AM205 Final Project
Fluid Simulations with the Lattice Boltzmann BGK Model

Danyun He, Jiayin Lu, Rong Liu

Abstract This paper uses the lattice Boltzmann equation (LBE) with Bhatnagar–Gross–Krook (BGK) collision operator to simulate imcompressible fluid on a 2D plane. After laying out the general equation, we outline the numerical methods and implementation algorithm to use a mesh to update the probability distribution function and thus simulate the streaming fluid. Special attention is given to the boundary cases, and two popular methods, on-grid bounce-back and Zou-He fixed velocity or pressure, are introduced, which we will also use in the later implementation. Then, we go into more details to demonstrate the application of Lattice Boltzmann model (LBM). We first simulate a steady plane Poiseuille Flow to verify the validity of our LBM algorithm. Then we simulate fluid passing an infinite cylinder with varying Reynolds numbers, and explore the different characteristics associated with different Reynolds numbers. Lastly, we explore two creative examples of flow passing complex geometries using LBM. In the end, we use Chapman-Enksog analysis to demonstrate how the LBE behave on the macroscopic Navier-Stokes level.

1 Introduction

Conventionally, researchers have used either finite volumn or finite element numerical method based on the Navier-Stokes Equation to simulate fluid dynamics in a macroscopic way. In the past two decades, however, some researchers tried to use a mesoscopic approach and apply the Boltzmann Equation on fluid simulation. The Boltzmann equation was initially developed for kinetic theory of gases. The intuition behind the method is to consider the fluid as small groups of particles. The resulting LBE has a few advantages: it allows algorithm parallelization, and it is simpler to implement than conventional numerical methods based on Navier-Stokes equation.

Various papers have been devoted to the different stages in development, application, and generalization of LBM. He and Luo [1] provided a rigorous derivation of LBE from the Boltzmann Equation. Zou and He [2] investigated different boundary conditions for LBM. Bao and Meskas [3] discuss their implementation process of LBM with great details.

In Section 2 and 3 of this paper, we introduce the general Boltzmann Equation and how we apply it on a mesh to solve for numerical simulations. We also introduce two boundary conditions and their implementation. Further, we discuss how we handle the corner nodes in the implementation. In Section 4, we first simulate the steady plane Poiseuille flow with LBM to examine our model's validity. Then we proceed to simulate Poiseuille flow passing a cylinder with varying Reynolds number and investigate their different bahaviors. We also provide two creative simulations of Poiseuille flow passing through complex geometries. The videos for our simulations are provided in Appendix. In Section 5, we derive Navier-Stokes Equation from the Boltzmann Equation, and show that the mesoscopic view on fluid with LBE and the macroscopic view on fluid with Navier-Stokes are actually equivalent. Lastly, in Section 6, we conclude with our current implementation's limitations, potential improvements and directions of future investigation to further our knowledge in the topic.

2 Governing Equation

2.1 The Boltzmann Equation

We need to first define a one-particle probability distribution function (PPDF), $f(r, e, t)$, to describe the probability that the particle at location r and time t will move in the e direction. We get the

Boltzmann transport equation [4]:

$$\frac{\partial f}{\partial t} + \vec{u} \cdot \nabla f = (\partial_t f)_{coll}$$

where the left hand side describes the streaming process, and the right hand side describes the collision process.

2.2 Application in Fluid Dynamics

Generally, the simple BGK collision operator is used in LBM, which is given by:

$$(\partial_t f)_{coll} = -\frac{1}{\tau}(f - f^{eq})$$

Therefore, the Boltzmann transport equation becomes:

$$f((x,y) + c\vec{e}\Delta_t, e, t + \Delta_t) - f((x,y), e, t) = -\frac{f((x,y), e, t) - f^{eq}((x,y), e, t)}{\tau} \tag{1}$$

Here, f^{eq} is a local equilibrium value for particles with the parameters location r, direction e, and time t, and τ is the relaxation time, determined by the fluid viscosity.

3 Numerical Methods

3.1 Discretization

In order to perform numerical computation, we confine the particles to discrete nodes in a mesh. Since we are interested in simulating flow on a 2D plane, we will proceed with the D2Q9 model, which incorporates nine velocity directions in two dimensions. An illustration of the ordering of the directions is given in Figure [1], and each direction is associated with a microscopic velocity $\vec{e}_i$ as listed and illustrated below:

$$\vec{e}_i = \begin{cases} (0,0), & i = 0 \\ (\cos[(i-1)\pi/2], \sin[(i-1)\pi/2]), & i = 1,2,3,4 \\ (\sqrt{2}\cos[(i-5)\pi/2 + \pi/4], \sqrt{2}\sin[(i-5)\pi/2 + \pi/4]), & i = 5,6,7,8 \end{cases}$$

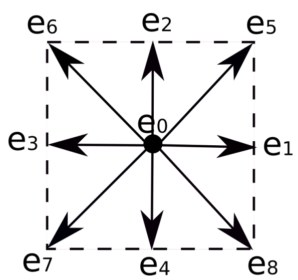

Figure 1: Directions of e

The two main steps of LBM are streaming and collision. In streaming, particle populations are moved in the corresponding direction of $\vec{e}_i$ as shown in Figure [2]. Because we have limited directions and discrete locations, for simplicity, we use $f_i((x,y),t)$ instead of $f(r,e,t)$ to denote the probability of streaming in direction e_i of the particle at location (x,y) in the mesh at time t. The streaming update is then given by:

$$f_i((x,y)+\vec{e}\Delta_t,t) = f_i((x,y),t) \tag{2}$$

If f_i is on the boundary, the streaming update of f_i is calculated differently depending on the boundary conditions, which we will give a detailed discussion later.

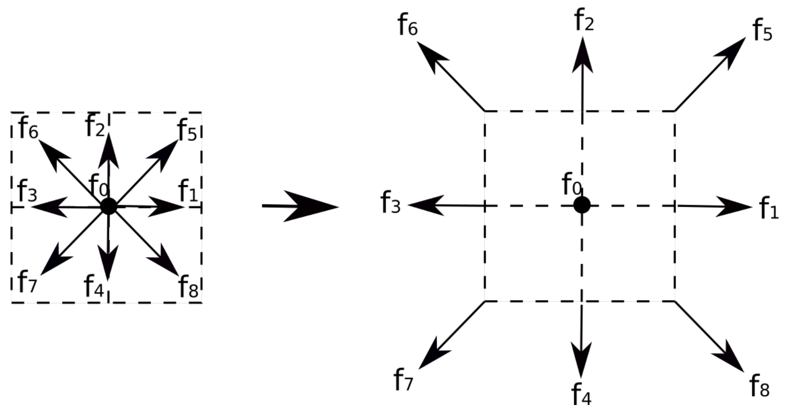

Figure 2: The Streaming Process

After streaming, with the updated f_i, we can also calculate the macroscopic fluid density ρ and velocity $\vec{u}$ as follows:

$$\rho((x,y),t) = \sum_{i=0}^{8} f_i((x,y),t) \tag{3}$$

$$\vec{u}((x,y),t) = \frac{1}{\rho}\sum_{i=0}^{8} c f_i((x,y),t)\vec{e}_i \tag{4}$$

where c is the lattice speed

$$c = \frac{\Delta x}{\Delta t}$$

The lattice spped is related to the speed of sound in fluid (c_s). In our isothermal model, the relationship of them is [5]:

$$c_s^2 = \frac{c^2}{3}$$

Lastly, we can apply the collision process to update f_i for the next time step. From Eq[1], we get an algorithm for the collision process to update the PPDF's across the mesh:

$$f_i((x,y)+\vec{e}\Delta_t,t+\Delta t) = f_i((x,y),t) - \frac{f_i((x,y),t) - f_i^{eq}((x,y),t)}{\tau} \tag{5}$$

where f_i^{eq}, the local equilibrium function that only depends on the local density ρ and velocity u. We can choose it to be

$$f_i^{eq} = w_i\rho[1 + 3(\vec{e}_i\cdot\vec{u}) + \frac{9}{2}(\vec{e}_i\cdot\vec{u})^2 - \frac{3}{2}\vec{u}\cdot\vec{u}] \tag{6}$$

where

$$w_i = \begin{cases} \frac{4}{9}, & i = 0 \\ \frac{1}{9}, & i = 1,2,3,4 \\ \frac{1}{36}, & i = 5,6,7,8 \end{cases}$$

We summarize the terms in D2Q9 model in Table[1].

i	0	1	2	3	4	5	6	7	8
w_i	$\frac{4}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$
e_{ix}	0	1	0	-1	0	1	-1	-1	1
e_{iy}	0	0	1	0	-1	1	1	-1	-1

Table 1: D2Q9 model

3.2 Algorithm

We use the stream-and-collide algorithm [3] as follows:

1. Initialization: set values of ρ, $\vec{u}$, f_i^{eq} and f_i at each grid point.
2. Streaming: Streaming process to update f_i as given by Eq[2]. If f_i is on the boundary, the streaming update of f_i is given special attention and calculated differently depending on the boundary conditions chosen.
3. Calculate ρ and $\vec{u}$ using Eq[3] and Eq[4].
4. Compute the equilibrium function f_i^{eq} by Eq[6]; If desired, calculate ρ and $\vec{u}$ based on f_i^{eq} and output and store ρ and $\vec{u}$ of this step.
5. Collision: update f_i using collision equation Eq[5].
6. Repeat step 2 to 5.

3.3 Boundary Conditions (BC)

Boundary nodes are missing the density distributions from one side, so their density distribution functions cannot be updated with the regular propagation algorithm. We introduce several most used boundary conditions here. As discussed by Zou and He [2], we can apply the bounce-back rule on no-slip boundaries, and with given velocity/pressure, we can derive the fixed velocity BC, fixed pressure BC, and also the special treatment for corner nodes.

3.3.1 On-Grid Bounce-Back

The first BC is referred to as on-grid bounce-back. It is commonly used to handle non-slip condition at the boundary. As the name suggests, when a f_i hits a boundary, it bounces off the boundary and reverses to the opposite direction. This process is illustrated in figure[3]. f_0, f_2, f_5,f_1,f_8,and f_4 are able to propagate in the regular way during streaming. On the other hand, f_3, f_6, and f_7, which point into the boundary, point to the opposite of their original directions and back into the fluid after streaming. Furthermore, this method is called 'on-grid' because it aligns the boundaries with the grid.

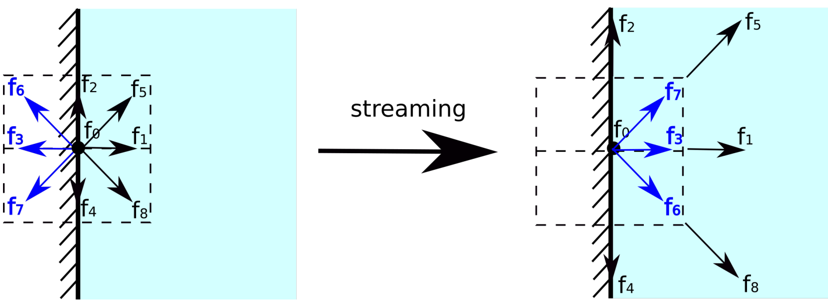

Figure 3: BC: Bounce Back

On-grid bounce-back has a great advantage that during its implementation, the algorithm only needs to determine whether a certain node is next to a node in the boundary, and it does not need to know the orientation of the entire boundary. This feature allows us to use this BC to handle boundaries with complex geometric shapes. This advantage will be further demonstrated in Section 4.3, where we provide simulations of LBM for Poiseuille flow passing complex geometries.

3.3.2 Zou-He Fixed Velocity

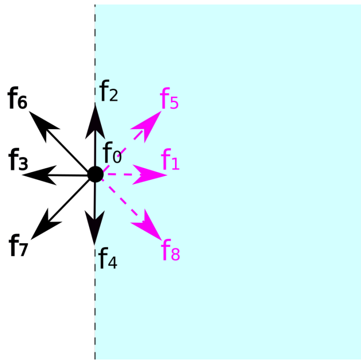

Figure 4: BC: Zou-He Fixed Pressure/Velocity

As illustrated in Figure[4], after streaming in, f_0, f_2, f_6, f_3, f_7, and f_4 are known. Suppose the velocity $u = [u, v]$ at a particular node on the wall is known, we could derive the fixed velocity BC as follows.

Using Eq[3] and Eq[4] we have:

$$f_1 + f_5 + f_8 = \rho - (f_0 + f_2 + f_3 + f_4 + f_6 + f_7) \tag{7}$$
$$f_5 - f_8 = \rho v - (f_2 - f_4 - f_6 + f_7) \tag{8}$$
$$f_1 + f_5 + f_8 = \rho u + (f_3 + f_6 + f_7) \tag{9}$$

Using Eq[7] and Eq[9] we derive the equation for density:

$$\rho = \frac{1}{1-u}[f_0 + f_2 + f_4 + 2(f_3 + f_6 + f_7)] \tag{10}$$

To determine the values of f_1, f_5 and f_8, Zou and He assume the bounce-back rule still holds for the non-equilibrium of the particle distribution ($f_i^{neq} = f_i - f_i^{eq}$) that are normal to the boundary, which gives

$$f_1 - f_1^{eq} = f_3 - f_3^{eq} \tag{11}$$

From Eq[6],

$$f_1^{eq} = \frac{\rho}{18}(2 + 6u + 6u^2 - 3v^2) \tag{12}$$
$$f_3^{eq} = \frac{\rho}{18}(2 - 6u + 6u^2 - 3v^2) \tag{13}$$

By substituting Eq[12] and Eq[13] into Eq[11], we solve f_1 where

$$f_1 = f_3 + \frac{2}{3}\rho u \tag{14}$$

Substituting Eq[14] into Eq[8] and assemble Eq[9], we can calculate f_5 and f_8:

$$f_5 = f_7 - \frac{1}{2}(f_2 - f_4) + \frac{1}{6}\rho u + \frac{1}{2}\rho v \tag{15}$$
$$f_8 = f_6 + \frac{1}{2}(f_2 - f_4) + \frac{1}{6}\rho u - \frac{1}{2}\rho v \tag{16}$$

3.3.3 Zou-He Fixed Pressure

Now suppose instead that the pressure (density) is known on the boundary and the velocity along the boundary direction is specified, we could derive the pressure (density) flow BC [2]. Taking the left boundary (inlet) in Figure[4] as an example, we assume that the pressure along the y-direction is known ($\rho = \rho_{in}$), and the y-direction velocity v is known. After streaming, f_0, f_2, f_6, f_3, f_7, and f_4 are known. The unknowns left to be solved are u, f_1, f_5, and f_8. Using Eq[3] and Eq[4] we have:

$$f_1 + f_5 + f_8 = \rho_{in} - (f_0 + f_2 + f_3 + f_4 + f_6 + f_7) \tag{17}$$

$$f_5 - f_8 = \rho_{in} v - (f_2 - f_4 - f_6 + f_7) \tag{18}$$

$$f_1 + f_5 + f_8 = \rho_{in} u + (f_3 + f_6 + f_7) \tag{19}$$

Consistency of Eq[17] and Eq[19] gives

$$u = 1 - \frac{f_0 + f_2 + f_4 + 2(f_3 + f_6 + f_7)}{\rho_{in}} \tag{20}$$

Similar to the approach in fixed velocity BC above, we use the bounce-back rule for the non-equilibrium part of the particle distribution as in Eq[11] and obtain

$$f_1 = f_3 + \frac{2}{3}\rho_{in} u \tag{21}$$

$$f_5 = f_7 - \frac{1}{2}(f_2 - f_4) + \frac{1}{6}\rho_{in} u + \frac{1}{2}\rho_{in} v \tag{22}$$

$$f_8 = f_6 + \frac{1}{2}(f_2 - f_4) + \frac{1}{6}\rho_{in} u - \frac{1}{2}\rho_{in} v \tag{23}$$

Thus, the unknowns u, f_1, f_5, and f_8 are solved.

3.3.4 Corner node

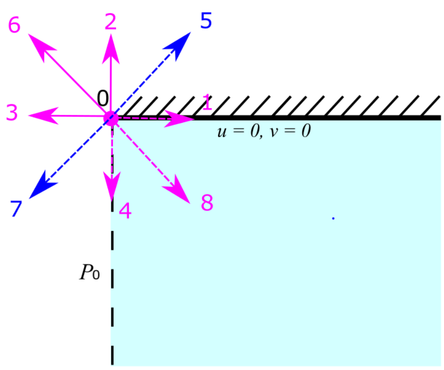

Figure 5: top-left corner node

At the corner, only three $f_i's$ are known after streaming. Thus, the derivations above can not be applied at the corner and it needs special treatment. Take the node at the top left as an example. Assume ρ is known, and $u = v = 0$. As shown in Figure[5], after streaming, f_0, f_2, f_3, f_6 are known, while f_1, f_4, f_5, f_7, f_8 are undetermined. At the corner, we have two walls, so we can obtain two equations using the bounce-back rule for the non-equilibrium part:

$$f_1 - f_1^{eq} = f_3 - f_3^{eq} \tag{24}$$

$$f_2 - f_2^{eq} = f_4 - f_4^{eq} \tag{25}$$

Assembling Eq[24], Eq[25], Eq[6], and the condition $u = v = 0$, we have

$$f_1^{eq} = f_2^{eq} = f_3^{eq} = f_4^{eq} = \frac{\rho}{9}$$

and thus obtain

$$f_1 = f_3 \quad \text{and} \quad f_2 = f_4 \tag{26}$$

The remaining undetermined values are f_5, f_7, and f_8. Apply Eq[26] to Eq[18] and Eq[19] with $u = 0$ gives

$$f_5 + f_8 = f_6 + f_7 \tag{27}$$
$$f_5 - f_8 = -f_6 + f_7 \tag{28}$$

Further, we obtain

$$f_5 = f_7 \quad \text{and} \quad f_8 = f_6 \tag{29}$$

Using Eq[29] in Eq[17], we find

$$f_5 = f_7 = \frac{1}{2}[\rho_{in} - (f_0 + f_1 + f_2 + f_3 + f_4 + f_6 + f_8)] \tag{30}$$
$$= \frac{1}{2}[\rho_{in} - (f_0 + 2f_2 + 2f_3 + 2f_6)] \tag{31}$$

4 LBM Simulations and Discussion

In this section, we use our D2Q9 LBM algorithm to simulate the steady plane Poiseuille flow, steady Poiseuille flow passing through a cylinder, and steady Poiseuille flow passing through complex geometries. Discussion on the validity of our LBM algorithm and on the exploration of different Reynolds number of fluid flows are given. The simulation results are presented by videos we made, with the links provided in Appendix.

In all our implementation below, we use LBE in its dimensionless form and set $\Delta x = 1 = \Delta t = 1$. Therefore, the lattice speed $c = \frac{\Delta x}{\Delta t} = 1$.

4.1 Steady Plane Poiseuille Flow

In this section, we test the validity of our LBM algorithm implementation. We use our code to simulate the 2D steady plane Poiseuille flow, and compare the simulation results with the expected behaviors of the flow.

4.1.1 Plane Poiseuille Flow Set Up and Solution

The 2D plane Poiseuille flow is a steady, laminar and incompressible flow in a channel of two flat, parallel and still plates. The flow is driven by pressure difference at the inlet and outlet of the channel, where the inlet pressure, P_0, is larger than the outlet pressure, P_1.

The geometry setup of Poiseuille flow is provided in Figure[6]. Here, the channel is of length L, and the two parallel plates are of distance H. We define the pressure difference as $\Delta P = P_1 - P_0$.

The Navier-Stokes equation for the Poiseuille flow is simplified by the straightforward geometry setup:

$$\mu \frac{\partial^2 u}{\partial y^2} = \frac{\partial p}{\partial x}, \ where \ \frac{\partial p}{\partial x} = \frac{\Delta P}{L}$$

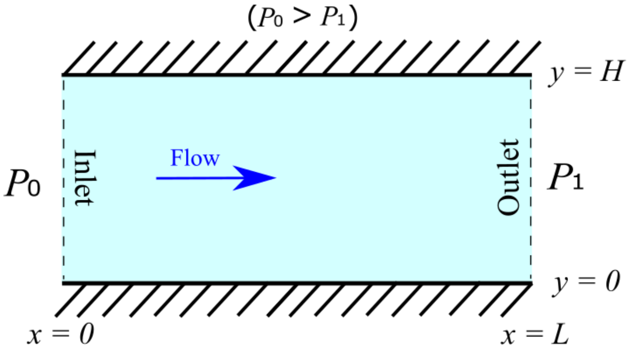

Figure 6: 2D Steady Plane Poiseuille Flow

The velocity components of the Poisseuille flow, u, v, have no horizontal variations, and $v \equiv 0$. Also, assuming no-slip condition, the velocity on the plates are 0 for all time.

Because of the simplicity of the model, the exact solution for Poiseuille flow's steady state velocity can be derived [6], and is given by:

$$u(y) = -\frac{1}{2\mu}\frac{dp}{dx}(hy - y^2) \tag{32}$$

where μ is the dynamic viscosity related to the kinematic one by $\mu = \nu \cdot \rho$.

4.1.2 LBM: Initial Conditions, BCs, and Corner Nodes

In our simulation, our initial conditions are demonstrated in Figure[7] and Figure[8] where

$$u(x, y, 0) = v(x, y, 0) = 0$$

$$p(0, y, 0) = P_0 \quad p(L, y, 0) = P_1$$

$$p(x, y, 0) = P_{avg} \ for \ x \neq 0 \ and \ x \neq \ L, \ where \ P_{avg} = \frac{P_0 + P_1}{2}$$

With the initial pressure and velocity, we can then use our LBM algorithm, calculate f_i^{eq} at time $t = 0$, and initiate our f_i at time $t = 0$ to be the values of f_i^{eq}, which is a common practice in LBM implementation.

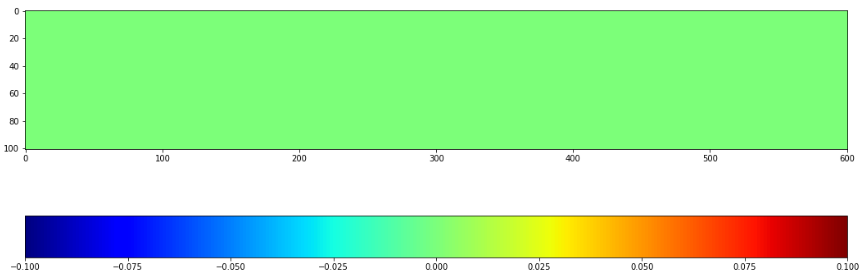

Figure 7: initial velocity

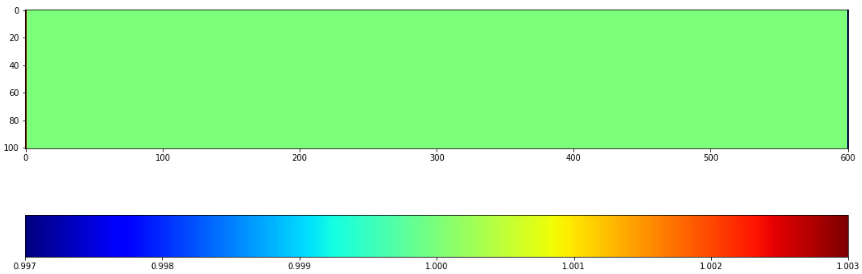

Figure 8: initial density

Our BCs are

$$p(0, y, t) = P_0 \quad p(L, y, t) = P_1$$

$$u(x, 0, t) = v(x, 0, t) = 0 \quad u(x, H, t) = v(x, H, t) = 0$$

We apply Zou-He fixed pressure BC at both inlet and outlet, and we apply on-grid bounce-back BC on the upper and lower plates

For the four corner nodes, some special treatments of the streaming step are needed. Take the top-left bottom node as an example, we use $u = v = 0$ subject to the BC, and $P_{in} = P_0$ subject to the fixed pressure BC at the inlet of the channel. Using the corner condition presented in section 3.3, we derive the equations for top-left corner node as:

$$f_1 = f_3, \quad f_8 = f_6, \quad f_4 = f_2$$
$$f_5 = f_7 = \frac{1}{2}(P_0 - (f_0 + f_1 + f_2 + f_3 + f_4 + f_6 + f_8))$$

The parameters we use in our simulation are: $L = 600$, $H = 100$ and $\Delta P = -0.006$. The criteria of reaching steady state is [3]:

$$\frac{\sum_{ij} |u_{ij}^{n+1} - u_{ij}^n|}{\sum_{ij} |u_{ij}^{n+1}|} \leq 5.0 \times 10^{-9}$$

where u_{ij}^n is the velocity in the x-direction on the grid point (x_i, y_j) at the n^{th} time step.

4.1.3 LBM: Simulation Results and Algorithm Validity Check

Figure[9] and Figure[10] show the velocity and density information of the steady state Poiseuille flow that we simulated. As seen from the plot, at steady state, the Poiseuille flow obtains maximum velocity at $y = \frac{1}{2}H$ in the channel, and decreases gradually, until reaching zero velocity at the upper and lower plates. Also, the pressure gradient in steady state is transitioning from P_0 to P_1 smoothly. The two simulation plots match the expected velocity and density behaviors for steady state Poiseuille flow.

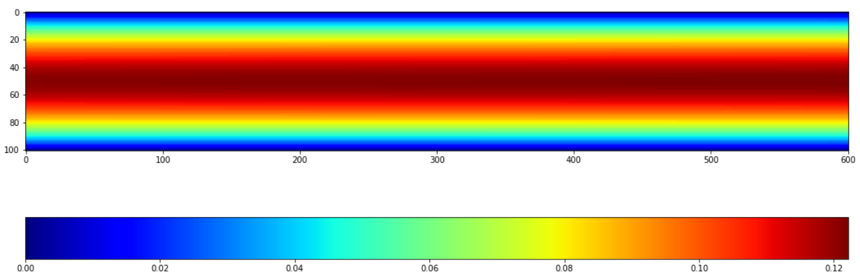

Figure 9: steady-state velocity

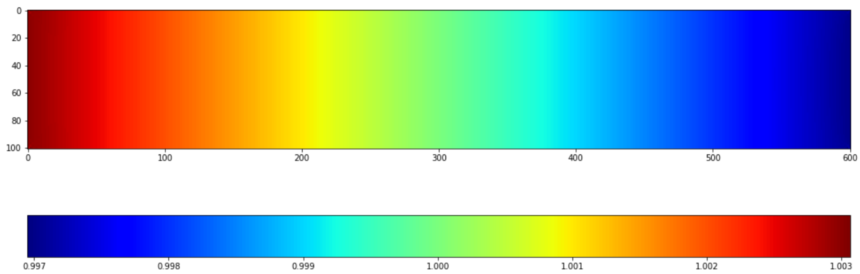

Figure 10: steady-state density

To further validate our LBM algorithm, we compare the steady-state velocity from our simulation to the exact velocity solution. As seen in Figure[11], a parabolic velocity profile of the Poiseuille flow is observed as expected, and our simulation velocity match with exact velocity solution closely.

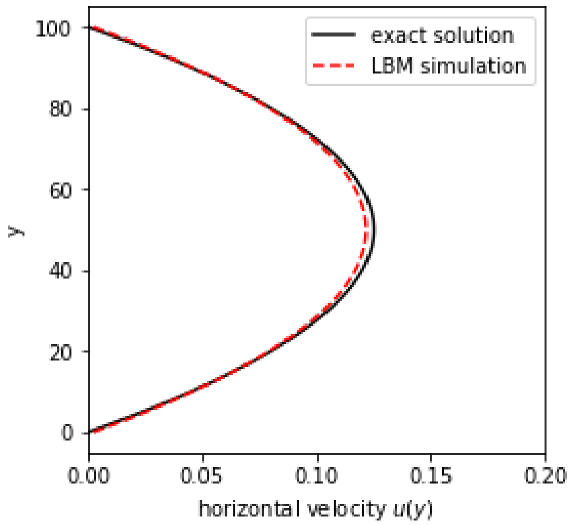

Figure 11: Comparison of simulated and exact velocity profile of Poiseuille flow

4.2 Poiseuille Flow Passing a Cylinder

After testing that our LBM algorithm is valid, in this section, we simulate the steady plane Poiseuille flow passing through an infinite circular cylinder immersed in the flow. We vary the flow's Reynolds number, and explore the interesting different behaviors associated with flows of different Reynolds numbers.

4.2.1 Geometry and Set Up

As shown in Figure[12], we start with simulating a Poiseuille flow with $\tau = 0.6$ till its steady state in a channel of $H = 100$, $L = 600$, and pressure difference $\Delta P = P_1 - P_0$.

We then immerse a circular cylinder with radius $r = 4$ in the channel, with origin at $(x = \frac{1}{10}L, y = \frac{1}{2}H)$. We use the steady state Poiseuille flow velocity and density information to initialize the flow past cylinder simulation, with fixed steady-state velocity profile at the left boundary (inlet) and fixed pressure P_1 at the right boundary (outlet).

Three simulations of the flow were performed, each with a different Reynolds number. Specifically, we look at the different flow behaviors, in terms of velocity and density, at $Re = 1$, 20 and 80, respectively.

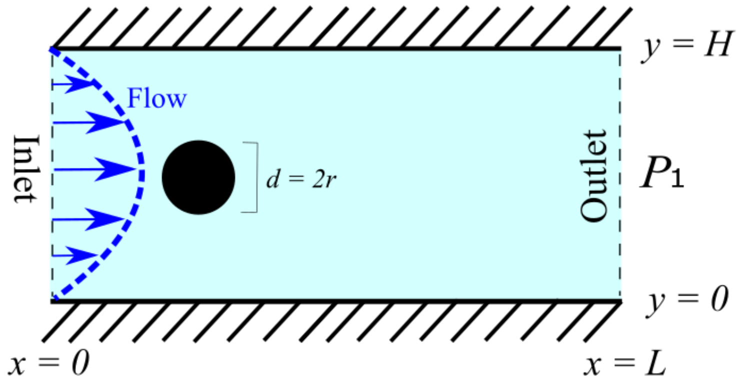

Figure 12: Steady Poiseuille Flow Past Cylinder

4.2.2 Reynolds Number and its Implementation

Reyolds number is an important dimensionless value in fuild dynamics. It is the ratio of the fluid's inertial forces to viscous forces, describing how laminar or turbulent the fluid flow is. Reynolds number is given by:

$$Re = \frac{\rho v l}{\mu} = \frac{vl}{\nu}$$

where:
v: characteristic velocity of the fluid;
l: characteristic length;
ρ: density of the fluid;
μ: dynamic viscosity of the fluid;
ν: kinematic viscosity of the fluid;

By dimensional analysis, it is shown that for a body of given shape, the details of the flow actually only depend on Re. In another word, fluid systems with the same Reynolds number have the same flow characteristics, even if the fluid, speed and characteristic lengths vary.

When Re is small, viscous forces dominate the flow, and we expect the flow to be laminar. We expect to observe smooth and steady symmetric flows in our flow past cylinder simulations for small Re. As Re increases, inertial forces gain in significance, and the behavior of the flow should become more turbulent in comparison. We expect to observe unsteady and asymmetric flow for large Re.

For our simulation, we want to vary Re and investigate the flow behaviors. To achieve this goal, we can derive a relationship between ΔP, the pressure difference at inlet and outlet of the steady plane Poiseuille flow setting, and Re, the Reynolds number for the flow past cylinder setting. The derivation detail of ΔP based on a chosen Re is given below:

Re of flow past cylinder can be calculated as [6]:

$$Re = \frac{d \cdot u_{ave}}{\nu} \tag{33}$$

where the characteristic length is cylinder diameter $d = 2r$, and average velocity of the steady Poiseuille flow u_{ave} is used in the calculation as velocity.

u_{ave} is calculated by integrating $u(y)$ in Eq[32] from $H = 0$ to $H = 100$, and dividing the results by the channel's height $H = 100$:

$$u_{ave} = \frac{\int_0^H u(y)dy}{H} = -\frac{1}{2\mu}\frac{dp}{dx}\frac{H^2}{6} = -\frac{1}{2\mu}\frac{\Delta P}{L}\frac{H^2}{6} \tag{34}$$

Combining Eq[33] and Eq[34], we have:

$$\Delta P = -\frac{12}{dh^2}\mu\nu Re \cdot L$$

Because:

$$\mu = \nu \cdot \rho_{ave} = \nu \cdot (\frac{P_{ave}}{c_s^2}) = \nu \cdot (\frac{3}{c^2} \cdot P_{ave})$$

Substituting $c = 1$ gives:

$$\mu = \nu \cdot (3 \cdot P_{ave})$$

We can further write:

$$\Delta P = -\frac{36}{h^2 \cdot d}\nu^2 \cdot Re \cdot L \cdot P_{ave}$$

In our implementation, the average pressure in the steady plane Poiseuille flow, $P_{ave} = 1$, and the kinematic viscosity $\nu = \frac{2\tau-1}{6}\frac{(\Delta x)^2}{\Delta t} = \frac{2\tau-1}{6}$.

With ΔP calculated, we can then set the inlet pressure, P_0, and the outlet pressure, P_1, for the plane Poiseuille flow:

$$P_0 = 1 - \frac{\Delta P}{2} \qquad P_1 = 1 + \frac{\Delta P}{2}$$

4.2.3 LBM: Initial Conditions, BCs, and Corner Nodes

As discussed previously, the initial conditions for our flow past cylinder simulation is the steady-state Poiseuille flow velocity and density profiles, with the Poiseuille flow channel ΔP determined by our chosen Re of the flow past cylinder setting. We can then calculate f_i^{eq} based on the velocity and density information and set $f_i = f_i^{eq}$ at time $t = 0$.

The BCs for our simulations are:

- No-slip condition on the upper plate, lower plate and on the cylinder surface. In another word, we require $u = 0, v = 0$ on these surfaces. We apply on-grid bounce-back method on the surfaces to achieve the condition.
- Fixed steady-state parabolic Poiseuille flow velocity profile at the inlet, and fixed pressure P_1 at the outlet. We apply Zou-He BC to achieve the fixed velocity and pressure conditions.

Special attention is needed while applying Zou-He fixed pressure BC at the outlet, and while treating the top left and bottom left corner nodes (corner nodes at inlet boundary).

When we apply Zou-He fixed pressure BC at the outlet, we need to know v, the velocity in the y direction, at those boundary nodes. However, different from the simple plane Poiseuille flow, v here is not necessarily zero. Instead, we can apply the Neumann boundary condition here for v. As shown in Figure[13], if the nodes directly to the left of the outlet boundary nodes have vertical velocity $\tilde{v}(y)$ and density $\tilde{p}(y)$, and we assume that the y-direction fluid flux from these nodes is 0, then we have, for the velocity and density at the outlet boundary $v(y)$, P_1:

$$\tilde{v}(y) \cdot \tilde{p}(y) = v(y) \cdot P_1$$

So:

$$v(y) = \frac{\tilde{v}(y) \cdot \tilde{p}(y)}{P_1}$$

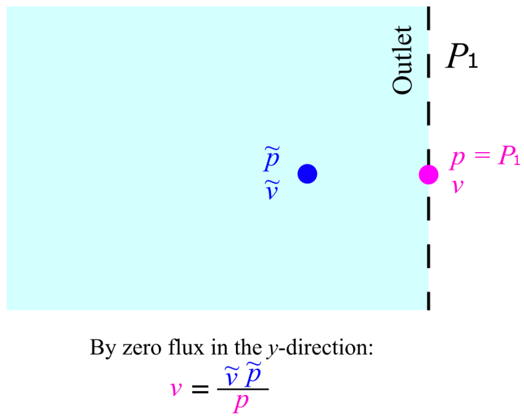

Figure 13: Neumann BC: Zero Flux in the y-direction at outlet

For the corner nodes at the inlet boundary, we treat them similarly as described in corner nodes of the Poiseuille flow section. However, here the pressure information on these corner nodes is missing. Our solution is to extrapolate the pressure information from their nearest nodes on the inlet boundary. For instance, for the top left corner node, we use the pressure information from the node directly below it to calculate its missing f_i's.

4.2.4 LBM: Simulation Results and Discussion on Flow Behaviors

Figure[14], [15] and [16] show the density information of flow past cylinder at different Reynolds number. When $Re = 1$, we observe high density in front of the cylinder, and low density at the back of the cylinder. When Re increase, at $Re = 20$, the high density still occurs at the front of the cylinder, but low density has moved to above and below the cylinder. In both of the cases discussed, the density distribution are symmetrical above and below the cylinder. As Re increases to $Re = 80$, the density situation becomes more interesting. Density is not symmetrically distributed above and below the cylinder anymore. High density is still observed at the front of the cylinder; lower, asymmetric density wake is observed behind the cylinder.

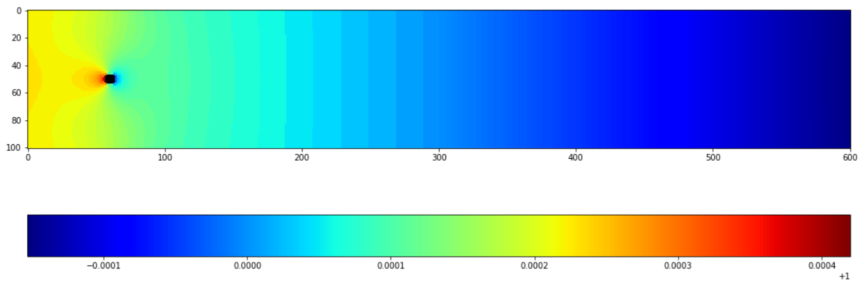

Figure 14: Re = 1: density at time = 60001

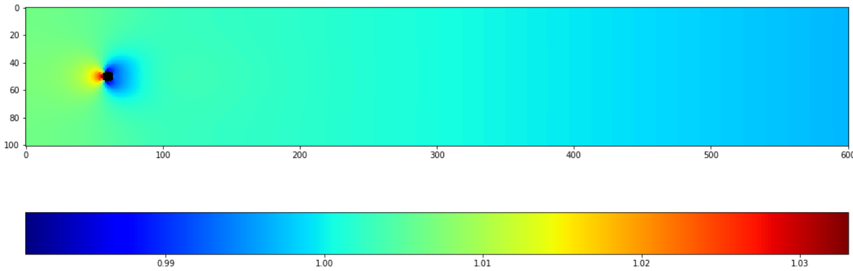

Figure 15: Re = 20: density at time = 60001

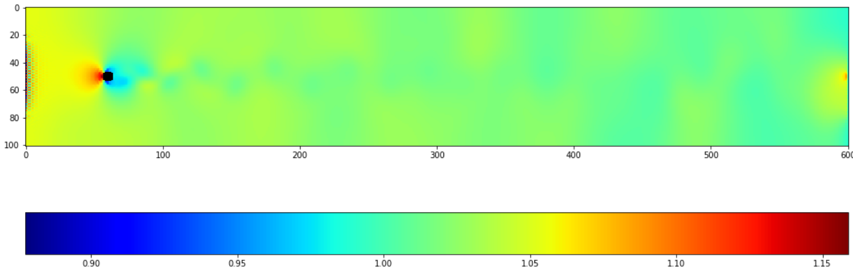

Figure 16: Re = 80: density at time = 60001

Figure[17], [18], [19] show the velocity magnitude information of the three flows. As seen from the plots, when $Re = 1$, the velocity magnitude is identical above and below the cylinder, and there is a smaller velocity region around the cylinder. When Re increases to 20, velocity magnitude is still symmetric above and below the cylinder. However, now there appears to be a stable region of vortices downstream of the cylinder. As Re grows to 80, the flow is separated by the cylinder and the velocity magnitude is not symmetric above and below the cylinder anymore. The eddies from above and below the cylinder are mixing together, and a swirling wake appears behind the cylinder. Moreover, the length of the wake behind the cylinder increases as the wake move further away from the cylinder.

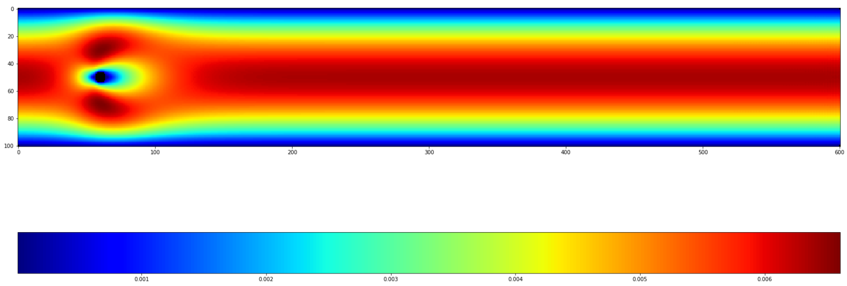

Figure 17: Re = 1: velocity at time = 60001

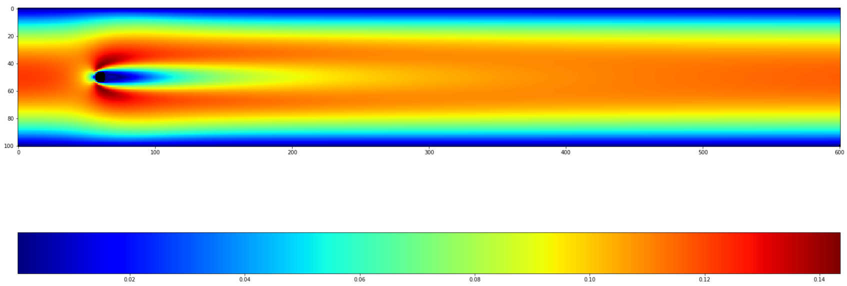

Figure 18: Re = 20: velocity at time = 60001

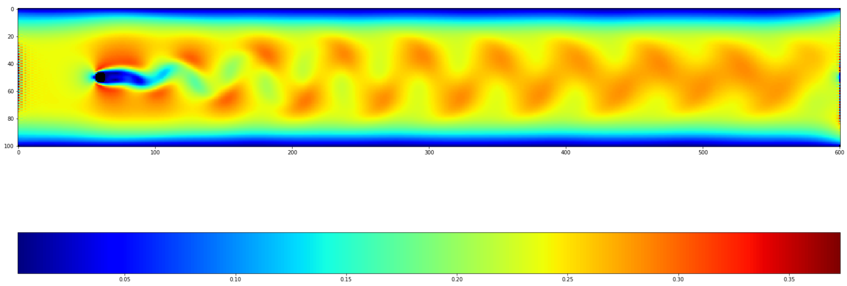

Figure 19: Re = 80: velocity at time = 60001

The behaviors of the density and velocity of the flows corresponding to different Reynolds number match our expectations. As discussed in the previous Reynolds number section, we expect the flow to be more laminar and stable when the flow has a small Re, and become more unstable and turbulent with a large Re. The transition is observed in our three simulations, where a wake past the cylinder develops as Re increases.

4.3 Poiseuille Flow Passing Complex Geometries

We have mentioned that the algorithm of LBE and on-grid bounce-back are especially suitable to handle complex boundaries. Therefore, in this section, we offer two creative examples of fluid simulation passing complex geometries. We use the same implementation steps as in the last section. In terms of the setting, we use the same mesh dimension and number of time steps as before, and we use an Re number of roughly 15. In the first example, we place obstacles of the shape "AM205", and in the second example, we place obstacles of the shapes moon and stars.

The resulting plots of density and velocity are given in Figures[20], [21], [22], and [23]. As expected, the general density gradually decreases as the fluid passes around the obstacles. Moreover, when the fluid is blocked by the obstacles, the velocity is lower behind them and higher around them.

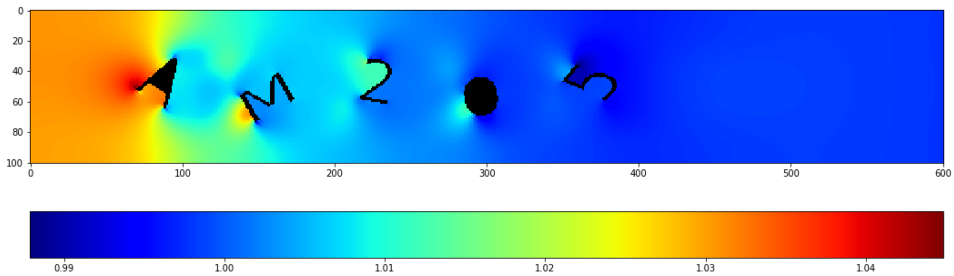

Figure 20: text simulation(Re = 15): density at time = 60001

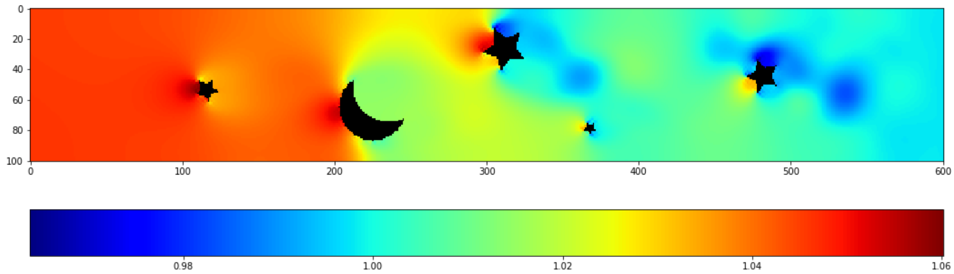

Figure 21: sky simulation(Re = 15): density at time = 60001

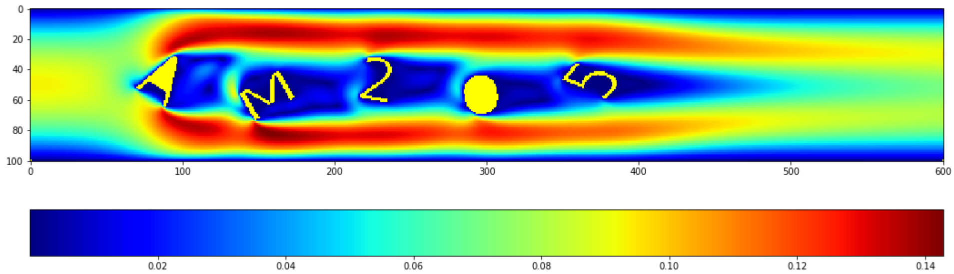

Figure 22: text simulation(Re = 15): velocity at time = 60001

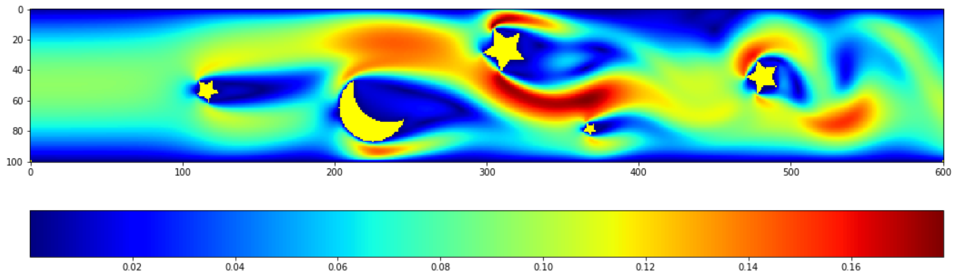

Figure 23: sky simulation(Re = 15): velocity at time = 60001

5 Link to Navier-Stokes

There are several approaches to link LBE to the macroscopic equations. Chapman-Enskog analysis is a classic tool to derive the Navier-Stokes equations. This analysis is named after two mathematical physicists, Sydney Chapman and David Enskog, who developed this method independently in 1916 and 1917 respectively. Another common technique was introduced by Sone [7] using asymptotic analysis. This approach studies the Boltzmann equation with small Knudsen number and finite Reynolds numbers. In 1949, Grad proposed the Hermite expansion series as a way to approximate the solutions of the Boltzmann equation in terms of the Hermite polynomials [8]. Ikenberry and Truesdell introduced a systematic procedure using Maxwellian iteration technique in 1956, which could find a closure for the fluxes in the macroscopic flow equations [9].

In this paper, the classic Chapman-Enskog theory is presented [5]. We consider low compressible viscous flow past an arbitrary finite body in two or three dimensions. To achieve this purpose, we use Greek indices for the Cartesian indices x, y, and z in this section. Moreover, the following notations are used:

$$\rho = \text{density},\ p = \text{pressure},\ u = \text{velocity}.$$

The governing Navier-Stokes equations are:

- Continuity equation

$$\frac{\partial \rho}{\partial t} + \nabla(\rho u) = 0 \tag{35}$$

- Momentum equation

$$\partial_t(\rho u_\alpha) + \partial_\beta(\rho u_\alpha u_\beta) = -\partial_\alpha(c_s^2\rho) + \partial_\beta(2\nu\rho S_{\alpha\beta}) \tag{36}$$

where $S_{\alpha\beta} = \frac{1}{2}(\partial_\alpha u_\beta + \partial_\beta u_\alpha)$ is the strain-rate tensor. The pressure is $p = c_s^2\rho$, where $c_s^2 = c^2/3$, and $\nu = \frac{2\tau-1}{6}\delta$ is the kinematic viscosity.

We will present the Chapman-Enskog derivation of the Navier-Stokes Equation. Let $f_i(x,t)$ denote the density distribution function along e_i at (x,t). ϵ is used to indicate the Knudsen number K_n. We have perturbation expansion of f_i around f_i^{eq}:

$$f_i = \sum_{n=0}^{\infty} \epsilon^n f_i^{(n)} = f_i^{(0)} + \epsilon f_i^{(1)} + \epsilon^2 f_i^{(2)} + ... \tag{37}$$

where $f_i^{(0)} = f_i^{eq}$. In the perturbation analysis, the terms at the two lowest order, $f_i^{(0)} + \epsilon f_i^{(1)}$ is sufficiently accurate to represent the system, while the higher order terms act as correction terms. Therefore, we will use the first two terms in our later analysis.

Recall the LBE with BGK collision operater is

$$f_i(x + ce_i\Delta t, t + \Delta t) - f_i(x,t) = -\frac{\Delta t}{\tau}(f_i(x,t) - f_i^{eq}(x,t)) \tag{38}$$

Again, we denote the non-equilibrium part of f as $f^{neq} = f - f^{eq}$. By the mass and momentum conservation of BGK operator, we obtain

$$\sum_i f_i^{neq} = 0 \quad \text{and} \quad \sum_i ce_i f_i^{neq} = 0 \tag{39}$$

We further strengthen the conservation by assuming that

$$\sum_i f_i^{(n)} = 0 \quad \text{and} \quad \sum_i ce_i f_i^{(n)} = 0 \quad \text{for} \quad n \geq 1 \tag{40}$$

After performing Taylor expansion on LBE (Eq[38]), we have the discrete-velocity Boltzmann equation

$$\Delta t(\partial_t + e_{i\alpha}\partial_\alpha)f_i + \frac{\Delta t^2}{2}(\partial_t + e_{i\alpha}\partial_\alpha)^2 f_i + O(\Delta t^3) = -\frac{\Delta t}{\tau} f_i^{neq} \tag{41}$$

$O(\Delta t^3)$ is very small, so we neglect it and subtract $\frac{\Delta t}{2}(\partial_t + e_{i\alpha}\partial_\alpha)$ in Eq[41], and we obtain

$$\Delta t(\partial_t + e_{i\alpha}\partial_\alpha)f_i = -\frac{\Delta t}{\tau} f_i^{neq} + \frac{\Delta t^2}{2\tau}(\partial_t + e_{i\alpha}\partial_\alpha) f_i^{neq} \tag{42}$$

By multi-scale expansions, we expand the time derivative in terms of K_n and similarly label the spatial derivative without expanding.

$$\partial_t = \sum_{n=1} \epsilon^n \partial_t^{(n)} = \epsilon\partial_t^{(1)} + \epsilon^2\partial_t^{(2)} + ... \quad \text{and} \quad \partial_\alpha = \epsilon\partial_\alpha^{(1)} \tag{43}$$

Substituting Eq[37] and Eq[43] into Eq[42], we have

$$\begin{aligned} &\Delta t(\epsilon\partial_t^{(1)} + \epsilon^2\partial_t^{(2)} + e_{i\alpha}\epsilon\partial_\alpha^{(1)})(f_i^{eq} + \epsilon f_i^{(1)}) \\ &= -\frac{\Delta t}{\tau}(\epsilon f_i^{(1)} + \epsilon^2 f_i^{(2)}) + \frac{\Delta t^2}{2\tau}(\epsilon\partial_t^{(1)} + \epsilon^2\partial_t^{(2)} + e_{i\alpha}\epsilon\partial_\alpha^{(1)})(\epsilon f_i^{(1)} + \epsilon^2 f_i^{(2)}) \end{aligned} \tag{44}$$

where $f_i^{neq} = f_i - f_i^{eq} = \epsilon f_i^{(1)} + \epsilon^2 f_i^{(2)}$.

Expanding the equation in terms of K_n, we can find

$$O(\epsilon) : (\partial_t^{(1)} + e_{i\alpha}\partial_\alpha^{(1)})f_i^{eq} = -\frac{1}{\tau}f_i^{(1)}, \tag{45a}$$

$$O(\epsilon^2) : \partial_t^{(2)} f_i^{eq} + (\partial_t^{(1)} + e_{i\alpha}\partial_\alpha^{(1)}) \left(1 - \frac{\Delta t}{2\tau}\right) f_i^{(1)} = -\frac{1}{\tau}f_i^{(2)} \tag{45b}$$

Now we can connect LBE to macroscopic moments. In our D2Q9 model, we set $u = (u_x, u_y)^T$, $u^2 = u_x^2 + u_y^2$ and apply assigned weights and microscopic velocities (Table[1]) to the equilibrium function Eq[6]; we obtain f_i^{eq} for $i = 0, 1, 2, ..., 8$ as following:

$$f_0^{eq} = \frac{2\rho}{9}(2 - 3u^2) \tag{46a}$$

$$f_1^{eq} = \frac{\rho}{18}(2 + 6u_x + 9u_x^2 - 3u^2) \tag{46b}$$

$$f_2^{eq} = \frac{\rho}{18}(2 + 6u_y + 9u_y^2 - 3u^2) \tag{46c}$$

$$f_3^{eq} = \frac{\rho}{18}(2 - 6u_x + 9u_x^2 - 3u^2) \tag{46d}$$

$$f_4^{eq} = \frac{\rho}{18}(2 - 6u_y + 9u_y^2 - 3u^2) \tag{46e}$$

$$f_5^{eq} = \frac{\rho}{36}[1 + 3(u_x + u_y) + 9u_xu_y + 3u^2] \tag{46f}$$

$$f_6^{eq} = \frac{\rho}{36}[1 - 3(u_x - u_y) - 9u_xu_y + 3u^2] \tag{46g}$$

$$f_7^{eq} = \frac{\rho}{36}[1 - 3(u_x + u_y) + 9u_xu_y + 3u^2] \tag{46h}$$

$$f_8^{eq} = \frac{\rho}{36}[1 + 3(u_x - u_y) - 9u_xu_y + 3u^2] \tag{46i}$$

Using the equilibrium distribution[46], we can find the equilibrium moments explicitly with $c_s^2 = \frac{1}{3}$:

$$\Pi^{eq} = \sum_i f_i^{eq} = \rho \tag{47a}$$

$$\Pi_\alpha^{eq} = \sum_i f_i^{eq} e_{i\alpha} = \rho u_\alpha \tag{47b}$$

$$\Pi_{\alpha\beta}^{eq} = \sum_i f_i^{eq} e_{i\alpha} e_{i\beta} = \rho c_s^2 \delta_{\alpha\beta} + \rho u_\alpha u_\beta \tag{47c}$$

$$\Pi_{\alpha\beta\gamma}^{eq} = \sum_i f_i^{eq} e_{i\alpha} e_{i\beta} e_{i\gamma} = \rho c_s^2 (u_\alpha \delta_{\beta\gamma} + u_\beta \delta_{\alpha\gamma} + u_\gamma \delta_{\alpha\beta}) \tag{47d}$$

Note that in our model D2Q9, we have two dimensions (e.g. α and β). For $\Pi_{\alpha\beta\gamma}^{eq}$ in [47], γ could equal to α or β, and we should neglect the term $u_\gamma \delta_{\alpha\beta}$. We multiply Eq[45a] by 1, $e_{i\beta}$, and $e_{i\alpha} e_{i\beta}$ respectively, and then sum all the terms over i. By applying the equilibrium moments in [47] and the conservation rule in Eq[40], we obtain the $O(\epsilon)$ moment equations:

$$\partial_t^{(1)} \rho + \partial_\alpha^{(1)} (\rho u_\alpha) = 0 \tag{48a}$$

$$\partial_t^{(1)} (\rho u_\beta) + \partial_\alpha^{(1)} \Pi_{\alpha\beta}^{eq} = 0 \tag{48b}$$

$$\partial_t^{(1)} \Pi_{\alpha\beta}^{eq} + \partial_\gamma^{(1)} \Pi_{\alpha\beta\gamma}^{eq} = -\frac{1}{\tau} \Pi_{\alpha\beta}^{(1)} \tag{48c}$$

Similarly, we multiply the Eq[45b] by 1 and $e_{i\beta}$ respectively. We apply Eq[47] and Eq[40] to obtain the $O(\epsilon^2)$ moment equations:

$$\partial_t^{(2)} \rho = 0 \tag{49a}$$

$$\partial_t^{(2)} (\rho u_\beta) + \partial_\beta^{(1)} \left(1 - \frac{\Delta t}{2\tau}\right) \Pi_{\alpha\beta}^{(1)} = 0 \tag{49b}$$

where $\Pi^{(1)}_{\alpha\beta} = \sum_i e_{i\alpha} e_{i\beta} f^{(1)}_i$ is the perturbation moment.

Assembling the $O(\epsilon)$ moment equations in Eq[48a,b] and $O(\epsilon^2)$ moment equations in Eq[49a,b] by ϵ[48]+ϵ^2[49], we find

$$(\epsilon\partial^{(1)}_t + \epsilon^2\partial^{(2)}_t)\rho + \epsilon\partial^{(1)}_\alpha(\rho u_\alpha) = 0 \tag{50a}$$

$$(\epsilon\partial^{(1)}_\beta + \epsilon^2\partial^{(2)}_t)(\rho u_\beta) + \epsilon\partial^{(1)}_\alpha \Pi^{eq}_{\alpha\beta} = -\epsilon^2\partial^{(1)}_\beta\left(1 - \frac{\Delta t}{2\tau}\right)\Pi^{(1)}_{\alpha\beta} \tag{50b}$$

Now we need to find the value of $\Pi^{(1)}_{\alpha\beta}$ From Eq[48c], and we get

$$\Pi^{(1)}_{\alpha\beta} = -\tau(\partial^{(1)}_t \Pi^{eq}_{\alpha\beta} + \partial^{(1)}_\gamma \Pi^{eq}_{\alpha\beta\gamma}) \tag{51}$$

To get rid of the time derivative, we use equations in [48a,b] and find

$$\partial^{(1)}_t \rho = -\partial^{(1)}_\alpha(\rho u_\alpha) \tag{52a}$$

$$\partial^{(1)}_t(\rho u_\beta) = -\partial^{(1)}_\alpha \Pi^{eq}_{\alpha\beta} \tag{52b}$$

Applying Eq[47d] in the term $\partial^{(1)}_\gamma \Pi^{eq}_{\alpha\beta\gamma}$, we obtain

$$\begin{aligned}\partial^{(1)}_\gamma \Pi^{eq}_{\alpha\beta\gamma} &= \partial^{(1)}_\gamma[\rho c_s^2(u_\alpha\delta_{\beta\gamma} + u_\beta\delta_{\alpha\gamma} + u_\gamma\delta_{\alpha\beta})] \\ &= c_s^2[\partial^{(1)}_\beta(\rho u_\alpha) + \partial^{(1)}_\alpha(\rho u_\beta)] + c_s^2\delta_{\alpha\beta}\partial^{(1)}_\gamma(\rho u_\gamma)\end{aligned} \tag{53}$$

For the term $\partial^{(1)}_t \Pi^{eq}_{\alpha\beta}$, we apply Eq[47c] and product rule of derivatives to find

$$\begin{aligned}\partial^{(1)}_t \Pi^{eq}_{\alpha\beta} &= \partial^{(1)}_t(\rho c_s^2\delta_{\alpha\beta} + \rho u_\alpha u_\beta) \\ &= u_\alpha\partial^{(1)}_t(\rho u_\beta) + u_\beta\partial^{(1)}_t(\rho u_\alpha) - u_\alpha u_\beta\partial^{(1)}_t\rho + c_s^2\delta_{\alpha\beta}\partial^{(1)}_t\rho\end{aligned} \tag{54}$$

We replace the time derivatives in the above equation by using Eq[52], and we simplify the equation by product rule to get

$$\partial^{(1)}_t \Pi^{eq}_{\alpha\beta} = -\partial^{(1)}_\gamma(\rho u_\alpha u_\beta u_\gamma) - c_s^2(u_\alpha\partial^{(1)}_\beta\rho + u_\beta\partial^{(1)}_\alpha\rho) - c_s^2\delta_{\alpha\beta}\partial^{(1)}_\gamma(\rho u_\gamma) \tag{55}$$

Substituting Eq[55] and Eq[53] into Eq[51], we obtain

$$\Pi^{(1)}_{\alpha\beta} = -\tau\rho c_s^2(\partial^{(1)}_\beta u_\alpha + \partial^{(1)}_\alpha u_\beta) + \tau\partial^{(1)}_\gamma(\rho u_\alpha u_\beta u_\gamma) \tag{56}$$

For the two terms on the right hand side of Eq[56], we find the magnitude of the first term to be $O(c_s^2 u)$ and the second term $O(u^3)$. Thus, we could neglect the second term when $c_s^2 \gg u^2$, which is equivalent to Ma$^2 \ll 1$ where Ma is the Mach number $\frac{u}{c_s}$.

Applying Eq[43] and Eq[56] to Eq[50], we obtain

$$\partial_t\rho + \partial_\gamma\rho u_\gamma = 0 \tag{57a}$$

$$\partial_t(\rho u_\alpha) + \partial_\beta(\rho c_s^2\delta_{\alpha\beta} + \rho u_\alpha u_\beta) = \partial_\beta(1 - \frac{\Delta t}{2\tau})\rho c_s^2\tau(\partial_\beta u_\alpha + \partial_\alpha u_\beta) \tag{57b}$$

Note that Eq[57a] recovers the continuity equation as in Eq[35]. We rewrite Eq[57b] by setting pressure $p = \rho c_s^2$ and shear viscosity $\eta = \rho c_s^2(\tau - \frac{\Delta t}{2})$, and we are able to find

$$\partial_t(\rho u_\alpha) + \partial_\beta(\rho u_\alpha u_\beta) = -\partial_\alpha p + \partial_\beta\eta(\partial_\beta u_\alpha + \partial_\alpha u_\beta) \tag{58}$$

Eq[58] recovers the Navier-Stokes equation in Eq[36]. We finally derive that LBE solves the NSE. From Eq[57b], we find that $\frac{\tau}{\Delta t} \geq \frac{1}{2}$ is a necessary stability condition for BGK operator, which is satisfied in our numerical simulation ($\tau = 0.6, \Delta t = 1$).

6 Conclusion

In this paper, we have demonstrated the full process of Computational Fluid Dynamics simulation with the LBE model. The key idea is to treat fluid at a mesoscopic scale, as groups of particles, each confined to a mesh. We discussed and summarized the numerical scheme of LBM, and we provided detailed discussion on treating initial conditions, boundary conditions and corner nodes in implementation. We first simulate a steady plane Poiseuille flow to test the validity of our LBM algorithm. We then use the steady state Poiseuille flow profile to simulate fluid passing a cylinder with varying Reynolds number. We give a discussion on different flow behaviors associated with different Reynolds number. And we provide two creative simulations of flow passing through complex geometries. Lastly, we derived the Navier-Stokes equation from LBE and show the two are equivalent.

We treated our boundary with on-grid bounce-back and Zou-He fixed velocity/pressure condition. While the on-grid bounce-back is easy to implement, it is only first order accuracy due to its one-sided treatment on streaming at the boundary [3]. Further improvement could be made by using mid-grid bounce-back, which leads to a second order accuracy as it considers fictitious nodes. A novel technique to deal with boundary with flow embedded with complex solid object is purposed by Chang et al [10]. The method model the boundary with the boundary nodes do not coincides with the lattices, using Lagrangian markers, the closet nodes adjacent to the boundary and the second closest fluid nodes.The fluid velocity of the boundary node is then obtained by linear interpolation between the Lagrangian markers and second fluid nodes. This technique results in second order of accuracy.

In the future, we could also investigate fluid simulation in the macroscopic aspect, by implementing numerical methods for the Navier-Stokes equation. We can simulate the Navier-Stokes equation using Finite Element Method and Finite Volume Method, and we would like to compare these simulations with the LBM simulations in different aspects, including the difficulty of implementations, order of accuracy and computational cost. Mathematical analysis on stability and accuracy comparison of these methods can be performed as well.

Acknowledgements

We would like to thank Jovana Andrejevic for her valuable help and advice on our project. Thank you, Jovana!

We would also like to thank Vamsi Arza for his advice on getting started with the project. Thank you, Vamsi!

Lastly, we would like to thank Professor Chris Rycroft and the AM205 course TFs, Tara Sowrirajan, Rui Fang, Martin Jin, Nao Ouyang, for an amazing course on numerical methods this semester! We have learned a lot, and we really appreciate all the efforts you devoted in teaching the course and making the course contents fun!

References

[1] Xiaoyi He and Li-Shi Luo. A priori derivation of the lattice boltzmann equation. *Physical Review E*, 55(6):R6333, 1997.

[2] Qisu Zou and Xiaoyi He. On pressure and velocity boundary conditions for the lattice boltzmann bgk model. *Physics of fluids*, 9(6):1591–1598, 1997.

[3] Yuanxun Bill Bao and Justin Meskas. Lattice boltzmann method for fluid simulations. *Department of Mathematics, Courant Institute of Mathematical Sciences, New York University*, 2011.

[4] R Robert Nourgaliev, Truc-Nam Dinh, Theo G Theofanous, and D Joseph. The lattice boltzmann equation method: theoretical interpretation, numerics and implications. *International Journal of Multiphase Flow*, 29(1):117–169, 2003.

[5] Erlend Magnus Viggen Goncalo Silva Orest Shardt Alexandr Kuzmin Krüger, Timm and Halim Kusumaatmaja. *The Lattice Boltzmann Method : Principles and Practice.* Cham: Springer International Publishing : Imprint: Springer, 2017.

[6] Matteo Portinari. *2D and 3D Verification an Validation of the Lattice Boltzmann Method.* PhD thesis, École Polytechnique de Montréal, 2015.

[7] Yoshio Sone. *Kinetic theory and fluid dynamics.* Springer Science & Business Media, 2012.

[8] H Grad. H. grad, commun. pure appl. math. 2, 331 (1949). *Commun. Pure Appl. Math.*, 2:331, 1949.

[9] Ernest Ikenberry and Clifford Truesdell. On the pressures and the flux of energy in a gas according to maxwell's kinetic theory, i. *Journal of Rational Mechanics and Analysis*, 5(1):1–54, 1956.

[10] Cheng Chang, Chih-Hao Liu, and Chao-An Lin. Boundary conditions for lattice boltzmann simulations with complex geometry flows. *Computers & Mathematics with Applications*, 58(5):940–949, 2009.

Appendix

Appendix 1. Links to Videos of LBM Poiseulle Flow Passing Cylinder

1. Density with Re = 1: https://youtu.be/L8EqA68TPRs
2. Density with Re = 20: https://youtu.be/yqRw6tA3dPI
3. Density with Re = 80: https://youtu.be/NT27T6a804s
4. Velocity with Re = 1: https://youtu.be/y0hGixjra38
5. Velocity with Re = 20: https://youtu.be/2pdSO0e6KqU
6. Velocity with Re = 80: https://youtu.be/TwdIDqnrWH8

Appendix 2. Code

Code for Plane Poiseuille Flow and Flow Passing Cylinder Simulation

```
#include <iostream>
#include <cmath>
#include <string>
#include "omp.h"
#include <fstream>

using namespace std;

/*
 * Customizable set up:
 * H, L: Channel Height, Length,;
 * r: radius of cylinder;
 * tau: relaxation time parameter
 * Re: Reynold's number of Flow Past Cylinder setting
 * nt: number of threads to run in parallel computing in OpenMP
 * ts: number of timesteps to run in Flow past Cylinder; Should be a
 * multiple of 2000 (evenly spaced timesteps saving)
 */
const int H = 100;
const int L = 600;
const double r = 4.0;
const double tau = 0.6;
const double Re = 80.0;
const int nt = 25;
const int ts = 60000;

/*pg: Poiseuille Flow Channel Grid*/
/*mg: Mesh Grid setup with cylinder immersed*/
/*dx = 1; dt = 1; c = dx/dt = 1*/
const int n = H+3;
const int m = L+1;
int pg[n][m];
int mg[n][m];
double c = 1.0;
double vis = (2.*tau-1)/6.0;
double delta_p = -36.0 * Re* pow(vis,2.) *L /(pow(H,2.)* 2. * r);
double p0 = 1. - delta_p/2.0;
double p1 = 1. + delta_p/2.0;

/*Generate mg and pg*/
void createGrid(){
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < m; k++){

            if (pow(j-1.0/2.*(H+2.), 2.) + pow((k-1.0/10.*L), 2.) < pow(r, 2.)){
                mg[j][k] = 2;
                pg[j][k] = 0;
            } else {
                    mg[j][k] = 0;
```

```
                    pg[j][k] = 0;
                }
            if (j == 0 || j == H+2) {
                mg[j][k] = 2;
                pg[j][k] = 2;
            }
            if (j == 1 || j == H+1) {
                mg[j][k] = 1;
                pg[j][k] = 1;
            }
        }
    };

    for (int j = int(1.0/2.*(H+2.) - r)-1; j < int(1.0/2.*(H+2.) + r)+2; j++){
        for (int k = int(1.0/10.*L - r)-1; k < int(1.0/10.*L + r)+2; k++){
            if (mg[j][k] == 0) {
                if (mg[j-1][k] == 2 || mg[j+1][k] == 2 || mg[j][k-1] == 2 || mg[j][k+1] == 2 || mg[j+1][k+1] == 2 || mg[j+1][k-1] == 2 ||
                        mg[j-1][k+1] == 2 || mg[j-1][k-1] == 2) {
                    mg[j][k] = 1;
                }
            }
        }
    };

    /*Visualize the MeshGrid generated if correct*/
    /*
    for (int j = 0; j < H+3; j++) {
        for (int k = 0; k < L+1; k++){
            cout << mg[j][k] << ' ';
        }
        cout << endl;
    }
    for (int j = 0; j < H+3; j++) {
        for (int k = 0; k < L+1; k++){
            cout << pg[j][k] << ' ';
        }
        cout << endl;
    }
    */

};

/*Define Lattice Boltzmann related variables/functions:*/
double e[9][2] = {
    {0., 0.},
    {1., 0.},
    {0., 1.},
    {-1., 0.},
    {0., -1.},
    {1., 1.},
    {-1., 1.},
    {-1., -1.},
    {1., -1.}
};

/*to store:
 * dens: density information
 * vel: velocity information
 * f_eq: equilibrium probability information for 9 directions
 * f0 ... f8: probability information for 9 directions;
 *           - [][][0]: pre-stream
 *           - [][][1]: post-stream
 *           - [][][2]: post-collision equilibrium for this timestep
 *
 *   */
double dens[n][m];
double vel[n][m][2];
double f_eq[n][m][9];
double f0[n][m][3];
double f1[n][m][3];
double f2[n][m][3];
```

```
double f3[n][m][3];
double f4[n][m][3];
double f5[n][m][3];
double f6[n][m][3];
double f7[n][m][3];
double f8[n][m][3];

/*Store velocity in x direction and density from last timestep*/
/*Useful for steady state Poisseuille flow criteria, when calculating relative velocity change*/
double oldVelX[n][m];

/*Store steady state Poisseuille flow velocity in x direction as Inlet fixed velocity for flow past Cylinder*/
double iniVelC[n];

/*Weights for calculating equilibrium probability*/
double w[9] = {4./9, 1./9, 1./9, 1./9, 1./9, 1./36, 1./36, 1./36, 1./36};

/*Calculate macroscopic density from mesoscopic probability f0...f8*/
void density(int s, string str){

    if (str == "Poiseuille") {
        #pragma omp parallel num_threads(nt)
        {
            #pragma omp for
            for (int j = 1; j < n-1; j++) {
                for (int k = 0; k < m; k++){
                    dens[j][k] = f0[j][k][s] + f1[j][k][s] + f2[j][k][s] + f3[j][k][s] + f4[j][k][s] +
                                f5[j][k][s] + f6[j][k][s] + f7[j][k][s] + f8[j][k][s];
                }
            }
        }
    }

    if (str == "Cylinder") {
        #pragma omp parallel num_threads(nt)
        {
            #pragma omp for
            for (int j = 1; j < n-1; j++) {
                for (int k = 0; k < m; k++){
                    if (mg[j][k] == 2) {
                        dens[j][k] = 0;
                    } else {
                        dens[j][k] = f0[j][k][s] + f1[j][k][s] + f2[j][k][s] + f3[j][k][s] + f4[j][k][s] +
                                f5[j][k][s] + f6[j][k][s] + f7[j][k][s] + f8[j][k][s];
                    }

                }
            }
        }
    }


};

/*Calculate macroscopic velocity from mesoscopic probability f0...f8 and macroscopic density*/
void velocity(int s, string str){
    if (str == "Poiseuille") {
        #pragma omp parallel num_threads(nt)
        {
            #pragma omp for
            for (int j = 1; j < n-1; j++) {
                for (int k = 0; k < m; k++){
                    for (int i = 0; i < 2; i++){
                        vel[j][k][i] = 1./dens[j][k] * c * (f0[j][k][s] * e[0][i] + f1[j][k][s] * e[1][i]
                                    + f2[j][k][s] * e[2][i] + f3[j][k][s] * e[3][i] + f4[j][k][s] * e[4][i]
```

```
                                + f5[j][k][s] * e[5][i] + f6[j][k][s] * e[6][i] + f7[j][k][s] * e[7][i]
                                + f8[j][k][s] * e[8][i]);
                    }
                }
            }
        }
    }

    if (str == "Cylinder") {
        #pragma omp parallel num_threads(nt)
        {
            #pragma omp for
            for (int j = 1; j < n-1; j++) {
                for (int k = 0; k < m; k++){
                    for (int i = 0; i < 2; i++){

                        if (mg[j][k] == 2) {
                            vel[j][k][i] = 0;
                        } else {
                            vel[j][k][i] = 1./dens[j][k] * c * (f0[j][k][s] * e[0][i] + f1[j][k][s] * e[1][i]
                                + f2[j][k][s] * e[2][i] + f3[j][k][s] * e[3][i] + f4[j][k][s] * e[4][i]
                                + f5[j][k][s] * e[5][i] + f6[j][k][s] * e[6][i] + f7[j][k][s] * e[7][i]
                                + f8[j][k][s] * e[8][i]);
                        }
                    }
                }
            }
        }
    }



};

/*Calculate equilibrium probability of 9 directions, based on velocity and density information*/
void feq(string str){
    if (str == "Poiseuille") {
        #pragma omp parallel num_threads(nt)
        {
            #pragma omp for
            for (int j = 1; j < n-1; j++) {
                for (int k = 0; k < m; k++){
                    for (int i = 0; i < 9; i++){

                        double s = w[i] * (3. * (e[i][0] * vel[j][k][0] + e[i][1] * vel[j][k][1])/c +
                            9/2.0 * pow((e[i][0] * vel[j][k][0] + e[i][1] * vel[j][k][1]),2.)/pow(c, 2.) -
                            3/2.0 * (vel[j][k][0] * vel[j][k][0] + vel[j][k][1] * vel[j][k][1])/pow(c, 2.));

                        f_eq[j][k][i] = w[i] * dens[j][k] + dens[j][k] * s;

                    }
                }
            }
        }
    }

    if (str == "Cylinder") {
        #pragma omp parallel num_threads(nt)
        {
            #pragma omp for
            for (int j = 1; j < n-1; j++) {
                for (int k = 0; k < m; k++){
                    for (int i = 0; i < 9; i++){

```

```
                        if (mg[j][k] == 2) {
                            f_eq[j][k][i] = 0;
                        } else {
                            double s = w[i] * (3. * (e[i][0] * vel[j][k][0] + e[i][1] * vel[j][k][1])/c +
                                9/2.0 * pow((e[i][0] * vel[j][k][0] + e[i][1] * vel[j][k][1]),2)/pow(c, 2.) -
                                3/2.0 * (vel[j][k][0] * vel[j][k][0] + vel[j][k][1] * vel[j][k][1])/pow(c, 2.));

                            f_eq[j][k][i] = w[i] * dens[j][k] + dens[j][k] * s;

                        }
                    }
                }
            }
        }
    }
};

/*Initial condition setup for Poiseuille Flow*/
void initial(){
    #pragma omp parallel num_threads(nt)
    {
        #pragma omp for
        for (int j = 1; j < n-1; j++) {
            for (int k = 0; k < m; k++){

                if (k == 0){
                    dens[j][k] = p0;
                } else if (k == m-1){
                    dens[j][k] = p1;
                } else {
                    dens[j][k] = (p0+p1)/2;
                }

                vel[j][k][0] = 0.;
                vel[j][k][1] = 0.;
            }
        }
    }

    feq("Poiseuille");
    #pragma omp parallel num_threads(nt)
    {
        #pragma omp for
        for (int j = 1; j < n-1; j++) {
            for (int k = 0; k < m; k++){

                f0[j][k][0] = f_eq[j][k][0];
                f1[j][k][0] = f_eq[j][k][1];
                f2[j][k][0] = f_eq[j][k][2];
                f3[j][k][0] = f_eq[j][k][3];
                f4[j][k][0] = f_eq[j][k][4];
                f5[j][k][0] = f_eq[j][k][5];
                f6[j][k][0] = f_eq[j][k][6];
                f7[j][k][0] = f_eq[j][k][7];
                f8[j][k][0] = f_eq[j][k][8];

            }
        }
    }

};

/*Boundary Conditions*/
/*BC: On-grid Bounce Back for upper and lower walls, and the cylinder if appears*/
/*On-grid Bounce Back or Update normally: This handles all grids (j, k) except for the
 *Inlet, Outlet and 4 corners.
 *(j, k) should also not be the dummy walls padded with 2 at mg/pg: [0][:] and [n-1][:]
 */
void onGrid(string s, int j, int k){
```

```
    if (s == "Poiseuille"){
        if (pg[j][k] == 1){
            f0[j][k][1] = f0[j][k][0];
            if (pg[j-1][k] == 2){
                f4[j][k][1] = f2[j][k][0];
            } else {
                f4[j][k][1] = f4[j-1][k][0];
            }

            if (pg[j][k-1] == 2){
                f1[j][k][1] = f3[j][k][0];
            } else {
                f1[j][k][1] = f1[j][k-1][0];
            }

            if (pg[j+1][k] == 2){
                f2[j][k][1] = f4[j][k][0];
            } else {
                f2[j][k][1] = f2[j+1][k][0];
            }


            if (pg[j][k+1] == 2){
                f3[j][k][1] = f1[j][k][0];
            } else {
                f3[j][k][1] = f3[j][k+1][0];
            }

            if (pg[j-1][k+1] == 2){
                f7[j][k][1] = f5[j][k][0];
            } else {
                f7[j][k][1] = f7[j-1][k+1][0];
            }

            if (pg[j-1][k-1] == 2){
                f8[j][k][1] = f6[j][k][0];
            } else {
                f8[j][k][1] = f8[j-1][k-1][0];
            }

            if (pg[j+1][k-1] == 2){
                f5[j][k][1] = f7[j][k][0];
            } else {
                f5[j][k][1] = f5[j+1][k-1][0];
            }

            if (pg[j+1][k+1] == 2){
                f6[j][k][1] = f8[j][k][0];
            } else {
                f6[j][k][1] = f6[j+1][k+1][0];
            }

        }

        if (pg[j][k] == 0){
            f0[j][k][1] = f0[j][k][0];
            f1[j][k][1] = f1[j][k-1][0];
            f2[j][k][1] = f2[j+1][k][0];
            f3[j][k][1] = f3[j][k+1][0];
            f4[j][k][1] = f4[j-1][k][0];
            f5[j][k][1] = f5[j+1][k-1][0];
            f6[j][k][1] = f6[j+1][k+1][0];
            f7[j][k][1] = f7[j-1][k+1][0];
            f8[j][k][1] = f8[j-1][k-1][0];
        }


    }

    if (s == "Cylinder"){
        if (mg[j][k] == 1){
            f0[j][k][1] = f0[j][k][0];
```

```
            if (mg[j-1][k] == 2){
                f4[j][k][1] = f2[j][k][0];
            } else {
                f4[j][k][1] = f4[j-1][k][0];
            }

            if (mg[j][k-1] == 2){
                f1[j][k][1] = f3[j][k][0];
            } else {
                f1[j][k][1] = f1[j][k-1][0];
            }

            if (mg[j+1][k] == 2){
                f2[j][k][1] = f4[j][k][0];
            } else {
                f2[j][k][1] = f2[j+1][k][0];
            }


            if (mg[j][k+1] == 2){
                f3[j][k][1] = f1[j][k][0];
            } else {
                f3[j][k][1] = f3[j][k+1][0];
            }

            if (mg[j-1][k+1] == 2){
                f7[j][k][1] = f5[j][k][0];
            } else {
                f7[j][k][1] = f7[j-1][k+1][0];
            }

            if (mg[j-1][k-1] == 2){
                f8[j][k][1] = f6[j][k][0];
            } else {
                f8[j][k][1] = f8[j-1][k-1][0];
            }

            if (mg[j+1][k-1] == 2){
                f5[j][k][1] = f7[j][k][0];
            } else {
                f5[j][k][1] = f5[j+1][k-1][0];
            }

            if (mg[j+1][k+1] == 2){
                f6[j][k][1] = f8[j][k][0];
            } else {
                f6[j][k][1] = f6[j+1][k+1][0];
            }

        }
        if (mg[j][k] == 0){
            f0[j][k][1] = f0[j][k][0];
            f1[j][k][1] = f1[j][k-1][0];
            f2[j][k][1] = f2[j+1][k][0];
            f3[j][k][1] = f3[j][k+1][0];
            f4[j][k][1] = f4[j-1][k][0];
            f5[j][k][1] = f5[j+1][k-1][0];
            f6[j][k][1] = f6[j+1][k+1][0];
            f7[j][k][1] = f7[j-1][k+1][0];
            f8[j][k][1] = f8[j-1][k-1][0];
        }

    }

};

/*Zou-He BCs for Inlet & Outlet*/
void ZouHe(string str1, string str2, int j, int k){

    if (str1 == "Poiseuille"){
        /*p0, vy = 0*/
        if (str2 == "inlet"){
```

```
            f0[j][k][1] = f0[j][k][0];
            f2[j][k][1] = f2[j+1][k][0];
            f6[j][k][1] = f6[j+1][k+1][0];
            f3[j][k][1] = f3[j][k+1][0];
            f7[j][k][1] = f7[j-1][k+1][0];
            f4[j][k][1] = f4[j-1][k][0];

            double vx = c * (1.- 1./p0 * (f0[j][k][1] + f2[j][k][1] +
                    f4[j][k][1] + 2. * (f3[j][k][1] + f6[j][k][1] +
                    f7[j][k][1])));

            double s1 = 1./9 * (3. * (e[1][0] * vx) /c + 9./2 * pow((e[1][0] * vx), 2.) / pow(c, 2.)
                            - 3./2 * (vx * vx)/pow(c, 2.));
            double s3 = 1./9 * (3. * (e[3][0] * vx) /c + 9./2 * pow((e[3][0] * vx), 2.) / pow(c, 2.)
                            - 3./2 * (vx * vx)/pow(c, 2.));
            double f_eq1 =  1./9 * p0 + p0 * s1;
            double f_eq3 =  1./9 * p0 + p0 * s3;

            f1[j][k][1] = f3[j][k][1] + f_eq1 - f_eq3;
            f5[j][k][1] = 1./2 * (p0 * vx/c - f1[j][k][1] - f2[j][k][1] +
                               f3[j][k][1] + f4[j][k][1] + 2*f7[j][k][1]);
            f8[j][k][1] = 1./2 * (p0 * vx/c - f1[j][k][1] + f2[j][k][1] +
                               f3[j][k][1] - f4[j][k][1] + 2*f6[j][k][1]);
        }

        /*p1, vy = 0*/
        if (str2 == "outlet"){
            f0[j][k][1] = f0[j][k][0];
            f2[j][k][1] = f2[j+1][k][0];
            f5[j][k][1] = f5[j+1][k-1][0];
            f1[j][k][1] = f1[j][k-1][0];
            f8[j][k][1] = f8[j-1][k-1][0];
            f4[j][k][1] = f4[j-1][k][0];

            double vx = c * (-1. +  1./p1 * (f0[j][k][1] + f2[j][k][1] +
                  f4[j][k][1] + 2. * (f1[j][k][1] + f5[j][k][1] +
                    f8[j][k][1])));
            double s1 = 1./9 * (3. * (e[1][0] * vx) /c + 9./2 * pow((e[1][0] * vx), 2.) / pow(c, 2.)
                            - 3./2 * (vx * vx)/pow(c, 2.));
            double s3 = 1./9 * (3. * (e[3][0] * vx) /c + 9./2 * pow((e[3][0] * vx), 2.) / pow(c, 2.)
                            - 3./2 * (vx * vx)/pow(c, 2.));
            double f_eq1 =  1./9 * p1 + p1 * s1;
            double f_eq3 =  1./9 * p1 + p1 * s3;

            f3[j][k][1] = f1[j][k][1] + f_eq3 - f_eq1;
            f6[j][k][1] = 1./2 * (-p1*vx/c + f1[j][k][1] - f2[j][k][1] -
                               f3[j][k][1] + f4[j][k][1] + 2*f8[j][k][1]);
            f7[j][k][1] = 1./2 * (-p1 * vx/c + f1[j][k][1] + f2[j][k][1] -
                               f3[j][k][1] - f4[j][k][1] + 2*f5[j][k][1]);

        }
    }

    if (str1 == "Cylinder"){

        /*vx, vy = 0*/
        if (str2 == "inlet"){
            f0[j][k][1] = f0[j][k][0];
            f2[j][k][1] = f2[j+1][k][0];
            f6[j][k][1] = f6[j+1][k+1][0];
            f3[j][k][1] = f3[j][k+1][0];
            f7[j][k][1] = f7[j-1][k+1][0];
            f4[j][k][1] = f4[j-1][k][0];


            double vx = iniVelC[j];
            double p0C = (2.*(f3[j][k][1] + f6[j][k][1] + f7[j][k][1]) + f0[j][k][1]
                        + f2[j][k][1] + f4[j][k][1])/(1.-vx/c);
```

```
            double s1 = 1./9 * (3. * (e[1][0] * vx) /c + 9./2 * pow((e[1][0] * vx), 2.) / pow(c, 2.)
                        - 3./2 * (vx * vx)/pow(c, 2.));
            double s3 = 1./9 * (3. * (e[3][0] * vx) /c + 9./2 * pow((e[3][0] * vx), 2.) / pow(c, 2.)
                        - 3./2 * (vx * vx)/pow(c, 2.));
            double f_eq1 =  1./9 * p0C + p0C * s1;
            double f_eq3 =  1./9 * p0C + p0C * s3;

            f1[j][k][1] = f3[j][k][1] + f_eq1 - f_eq3;
            f5[j][k][1] = 1./2 * (p0C * vx/c - f1[j][k][1] - f2[j][k][1] +
                           f3[j][k][1] + f4[j][k][1] + 2*f7[j][k][1]);
            f8[j][k][1] = 1./2 * (p0C * vx/c - f1[j][k][1] + f2[j][k][1] +
                           f3[j][k][1] - f4[j][k][1] + 2*f6[j][k][1]);
        }

        /*p1, vy = 0*/
        if (str2 == "outlet"){
            f0[j][k][1] = f0[j][k][0];
            f2[j][k][1] = f2[j+1][k][0];
            f5[j][k][1] = f5[j+1][k-1][0];
            f1[j][k][1] = f1[j][k-1][0];
            f8[j][k][1] = f8[j-1][k-1][0];
            f4[j][k][1] = f4[j-1][k][0];

            double vx = c * (-1. +  1./p1 * (f0[j][k][1] + f2[j][k][1] +
                 f4[j][k][1] + 2. * (f1[j][k][1] + f5[j][k][1] +
                   f8[j][k][1])));
            double s1 = 1./9 * (3. * (e[1][0] * vx) /c + 9./2 * pow((e[1][0] * vx), 2.) / pow(c, 2.)
                        - 3./2 * (vx * vx)/pow(c, 2.));
            double s3 = 1./9 * (3. * (e[3][0] * vx) /c + 9./2 * pow((e[3][0] * vx), 2.) / pow(c, 2.)
                        - 3./2 * (vx * vx)/pow(c, 2.));
            double f_eq1 =  1./9 * p1 + p1 * s1;
            double f_eq3 =  1./9 * p1 + p1 * s3;

            f3[j][k][1] = f1[j][k][1] + f_eq3 - f_eq1;
            f6[j][k][1] = 1./2 * (-p1*vx/c + f1[j][k][1] - f2[j][k][1] -
                           f3[j][k][1] + f4[j][k][1] + 2*f8[j][k][1]);
            f7[j][k][1] = 1./2 * (-p1 * vx/c + f1[j][k][1] + f2[j][k][1] -
                           f3[j][k][1] - f4[j][k][1] + 2*f5[j][k][1]);
        }
    }
};

/*BCs for Corners*/
void corner(string s, int j, int k) {

    /*using Outlet fix pressure p1 information*/
    if (j == 1 && k == m-1) {
        f0[j][k][1] = f0[j][k][0];
        f2[j][k][1] = f2[j+1][k][0];
        f5[j][k][1] = f5[j+1][k-1][0];
        f1[j][k][1] = f1[j][k-1][0];

        f3[j][k][1] = f1[j][k][1];
        f7[j][k][1] = f5[j][k][1];
        f4[j][k][1] = f2[j][k][1];

        f6[j][k][1] = 1./2 * (p1 - f0[j][k][1] - f1[j][k][1] -
                      f2[j][k][1] - f3[j][k][1] - f4[j][k][1] -
                      f5[j][k][1] - f7[j][k][1]);
        f8[j][k][1] = f6[j][k][1];

    }

    if (j == n-2 && k == m-1) {
        f0[j][k][1] = f0[j][k][0];
        f1[j][k][1] = f1[j][k-1][0];
        f8[j][k][1] = f8[j-1][k-1][0];
        f4[j][k][1] = f4[j-1][k][0];
```

```
        f3[j][k][1] = f1[j][k][1];
        f6[j][k][1] = f8[j][k][1];
        f2[j][k][1] = f4[j][k][1];

        f5[j][k][1] = 1./2 * (p1 - f0[j][k][1] - f1[j][k][1] -
                      f2[j][k][1] - f3[j][k][1] - f4[j][k][1] -
                      f6[j][k][1] - f8[j][k][1]);
        f7[j][k][1] = f5[j][k][1];
    }

    if (s == "Poiseuille") {

        /*using Inlet fixed pressure p0 information*/
        if (j == 1 && k == 0) {
            f0[j][k][1] = f0[j][k][0];
            f2[j][k][1] = f2[j+1][k][0];
            f6[j][k][1] = f6[j+1][k+1][0];
            f3[j][k][1] = f3[j][k+1][0];

            f1[j][k][1] = f3[j][k][1];
            f8[j][k][1] = f6[j][k][1];
            f4[j][k][1] = f2[j][k][1];

            f5[j][k][1] = 1./2 * (p0 - f0[j][k][1] - f1[j][k][1] -
                          f2[j][k][1] - f3[j][k][1] - f4[j][k][1] -
                          f6[j][k][1] - f8[j][k][1]);
            f7[j][k][1] = f5[j][k][1];
        }

        if (j == n-2 && k == 0) {
            f0[j][k][1] = f0[j][k][0];
            f3[j][k][1] = f3[j][k+1][0];
            f7[j][k][1] = f7[j-1][k+1][0];
            f4[j][k][1] = f4[j-1][k][0];

            f1[j][k][1] = f3[j][k][1];
            f5[j][k][1] = f7[j][k][1];
            f2[j][k][1] = f4[j][k][1];

            f6[j][k][1] = 1./2 * (p0 - f0[j][k][1] - f1[j][k][1] -
                          f2[j][k][1] - f3[j][k][1] - f4[j][k][1] -
                          f5[j][k][1] - f7[j][k][1]);
            f8[j][k][1] = f6[j][k][1];
        }

    }

    if (s == "Cylinder") {

        /*Inlet fixed velocity, but pressure unknown*/
        if (j == 1 && k == 0) {
            f0[j][k][1] = f0[j][k][0];
            f2[j][k][1] = f2[j+1][k][0];
            f6[j][k][1] = f6[j+1][k+1][0];
            f3[j][k][1] = f3[j][k+1][0];

            f1[j][k][1] = f3[j][k][1];
            f8[j][k][1] = f6[j][k][1];
            f4[j][k][1] = f2[j][k][1];

            /*Extrapolate pressure information from nearest point at boundary*/
            double pjk = f0[j+1][k][1] + f1[j+1][k][1] + f2[j+1][k][1] + f3[j+1][k][1]
    + f4[j+1][k][1] + f5[j+1][k][1] + f6[j+1][k][1] + f7[j+1][k][1] + f8[j+1][k][1];
            f5[j][k][1] = 1./2 * (pjk - f0[j][k][1] - f1[j][k][1] -
                          f2[j][k][1] - f3[j][k][1] - f4[j][k][1] -
                          f6[j][k][1] - f8[j][k][1]);
            f7[j][k][1] = f5[j][k][1];
        }

        if (j == n-2 && k == 0) {
            f0[j][k][1] = f0[j][k][0];
```

```
            f3[j][k][1] = f3[j][k+1][0];
            f7[j][k][1] = f7[j-1][k+1][0];
            f4[j][k][1] = f4[j-1][k][0];

            f1[j][k][1] = f3[j][k][1];
            f5[j][k][1] = f7[j][k][1];
            f2[j][k][1] = f4[j][k][1];

            /*Extrapolate pressure information from nearest point at boundary*/
            double pjk = f0[j-1][k][1] + f1[j-1][k][1] + f2[j-1][k][1] + f3[j-1][k][1] + f4[j-1][k][1] + f5[j-1][k][1] + f6[j-1][k][1] + f7[j-1][k][1] + f8[j-1][k][1];
            f6[j][k][1] = 1./2 * (pjk - f0[j][k][1] - f1[j][k][1] -
                          f2[j][k][1] - f3[j][k][1] - f4[j][k][1] -
                          f5[j][k][1] - f7[j][k][1]);
            f8[j][k][1] = f6[j][k][1];
        }

    }

}

int main ()
{
    /*Create pg for Poiseuille Flow and mg for cylinder immersed flow*/
    createGrid();

/*==================================================================*/
    /*Poiseuille Flow Simulation*/
    /*First, simulate Poiseuille Flow till steady state*/
    initial();

    int ct = 1;
    double relVel = 1.;

    ofstream out_dens0 ("/home/jiayinlu/Desktop/Kay/LB/Re80/Poiseuille/density/textfile/density"+ to_string(ct) +".txt");
    ofstream out_velx0 ("/home/jiayinlu/Desktop/Kay/LB/Re80/Poiseuille/velocity/velx/textfile/velx"+ to_string(ct) +".txt");

    for (int j = 1; j < n-1; j++) {
        for (int k = 0; k < m; k++){

            out_dens0 << dens[j][k] << " " ;
            out_velx0 << vel[j][k][0] << " " ;

        }
        out_dens0 << endl;
        out_velx0 << endl;
    }

    /*Steady State Criteria; number of timesteps*/
    while (relVel > 5.0 * pow(10, -9.)){

        /*Streaming*/
        /*Exclude the 2's over upper & lower walls*/
        /*Exclude the four corners*/
        #pragma omp parallel num_threads(nt)
        {
            #pragma omp for
            for (int j = 1; j < n-1; j++) {
                for (int k = 0; k < m; k++){

                    if (k != 0 && k != (m-1)){
                        /*Update all points expect inlet, outlet and corners*/
                        onGrid("Poiseuille", j, k);

                    }

                    if (k == 0 && j != 1 && j != n-2){
                        /*inlet BC*/
                        ZouHe("Poiseuille", "inlet", j, k);

```

```
                    }

                    if (k == (m-1) && j != 1 && j != n-2) {
                        /*outlet BC*/
                        ZouHe("Poiseuille", "outlet", j, k);

                    }
                }
            }
        }

        /*four corners BC streaming*/
        corner("Poiseuille", 1, m-1);
        corner("Poiseuille", n-2, m-1);
        corner("Poiseuille", 1, 0);
        corner("Poiseuille", n-2, 0);

        /*Store x direction velocity information before changing it*/
        #pragma omp parallel num_threads(nt)
        {
            #pragma omp for
            for (int j = 1; j < n-1; j++) {
                for (int k = 0; k < m; k++){
                    oldVelX[j][k] = vel[j][k][0];
                }
            }
        }

        /*update post streaming density and velocity and f_eq*/
        density(1, "Poiseuille");
        velocity(1, "Poiseuille");
        feq("Poiseuille");

        /*Collision*/
        #pragma omp parallel num_threads(nt)
        {
            #pragma omp for
            for (int j = 1; j < n-1; j++) {
                for (int k = 0; k < m; k++){

                    /*Update post-collision probability as the pre-streaming*/
                    /*probability for next timestep*/
                    f0[j][k][0] = f0[j][k][1] - 1./tau * (f0[j][k][1] - f_eq[j][k][0]);
                    f1[j][k][0] = f1[j][k][1] - 1./tau * (f1[j][k][1] - f_eq[j][k][1]);
                    f2[j][k][0] = f2[j][k][1] - 1./tau * (f2[j][k][1] - f_eq[j][k][2]);
                    f3[j][k][0] = f3[j][k][1] - 1./tau * (f3[j][k][1] - f_eq[j][k][3]);
                    f4[j][k][0] = f4[j][k][1] - 1./tau * (f4[j][k][1] - f_eq[j][k][4]);
                    f5[j][k][0] = f5[j][k][1] - 1./tau * (f5[j][k][1] - f_eq[j][k][5]);
                    f6[j][k][0] = f6[j][k][1] - 1./tau * (f6[j][k][1] - f_eq[j][k][6]);
                    f7[j][k][0] = f7[j][k][1] - 1./tau * (f7[j][k][1] - f_eq[j][k][7]);
                    f8[j][k][0] = f8[j][k][1] - 1./tau * (f8[j][k][1] - f_eq[j][k][8]);

                    /*Storing post-collision equilibrium for this timestep*/
                    /*in fi[][][2]*/
                    f0[j][k][2] = f_eq[j][k][0];
                    f1[j][k][2] = f_eq[j][k][1];
                    f2[j][k][2] = f_eq[j][k][2];
                    f3[j][k][2] = f_eq[j][k][3];
                    f4[j][k][2] = f_eq[j][k][4];
                    f5[j][k][2] = f_eq[j][k][5];
                    f6[j][k][2] = f_eq[j][k][6];
                    f7[j][k][2] = f_eq[j][k][7];
                    f8[j][k][2] = f_eq[j][k][8];

                }
            }
        }

        density(2, "Poiseuille");
        velocity(2, "Poiseuille");

        /*Calculate relative velocity change*/
```

```
        /*Save density and velocity information at this timestep*/
        double relVelN = 0.0;
        double relVelD = 0.0;

        #pragma omp parallel num_threads(nt)
        {
            #pragma omp for
            for (int j = 1; j < n-1; j++) {
                for (int k = 0; k < m; k++){

                    relVelN = relVelN + abs(vel[j][k][0] - oldVelX[j][k]);
                    relVelD = relVelD + abs(vel[j][k][0]);

                }
            }
        }

        relVel = relVelN/relVelD;
        ct ++;
    };

    ofstream out_dens1 ("/home/jiayinlu/Desktop/Kay/LB/Re80/Poiseuille/density/textfile/density"+ to_string(ct) +".txt");
    ofstream out_velx1 ("/home/jiayinlu/Desktop/Kay/LB/Re80/Poiseuille/velocity/velx/textfile/velx"+ to_string(ct) +".txt");

    for (int j = 1; j < n-1; j++) {
        for (int k = 0; k < m; k++){

            out_dens1 << dens[j][k] << " " ;
            out_velx1 << vel[j][k][0] << " " ;

        }
        out_dens1 << endl;
        out_velx1 << endl;
    }

    /*Store steady Poisseuille Flow x velocity information for Fixed Velocity Inlet in Cylinder case*/
    #pragma omp parallel num_threads(nt)
    {
        #pragma omp for
        for (int j = 1; j < n-1; j++) {
            iniVelC[j] = vel[j][m-1][0];
        }
    };
/*==============================================================*/

    /*Flow past cylinder*/
    /*Set up initial conditions for Flow Past Cylinder; Use the above steady state Poisseulle Flow profile*/
    #pragma omp parallel num_threads(nt)
    {
        #pragma omp for
        for (int j = 1; j < n-1; j++) {
            for (int k = 0; k < m; k++){

                f0[j][k][0] = f_eq[j][k][0];
                f1[j][k][0] = f_eq[j][k][1];
                f2[j][k][0] = f_eq[j][k][2];
                f3[j][k][0] = f_eq[j][k][3];
                f4[j][k][0] = f_eq[j][k][4];
                f5[j][k][0] = f_eq[j][k][5];
                f6[j][k][0] = f_eq[j][k][6];
                f7[j][k][0] = f_eq[j][k][7];
                f8[j][k][0] = f_eq[j][k][8];
            }
        }
    };

    int ct2 = 1;

```

```
    /*Determine number of timesteps*/
    while (ct2 < ts+2) {

        /*Streaming*/
        /*Exclude the 2's over upper & lower walls*/
        /*Exclude the four corners*/
        #pragma omp parallel num_threads(nt)
        {
            #pragma omp for
            for (int j = 1; j < n-1; j++) {
                for (int k = 0; k < m; k++){

                    if (k != 0 && k != (m-1)){
                        /*Update all points expect inlet, outlet and corners*/
                        onGrid("Cylinder", j, k);

                    }

                    if (k == 0 && j != 1 && j != n-2){
                        /*inlet BC*/
                        ZouHe("Cylinder", "inlet", j, k);

                    }

                    if (k == (m-1) && j != 1 && j != n-2) {
                        /*outlet BC*/
                        ZouHe("Cylinder", "outlet", j, k);

                    }
                }
            }
        }

        /*four corners BC streaming*/
        corner("Cylinder", 1, m-1);
        corner("Cylinder", n-2, m-1);
        corner("Cylinder", 1, 0);
        corner("Cylinder", n-2, 0);

        /*update post streaming density and velocity and f_eq*/
        density(1, "Cylinder");
        velocity(1, "Cylinder");
        feq("Cylinder");

        /*Collision*/
        #pragma omp parallel num_threads(nt)
        {
            #pragma omp for
            for (int j = 1; j < n-1; j++) {
                for (int k = 0; k < m; k++){

                    /*Update post-collision probability as the pre-streaming probability for next timestep*/
                    f0[j][k][0] = f0[j][k][1] - 1./tau * (f0[j][k][1] - f_eq[j][k][0]);
                    f1[j][k][0] = f1[j][k][1] - 1./tau * (f1[j][k][1] - f_eq[j][k][1]);
                    f2[j][k][0] = f2[j][k][1] - 1./tau * (f2[j][k][1] - f_eq[j][k][2]);
                    f3[j][k][0] = f3[j][k][1] - 1./tau * (f3[j][k][1] - f_eq[j][k][3]);
                    f4[j][k][0] = f4[j][k][1] - 1./tau * (f4[j][k][1] - f_eq[j][k][4]);
                    f5[j][k][0] = f5[j][k][1] - 1./tau * (f5[j][k][1] - f_eq[j][k][5]);
                    f6[j][k][0] = f6[j][k][1] - 1./tau * (f6[j][k][1] - f_eq[j][k][6]);
                    f7[j][k][0] = f7[j][k][1] - 1./tau * (f7[j][k][1] - f_eq[j][k][7]);
                    f8[j][k][0] = f8[j][k][1] - 1./tau * (f8[j][k][1] - f_eq[j][k][8]);

                    /*Storing post-collision equilibrium for this timestep in fi[][][2]*/
                    f0[j][k][2] = f_eq[j][k][0];
                    f1[j][k][2] = f_eq[j][k][1];
                    f2[j][k][2] = f_eq[j][k][2];
                    f3[j][k][2] = f_eq[j][k][3];
                    f4[j][k][2] = f_eq[j][k][4];
                    f5[j][k][2] = f_eq[j][k][5];
                    f6[j][k][2] = f_eq[j][k][6];
```

```
                    f7[j][k][2] = f_eq[j][k][7];
                    f8[j][k][2] = f_eq[j][k][8];

                }
            }
        }

        density(2, "Cylinder");
        velocity(2, "Cylinder");

        /*Save only 2000 evenly spaced points in timestep iteration*/
        if ((ct2-1)%(ts/2000)==0){
            /*Save density and velocity information at this timestep*/
            ofstream out_dens ("/home/jiayinlu/Desktop/Kay/LB/Re80/Cylinder/density/textfile/density"+ to_string(ct2) +".txt");
            ofstream out_velx ("/home/jiayinlu/Desktop/Kay/LB/Re80/Cylinder/velocity/velx/textfile/velx"+ to_string(ct2) +".txt");
            ofstream out_vely ("/home/jiayinlu/Desktop/Kay/LB/Re80/Cylinder/velocity/vely/textfile/vely"+ to_string(ct2) +".txt");
            for (int j = 1; j < n-1; j++) {
                for (int k = 0; k < m; k++){

                    out_dens << dens[j][k] << " " ;
                    out_velx << vel[j][k][0] << " " ;
                    out_vely << vel[j][k][1] << " " ;

                }
                out_dens << endl;
                out_velx << endl;
                out_vely << endl;
            }
        }

        ct2 ++;
    };

    return 0;
}
```

Code for Plotting and Making Videos of Flow Simulation

```
# Plot density
Res = ['Re1','Re20','Re80']
settings = ['Cylinder','Poiseuille']

for Re,setting in product(Res,settings):
    path = os.path.join(Re,setting,'density')
    textfile_path = os.path.join(path,'textfile')
    plot_path = os.path.join(path,'plot')
    files = os.listdir(textfile_path)
    for file in files:
        fn = 'density'+'{0:05}'.format(int(file.split('.')[0][7:]))+'.png'
        Cdens = np.genfromtxt(os.path.join(textfile_path,file))
        plt.figure(figsize = (18, 10))
        masked_Cdens = np.ma.masked_where(Cdens == 0, Cdens)
        cmap = matplotlib.cm.jet
        cmap.set_bad('k',1.)
        plt.imshow(masked_Cdens, interpolation='nearest', cmap=cmap)
                  #vmin=0.994,vmax=1.0065)
        plt.colorbar(orientation='horizontal')
        plt.savefig(os.path.join(plot_path,fn))
        plt.close()

# Plot Velocity
Res = ['Re20','Re80']
for Re in Res:
    path = os.path.join(Re,'Cylinder','velocity')
    path_x = os.path.join(path,'velx','textfile')
    path_y = os.path.join(path,'vely','textfile')
    plot_path = os.path.join(path,'plot')
    for fn_txt_x in os.listdir(path_x):
```

```
        fn_txt_y = 'vely'+fn_txt_x[4:]
        fn_img = 'vel'+'{0:05}'.format(int(fn_txt_x.split('.')[0][4:]))+'.png'
        Cvelx = np.genfromtxt(os.path.join(path_x,fn_txt_x))
        Cvely = np.genfromtxt(os.path.join(path_y,fn_txt_y))

        Cvelm = np.zeros((len(Cvely), int(len(Cvely[1,:]))))
        for i in range(0, len(Cvelm)):
            for j in range(0, len(Cvelm[1,:])):
                Cvelm[i, j] = (Cvelx[i, j]**2 + Cvely[i, j]**2)**0.5

        plt.figure(figsize = (28, 20))
        masked_velm = np.ma.masked_where(Cvelm == 0, Cvelm)
        cmap = matplotlib.cm.jet
        cmap.set_bad('k',1.)
        plt.imshow(masked_velm, interpolation='nearest', cmap=cmap)
        plt.colorbar(orientation='horizontal')
        plt.savefig(os.path.join(plot_path,fn_img))
        plt.close()

# Make Video
for Re in Res:
    image_folder = os.path.join(Re,'Cylinder','density','plot')
    video_name = Re+'_density.avi'

    images = [img for img in sorted(os.listdir(image_folder))
              if img.endswith(".png")]
    frame = cv2.imread(os.path.join(image_folder, images[0]))
    height, width, layers = frame.shape

    video = cv2.VideoWriter(video_name, -1, 6, (width,height))

    for image in images:
        image_file = os.path.join(image_folder, image)
        video.write(cv2.imread(image_file))

    cv2.destroyAllWindows()
    video.release()
```

Code for Creating Maps based on Arbitrary Images

```
import numpy as np
from skimage import io
import matplotlib.pyplot as plt

# Read target shape
Pdens = np.genfromtxt('Re1/Cylinder/density/textfile/density1.txt')

# Read image into a matrix of 2 and 0
def read_img(fn):
    target_y,target_x = 103, 601
    img_data = io.imread(fn)
    img = np.array([[np.linalg.norm(img_pixel) for img_pixel in img_row]
                    for img_row in img_data])
    img = np.where(img==255,int(2),0)
    A_y, A_x = img.shape
    pad_top = (target_y - A_y)//2
    pad_bottom = target_y - A_y - pad_top
    pad_left = (target_x - A_x)//4
    pad_right = target_x - A_x - pad_left
    img = np.pad(img,((pad_top,pad_bottom),(pad_left,pad_right)),
                 'constant',constant_values=0)
    img[0] = [2]*target_x
    img[1] = [1]*target_x
    img[-1] = [2]*target_x
    img[-2] = [1]*target_x
    for i in range(1,target_y-1):
        for j in range(1,target_x-1):
            if img[i,j] == 0:
                if img[i+1,j]==2 or img[i-1,j]==2\
                   or img[i,j+1]==2 or img[i,j-1]==2\
                   or img[i+1,j+1]==2 or img[i+1,j-1]==2\
```

```
                        or img[i-1,j+1]==2 or img[i-1,j-1]==2:
                            img[i,j] = 1
    return img

img_AM205 = read_img("map_images/AM205.png")

# Save data
np.savetxt('map_images/AM205_left.txt',img_AM205,fmt='%0.0f')
```