, 'YData', yc);
93         set(err_plot, 'XData', 1:n, 'YData', err_l2_hist(1:n)
);
94         drawnow;
95
96
97         %           % Save frame to video
98         % frame=getframe(gcf);
99         % writeVideo(k, frame);
100     end
101 end
102
103
104 % %Close Video
105 % close(k);
106 % disp('Video saved as IBM_OscShearFlow.mp4');
107
108
109 function u_analytical=vel_anal(y, t, U0, omega, visc)
110     alpha=sqrt(omega/(2*visc));
111     Ly=y(end);
112     y_from_top=Ly-y;
113     u_analytical=U0*exp(-alpha*y_from_top) .* sin(omega*t
-alpha*y_from_top);
114 end

```

```

115
116
117 function L=laplacian(f, dx, dy)
118     L=zeros(size(f));
119     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;
120 end
121
122 function u_ib=interpolate(u, x_ib, y_ib, dx, dy)
123     u_ib=zeros(size(x_ib));
124     for k=1:length(x_ib)
125         i=floor(x_ib(k)/dx)+1;
126         j=floor(y_ib(k)/dy)+1;
127         wx=(x_ib(k)-(i-1)*dx)/dx;
128         wy=(y_ib(k)-(j-1)*dy)/dy;
129         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);
130     end
131 end
132
133 function [fx, fy]=spreadf(fx_ib, fy_ib, x_ib, y_ib, size_u,
        dx, dy)
134     fx=zeros(size_u);
135     fy=zeros(size_u);
136     for k=1:length(x_ib)
137         i=floor(x_ib(k)/dx)+1;
138         j=floor(y_ib(k)/dy)+1;
139         wx=(x_ib(k)-(i-1)*dx)/dx;
140         wy=(y_ib(k)-(j-1)*dy)/dy;
141         fx(j,i) =fx(j,i) +(1-wx)*(1-wy)*fx_ib(k);
142         fx(j,i+1)=fx(j,i+1) +wx*(1-wy)*fx_ib(k);
143         fx(j+1,i) =fx(j+1,i) +(1-wx)*wy*fx_ib(k);
144         fx(j+1,i+1)=fx(j+1,i+1) +wx*wy*fx_ib(k);
145
146         fy(j,i)=fy(j,i) +(1-wx)*(1-wy)*fy_ib(k);
147         fy(j,i+1)=fy(j,i+1)+wx*(1-wy)*fy_ib(k);
148         fy(j+1,i)=fy(j+1,i)+(1-wx)*wy*fy_ib(k);
149         fy(j+1,i+1) =fy(j+1,i+1) +wx*wy*fy_ib(k);
150     end
151 end
152
153 function [p, err]=sor_solver(p, rhs, Lx, Ly, Nx, Ny)
154     dx=Lx/Nx;
155     dy=Ly/Ny;
156     B=1.9;
157     tol=1e-10;
158     maxit=8000;
159     err=1e10;

```

```

160     it=0;
161
162     while err > tol && it < maxit
163         p_old=p;
164         for i=2:Nx+1
165             for j=2:Ny+1
166                 if i < Nx+1 && j < Ny+1
167                     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)));
168                 end
169             end
170         end
171         err=norm(p(:)-p_old(:), 2);
172         it=it+1;
173     end
174 end

```

A.2.4 Pulsating flow

```

1 clc; clear; close all;
2
3 % Parameters
4 Ny=48;
5 Ly=0.2;
6 dy=Ly/Ny;
7 visc=0.01;
8 rho=1;
9
10 U0=1;
11 f=5;
12 omega=2*pi*f;
13 T=2*pi/omega;
14 dt=T/2000;
15
16 tsteps=5500;
17 dpx_amp=dpx_calc(U0, omega, visc, rho, Ly);
18
19
20 y=linspace(-dy/2, Ly+dy/2, Ny+2)';
21 yc=y(2:end-1);
22 u=zeros(Ny+2, 1); % including ghost nodes
23 ut=zeros(Ny+2, 1);
24
25 y_ib=[0, Ly];
26 x_ib=ones(size(y_ib)); % dummy x since it's 1D
27
28 % Plot setup
29 figure('Name','IBM Womersley Flow');

```

```

30 subplot(2,1,1);
31 hProf=plot(u(2:end-1), yc, 'b-', 'LineWidth', 2); hold on;
32 hAnal=plot(u(2:end-1), yc, 'r--', 'LineWidth', 2);
33 xlabel('u'); ylabel('y'); title('Velocity Profile'); grid on;
    legend('Numerical','Analytical','Location','south');
34
35 subplot(2,1,2);
36 hErr=plot(0,0); xlabel('Timestep'); ylabel('Relative L2 Error
    '); grid on;
37 err_l2_hist=zeros(tsteps,1);
38
39
40 u(2:end-1)=vel_womers(yc, 0, dpx_amp, omega, visc, rho, Ly);
41
42 % Time loop
43 for n=1:tsteps
44     t=n * dt;
45     dpdx=dpx_amp * sin(omega * t);
46     u_ib_desired=[0; 0];
47     u_ib=interpolation(u, y_ib, y, dy);
48     f_ib_old=zeros(size(u_ib));
49     alpha=0.3;
50
51     f_ib_raw=(u_ib_desired-u_ib)/dt;
52     f_ib=alpha * f_ib_old+(1-alpha) * f_ib_raw;
53     f_ib_old=f_ib;
54     F=spreadf(f_ib, y_ib, y, dy);
55
56     for j=2:Ny+1
57         diffu_n=visc * (u(j+1)-2*u(j)+u(j-1))/dy^2;
58         diffu_np1=visc * (ut(j+1)-2*ut(j)+ut(j-1))/dy^2;
59         f_avg=dpdx/rho+F(j);
60         ut(j)=u(j)+dt * (0.5 * (diffu_n+diffu_np1)+f_avg);
61     end
62     u=ut;
63
64     %analytical vel
65     u_anal=vel_womers(yc, t, dpx_amp, omega, visc, rho, Ly);
66
67     % Error
68     err=u(2:end-1)-u_anal;
69     err_l2_hist(n)=norm(err)/norm(u_anal);
70
71     if mod(n,20)==0 || n==1
72         set(hProf, 'XData', u(2:end-1), 'YData', yc);
73         set(hAnal, 'XData', u_anal, 'YData', yc);
74         set(hErr, 'XData', 1:n, 'YData', err_l2_hist(1:n));
75         drawnow;
76     end

```

```

77 end
78
79 fprintf('Final Relative L2 Error: %.2e\n', err_l2_hist(end));
80
81 % IBM helper funcs
82 function u_ib=interpolation(u, y_ib, y, dy)
83     u_ib=zeros(size(y_ib));
84     for k=1:length(y_ib)
85         j=floor(y_ib(k)/dy)+1;
86         wy=(y_ib(k)-y(j))/dy;
87         u_ib(k)=(1-wy)*u(j)+wy*u(j+1);
88     end
89 end
90
91 function F=spreadf(f_ib, y_ib, y, dy)
92     F=zeros(size(y));
93     for k=1:length(y_ib)
94         j=floor(y_ib(k)/dy)+1;
95         wy=(y_ib(k)-y(j))/dy;
96         F(j) =F(j) +(1-wy) * f_ib(k);
97         F(j+1)=F(j+1)+wy * f_ib(k);
98     end
99 end
100
101 function dpx_amp=dpx_calc(U0, omega, visc, rho, Ly)
102     i=1i;
103     spatial_factor=abs(1-cosh(sqrt(i * omega/visc)*0)/cosh(
sqrt(i * omega/visc)* Ly/2));
104     dpx_amp=-U0 * omega * rho/spatial_factor;
105 end
106
107 function u=vel_womers(y, t, dpx_amp, omega, visc, rho, Ly)
108     i=1i;
109     y_shifted=y-Ly/2;
110     denom=cosh(sqrt(i * omega/visc) * Ly/2);
111     spatial_part=1-cosh(sqrt(i * omega/visc) * y_shifted)/
denom;
112     time_factor=exp(i * (omega * t-pi/2));
113     prefactor=dpx_amp/(i * omega * rho);
114     u_complex=prefactor * spatial_part * time_factor;
115     u=real(u_complex);
116 end

```

A.2.5 Flow over a cylinder

```

1 clc; clear; close all;
2
3 Nx=256;
4 Ny=64;

```



```

5 Lx=2.0;
6 Ly=1.0;
7 dx=Lx/Nx;
8 dy=Ly/Ny;
9 x=linspace(0, Lx, Nx);
10 y=linspace(0, Ly, Ny);
11 [X, Y]=meshgrid(x, y);
12
13 rho=1.0;
14 nu=0.001;
15 U_inf=10;
16 dt=0.001;
17 steps=3000;
18 beta=1.5; % SOR over-relaxation factor
19
20 cx=1.0;
21 cy=0.5;
22 R=0.1;
23 Nb=100;
24 theta=linspace(0, 2*pi, Nb);
25 Xb=cx+R*cos(theta);
26 Yb=cy+R*sin(theta);
27
28 % Field includes ghost cells
29 u=U_inf * ones(Ny+2, Nx+2);
30 v=zeros(Ny+2, Nx+2);
31 p=zeros(Ny+2, Nx+2);
32 ut=u; vt=v;
33
34 FxL_old=zeros(1, Nb);
35 FyL_old=zeros(1, Nb);
36 alpha=0.5;
37 Fx_total=zeros(1, steps);
38 Fy_total=zeros(1, steps);
39
40 %Time loop
41 for n=1:steps
42     [u, v, p]=apply_bc(u, v, p, U_inf);
43     ut=GS_diffuse(u, nu, dt, dx, dy);
44     vt=GS_diffuse(v, nu, dt, dx, dy);
45
46     [uL, vL]=vel_interpol(ut, vt, Xb, Yb, dx, dy);
47     epsilon=1000;
48     FxL=-epsilon * uL;
49     FyL=-epsilon * vL;
50     FxL=alpha * FxL+(1-alpha) * FxL_old;
51     FyL=alpha * FyL+(1-alpha) * FyL_old;
52     FxL_old=FxL;
53     FyL_old=FyL;

```

```

54     Fx_total(n)=sum(FxL);    %drag_force
55     Fy_total(n)=sum(FyL);    %lift_force
56
57     [fx, fy]= spreadf(FxL, FyL, Xb, Yb, Nx, Ny, dx, dy);
58     ut=ut+dt * fx/rho;
59     vt=vt+dt * fy/rho;
60     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);
61     rhs=zeros(Ny+2, Nx+2);
62     rhs(2:end-1,2:end-1)=div/dt;
63     p=solve_poisson(p, rhs, dx, dy, beta);
64
65     % Velocity correction
66     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;
67     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;
68
69     % Plot every 200 steps
70     if mod(n,10) == 0
71         uc=0.5 *(u(2:end-1,2:end-1) +u(2:end-1 ,3:end));
72         vc= 0.5 * (v(2:end-1,2:end-1)+ v(3:end,2:end-1));
73         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);
74
75         figure(1); clf;
76         subplot(2,1,1);
77         contourf(X, Y, omega, 100, 'LineColor', 'none');
colorbar();
78         colormap(turbo);
79         caxis([-100 100]);
80         hold on; fill(cx+R*cos(theta), cy+R*sin(theta), 'c');
81         title(['Vorticity at step ', num2str(n)]);
82         axis equal tight;
83
84         subplot(2,1,2);
85         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);
86         hold on; fill(cx+R*cos(theta), cy+R*sin(theta), 'k');
87         title('Velocity Field'); axis equal tight;
88
89         time=dt * (1:steps);
90         drawnow;
91     end
92 end
93
94 function [u,v,p]=apply_bc(u,v,p,Uinf)
95     u(:,1)=Uinf; u(:,end)=u(:,end-1);
96     v(:,1)=0;      v(:,end)=0;

```

```

97     u(1,:)=u(2,:); u(end,:)=u(end-1,:);
98     v(1,:)=0;      v(end,:)=0;
99     p(:,1)=p(:,2); p(:,end)=p(:,end-1);
100    p(1,:)=p(2,:); p(end,:)=p(end-1,:);
101 end
102 function u=GS_diffuse(u, nu, dt, dx, dy)
103     [Ny,Nx]=size(u);
104     for iter=1:50
105         u_old=u;
106         for j=2:Ny-1
107             for i=2:Nx-1
108                 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));
109             end
110         end
111     end
112 end
113 function [uL, vL]=vel_interpol(u, v, Xb, Yb, dx, dy)
114     Nb=length(Xb);
115     uL=zeros(1,Nb); vL=zeros(1,Nb);
116     for k=1:Nb
117         i=floor(Xb(k)/dx)+2; j=floor(Yb(k)/dy)+2;
118         uL(k)=u(j,i);
119         vL(k)=v(j,i);
120     end
121 end
122
123
124 function [fx, fy]=spreadf(FxL, FyL, Xb, Yb, Nx, Ny, dx, dy)
125     fx=zeros(Ny+2, Nx+2); fy=zeros(Ny+2, Nx+2);
126     for k=1:length(Xb)
127         i=floor(Xb(k)/dx)+2; j=floor(Yb(k)/dy)+2;
128         fx(j,i)=fx(j,i)+FxL(k);
129         fy(j,i)=fy(j,i)+FyL(k);
130     end
131 end
132
133
134 function p=solve_poisson(p, rhs, dx, dy, beta)
135     [Ny,Nx]=size(p);
136     for it=1:200
137         p_old=p;
138         for j=2:Ny-1
139             for i=2:Nx-1
140                 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));
141             end
142         end
143         if max(abs(p(:)-p_old(:))) < 1e-6

```

```

144         break;
145     end
146 end
147 end

```

A.3 Verifying results of Paper

```

1  clc; clear;
2
3  Nx=256; Ny=64;
4  Lx=4; Ly=1;
5  dx=Lx/Nx; dy=Ly/Ny;
6  x=linspace(0, Lx-dx, Nx);
7  y=linspace(0, Ly-dy, Ny);
8  [X, Y]=meshgrid(x, y);
9  AR=Lx/Ly;
10
11 kappa=0.125;
12 a=kappa*Ly;
13 Repr=[1, 3];
14 Uw=0.1;
15 G=2*Uw/Ly;
16 rho=1;
17
18 theta=linspace(0, 2*pi, 100);
19 Nb=length(theta);
20 Xb0=a*cos(theta);
21 Yb0=a*sin(theta);
22
23
24 ytilde_values=linspace(-0.35, 0.35, 41); % same as Fig. 3
25 lift_vs_y=zeros(length(Repr), length(ytilde_values));
26
27 for r=1:length(Repr)
28     Rep=Repr(r);
29     nu=G*a^2/Rep;
30     dt=0.1; % LBM-compatible value
31     mu=rho*nu;
32
33     for j=1:length(ytilde_values)
34         y0=(ytilde_values(j)+0.5)*Ly;
35         x0=2; % center in x
36         ux=zeros(Ny, Nx); uy=zeros(Ny, Nx);
37         fx=zeros(Ny, Nx); fy=zeros(Ny, Nx);
38
39
40         Xb=x0+Xb0;
41         Yb=y0+Yb0;
42
43         for i=1:Ny
44             uy(i,:)=0;
45             ux(i,:)=Uw*(y(i)-0)/Ly;
46         end
47
48
49         for iter=1:1000
50
51             Ub=interpolate_vel(ux, uy, Xb, Yb, x, y, dx, dy);

```

```

52         Up=mean(Ub,1);
53         Omega=mean((Ub(:,1).*Yb0(:)-Ub(:,2).*Xb0(:))/a^2);
54
55         Up_local=[Up(1)+Omega*(-Yb0(:)), Up(2)+Omega*( Xb0(:))];
56
57         alpha=1000; % spring stiffness
58         Fb=alpha*(Up_local-Ub);
59
60         fx(:)=0; fy(:)=0;
61         [fx, fy]=spreadf(fx, fy, Fb, Xb, Yb, x, y, dx, dy);
62         ux=ux+dt*fx/rho;
63         uy=uy+dt*fy/rho;
64     end
65
66     lift_vs_y(r,j)=sum(Fb(:,2));
67 end
68 end
69
70
71 Fl_dimless=lift_vs_y ./ (rho*Uw^2*a*kappa^2); % equation taken from paper
72
73 figure;
74 hold on
75 colors=lines(length(Reps));
76 for i=1:length(Reps)
77     plot(ytilde_values, Fl_dimless(i,:), 'LineWidth', 2, 'Color', colors(i
78         ,:));
79 end
80 xlabel('$\tilde{y}_0$', 'Interpreter', 'latex')
81 ylabel('$\tilde{F}_L$', 'Interpreter', 'latex')
82 legend(arrayfun(@(r) sprintf('Re_p=%d', r), Reps, 'UniformOutput', false))
83 title('Lift force vs transverse position (IBM)')
84 grid on
85 function Ub=interpolate_vel(ux, uy, Xb, Yb, x, y, dx, dy)
86     [Ny, Nx]=size(ux);
87     Nb=length(Xb); Ub=zeros(Nb,2);
88     for k=1:Nb
89         i0=floor(Xb(k)/dx); j0=floor(Yb(k)/dy);
90         for ii=-1:2
91             for jj=-1:2
92                 i=mod(i0+ii-1, Nx)+1;
93                 j=min(max(j0+jj,1), Ny); % non-periodic in y
94                 phi=delta((Xb(k)-x(i))/dx)*delta((Yb(k)-y(j))/dy);
95                 Ub(k,1)=Ub(k,1)+ux(j,i)*phi;
96                 Ub(k,2)=Ub(k,2)+uy(j,i)*phi;
97             end
98         end
99     end
100 end
101
102 function [fx, fy]=spreadf(fx, fy, Fb, Xb, Yb, x, y, dx, dy)
103     [Ny, Nx]=size(fx);
104     Nb=length(Xb);
105     for k=1:Nb
106         i0=floor(Xb(k)/dx); j0=floor(Yb(k)/dy);
107         for ii=-1:2
108             for jj=-1:2
109                 i=mod(i0+ii, Nx)+1;
110                 j=mod(j0+jj, Ny)+1;
111                 phi=delta((Xb(k)-x(i))/dx)*delta((Yb(k)-y(j))/dy);

```

```

112             fx(j,i)=fx(j,i)+Fb(k,1)*phi*dx*dy;
113             fy(j,i)=fy(j,i)+Fb(k,2)*phi*dx*dy;
114         end
115     end
116 end
117 end
118
119
120 function val=delta(r)
121     r=abs(r);
122     if r < 1
123         val=0.125*(3-2*r+sqrt(1+4*r-4*r^2));
124     elseif r < 2
125         val=0.125*(5-2*r-sqrt(-7+12*r-4*r^2));
126     else
127         val=0;
128     end
129 end

```