On the Lattice Boltzmann Method: Implementation and Applications

Except where reference is made to the work of others, the work described in this dissertation is my own or was done in collaboration with my advisory committee. This dissertation does not include proprietary or classified information.

Kang Jin

Certificate of Approval:

Paul G. Schmidt Professor Mathematics and Statistics	Amnon J. Meir, Chair Professor Mathematics and Statistics
Wenxian Shen Professor Mathematics and Statistics	Jay Khodadadi Professor Mechanical Engineering

George T. Flowers
Interim Dean
Graduate School

On the Lattice Boltzmann Method: Implementation and Applications

Kang Jin

A Dissertation

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Doctor of Philosophy

Auburn, Alabama
December 19, 2008

On the Lattice Boltzmann Method: Implementation and Applications

Kang Jin

Permission is granted to Auburn University to make copies of this dissertation at its discretion, upon the request of individuals or institutions and at their expense. The author reserves all publication rights.

Signature of Author

Date of Graduation

Vita

Kang Jin was born in Shanghai, China in 1979. He did all his undergraduate study in Shanghai. He graduated from East China Normal University in 2001 with a Bachelors Degree in Mathematics. He then went to the United States in 2002. He accepted an offer from Auburn University, and began his graduate study as well as being a Teaching Assistant. He got the Master of Science in Mathematics in 2005, and Then continue his study as a Ph.D student in Auburn University.

Dissertation Abstract

On the Lattice Boltzmann Method: Implementation and Applications

Kang Jin

Doctor of Philosophy, December 19, 2008
(B.S., East China Normal University, 2001)

125 Typed Pages

Directed by Amnon J. Meir

We studied the development and different types of the Lattice Boltzmann method. We gave several implementations. Then we presented two moving boundary treatments for the Lattice Boltzmann method, the second one is new. We also gave an incompresibility enhancement for the Lattice Boltzmann method in order to better simulate some problems using the moving boundary. Finally we gave a MHD solution using the Lattice Boltzmann method.

Acknowledgments

I thank Auburn University Mathematics and Statistics department for offering me the Graduate Teaching Assistant position. This is very important to me. Without this offer I could not be where I am today. I also thank the faculty in Math department who give me lots of help. I thank my committee members Dr. Wenxian Shen, Dr. Paul Schimdt and Dr. Jay Khodadadi. Special thank to my advisor, Dr. Amnon J. Meir. His support and advice are the key to my work.

Style manual or journal used Journal of Approximation Theory (together with the style known as "aums"). Bibliograpy follows van Leunen's *A Handbook for Scholars*.

Computer software used The document preparation package TEX (specifically LATEX) together with the departmental style-file `aums.sty`.

Table of Contents

List of Figures . . . x

1 Introduction . . . 1
- 1.1 Background . . . 1
- 1.2 The FHP method . . . 3
  - 1.2.1 Basic model . . . 3
  - 1.2.2 Macroscopic quantities . . . 7
- 1.3 The Lattice BGK method . . . 7
  - 1.3.1 The D2Q7 method . . . 7
  - 1.3.2 Other frequently used Lattice Boltzmann Methods . . . 22
- 1.4 Applications . . . 25

2 Moving Boundary Problems . . . 26
- 2.1 Background . . . 26
- 2.2 Moving boundary for LBM . . . 30
  - 2.2.1 Moving boundary without mass conservation . . . 30
  - 2.2.2 Moving boundary with mass conservation . . . 34
- 2.3 Sharp boundary definition details . . . 39
  - 2.3.1 Line . . . 40
  - 2.3.2 Arc . . . 41
- 2.4 Implementation . . . 41
- 2.5 Conclusion and Future Work . . . 42

3 Incompressibility . . . 45
- 3.1 Background . . . 45
- 3.2 An incompressibility enhanced scheme for LBGK . . . 47
  - 3.2.1 The scheme . . . 47
  - 3.2.2 An example . . . 48
- 3.3 Incompressibility with moving boundary . . . 48
- 3.4 Conclusion . . . 54

4 MHD With Constant B . . . 55
- 4.1 Backgound . . . 55
- 4.2 Implementations . . . 56
  - 4.2.1 Example I . . . 56
  - 4.2.2 Example II . . . 59
- 4.3 Conclusion . . . 61

5 Conclusion 62

Bibliography 64

Appendices 66

A FHP collision look-up table 67

B Partial Matlab code I: d2q9_rotating_polygon.m 69

C Partial matlab code II: d2q9_rotating_polygon_collision.m 82

List of Figures

1.1 The hexagonal grid. 4

1.2 FHP collision rule 6

1.3 The d2q7_lattice. 8

1.4 Left boundary of a hexagonal grid. 15

1.5 Driven cavity at $Re = 10$. 17

1.6 Driven cavity at $Re = 100$. 18

1.7 Driven cavity at $Re = 200$. 18

1.8 Driven cavity at $Re = 400$. 19

1.9 Ghia Ghia Shin's result. 19

1.10 Driven cavity at $Re = 800$. 20

1.11 Dimensionless x-velocity profile at the geometry center. 20

1.12 Dimensionless y-velocity profile at the geometry center 21

1.13 A uniform flow past a cylinder 23

1.14 The D2Q9 lattice. 24

2.1 No-slip Boundary 28

2.2 Slip Boundary 29

2.3 Above: boundary on the half-way; Below: boundary on the node 29

2.4 Cases when q>1/2 or q<1/2. 31

2.5 The change of state illustration. 33

2.6 A half-way bounce-back boundary. 35

2.7 Bounce-back boundary when $q < 1/2$. 37

2.8 Bounce-back boundary when $q > 1/2$. 38

2.9 An example of a boundary node. 39

2.10 The change of state in this bounce-back based scheme. 40

2.11 The definition of a straight line boundary. 41

2.12 The definition of an arc boundary. 42

2.13 The rotating triangle example. 43

3.1 A test using the incompressible Lattice BGK method 49

3.2 The velocity contour of the damper with the standard Lattice BGK method. 50

3.3 The velocity contour of the damper with the incompressible Lattice BGK method. 51

3.4 The density of the fluid in the damper with the standard Lattice BGK method. 52

3.5 The density of the fluid in the damper with the incompressible Lattice BGK method. 53

4.1 The boundary conditions of ϕ. 57

4.2 The result of the cubic box MHD test. 58

4.3 The example of a magnetic pump. 59

4.4 The velocity contour of the magnetic pump example. 60

Chapter 1

Introduction

1.1 Background

The Lattice Gas model and Lattice Boltzmann method are methods for simulating fluid flows. The flow of incompressible fliuds can be described by the Navier-Stokes equation

$$\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla)\vec{u} = -\nabla P + \nu \nabla^2 \vec{u} \tag{1.1}$$

and the continuity equation

$$\nabla \cdot \vec{u} = 0 \tag{1.2}$$

where $\vec{u}$ is the velocity, $P = p/\rho_0$ the kinematic pressure, p the pressure, ρ_0 the constant mass density and $\nu = \eta/\rho_0$ the kinematic shear viscosity, and η the dynamic shear viscosity.

The Lattice Boltzmann method is derived from the Lattice Gas method. These two methods are different from other methods such as finite difference, finite volume, and finite elements which are based on the discretization of partial differential equations (top-down models [1]). These two method are based on a discrete microscopic model which conserves desired quantities (such as mass and momentum), the partial differential equations can then be derived by multi-scale analysis (bottom-up models).

First introduced in 1973, by Hardy, de Pazzis and Pomeau (HPP) [2], the HPP method is a Lattice Gas method. It simulates the microscopic behavior of the fluid by utilizing a square grid. The basic idea was to create a simple Cellular Automaton

obeying nothing but conservation laws at a microscopic level that was able to reproduce the complexity of real fluid flows. Fluid particles of identical mass are only allowed to travel on the lattice at unit speed. All lattice sites, which are the intersections of the lattice, are exclusive. This means that only one particle is allowed to travel at each of the four directions of one site. This gives a maximum of 4 particles at each site. Each site can only take a finite number of states, actually $2^4 = 16$ states. At each time step, a collision occurs at each site, according to a collision rule which conserves the density and the momentum. Then particles travel in straight lines (free streaming) until they meet some other particle or the boundary.

This method is computer friendly, since only a 4-bit variable is needed, and only logical operations are repuired. Also, only the information from the four neighbours are needed at each collision and streaming, this method is easy to parallelize. Simple calculations as the HPP required, however, it leads to a macroscopical anisotropical Navier-Stokes equation. This defect prevents the HPP from being widely used to model fluid problems. In 1986 Frisch, Hasslacher and Pomeau (FHP) [3] introduced a lattice gas method based on a hexagonal grid. This grid change made the FHP method exhibit isotropy. Details of the FHP are discussed in the next section.

Similar to the HPP, logical operations made the FHP method easy to implement on computers. Now the biggest problem of the cellular automata is noise, since it is based on a Fermi-Dirac distribution of the equilibrium population because of the exclusion principle. The Fermi-Dirac distruibution is a distribution that applies to particles called fermions. Fermions have half-integral values of the quantum mechanical property called spin and are “antisocial” in the sense that two fermions cannot exist in the same state. Protons, neutrons, electrons, and many other elementary particles are fermions. The

results of the FHP is very noisy. Ensemble average and space average should both be used. This may result in a grid size thousand times larger than the orighnal problem. For example, if the final solution on a 100×100 grid is needed, averaging on 10×10 cells and ensemble averaging on 10 experiments, then the size of the calculation is 10 times the size that is on a 1000×1000 grid! The lack of Galilean invariance is another big problem of the FHP. The collision rules can be written in a look-up table. For the FHP method, this table should have a size of $2^7 \times 7$. For multi-dimensional simulations, huge look-up table associated with the collision rule made this almost impossible.

The Lattice Boltzmann method overcomes these defects very well. Instead of using boolean variables at each site, the Lattice Boltzmann method uses real numbers. The first method, proposed by McNamara and Zanetti [4], replaced the boolean variables with their ensemble average. The statistics noise is greatly reduced. After that the linear collision operator [5] came into being and then the enhanced collision rule [6]. The breakthrough was the single relaxation time approximation, known as the Lattice BGK method, named after Bhatnagar, Gross, and Krook [7]. This method dramatically reduced computation and gives pretty good results in various applications. In this article, we will discuss in detail the Lattice BGK method. Simulations up to Reynolds number 1000 are presented.

1.2 The FHP method

1.2.1 Basic model

The Lattice Gas Cellular Automata simulates molecular collision in a discretized fashion. Consider a hexagonal grid shown in Figure 1.1. Each site is surrounded by 6

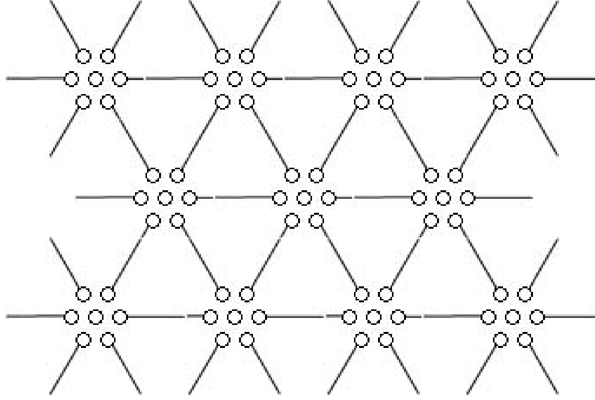

Figure 1.1: The hexagonal grid. One site can contain a maximum of 7 particles.

neighbours, connected by unit vectors

$$\vec{e}_i = (\sin(\frac{\pi}{2}(i-1)), \cos(\frac{\pi}{2}(i-1))), i = 1, ..., 6. \tag{1.3}$$

Exclusion principles allow a maximum of 7 particles at one site, one moving particle in each of the 6 directions, together with a rest particle in the middle. Here we use an occupation boolean variable $n_i(\vec{x}, t)$, $i = 0, 1, ..., 6$ (0 stands for the rest particle) to indicate particle presence or non-presence at the ith direction of site $\vec{x}$ at time t. Thus a 7-bit variable is enough to carry the information at one site. All particles have the same mass and the same speed.

One time step consists of a collision and a streaming. Collision only occurs at the sites, while streaming takes place on the connection between each two sites. the collision rules should conserve mass and momentum. Figure 1.2 shows the basic set of collision rules [8]. The left column are the in-states and the right are the out-states. In-state means that particles are moving towards the center of the site. After the collision

follows the out-state, particles then move away from the center of the site and begin streaming. So a full time step is:

$$\text{in-state} \Longrightarrow \text{collision} \Longrightarrow \text{out-state} \Longrightarrow \text{streaming} \Longrightarrow \text{in-state}.$$

By rotating, flipping, and combining these 7 rules, one can get a full set of 128 collision rules. Notice that some in-states will lead to two equivalent out-states. It is not necessary to pick an out-state randomly at every site. Notice that picking a random number is very time consuming. Instead, one can pick a single random boolean variable for all the sites at one time step.

The 6-direction discretization makes this method lack degrees of freedom on speed directions, yet it can display all the complexities of fliud phenomema [9]. This is the simplest isotropic model. By limiting the types of collision, the FHP can be divided into three types. The FHP-I only allows collisions of type (a) and (b). No rest particle is present. Types of collision other than (a) and (b) are replaced by simple streamig as if the particles don't collide at all. FHP-II adds the rest particle, together with collision type (c), (d), and (e). FHP-III includes all types of collisions.

A no-slip boundary condition can be imposed by a bounce back scheme, which means particles that hit the boundary at any angle should move back (bounce back) in the opposite direction. Reflection will lead to a slip boundary. The Dirichelet boundary condition can be imposed as a random variable on the boundary with a probability distuibution indicating the value at the boundary, then applying the collision followed by the bounce back scheme.

Figure 1.2: FHP collision rule

1.2.2 Macroscopic quantities

Noise is the biggest problem of the FHP method. Hence both space average and ensemble average should be used. The space average is achieved by averaging on small cells, for example, 16×16 cells. The density is given by $\rho = \sum_i n_i$. And ρ_0 is the mean density, which is the average on the whole grid.

Here are some method-dependent quantities derived by Frisch *et al.* [3] for the three types of FHP methods.

	FHP-I	FHP-II	FHP-III
d	$\rho_0/6$	$\rho_0/7$	$\rho_0/7$
c_s	$\frac{1}{\sqrt{2}}$	$\sqrt{\frac{3}{7}}$	$\sqrt{\frac{3}{7}}$
ν	$\frac{1}{12}\frac{1}{d(1-d)^3} - \frac{1}{8}$	$\frac{1}{28}\frac{1}{d(1-d)^3}\frac{1}{1-4d/7} - \frac{1}{8}$	$\frac{1}{28}\frac{1}{d(1-d)}\frac{1}{1-8d(1-d)/7} - \frac{1}{8}$

where d is the mean density per link, c_s is the speed of sound, ν is the kinematic viscosity.

1.3 The Lattice BGK method

1.3.1 The D2Q7 method

The scheme

Consider again the hexagonal lattice showed in figure 1.3. This time, the occupation number is replaced by its ensemble average value, or, the particle distribution function $f_i(\vec{x}, t)$. The meaning of this function is the probability of finding a particle moving in the ith direction of the site $\vec{x}$ at time t. The collision rules in FHP are replaced with a collision operator Ω_i, and the particle distribution function should satisfy the Lattice

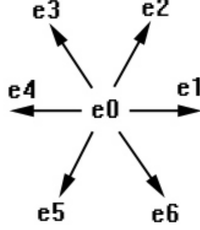

Figure 1.3: The d2q7_lattice.

Boltzmann equation

$$f_i(\vec{x}+e_i,t+1)-f_i(\vec{x},t)=\Omega_i. \tag{1.4}$$

This collision operator has lots of forms. Here we talk about the simplest one, the BGK single relaxation time model. Introduce the single relaxation parameter τ, and the equilibrium distribution function $f_i^{eq}(\vec{x},t)$. By assuming that the particle distribution function approaches the equilibrium state at a constant rate, we should get

$$\Omega_i=-\frac{1}{\tau}(f_i-f_i^{eq}). \tag{1.5}$$

This gives us the equation

$$f_i(\vec{x}+e_i,t+1)=(1-w)f_i(\vec{x},t)+wf_i^{eq}(\vec{x},t) \tag{1.6}$$

where the weight $w=\frac{1}{\tau}$. The equilibrium distribution function has the form

$$f_i^{eq}(\vec{x},t)=\rho(\vec{x},t)\left(\frac{1-z}{6}+\frac{D}{6c^2}(\vec{e}_i\cdot\vec{u})+\frac{D(D+2)}{12c^4}(\vec{e}_i\cdot\vec{u})^2-\frac{D\vec{u}^2}{12c^2}\right),i=1,...,6 \tag{1.7}$$

$$f_0^{eq}(\vec{x},t)=\rho(\vec{x},t)(z-\frac{u^2}{c^2}) \tag{1.8}$$

where $\rho(\vec{x},t) = \sum_i f_i$ is the density. Here z is a parameter, we choose $z = \frac{1}{2}$. Also D is the dimension, here $D = 2$. $c = |e_i|$, here $c = 1$. And the speed of sound c_s is controlled by the parameter z by

$$c_s = \sqrt{\frac{1-z}{2}}. \tag{1.9}$$

The kinematic viscosity can be adjusted by choosing a proper relaxation parameter τ, and the relation is

$$\nu = \frac{c^2}{D+2}\left(\tau - \frac{1}{2}\right). \tag{1.10}$$

Recovering the Navier-Stokes Equations

The conservation laws From the definition of the unit vectors e_i, one can get the following equations [11]

$$\sum_i e_{i\alpha} = 0, \tag{1.11}$$

$$\sum_i e_{i\alpha}e_{i\beta} = \frac{c^2 b}{D}\delta_{\alpha\beta}, \tag{1.12}$$

$$\sum_i e_{i\alpha}e_{i\beta}e_{i\gamma} = 0, \tag{1.13}$$

$$\sum_i e_{i\alpha}e_{i\beta}e_{i\gamma}e_{i\delta} = \frac{c^4 b}{D(D+2)}(\delta_{\alpha\beta}\delta_{\gamma\delta} + \delta_{\alpha\gamma}\delta_{\beta\delta} + \delta_{\alpha\delta}\delta_{\beta\gamma}), \tag{1.14}$$

and

$$\sum_i e_{i\alpha}e_{i\beta}e_{i\gamma}e_{i\delta}e_{i\epsilon} = 0,$$

where $e_{i\alpha}$ stands for the α direction component (one of the i, j directions on the 2 dimentional plane) of the unit vector $\vec{e}_i$. Using the first two, one can obtain the moments

of the equilibrium distribution function. First sum the equilibrium distribution function and get conservation of mass and momentum

$$\sum_i f_i^{eq} = \rho, \tag{1.15}$$

and

$$\sum_i f_i^{eq} e_{i\alpha} = \rho u_\alpha. \tag{1.16}$$

Also, from the rest of the equations, one gets

$$\sum_i f_i^{eq} e_{i\alpha} e_{i\beta} = \frac{\rho(1-z)c^2}{D}\delta_{\alpha\beta} + \rho u_\alpha u_\beta, \tag{1.17}$$

and

$$\sum_i f_i^{eq} e_{i\alpha} e_{i\beta} e_{i\gamma} = \frac{\rho c^2}{D+2}(u_\alpha \delta_{\beta\gamma} + u_\beta \delta_{\alpha\gamma} + u_\gamma \delta_{\alpha\beta}). \tag{1.18}$$

The Chapman-Enskog expansion The distribution funcition can be expanded as follows

$$f_i = f_i^{(0)} + \epsilon f_i^{(1)} + \epsilon^2 f_i^{(2)} + ... \tag{1.19}$$

where $|\epsilon| \ll 1$. Here we can use the Knudsen number Kn as ϵ. The Knudsen number is defined as

$$Kn = \frac{\lambda}{L}$$

where λ is the mean free path, and L is the characteristic length. One can think this expension of the distuibution function f as an equilibrium distribution function $f^{(0)}$ together with some pertubations $f^{(1)}$, $f^{(2)}$,..., of higher order in ϵ. We also expand $\vec{x}$ and

t as

$$\vec{x} = \frac{\vec{x}_1}{\epsilon} + ..., \; t = \frac{t_1}{\epsilon} + \frac{t_2}{\epsilon^2} + ... \tag{1.20}$$

where $\vec{x}_1 = o(\epsilon)$, $t_1 = o(\epsilon)$, $t_2 = o(\epsilon^2)$. In this case, one can get

$$\frac{\partial}{\partial x_\alpha} = \epsilon \frac{\partial}{\partial x_{1\alpha}} + ..., \tag{1.21}$$

and

$$\frac{\partial}{\partial t} = \epsilon \frac{\partial}{\partial t_1} + \epsilon^2 \frac{\partial}{\partial t_2} + ... \quad .$$

Now we perform a Taylor expension on the Lattice Boltzmann equation (1.4) in both space and time, we obtain

$$\left[\left(\frac{\partial}{\partial t} + e_{i\alpha} \frac{\partial}{\partial x_\alpha} \right) + \frac{1}{2} \left(\frac{\partial}{\partial t} + e_{i\alpha} \frac{\partial}{\partial x_\alpha} \right)^2 \right] f_i(\vec{x}, t) = \Omega_i. \tag{1.22}$$

Notice that Einstein summation is used. So $e_{i\alpha} \frac{\partial}{\partial x_\alpha} = \sum_{\alpha=1,2} e_{i\alpha} \frac{\partial}{\partial x_\alpha}$. Using the expensions of f, $\frac{\partial}{\partial x_\alpha}$, $\frac{\partial}{\partial t}$, together with equation (1.5), we get

$$\begin{aligned} &\left[\left(\epsilon \frac{\partial}{\partial t_1} + \epsilon^2 \frac{\partial}{\partial t_2} + \epsilon e_{i\alpha} \frac{\partial}{\partial x_{1\alpha}} \right) + \frac{1}{2} \left(\epsilon \frac{\partial}{\partial t_1} + \epsilon^2 \frac{\partial}{\partial t_2} + \epsilon e_{i\alpha} \frac{\partial}{\partial x_{1\alpha}} \right)^2 \right] \\ &\quad \times \left(f_i^{(0)} + \epsilon f_i^{(1)} + \epsilon^2 f_i^{(2)} \right) = -\frac{1}{\tau} (f_i^{(0)} + \epsilon f_i^{(1)} + \epsilon^2 f_i^{(2)} - f_i^{eq}). \end{aligned} \tag{1.23}$$

Set the 0th order approximation $f_i^{(0)}$ to be the equilibrium distribution function f_i^{eq}. The conservation of mass and momentum require that $\sum_i f_i^{(k)} = 0$ and $\sum_i f_i^{(k)} e_{i\alpha} = 0$,

for $k = 1, 2$. From these equations to first-order in ϵ we get

$$\frac{\partial}{\partial t_1} f_i^{(0)} + e_{i\alpha} \frac{\partial}{\partial x_{1\alpha}} f_i^{(0)} = -\frac{f_i^{(1)}}{\tau}. \tag{1.24}$$

Summing over i and from equation (1.15) and (1.16) we get

$$\frac{\partial \rho}{\partial t_1} + \frac{\partial}{\partial x_{1\alpha}} \rho u_\alpha = 0. \tag{1.25}$$

Now multiply equation (1.24) by the unit vectors $e_{i\beta}$ and again sum over i, using equation (**??**) gives

$$\frac{\partial}{\partial t_1} \rho u_\beta + \frac{\partial}{\partial x_{1\alpha}} \rho u_\alpha u_\beta - \frac{\partial}{\partial x_{1\alpha}} \frac{\rho(1-z)c^2}{D} \delta_{\alpha\beta} = 0. \tag{1.26}$$

From equation (1.23) to second-order in ϵ and by equation (1.24) we get

$$\begin{aligned} \left[\frac{\partial}{\partial t_2} + \frac{1}{2} \frac{\partial}{\partial t_1} \left(\frac{\partial}{\partial t_1} + e_{i\alpha} \frac{\partial}{\partial x_{1\alpha}} \right) + \frac{1}{2} e_{i\alpha} \frac{\partial}{\partial x_{1\alpha}} \left(\frac{\partial}{\partial t_1} + e_{i\beta} \frac{\partial}{\partial x_{1\beta}} \right) \right] f_i^{(0)} \\ + \left(\frac{\partial}{\partial t_1} + e_{i\alpha} \frac{\partial}{\partial x_{1\alpha}} \right) f_i^{(1)} = -\frac{1}{\tau} f_i^{(2)}. \end{aligned} \tag{1.27}$$

Summing over i and using equation (1.25), all $\frac{\partial}{\partial t_1} + e_{i\alpha} \frac{\partial}{\partial x_{1\alpha}}$ vanished, and one gets

$$\frac{\partial}{\partial t_2} \rho = 0.$$

Again multiplying the equation by a unit vector $e_{i\gamma}$ gives the following

$$\begin{aligned} \left[\frac{\partial}{\partial t_2} e_{i\gamma} + \frac{1}{2} \frac{\partial}{\partial t_1} \left(\frac{\partial}{\partial t_1} e_{i\gamma} + e_{i\alpha} e_{i\gamma} \frac{\partial}{\partial x_{1\alpha}} \right) + \frac{1}{2} e_{i\alpha} e_{i\gamma} \frac{\partial}{\partial x_{1\alpha}} \left(\frac{\partial}{\partial t_1} + e_{i\alpha} \frac{\partial}{\partial x_{1\alpha}} \right) \right] f_i^{(0)} \\ + \left(\frac{\partial}{\partial t_1} e_{i\gamma} + e_{i\alpha} e_{i\gamma} \frac{\partial}{\partial x_{1\alpha}} \right) f_i^{(1)} = -\frac{1}{\tau} f_i^{(2)}. \end{aligned} \tag{1.28}$$

By multiplying equation (1.24) by $e_{i\alpha} e_{i\gamma} \frac{\partial}{\partial x_{1\alpha}}$, one can rewrite the second term of $f_i^{(1)}$ as

$$\frac{\partial}{\partial x_{1\alpha}} e_{i\alpha} e_{i\gamma} f_i^{(1)} = -\tau \left(\frac{\partial}{\partial t_1} \frac{\partial}{\partial x_{1\alpha}} e_{i\alpha} e_{i\gamma} f_i^{(0)} + \frac{\partial}{\partial x_{1\beta}} \frac{\partial}{\partial x_{1\alpha}} e_{i\alpha} e_{i\beta} e_{i\gamma} f_i^{(0)} \right). \tag{1.29}$$

Combining this term with the third term of $f_i^{(0)}$, one gets

$$\begin{aligned} &\left[\frac{\partial}{\partial t_2} e_{i\gamma} + \frac{1}{2} \frac{\partial}{\partial t_1} \frac{\partial}{\partial t_1} e_{i\gamma} + e_{i\alpha} e_{i\gamma} \frac{\partial}{\partial x_{1\alpha}} \right. \\ &\left. - \left(\tau - \frac{1}{2} \right) \left(\frac{\partial}{\partial t_1} \frac{\partial}{\partial x_{1\alpha}} e_{i\alpha} e_{i\gamma} + \frac{\partial}{\partial x_{1\beta}} \frac{\partial}{\partial x_{1\alpha}} e_{i\alpha} e_{i\beta} e_{i\gamma} \right) \right] f_i^{(0)} \\ &+ \frac{\partial}{\partial t_1} e_{i\gamma} f_i^{(1)} = -\frac{1}{\tau} f_i^{(2)}. \end{aligned} \tag{1.30}$$

Summing over i, the right-hand side is 0. The second term of $f_i^{(0)}$ is 0 by equation (1.26). The term of $f_i^{(1)}$ is 0 by the conservation of momentum requirement. The third term of $f_i^{(0)}$ can be obtained from equation (**??**) to the order $O(u)$ and then converting time derivatives into spatial derivatives using equation (1.25), we get

$$\begin{aligned} \frac{\partial}{\partial t_2} \rho u_\gamma &= \left(\tau - \frac{1}{2} \right) \left[\frac{\partial}{\partial t_1} \frac{\partial}{\partial x_{1\alpha}} \frac{\rho(1-z)c^2}{D} \delta_{\alpha\gamma} \right. \\ &\left. + \frac{\partial}{\partial x_{1\alpha}} \frac{\partial}{\partial x_{1\beta}} \frac{\rho c^2}{D+2} \left(u_\alpha \delta_{\beta\gamma} + u_\beta \delta_{\alpha\gamma} + u_\gamma \delta_{\alpha\beta} \right) \right] \\ &= \left(\tau - \frac{1}{2} \right) \left[\frac{\partial}{\partial x_{1\alpha}} \frac{\partial}{\partial x_{1\alpha}} \frac{\rho c^2}{D+2} u_\gamma \right. \\ &\left. + \frac{\partial}{\partial x_{1\gamma}} \left(\left(\frac{2c^2}{D+2} - \frac{(1-z)c^2}{D} \right) \frac{\partial}{\partial x_{1\alpha}} \rho u_\alpha \right) \right]. \end{aligned} \tag{1.31}$$

By setting the kinematic shear viscosity $\nu = \left(\tau - \frac{1}{2}\right) \dfrac{c^2}{D+2}$ and the kinematic bulk viscosity $\varsigma = \left(\tau - \frac{1}{2}\right) \left(\dfrac{2c^2}{D+2} - \dfrac{(1-z)c^2}{D}\right)$, the above equation becomes

$$\frac{\partial}{\partial t_2}\rho u_\gamma = \nu \frac{\partial^2}{\partial x_{1\alpha}^2}\rho u_\gamma + \frac{\partial}{\partial x_{1\gamma}}\left(\varsigma \frac{\partial}{\partial x_{1\alpha}}\rho u_\alpha\right). \tag{1.32}$$

Using all these equations (provided above), one can show that the Navier-Stokes equation, the momentum equation

$$\frac{\partial \rho}{\partial t}u_\alpha + \frac{\partial}{\partial x_\beta}\rho u_\beta u_\alpha = -\frac{\partial}{\partial x_\beta}\left[\frac{\rho(1-z)c^2}{D}\delta_{\alpha\beta}\right] + \nu\frac{\partial^2}{\partial x_\beta^2}\rho u_\alpha + \frac{\partial}{\partial x_\alpha}\left(\varsigma\frac{\partial}{\partial x_\beta}\rho u_\beta\right) \tag{1.33}$$

and the continuity equation

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_\alpha}\rho u_\alpha = 0 \tag{1.34}$$

are satisfied. For an incompressible flow, these two equations reduce to equation (1.1) and (1.2).

Boundary and initial conditions

The bounce back scheme is still good for the no-slip boundary condition. Bounce back boundary conditions only give first-order accuracy. A Dirichlet boundary condition can be achieved by solving the system of equations at the boundary sites

$$f_1 + \frac{1}{2}f_2 - \frac{1}{2}f_3 - f_4 - \frac{1}{2}f_5 + \frac{1}{2}f_6 = \rho u_x, \tag{1.35}$$

$$\frac{\sqrt{3}}{2}\left(f_2 + f_3 - f_5 - f_6\right) = \rho u_y, \tag{1.36}$$

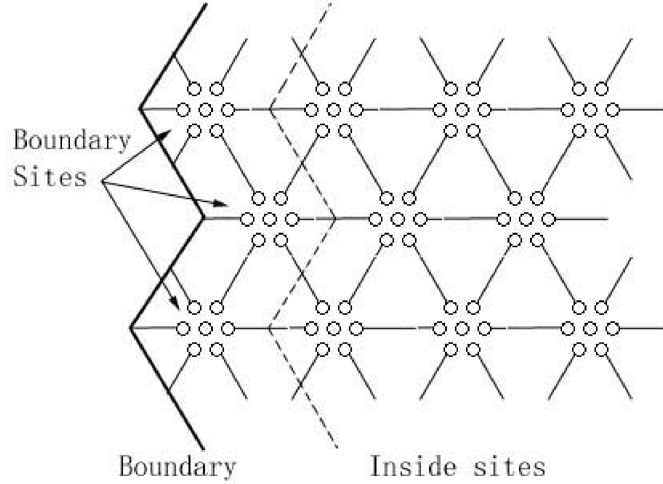

Figure 1.4: Left boundary of a hexagonal grid.

where $\vec{u} = (u_x, u_y)$ is the velocity vector. For example, in the simulation of a driven cavity flow, assume the top boundary has the speed $\vec{u} = (u_0, 0)$. The f_2, f_3 (up directions) are from the inside sites, and you can keep f_1, f_4 (horizontal directions) as constants. Only f_5, f_6 (down directions) are unknowns. This is a linear system of equations involving two unknowns. Notice that when $\vec{u} = (0, 0)$, it's a bounce back scheme.

The left boundary shown in Figure 1.4 is not smooth in a microscopic view because of the hexagonal structure of the grid. But it is smooth enough in a macroscopic view. One can take the macroscopic boundary as the average of this boundary.

For initial conditions, one may use the equilibrium distribution from the given values of ρ and $\vec{u}$. Bad initial distribution, for example, far away from the equilibrium distribution, will make the method unstable, and eventually lead to blow up.

Implementation

Consider a driven cavity flow again. This time, an array of 7 floating-point variable is needed for the information at one site. So we create a $T \times 7$ matrix M, where T is the total number of sites. We number the sites the same way as in FHP. The program structure is also pretty much the same as in FHP, except that in the LBGK method, the collision and the streaming are combined together by the Lattice Boltzmann equation. The equilibrium distribution function is a mapping from the given ρ and $\vec{u}$ to the matrix M. So first we can use this to initialize M. In the collision, we calculate the ρ and $\vec{u}$ by

$$\begin{aligned} \rho(\vec{x}, t) &= \sum_i M(n, i), && (1.37) \\ u_x(\vec{x}, t) &= \vec{M}(n) \cdot \vec{e}_{ix}, && (1.38) \end{aligned}$$

and

$$u_y(\vec{x}, t) = \vec{M}(n) \cdot \vec{e}_{iy}, \tag{1.39}$$

where n is the number of the site corresponding to $\vec{x}$, and $\vec{e}_i = (\sin(\frac{\pi}{2}(i-1)), \cos(\frac{\pi}{2}(i-1)))$, which is the link to the 6 neighbour sites. Then apply the weighted equilibrium distribution function (equation 1.6) with a proper τ.

Results and data analysis

Driven Cavity Here we present a driven cavity example again. In figures 1.5, 1.6, 1.7, 1.8, and 1.10, we give the velocity vectors (left) and velocity contour (right) at the steady state for Reynolds number 10, 100, 200, 400 and 800. We give the result of Ghia, Ghia, and Shin [12] for Reynolds number at 400 for comparison (The computations were

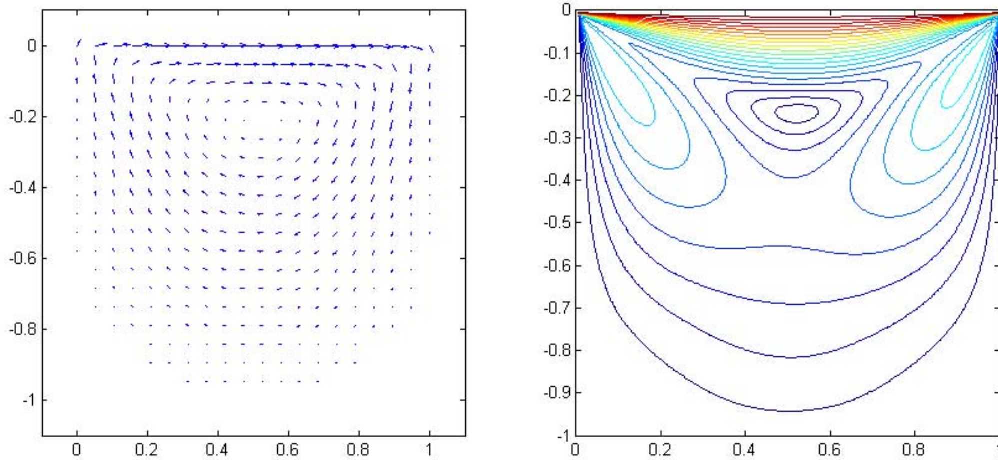

Figure 1.5: Driven cavity at $Re = 10$.

performed using the time-marching capabilities of WIND to approach the steady-state flow starting from the freestream conditions). We also give the velocity profiles for u and v through the geometirc center of the cavity. For comparison, refer to Shuling Hou and Qisu Zou et al [13].

Flow past a cylinder

This is an example of a uniform flow past a cylinder. This example is done on a 360×1000 grid. The speed of the uniform flow coming from the left is 0.5. And the flow is free to flow out at the right end. The cylinder was placed at the center of the left inlet with a diameter 120. The Reynolds number is 400. The top and bottom are no-slip boundaries. It is well know that at a Reynolds number greater than 100, the flow past a cylinder will give a Von Karman vortex street. Here we give the figures of both velocity

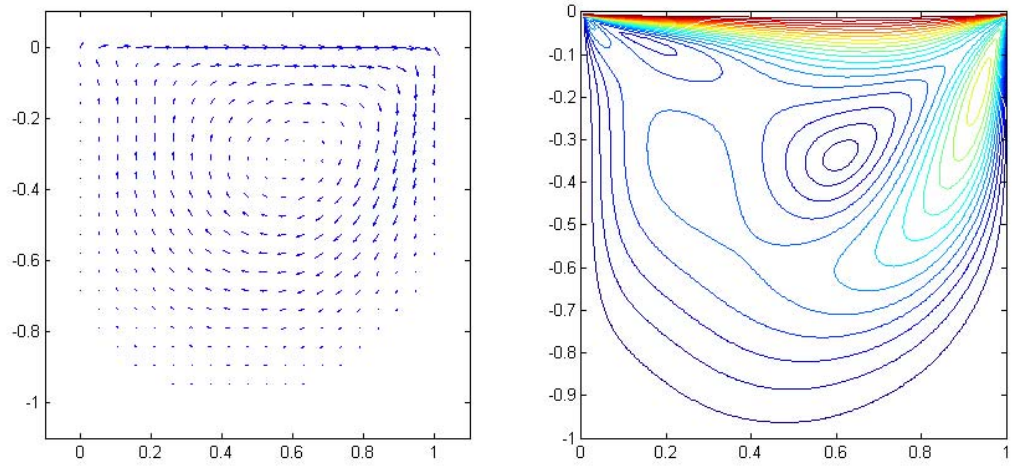

Figure 1.6: Driven cavity at $Re = 100$.

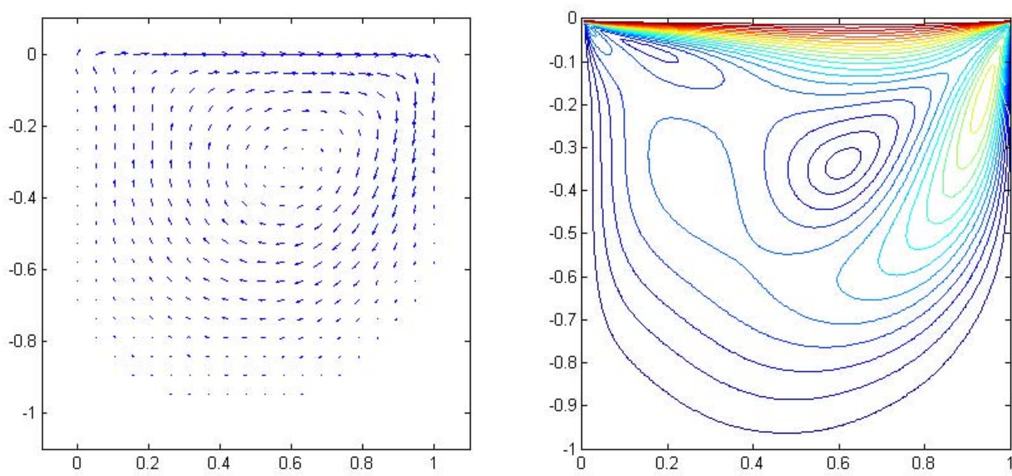

Figure 1.7: Driven cavity at $Re = 200$.

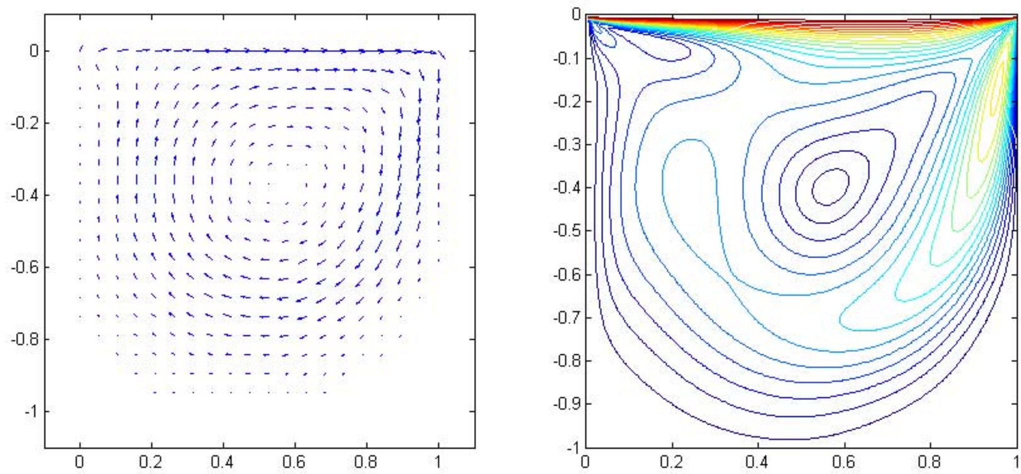

Figure 1.8: Driven cavity at $Re = 400$.

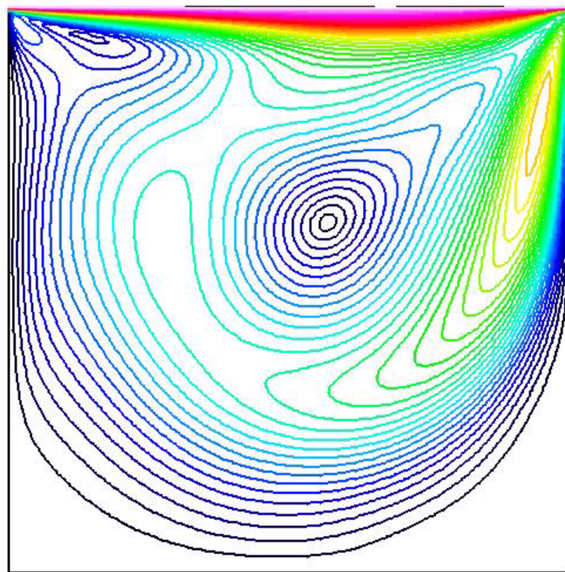

Figure 1.9: Ghia Ghia Shin's result. The plot of the velocity contour with a Reynolds number of 400

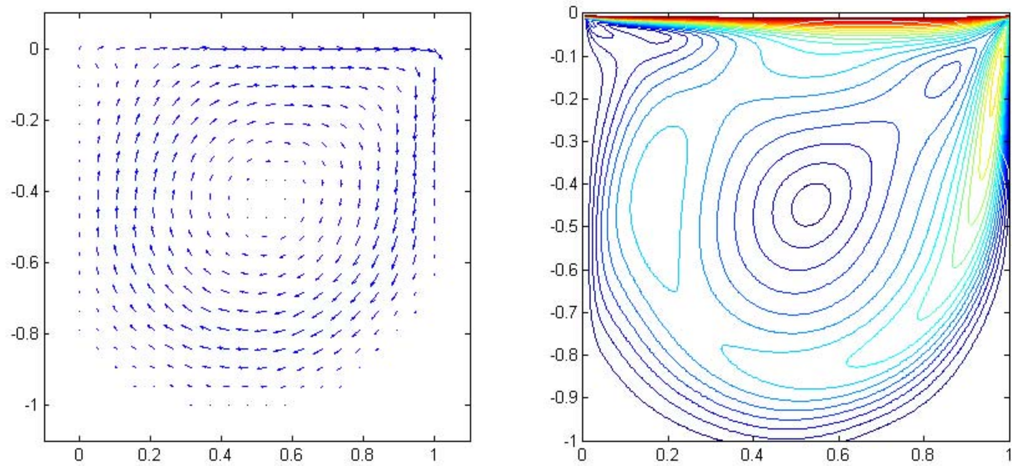

Figure 1.10: Driven cavity at $Re = 800$.

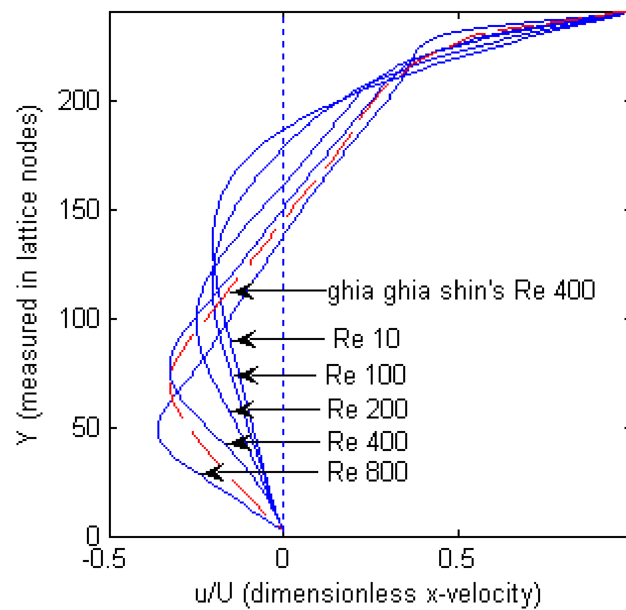

Figure 1.11: Dimensionless x-velocity profile at the geometry center of the cavity for Reynolds number 10, 100, 200, 400, 800. Dashed line is the result from Ghia Ghia Shin at Reynolds number 400.

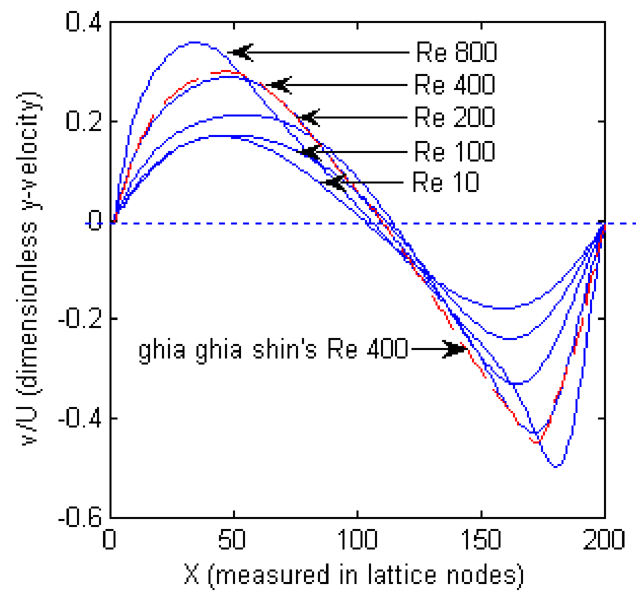

Figure 1.12: Dimensionless y-velocity profile at the geometry center of the cavity for Reynolds number 10, 100, 200, 400, 800. Dashed line is the result from Ghia Ghia Shin at Reynolds number 400.

contour and vorticity contour at time step 5000. Both figures show the back half of the cylinder.

1.3.2 Other frequently used Lattice Boltzmann Methods

D2Q9 method

D2Q9 method is the most commonly used 2 dimensional LBM. It has 3 discrete grid speed: 0, c and $\sqrt{2}c$. It has 9 directions: 8 moving directions, plus the rest particle. This can be discribed by the link vectors that point to the neighbors, which read

$$e_i = (0,0) \qquad i=0, \tag{1.40}$$

$$e_i = (\pm c, 0), (0, \pm c), \qquad i = 1,2,3,4, \tag{1.41}$$

and

$$e_i = (\pm c, \pm c), \qquad i = 5,6,7,8. \tag{1.42}$$

Figure 1.14 shows the D2Q9 Lattice. The equilibrium dsitribution is

$$f_i^{eq}(x,t) = W_i \rho(x,t) \left[1 + 3\frac{e_i \cdot \overrightarrow{u}}{c^2} + \frac{9}{2}\frac{(e_i \cdot \overrightarrow{u})^2}{c^4} - \frac{3}{2}\frac{\overrightarrow{u}^2}{c^2} \right], \qquad i = 0, ..., 8, \tag{1.43}$$

where

$$W_i = \frac{4}{9}, \qquad i = 0, \tag{1.44}$$

$$W_i = \frac{1}{9}, \qquad i = 1,2,3,4, \tag{1.45}$$

$$W_i = \frac{1}{36}, \qquad i = 5,6,7,8, \tag{1.46}$$

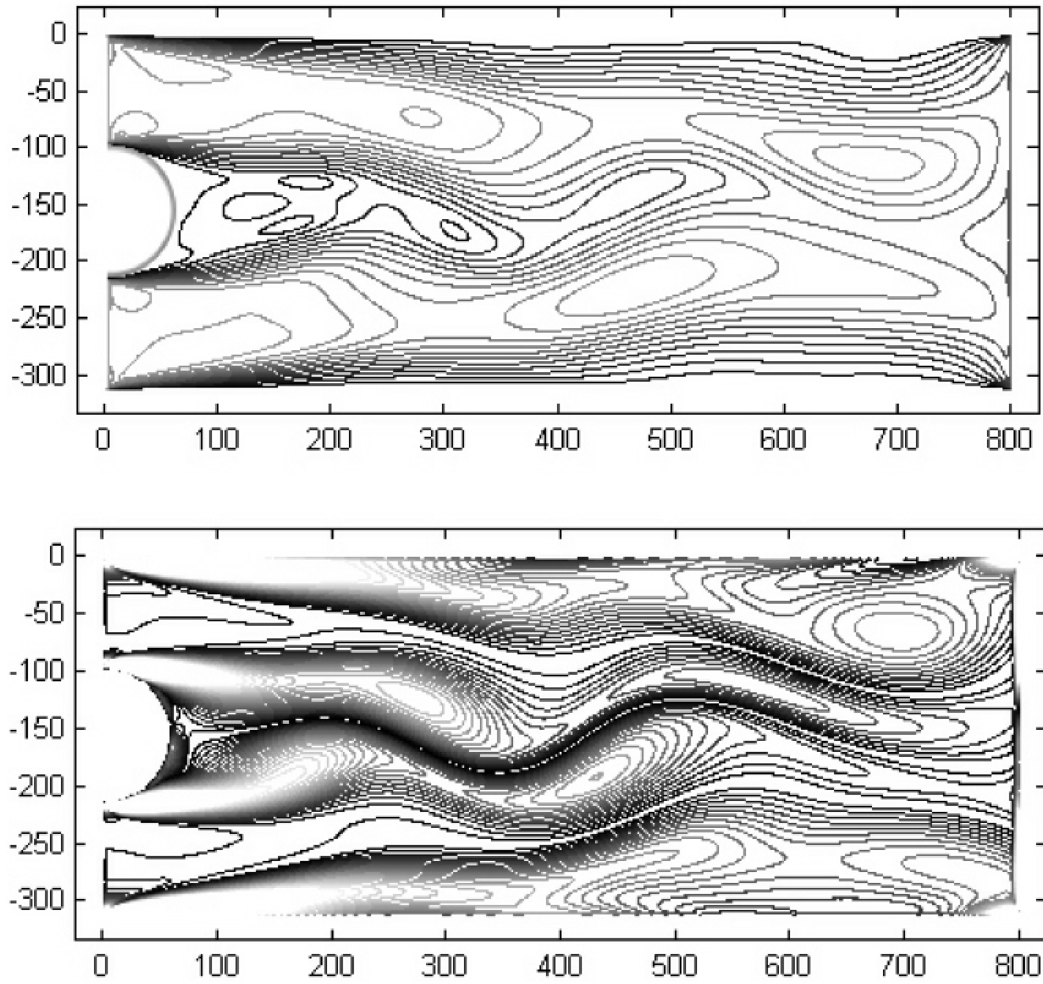

Figure 1.13: An example of a uniform flow past a cylinder for $Re = 400$. Above: velocity contour. Below: vorticity contour.

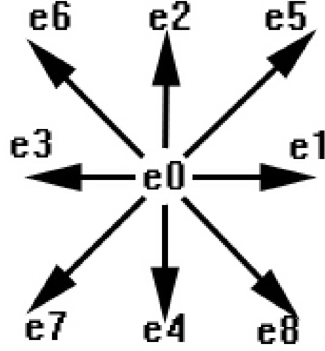

Figure 1.14: The D2Q9 lattice.

and the kinematic viscosity is given by

$$\nu = \frac{c^2}{3}\left(\tau - \frac{1}{2}\right). \tag{1.47}$$

D3Q19 method

The D3Q19 is a 19-direction 3-dimensional LBM. The link vectors are

$$e_i = (0,0,0) \qquad i = 0, \tag{1.48}$$

$$e_i = (\pm c,0,0),(0,\pm c,0),(0,0,\pm c), \qquad i = 1,...,6, \tag{1.49}$$

and

$$e_i = (\pm c,\pm c,0),(\pm c,0,\pm c),(0,\pm c,\pm c) \qquad i = 7,...,18. \tag{1.50}$$

Note that just like the D2Q9 LBM, D3Q19 also features 3 discrete speed: 0, c, and $\sqrt{2}c$, so it shares the same form of equilibrium distribution with D2Q9, with a different set of

W_is, which read

$$
\begin{aligned}
W_i &= \frac{1}{3}, \qquad i = 0, &(1.51)\\
W_i &= \frac{1}{18}, \qquad i = 1, ..., 6, &(1.52)\\
W_i &= \frac{1}{36}, \qquad i = 7, ..., 18. &(1.53)
\end{aligned}
$$

1.4 Applications

In the next 3 Chapters, we will give several applications of the Lattice Boltzmann method. In Chapter 2, we will talk about the moving boundary problems using the Lattice Boltzmann method. We present two different moving boundary treatments. In Chapter 3, we give an incompressibility enhancement of the Lattice Boltzmann method. In Chapter 4, we give an application that couples the Lattice Boltzmann method with a Poisson's equation to solve a MHD flow problem.

Chapter 2

Moving Boundary Problems

2.1 Background

Basic types of problems

The moving boundary problems are a type of CFD problems which have time dependent solid moving boundaries. Numerous physical phenomena involve solid-fluid interaction. This is in contrast to a static boundary where the boundary nodes are fixed on a fixed mesh. These types of problems include deformation of structures, moving solid objects, and boundaries that evolve in time. The main methods of solving these moving boundary problems can be briefly classified into two major categories: the Lagrangian methods, and the Eulerian methods.

On one hand, the Lagrangian methods maintain a mesh that follows the movement of the solid. The solid-fluid interface is explicitly and accurately tracked. And boundary conditions can be applied at the exact location of the interface at each time step. However, the regeneration of the mesh is time consuming. Also, in some cases, the resulting grid may be unevenly distributed. This degrades the accuracy on the boundary.

The Eulerian methods, on the other hand, use a mesh that remains fixed. These method can roughly be categorized into diffuse type and sharp type. For the diffuse type, the exact location of the solid-fluid interface is unknown. The methods accuracy is to the order of the grid scale. The boundary conditions are applied in the sense of the governing equations of the fluid and the solid. The obvious advantage is that there is no need to regenerate the mesh at every time step. However, due to the unclarity of the

exact boundary location, when the interface is arbitrarily shaped, improved resolution is diffcult to obtain.

Perhaps the more commonly used method is the immersed moving boundary method (Peskin 1977). A fixed Cartesian grid is used, and an explicit sharp interface is defined. This method is a mixture of Eulerian and Lagrangian framework. Clearly, this method shares the virtures of both the Eulerian and the Lagrangian method: accurate boundary condition can be applied without regenerating the mesh. At the same time, however, several problems arise. One issue that needs to be dealt with is keeping track of those nodes that go from the solid state to fluid state, or fluid state to solid state. This is an issue that is not encountered in a pure Lagrangian or pure Eulerian method. A steep change will result in a huge discontinuity on the boundary. One method is to use a fractional-step scheme (H. S. Udaykumar), which is based on a one-dimensional interpolation.

Basic boundary types for LBM

For Lattice Boltzmann method, methods that are close to the immersed moving boundary can be used. This is due to the nature of the lattice boltzmanm method, that is, it uses a fixed Cartisian grid, and it is easy to manage local behaviors. In the next section, we will present two methods for treating moving boundaries in the LBM. The first one is a widely used method which based on a quadratic interpolation. This method is second order accurate, but at the cost of violating mass conservation. The second method we will present is a newly developed method, which is closer in spirit to the fractional-step scheme. It conserves mass, and the change of state is gradual, without a steep discontinuity.

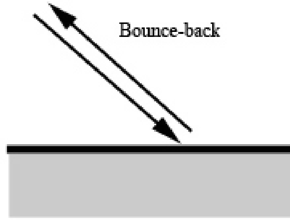

Figure 2.1: No-slip Boundary

Before continuing, we describe several basic boundary types for LBM. The two most common types of boundary conditions are: Dirichlet boundary condition and Newmann boundary condition. In LBM, the simple realizations are of no-slip boundary and slip boundary (Figure 2.1, 2.2). The no-slip boundary uses a bounce-back scheme, without considering the angle of incidence of the particle. This makes the no-slip boundary very easy to implement, since one doesn't need to know the exact shape of the boundary. The slip boundary is of reflection type. The angle of incidence should equal to the angle of reflection. This makes the slip boundary much more difficult and complicated to implement.

The moving particles in LBM travell to the neighbor node in one time step. In order to make sure that all particles are on the nodes, one could either set the boundary at the nodes, or in the middle of two neighbor nodes. Figure 2.3 shows the two normally used boundary positions. One of the difficulties of the moving boundary is to find a way so that the boundary can be set in any intermediate position.

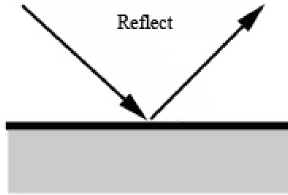

Figure 2.2: Slip Boundary

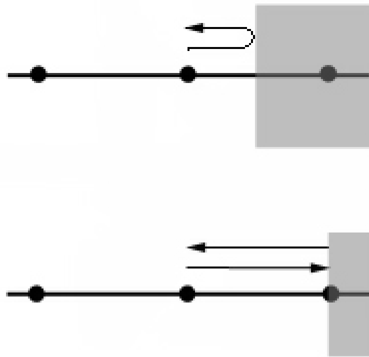

Figure 2.3: Above: boundary on the half-way; Below: boundary on the node

2.2 Moving boundary for LBM

2.2.1 Moving boundary without mass conservation

The scheme

A moving boundary treatment for LBM was proposed by P. Lallemand and L.S. Luo [14]. This method is based on the simple bounce-back boundary scheme and quadratic interpolations, yielding a no-slip boundary condition. As stated in the previous section, the difficulty lies in defining the boundary at a position other than the half-way between nodes and on the nodes. For simplicity, we just study the case of one particle direction. Figure 2.4 shows two cases when the boundary is not at a standard position. Here q is the distance between the boundary and the closest node, assuming that the distance between any two neighboring nodes is 1.

For the case $q < 1/2$, the problem is how to define the left direction particle on node r_j, since it should came from the right direction particle traveling between r_j and r'_j at the previous time step. The idea is to implement a quadratic interpolation at position x using the information on r_j, r'_j, and r''_j. For the case $q > 1/2$, position x is outside of r_j, r'_j, and r''_j. To avoid using extrapolation, one can do the streaming (propagation) first, and then implement a quadratic interpolation on node r_j using the information on x, r'_j, and r''_j.

Also to be considered is the extra term F_i introduced by the fluid-solid interaction. By considering the mass conservation $\sum F_i = 0$ and the momentum conservation $\sum \overrightarrow{e_i} F_i = \rho \overrightarrow{u_w}$, one can get $F_i = w_i(\overrightarrow{e_i} \cdot \overrightarrow{u_w})$, where $\overrightarrow{u_w}$ is the speed of the moving boundary, and w_i is the weight of the mass on i direction.

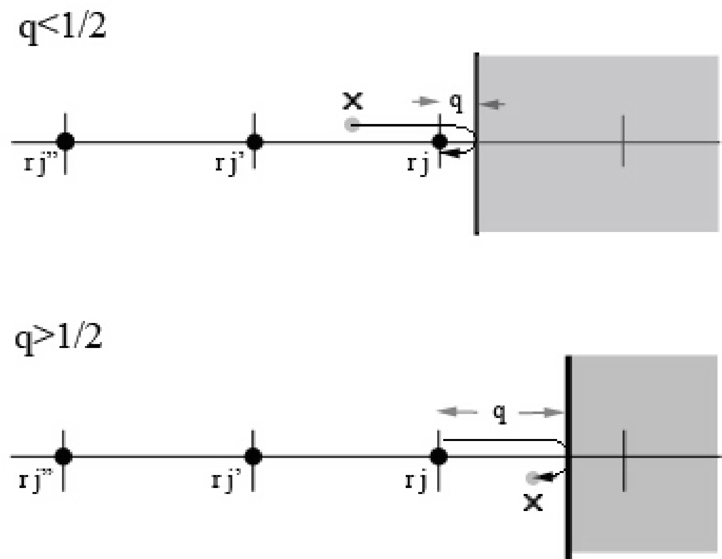

Figure 2.4: Cases when q>1/2 or q<1/2.

The actual formulas are: For $q < 1/2$

$$f_i(r_j,t) = q(1+2q)\widehat{f}_i(r_j,t) + (1-4q^2)\widehat{f}_i(r'_j,t) - q(1-2q)\widehat{f}_i(r''_j,t) + w_i(\overrightarrow{e_i}\cdot\overrightarrow{u_w}), \quad (2.1)$$

And for $q > 1/2$

$$f_i(r_j,t) = \frac{1}{q(2q+1)}\widehat{f}_i(r_j,t) + \frac{2q-1}{q}f_i(r'_j,t) - \frac{2q-1}{2q+1}f_i(r''_j,t) + \frac{w_i}{q(2q+1)}(\overrightarrow{e_i}\cdot\overrightarrow{u_w}), \quad (2.2)$$

where $\widehat{f}_i$ is the distribution from the previous time step, and for D2Q9 LBM,

$$w_i = \begin{cases} 2/3, & i = 1,2,3,4 \\ 1/6, & i = 5,6,7,8 \end{cases} . \quad (2.3)$$

Note in the case $q > 1/2$, there is a correction of $1/[q(2q+1)]$ [14]. This is obtained by considering the analytic solution for the Couette flow. These two formulas coincide when $q = 1/2$, and reduce to a simple bounce-back scheme.

Change of state treatment

For immersed moving boundary problems with a Cartisian grid and a sharp interface, the change of state (a node goes either from fluid state to solid state, or from solid state to fluid state) should be carefully considered. Figure 2.5 shows nodes that change state as a sharp interface moves. The nodes that changed their state are marked with triangles. The two different states are marked with squares and circles.

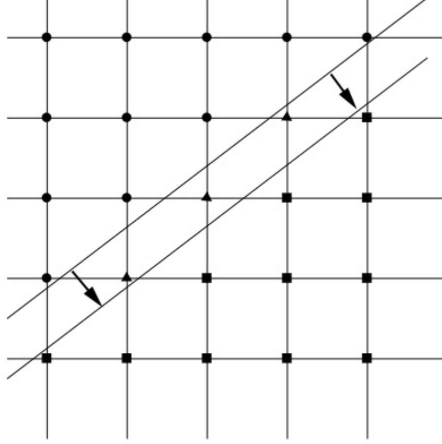

Figure 2.5: The change of state illustration.

On the one hand, the change of a node from the fluid state to solid state is simplely considered a non-issue, since the fluid motion is not calculated in solid. And the treatment is to set the distribution on those nodes to zero. On the other hand, the change of a node from the solid state to fluid state is not trivial. Some method has to be used to "create" the particle distribution of the LBM at those nodes.

Several methods could be used. A commonly used method is to use the information from neighbors and extrapolate the distributions. One should use the direction which is closest to $\overrightarrow{n}$, the out-normal vector of the moving interface. That is, to pick the direction $\overrightarrow{e_i}$ which maximizes the quantity $\overrightarrow{n} \cdot \overrightarrow{e_i}$. Then, use a linear or quadratic extrapolation, or some other method, to get the distributions on the newly changed state nodes. Another method is to use the velocity of the moving interface and the average density of the system, or a local average density, to get the distributions.

An example

2.2.2 Moving boundary with mass conservation

An obvious compromise of the quadratic interpolation moving boundary is that the mass is not conserved. Perhaps the total mass conservation in the whole system is not violated, but locally the problem could be serious in some cases. In Figure 2.5 consider the moving boundary as a slim bar. What the quadratic interpolation does is, it continuously eleminate particles that are in front of the bar along the moving direction, and creates particles behind it. This results in an effect that fluid particles are continuously infiltrating through the bar, this reduces the presure difference between the two sides of the bar. If this issue is critical to the whole system, then some other method should be used. This is the motivation for us to develop a new method for LBM that conserves mass.

The scheme

Let's consider again a half-way bounce-back boundary shown in Figure 2.6. And first let's only consider this in one direction. Then at the next time step, due to the streaming, the following changes will be performed.

$$\begin{aligned} f_1 &= f_1', \\ f_2' &= f_2, \end{aligned}$$

and

$$f_2 = f_1.$$

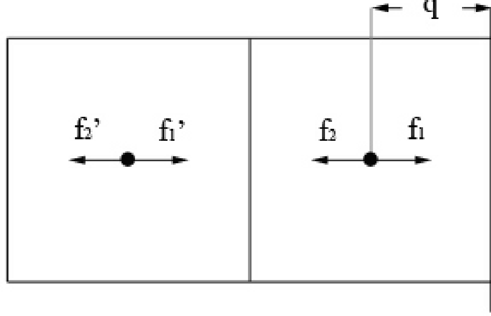

Figure 2.6: A half-way bounce-back boundary.

In the case when $q < 1/2$, the boundary nodes no longer occupy a whole unit space, but just a part of it. The first graph in Figure 2.7 shows this case. In order to explain things, the second graph in Figure 2.7 reflects the left direction distribution to the right side. This makes all the particles to be right direction particles. After the streaming process, all particles move to the right by one unit distance, as indicated in the third graph of Figure 2.7. Then the space occupied by f_1, f_2, and f_2' (second graph of figure 2.7) has been replaced by f_1', f_1, and f_2 (third graph of figure 2.7). By comparing the space being occupied, one gets the transition formulas

$$f_1 = (1/2+q)f_1', \tag{2.4}$$

$$f_2' = f_2 + \frac{(1/2-q)}{(1/2+q)}f_1, \tag{2.5}$$

and

$$f_2 = \frac{2q}{(1/2+q)}f_1 + (1/2-q)f_1'. \tag{2.6}$$

Similarly, as shown in Figure 2.8, in the case when q>1/2, the transition formulas read

$$
\begin{aligned}
f_1 &= \frac{(q-1/2)}{(q+1/2)} f_1 + f_1', & (2.7)\\
f_2' &= \frac{f_2}{q+1/2}, & (2.8)
\end{aligned}
$$

and

$$f_2 = \frac{q-1/2}{q+1/2} f_2 + \frac{1}{q+1/2} f_1. \qquad (2.9)$$

In either of these two cases, the occupied space of the boundary nodes does not equal to a unit space, thus the f_is can not represent the true distribution of a LBM on that node. One can use the normalized distribution $\widetilde{f_i} = f_i/(1/2+q)$ to get macro quantities like density and velocity on the boundary nodes. For a D2Q9 method, one needs to find the q for all directions. Figure 2.9 shows a typical example of a boundary node. The occupied space on each direction is different. But in most of the cases, for opposite directions, they are the same. The collision process includes normalizing the distribution, collision, then un-normalizing. In this process, the mass is not perfectly conserved. However, if the speed of the moving boundary is much slower than the grid speed, this imperfection can be neglect.

The change of state

In this scheme, the change of state transition is smooth and requires no extra work. As shown in Figure 2.10, the upper graph shows a situation when the boundary is very close to a node, which is the case $q < 1/2$. Although the occupied space is small on the boundary node, the normalized distribution should not be too different from the neighbor

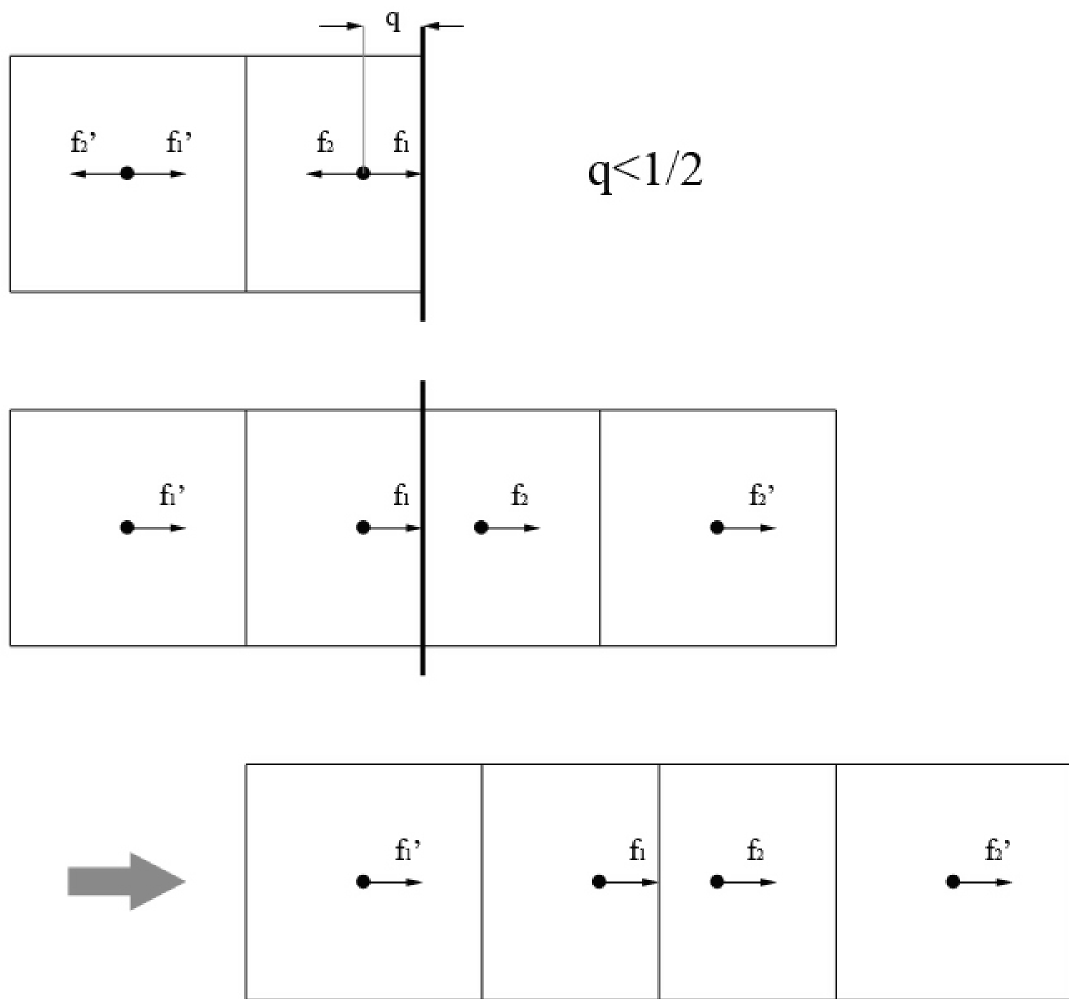

Figure 2.7: Bounce-back boundary when $q < 1/2$.

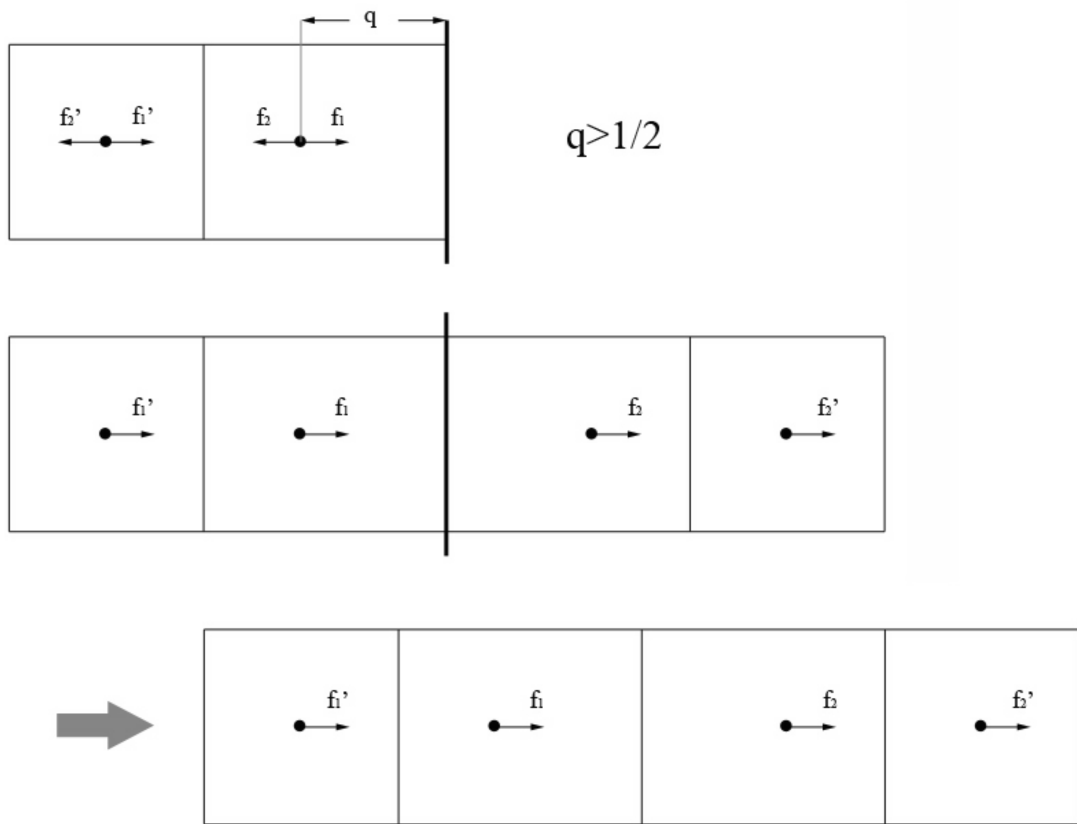

Figure 2.8: Bounce-back boundary when $q > 1/2$.

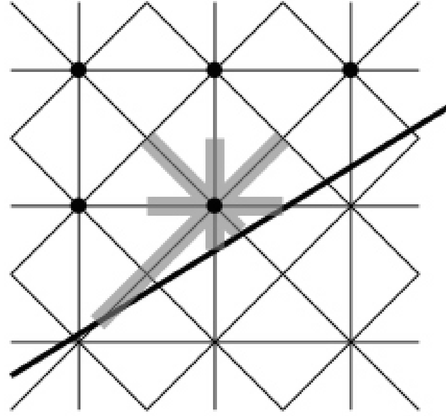

Figure 2.9: An example of a boundary node.

node. Consider the case that after a time step, this node is coverd by the boundary, as shown in the lower graph of Figure 2.10. Now the node is gone, but most of the volume is still there. And the neighbor node becomes the boundary node with $q > 1/2$. So very naturally, it will take over this part that the previous boundary node left. Given that the boundary moves slowly enough, the sum of the occupied space of the previous two node will not differ too much from the occupied space of the later boundary node. Thus a smooth transition is achieved.

2.3 Sharp boundary definition details

The implementation of the moving boundary requires the knowledge of the exact location of the sharp boundary. We demostrate the definition of two simple and widely used boundaries: line and arc. Before locating the exact boundary, one needs to find the

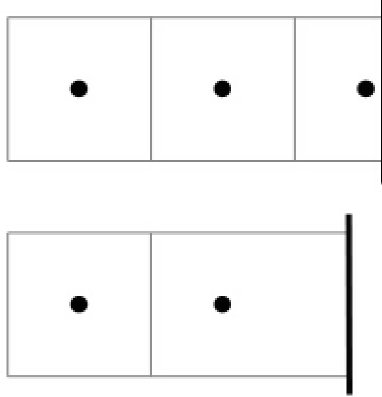

Figure 2.10: The change of state in this bounce-back based scheme.

boundary layer first. This is a layer that contains nodes that are closest to the boundary in all directions.

2.3.1 Line

The definition of a line can be done by using a normal vector of this line. The advantage of this is that, by taking the projection along the normal vector, it is very easy to get the perpendicular distance to the line. Figure 2.11 shows a straight line boundary, which lies between $\mathbf{v}$ and $\mathbf{v\prime}$. $\mathbf{n}$ is the normal vector to the line, and l is the shortest distance between the line and the center point c. We wish to find q, the distance between v and the line along $\mathbf{v\prime} - \mathbf{v}$. Assume that $|\mathbf{v} - \mathbf{v\prime}| = 1$, q is given by

$$q = \frac{\mathbf{v} \cdot \mathbf{n} - l}{(\mathbf{v} - \mathbf{v}') \cdot \mathbf{n}}.$$

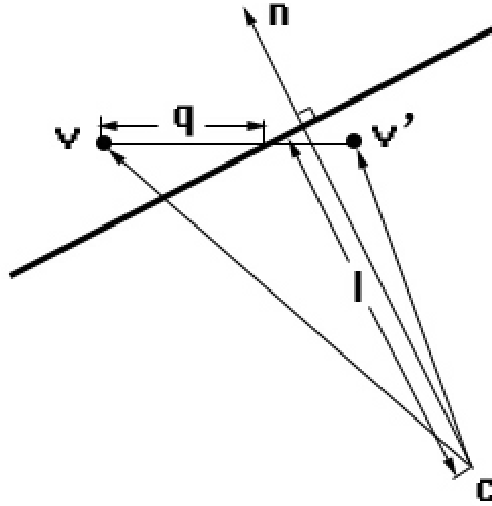

Figure 2.11: The definition of a straight line boundary.

2.3.2 Arc

The arc is of course defined by a partial circle. Figure 2.12 shows an arc boundary, where r is the radius of the arc. Let $\mathbf{e}_i = \mathbf{v}' - \mathbf{v}$, the distance of v to the arc along $\mathbf{e}_i$ is given by

$$q = \mathbf{v} \cdot \frac{\mathbf{e}_i}{|\mathbf{e}_i|} - \sqrt{r^2 - \left| \mathbf{v} - (\mathbf{v} \cdot \frac{\mathbf{e}_i}{|\mathbf{e}_i|}) \frac{\mathbf{e}_i}{|\mathbf{e}_i|} \right|^2}.$$

2.4 Implementation

Here we present an example of this moving boundary scheme. This example is a rotating triangle in a square domain full of fluid. All boundaries are no-slip boundaries. This example is done in a 300300 grid using D2Q9 Lattice Boltzmann method. Figure

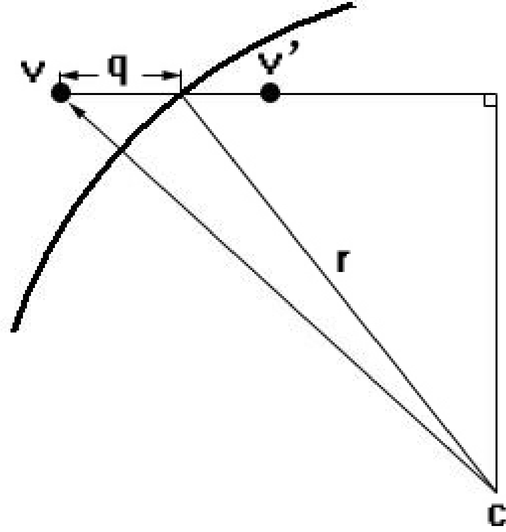

Figure 2.12: The definition of an arc boundary.

2.13 shows the velocity contour with streamlines. This picture is taken after 2 full cycles (720 degree).

2.5 Conclusion and Future Work

In this chapter we presented two moving boundary treatment for the Lattice Boltzmann method. Both define sharp boundaries, and both require that the speed of the moving boundary is much slower than the grid speed c. The first uses quadratic interpolation. It is of second order accurate, but at the cost of voilating the mass conservation. The second is of first order accurate, and conserves mass.

We would like to point out a future work of the second mass-conserved moving boundary treatment. One draw back of this scheme is that it lacks the shear stress. The reason is that it only consider the normal direction movement of the boundary, but not

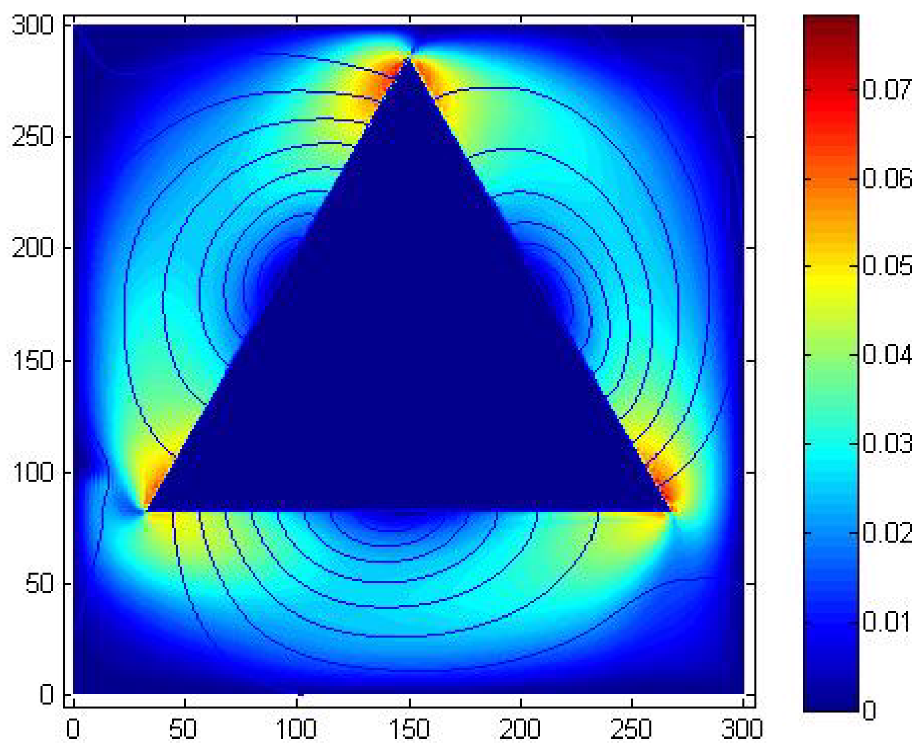

Figure 2.13: The rotating triangle example.

the tangential direction movement. The first treatment is also the same, but since it count the momentum transfer, the effect is not severe. This may explain why the part of the streamlines that connect to the boundary in figure 2.13 is almost perpendicular to the boundary. So a future work of this moving boundary treatment is to consider the shear stress that is given by the tangential direction movement of the boundary.

Chapter 3

Incompressibility

3.1 Background

In the previous chapters, we used Lattice Boltzmann method on incompressible flows. However, we need to point out that an incompressible flow is an ideal flow. It only exists in theory. In practice, any flow is compressible to some extent. The Lattice Boltzmann method, which is based on the Boltzmann equation for gas, simulates compressible fluids with some finite speed of sound c_s. When the fluid speed is sufficiently small compared with c_s, we should get a solution that converges to the incompressible limit.

In recent years, several improvement were made to the Lattice Boltzmann method in order to better approximate the incompressibility. This can be roughly categorized into two groups: the improved single-relaxation time model (Lattice BGK method), and the multi-relaxation time model [15]. The unique nature of the multi-relaxation time model makes it a better method for simulating incompressible flows. However, due to the simplicity of implementation and being 30% more efficient than the multi-relaxation time model, the Lattice BGK method is the favorite method for many. Unfortunately, theoretically it is impossible to maintain a constant density in the Lattice BGK method. In this chapter, we focus on the Lattice BGK method.

In the real world, for an incompressible fluid, the density $\rho = \rho_0 + \delta\rho$, where ρ_0 is a constant and $\delta\rho$ is the density fluctuation, which should be of the order $O(M^2)$ (M is the Mach number, and $M \longrightarrow 0$). X. He and L-S Luo [16] improved the Lattice

BGK method by substituting $\rho = \rho_0 + \delta\rho$ into the equilibrium distribution function f_i^{eq}, neglecting the terms of the order $O(M^2)$ and higher. For a D2Q9 Lattice BGK method, the result is a new equilibrium distribution function, which reads

$$f_i^{eq}(x,t) = w_i \left\{ \rho + \rho_0 \left[3\frac{(\overrightarrow{e_i} \cdot \overrightarrow{u})}{c^2} + \frac{9}{2}\frac{(\overrightarrow{e_i} \cdot \overrightarrow{u})^2}{c^4} - \frac{3}{2}\frac{\overrightarrow{u}^2}{c^2} \right] \right\}. \tag{3.1}$$

This improved method can simulate both steady and unsteady flow problems.

Another improved Lattice BGK method is provided by Guo, Shi and Wang [17]. This scheme redefined the distribution when the fluid is at rest ($\overrightarrow{u} = 0$). The new equilibrium distribution function (again, for D2Q9) is

$$g_i^{eq} = v_i \frac{p}{c^2} + w_i \left[3\frac{(\overrightarrow{e_i} \cdot \overrightarrow{u})}{c^2} + \frac{9}{2}\frac{(\overrightarrow{e_i} \cdot \overrightarrow{u})^2}{c^4} - \frac{3}{2}\frac{\overrightarrow{u}^2}{c^2} \right], \tag{3.2}$$

and the macroscope velocity and pressure are given by

$$u = \sum_{i=1}^{8} c\overrightarrow{e_i} g_i \tag{3.3}$$

and

$$p = \frac{c^2}{-v_i} \left[\sum_{i=1}^{8} g_i - \frac{3}{2}\frac{\overrightarrow{u}^2}{c^2} \right]. \tag{3.4}$$

This scheme is an artificially incompressible Lattice BGK method due to the "negative rest particle distribution". It also can simulate both steady and unsteady flow problems.

3.2 An incompressibility enhanced scheme for LBGK

3.2.1 The scheme

In a traditional Lattice BGK method, pressure and density are proprotional to each other. Theoretically, in an incompressible flow, pressure disturbances travel at infinite speed. But in Lattice BGK method, pressure disturbances travel at the grid speed c. To improve the incompressibility, it is reasonable to increase the speed at which pressure disturbances travel. Practically, an incompressible flow is a flow in which the density fluctuation $\delta\rho$ will dissipate in a very short time, or more precisely, the system will reach a constant density in a very short time.

To achive a faster "dissipation", we implement an extra "propagation" after every propagation step. Each node should "push out" or "pull in" the amount that equals to the density fluctuation. The new distribution function then reads

$$\widehat{f}_i(\vec{x}+e_i,t) = f_i(\vec{x}+e_i,t) + \frac{\rho-\rho_0}{\rho} f_i(\vec{x},t), \tag{3.5}$$

where $\rho = \sum f_i$, and is ρ_0 the "supposed" constant density. In practice, it can be the constant initial density. This is a re-distribution of mass that is due to the density fluctuation of the Lattice BGK method. It would be natural to re-distribute the extra mass to the neighbors. Since the re-distributed amount is proportional to f_i, the velocity is kept the same.

3.2.2 An example

We give an example using the new incompressible Lattice BGK method. Consider a closed channel with a density fluctuation that is initially set as 1.05 at the left side, and gradually decreases to 0.95 at the right side. We compare the results with a standard Lattice BGK method. We were not supprised to see that our new proposed incompressible Lattice BGK method reached (almost) constant density much faster than the standard Lattice BGK method due to the extra propagation.

3.3 Incompressibility with moving boundary

An interesting example is obtained if we combine the mass conserving moving boundary with the incompressible Lattice BGK method. As a matter of fact, it makes more sense now, since the continuity equation

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \overrightarrow{u}) = 0$$

is satisfied. The incompressibility

$$\frac{D\rho}{Dt} = \frac{\partial \rho}{\partial t} + \nabla \rho \cdot \overrightarrow{u} = 0$$

leads to

$$\nabla \cdot \overrightarrow{u} = 0.$$

We consider an example of a hydraulic rotation damper. A damper is a device that restrains or depresses motion. It transfers the energy of the motion to kinetic energy (heat) due to friction so that a sudden strong motion that might be dangerous to a

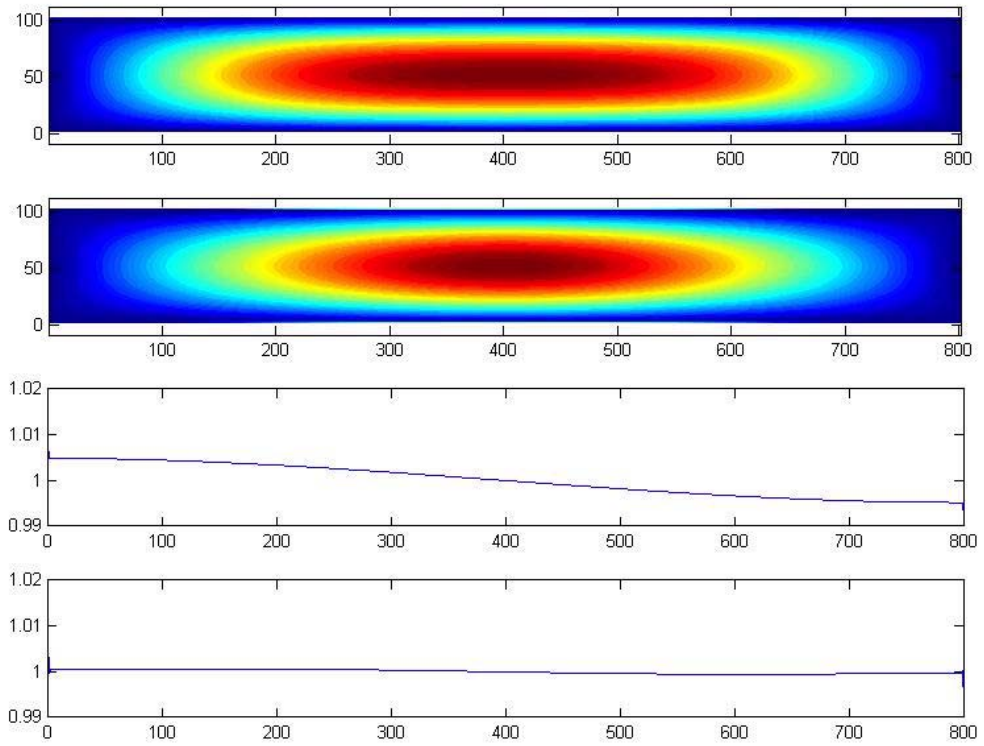

Figure 3.1: A test using the incompressible Lattice BGK method. From top to bottom: 1. velocity contour with normal Lattice BGK; 2. velocity contour with incompressible Lattice BGK; 3. density plot with normal Lattice BGK; 4. density plot with incompressible Lattice BGK.

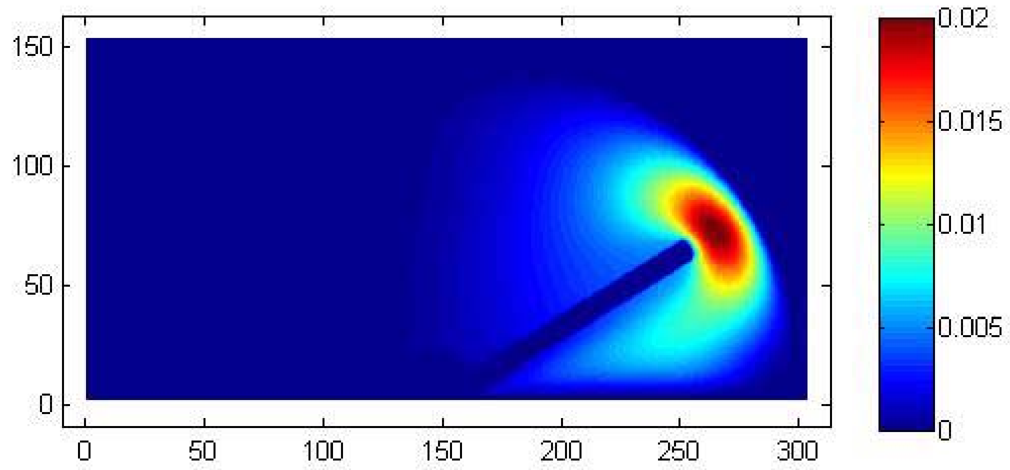

Figure 3.2: The velocity contour of the damper with the standard Lattice BGK method.

system is prohibitted. In this example, we consider a half cylinder system which holds a hydraulic fluid. A devider is attached to the center of the half cylinder and allowed to rotate. Between the divider and the cylinder wall is a gap that allows the fluid to go from one side of the divider into the other. Because of the incompressibility of the fluid, the rotation of the divider will force the fluid to go throught the small gap, which create a counter force, thus damps the rotation.

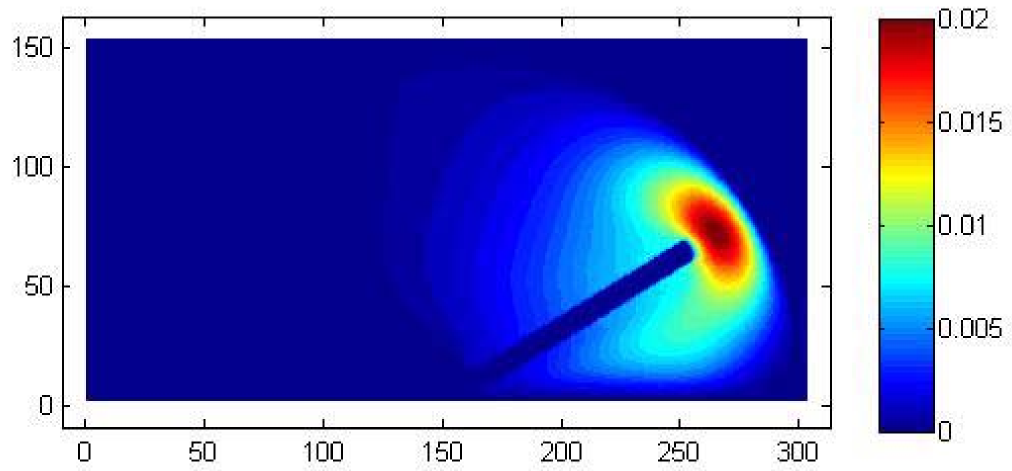

Figure 3.3: The velocity contour of the damper with the incompressible Lattice BGK method.

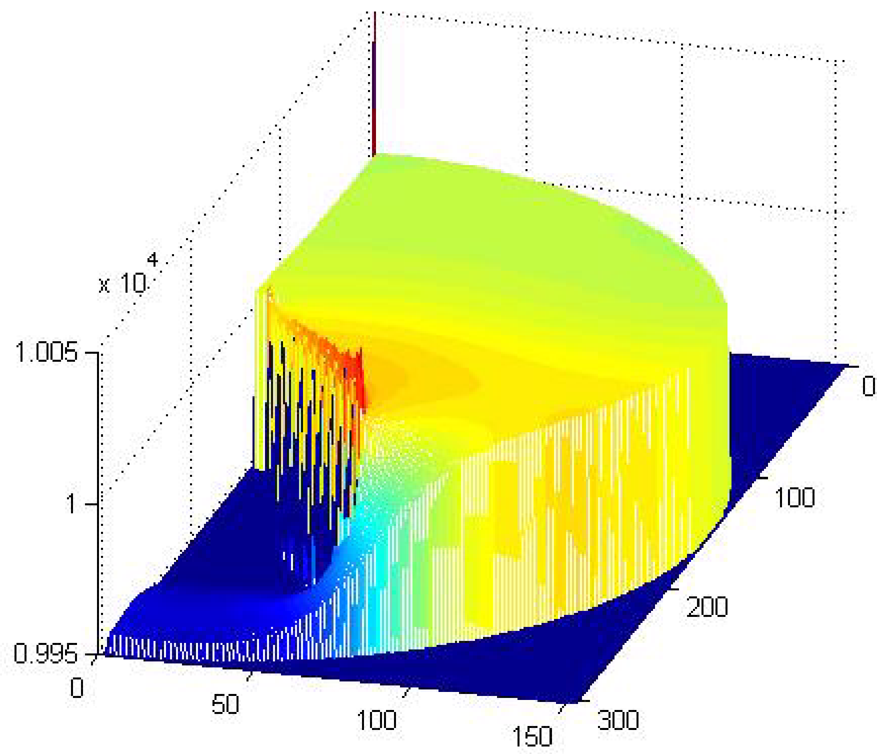

Figure 3.4: The density of the fluid in the damper with the standard Lattice BGK method.

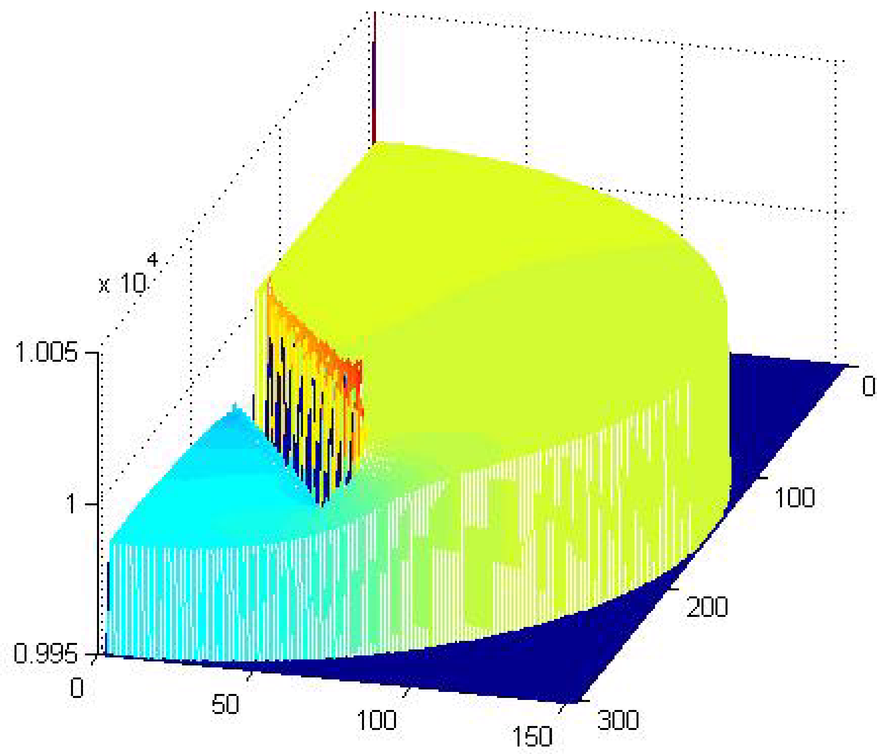

Figure 3.5: The density of the fluid in the damper with the incompressible Lattice BGK method.

3.4 Conclusion

In this chapter we present an incompressibility enhancement for the Lattice Boltzmann method. Since the pressure travel at an infinite speed, we perform an extra propagation of the density disturbances. The result is that, it gives about the same velocity solution as the normal Lattice Boltzmann method. But in some extream situations where the density disturbances are too strong, the incompressibility enhancement will keep the particle distribution in a "safe" mode so that the Lattice Boltzmann method can continue without encounter any problem.

Chapter 4

MHD With Constant B

4.1 Backgound

We present an application of the Lattice Boltzmann method to Magnetohydrodynamics (MHD). MHD is the theory of macroscopic interaction of electrically conducting fluids and electromagnetic fields. Examples of such fluids include plasmas, liquid metals, and salt water. The ideal MHD is that, in a moving conducting fluid, the magnetic field will induce current, the interaction of the current and magnetic field creates a force in this field and alters the velocity of the fluid, and also changes the magnetic field itself. Assuming that the fluid is an incompressible viscous fluid, the governing equation of a MHD system are the Navier-Stokes equation

$$\vec{u}_t + (\vec{u} \cdot \nabla)\vec{u} = -\nabla P + \nu \nabla^2 \vec{u} + \overrightarrow{J} \times \overrightarrow{B} + \overrightarrow{F}, \tag{4.1}$$

and the Ohm's law

$$\overrightarrow{J} = \sigma(-\nabla\phi + \overrightarrow{E} + \overrightarrow{u} \times \overrightarrow{B}), \tag{4.2}$$

together with the continuity equations

$$\nabla \cdot \overrightarrow{u} = 0 \quad \text{and} \quad \nabla \cdot \overrightarrow{J} = 0, \tag{4.3}$$

where $\vec{u}$ is the velocity, $\overrightarrow{J}$ is the current, $\overrightarrow{B}$ is the magnetic field, $\overrightarrow{E}$ is the electric field, P is the pressure, ϕ is the electric potential, ν is the kinematic viscosity, and σ is the fluid conductivity.

We implement a LBM for the fluid. To solve Ohm's law (4.2), we take the divergence of both sides. Since the divergence of $\overrightarrow{J}$ and $\overrightarrow{E}$ are equal to zero (assuming there is no net electric charge in the field), we get a poisson's equation

$$\triangle\phi = \nabla \cdot (\overrightarrow{u} \times \overrightarrow{B}).$$

In one time step, we solve the poisson's equation for ϕ with a boundary condition that yields a well-posed problem, then use ϕ to update $\overrightarrow{J}$, then update $\overrightarrow{u}$ using LBM. For simplicity, we neglect the induced magnetic field. So in this case, $\overrightarrow{B}$ is a constant external magnetic field.

4.2 Implementations

4.2.1 Example I

In this example, we consider a cubic box which contains electrically conducting incompressible fluid. The boundary conditions of ϕ are shown in Figure 4.1. The little arrows indicate dirichlet boundary, and neumann boundary elsewhere. The front side of the dirichlet boundary is set to 1, and the back side is set to 0. The magnetic field $\overrightarrow{B}$ is set to 1 all over the cubic box, the direction is shown in the figure. The external electric field $\overrightarrow{E}$ is set to 0. We used D3Q19 for the fluid.

Figure 4.2 shows the result.

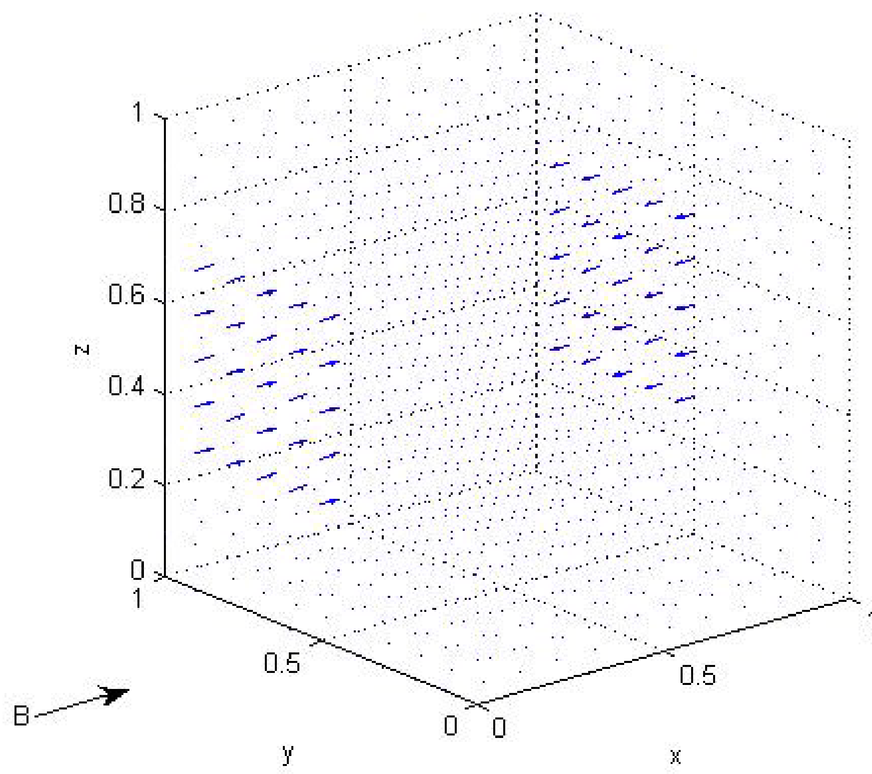

Figure 4.1: The boundary conditions of ϕ.

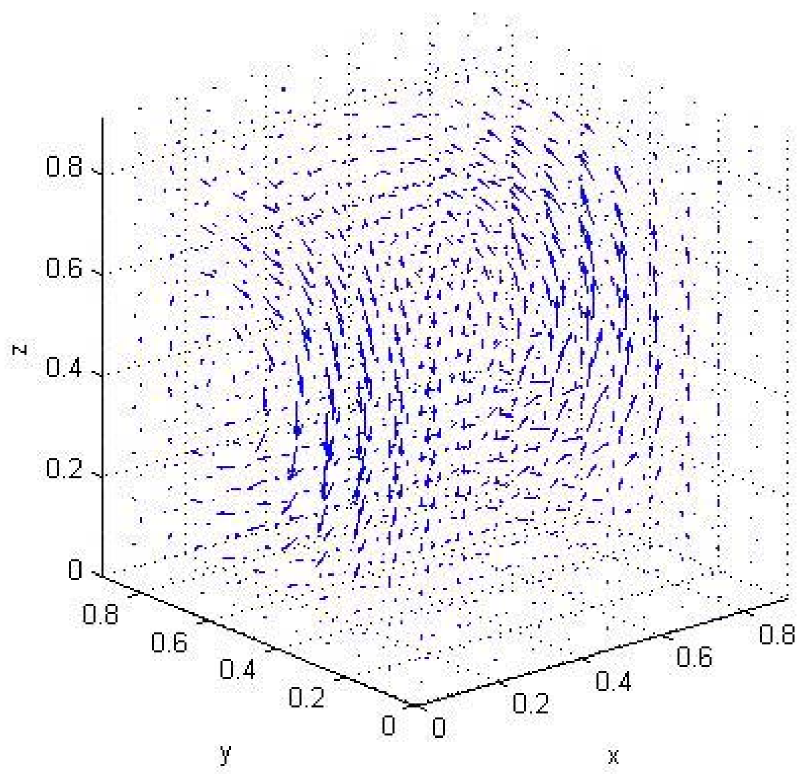

Figure 4.2: The result of the cubic box MHD test.

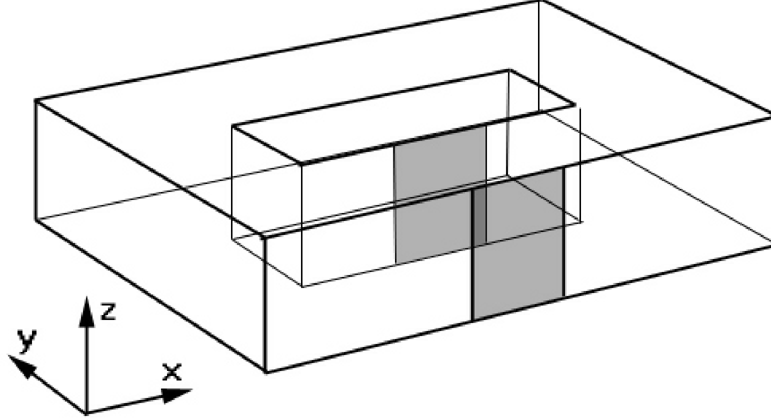

Figure 4.3: The example of a magnetic pump.

4.2.2 Example II

This is a magnetic pump example. In this example, we consider a loop which contains electrically conducting incompressible fluid. The boundary conditions of the electric potential ϕ are shown in Figure 4.3. The boundary of ϕ at the front shaded area is set to 0, and at the back shaded area is set to 1. Other boundaries are all Neumann boundaries. The magnetic field $\overrightarrow{B}$ is set to 1 on z direction and 0 on x and y directions. The external electric field $\overrightarrow{E}$ is set to 0. The electic conductivity σ of the fluid is set to 1. We used D3Q19 for the fluid. Figure 4.4 gives the result at $t = 500$.

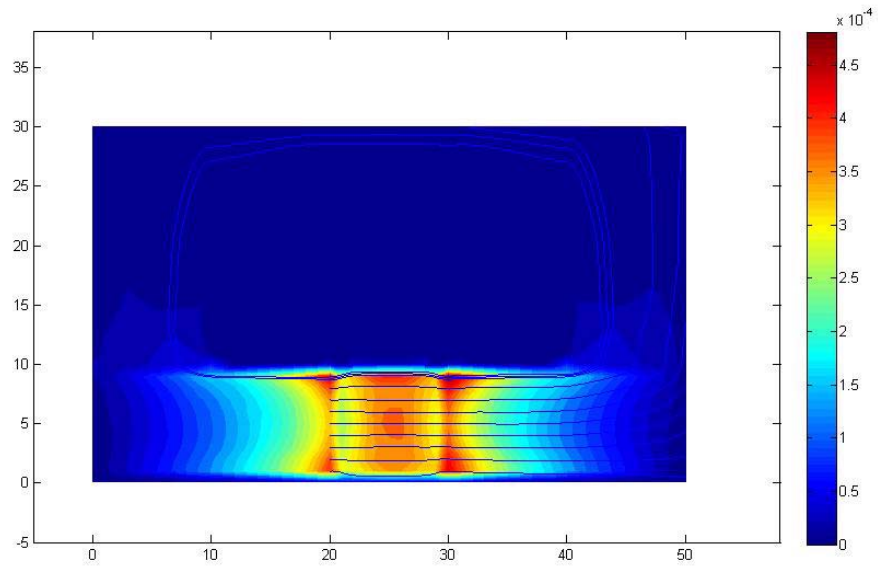

Figure 4.4: The velocity contour of the magnetic pump example.

4.3 Conclusion

In this chapter we coupled the Lattice Boltzmann method with the Ohm's law. We presented two examples. They showed that the 3-dimension D3Q19 is a very fast solver for the Navier-Stokes equation.

Chapter 5

Conclusion

In the last chapter, we give an overall conclusion of the Lattice Boltzmann method and the applications.

The Lattice Boltzmann is a fast solver of the Navier-Stokes equation. Several different discretizations are available. Take the 2-dimension methods as an example: the D2Q7 is the fastest 2-dimension method due to the least number of discrete speed, while the D2Q9 is a little slower but gives a little more accurate result. So one can pick the right method to keep the best balance between performance and accuracy.

The biggest advantage of the Lattice Boltzmann method over other methods is that it is very easy to handle complicated no-slip boundaries. This is due to the fact that to implement the bounce back boundary condition, one doesn't need to know the angle of incidence. So on programming using the Lattice Boltzmann method, one just need to specify the fluid nodes and the solid nodes.

By using the moving boundary treatments we presented in Chapter 2, the Lattice Boltzmann method can be used on even more complicated situations without compromising the performance. This is due to the fact that Lattice Boltzmann method uses a fixed grid and an immersed moving boundary. We presented two moving boundary treatments in Chapter 2: the second order accurate moving boundary, and the first order mass-conserved moving boundary. As we discussed in the conclusion section of Chapter 2, one future work of the mass conserved moving boundary is to consider the shear stress.

The Lattice Boltzmann method is also easy to couple with other method to simulate problems that are related to fluid. We gave examples of MHD flows in Chapter 4.

Overall, the Lattice Boltzmann method can be widely used to different types of problems, and it can do very well.

Bibliography

[1] Dieter A. Wolf-Gladrow, *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*, An Introduction, Springer-Verlag Berlin Heidelberg 2000. ISSN 0075-8434 ISBN 3-540-66973-6

[2] J. Hardy, Y. Pomeau & O. de Pazzis, *Time evolution of two-dimensional model system.* I. Invariant states and time correlation functions, J. Math. Phys. 14 (1973), pp. 1746-1759.

[3] U. Frisch, B. Hasslacher, and Y. Pomeau. *Lattice-gas automata for the Navier-Stokes equation.* Physical Review Letters, 56:1505-1508, 1986.

[4] G. R. McNamara and G. Zanetti. *Use of the Boltzmann equation to simulate lattice-gas automata.* Physical Review Letters, 61:2332-2335, 1988.

[5] F. J. Higuera and J. Liménez. *Boltzmann approach to lattice gas simulations.* Europhysics Letters, 9 (7):663-668, 1989.

[6] F. J. Higuera, S. Succi, and R. Benzi. *Lattice gas dynamics with enhanced collisions.* Europhysics Letters, 9 (4):345-349, 1989.

[7] P. L. Bhatnagar, E. P. Gross, and M Krook. *A model for collision processes in gases.* I. Small amplitude processes in charged and neutral one-component systems. Physical Review, 94 (3):511-525, 1954.

[8] J. Buick, W. Easson, and C. Greated. *Simulation of wave motion using a lattice gas model.* International Journal for Numberical Methods in Fluids, 22:313-321, 1996.

[9] Sauro Succi, *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*, Clarendon Press, Oxford, 2001. ISBN 0-19-850398-9

[10] D. R. Noble, S. Chen, J. G. Georgiadis, and R. O. Buckius. *A consistent hydrodynamic boundary condition for the lattice Boltzmann method.* Physics of Fluids, 7 (1):203-209, 1995.

[11] S. Wolfram. *Cellular automaton fluids* 1: Basic theory. Journal of Statistical Physics, 45(3/4): p471-529, 1986.

[12] Ghia, Ghia, and Shin, *High Resolutions for incompressible flow using the navier-stokes equations and a multigrid method*, Journal of Computational Physics, Vol. 48, p387-411, 1982.

[13] Shuling Hou and Qisu Zou, Shiyi Chen, Gary Doolen, Allen C. Cogley. *Simulation of Cavity Flow by the Lattice Boltzmann Method*. Journal of Computational Physics 118, p329-347 1995.

[14] Pierre Lallemand, Li-Shi Luo. *Lattice Boltzmann method for moving boundaries.* Journal of Computational Physics 184 (2003) 406-421

[15] Paul J. Dellar, *Incompressible limits of lattice Boltzmann equations using multiple relaxation times*, Journal of Computational Physics 190 (2003) 351-370

[16] Xiao He and Li-Shi Luo, *Lattice Boltzmann Model for the Incompressible Navier-Stokes Equation*, Journal of Statistical Physics, Vol.88, Nos. 3/4, 1997

[17] Zhaoli Guo, Baochang Shi, and Nengchao Wang, *Lattice BGK Model for Incompressible Navier-Stokes Equation*, Journal of Computational Physics 165, 288-306 (2000)

Appendices

Appendix A

FHP collision look-up table

This is the collision look-up table. It gives the two out-states corresponding to each of the 128 in-states (right in bracket).

0	[0 0]	16	[16 16]	32	[32 32]	48	[48 48]
1	[1 1]	17	[96 96]	33	[33 33]	49	[49 49]
2	[2 2]	18	[9 36]	34	[65 65]	50	[41 81]
3	[3 3]	19	[98 37]	35	[35 35]	51	[51 51]
4	[4 4]	20	[72 72]	36	[18 9]	52	[104 25]
5	[66 66]	21	[42 42]	37	[19 98]	53	[105 114]
6	[6 6]	22	[13 74]	38	[69 11]	54	[27 45]
7	[7 7]	23	[102 75]	39	[39 39]	55	[107 107]
8	[8 8]	24	[24 24]	40	[80 80]	56	[56 56]
9	[36 18]	25	[52 104]	41	[81 50]	57	[57 57]
10	[68 68]	26	[84 44]	42	[21 21]	58	[116 89]
11	[38 69]	27	[45 54]	43	[83 101]	59	[117 117]
12	[12 12]	28	[28 28]	44	[26 84]	60	[60 60]
13	[74 22]	29	[90 108]	45	[54 27]	61	[122 122]
14	[14 14]	30	[30 30]	46	[77 86]	62	[93 93]
15	[15 15]	31	[110 110]	47	[87 87]	63	[63 63]

64	[64 64]	80	[40 40]	96	[17 17]	112	[112 112]
65	[34 34]	81	[50 41]	97	[97 97]	113	[113 113]
66	[5 5]	82	[73 100]	98	[37 19]	114	[53 105]
67	[67 67]	83	[101 43]	99	[99 99]	115	[115 115]
68	[10 10]	84	[44 26]	100	[82 73]	116	[89 58]
69	[11 38]	85	[106 106]	101	[43 83]	117	[59 59]
70	[70 70]	86	[46 77]	102	[75 23]	118	[91 109]
71	[71 71]	87	[47 47]	103	[103 103]	119	[119 119]
72	[20 20]	88	[88 88]	104	[25 52]	120	[120 120]
73	[100 82]	89	[58 116]	105	[114 53]	121	[121 121]
74	[22 13]	90	[108 29]	106	[85 85]	122	[61 61]
75	[23 102]	91	[109 118]	107	[55 55]	123	[123 123]
76	[76 76]	92	[92 92]	108	[29 90]	124	[124 124]
77	[86 46]	93	[62 62]	109	[118 91]	125	[125 125]
78	[78 78]	94	[94 94]	110	[31 31]	126	[126 126]
79	[79 79]	95	[95 95]	111	[111 111]	127	[127 127]

Appendix B

Partial Matlab code I: d2q9_rotating_polygon.m

```
%
% Test of a D2Q9 Lattice BGK model on a square area w/ a rotating polygon.
% An implementation of the moving boundary with mass conservation.
% An implementation of the incompressible enhancement.
%

%% Define the general global variables

global cont_mode;
    cont_mode = 0
global dimension;  %  The square domain
    dimension = [1 1];
global n_poly;  % Number of the sides of the polygon
    n_poly = 3;
global center;
    center = [.5 .5];  % Center of the rotating polygon
global N;  % Number of fineness
    N = 100;
global diameter;  % The diameter of the rotating object
    diameter = 1/N + 0.9;
global D;  % The hydraulic diameter
    if n_poly == 3
        H = N;
```

```
        S = N*sqrt(3)/2*diameter;
        D = 2*(H^2-sqrt(3)/4*S^2)/(4*H+3*S)
    elseif n_poly == 4
        H = N;
        L = H/sqrt(2);
        D = 2*(H^2-L^2)/(4*H+4*L)
    end
global x y;  % Define the domain length, width (including the ghost layer)
    x = dimension(1)*N + 3;
    y = dimension(2)*N + 3;
global speed;  % Rotation angle speed
    T = 3000;
    speed = pi/T;
global r;  % Density
    r = 1;
global Re;  % Reynolds number
    Re = 31;
global omega;
    viscosity = r*D^2*speed/Re
    omega = 1/(viscosity*3+1/2)
global F; % Distribution function matrix
    F = zeros(x*y, 9);
global U; % Velocity field
    U = zeros(x*y, 2);
% global SequenceU;
%     SequenceU = zeros(x*y,2,500);
global Rho; % Density
```

```
    Rho = zeros(x*y, 1);
global Fc; % Distribution function after the collision
    Fc = F;
global c;  % c is the unit speed
    c = 1;
global e;
    e = [0 0];  % i = 1
    e = [e; 1 0; 0 1; -1 0; 0 -1];  % i = 2,3,4,5
    e = [e; 1 1; -1 1; -1 -1; 1 -1];  % i = 6,7,8,9

%       7     3     6
%         \   |   /
%           \ | /
%       4 --- 1 ----2
%           / | \
%         /   |   \
%       8     5     9

global w;
    w = [4/9 1/9 1/9 1/9 1/9 1/36 1/36 1/36 1/36];
global dx;
    dx = 1;   % the increment on positive x direction
global dy;
    dy = x;   % the increment on positive y direction
global opposite;
    opposite = [1 4 5 2 3 8 9 6 7];
global q2 q3 q4 q5 q6 q7 q8 q9;
```

```
    q2 = [];
    q3 = [];
    q4 = [];
    q5 = [];
    q6 = [];
    q7 = [];
    q8 = [];
    q9 = [];
global i_polygon_update2 i_polygon_update3 i_polygon_update4 i_polygon_update5
i_polygon_update6 i_polygon_update7 i_polygon_update8 i_polygon_update9;
    i_polygon_update2 = [];
    i_polygon_update3 = [];
    i_polygon_update4 = [];
    i_polygon_update5 = [];
    i_polygon_update6 = [];
    i_polygon_update7 = [];
    i_polygon_update8 = [];
    i_polygon_update9 = [];
global A B C;
    A = 0; B = 0; C = 0;

%% Numbering order: x -> y

% An example of this code with polygon = 3
% Starting from (1, 1) which is the lower left corner
%
%                | y                |----------|
```

```
%               |                |    /\    |
%               |                |   /  \   |   <- rotating triangle
%                ------- x       |  /____\  |     (counter-clockwise rotation)
%                            (x+1)|          |
%                              (1)|----------|(x)
%
%   ----------------------------------------------------------------------
%  | The index should start with i_                                       |
%  | Those without the i_ are binary index that are for logical operation |
%  |______________________________________________________________________|
%
%                            actual boundary
%                                  |
%                               boundary   ghost
%                                 node     node
%                                  |         |
%       x         x        x         x         x
%                          |         |
%   ------- inside ------->|
%
%   ---------- domain ------------->|
%
%
% Since we are using on-the node bounce-back boundary, the inside nodes
% number will be (N - 1)
% center of each space.
%
```

```
%       |------|------|------|------|---....---|------|------|------|
%     ghost    0                                       1    ghost
%
% nodes showed 1      2      3      4         N-1     N     N+1
%
%              |<----------------  domain  ----------------->|
%                     |<---------  inside  ---------->|
%
%



%% Define the domain


global domain;
    domain = zeros(x*y,1);


% Each node will be assign a number:
%       0 for ghost_boundary
%       1 for boundary
%       2 for inside
%
% Domain = boundary + inside
%


for j = 2:y-1
```

```
    for i = 2:x-1
        domain((j-1)*x+i) = 1;
    end
end
for j = 3:y-2
    for i = 3:x-2
        domain((j-1)*x+i) = 2;
    end
end

%% Define the rotating polygon

global radius;  % The radius of the inscribed circle
    radius = diameter/2*sin( (pi - 2*pi/n_poly)/2 );
global theta;  %  Initial polygon position
    theta = pi*3/2;

% Define the sides of the polygon by the normal vector
global v_poly;
    v_poly = zeros(n_poly, 2);
for i = 1:n_poly
    v_poly(i,:) = [cos(theta+2*pi/n_poly*(i-1)) sin(theta+2*pi/n_poly*(i-1))];
end

global polygon;
    polygon = zeros(x*y, 1);
for i = 2:x-1
```

```
    for j = 2:y-1
        v = [((i-2)/N - center(1)) ((j-2)/N - center(2))];
        if v_poly*v'<radius
            polygon((j-1)*x+i) = 1;
        end
    end
end

% i_polygon = find(polygon==1);

global i_domain;
    i_domain = find( ( domain-polygon.*2 ) > 0);
global i_boundary;
    i_boundary = find( ( domain-polygon.*2 ) == 1 );

global v_poly_update;
    v_poly_update = zeros(n_poly, 2);
for i = 1:n_poly
    v_poly_update(i,:) = [cos(theta+2*pi/n_poly*(i-1)) sin(theta+2*pi/n_poly
*(i-1))];
end

polygon_update = zeros(x*y,1);

for i = 2:x-1
    for j = 2:y-1
        v = [((i-2)/N - center(1)) ((j-2)/N - center(2))];
```

```
        if v_poly_update*v'<radius
            polygon_update((j-1)*x+i) = 1;
        end
    end
end

v_poly = v_poly_update;

i_non_domain = find( ( domain-polygon_update.*2 ) <= 0);

polygon_update2 = polygon_update;
polygon_update3 = polygon_update;
polygon_update4 = polygon_update;
polygon_update5 = polygon_update;
polygon_update6 = polygon_update;
polygon_update7 = polygon_update;
polygon_update8 = polygon_update;
polygon_update9 = polygon_update;

polygon_update2(1+dx:end) = polygon_update(1:end-dx);
polygon_update3(1+dy:end) = polygon_update(1:end-dy);
polygon_update4(1:end-dx) = polygon_update(1+dx:end);
polygon_update5(1:end-dy) = polygon_update(1+dy:end);
polygon_update6(1+dx+dy:end) = polygon_update(1:end-dx-dy);
polygon_update7(1+dy:end-dx) = polygon_update(1+dx:end-dy);
polygon_update8(1:end-dx-dy) = polygon_update(1+dx+dy:end);
polygon_update9(1+dx:end-dy) = polygon_update(1+dy:end-dx);
```

```
polygon_update2 = xor(polygon_update, polygon_update2) & ( domain-polygon.*2 );
polygon_update3 = xor(polygon_update, polygon_update3) & ( domain-polygon.*2 );
polygon_update4 = xor(polygon_update, polygon_update4) & ( domain-polygon.*2 );
polygon_update5 = xor(polygon_update, polygon_update5) & ( domain-polygon.*2 );
polygon_update6 = xor(polygon_update, polygon_update6) & ( domain-polygon.*2 );
polygon_update7 = xor(polygon_update, polygon_update7) & ( domain-polygon.*2 );
polygon_update8 = xor(polygon_update, polygon_update8) & ( domain-polygon.*2 );
polygon_update9 = xor(polygon_update, polygon_update9) & ( domain-polygon.*2 );

i_polygon_update2_backup = i_polygon_update2;
i_polygon_update3_backup = i_polygon_update3;
i_polygon_update4_backup = i_polygon_update4;
i_polygon_update5_backup = i_polygon_update5;
i_polygon_update6_backup = i_polygon_update6;
i_polygon_update7_backup = i_polygon_update7;
i_polygon_update8_backup = i_polygon_update8;
i_polygon_update9_backup = i_polygon_update9;

i_polygon_update2 = find(polygon_update2==1);
i_polygon_update3 = find(polygon_update3==1);
i_polygon_update4 = find(polygon_update4==1);
i_polygon_update5 = find(polygon_update5==1);
i_polygon_update6 = find(polygon_update6==1);
i_polygon_update7 = find(polygon_update7==1);
i_polygon_update8 = find(polygon_update8==1);
i_polygon_update9 = find(polygon_update9==1);
```

```
q2_backup = q2;
q3_backup = q3;
q4_backup = q4;
q5_backup = q5;
q6_backup = q6;
q7_backup = q7;
q8_backup = q8;
q9_backup = q9;

q2 = []; % q2 is the boundary length from the polygon to the nearest node on
direction 2
q3 = [];
q4 = [];
q5 = [];
q6 = [];
q7 = [];
q8 = [];
q9 = [];
q2 = q_rotating_polygon(i_polygon_update2, 2);
q3 = q_rotating_polygon(i_polygon_update3, 3);
q4 = q_rotating_polygon(i_polygon_update4, 4);
q5 = q_rotating_polygon(i_polygon_update5, 5);
q6 = q_rotating_polygon(i_polygon_update6, 6);
q7 = q_rotating_polygon(i_polygon_update7, 7);
q8 = q_rotating_polygon(i_polygon_update8, 8);
```

```
q9 = q_rotating_polygon(i_polygon_update9, 9);

%% Starting initialization

tic
    Rho(i_domain,:) = 1;

    for i = 1:9
        F(i_domain,i) = Rho(i_domain,:)*w(i).*(    1 + 3/c^2*U(i_domain,:)*e(i,:)'
+ 9/(2*c^4)*(U(i_domain,:)*e(i,:)').^2  - 3/(2*c^2)*(U(i_domain,1).^2
+ U(i_domain,2).^2)   );
    end

    disp('Initialization completed!')
toc

if cont_mode == 1
    load D:\LBM\d2q9\data\rotating_triangle\N100d.9speed3000Re31@50cycles.mat
end

%%

tic
```

```
if cont_mode == 1
    t = k;
    for k = t+1:t+6000*50
        theta = theta + speed;
        d2q9_rotating_polygon_collision;
        k
        if mod(k, 1) == 0
            U_contour_wide(F)
            sum(Rho)/numel(i_domain)
%          vis_F(F)
        end
    end


else
    for k = 1:6000*50
        theta = theta + speed;
        d2q9_rotating_polygon_collision;
        k
        if mod(k, 3000) == 0
            U_contour_wide(F)
            vis_F(F)
            sum(Rho)/numel(i_domain)
%          vis_F(F)
        end
    end
end
time = toc
```

Appendix C

Partial matlab code II: d2q9_rotating_polygon_collision.m

```
function d2q9_rotating_polygon_collision

%% Define the general global variables

global N;
global n_poly;
global x y;
global speed;
global omega;
global F;
global U;
global Fc;
global c;
global e;
global w;
global v_poly;
global i_boundary;
global inside;
global center;
global domain;
global i_domain;
global Rho;
```

```
global dx;
global dy;
global polygon;
global opposite;
global q2 q3 q4 q5 q6 q7 q8 q9;
global i_polygon_update2 i_polygon_update3 i_polygon_update4 i_polygon_update5
 i_polygon_update6 i_polygon_update7 i_polygon_update8 i_polygon_update9;
global A B C;

%% Define the rotating polygon



global radius;
global theta;

% Define the sides of the polygon by the normal vector

global v_poly_update;
    v_poly_update = zeros(n_poly, 2);
for i = 1:n_poly
    v_poly_update(i,:) = [cos(theta+2*pi/n_poly*(i-1)) sin(theta+2*pi/n_poly
*(i-1))];
end

polygon_update = zeros(x*y,1);

for i = 2:x-1
```

```
    for j = 2:y-1
        v = [((i-2)/N - center(1)) ((j-2)/N - center(2))];
        if v_poly_update*v'<radius
            polygon_update((j-1)*x+i) = 1;
        end
    end
end

v_poly = v_poly_update;
i_domain = find( ( domain-polygon_update.*2 ) > 0);
i_non_domain = find( ( domain-polygon_update.*2 ) <= 0);

polygon_update2 = polygon_update;
polygon_update3 = polygon_update;
polygon_update4 = polygon_update;
polygon_update5 = polygon_update;
polygon_update6 = polygon_update;
polygon_update7 = polygon_update;
polygon_update8 = polygon_update;
polygon_update9 = polygon_update;

polygon_update2(1+dx:end) = polygon_update(1:end-dx);
polygon_update3(1+dy:end) = polygon_update(1:end-dy);
polygon_update4(1:end-dx) = polygon_update(1+dx:end);
polygon_update5(1:end-dy) = polygon_update(1+dy:end);
polygon_update6(1+dx+dy:end) = polygon_update(1:end-dx-dy);
polygon_update7(1+dy:end-dx) = polygon_update(1+dx:end-dy);
```

```
polygon_update8(1:end-dx-dy) = polygon_update(1+dx+dy:end);
polygon_update9(1+dx:end-dy) = polygon_update(1+dy:end-dx);

polygon_update2 = xor(polygon_update, polygon_update2)
& ( domain-polygon_update.*2 );
polygon_update3 = xor(polygon_update, polygon_update3)
& ( domain-polygon_update.*2 );
polygon_update4 = xor(polygon_update, polygon_update4)
& ( domain-polygon_update.*2 );
polygon_update5 = xor(polygon_update, polygon_update5)
& ( domain-polygon_update.*2 );
polygon_update6 = xor(polygon_update, polygon_update6)
& ( domain-polygon_update.*2 );
polygon_update7 = xor(polygon_update, polygon_update7)
& ( domain-polygon_update.*2 );
polygon_update8 = xor(polygon_update, polygon_update8)
& ( domain-polygon_update.*2 );
polygon_update9 = xor(polygon_update, polygon_update9)
& ( domain-polygon_update.*2 );

i_polygon_update2_backup = i_polygon_update2;
i_polygon_update3_backup = i_polygon_update3;
i_polygon_update4_backup = i_polygon_update4;
i_polygon_update5_backup = i_polygon_update5;
i_polygon_update6_backup = i_polygon_update6;
i_polygon_update7_backup = i_polygon_update7;
i_polygon_update8_backup = i_polygon_update8;
```

```
i_polygon_update9_backup = i_polygon_update9;

i_polygon_update2 = find(polygon_update2==1);
i_polygon_update3 = find(polygon_update3==1);
i_polygon_update4 = find(polygon_update4==1);
i_polygon_update5 = find(polygon_update5==1);
i_polygon_update6 = find(polygon_update6==1);
i_polygon_update7 = find(polygon_update7==1);
i_polygon_update8 = find(polygon_update8==1);
i_polygon_update9 = find(polygon_update9==1);


q2_backup = q2;
q3_backup = q3;
q4_backup = q4;
q5_backup = q5;
q6_backup = q6;
q7_backup = q7;
q8_backup = q8;
q9_backup = q9;

q2 = []; % q2 is the boundary length from the polygon to the nearest node
on direction 2
q3 = [];
q4 = [];
q5 = [];
q6 = [];
```

```
q7 = [];
q8 = [];
q9 = [];
q2 = q_rotating_polygon(i_polygon_update2, 2);
q3 = q_rotating_polygon(i_polygon_update3, 3);
q4 = q_rotating_polygon(i_polygon_update4, 4);
q5 = q_rotating_polygon(i_polygon_x_update5, 5);
q6 = q_rotating_polygon(i_polygon_update6, 6);
q7 = q_rotating_polygon(i_polygon_update7, 7);
q8 = q_rotating_polygon(i_polygon_update8, 8);
q9 = q_rotating_polygon(i_polygon_update9, 9);



%% Find and fill in newly released zone

Fc = F;

released = xor((polygon | polygon_update), polygon_update);
i_released = find(released==1);


count = numel(i_released);
for k = 1:count
    coordi = [mod(i_released(k),x) floor(i_released(k)/x)+1];
    v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
```

```
    u = [-v(2) v(1)];
    u = u*speed*norm(v)/norm(u);
    temp = [0 0 0 0 0 0 0 0 0];


    for i = 2:9
        temp1 = find(i_domain == i_released(k)+e(i,:)*[dx;dy]);
% Check if the previous node is in domain
        temp2 = find(i_released == i_released(k)+e(i,:)*[dx;dy] );
% Check if the previous node is also a released node
        temp3 = find(i_domain == i_released(k)-e(i,:)*[dx;dy] );
% Check if the next node is in domain
        temp4 = eval( ['find(i_polygon_update' num2str(i) ' == i_released(k))'] );
% Check if q value is available

        if temp1 & isempty(temp2) & isempty(temp3) & temp4

            temp(1) = temp(1) + 1;
            temp(i) = 1;

            % For F
            F(i_released(k), i) = Fc(i_released(k)+e(i,:)*[dx;dy], i);
            F(i_released(k), opposite(i)) = Fc(i_released(k)+e(i,:)*[dx;dy],
opposite(i));

        end
```

```
        if temp1 & temp3

            % For F
            F(i_released(k), i) = ( Fc(i_released(k)+e(i,:)*[dx;dy], i)
+Fc(i_released(k)-e(i,:)*[dx;dy], i) )/2;
            F(i_released(k), opposite(i)) = ( Fc(i_released(k)+e(i,:)*[dx;dy],
opposite(i))
+Fc(i_released(k)-e(i,:)*[dx;dy], opposite(i)) )/2;

        end
    end

    % Fill in the rest particle
    for i = 2:9
        if temp(i) == 1

            % For F
            F(i_released(k), 1) = F(i_released(k), 1) + Fc(i_released(k)+e(i,:)
*[dx;dy], 1);

        end
    end

    % For F
    if temp(1)~=0
        F(i_released(k), 1) = F(i_released(k), 1)/temp(1);
```

```
    elseif temp(1)==0
        for i = 2:9
            F(i_released(k), 1) = Fc(i_released(k), 1) + Fc(i_released(k)+e(i,:)
*[dx;dy], 1);
        end
        F(i_released(k), 1) = F(i_released(k), 1)/8;
    end

end


%% Find and clear newly covered zone

covered = xor((polygon | polygon_update), polygon) & domain;
i_covered = find(covered==1);



count = numel(i_covered);
for k = 1:count
    coordi = [mod(i_covered(k),x) floor(i_covered(k)/x)+1];
    v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
    u = [-v(2) v(1)];
    u = u*speed*norm(v)/norm(u);
    temp = [0 0 0 0 0 0 0 0 0];
    for i = 2:9
        temp1 = find(i_domain == i_covered(k)+e(i,:)*[dx;dy] );
% Check if the next node is in domain
```

```
        temp2 = find(i_covered == i_covered(k)+e(i,:)*[dx;dy] );
% Check if the next node is also a covered node
        temp4 = eval( ['find(i_polygon_update' num2str(i) '_backup
== i_covered(k))'] );
% Check if q value is available

        if temp1 & isempty(temp2) & temp4
            temp(i) = 1;
        end
    end
    for i = 2:9

        if temp(i) == 1

            F(i_covered(k)+e(i,:)*[dx;dy], i) = Fc(i_covered(k)+e(i,:)*[dx;dy], i)
+ Fc(i_covered(k), i) * ( eval(['q' num2str(i) '_backup( find(i_polygon_update'
num2str(i) '_backup == i_covered(k) ) )' ]) + 1/2);
            F(i_covered(k)+e(i,:)*[dx;dy], opposite(i)) = Fc(i_covered(k)
+e(i,:)*[dx;dy], opposite(i)) + Fc(i_covered(k), opposite(i))
* ( eval(['q' num2str(i) '_backup( find(i_polygon_update' num2str(i) '
_backup == i_covered(k) ) )' ]) + 1/2);

            % boundary uncorrection
            F(i_covered(k)+e(i,:)*[dx;dy], i) = F(i_covered(k)+e(i,:)*[dx;dy], i)
/ (3/2 + eval(['q' num2str(i) '_backup( find(i_polygon_update' num2str(i) '
_backup == i_covered(k) ) )' ]) );
            F(i_covered(k)+e(i,:)*[dx;dy], opposite(i)) = F(i_covered(k)
```

```
+e(i,:)*[dx;dy], opposite(i)) / (3/2 + eval(['q' num2str(i) '_backup( find
(i_polygon_update' num2str(i) '_backup == i_covered(k) ) )' ]) );

        end
    end

end

B = B + sum(sum(F))/numel(i_domain);

%% Calculate Rho and U

Rho(:,:) = 0;
for i = 1:9
    Rho(i_domain,:) = Rho(i_domain,:) + F(i_domain,i);
end
U(:,:) = 0;
for i = 1:9
    U(i_domain,:) = U(i_domain,:) + F(i_domain,i)*e(i,:);
end

U(i_domain,1) = U(i_domain,1)./Rho(i_domain);
U(i_domain,2) = U(i_domain,2)./Rho(i_domain);



%% Calculate equilibrium distribution
```

```
for i = 1:9

    Fc(i_domain,i) = Rho(i_domain,:)*w(i).*(    1 + 3/c^2*U(i_domain,:)*e(i,:)'

+  9/(2*c^4)*(U(i_domain,:)*e(i,:)').^2  - 3/(2*c^2)*(U(i_domain,1).^2

+  U(i_domain,2).^2)   );

end



%% Collision


Fc(i_domain,:) = (1-omega)*F(i_domain,:) + omega*Fc(i_domain,:);



%% Streaming (propagation)


% For F

for i = 2:9

    F(i_domain + e(i,:)*[dx; dy], i) = Fc(i_domain, i);

end


F(i_non_domain,:) = 0;



%% Boundary correction on F
```

```
% For F

F(i_polygon_update2_backup, 2) = Fc(i_polygon_update2_backup, 2)
.*(1/2+q2_backup);
F(i_polygon_update2_backup, 4) = Fc(i_polygon_update2_backup, 4)
.*(1/2+q2_backup);
F(i_polygon_update4_backup, 4) = Fc(i_polygon_update4_backup, 4)
.*(1/2+q4_backup);
F(i_polygon_update4_backup, 2) = Fc(i_polygon_update4_backup, 2)
.*(1/2+q4_backup);

F(i_polygon_update3_backup, 3) = Fc(i_polygon_update3_backup, 3)
.*(1/2+q3_backup);
F(i_polygon_update3_backup, 5) = Fc(i_polygon_update3_backup, 5)
.*(1/2+q3_backup);
F(i_polygon_update5_backup, 5) = Fc(i_polygon_update5_backup, 5)
.*(1/2+q5_backup);
F(i_polygon_update5_backup, 3) = Fc(i_polygon_update5_backup, 3)
.*(1/2+q5_backup);


F(i_polygon_update6_backup, 6) = Fc(i_polygon_update6_backup, 6)
.*(1/2+q6_backup);
F(i_polygon_update6_backup, 8) = Fc(i_polygon_update6_backup, 8)
.*(1/2+q6_backup);
F(i_polygon_update8_backup, 8) = Fc(i_polygon_update8_backup, 8)
.*(1/2+q8_backup);
```

```
F(i_polygon_update8_backup, 6) = Fc(i_polygon_update8_backup, 6)
.*(1/2+q8_backup);

F(i_polygon_update7_backup, 7) = Fc(i_polygon_update7_backup, 7)
.*(1/2+q7_backup);
F(i_polygon_update7_backup, 9) = Fc(i_polygon_update7_backup, 9)
.*(1/2+q7_backup);
F(i_polygon_update9_backup, 9) = Fc(i_polygon_update9_backup, 9)
.*(1/2+q9_backup);
F(i_polygon_update9_backup, 7) = Fc(i_polygon_update9_backup, 7)
.*(1/2+q9_backup);


% Correction on newly released zone

count = numel(i_released);
for k = 1:count
    coordi = [mod(i_released(k),x) floor(i_released(k)/x)+1];
    v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
    u = [-v(2) v(1)];
    u = u*speed*norm(v)/norm(u);

    for i = 2:9
        temp1 = find(i_domain == i_released(k)+e(i,:)*[dx;dy]);
% Check if the previous node is in domain
        temp2 = find(i_released == i_released(k)+e(i,:)*[dx;dy] );
% Check if the previous node is also a released node
```

```
        temp3 = find(i_released == i_released(k)-e(i,:)*[dx;dy] );
% Check if the next node is a released node
        temp4 = eval( ['find(i_polygon_update' num2str(i) '
== i_released(k))'] );
% Check if q value is available


        if temp1 & isempty(temp2) & isempty(temp3) & temp4


            % For F
            F(i_released(k), i) = F(i_released(k), i) * ( eval(['q' num2str(i)
'_backup( find(i_polygon_update' num2str(i) '_backup == i_released(k)
+e(i,:)*[dx;dy]) )' ]) - 1/2);
            F(i_released(k), opposite(i)) = F(i_released(k), opposite(i))
* ( eval(['q' num2str(i) '_backup(find(i_polygon_update' num2str(i) '
_backup == i_released(k)+e(i,:)*[dx;dy]) )' ]) - 1/2);


            F(i_released(k)+e(i,:)*[dx;dy], i) = F(i_released(k)+e(i,:)
*[dx;dy], i) / (1/2 + eval(['q' num2str(i) '_backup( find(i_polygon_update
' num2str(i) '_backup == i_released(k)+e(i,:)*[dx;dy] ) )' ]) );
            F(i_released(k)+e(i,:)*[dx;dy], opposite(i)) = F(i_released(k)
+e(i,:)*[dx;dy], opposite(i)) / (1/2 + eval(['q' num2str(i) '_backup(
find(i_polygon_update' num2str(i) '_backup == i_released(k)+e(i,:)
*[dx;dy] ) )' ]) );




        end
    end
```

```
end

% Correction on newly covered zone

count = numel(i_covered);
for k = 1:count
    coordi = [mod(i_covered(k),x) floor(i_covered(k)/x)+1];
    v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
    u = [-v(2) v(1)];
    u = u*speed*norm(v)/norm(u);
    temp = [0 0 0 0 0 0 0 0 0];
    for i = 2:9
        temp1 = find(i_domain == i_covered(k)+e(i,:)*[dx;dy] );
% Check if the next node is in domain
        temp2 = find(i_covered == i_covered(k)+e(i,:)*[dx;dy] );
% Check if the next node is also a covered node
        temp4 = eval( ['find(i_polygon_update' num2str(i) '_backup
== i_covered(k))'] );
% Check if q value is available

        if temp1 & isempty(temp2) & temp4
            temp(i) = 1;
        end
    end
```

```
    for i = 2:9
        if temp(i) == 1

            % For F
            F(i_covered(k)+e(i,:)*[dx;dy], i) = F(i_covered(k)+e(i,:)*[dx;dy], i)
* (3/2 + eval(['q' num2str(i) '_backup( find(i_polygon_update' num2str(i) '
_backup == i_covered(k) ) )' ]) );
            F(i_covered(k)+e(i,:)*[dx;dy], opposite(i)) = F(i_covered(k)+e(i,:)
*[dx;dy], opposite(i)) * (3/2 + eval(['q' num2str(i) '_backup( find
(i_polygon_update' num2str(i) '_backup == i_covered(k) ) )' ]) );

        end
    end

end


F_backup = F;


%% Boundary correction on the polygon for q<1/2

U_speed = U;
U_speed(:,:) = 0;
```

```
count = numel(q2);
for i = 1:count
    if q2(i)<1/2
        coordi = [mod(i_polygon_update2(i),x) floor(i_polygon_update2(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
        U_speed(i_polygon_update2(i),:) = u';
        F(i_polygon_update2(i), opposite(2)) = (1/2+q2(i))*F_backup
(i_polygon_update2(i)+dx, opposite(2));
        F(i_polygon_update2(i), 2) = (2*q2(i)/(1/2+q2(i)))*F_backup
(i_polygon_update2(i), opposite(2)) + (1/2-q2(i))*F_backup
(i_polygon_update2(i)+dx, opposite(2));
        F(i_polygon_update2(i)+dx, 2) = F_backup(i_polygon_update2(i), 2)
+ (1/2-q2(i))/(1/2+q2(i))*F_backup(i_polygon_update2(i), opposite(2));

    end
end

% Direction 3
count = numel(q3);
for i = 1:count
    if q3(i)<1/2
        coordi = [mod(i_polygon_update3(i),x) floor(i_polygon_update3(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
```

```
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
        U_speed(i_polygon_update3(i),:) = u';
        F(i_polygon_update3(i), opposite(3)) = (1/2+q3(i))*F_backup
(i_polygon_update3(i)+dy, opposite(3));
        F(i_polygon_update3(i), 3) = (2*q3(i)/(1/2+q3(i)))*F_backup
(i_polygon_update3(i), opposite(3)) + (1/2-q3(i))*F_backup
(i_polygon_update3(i)+dy, opposite(3));
        F(i_polygon_update3(i)+dy, 3) = F_backup(i_polygon_update3(i), 3)
+ (1/2-q3(i))/(1/2+q3(i))*F_backup(i_polygon_update3(i), opposite(3));

    end
end

% Direction 4
count = numel(q4);
for i = 1:count
    if q4(i)<1/2
        coordi = [mod(i_polygon_update4(i),x) floor(i_polygon_update4(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
        U_speed(i_polygon_update4(i),:) = u';
```

```
        F(i_polygon_update4(i), opposite(4)) = (1/2+q4(i))*F_backup
(i_polygon_update4(i)-dx, opposite(4));
        F(i_polygon_update4(i), 4) = (2*q4(i)/(1/2+q4(i)))*F_backup
(i_polygon_update4(i), opposite(4)) + (1/2-q4(i))*F_backup
(i_polygon_update4(i)-dx, opposite(4));
        F(i_polygon_update4(i)-dx, 4) = F_backup(i_polygon_update4(i), 4)
+ (1/2-q4(i))/(1/2+q4(i))*F_backup(i_polygon_update4(i), opposite(4));

    end
end

% Direction 5
count = numel(q5);
for i = 1:count
    if q5(i)<1/2
        coordi = [mod(i_polygon_update5(i),x) floor(i_polygon_update5(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
        U_speed(i_polygon_update5(i),:) = u';
        F(i_polygon_update5(i), opposite(5)) = (1/2+q5(i))*F_backup
(i_polygon_update5(i)-dy, opposite(5));
        F(i_polygon_update5(i), 5) = (2*q5(i)/(1/2+q5(i)))*F_backup
(i_polygon_update5(i), opposite(5)) + (1/2-q5(i))*F_backup
(i_polygon_update5(i)-dy, opposite(5));
```

```
        F(i_polygon_update5(i)-dy, 5) = F_backup(i_polygon_update5(i), 5)
+ (1/2-q5(i))/(1/2+q5(i))*F_backup(i_polygon_update5(i), opposite(5));

    end
end

% Direction 6
count = numel(q6);
for i = 1:count
    if q6(i)<1/2
        coordi = [mod(i_polygon_update6(i),x) floor(i_polygon_update6(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
        U_speed(i_polygon_update6(i),:) = u';
        F(i_polygon_update6(i), opposite(6)) = (1/2+q6(i))*F_backup
(i_polygon_update6(i)+dx+dy, opposite(6));
        F(i_polygon_update6(i), 6) = (2*q6(i)/(1/2+q6(i)))*F_backup
(i_polygon_update6(i), opposite(6)) + (1/2-q6(i))*F_backup
(i_polygon_update6(i)+dx+dy, opposite(6));
        F(i_polygon_update6(i)+dx+dy, 6) = F_backup(i_polygon_update6(i), 6)
+ (1/2-q6(i))/(1/2+q6(i))*F_backup(i_polygon_update6(i), opposite(6));

    end
end
```

```
% Direction 7
count = numel(q7);
for i = 1:count
    if q7(i)<1/2
        coordi = [mod(i_polygon_update7(i),x) floor(i_polygon_update7(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
        U_speed(i_polygon_update7(i),:) = u';
        F(i_polygon_update7(i), opposite(7)) = (1/2+q7(i))*F_backup
(i_polygon_update7(i)-dx+dy, opposite(7));
        F(i_polygon_update7(i), 7) = (2*q7(i)/(1/2+q7(i)))*F_backup
(i_polygon_update7(i), opposite(7)) + (1/2-q7(i))*F_backup
(i_polygon_update7(i)-dx+dy, opposite(7));
        F(i_polygon_update7(i)-dx+dy, 7) = F_backup(i_polygon_update7(i), 7)
+ (1/2-q7(i))/(1/2+q7(i))*F_backup(i_polygon_update7(i), opposite(7));

    end
end

% Direction 8
count = numel(q8);
for i = 1:count
    if q8(i)<1/2
```

```
        coordi = [mod(i_polygon_update8(i),x) floor(i_polygon_update8(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
        U_speed(i_polygon_update8(i),:) = u';
        F(i_polygon_update8(i), opposite(8)) = (1/2+q8(i))*F_backup
(i_polygon_update8(i)-dx-dy, opposite(8));
        F(i_polygon_update8(i), 8) = (2*q8(i)/(1/2+q8(i)))*F_backup
(i_polygon_update8(i), opposite(8)) + (1/2-q8(i))*F_backup
(i_polygon_update8(i)-dx-dy, opposite(8));
        F(i_polygon_update8(i)-dx-dy, 8) = F_backup(i_polygon_update8(i), 8)
+ (1/2-q8(i))/(1/2+q8(i))*F_backup(i_polygon_update8(i), opposite(8));

    end
end

% Direction 9
count = numel(q9);
for i = 1:count
    if q9(i)<1/2
        coordi = [mod(i_polygon_update9(i),x) floor(i_polygon_update9(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);
```

```
        % For F
        U_speed(i_polygon_update9(i),:) = u';
        F(i_polygon_update9(i), opposite(9)) = (1/2+q9(i))*F_backup
(i_polygon_update9(i)+dx-dy, opposite(9));
        F(i_polygon_update9(i), 9) = (2*q9(i)/(1/2+q9(i)))*F_backup
(i_polygon_update9(i), opposite(9)) + (1/2-q9(i))*F_backup
(i_polygon_update9(i)+dx-dy, opposite(9));
        F(i_polygon_update9(i)+dx-dy, 9) = F_backup(i_polygon_update9(i), 9)
+ (1/2-q9(i))/(1/2+q9(i))*F_backup(i_polygon_update9(i), opposite(9));

    end
end




%% Boundary correction on the polygon for q>=1/2

% F_backup2 = F;
% Direction 2
count = numel(q2);
for i = 1:count
    if q2(i)>=1/2
        coordi = [mod(i_polygon_update2(i),x) floor(i_polygon_update2(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);
```

```
        % For F
        U_speed(i_polygon_update2(i),:) = u';
        F(i_polygon_update2(i), opposite(2)) = (q2(i)-1/2)/(q2(i)+1/2)
*F_backup(i_polygon_update2(i), opposite(2)) + F_backup(i_polygon_update2(i)
+dx, opposite(2));
        F(i_polygon_update2(i), 2) = (q2(i)-1/2)/(q2(i)+1/2)*F_backup
(i_polygon_update2(i), 2) + 1/(q2(i)+1/2)*F_backup(i_polygon_update2(i),
opposite(2));
        F(i_polygon_update2(i)+dx, 2) = 1/(q2(i)+1/2)*F_backup
(i_polygon_update2(i), 2);

    end
end

% Direction 3
count = numel(q3);
for i = 1:count
    if q3(i)>=1/2
        coordi = [mod(i_polygon_update3(i),x) floor(i_polygon_update3(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
        U_speed(i_polygon_update3(i),:) = u';
        F(i_polygon_update3(i), opposite(3)) = (q3(i)-1/2)/(q3(i)+1/2)
*F_backup(i_polygon_update3(i), opposite(3)) + F_backup(i_polygon_update3
```

```
(i)+dy, opposite(3));
        F(i_polygon_update3(i), 3) = (q3(i)-1/2)/(q3(i)+1/2)*F_backup
(i_polygon_update3(i), 3) + 1/(q3(i)+1/2)*F_backup(i_polygon_update3(i),
opposite(3));
        F(i_polygon_update3(i)+dy, 3) = 1/(q3(i)+1/2)*F_backup
(i_polygon_update3(i), 3);

    end
end

% Direction 4
count = numel(q4);
for i = 1:count
    if q4(i)>=1/2
        coordi = [mod(i_polygon_update4(i),x) floor(i_polygon_update4(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
        U_speed(i_polygon_update4(i),:) = u';
        F(i_polygon_update4(i), opposite(4)) = (q4(i)-1/2)/(q4(i)+1/2)
*F_backup(i_polygon_update4(i), opposite(4)) + F_backup(i_polygon_update4(i)
-dx, opposite(4));
        F(i_polygon_update4(i), 4) = (q4(i)-1/2)/(q4(i)+1/2)*F_backup
(i_polygon_update4(i), 4) + 1/(q4(i)+1/2)*F_backup(i_polygon_update4(i),
opposite(4));
```

```
        F(i_polygon_update4(i)-dx, 4) = 1/(q4(i)+1/2)*F_backup
(i_polygon_update4(i), 4);

    end
end

% Direction 5
count = numel(q5);
for i = 1:count
    if q5(i)>=1/2
        coordi = [mod(i_polygon_update5(i),x) floor(i_polygon_update5(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
        U_speed(i_polygon_update5(i),:) = u';
        F(i_polygon_update5(i), opposite(5)) = (q5(i)-1/2)/(q5(i)+1/2)
*F_backup(i_polygon_update5(i), opposite(5)) + F_backup(i_polygon_update5(i)
-dy, opposite(5));
        F(i_polygon_update5(i), 5) = (q5(i)-1/2)/(q5(i)+1/2)*F_backup
(i_polygon_update5(i), 5) + 1/(q5(i)+1/2)*F_backup(i_polygon_update5(i),
opposite(5));
        F(i_polygon_update5(i)-dy, 5) = 1/(q5(i)+1/2)*F_backup
(i_polygon_update5(i), 5);

    end
```

```
end

% Direction 6
count = numel(q6);
for i = 1:count
    if q6(i)>=1/2
        coordi = [mod(i_polygon_update6(i),x) floor(i_polygon_update6(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
        U_speed(i_polygon_update6(i),:) = u';
        F(i_polygon_update6(i), opposite(6)) = (q6(i)-1/2)/(q6(i)+1/2)
*F_backup(i_polygon_update6(i), opposite(6)) + F_backup(i_polygon_update6(i)
+dx+dy, opposite(6));
        F(i_polygon_update6(i), 6) = (q6(i)-1/2)/(q6(i)+1/2)*F_backup
(i_polygon_update6(i), 6) + 1/(q6(i)+1/2)*F_backup(i_polygon_update6(i),
opposite(6));
        F(i_polygon_update6(i)+dx+dy, 6) = 1/(q6(i)+1/2)*F_backup
(i_polygon_update6(i), 6);

    end
end

% Direction 7
count = numel(q7);
```

```
for i = 1:count
    if q7(i)>=1/2
        coordi = [mod(i_polygon_update7(i),x) floor(i_polygon_update7(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
        U_speed(i_polygon_update7(i),:) = u';
        F(i_polygon_update7(i), opposite(7)) = (q7(i)-1/2)/(q7(i)+1/2)
*F_backup(i_polygon_update7(i), opposite(7)) + F_backup(i_polygon_update7(i)
-dx+dy, opposite(7));
        F(i_polygon_update7(i), 7) = (q7(i)-1/2)/(q7(i)+1/2)*F_backup
(i_polygon_update7(i), 7) + 1/(q7(i)+1/2)*F_backup(i_polygon_update7(i),
opposite(7));
        F(i_polygon_update7(i)-dx+dy, 7) = 1/(q7(i)+1/2)*F_backup
(i_polygon_update7(i), 7);

    end
end

% Direction 8
count = numel(q8);
for i = 1:count
    if q8(i)>=1/2
        coordi = [mod(i_polygon_update8(i),x) floor(i_polygon_update8(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
```

```
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
        U_speed(i_polygon_update8(i),:) = u';
        F(i_polygon_update8(i), opposite(8)) = (q8(i)-1/2)/(q8(i)+1/2)
*F_backup(i_polygon_update8(i), opposite(8)) + F_backup(i_polygon_update8(i)
-dx-dy, opposite(8));
        F(i_polygon_update8(i), 8) = (q8(i)-1/2)/(q8(i)+1/2)*F_backup
(i_polygon_update8(i), 8) + 1/(q8(i)+1/2)*F_backup(i_polygon_update8(i),
opposite(8));
        F(i_polygon_update8(i)-dx-dy, 8) = 1/(q8(i)+1/2)*F_backup
(i_polygon_update8(i), 8);

    end
end

% Direction 9
count = numel(q9);
for i = 1:count
    if q9(i)>=1/2
        coordi = [mod(i_polygon_update9(i),x) floor(i_polygon_update9(i)/x)+1];
        v = [((coordi(1)-2)/N - center(1)) ((coordi(2)-2)/N - center(2))];
        u = [-v(2) v(1)];
        u = u*speed*norm(v)/norm(u);

        % For F
```

```
        U_speed(i_polygon_update9(i),:) = u';

        F(i_polygon_update9(i), opposite(9)) = (q9(i)-1/2)/(q9(i)+1/2)
*F_backup(i_polygon_update9(i), opposite(9)) + F_backup(i_polygon_update9(i)
+dx-dy, opposite(9));

        F(i_polygon_update9(i), 9) = (q9(i)-1/2)/(q9(i)+1/2)*F_backup
(i_polygon_update9(i), 9) + 1/(q9(i)+1/2)*F_backup(i_polygon_update9(i),
opposite(9));

        F(i_polygon_update9(i)+dx-dy, 9) = 1/(q9(i)+1/2)*F_backup
(i_polygon_update9(i), 9);


    end
end

polygon = polygon_update;



%% Bounce back boundary condition

% This is mainly for the outer circular cylinder wall

% For F
Fc(i_boundary,:) = F(i_boundary,:);

F(i_boundary,2) = Fc(i_boundary,4);
F(i_boundary,4) = Fc(i_boundary,2);
F(i_boundary,3) = Fc(i_boundary,5);
```

```
F(i_boundary,5) = Fc(i_boundary,3);
F(i_boundary,6) = Fc(i_boundary,8);
F(i_boundary,8) = Fc(i_boundary,6);
F(i_boundary,7) = Fc(i_boundary,9);
F(i_boundary,9) = Fc(i_boundary,7);


%% Boundary uncorrection on F

F(i_polygon_update2, 2) = F(i_polygon_update2, 2)./(1/2+q2);
F(i_polygon_update2, 4) = F(i_polygon_update2, 4)./(1/2+q2);
F(i_polygon_update4, 4) = F(i_polygon_update4, 4)./(1/2+q4);
F(i_polygon_update4, 2) = F(i_polygon_update4, 2)./(1/2+q4);

F(i_polygon_update3, 3) = F(i_polygon_update3, 3)./(1/2+q3);
F(i_polygon_update3, 5) = F(i_polygon_update3, 5)./(1/2+q3);
F(i_polygon_update5, 5) = F(i_polygon_update5, 5)./(1/2+q5);
F(i_polygon_update5, 3) = F(i_polygon_update5, 3)./(1/2+q5);

F(i_polygon_update6, 6) = F(i_polygon_update6, 6)./(1/2+q6);
F(i_polygon_update6, 8) = F(i_polygon_update6, 8)./(1/2+q6);
F(i_polygon_update8, 8) = F(i_polygon_update8, 8)./(1/2+q8);
F(i_polygon_update8, 6) = F(i_polygon_update8, 6)./(1/2+q8);

F(i_polygon_update7, 7) = F(i_polygon_update7, 7)./(1/2+q7);
F(i_polygon_update7, 9) = F(i_polygon_update7, 9)./(1/2+q7);
F(i_polygon_update9, 9) = F(i_polygon_update9, 9)./(1/2+q9);
```

```
F(i_polygon_update9, 7) = F(i_polygon_update9, 7)./(1/2+q9);
```