Graduation Project

Exploration of the lattice Boltzmann method in drag problems

Carlos Andrés Ponce Durán

Advisor:
Andres Leonardo González Mancera

Mechanical Engineering Department
Universidad de los Andes
Bogotá, Colombia
December 2015

Contents

Figures

Tables

1 Introduction

Computational methods have emerged as powerful techniques to investigate and explore physical and chemical phenomena and as powerful methods to solve real engineering problems. The finite element method (FEM) was applied for the first time to solve a structural problem in 1956 and it became a powerful technique to solve partial differential equation, heat transfer, and fluid dynamics problems. Also, at the same time, the finite difference method (FDM) was used to solve fluid dynamics problems. Years later, in 1980, the finite volume method (FVM) was developed in order to solve these fluid problems. Since then the FVM has been used extensively in phenomena transport problems (Mohamad, 2011).

All of these methods belong to the same family of weighted residual methods, and the only difference between these methods is the nature of the base and weighting functions. Then, in 1988, a new computational method was born to solve fluid problems: the lattice Boltzmann method (LBM). This method was introduced to overcome inconveniences of its predecessor, the lattice gas cellular automata, and to keep its advantages: parallelization and simplicity (Viggen, 2009). Since then LBM has emerged as a powerful alternative to solve fluid dynamics problems and is the main focus of this project.

In traditional computational fluid dynamics (CFD), Navier–Stokes equations (NS) solve mass, momentum and energy conservation equations on discrete nodes, elements, or volumes. This means that the nonlinear partial differential equations convert into a set of nonlinear algebraic equations, which are solved iteratively. In LBM, the fluid is replaced by groups/clusters of particles that stream along given directions (lattice links) and collide at the lattice nodes. The collision and streaming processes are local, therefore it can be programmed naturally for parallel processing machines (Guo & Shu, 2013). Another beauty of the LBM is handling complex phenomena such as moving boundaries (multiphase, solidification, and melting problems), naturally, without a need for face tracing method as it is in the traditional CFD.

LBM advantages over other CFD methods include:

- Simple algorithm structure.
- Thermodynamic models are easy to incorporate.
- Automated data pre-processing and mesh generation in a relatively small amount of time.

- Natural manipulation of complex phenomena like moving boundaries (multi-phase/fusion problems) without the need to implement face tracking.
- As opposed to other CFD methods, the Laplace equation is not solved in every time step, saving some computational resources.
- Incredibly easy to parallelize.

Even though the LBM has many advantages over other computational methods and takes a radically different path in solving fluid dynamics problems, it still has its major drawbacks, among which is that the models are restricted to small Mach numbers. This hasn't stopped this relatively new method from developing at great strides; open source projects such as Palabos (Palabos, 2015) demonstrate the potential of these methods and the will of its developers in advancing these computational techniques.

1.1 This Project

Now let's state what the objectives of this project are and the basic structure on which we will develop the ideas.

1.1.1 Objectives

- Understand the lattice Boltzmann method and the Palabos open source code to be able to model compressible fluid computationally.
- Develop a lattice Boltzmann model for the fluid flux around a sphere in 3D within the Palabos open source code.
- Compute and compare the drag coefficient in the model as a function of Reynolds for different Lattice Boltzmann techniques (BGK, Smagorinsky, and MRT).

The project is essentially the last objective that was stated. We want to develop a 3-D model of a sphere within a fluid, with the possibility of changing the Reynolds number of the system, and compute the drag coefficient on the sphere for the three techniques mentioned. This way, we can compare the results to real life experiments and make an assessment on how useful each model could be to calculate drag.

1.1.2 Project Structure

This project is divided into two main sections. The first part contains all of the theory behind the physics involved and the models developed, while the second part describes the methodology, results and their analyses.

The project contains the following five chapters:

- **Chapter 1. Introduction:** This chapter.
- **Chapter 2. Theory:** Fluid mechanics, kinetic theory of gases, lattice Boltzmann method, drag theory.
- **Chapter 3. Methodology:** Palabos, the model developed, simulation parameters.
- **Chapter 4. Results:** Drag coefficient comparison between different techniques and experimental values.
- **Chapter 5. Analysis:** Analysis of the results obtained. Resolution sensitivity analysis.
- **Chapter 6. Discussion and conclusion:** A summary of the research and some ideas on how it may be continued.

2 Theory

This chapter will explain the theory behind fluid dynamics, kinetic theory, the lattice Boltzmann method, drag theory, and everything else that is related to the execution of this project.

2.1 Methods and Scales

There are two main approaches in simulating the transport equations (heat, mass, and momentum), continuum and discrete. In the continuum approach, ordinary or partial differential equations can be achieved by applying conservation of energy, mass, and momentum for an infinitesimal control volume. Since it is difficult to solve the governing differential equations for many reasons (nonlinearity, complex boundary conditions, complex geometry), techniques such as finite difference, finite volume, or finite elements are used to convert the differential equations within a given boundary and initial conditions into a system of algebraic equations. The algebraic equations can be solved iteratively until convergence is achieved.

A general CFD method follows the next steps. First, the governing equations are identified (mainly partial differential equations). The next step is to discretize the domain into volume, girds, or elements, depending on the method of solution. We can look at this step as each volume, or node, or element contains a collection of particles (very large number, in the order of 10^{16}). This scale is the *macroscopic scale.* The velocity, pressure, temperature of all those particles is represented by a nodal value, or averaged over a finite volume.

On the other extreme, the medium can be considered made of small particles and these particles collide with each other. This scale is the *microscopic scale.* Hence, we need to identify the inter-particle or inter-molecular forces and solve ordinary differential equation of Newton's second law. At each time step, we need to identify location and velocity of each particle, i.e, trajectory of the particles. At this level, there is no definition of temperature, pressure, and thermo-physical properties, such as viscosity, thermal conductivity, heat capacity, etc. For instance, temperature and pressure are related to the kinetic energy of the particles (mass and velocity) and frequency of particles hitting on the boundaries, respectively. This method is called molecular dynamics (MD) simulations (Mohamad, 2011).

Fundamentally, MD is simple and can handle phase change and complex geometries without any difficulties and without introducing extra ingredients. However, it is important to specify the appropriate inter-particle force function. The main drawback or obstacle of using MD in simulation of a relatively large system is the computer resource and data reduction process, which will not be available for us in the seen future. A simple MD model example is the lattice gas automata (LGA), the precursor of the modern lattice Boltzmann method, which will be described in the next chapter.

We can now talk about a middleman between the general CFD method (macro) and the MD method (micro), the lattice Boltzmann method. The main idea of Boltzmann is to bridge the gap between micro-scale and macro-scale by not considering each particle behavior alone but behavior of a collection of particles as a unit.The property of the collection of particles is represented by a distribution function. The keyword is the distribution function. The distribution function acts as a representative for collection of particles. This scale is called the meso-scale. The mentioned methods are shown in figure 2.1.

Another way of understanding these three scales (macro, meso, and micro) through a physical lens is the following. The coarse level of description which we can perceive directly, we call the macroscopic scale. This means that properties of a fluid which we can perceive, such as density or velocity, refer to this scale. The equations of fluid mechanics, which describe how these macroscopic variables evolve, are then equations for the macroscopic scale.

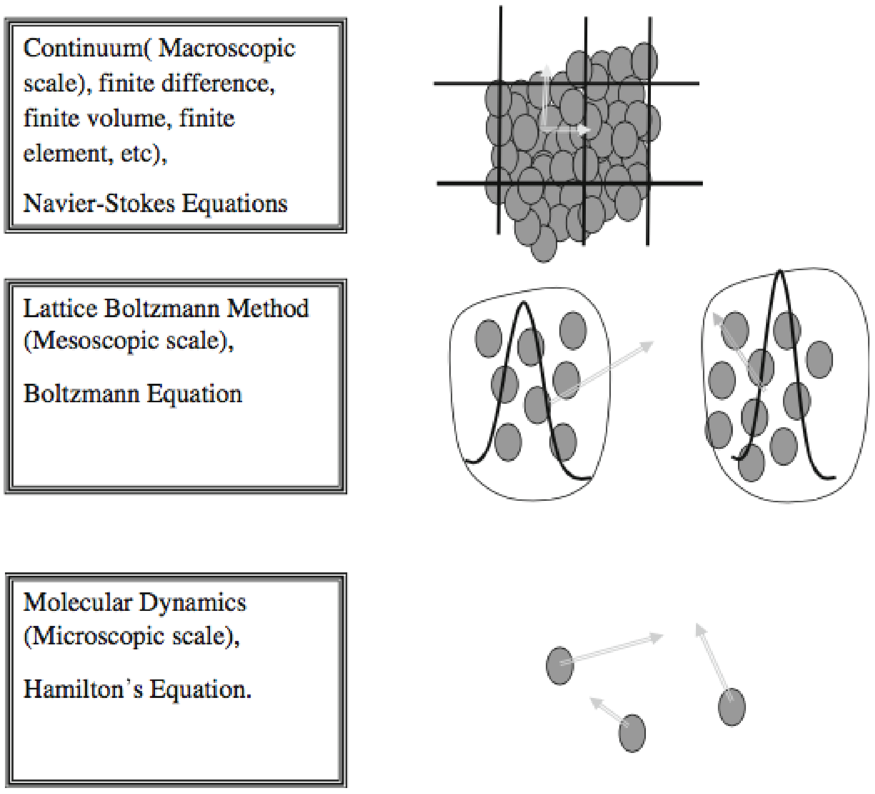

Continuum(Macroscopic scale), finite difference, finite volume, finite element, etc),
Navier-Stokes Equations
Lattice Boltzmann Method (Mesoscopic scale),
Boltzmann Equation
Molecular Dynamics (Microscopic scale),
Hamilton's Equation.

Figure 2.1: Simulation Techniques and their scales. The lattice Boltzmann method is in an in-between scale called the meso-scale which is not considered neither macro nor micro.

On the other hand, in the microscopic scale, we look at a much more complete level of detail. Instead of regarding the fluid as a smooth continuum, we look at all the different molecules of which it consists. Each molecule then has a mass m, a position $\mathbf{x}$, and a velocity $\mathbf{c}$, in addition to other variables to describe its internal state, such as its rotation and its vibration. The motion of each of these molecules can be described by Newton's laws of motion, though some quantum mechanics must also be applied if the internal configuration is considered.

To connect these two scales it is imperative that we consider expected values, or probabilistic averages, of the microscopic system. For a volume V, the density, momentum density, and energy density can be obtained by the following equations, respectively.

$$\rho(\mathbf{x}, t) = \lim_{V \to 0} \mathsf{E}\left(\frac{1}{V}\sum_{\mathbf{x}_i \in V} m_i\right) \tag{2.1}$$

$$\rho\mathbf{u}(\mathbf{x}, t) = \lim_{V \to 0} \mathsf{E}\left(\frac{1}{V}\sum_{\mathbf{x}_i \in V} m_i \mathbf{c}_i\right) \tag{2.2}$$

$$\rho E(\mathbf{x},t) = \lim_{V\to 0} \mathsf{E}\left(\frac{1}{2V}\sum_{\mathbf{x}_i\in V} m_i|\mathbf{c}_i|^2\right) \tag{2.3}$$

Physically, the mesoscopic scale, is the scale in which the level of detail is somewhere between the extremely detailed microscopic scale and the tangible macroscopic scale. Instead of tracking every particle, like in the micro-scale, we track the distribution of particles.

On the microscopic and mesoscopic scales, ideal gases are by far the simplest type of fluid to deal with. In a gas which is not too dense, the molecules are far enough apart that their interaction is approximately always through one on one collisions. For a very dense gas, the molecules are much closer together and the assumption that collisions are always one on one is no longer a sufficient approximation. On the macroscopic scale, though, the difference between gases and liquids is much less clear, as the same equations hold for both. The difference is manifested mainly by a difference in material parameters. For instance, liquids tend to have a significantly higher speed of sound than gases.

2.2 Cellular Automata and Lattice Gas Automata

One of the most basic MD models developed is the lattice gas cellular automaton models. These, and the cellular automata were the precursors of LBM. We dedicate a small section to basic cellular automata and the FHP model because they represent a somewhat simpler and possibly more intuitive framework for learning gases on a lattice. This material is not essential to applying LBM but it is interesting in its own right and might be helpful to developing a fuller understanding of LBM.

2.2.1 Cellular Automata

Sukop and Daniel T. Thorne (2005) explains a cellular automaton (CA) as an algorithmic entity that occupies a position on a grid or lattice point in space and interacts with its identical neighbors. A cellular automaton generally examines its own state and the states of some number of its neighbors at any particular time step and then resets its own state for the next time step according to simple rules. Hence, the rules and the initial and boundary conditions imposed on the group of cellular automata uniquely determine their evolution in time.

The simplest cellular automata models are those that exist in one dimension on a line and consider only their own states and those of their two nearest, adjacent neighbors. If these automata have only two possible states (0 and 1, for example), then there are 256 possible rules for updating the central automaton (2 options for each of the

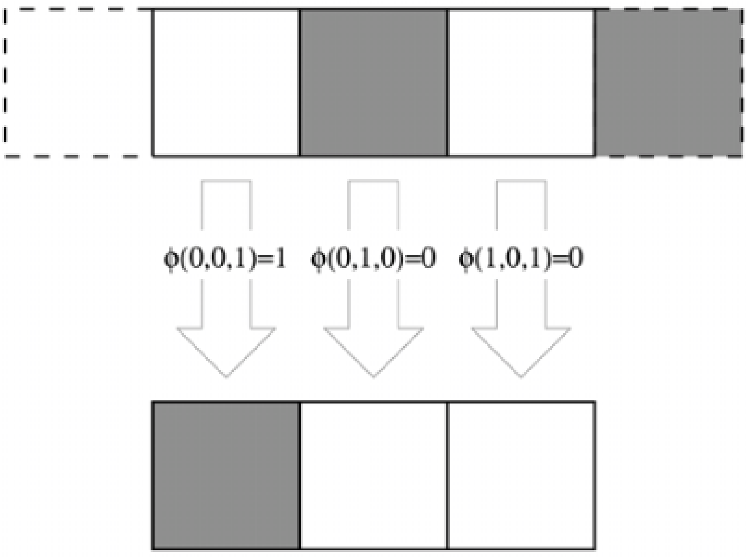


Figure 2.2: One-dimensional automaton. The basic components of a cellular automaton are shown: a tiling of space, an update rule, and a time step.

8 possible ways of organizing 3 automata with binary options $\rightarrow 2^8 = 256$ possible outcomes). We can write the update rule symbolically as $a_i' = \phi(a_{i-1}, a_i, a_{i+1})$ where a_i' is the updated state, ϕ is one of 256 functions, and a_{i-1}, a_i, a_{i+1} are the initial states of the automaton itself and its left and right neighbors, respectively (Chopard & Droz, 1998). An example of this can be seen in the figure 2.2.

Many cellular automata can be computed using binary arithmetic. Wolfram (1986, 2002) presented a complete classification and analysis of the 265 rules for the 2-state, 2-neighbor automata.

2.2.2 Lattice Gas Automata

The purpose of (LGA) is to simulate the behavior and interaction of many single particles in a gas as simply as possible (Wolf-Gladrow, 2000). They can be seen as very simple MD methods. Macroscopic quantities such as particle density and velocity can be recovered from this microscopic scope, making it possible to study the macroscopic behavior of a fluid in different geometries with this model.

The gas is modeled as a multitude of hard spheres moving along a regular grid, with a discrete set of possible velocities $\mathbf{c}_i$ for each particle. Collision between particles is handled by a set of elastic collision rules that must conserve the system's quantities of mass m and momentum $\mathbf{p}$. For different models these rules will be different, but the important thing is that the laws of conservation of mass and momentum are not violated.

For each time step, each particle is moved forward one step in the direction of its velocity. When two or more particles meet in the same node after a time step, a collision occurs. To conserve mass and momentum, the number of particles and the total velocity of all the particles in the node must be the same before and after the collision. When two particles collide head-on, for example, they are thrown out at right angles to their original velocities. This conserves momentum, as the sum of the velocities of the two particles is zero in both configurations.

The particle density in each node can be calculated simply from the number of particles present in the node, or

$$\rho(\mathbf{x}, t) = \sum_i n_i(\mathbf{x}, t) \tag{2.4}$$

where $\rho(\mathbf{x}, t)$ is the particle density at the node with position $\mathbf{x}$ at time t, and n_i is the Boolean occupation number, or the number of particles present (either 0 or 1) at this node with velocity $\mathbf{c}_i$. The quantities $\mathbf{x}$ and t are in lattice units. Similarly, the total momentum in each node can be calculated by

$$\rho(\mathbf{x}, t)\mathbf{u}(\mathbf{x}, t) = \sum_i \mathbf{c}_i n_i(\mathbf{x}, t) \tag{2.5}$$

where $\mathbf{u}(\mathbf{x}, t)$ refers to the mean velocity of the particles at this node. Hard walls can be implemented in the HPP model as special nodes which cause incoming particles to be reflected back. The boundaries of the simulated system can be hard walls, or they can be periodic. Periodic boundaries imply that a particle which exits the system at one edge will re-enter the system at the opposite edge. With hard boundaries, the behaviour of a fluid trapped in a box is simulated, while with periodic boundaries it is the behaviour of a fluid in a periodic system which is simulated. It is naturally possible to combine the two boundaries, for instance by having hard vertical boundaries on two sides and periodic boundaries on the other two. Additionally, force can be simulated in an LGA by randomly changing some particles' velocity to the direction of the force with a given probability. We will later on see how the lattice Boltzmann method model is very similar to these LGA models.

The FHP model

As an example of an LGA we explain the basics of the FHP model, which was suggested by Frisch, Hasslacher and Pomeau (hence the name) in 1986. The model developed took place in a hexagonal lattice, and it was shown that by moving from a square lattice (the simplest 2-D model developed before) to a hexagonal one, the incompressible Navier-Stokes equations could be recovered due to the extra rotational invariance added (Viggen, 2009).

The lattice vectors in the FHP model are $\mathbf{c}_1 = (1,0)$, $\mathbf{c}_2 = (1/2, \sqrt{3}/2)$, $\mathbf{c}_3 = (1/2, \sqrt{3}/2)$, $\mathbf{c}_4 = (-1,0)$, $\mathbf{c}_5 = (-1/2, -\sqrt{3}/2)$, $\mathbf{c}_6 = (1/2, -\sqrt{3}/2))$. These are shown in figure 2.3.

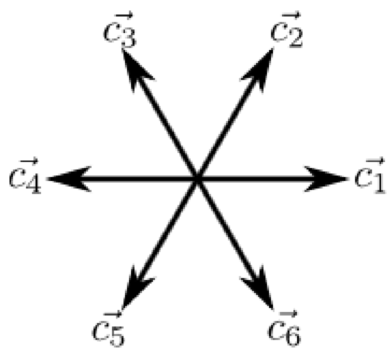


Figure 2.3: Lattice velocity vectors in the FHP model.

A hexagonal lattice allows two possible resolutions for a direct collision that conserve both mass and momentum, illustrated in figure 2.4. This is different from the HPP (square) model, where there is only one possible resolution. The resolution which is to occur is chosen randomly for every collision, with equal probability. Due to this stochastic element in the model, it is no longer fully deterministic, and does not have the time reversal property of the simpler HPP model. The FHP model also has a resolution for a three-particle collision. Such a collision reflects each particle back the way it came. This is illustrated in figure 2.5. For all other collisions, no change in particle distributions should be performed.

2.3 Kinetic Theory

The kinetic theory is the basis for the LBM. In the next sections a small review and the basics of kinetic theory that are relevant to the LBM will be shown.

2.3.1 Particle Dynamics

Succi (2001) explains the basics of particle dynamics in the following way. The main building block of all the materials in nature are atoms and molecules. These particles can be visualized as solid spheres moving “randomly” in conservatory manner in a free space. The motion of these particles satisfies conservation of mass, momentum, and energy. Hence, Newton’s second law (momentum conservation) can be applied, which states that the rate of change of linear momentum is equal to the net applied force.

$$\mathbf{F} = \frac{d(m\mathbf{c})}{dt} \tag{2.6}$$

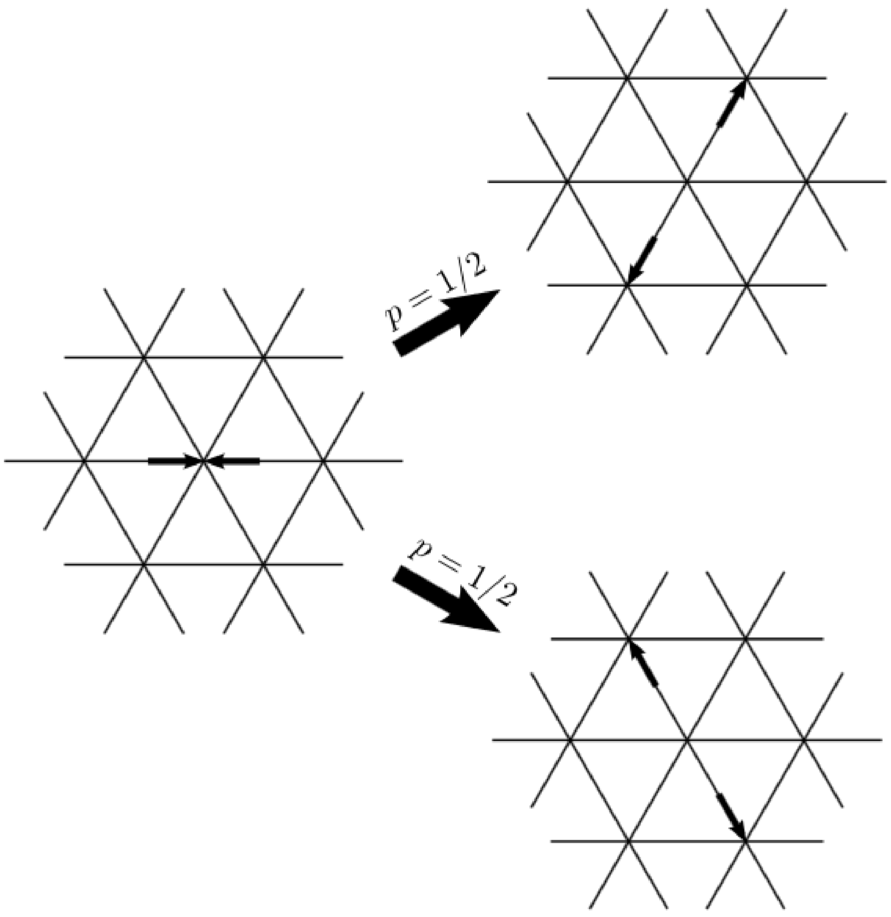


Figure 2.4: Direct collision rules in the FHP model. The two possible outcomes have the same probability.

where F stands for the inter-molecular and external forces, m is the mass of the particle, $\mathbf{c}$ is the velocity vector of the particle and t is the time. For a constant mass, the equation can be simplified as

$$\mathbf{F} = m\frac{d\mathbf{c}}{dt} = m\mathbf{a} \tag{2.7}$$

where $\mathbf{a}$ is the acceleration vector. The position of the particle can be determined from the definition of velocity,

$$\mathbf{c} = \frac{d\mathbf{x}}{dt} \tag{2.8}$$

where $\mathbf{x}$ is the position vector of the particle relative to the origin. In a MD simulation, the above equations are solved provided that $\mathbf{F}$ is a defined function.

If an external force, $\mathbf{F}$, is applied to a particle of mass m, the velocity of the particle

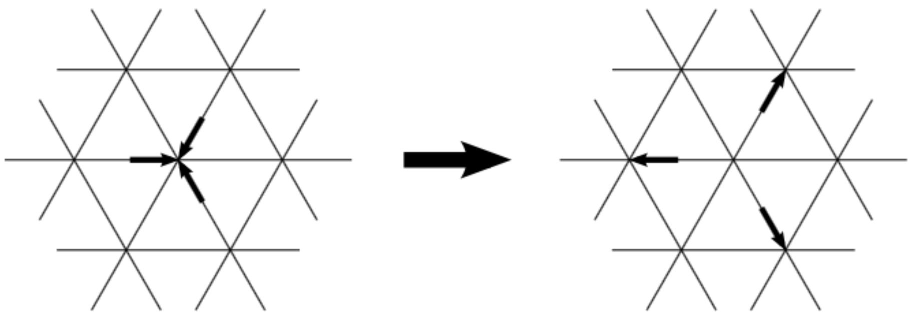


Figure 2.5: Triple collision rule in the FHP model. Particles are reflected back in the same direction whence they came.

will change from $\mathbf{c}$ to $\mathbf{c} + \mathbf{F}dt/m$ and its position will change from $\mathbf{x}$ to $\mathbf{x} + \mathbf{c}dt$. In the absence of an external force, the particle moves freely from one location to another without changing its direction and speed, assuming no collision takes place.

The magnitude of the particle velocity increases and interaction between the particles increases as the internal energy of the system increases, such as when heating occurs. Increases in the kinetic energy of the molecules are referred as increases in temperature in the macroscopic world. The particles are continuously hitting against the container walls. The force exerted by those actions per unit area is referred as pressure in the macroscopic measure. From this simple model, we can see that there is a relationship between temperature and pressure, as the temperature increases, which means the kinetic energy of the molecules increase, we expect that the probability of particles bombarding the container wall, increases.

2.3.2 Pressure and Temperature

Let us assume that a single particle moving with a speed, c_x (in the x direction), inside a tube of length L and hitting the ends of the tube, continuously. The force exerted by the particle on an end is equal to the rate of change of the momentum (Hänel, 2004), then

$$F\Delta t = mc_x - (-mc_x) = 2mc_x \tag{2.9}$$

where Δt is time between hits. This equation is an integration of Newton's second law. The time between hits is equal to $2L/c_x$, which is the time needed for the particle to travel from one end to another end and return to the same location. From this we get $2LF/c_x = 2mc_x$, which in turn means that

$$F = \frac{mc_x^2}{L} \tag{2.10}$$

The results can be generalized for N partices. The total force exerted by N particles is proportional to Nmc^2/L. In general, instead of having a velocity in one direction, we have the the general velocity $c^2 = c_x^2 + c_y^2 + c_z^2$. In this scale it is fair to make the assumption that these components are equal, and therefore $c^2 = 3c_x^2$. Then, the total force can be written as,

$$F = \frac{Nmc^2}{3L} \tag{2.11}$$

The pressure is defined as a force per unit area, perpendicular to the force vector, $P = F/A$. Then, the pressure exerted by N particle on the ends of the tube is equal to

$$P = \frac{Nmc^2}{3LA} = \frac{Nmc^2}{3V} \tag{2.12}$$

where V stands for volume, which is equal to LA.

This simple picture of molecular motion relates the pressure in macroscopic sense to the kinetic energy of the molecules. In this sense we can separate the kinetic energy term from the equation, $mc^2/2$, to get

$$P = \frac{mc^2}{2}\frac{2N}{3V} \tag{2.13}$$

In other words, the pressure is related to the kinetic energy (KE) in the following way:

$$P = \frac{2}{3}nKE \tag{2.14}$$

where n is the number of molecules per unit volume.

In this simple, ideal gas model, we neglected the effect of molecular interaction and effect of the molecular size. However, for a gas at a room temperature, the results are surprisingly true (Mohamad, 2011). In a real system, the particle has volume and collisions take place between the particles.

Experimentally, it is well-established that for gases far from the critical conditions, in the "ideal gas" state, the state equation can be expressed as

$$PV = nRT \tag{2.15}$$

where n is the number of moles ($n = N/N_A$), where N_A is Avogadro's number and

R is the gas constant. Equating equations 2.13 and 2.15 gives us

$$\frac{N}{N_A}RT = \frac{mc^2}{2}\frac{2N}{3V} \tag{2.16}$$

Introducing the Boltzmann constant $k = R/N_A$, we can deduce that the kinetic energy of a gas is

$$KE = \frac{mc^2}{2} = \frac{3}{2}kT \tag{2.17}$$

This is a very interesting result given that now we know that the temperature and the pressure in the macroscopic world are no more than a higher scale of the observed kinetic energy of the particles in the macroscopic world: the macroscopic world is the humanly observable behavior of microscopic behavior.

2.3.3 Distribution Function

The following derivation of the lattice Boltzmann equation is based on Viggen (2014). In 1859, Maxwell recognized that dealing with a huge number of particles is very difficult to formulate, even though the governing equation (Newton's second law) was known, this due to the fact that tracing the trajectory of each molecule is out of hand for a macroscopic system: the number of particles is too large to formulate their motion and dynamics effectively in a large scale. Then, the idea of averaging came into picture.

The idea of Maxwell is that the knowledge of velocity and position of each molecule at every instant of time is not important. The distribution function is the important parameter to characterize the effect of the molecules: what percentage of the molecules in a certain location of a container have velocities within a certain range, at a given instant of time. The molecules of a gas have a wide range of velocities colliding with each others, the fast molecules transfer momentum to the slow molecule after every collision. The result of each one of these collisions must comply with the law of conservation of momentum. For a gas in thermal equilibrium, the distribution function is not a function of time, where the gas is distributed uniformly in the container; the only unknown is the velocity distribution function.

The distribution function, $f(\mathbf{x}, \mathbf{c}, t)$, then can be seen as a generalisation of density; it represents the density of particles in both physical space and velocity space simultaneously, i.e. phase space. In other words, $f(\mathbf{x}, \mathbf{c}, t)$ indicates the density (or probability) of particles with position $\mathbf{x}$ and velocity $\mathbf{c}$ at time t.

The distribution function lets us find other, more familiar quantities. $f(\mathbf{x}, \mathbf{c}, t)d\mathbf{c}$ is the spatial density of particles which have velocities which lie inside an infinitesimal velocity space volume $d\mathbf{c}$ at $\mathbf{c}$. Furthermore, $f(\mathbf{x}, \mathbf{c}, t)d\mathbf{x}d\mathbf{c}$ is the mass of particles with such velocities and positions which lie inside an infinitesimal physical space volume $d\mathbf{x}$ at $\mathbf{x}$.

This distribution function represents the mesoscopic scale. It grabs the information of an average number of particles but does not represent them individually, nor does it represent a physically observable quantity within a finite element/volume. This way of looking at a fluid problem lets us get into the particle scale in a pragmatic way.

This distribution function is sufficient to find macroscopic properties such as fluid density, fluid velocity, and internal energy in the fluid. These properties can be found as moments, where the distribution function $f(\mathbf{x}, \mathbf{c}, t)$ is weighted with some function of $\mathbf{c}$ and integrated over the entire velocity space. These moments, which link the mesoscopic and macroscopic scales, are somewhat similar to the equations 2.1 - 2.3 which link the microscopic and macroscopic scales.

For the simplest moment we do not weigh with anything. As mentioned previously, $f(\mathbf{x}, \mathbf{c}, t)d\mathbf{c}$ is the spatial density of particles that have velocities within a certain infinitesimal velocity range. Thus, if we integrate over all the velocities, i.e. the entire velocity space, we get the physical mass density, most commonly simply called density

$$\rho(\mathbf{x}, t) = \int f(\mathbf{x}, \mathbf{c}, t)d\mathbf{c} \tag{2.18}$$

Now, if we weight with $\mathbf{c}$, we get the equation $\mathbf{c}f(\mathbf{x}, \mathbf{c}, t)d\mathbf{c}$, which is the momentum density of the particles in the given velocity range. Integrating over all the velocities gives us the total momentum density

$$\rho\mathbf{u}(\mathbf{x}, t) = \int \mathbf{c}f(\mathbf{x}, \mathbf{c}, t)d\mathbf{c} \tag{2.19}$$

Where $\mathbf{u}$ is the average velocity of the particles, which corresponds to the fluid velocity in fluid mechanics.

Similarly, weighing in with $\frac{1}{2}|\mathbf{c}|^2$ and integrating over the velocity space gives the energy density

$$\rho E(\mathbf{x}, t) = \frac{1}{2}\int |\mathbf{c}|^2 f(\mathbf{x}, \mathbf{c}, t)d\mathbf{c} \tag{2.20}$$

Where E is the specific energy. We have assumed here that the kinetic energy is given only by the translational movement of the particles; this is only correct if the gas is monatomic, as we earlier assumed. For a polyatomic gas, there would also be contributions from rotational and vibrational energy,which cannot be as easily represented through the distribution function.

This total kinetic energy may be split up into two parts, the kinetic energy density due to the bulk movement of the fluid, $\frac{1}{2}\rho|\mathbf{u}|^2$, and the internal energy density ρe, which is due to random thermal movement of particles, and is independent of the fluid velocity $\mathbf{u}$. In this way, we can express the specific energy as

$$\rho E = \rho\left(e + \frac{1}{2}|\mathbf{u}|^2\right) \tag{2.21}$$

Where e is the specific internal energy. It is also useful to split the particle velocity $\mathbf{c}$ into two components: The fluid velocity $\mathbf{u}$ and the peculiar velocity $\mathbf{v}$:

$$\mathbf{c} = \mathbf{v} + \mathbf{u} \tag{2.22}$$

Since it describes the deviation from the mean velocity, the peculiar velocity cannot contribute to the momentum. With this manipulation it is possible to find the internal energy density as a function of the peculiar velocity and the distribution function:

$$\rho(\mathbf{x}, t) = \frac{1}{2}\int |\mathbf{v}|^2 f(\mathbf{x}, \mathbf{c}, t) d\mathbf{c} \tag{2.23}$$

2.3.4 Equilibrium and the Maxwell-Boltzmann distribution

When two particles collide, their velocities are changed. Their new velocities depend on their pre-collision positions and velocities and the intermolecular forces during the collision. However, we can assume that collisions tend to distribute the particles' velocities evenly in all directions around their mean velocity $\mathbf{u}$. This means that if we take a gas of particles with any initial distribution and leave it for long enough, it will eventually reach an equilibrium state where all directions of the peculiar velocity $\mathbf{v}$ are equally probable.

This even distribution of velocities means that the distribution function is only dependent on the peculiar velocity. The distribution function has the same value for peculiar velocities where $|\mathbf{v}|^2 = v_x^2 + v_y^2 + v_z^2$ is the same. We can therefore simplify the notation for the distribution function at equilibrium to $f^{(0)}(|\mathbf{v}|)$. We also assume that the distribution function is separable in the different $\mathbf{v}$ coordinates, so

$$f^{(0)}(|\mathbf{v}|) = f_x^{(0)}(v_x) f_y^{(0)}(v_y) f_z^{(0)}(v_z) \tag{2.24}$$

For a constant $|\mathbf{v}|^2$, $f^{(0)}(|\mathbf{v}|)$ is constant, and

$$\mathsf{ln} f_x^{(0)}(v_x) + \mathsf{ln} f_y^{(0)}(v_y) + \mathsf{ln} f_z^{(0)}(v_z) = \mathsf{constant} \tag{2.25}$$

This is only solved if the equilibrium distributions for the different axes are of the form

$$\mathsf{ln} f_x^{(0)}(v_x) = a - b v_x^2 \tag{2.26}$$

Which in turn means that

$$f_x^{(0)}(v_x) = \mathsf{e}^a \mathsf{e}^{-b v_x^2} \tag{2.27}$$

where a and b are two constants that are independent of v. Thus, the equilibrium distribution is of the form

$$f^{(0)}(|\mathbf{v}|) = \mathsf{e}^{3a} \mathsf{e}^{-b(v_x^2+v_y^2+v_z^2)} = \mathsf{e}^{3a} \mathsf{e}^{-b|\mathbf{v}|^2} \tag{2.28}$$

The constants a and b can be determined from the moments of $f^{(0)}$. First, the moment of density is found:

$$\rho = \int f^{(0)}(|\mathbf{v}|) d\mathbf{c} = \mathsf{e}^{3a} \int\limits_{-\infty}^{\infty} \mathsf{e}^{-b v_x^2} dv_x \int\limits_{-\infty}^{\infty} \mathsf{e}^{-b v_y^2} dv_y \int\limits_{-\infty}^{\infty} \mathsf{e}^{-b v_z^2} dv_z = \mathsf{e}^{3a} \left(\frac{\pi}{b}\right)^{3/2} \tag{2.29}$$

Rearranging we can notice that $\mathsf{e}^{3a} = \rho \left(\frac{b}{\pi}\right)^{3/2}$, so then, the equilibrium distribution becomes

$$f^{(0)}(|\mathbf{v}|) = \rho \left(\frac{b}{\pi}\right)^{3/2} \mathsf{e}^{-b|\mathbf{v}|^2} \tag{2.30}$$

Finally, b can be determined by using the moment of energy previously found. Since $f^{(0)}(|\mathbf{v}|)$ is spherically symmetric around $\mathbf{v} = 0$, we can perform the substitution

$d\mathbf{c} = 4\pi|\mathbf{v}|^2 d|\mathbf{v}|$. The integral then becomes

$$\rho e = \frac{1}{2}\int_0^\infty |\mathbf{v}|^2 f^{(0)}(|\mathbf{v}|) 4\pi |\mathbf{v}|^2 d|\mathbf{v}| = 2\rho\pi \left(\frac{b}{\pi}\right)^{3/2} \int_0^\infty |\mathbf{v}|^4 \mathrm{e}^{-b|\mathbf{v}|^2} d|\mathbf{v}| = \frac{3\rho}{4b} \tag{2.31}$$

This lets us identify b using several different thermodynamic quantities,

$$b = \frac{3}{4e} = \frac{\rho}{2p} = \frac{m}{2k_bT} \tag{2.32}$$

So, finally, the equilibrium distribution has the possible form (which depends on which thermodynamic variables are used to describe it):

$$\begin{aligned} f^{(0)}(|\mathbf{v}|) &= \rho\left(\frac{3}{4\pi e}\right)^{3/2} \exp\left[\frac{-3|\mathbf{v}|^2}{4e}\right] \\ &= \rho\left(\frac{\rho}{2\pi p}\right)^{3/2} \exp\left[\frac{-\rho|\mathbf{v}|^2}{2p}\right] \\ &= \rho\left(\frac{m}{2\pi k_bT}\right)^{3/2} \exp\left[\frac{-m|\mathbf{v}|^2}{2k_bT}\right] \end{aligned} \tag{2.33}$$

This equilibrium distribution is called the Maxwell-Boltzmann distribution because it was first found by James Clerk Maxwell using a derivation similar to the one used here, and was later found by Ludwig Boltzmann using more rigorous statistical mechanics. Figure 2.6 shows an example of the equilibrium distribution function, normalized, of a neon gas.

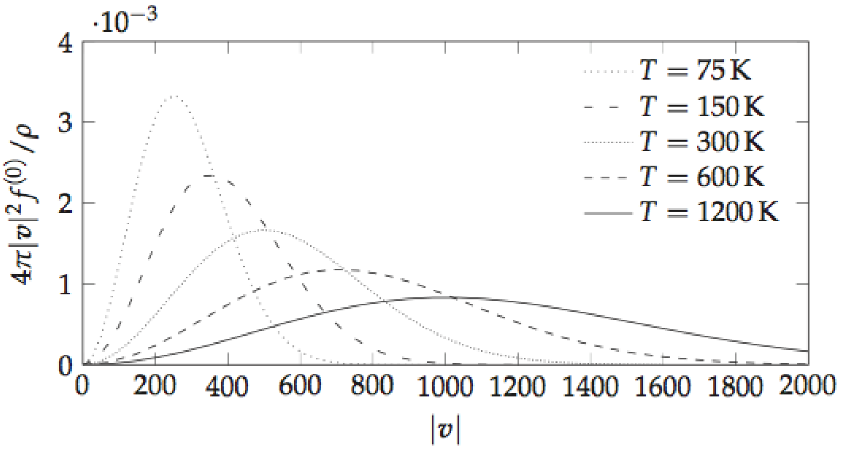


Figure 2.6: Normalized equilibrium distribution function for a neon gas.

2.4 The Boltzmann Equation

So far we have talked about the distribution function, its moments, and its value at equilibrium, but we still know nothing about how it actually evolves with time. We will therefore now derive the equation that describes the evolution of the distribution function.

The distribution function is a function of $\mathbf{x}$, $\mathbf{c}$, and t. Therefore, its total differential with respect to t is

$$\frac{df}{dt} = \left(\frac{\partial f}{\partial x_\alpha}\right)\frac{dx_\alpha}{dt} + \left(\frac{\partial f}{\partial c_\alpha}\right)\frac{dc_\alpha}{dt} + \left(\frac{\partial f}{\partial t}\right)\frac{dt}{dt} \tag{2.34}$$

where the subscript α is used according to the index notation rules. Here, dx_α/dt is the particles' velocity c_α, and c_α/dt is their acceleration, which by Newton's second law is given by the body force density as $dc_\alpha/dt = F_\alpha/\rho$. Then, the equation can be rewritten as

$$\frac{df}{dt} = \frac{\partial f}{\partial t} + c_\alpha\left(\frac{\partial f}{\partial x_\alpha}\right) + \frac{F_\alpha}{\rho}\left(\frac{\partial f}{\partial c_\alpha}\right) \tag{2.35}$$

Writing this equation on vector form with df/dt rewritten as the collision operator $\Omega(f)$, gives

$$\frac{\partial f}{\partial t} + c\cdot\nabla f + \frac{\mathbf{F}}{\rho}\cdot\nabla_c f = \Omega(f) \tag{2.36}$$

Here, $\nabla_c f$ is the gradient of f in velocity space. This equation is called the Boltzmann equation after Ludwig Boltzmann, who devised it in the late XIX century.

Collision Operator

The collision operator may have many different forms, as long as it fulfils certain conditions. Three quantities are always conserved in a collision: mass, momentum, and, if the collisions are elastic, translational energy. Another criterion for collision operators is that they must ensure that the distribution function always evolves towards equilibrium.

Another criterion for collision operators is that they must ensure that the distribution function always evolves towards equilibrium. This collision operator fulfils the conditions discussed, but is very cumbersome. Alternative collision models were later proposed. The goal was to find a collision model that was simpler than Boltzmann's original one, but which still gave a largely correct macroscopic behaviour. In the scientific field of lattice Boltzmann methods, variants of the BGK collision operator are generally used:

$$\Omega(f) = -\frac{1}{\tau}(f - f^{(0)}) \tag{2.37}$$

Here, τ is called the relaxation time. This collision operator proposed by Bhatnagar, Gross, and Krook (Hence the name) in 1954 as a very simple model for particle collisions. It captures this behaviour by directly modeling the relaxation process instead of attempting to follow the details of the collisions. We will discuss it further in a later section.

2.4.1 The Discrete-velocity Boltzmann Equation

The first step in discretizing the Boltzmann equation is to discretize velocity space. To do this, the first step is to approximate the Maxwell-Boltzmann distribution (eq. 2.33) by expanding it up to the second order:

$$\begin{aligned} f^{(0)}(\mathbf{x}, \mathbf{c}, t) &= \frac{\rho}{(2\pi c_0^2)^{3/2}} \mathsf{exp}[-(c_\alpha c_\alpha - 2c_\alpha u_\alpha + u_\alpha u_\alpha)/2c_0^2] \\ &\approx \frac{\rho}{(2\pi c_0^2)^{3/2}} \mathsf{exp}\left[-c_\alpha c_\alpha/2c_0^2 \left(1 + \frac{c_\alpha u_\alpha}{c_0^2} + \frac{(c_\alpha u_\alpha)^2}{2c_0^4} - \frac{u_\alpha u_\alpha}{2c_0^2}\right)\right] \end{aligned} \tag{2.38}$$

Next step is to discretize the velocity space, restricting $\mathbf{c}$ to a finite set of velocities $\mathbf{c}_i$. This way, the distribution function $f(\mathbf{x}, \mathbf{c}, t)$ becomes $f_i(\mathbf{x}, t)$, representing the density at the position $\mathbf{x}$ and time t of particles with velocity $\mathbf{c}_i$. Additionally, to make things clearer, we replace the coefficient $\mathsf{exp}[-c_\alpha c_\alpha/2c_0^2]/(2\pi c_0^2)^{3/2}$ with a single weighting coefficient w_i that depends on the discrete velocity i, ending up with the classic discrete equilibrium distribution:

$$f_i^{(0)} = \rho w_i \left(1 + \frac{c_\alpha u_\alpha}{c_0^2} + \frac{(c_\alpha u_\alpha)^2}{2c_0^4} - \frac{u_\alpha u_\alpha}{2c_0^2}\right) \tag{2.39}$$

This is arguably the optimally stable isothermal polynomial discrete equilibrium distribution. Having discretized the velocity space and using the the discrete analogue of the BGK operator (eq. 2.37), the Boltzmann equation becomes the discrete-velocity Boltzmann equation (DVBE)

$$\frac{\partial f_i}{\partial t} + c_{i\alpha}\frac{\partial f_i}{\partial x_\alpha} = -\frac{1}{\tau}(f_i - f_i^{(0)}) \tag{2.40}$$

2.4.2 The Lattice Boltzmann Equation

While the discrete-velocity Boltzmann equation (eq. 2.40) is discrete in velocity space, it is still continuous in physical space and time. To find the fully discrete lattice Boltzmann equation, it is necessary to perform further discretization. A possible way of discretizing it is to integrate along characteristics.

Let's assume that we can write the distribution function as $f_i = f_i(\mathbf{x}(a), t(a))$, where a denotes the position along the characteristic. The total differential of f_i in a, assuming no external forces, is

$$\frac{df_i}{da} = \left(\frac{\partial f_i}{\partial t}\right)\frac{dt}{da} + \left(\frac{\partial f_i}{\partial x_\alpha}\right)\frac{dx_\alpha}{da} = -\frac{1}{\tau}(f_i - f_i^{(0)}) \tag{2.41}$$

The right equality in the previous equation holds if the total differential is the left-hand side of the DVBE (eq. 2.40), which is true if

$$\frac{dt}{da} = 1, \qquad \frac{dx_\alpha}{da} = c_{i\alpha} \tag{2.42}$$

Thus, f_i can be written as $f_i(\mathbf{x} + \mathbf{c}_i a, t + a)$. Integrating eq. 2.41 from $a = 0$ to $a = \Delta t$, that is, from one time step to the next, and using the fundamental theorem of calculus on the left side, we find

$$f_i(\mathbf{x}+\mathbf{c}_i\Delta t, t+\Delta t) - f_i(\mathbf{x}, t) = -\frac{1}{\tau}\int_0^{\Delta t}[f_i(\mathbf{x}+\mathbf{c}_i a, t+a) - f_i^{(0)}(\mathbf{x}+\mathbf{c}_i a, t+a)]da \tag{2.43}$$

When implementing the lattice Boltzmann equation on a computer, it is convenient to scale time and space so that the lattice resolution Δx and time resolution Δt both equal one. This choice of units is called lattice units. With the most common first order discretization, the integral in the precious equation is approximated with the rectangle method, giving rise to the first order lattice Boltzmann equation:

$$f_i(\mathbf{x} + \mathbf{c}_i, t + 1) - f_i(\mathbf{x}, t) = -\frac{1}{\tau}[f_i(\mathbf{x}, t) - f_i^{(0)}(\mathbf{x}, t)] \tag{2.44}$$

This is the equation that is generally used in the lattice Boltzmann method and the one that was used for the present project.

2.5 The Lattice Boltzmann Method

The previous sections have been quite mathematical, focusing on the derivation of the lattice Boltzmann equation. The point to all those equations and derivations is to get at the computational lattice Boltzmann method, which will be presented just now.

In a regular spatial grid of nodes (a lattice), each node contains a number of distribution functions $f_i(\mathbf{x}, t)$. These represent the density of particles with velocity $\mathbf{c}_i$ in the node at $\mathbf{x}$ and time t. These velocity vectors are chosen so that particles are brought from one node to its neighbours (or remain stationary in the case $\mathbf{c}_i = 0$) during one time step.

The distribution functions f_i can be used to find the more familiar macroscopic quantities of mass density and momentum density:

$$\rho(\mathbf{x}, t) = \sum_i f_i(\mathbf{x}, t) \tag{2.45}$$

$$\rho\mathbf{u}(\mathbf{x}, t) = \sum_i \mathbf{c}_i f_i(\mathbf{x}, t) \tag{2.46}$$

the fluid velocity is found as $\mathbf{u}(\mathbf{x}, t) = (\sum_i \mathbf{c}_i f_i)/(\sum_i f_i)$. The pressure is determined through the isothermal relation $p = c_0^2\rho$, where the ideal speed of sound c_0 is determined by the choice of velocity set, but in most cases it is $c_0 = 1/\sqrt{3}$.

Once in each time step, the particles in each node collide, which is modelled as a relaxation of the distribution function towards the equilibrium distribution (eq. 2.39)

$$f_i^{(0)} = \rho w_i \left(1 + \frac{\mathbf{c}_i \cdot \mathbf{u}}{c_0^2} + \frac{(\mathbf{c}_i \cdot \mathbf{u})^2}{2c_0^4} - \frac{\mathbf{u} \cdot \mathbf{u}}{2c_0^2}\right) \tag{2.47}$$

This is constructed from the macroscopic moments of density ρ and fluid velocity $\mathbf{u}$, found using the above relations. The weighting coefficients w_i are dependent on the choice of velocity set in the model. What happens in each time step is that in each node collisions between incoming particles f_i are represented by a relaxation with characteristic time τ towards the equilibrium distribution $f_i^{(0)}$, resulting in a new distribution of particles which is streamed on to the neighboring nodes. All this respresented by the first order lattice Boltzmann equation new distribution of particles which is streamed on to the neighbouring nodes. All this is represented by the first order lattice Boltzmann equation, eq. 2.5:

$$f_i(\mathbf{x} + \mathbf{c}_i, t + 1) - f_i(\mathbf{x}, t) = -\frac{1}{\tau}[f_i(\mathbf{x}, t) - f_i^{(0)}(\mathbf{x}, t)]$$

The right-hand side represents the distribution of particles after collisions have taken place, and the left-hand side represents these particles appearing in neighboring nodes in the next time step.

Logically, the lattice Boltzmann algorithm consists of the following steps in each node:

- **Macroscopic quantities:** From the known distribution of particles f_i in each node, calculate the node's fluid density ρ and fluid velocity $\mathbf{u}$.
- **Equilibrium distribution:** From the fluid density ρ and the fluid velocity $\mathbf{u}$ calculated in the previous step, calculate the node's equilibrium distribution $f_i^{(0)}$.
- **Collision:** The post-collision distribution functions are calculated according to the right-hand side of eq. 2.5.
- **Streaming:** The post-collision distribution functions are streamed to neighbouring nodes according to their velocities.

Apart from the streaming step, each step is completely local within each node, meaning no quantities outside that node are used. In the streaming step, particles only stream to neighbouring nodes. This locality is a very important property of the lattice Boltzmann method which allows for massive parallelization of the algorithm.

2.5.1 Velocity sets

The lattice Boltzmann method is handles the DVBE, which is the discretized velocity Boltzmann equation. This means that the velocities have to belong to a discretized set of velocities when implemented in a discrete algorithm.

These velocity sets have to meet some criteria in order to be usable in the lattice Boltzmann method proposed. We won't get into these criteria, but suffice it to say that the lattice velocity set and its weights w_i have to meet isotropy conditions.

In lattice Boltzmann terminology, it is common to denote different velocity sets and the lattices that they form as $DdQq$, d being the number of spatial dimensions, and q the number of velocities. Some examples of this are shown in figures 2.7 and 2.8. The project developed will be carried out in 3-D and will use the velocity set $D3Q15$.

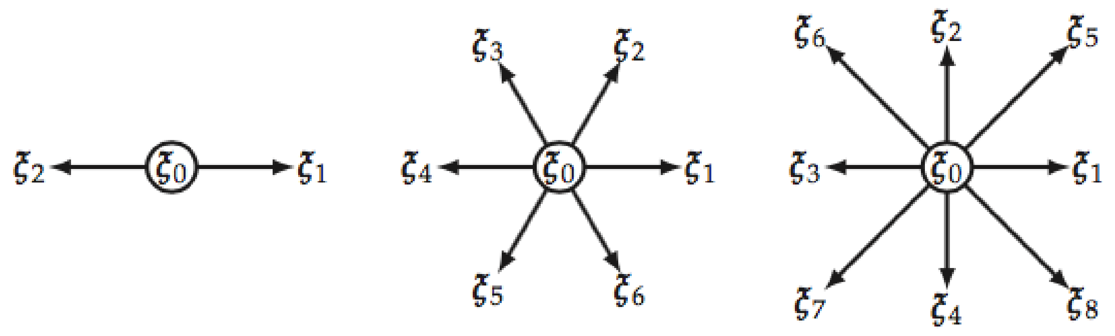


Figure 2.7: Velocity sets in 1-D and 2-D. From left to right: D1Q3, D2Q7, D2Q9.

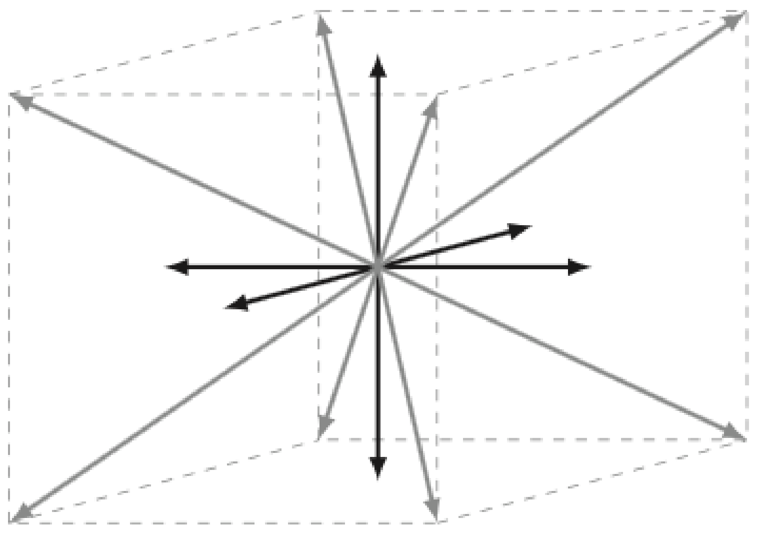


Figure 2.8: D3Q15 velocity set.

2.5.2 Single Relaxation Time BGK

Even though we have already introduced the BGK collision operator, this small section is intended to present it on its own. This is relevant for the project because of the relevance this collision operator has in its outcome.

This collision operator has proven the most popular because of its simplicity. As shown before, the BGK collision operator is given by

$$\Omega_i = -\frac{1}{\tau}[f_i - f_i^{(0)}] \tag{2.48}$$

where τ is a free parameter known as the relaxation time and $f_i^{(0)}$ is the equilibrium distribution of particles. The operator itself represents a relaxation of the distribution function f_i towards the equilibrium value $f_i^{(0)}$.

The BGK operator must preserve both mass and momentum,

$$\sum_i \Omega_i = 0 \tag{2.49}$$

$$\sum_i \mathbf{c}_i \Omega_i = 0 \tag{2.50}$$

As we've already seen, the first order lattice Boltzmann equation with the BGK operator is

$$f_i(\mathbf{x} + \mathbf{c}_i, t+1) - f_i(\mathbf{x}, t) = -\frac{1}{\tau}[f_i(\mathbf{x}, t) - f_i^{(0)}(\mathbf{x}, t)]$$

or, leaving the future particle distribution alone on the left-hand side and grouping terms:

$$f_i(\mathbf{x} + \mathbf{c}_i, t+1) = \left(1 - \frac{1}{\tau}\right) f_i(\mathbf{x}, t) + \frac{1}{\tau} f_i^{(0)}(\mathbf{x}, t)] \tag{2.51}$$

From this we can make some observations. We can see that when $\tau = 1$, the right side of the equation becomes $f_i^{(0)}$, which is the case of complete relaxation, when all traces of $f_i(\mathbf{x}, t)$ are removed and only $f_i^{(0)}$ is propagated. When$\tau > 1$, it is called sub-relaxation, as the particle distribution is not completely relaxed to equilibrium. On the other hand, $\tau < 1$ is called over-relaxation, since the particle distribution is moved beyond equilibrium. τ cannot be too low, since numerical instabilities appear when τ tends to 0.5.

The relaxation time τ can be defined as

$$\tau = \frac{1}{\Omega} = \frac{\nu_{lb}}{c_0^2} + 0.5 = 3\nu_{lb} + 0.5 \tag{2.52}$$

Where c_0^2 is the square speed of sound, which is, for the most part, equal to 1/3.

2.5.3 Multiple Relaxation Time (MRT)

MRT operators evolved out of early experiments with general collision operators. The distribution functions f_i can be represented as a q-dimensional vector f, which can be transformed to another basis. With MRT models, this basis is q hydrodynamic and non-hydrodynamic moments of f_i.

The hydrodynamic moments are typically $\Pi_0 = \rho$, $\Pi_\alpha = \rho\mathbf{u}$, and $\Pi_{\alpha\beta}$, the moments which are relevant for the link to hydrodynamics. The nonhydrodynamic moments, moments of higher order, are not directly relevant for the hydrodynamics behaviour of the model, but must usually be present to fill out the moment basis (Viggen, 2014).

The important thing to note in the MRT method is that the relaxation to equilibrium is performed in the moment basis instead of the f basis, like the BGK operator does. After relaxation, the moments are then transformed back to the f_i basis for streaming. The advantage of relaxing in the moment basis is that different moments can be relaxed at different rates.

The transformation to moment basis is done using the moment transformation matrix M_{ij} in the following manner:

$$\mathbf{Mf} = \mathbf{m}, \qquad \mathbf{Mf}^{(0)} = \mathbf{m}^{(0)} \tag{2.53}$$

Where $\mathbf{m}$ and $\mathbf{m}^{(0)}$ are the resulting moment vectors of the transformation.

The qxq collision matrix $\boldsymbol{\Omega}$ is assumed to be diagonalisable, or, in other words, of the form

$$\boldsymbol{\Omega} = \mathbf{M}^{-1}\mathbf{TM} \tag{2.54}$$

Where the relaxation matrix $\mathbf{T}$ is usually a diagonal matrix.

If we take the generalised lattice Boltzmann equation (with a general collision operator),

$$f_i(\mathbf{x} + \mathbf{c}_i, t+1) - f_i(\mathbf{x}, t) = \sum_j \Omega_{ij}[f_j(\mathbf{x}, t) - f_j^{(0)}(\mathbf{x}, t)] \tag{2.55}$$

and left-multiply it by $\mathbf{M}$, it results in the following relaxation equation in moment space,

$$\mathbf{m}_{\mathsf{out}} = \mathbf{m} + \mathbf{T}(\mathbf{m} - \mathbf{m}^{(0)}) \tag{2.56}$$

Here, $\mathbf{m}_{\mathsf{out}}$ can be seen as the post-collision moment vector, or the moment vector of the particles being streamed out of the node. We see that each element in the diagonal matrix $\mathbf{T}$ is a relaxation time for one of the moments in $\mathbf{m}$.

More generally, the different moments can have different relaxation times. The post-collision moments $\mathbf{m}_{\mathsf{out}}$ can't be propagated directly, and must be converted back to the distribution function basis by $f = \mathbf{M}^{-1}\mathbf{m}$ before streaming. Thus, in principle, the MRT algorithm works by streaming, conversion to moment basis, relaxation of the moments, conversion back to the distribution function basis, and streaming again. In practice, it is more efficient to compute $\mathbf{\Omega}$ directly and perform the relaxation as in eq. 2.55.

The usefulness of MRT lies in setting different relaxation times for the different non-hydrodynamic moments. For a given lattice and a given moment basis, an analysis can be carried out to find the optimal nonhydrodynamic relaxation times to optimise certain aspects of the behaviour of the lattice Boltzmann method. A downside is that such analyses do not give universal results. The results are specific to each velocity set, each choice of moment basis, and each desired optimal behaviour. These analyses can also be difficult both to perform and to comprehend.

This formulation has two important consequences. First, one has the maximum number of adjustable relaxation times, one for each class of kinetic modes. Second, one has maximum freedom in the construction of the equilibrium functions of the non-conserved moments. One immediate result of using this formulation instead of the BGK model is a significant improvement in numerical stability (d'Humières, Ginzburg, Krafczyk, Lallemand, & Luo, 2002). It is worth it to notice that the above procedures are general and are independent of the number of discrete velocities and the number of space dimensions of the system.

This means that the MRT formulation is able to withstand higher Reynolds than the conventional BGK method. This should let us simulate flows with higher turbulence than with just the BGK model, as shown by Lallemand and Luo (2000).

2.5.4 Large Eddy Simulation and Smagorinsky

There exists a fundamental in the effective simulation of turbulent flows. At low values of transport coefficients, very small scale fluid structures can evolve. Such structures will have observable effects on larger scale structures. If a simulation of a turbulent flow is able to solve all the way down to the Kolmogorov-scale (the smallest scales in turbulent flows, where viscosity dominates and the turbulent kinetic energy is dissipated into heat) it is said to be a direct numerical simulation (DNS). The problem then is that at high Reynolds flows, the number of required nodes is very impractical for implementation on computers.

In such situation, one must model the effects of unresolved eddies on the large-scale structures (which are generally what we are interested in). There are a number of ways to capture the effects of the so called sub-grid scales, and one of the main ways is through Large-Eddy Simulations (LES).

The main idea behind LES is to reduce the computational cost of modelling turbulent flows by reducing the range of time and length scales that are being solved through a low-pass filtering of the Navier–Stokes equations. This low-pass filtering, which can be viewed as a time and space averaging, effectively removes small-scale information from the numerical solution. This information is not irrelevant and needs further modeling, a task which is an active area of research for problems in which small-scales can play an important role (Keating, 2011).

The eddy-viscosity models are based on an artificial eddy viscosity approach, one in which the effects of turbulence are lumped into a turbulent viscosity. This approach treats dissipation of kinetic energy at sub-grid scales as analogous to molecular diffusion. In this case, the deviatoric part of τ_{ij} is modeled as:

$$\tau_{ij} - \frac{1}{3}\tau_{ij}\delta_{ij} = -2\nu_t\bar{S}_{ij} \tag{2.57}$$

where ν_t is the turbulent eddy viscosity and $\bar{S}_{ij}$ is the rate-of-strain tensor defined as follows:

$$\bar{S}_{ij} = \frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right) \tag{2.58}$$

The first sub-grid scale model developed was the Smagorinsky model. It models the eddy viscosity as:

$$\nu_t = (C_s\Delta_g)^2\,|S| \tag{2.59}$$

where $\Delta_g = (\text{Volume})^{\frac{1}{3}}$ is the grid size, C_s is the Smagorinsky constant which usually has a value between 0.1 and 0.2, and $\bar{S} = \sqrt{2S_{ij}S_{ij}}$.

2.5.5 Boundary Conditions

In the present section we will only present three types of boundary conditions. This due to the fact that these boundary conditions presented are the ones used by the model that was developed. In general, there's a great flexibility in applying boundary conditions in LBM. In fact, the ability to easily incorporate complex solid boundaries is one of the most exciting aspects of these models and has made it possible to simulate realistic porous media, for example.

Periodic Boundaries

The simplest boundary conditions are periodic in that the edges are treated as if they are attached to opposite edges. Most early papers in the LBM used these boundaries along with bounceback boundaries. In simulating flow in a slit for example, bounceback boundaries would be applied at the slit walls and periodic boundaries would be applied to the 'open' ends of the slit.

Fully periodic boundaries are also useful in some cases (for example, simulation of an infinite domain of multiphase fluids). For boundary nodes, neighboring points are on the opposite boundary (Sukop & Daniel T. Thorne, 2005). Coding this type of boundaries is pretty trivial. The only thing that needs to be done is make the boundary nodes in one edge neighbors to the boundary nodes in their opposite edge.

Bounceback Boundaries

Bounceback boundaries are also simple and have played a major role in making LBM popular among modelers interested in simulating fluids in domains characterized by complex geometries such as those found in porous media. Their beauty lies in that one simply needs to designate a particular node as a solid obstacle and no special programming treatment is required after that. Thus, it is trivial to incorporate images of porous media for example and immediately compute the flow in them.

So, solids are separated into two types – boundary solids that lie at the solid-fluid interface and isolated solids that do not contact fluid. With this division it is possible to eliminate unnecessary computations at inactive nodes; this can be particularly important in the simulation of fluid flow in fractured media for example, where the fraction of the total domain occupied by open space accessible to fluids can be very small.

An illustration of how the bounceback boundaries work can be seen in figure 2.9. Code to accomplish this can be included within the collision routine. While traversing the lattice to perform the collision step, if a node is determined to be a boundary solid, the normal collision computation is omitted and instead, the densities are bounced back in the same direction (but negative) they came in. The next streaming step moves the densities back into the fluid domain.

Dirichlet Boundaries

These type of boundaries specify the values that a solution needs to take along the boundary of the domain. For example, one could specify the values of velocity in the boundaries of the problem.

To be able to get the directional densities after specifying a velocity, for example, one needs to set up a system of independent equations. In the most common case, when setting up a specific velocity **u**, one has the macroscopic density formula:

$$\rho = \sum_i f_i$$

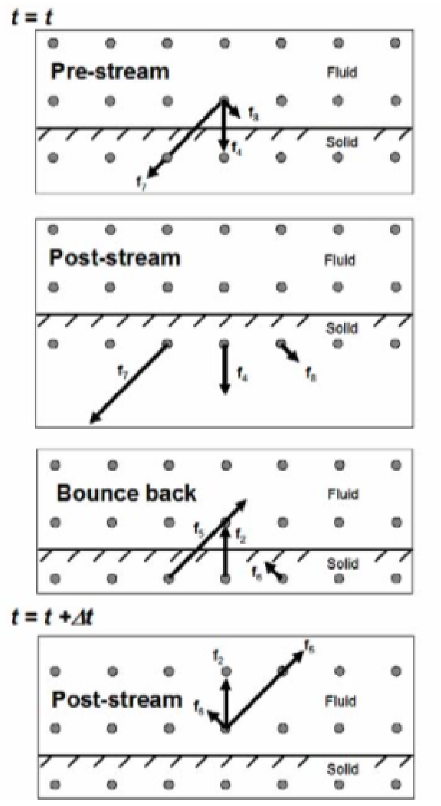


Figure 2.9: Bounceback boundaries. Densities get to the node and are bounced back during the collision step.

Also, by considering the individual directional densities that can contribute to each of the coordinates, one can have either one, two, or three, equations (depending on the dimension of the lattice) from the formula for macroscopic velocity:

$$\mathbf{u} = \frac{1}{\rho}\sum_i f_i\mathbf{c}_i$$

An extra equation could be written by assuming that the bounceback condition, for example, holds in the direction normal to the boundary. This system of equations can then be solved to determine the directional densities of the boundary nodes.

2.5.6 Instability in BGK

Stability in the lattice Boltzmann methods is an important problem. Certain characteristics of the conventional BGK model make it inaccurate for turbulent flows or flows of a relatively high Reynolds number. This has led to the development of techniques that attempt to improve the stability of the models, such as the Multiple Relaxation Time method and the Smagorinsky model (sections 2.5.3 and 2.5.4, respectively).

Even though the mechanism of the lattice Boltzmann method's instability is not totally understood, it is normally attributed to the occurrence of unphysical negative distribution functions, the interplay between acoustic modes and other modes in a low viscosity regime, improper treatments of boundary conditions, etc. (Cheng & Zhang, 2011).

One clear problem of the conventional BGK model is the inefficiency in low viscosity regimes. To simulate high Reynolds number flows in finite lattice resolutions, we need to reduce the viscosity to as low a value as possible. But in low viscosity conditions, instability frequently occurs. Therefore, any means that can reduce viscosity without introducing instability and additional error is valuable.

The overrelaxation discretization of the BGK model allows one to choose a time step that can be larger than the relaxation time. This decouples viscosity from the time step, which suggests that the BGK model is capable of operating at arbitrarily high Reynolds numbers by making the relaxation time sufficiently small. However, in this low viscosity regime, the BGK model suffers from numerical instabilities which readily manifest themselves as local blowups and crazy oscillations (Brownlee, Gorban, & Levesley, 2007).

Another way to put it is that in high Reynolds problems a low viscosity regime is set on the problem. This translates to a very low relaxation time in the BGK model, τ, which, as it gets lower and lower, or closer and closer to its inferior bound, 0.5, sharp gradients tend to develop at both the macro and mesoscopic levels, meaning the onset of numerical instabilities (Keating, 2011).

Another barrier to the stability of certain LB schemes is the presence of parasitic oscillations in the vicinity of sharp gradients in the flow. These may be caused by shock waves travelling through the fluid as well as thin shear layers or singularities in boundary conditions. The MRT operation can be used to delete the contribution to these oscillations made by the higher order modes, by equilibrating them. The cost of this is a fixed increase in the dissipation coefficients in the post Navier-Stokes dynamics (Gorban & Packwood, 2012).

It is clear that more research should go into the area of instabilities in the lattice Boltzmann models. A clear answer to the mechanism of these instabilities is not present in the literature studied.

2.5.7 Lattice Boltzmann Units

Latt (2008) does an excellent job at explaining the units in the Lattice Boltzmann sphere. He says Lattice Boltzmann simulations are supposed to represent the physics of an actually existing, real system. Two constraints determine the choice of units. First, the simulation should be equivalent, in a well defined sense, to the physical system. Second, the parameters should be fine-tuned in order to reach the required accuracy, i.e. the grid should be sufficiently resolved, the discrete time step sufficiently small, etc.

In an incompressible fluid, the density takes a constant value $\rho = \rho_0$ which does not vary in time and space. The equations of motion, the Navier-Stokes equations, are governed by the laws for mass and momentum conservation. The conservation law for mass states that the velocity field is divergence-less:

$$\nabla_p \cdot u_p = 0 \tag{2.60}$$

Here, the index p indicates indicates that variables and derivatives are evaluated in physical units. The conservation law for momentum leads to the following relation (note that this is in index notation):

$$\partial_{t_p} u_p + (u_p \cdot \nabla_p) u_p = -\frac{1}{\rho_{0_p}} \nabla_p p_p + \nu_p \nabla_p^2 u_p \tag{2.61}$$

where p_p is the pressure and ν_p the kinematic viscosity, both in physical units due to the subscript p.

Dimensionless formulation

These two equations are now cast into a dimensionless form. For this, a length scale l_0 and a time scale t_0 are introduced which are representative for the flow configuration. The length l_0 could for example stand for the size of an obstacle which is immersed in the fluid, and t_0 could be the time needed by a passive scalar in the fluid to travel a distance l_0. The physical variables such as the time t_p and the position vector r_p, are replaced by their dimensionless counterpart:

$$t_d = \frac{t_p}{t_{0,p}} \qquad r_d = \frac{r_p}{r_{0,p}} \tag{2.62}$$

Here, the subscript d stands for dimensionless. In the same manner, a unit conversion is introduced for other variables, based on a dimensional analysis:

$$u_p = \frac{l_{0,p}}{t_{0,p}} u_d, \qquad \partial_{t_p} = \frac{1}{t_{0,p}} \partial_{t_d}, \qquad \nabla_p = \frac{1}{l_{0,p}} \nabla_d, \qquad p_p = \rho_0 \frac{l_{0,p}^2}{t_{0,p}^2} p_d \tag{2.63}$$

Putting these equations into the equations 2.60 and 2.61 leads to the dimensionless version of the Navier-Stokes equations:

$$\nabla_d \cdot u_d = 0 \tag{2.64}$$

$$\partial_{t_d} u_d + (u_d \cdot \nabla_d) u_d = -\nabla_d p_d + \frac{1}{Re} \nabla_d^2 u_d \tag{2.65}$$

where the dimensionless Reynolds number is defined as

$$Re = \frac{l_0^2}{t_0 \nu} \tag{2.66}$$

It is important to note that two flows that obey the Navier-Stokes equations are equivalent if they are put in the same geometry and have the same Reynolds number.

Discretization of the dimensionless system

The discrete space interval δ_x is defined as the reference length divided by the number of cells N used to discretize this length. In the same way, δ_t is defined as the reference time divided by the number of iteration steps N_{iter} needed to reach this time. Recall that both reference variables are unity in the dimensionless system. Thus, the discretization parameters are

$$\delta_x = 1/N \qquad \delta_t = 1/N_{iter} \tag{2.67}$$

Other variables, such as velocity and viscosity, are easily converted between the dimensionless and lattice Boltzmann units through a dimensionless analysis:

$$u_d = \frac{\delta_x}{\delta_t} u_{lb} \qquad \nu_d = \frac{1}{Re} = \frac{\delta_x^2}{\delta_t} \nu_{lb} \tag{2.68}$$

And from this we can get

$$u_{lb} = \frac{\delta_t}{\delta_x} u_d \qquad \nu_{lb} = \frac{\delta_t}{\delta_x^2} \frac{1}{Re} \tag{2.69}$$

Finally, defining the reference velocity $u_0 = l_0/t_0$, one finds that

$$u_{0,d} = 1 \qquad u_{0,lb} = \frac{\delta_t}{\delta_x} \tag{2.70}$$

Density can also be represented in lattice units, for instance with the rest state density normalised to $p_{0,lb} = 1$. δ_x and δ_t are clearly insufficient to convert between lattice and physical density as their units are metres and seconds, respectively.

Converting density to physical units requires units of kilograms. We therefore define the density conversion factor C_p so that

$$\rho_p = C_\rho \rho_{lb} \tag{2.71}$$

where the conversion factor has units of kg/m^3. This conversion factor can be set arbitrarily in order to have a $\rho_{lb} = 1$.

2.6 Drag

Since the project involves calculating the drag coefficient around a sphere, we present in this small section the basics of drag and drag around a sphere.

The aerodynamic drag on an object depends on several factors, including the shape, size, inclination, and flow conditions. All of these factors are related to the value of the drag through the drag equation:

$$F_d = \frac{1}{2} C_d \rho u^2 A \tag{2.72}$$

Where F_d is the drag force, which opposes motion, C_d is the drag coefficient, u the velocity of the fluid, and A is the reference area of the object in question (NASA, 2015). This reference area is usually the effective area that faces the flow of the fluid in question. This reference area, for example, is equal to a circle when the object being studied/measured is a sphere.

The drag coefficient is a dimensionless number that characterizes all of the complex factors that affect drag. The drag coefficient is usually determined experimentally using a model in a wind tunnel. In the tunnel, the velocity, density, and size of the model are known. Measuring the drag then determines the value of the drag coefficient as given by the above equation.

The drag coefficient and the drag equation can then be used to determine the drag on a similar shaped object at different flow conditions as long as several flow similarity parameters are matched.

Lattice Boltzmann drag on a sphere

Using equation 2.72 and what we've learned from the lattice Boltzmann units in the previous section, we can obtain an expression for the drag coefficient in lattice (lb) units for a sphere:

$$C_{D,lb} = \frac{2F_{D,lb}}{u_{lb}^2 \pi r_{lb}^2} \tag{2.73}$$

where r_{lb} is the radius of the sphere and πr_{lb}^2 is the reference area of the object.

Drag on a sphere

The correlation for drag coefficient in uniform flow around a sphere is a staple of fluid flow calculations and fluid mechanics education. Engineers use the drag coefficients from this chart to calculate pressure drops and flow rates for flows around spheres, including settling and ballistics flows. A correlation is presented that captures the drag coefficient versus Reynolds number for all values of Reynolds number (creeping flow, recirculating, and turbulent) in equation (Morrison, 2013), which can be seen graphed in figure 2.10. The equation is a heuristic approximation of experimental results.

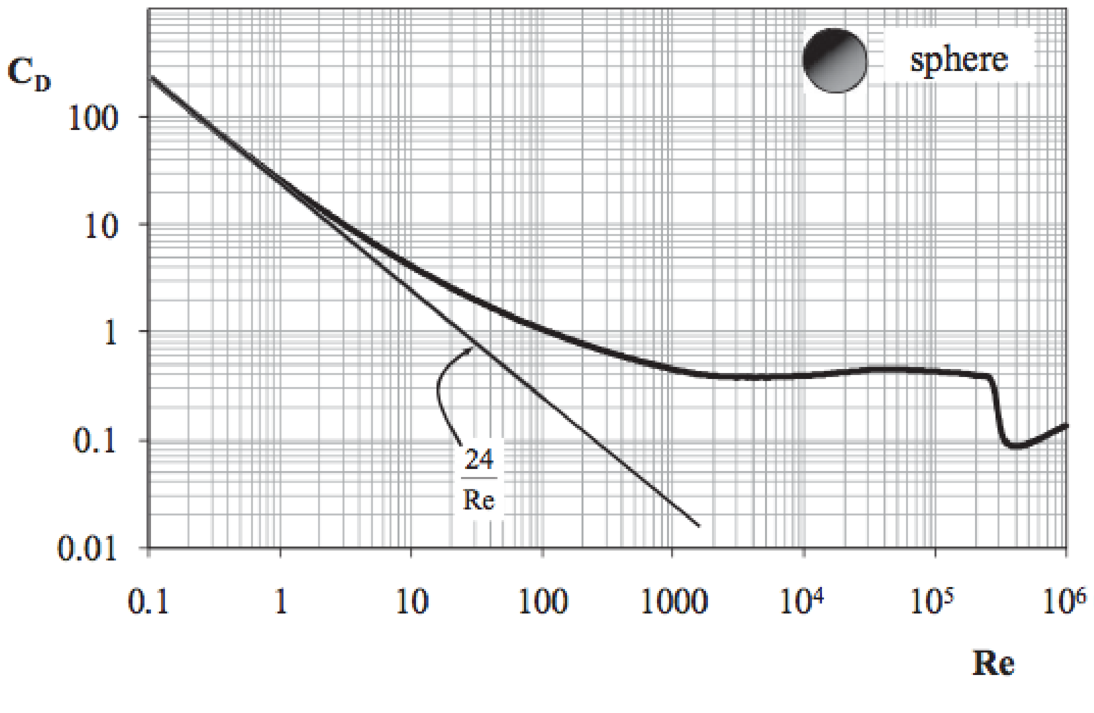


Figure 2.10: Drag coefficient on a sphere as a function of the Reynolds number.

This drag as a function of the Reynolds number is the purpose of study in this project. We want to see how close the lattice Boltzmann method comes to this graph.

For clarity, we present the equation of the drag on a sphere as a function of the

Reynolds number present in figure 2.10:

$$C_D = \frac{24}{Re} + \frac{2.6\left(\frac{Re}{5}\right)}{1+\left(\frac{Re}{5}\right)^{1.52}} + \frac{0.411\left(\frac{Re}{263000}\right)^{-7.94}}{1+\left(\frac{Re}{263000}\right)^{-8}} + \left(\frac{Re^{0.8}}{461000}\right) \tag{2.74}$$

With this equation it is possible to compare the results obtained from the lattice Boltzmann method and the experimental, real world results.

3 Methodology

In this section the methodology of the research project is shown. In other words, here we will show how the results were obtained. In the previous chapter we laid out all of the relevant theory, all of the background knowledge needed to achieve the goals stated, and in this one we will refer to how the results are computed, which model was used, how the data was processed, etc.

3.1 Palabos

The first thing to notice is that we didn't start our own lattice Boltzmann code from scratch. Instead of developing a simple code by ourselves, we delved into the open source world of Palabos.

The library Palabos offers a framework in C++ for fluid flow simulations with the LB method. Originally conceived as a research tool for lattice Boltzmann models, the code has evolved into a general-purpose software for computational fluid dynamics. The programming interface is straightforward and offers an access to the rich world of lattice Boltzmann, even to and audience with restricted theoretical knowledge of this method. It is basically a C++ solver with wide-spread use in the lattice Boltzmann community. It covers a large range of applications and specializes on multi-phase flow, flow through porous media, and complex flow with chemical reactions.

A substantial amount of development time for Palabos went into formulating a general programming concept for LB simulation which offers an appropriate balance between generality, ease of use, and numerical efficiency. The difficulty of this task can be understood if one considers that the core ingredients of the LB method are guided by physical considerations rather than criteria from numerical analysis. As a consequence, it is straightforward to implement a LB code for a specific physical model, a rectangular-shaped numerical grid, and simple boundary conditions. However, any of the extensions of the code required for the implementation of practical problems in fluid engineering require careful considerations and a solid amount of programming efforts. It is frequent to find LB codes which implement one specific advanced feature (for example parallelism), but then find themselves in a too rigid shape to allow other extensions (for example coupling of two grids for multi-phase flows).

A central concept in Palabos are so-called "dynamics objects" which are associated to each fluid cell and determine the nature of the local collision step. Full locality of inter-particle collisions is a key ingredient of LB models, a fact which is acknowledged in Palabos by promoting the raw numerical efficiency of such model. Specific data structures are also available for non-local ingredients, for implementing for example specific boundary conditions or inter-particle forces in multi-phase models.

The grid structure of the simulations is based on a multi-block approach. Each block behaves like a rudimentary LB implementation, while advanced software ingredients such as parallelism and sparse domain implementations are covered through specific interface couplings between blocks. The C++ code of Palabos makes a massive use of its generic value in its many facets. Basically, generic programming is intended to offer a single code that can serve many purposes. On one hand, the code implements dynamic genericity through the use of object-oriented interfaces. One use of this is that the behavior of lattice sites can be modified during program execution, to distinguish for example between bulk and boundary cells, or to modify the fluid viscosity or the value of a body force dynamically. On the other hand, C++ templates are used to achieve static genericity. As a result, it is sufficient to write a single generic code for various 3D lattice structures, such as D3Q15, D3Q19, and D3Q27.

Its intended audience covers scientists and engineers with a background in LB modeling, or at least a solid background in computational fluid dynamics and a basic knowledge of lattice Boltzmann. The software is freely available. It is distributed under the hope that it will promote research in the field of LB modeling, and help researchers concentrate on actual physical problems instead of staying stuck in tedious software development. Furthermore, implementing new LB models in Palabos offers a simple means of promoting new models and exchanging the information between research groups.

The Palabos project has been around for a while and its greatest advantage is probably the native parallelization it offers. It can, automatically, break the domain of your model into the amount of core resources you have available. This means that if you have 1000 processing cores available, your simulation will run considerably faster than if you had only 1 processor. This lets researches run better, faster, and more accurate simulations than ever before, and at no additional cost to them or their institutions.

For more information on the Palabos project, visit their site at www.palabos.org.

3.2 Model

Now we present the model used for the research project. All of the code developed can be found in the appendix. Since the full code used was developed in the Palabos language, there's no point in presenting it and explaining it in detail. Ultimately, the theory discussed earlier is what's behind the model.

Remembering that the objective was to graph the drag coefficient as a function of the Reynolds number, the model used varied according to the Reynolds number, but many parameters stayed the same across all Reynolds. In the following section we explain which parameters were constant for all the models.

The essential model is that of a sphere inside a rectangular box; the box has an inlet and an outlet, and the length of the box is big enough so that the wake produced by the fluid flow around the sphere can stabilize.

3.2.1 Domain

The domain consists of a box of dimensions $2.95\,m \times 0.9\,m \times 0.9\,m$. This longer distance was placed in order to give enough space for the wake behind the sphere to stabilize before hitting any boundary. This box was discretized by 69 nodes on the shorter length, giving rise to a discrete space step of $\delta_x = 0.9\,m/69 = 0.013\,m$. As mentioned earlier, the time step depends on the Reynolds of the system, making it vary with each different inlet velocity.

This discretization means that the dimensions of the box in lattice units is $227 \times 69 \times 69$, which means there are 1.08×10^6 nodes in the whole lattice domain. Now, not all of these nodes were part of the simulation; the sphere takes some space in the domain.

The sphere was imported from an stl file, it had a diameter of $0.1\,m$, and was placed at a slight nudge from the center of the box at $(0.45\,m, 0.455\,m, 0.455\,m)$. This means that the sphere had a diameter of 8 lattice units. Another hard parameter in these simulations is the lattice velocity u_{lb} which was set to be $u_{lb} = 0.04\,lu/ts$ (lu refers to lattice units, and ts to time steps) according to lattice Boltzmann guidelines. The velocity set chosen was the $D3Q19$ (shown in figure 3.1).

Another important parameter is the kinematic viscosity ν of the system. This variable was chose to be equal to 0.001. A summary of all of this information can be seen in table 3.1.

With all these parameters it is possible to calculate the Reynolds number, according

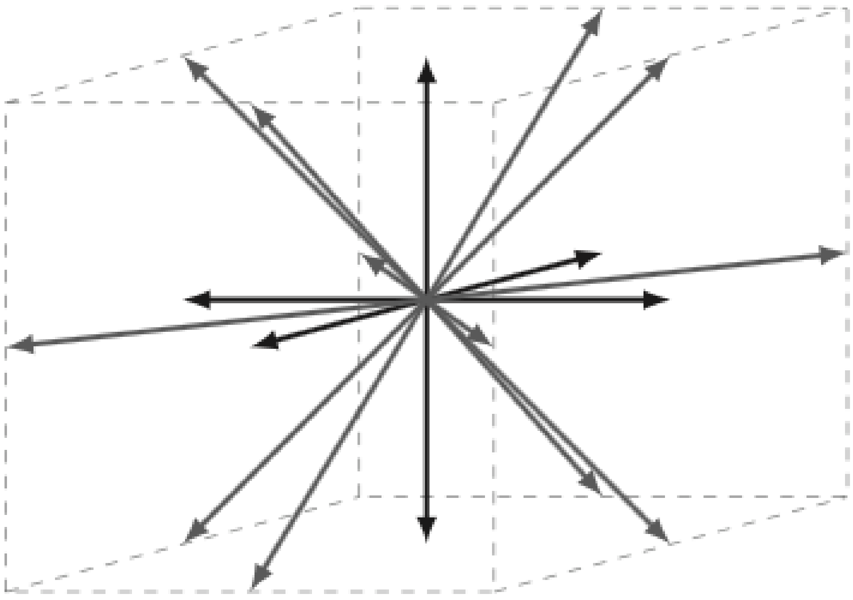
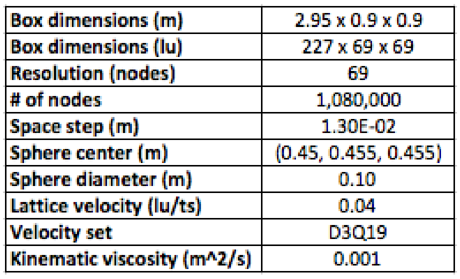


Figure 3.1: D3Q19 velocity set.



Box dimensions (m)	2.95 x 0.9 x 0.9
Box dimensions (lu)	227 x 69 x 69
Resolution (nodes)	69
# of nodes	1,080,000
Space step (m)	1.30E-02
Sphere center (m)	(0.45, 0.455, 0.455)
Sphere diameter (m)	0.10
Lattice velocity (lu/ts)	0.04
Velocity set	D3Q19
Kinematic viscosity (m^2/s)	0.001

Table 3.1: Simulation parameters.

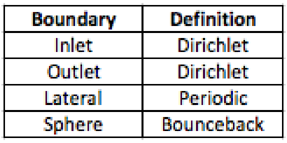
to the equation

$$Re = uL/\nu \tag{3.1}$$

Where L refers the the characteristic length of the system, which in this case is the diameter of the sphere, ν is the kinematic viscosity, and u is the velocity of the fluid in the system. Here we can see that the Reynolds number depends on the velocity we input into the system, since the other variables are fixed. In this way, we can control the Reynolds number of the system by controlling the speed in which the fluid enters the system.

3.2.2 Boundary Conditions

As for the boundary conditions, there are seven entities to which attribute boundary conditions: the inlet, the outlet, four lateral boundaries, and the sphere. The types of boundaries for each can be seen in table 3.2. A simple schematic of the model can be seen in figure 3.2.



Boundary	Definition
Inlet	Dirichlet
Outlet	Dirichlet
Lateral	Periodic
Sphere	Bounceback

Table 3.2: Boundary definitions of the model.

The four lateral boundaries have a simple definition as periodic. This because after tinkering with the model we found that this was more stable than the bounceback or free slip counterparts and doesn't affect the outcome of the model in any negative way.

The sphere is defined as a bounceback boundary. Particles need to bounce back from this obstacle in order to create a computational model of a physical thing in that spot.

The inlet and outlet are both defined as Dirichlet boundaries. The inlet has a Dirichlet boundary condition on velocity, which is set and increasing every time step in a continuous way, according to the sine function, until it reaches the target velocity for a given Reynolds number. On the other hand, the outlet has a Dirichlet boundary condition on the density (which could translate to pressure because of the equation of state). It sets an outside density of 1.0 so as to simulate a constant equilibrium pressure outside of the domain to which the flow has to comply.

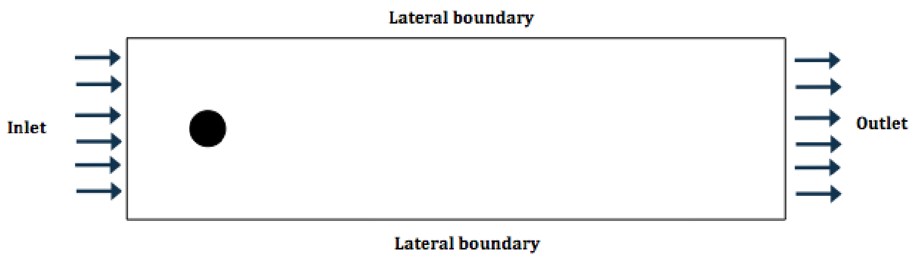


Figure 3.2: Schematic of the model. This simple 2-D schematic from a slice on the y or z axis shows the basics of the model: an inlet near the sphere, lateral boundaries and an outlet.

3.3 Simulations

Now that we know the parameters of our model let's see what the procedure for the simulations and the capture of the drag coefficient is.

First, the drag coefficient was computed and logged into a data file after every certain number of time steps and according to equation 2.73:

$$C_{D,lb} = \frac{2F_{D,lb}}{u_{lb}^2 \pi r_{lb}^2}$$

The palabos code let us calculate the force on the sphere in lattice units (which is nothing but the sum of the change in momenta around the sphere) and then we only had to plug that value into equation 2.73 and export it to a data file. The variable u_{lb} was already established and equal to $0.04\, lu/ts$ and the sphere radius in lattice units r_{lb} can be calculated with the use of the space step amount and is equal to $4\, lu$.

After running the simulation for a specific Reynolds number, the drag coefficient would be computed and written into a data file, and after a couple of hours of simulation we would observe the drag coefficient data to check if the simulation had converged appropriately. We would look for a response such as the one shown in figure 3.3, where it is clear that the simulation has run with no problems whatsoever and then write the converged drag coefficient, as well as the Reynolds it corresponds to, in order build the graph that we are looking for (drag coefficient vs. Reynolds).

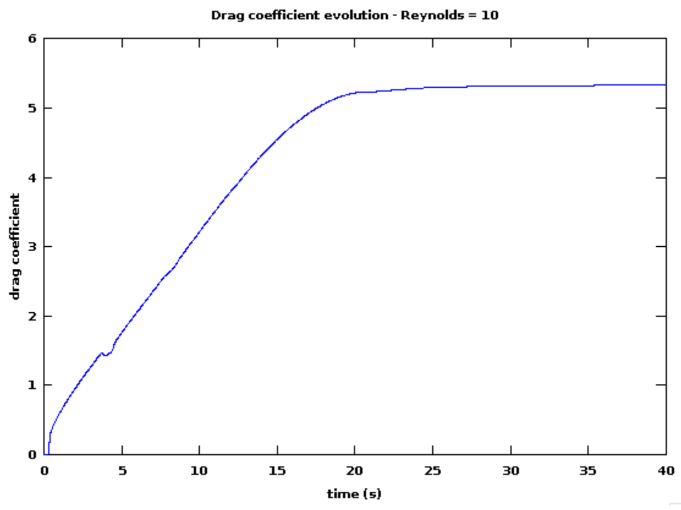
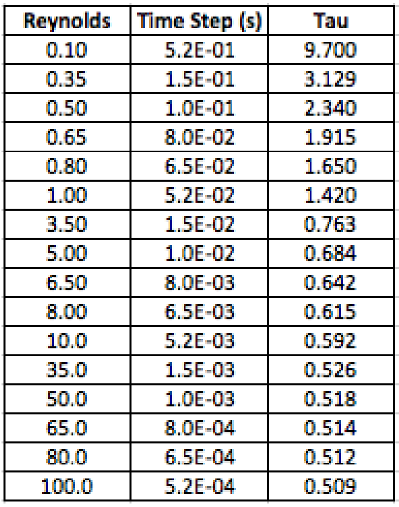


Figure 3.3: Evolution of the drag coefficient during a simulation. This example shows the drag coefficient converging on a value of about 5.3 for a Reynolds of 10 with the BGK collision operator.

Since our resources were limited we had to decide for which Reynolds we would compute the drag coefficient. The Reynolds' numbers chosen are shown in table 3.3. This table also shows the value of the relaxation time τ, since this important value depends ultimately on the Reynolds number; it also shows the value of the time step as a function of Reynolds. The Reynolds numbers were chosen on the base of a logarithmic scale, such as the one used for the drag coefficient on a sphere of figure 2.10.

Finally, we ran the simulations with these parameters for the three methods already mentioned: BGK, Smagorinsky, and MRT.



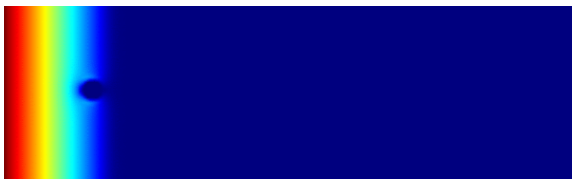
Reynolds	Time Step (s)	Tau
0.10	5.2E-01	9.700
0.35	1.5E-01	3.129
0.50	1.0E-01	2.340
0.65	8.0E-02	1.915
0.80	6.5E-02	1.650
1.00	5.2E-02	1.420
3.50	1.5E-02	0.763
5.00	1.0E-02	0.684
6.50	8.0E-03	0.642
8.00	6.5E-03	0.615
10.0	5.2E-03	0.592
35.0	1.5E-03	0.526
50.0	1.0E-03	0.518
65.0	8.0E-04	0.514
80.0	6.5E-04	0.512
100.0	5.2E-04	0.509

Table 3.3: Reynolds numbers chosen and the corresponding time parameters.

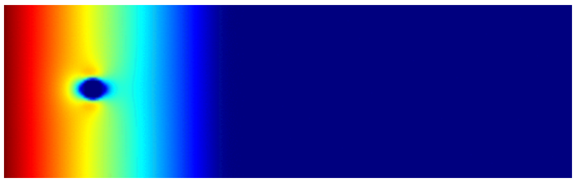
4 Results

The first results obtained were the velocity fields of entire simulation runs. This means that the models developed and the simulations ran were a success. This validates the exercise that is being made and supports further results from this model. A time sequence of the velocity field of a slice of the model, that cuts through the sphere, is shown in figure 4.1.

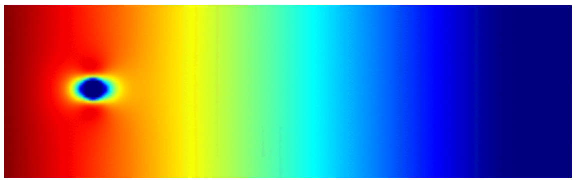
In the figure 4.2 we show the results for the main goal of the simulations: a graph of the drag coefficient as a function of the Reynolds number. In it the results of the three lattice Boltzmann methods are plotted (BGK, MRT, Smagorinsky), as well as the experimental drag coefficient (blue), according to the experimental drag: equation 2.74.



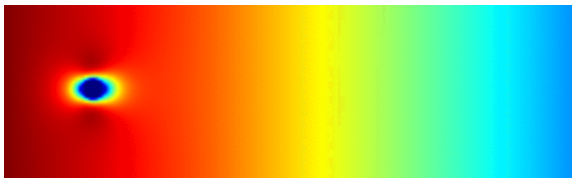
(a) t = 0.4 s



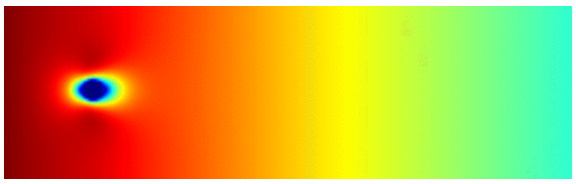
(b) t = 0.8 s



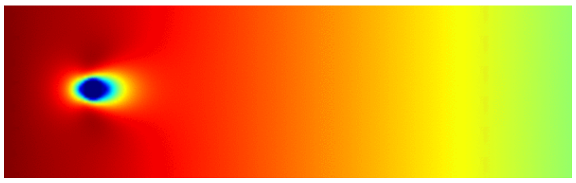
(c) t = 1.8 s



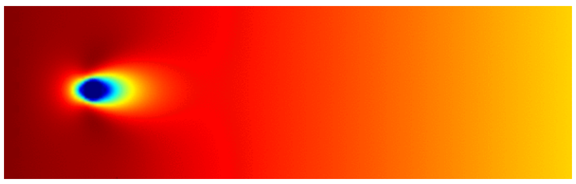
(d) t = 3.4 s



(e) t = 5.2 s



(f) t = 6.9 s



(g) t = 10.3 s

Figure 4.1: Time sequence of a z slice of the domain for a Reynolds number of 10.

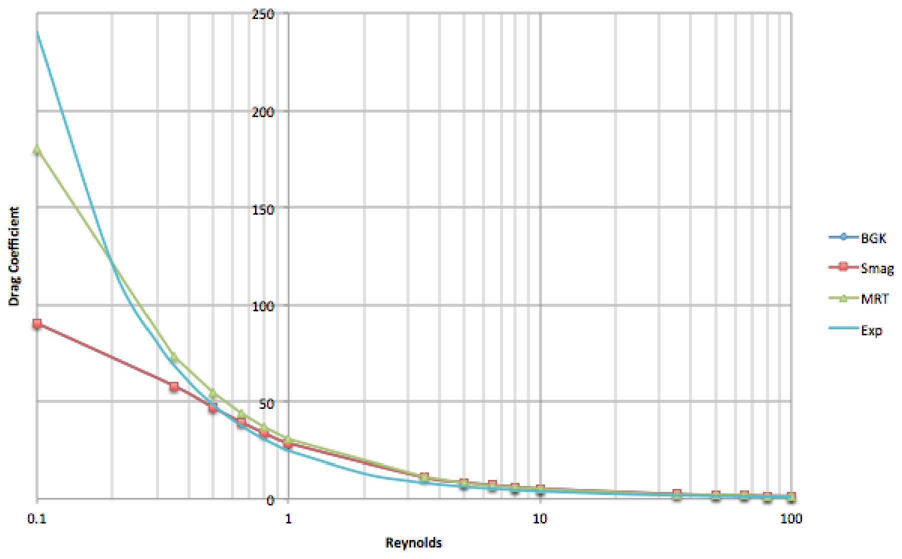


Figure 4.2: Drag Coefficient as a function of Reynolds for the three LB techniques.

5 Analysis

By looking at figure 4.2 one can start by noting how the simulations agree, for a large part, with reality. It seems that the drag simulations have been validated, at least on certain Reynolds number ranges. One can also note superficially how the MRT method yields better results (its data seems to be more in line with the experimental data) than the other two methods. It also appears to be that the BGK method and the Smagorinsky one yield practically the same results.

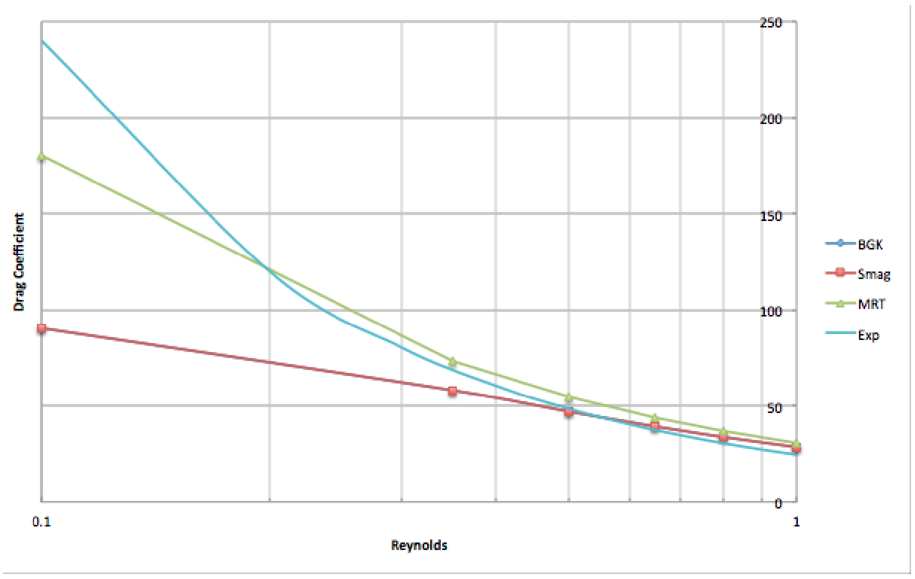


Figure 5.1: Drag Coefficient as a function of Reynolds for the three LB techniques - Reynolds from 0.1 to 1.

In order to analyse the data in a better way visually, we break figure 4.2 into two. In figure 5.1 we can see the same plots but now from a Reynolds value of 0.1 to 1. Here, it is very clear how MRT agrees more with the experimental data than the other methods, and how its curve has a similar shape to that of the real world's, while the BGK and Smagorinsky methods have a linear behavior that's not in agreement with reality.

The reason for this difference between the results of the MRT simulations and the other two methods may be because of its inclusion of higher order moments. As espoused in sections 2.5.3 and 2.5.6, it is possible that there is a presence of parasitic oscillations in the vicinity of sharp gradients in the flow. The MRT operation can be used to delete the contribution to these oscillations made by the higher order modes, by equilibrating them. Another reason for this discrepancy between methods could

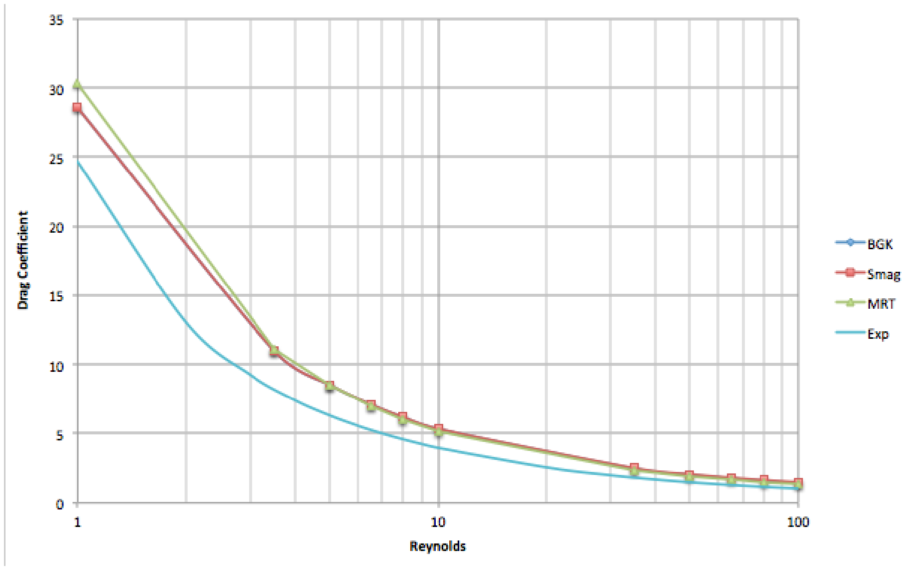


Figure 5.2: Drag Coefficient as a function of Reynolds for the three LB techniques - Reynolds from 1 to 100.

be the significant improvement in numerical stability brought by the MRT method, as mentioned by d'Humières et al. (2002) and demonstrated by Lallemand and Luo (2000).

Figure 5.2 shows some interesting results. It seems to be that the higher the Reynolds number, the higher the convergence between all the models with the experimental data. The MRT model also seems to converge with the experimental data at a faster rate than the other two methods: one can notice it separating from the other two methods and getting closer to the experimental data. All in all, the BGK model and the Smagorinsky show no significant difference between them; one might hypothesize that their data is statistically indistinguishable from each others.

5.1 Root Mean Square Error

In order to evaluate which method offers the best results and to see how much the simulated data, in general, differs from the experimental data, a root mean square error analysis is performed.

The root mean square error of a series of data is defined as

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}} \tag{5.1}$$

Where $\hat{y}_i$ is the predicted or expected value, while y_i refers to the observed data; n is, of course, the amount of data points.

Applying this equation to the data collected, we get the results of table 5.1. Here we can clearly see that our suspicions were true. The method that is closest to the observed drag coefficient is the MRT, and the other two methods, BGK and Smagorinsky, are virtually indistinguishable from each other.

Method	RMSE
BGK	37.59
Smagorinsky	37.57
MRT	15.36

Table 5.1: Root mean square error for the techniques used.

5.2 Resolution Sensitivity

In order to perform a sensitivity analysis the simulations for the three different techniques had to be run for the same Reynolds number, but across a resolution spectrum. The Reynolds number chosen was 80 and after running the simulations for each different resolution, the results obtained are shown in table 5.2 and figure 5.3.

Resolution	# Nodes	BGK	Smagorinsky	MRT	Experimental
60	6.8E+05	1.674	1.742	1.609	1.159
80	1.6E+06	1.571	1.612	1.500	1.159
100	3.2E+06	1.541	1.570	1.470	1.159
120	5.5E+06	1.488	1.509	1.422	1.159

Table 5.2: Resolution analysis. Drag coefficient for the different techniques and resolutions with a Reynolds number of 80.

From this data it is possible to conclude that the more resolution one is able to run, the better the results. It is clear how, as the amount of nodes increases, the drag coefficient gets closer to the real value. This means that if the precision of the drag answer when simulating is important, there needs to be a considerable investment of resources into the resolution of the model. Additionally, we can see how the BGK technique yields slightly better results than the Smagorinsky model, unlike the original results where there was no perceived differences.

5.3 Periodicity

It came to our attention that the periodicity in the lateral boundaries could have distorted the model. To verify that the application of this type of boundaries on the lateral walls didn't affect the model in a negative or significant way, we ran the models with wider boxes, which means that the space between the lateral walls is

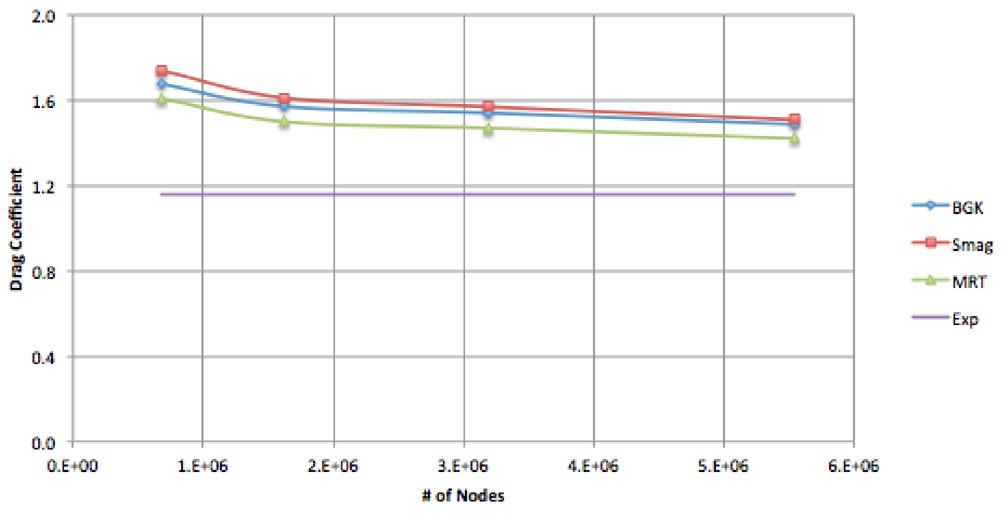


Figure 5.3: Graphed resolution analysis. Drag Coefficient for the different techniques and resolutions with a chosen Reynolds of 80. The experimental data can be observed as a straight line.

bigger. By running the simulations with wider boxes and comparing those results with the ones obtained in section 4, we can make sure that the main simulations were not affected by the periodic boundaries or the distance between the lateral boundaries and the obstacle.

In this way, five simulations for each of the three different techniques were run again with the parameters shown in table 5.3. All the other parameters were the same as those mentioned in table 3.1.

Along with this widening of the boxes must come an increase of the resolution. Without this increase in the resolution there would be a bigger δ_x, which means less nodes per length of the obstacle, which would ultimately lead to a comparative decrease in the precision of the drag calculation in the models (as shown in section 5.2). To get around this it is necessary to increase the resolution of the models to maintain the δ_x (space step) of $1.3 \times 10^{-2}\,m$ used for the main simulations and shown in table 3.1. These increased resolution values, with their respective space steps, for the analysis simulations are shown in table 5.4. This will let us compare adequately the drag of the main model and of these simulations.

Simulation #	lx (m)	ly (m)	lz (m)
1	2.95	1.0	1.0
2	2.95	1.1	1.1
3	2.95	1.2	1.2
4	2.95	1.3	1.3
5	2.95	1.4	1.4

Table 5.3: Physical dimensions parameters for the periodicity analysis. l_x, l_y, and l_z are the physical dimensions of the box in the x, y and z directions, respectively. The lengths l_y and l_z are equal and represent the characteristic length for the definition of the space step.

Simulation #	Characteristic Length (m)	Resolution	Space Step (m)
1	1.0	78	1.30E-02
2	1.1	85	1.31E-02
3	1.2	93	1.30E-02
4	1.3	101	1.30E-02
5	1.4	108	1.31E-02

Table 5.4: Resolution and space step parameters for the periodicity analysis.

The results are shown in figures 5.4. This figure shows the drag coefficient for the periodicity analysis simulations as a function of the characteristic length. It can be seen how the drag coefficient behaves in a cyclic pattern around a mean as the characteristic length increases. This means that the characteristic length used in the main simulations does not distort the model created.

Since the drag coefficient results behaved in a cyclic manner in figure 5.4, this led us to believe that the variation on the drag coefficient depended on the space step. This was clear in the way the space step was calculated for these simulations: the space step of the main simulations was targeted, but since the resolution of the models are integers there is bound to be a rounding error; the cyclic pattern of the drag coefficient results from these rounding errors in the space step. To verify this, we plotted the drag coefficient of the models simulated as a function of the space step in figure 5.5. This figure clearly shows how the space step is responsible for the cyclic pattern observed in figure 5.4. A higher space step means a worse resolution and a higher calculated drag coefficient, as shown in section 5.2.

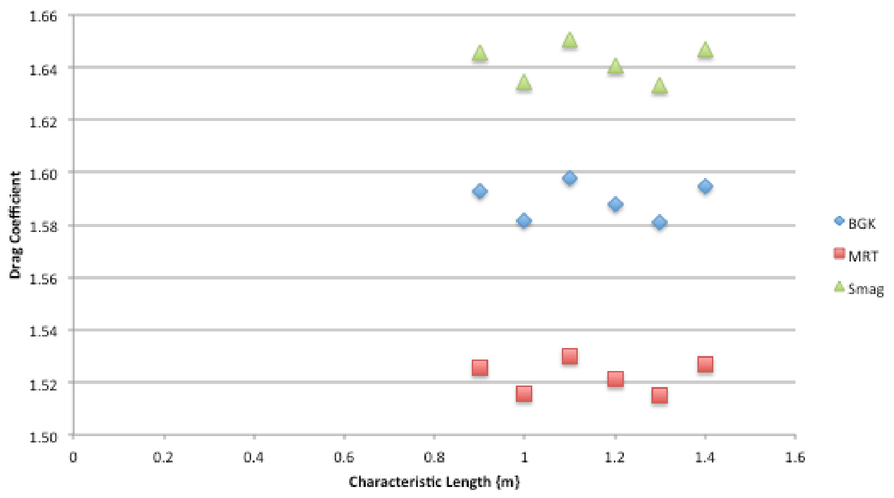


Figure 5.4: Periodicity simulations results. The results for the drag coefficient for periodicity simulations are shown as a function of the characteristic length.

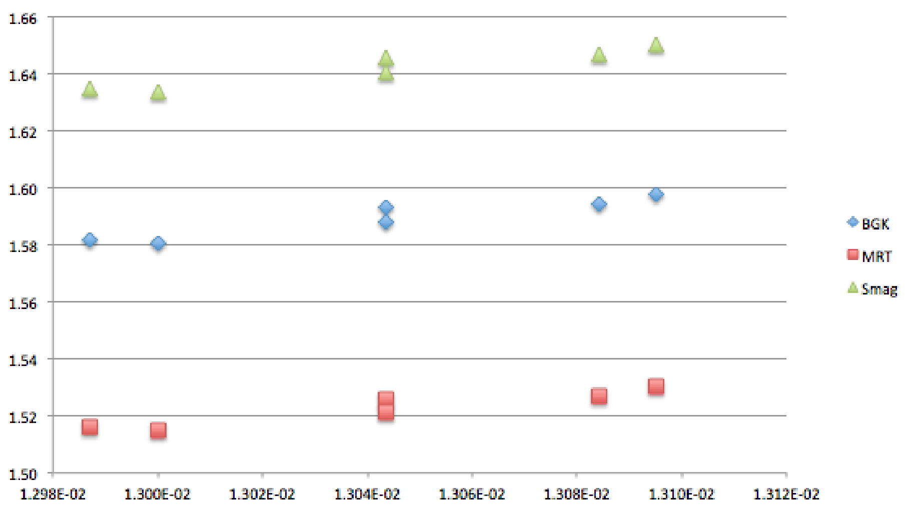
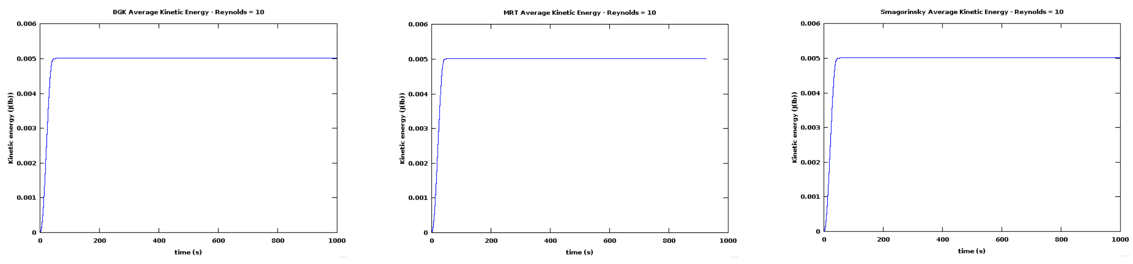


Figure 5.5: Periodicity simulations results as function of the space step. An increase in the space step means a decrease in the resolution and a consequential decrease in the precision of the drag calculated.

5.4 Long Run Times

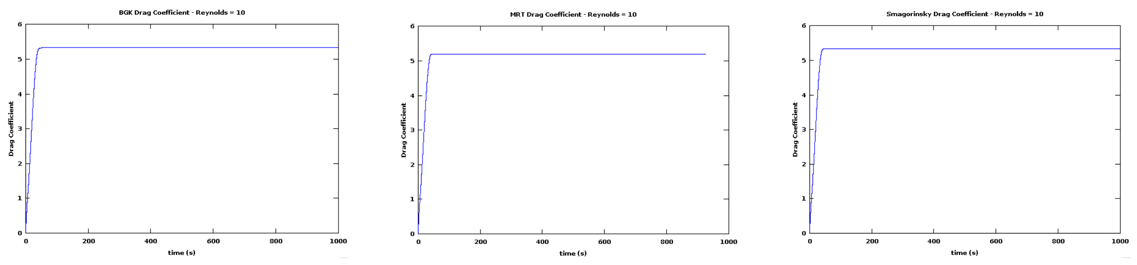
Another issue that came to our attention was the convergence of the results and the possibility that the models were in a metastable equilibrium for the results shown. This would mean that the drag coefficient plots for the simulations made, as shown in figure 3.3, has not reached its final state of equilibrium.

To test for these circumstances we ran simulations for the three methods (BGK, MRT, Smagorinsky) at a Reynolds of 10 and for a total run time of 1000 seconds. The results are shown in figures 5.6 and 5.7. Figure 5.6 shows the average kinetic energy for the entire domain in the simulations, while figure 5.7 shows the drag coefficient calculated on the obstacle. From these figures it is easy to notice how the previous results obtained are not part of a metastable equilibrium in this time frame, and how they agree with each other: the plots reach a stable state wherein no further change to the system is significant or noticeable.



(a) BGK **(b)** MRT **(c)** Smagorinsky

Figure 5.6: Average kinetic energy for long run time simulations. Results for the different lattice Boltzmann methods are shown for a Reynolds of 10 and a total run time of 1000 seconds, in lattice units.



(a) BGK **(b)** MRT **(c)** Smagorinsky

Figure 5.7: Drag coefficient for long run time simulations. Results for the different lattice Boltzmann methods are shown for a Reynolds of 10 and a total run time of 1000 seconds.

6 Discussion and Conclusions

This work has explained how the lattice Boltzmann method works, ran a simulation of fluid around a sphere, and computed drag coefficients based on the force exerted by the fluid to an obstacle.

This project demonstrates how the computation of the drag coefficient from lattice Boltzmann simulations is a very viable option. It showed how through three different techniques one could get to reasonable drag coefficients for low Reynolds numbers. Additionally, the easiness with which an object can be imported into the domain of the simulation should be very provocative for researchers that wish to test their drag models computationally.

The results obtained suggest that the MRT method is the most reliable one, among the ones that were studied, and the author suggests that in future research projects the MRT method should be the emphasis: perhaps the relaxation time matrix could be altered in order to provide accuracy for the models, or perhaps in order to be able to run the models at higher Reynolds in some time in the future.

Evidence was also gathered that suggests the Smagorinsky model adds nothing to the computation of drag in BGK. Certainly, if a simulation were to take place and the computation of the drag force on an object was the objective, it is preferable to use the BGK collision operator over the Smagorinsky model. This due to the comparative efficiency of BGK compared to the Smagorinsky model.

One problem encountered during the execution of the simulations was the restriction on the simulation at high Reynolds numbers. The author could not get around the instabilities caused by the small τ in the models with the higher Reynolds. Reasons for these instabilities were discussed in this document and a lot of work in this area is necessary and encouraged. Increasing the resolution by a great amount could solve this problem, but to be able to do this, great computational resources are needed. Working with clusters of thousands of cores could prove successful in this endeavor.

The author also wishes to emphasize the usefulness of the Palabos code. Instead of wasting months developing inefficient code for a very narrow range of research projects, one can just get accustomed to the Palabos syntax and model creating, and in this way do much more than one would have done otherwise. This expands the possibilities of researchers in a very real way.

Appendix

A Code

The following code is for the BGK collision operator and the Smagorinsky technique. Which of these is active depends on the definition of the Boolean variable useSmago. If this variable is true the model follows Smagorinsky, if not it follows BGK. The code for the MRT model is not shown because it would be redundant. The MRT model is the exact same as the BGK model, except the descriptor for the velocity set is different: instead of being *D3Q19Descriptor* it is *MRTD3Q19Descriptor*. All of the parameters in the model are input with a companion xml file.

```cpp

#include "palabos3D.h"
#include "palabos3D.hh"
#include <ctime>

using namespace plb;
using namespace std;

typedef double T;
typedef Array<T,3> Velocity;
#define DESCRIPTOR descriptors::D3Q19Descriptor
#define PADDING 8

static std::string outputDir("./tmp/");


// This structure holds all the user-defined parameters, and some
// derived values needed for the simulation.
struct Param
{
    T nu;                               // Kinematic viscosity.
    T lx, ly, lz;                       // Size of computational domain, in physical units.
    T cx, cy, cz;                       // Position of the center of the obstacle, in physical units.
    plint cxLB, cyLB, czLB;             // Position of the center of the obstacle, in lattice units.
    bool freeSlipWall;                  // Use free-slip condition on obstacle, as opposed to no-slip?
    bool lateralFreeSlip;               // Use free-slip lateral boundaries or periodic ones?
    T maxT, statT, imageT, vtkT;        // Time, in physical units, at which events occur.
    plint resolution;                   // Number of lattice nodes along a reference length.
    T inletVelocity;                    // Inlet x-velocity in ↩
        physical units.
    T uLB;                              // Velocity in lattice units ↩
        (numerical parameters).
    bool useSmago;                      // Use a Smagorinsky LES ↩
        model or not.
    T cSmago;                           // Parameter for the ↩
        Smagorinsky LES model.
    plint nx, ny, nz;                   // Grid resolution of ↩
        bounding box.
    T omega;                            // Relaxation parameter.
    T dx, dt;                           // Discrete space and time ↩
        steps.
    plint maxIter, statIter;            // Time for events in lattice↩
         units.
    plint imageIter, vtkIter;
    T outletSpongeZoneWidth;            // Width of the outlet sponge↩
         zone.
    plint numOutletSpongeCells;         // Number of the lattice ↩
        nodes contained in the outlet sponge zone.
    int outletSpongeZoneType;           // Type of the outlet sponge ↩
        zone (Viscosity or Smagorinsky).
    T targetSpongeCSmago;               // Target Smagorinsky ↩
        parameter at the end of the Smagorinsky sponge Zone.
    plint initialIter;                  // Number of initial ↩
        iterations until the inlet velocity reaches its final value.

    Box3D inlet, outlet, lateral1;      // Outer domain boundaries in↩
         lattice units.
    Box3D lateral2, lateral3, lateral4;

    std::string geometry_fname;

    Param()
    { }

    Param(std::string xmlFname)
    {
        XMLreader document(xmlFname);
        document["geometry"]["filename"].read(geometry_fname);
        document["geometry"]["center"]["x"].read(cx);
        document["geometry"]["center"]["y"].read(cy);
        document["geometry"]["center"]["z"].read(cz);
        document["geometry"]["freeSlipWall"].read(freeSlipWall);
        document["geometry"]["lateralFreeSlip"].read(lateralFreeSlip)↩
            ;
        document["geometry"]["domain"]["x"].read(lx);
        document["geometry"]["domain"]["y"].read(ly);
        document["geometry"]["domain"]["z"].read(lz);

        document["numerics"]["nu"].read(nu);
        document["numerics"]["inletVelocity"].read(inletVelocity);
        document["numerics"]["resolution"].read(resolution);
        document["numerics"]["uLB"].read(uLB);
        document["numerics"]["useSmago"].read(useSmago);
        if (useSmago) {
            document["numerics"]["cSmago"].read(cSmago);
        }
        document["numerics"]["outletSpongeZoneWidth"].read(↩
           outletSpongeZoneWidth);
        std::string zoneType;
        document["numerics"]["outletSpongeZoneType"].read(zoneType);
        if ((util::tolower(zoneType)).compare("viscosity") == 0) {
            outletSpongeZoneType = 0;
        } else if ((util::tolower(zoneType)).compare("smagorinsky") ↩
           == 0) {
            outletSpongeZoneType = 1;
        } else {
            pcout << "The sponge zone type must be either \"Viscosity↩
               \" or \"Smagorinsky\"." << std::endl;
            exit(-1);
        }
        document["numerics"]["targetSpongeCSmago"].read(↩
           targetSpongeCSmago);

        document["numerics"]["initialIter"].read(initialIter);

        document["output"]["maxT"].read(maxT);
        document["output"]["statT"].read(statT);
        document["output"]["imageT"].read(imageT);
        document["output"]["vtkT"].read(vtkT);

        computeLBparameters();
    }

    void computeLBparameters()
    {
        dx = ly / (resolution - 1.0);
        dt = (uLB/inletVelocity) * dx;
        T nuLB = nu * dt/(dx*dx);
        omega = 1.0/(DESCRIPTOR<T>::invCs2*nuLB+0.5); //the ↩
           descriptor is defined after the Param declaration

        if (lateralFreeSlip) {
            nx = util::roundToInt(lx/dx) + 1;
            ny = util::roundToInt(ly/dx) + 1;
            nz = util::roundToInt(lz/dx) + 1;
        } else {
            nx = util::roundToInt(lx/dx) + 1;
            ny = util::roundToInt(ly/dx);
            nz = util::roundToInt(lz/dx);
        }
        cxLB = util::roundToInt(cx/dx);
        cyLB = util::roundToInt(cy/dx);
        czLB = util::roundToInt(cz/dx);
        maxIter   = util::roundToInt(maxT/dt);
        statIter  = util::roundToInt(statT/dt);
        imageIter = util::roundToInt(imageT/dt);
        vtkIter   = util::roundToInt(vtkT/dt);
        numOutletSpongeCells = util::roundToInt(outletSpongeZoneWidth↩
           /dx);

        inlet    = Box3D(0,      0,      0,      ny-1,   0,      nz↩
            -1);
        outlet   = Box3D(nx-1,   nx-1,   0,      ny-1,   0,      nz↩
            -1);
        lateral1 = Box3D(1,      nx-2,   0,      0,      0,      nz↩
            -1);
        lateral2 = Box3D(1,      nx-2,   ny-1,   ny-1,   0,      nz↩
            -1);
        lateral3 = Box3D(1,      nx-2,   1,      ny-2,   0,      0);
        lateral4 = Box3D(1,      nx-2,   1,      ny-2,   nz-1,   nz↩
            -1);
    }

    Box3D boundingBox() const
    {
        return Box3D(0, nx-1, 0, ny-1, 0, nz-1);
    }

    T getInletVelocity(plint iIter)
    {
        static T pi = std::acos((T) -1.0);

        if (iIter >= initialIter) {
            return uLB;
        }

        if (iIter < 0) {
            iIter = 0;
        }

        return uLB * std::sin(pi * iIter / (2.0 * initialIter));
    }
};

Param param;

// Instantiate the boundary conditions for the outer domain.
void outerDomainBoundaries(MultiBlockLattice3D<T,DESCRIPTOR> *lattice↩
    ,
                           MultiScalarField3D<T> *rhoBar,
                           MultiTensorField3D<T,3> *j,
                           OnLatticeBoundaryCondition3D<T,DESCRIPTOR>↩
                               *bc)
{
    Array<T,3> uBoundary(param.getInletVelocity(0), 0.0, 0.0);

    if (param.lateralFreeSlip) {
        pcout << "Free-slip lateral boundaries." << std::endl;

        lattice->periodicity().toggleAll(false);
        rhoBar->periodicity().toggleAll(false);
        j->periodicity().toggleAll(false);

        bc->setVelocityConditionOnBlockBoundaries(*lattice, param.↩
            inlet, boundary::dirichlet);
        setBoundaryVelocity(*lattice, param.inlet, uBoundary);

        bc->setVelocityConditionOnBlockBoundaries(*lattice, param.lateral1, boundary::freeslip);
        bc->setVelocityConditionOnBlockBoundaries(*lattice, param.lateral2, boundary::freeslip);
        bc->setVelocityConditionOnBlockBoundaries(*lattice, param.lateral3, boundary::freeslip);
        bc->setVelocityConditionOnBlockBoundaries(*lattice, param.lateral4, boundary::freeslip);
        setBoundaryVelocity(*lattice, param.lateral1, uBoundary);
        setBoundaryVelocity(*lattice, param.lateral2, uBoundary);
        setBoundaryVelocity(*lattice, param.lateral3, uBoundary);
        setBoundaryVelocity(*lattice, param.lateral4, uBoundary);

        // The VirtualOutlet is a sophisticated outflow boundary condition.
        Box3D globalDomain(lattice->getBoundingBox());
        std::vector<MultiBlock3D*> bcargs;
        bcargs.push_back(lattice);
        bcargs.push_back(rhoBar);
        bcargs.push_back(j);
        T outsideDensity = 1.0;
        int bcType = 1;
        integrateProcessingFunctional(new VirtualOutlet<T,DESCRIPTOR>(outsideDensity, globalDomain, bcType),
                param.outlet, bcargs, 2);
        setBoundaryVelocity(*lattice, param.outlet, uBoundary);
    } else {
        pcout << "Periodic lateral boundaries." << std::endl;

        lattice->periodicity().toggleAll(true);
        rhoBar->periodicity().toggleAll(true);
        j->periodicity().toggleAll(true);

        lattice->periodicity().toggle(0, false);
        rhoBar->periodicity().toggle(0, false);
        j->periodicity().toggle(0, false);

        bc->addVelocityBoundary0N(param.inlet, *lattice);
        setBoundaryVelocity(*lattice, param.inlet, uBoundary);

        // The VirtualOutlet is a sophisticated outflow boundary condition.
        // The "globalDomain" argument for the boundary condition must be
        // bigger than the actual bounding box of the simulation for
        // the directions which are periodic.
        Box3D globalDomain(lattice->getBoundingBox());
        globalDomain.y0 -= 2; // y-periodicity
        globalDomain.y1 += 2;
        globalDomain.z0 -= 2; // z-periodicity
        globalDomain.z1 += 2;
        std::vector<MultiBlock3D*> bcargs;
        bcargs.push_back(lattice);
        bcargs.push_back(rhoBar);
        bcargs.push_back(j);
```

```cpp
        T outsideDensity = 1.0;
        int bcType = 1;
        integrateProcessingFunctional(new VirtualOutlet<T,DESCRIPTOR>(outsideDensity, globalDomain, bcType),
                param.outlet, bcargs, 2);
        setBoundaryVelocity(*lattice, param.outlet, uBoundary);
    }
}

// Write VTK file for the flow around the obstacle, to be viewed with Paraview.
void writeVTK(OffLatticeBoundaryCondition3D<T,DESCRIPTOR,Velocity>& bc, plint iT)
{
    VtkImageOutput3D<T> vtkOut(createFileName("volume", iT, PADDING));
    vtkOut.writeData<float>( *bc.computeVelocityNorm(param.boundingBox()),
                             "velocityNorm", param.dx/param.dt );
    vtkOut.writeData<3,float>(*bc.computeVelocity(param.boundingBox()), "velocity", param.dx/param.dt);
    vtkOut.writeData<float>( *bc.computePressure(param.boundingBox()),
                             "pressure", param.dx*param.dx/(param.dt*param.dt) );
}

// Write PPM images on slices.
void writeGIF(OffLatticeBoundaryCondition3D<T,DESCRIPTOR,Velocity>& bc, plint iT)
{
    const plint imSize = 600;
    //Box3D xSlice(param.cxLB, param.cxLB, 0,          param.ny-1, 0, param.nz-1);
    //Box3D ySlice(0,          param.nx-1, param.cyLB, param.cyLB, 0, param.nz-1);
    Box3D zSlice(0,          param.nx-1, 0,          param.ny-1, param.czLB, param.czLB);

    ImageWriter<T> writer("leeloo");
    //writer.writeScaledGif(createFileName("vnorm_xslice", iT, PADDING), *bc.computeVelocityNorm(xSlice));
    //writer.writeScaledGif(createFileName("vnorm_yslice", iT, PADDING), *bc.computeVelocityNorm(ySlice), imSize, imSize);
    writer.writeScaledGif(createFileName("vnorm_zslice", iT, PADDING), *bc.computeVelocityNorm(zSlice), imSize, imSize);
}

void runProgram()
{
    std::clock_t start;
    double duration;

    start = std::clock();

    /*
```

```
     * Read the obstacle geometry.
     */

    pcout << std::endl << "Reading STL data for the obstacle geometry↩
        ." << std::endl;
    Array<T,3> center(param.cx, param.cy, param.cz);
    Array<T,3> centerLB(param.cxLB, param.cyLB, param.czLB);
    // The triangle-set defines the surface of the geometry.
    TriangleSet<T> triangleSet(param.geometry_fname, DBL);

    // Place the obstacle in the correct place in the simulation ↩
        domain.
    // Here the "geometric center" of the obstacle is computed ↩
        manually,
    // by computing first its bounding cuboid. In cases that the STL
    // file with the geometry of the obstacle contains its center as
    // the point, say (0, 0, 0), then the following variable
    // "obstacleCenter" must be set to (0, 0, 0) manually.
    Cuboid<T> bCuboid = triangleSet.getBoundingCuboid();
    Array<T,3> obstacleCenter = (T) 0.5 * (bCuboid.lowerLeftCorner + ↩
        bCuboid.upperRightCorner);
    triangleSet.translate(-obstacleCenter);
    triangleSet.scale(1.0/param.dx); // In lattice units from now on↩
        ...
    triangleSet.translate(centerLB);
    triangleSet.writeBinarySTL(outputDir+"obstacle_LB.stl");

    // The DEFscaledMesh, and the triangle-boundary are more ↩
        sophisticated data
    // structures used internally by Palabos to treat the boundary.
    plint xDirection = 0;
    plint borderWidth = 1;      // Because Guo acts in a one-cell ↩
        layer.
                                // Requirement: margin>=borderWidth.
    plint margin = 1;           // Extra margin of allocated cells ↩
        around the obstacle, for the case of moving walls.
    plint blockSize = 0;        // Size of blocks in the sparse/↩
        parallel representation.
    DEFscaledMesh<T> defMesh(triangleSet, 0, xDirection, margin, ↩
        Dot3D(0, 0, 0));
    TriangleBoundary3D<T> boundary(defMesh);
    //boundary.getMesh().inflate();

    pcout << "tau = " << 1.0/param.omega << std::endl;
    pcout << "dx = " << param.dx << std::endl;
    pcout << "dt = " << param.dt << std::endl;
    pcout << "Number of iterations in an integral time scale: " << (↩
        plint) (1.0/param.dt) << std::endl;

    /*
     * Voxelize the domain.
     */

    // Voxelize the domain means: decide which lattice nodes are ↩
        inside the obstacle and which are outside.
    pcout << std::endl << "Voxelizing the domain." << std::endl;
    plint extendedEnvelopeWidth = 2;   // Extrapolated off-lattice ↩
        BCs.
    const int flowType = voxelFlag::outside;
    VoxelizedDomain3D<T> voxelizedDomain (
            boundary, flowType, param.boundingBox(), borderWidth, ↩
                extendedEnvelopeWidth, blockSize );
    pcout << getMultiBlockInfo(voxelizedDomain.getVoxelMatrix()) << ↩
        std::endl;

    /*
     * Generate the lattice, the density and momentum blocks.
     */

    pcout << "Generating the lattice, the rhoBar and j fields." << ↩
        std::endl;
    MultiBlockLattice3D<T,DESCRIPTOR> *lattice = new ↩
        MultiBlockLattice3D<T,DESCRIPTOR>(voxelizedDomain.↩
        getVoxelMatrix());
    if (param.useSmago) {
        defineDynamics(*lattice, lattice->getBoundingBox(),
                new SmagorinskyBGKdynamics<T,DESCRIPTOR>(param.omega,↩
                    param.cSmago));
        pcout << "Using Smagorinsky BGK dynamics." << std::endl;
    } else {
        defineDynamics(*lattice, lattice->getBoundingBox(),
                new BGKdynamics<T,DESCRIPTOR>(param.omega));
        pcout << "Using BGK dynamics." << std::endl;
    }
    bool velIsJ = false;
    defineDynamics(*lattice, voxelizedDomain.getVoxelMatrix(), ↩
        lattice->getBoundingBox(),
            new NoDynamics<T,DESCRIPTOR>(), voxelFlag::inside);
    lattice->toggleInternalStatistics(false);

    MultiBlockManagement3D sparseBlockManagement(lattice->↩
        getMultiBlockManagement());

    // The rhoBar and j fields are used at both the collision and at ↩
        the implementation of the
    // outflow boundary condition.
    plint envelopeWidth = 1;
    MultiScalarField3D<T> *rhoBar = generateMultiScalarField<T>((↩
        MultiBlock3D&) *lattice, envelopeWidth).release();
    rhoBar->toggleInternalStatistics(false);

    MultiTensorField3D<T,3> *j = generateMultiTensorField<T,3>((↩
        MultiBlock3D&) *lattice, envelopeWidth).release();
    j->toggleInternalStatistics(false);

    std::vector<MultiBlock3D*> lattice_rho_bar_j_arg;
    lattice_rho_bar_j_arg.push_back(lattice);
    lattice_rho_bar_j_arg.push_back(rhoBar);
    lattice_rho_bar_j_arg.push_back(j);
    integrateProcessingFunctional(
            new ExternalRhoJcollideAndStream3D<T,DESCRIPTOR>(),
            lattice->getBoundingBox(), lattice_rho_bar_j_arg, 0);
    integrateProcessingFunctional(
            new BoxRhoBarJfunctional3D<T,DESCRIPTOR>(),
            lattice->getBoundingBox(), lattice_rho_bar_j_arg, 3); // rhoBar and j are computed at level 3 because
                                                                  // the boundary conditions are on levels 1 and 2.

    /*
     * Generate the off-lattice boundary condition on the obstacle and the outer-domain boundary conditions.
     */

    pcout << "Generating boundary conditions." << std::endl;

    OffLatticeBoundaryCondition3D<T,DESCRIPTOR,Velocity> *boundaryCondition;

    BoundaryProfiles3D<T,Velocity> profiles;
    bool useAllDirections=true;
    OffLatticeModel3D<T,Velocity>* offLatticeModel=0;
    if (param.freeSlipWall) {
        profiles.setWallProfile(new FreeSlipProfile3D<T>);
    }
    else {
        profiles.setWallProfile(new NoSlipProfile3D<T>);
    }
    offLatticeModel =
         new GuoOffLatticeModel3D<T,DESCRIPTOR> (
            new TriangleFlowShape3D<T,Array<T,3> >(voxelizedDomain.getBoundary(), profiles),
            flowType, useAllDirections );
    offLatticeModel->setVelIsJ(velIsJ);
    boundaryCondition = new OffLatticeBoundaryCondition3D<T,DESCRIPTOR,Velocity>(
            offLatticeModel, voxelizedDomain, *lattice);

    boundaryCondition->insert();

    // The boundary condition algorithm or the outer domain.
    OnLatticeBoundaryCondition3D<T,DESCRIPTOR>* outerBoundaryCondition
        = createLocalBoundaryCondition3D<T,DESCRIPTOR>();
    outerDomainBoundaries(lattice, rhoBar, j, outerBoundaryCondition);

    /*
     * Implement the outlet sponge zone.
     */

    if (param.numOutletSpongeCells > 0) {
        T bulkValue;
        Array<plint,6> numSpongeCells;

        if (param.outletSpongeZoneType == 0) {
            pcout << "Generating an outlet viscosity sponge zone." << std::endl;
            bulkValue = param.omega;
        } else if (param.outletSpongeZoneType == 1) {
            pcout << "Generating an outlet Smagorinsky sponge zone." << std::endl;
            bulkValue = param.cSmago;
        } else {
            pcout << "Error: unknown type of sponge zone." << std::endl;
            exit(-1);
        }

        // Number of sponge zone lattice nodes at all the outer domain boundaries.
        // So: 0 means the boundary at x = 0
        //     1 means the boundary at x = nx-1
        //     2 means the boundary at y = 0
        //     and so on...
        numSpongeCells[0] = 0;
        numSpongeCells[1] = param.numOutletSpongeCells;
        numSpongeCells[2] = 0;
        numSpongeCells[3] = 0;
        numSpongeCells[4] = 0;
        numSpongeCells[5] = 0;

        std::vector<MultiBlock3D*> args;
        args.push_back(lattice);

        if (param.outletSpongeZoneType == 0) {
            applyProcessingFunctional(new ViscositySpongeZone<T,DESCRIPTOR>(
                        param.nx, param.ny, param.nz, bulkValue, numSpongeCells),
                    lattice->getBoundingBox(), args);
        } else {
            applyProcessingFunctional(new SmagorinskySpongeZone<T,DESCRIPTOR>(
                        param.nx, param.ny, param.nz, bulkValue, param.targetSpongeCSmago, numSpongeCells),
                    lattice->getBoundingBox(), args);
        }
    }

    /*
     * Setting the initial conditions.
     */

    // Initial condition: Constant pressure and velocity-at-infinity everywhere.
    Array<T,3> uBoundary(param.getInletVelocity(0), (T)0.0, (T)0.0);
    initializeAtEquilibrium(*lattice, lattice->getBoundingBox(), (T)1.0, uBoundary);
    applyProcessingFunctional(
            new BoxRhoBarJfunctional3D<T,DESCRIPTOR>(),
            lattice->getBoundingBox(), lattice_rho_bar_j_arg); // Compute rhoBar and j before VirtualOutlet is executed.
    //lattice->executeInternalProcessors(1); // Execute all processors except the ones at level 0.
    //lattice->executeInternalProcessors(2);
    //lattice->executeInternalProcessors(3);

    /*
     * Starting the simulation.
     */

    plb_ofstream energyFile((outputDir+"avenergy.dat").c_str());
    plb_ofstream dragFile((outputDir+"drag.dat").c_str());
    plb_ofstream infoFile((outputDir+"info.dat").c_str());

    pcout << std::endl;
    pcout << "Starting simulation." << std::endl;

    if (param.useSmago) {
        infoFile << "Dynamics Smagorinsky" << std::endl;
    } else {
        infoFile << "Dynamics BGK" << std::endl;
    }
    infoFile << "Reynolds= " << param.inletVelocity*0.1/param.nu << std::endl;
    infoFile << "Resolution= " << param.resolution << std::endl;
    infoFile << "dx= " << param.dx << std::endl;
    infoFile << "dt= " << param.dt << std::endl;
    infoFile << "Numberofiterationsinanintegraltimescale: " << (plint) (1.0/param.dt) << std::endl;
    infoFile << "uLB= " << param.uLB << std::endl;
    infoFile << "nu= " << param.nu << std::endl;
    T nuLB = param.nu * param.dt/(param.dx*param.dx);
    infoFile << "nuLB= " << nuLB << std::endl;
    infoFile << "tau= " << 1.0/param.omega << std::endl;
    infoFile << "inletVelocity= " << param.inletVelocity << std::endl;
    infoFile << "freeSlipWall= " << param.freeSlipWall << std::endl;
    infoFile << "lateralFreeSlip= " << param.lateralFreeSlip << std::endl;
    infoFile << "initialIter= " << param.initialIter << std::endl;
    infoFile << getMultiBlockInfo(voxelizedDomain.getVoxelMatrix()) << std::endl;

    static T pi = std::acos((T) -1.0);
    static T rlb = 0.05 / param.dx;

    for (plint i = 0; i < param.maxIter; ++i) {
        if (i <= param.initialIter) {
            Array<T,3> uBoundary(param.getInletVelocity(i), 0.0, 0.0);
            setBoundaryVelocity(*lattice, param.inlet, uBoundary);
        }

        if (i \% param.statIter == 0 && param.statIter > 0) {
             pcout << "At iteration " << i << ", t = " << i*param.dt << std::endl;
             Array<T,3> force(boundaryCondition->getForceOnObject());
             //T factor = util::sqr(util::sqr(param.dx)) / util::sqr(param.dt);
             T factor = 2/(util::sqr(param.uLB)*pi*util::sqr(rlb));
             pcout << "Drag coefficient on object over fluid density: Cd[x] = " << force[0]*factor << ", Cd[y] = "
                  << force[1]*factor << ", Cd[z] = " << force[2]*factor << std::endl;
             dragFile << i*param.dt << " " << force[0]*factor << " " << force[1]*factor << " " << force[2]*factor << std::endl;
             T avEnergy = boundaryCondition->computeAverageEnergy() * util::sqr(param.dx) / util::sqr(param.dt);
             pcout << "Average kinetic energy over fluid density: E = " << avEnergy << std::endl;
             energyFile << i*param.dt << "  " << avEnergy << std::endl;
             pcout << std::endl;
        }

        if (i \% param.vtkIter == 0) {
            pcout << "Writing VTK at time t = " << i*param.dt << endl;
            writeVTK(*boundaryCondition, i);
            duration = ( std::clock() - start ) / (double) CLOCKS_PER_SEC;
            infoFile << "t=" << i*param.dt << " " << duration << std::endl;
            pcout << std::endl;
        }

        if (i \% param.imageIter == 0) {
            pcout << "Writing GIF image at time t = " << i*param.dt << endl;
            writeGIF(*boundaryCondition, i);
            pcout << std::endl;
        }

        lattice->executeInternalProcessors();
        lattice->incrementTime();
    }

    energyFile.close();
    dragFile.close();
    infoFile.close();
    delete outerBoundaryCondition;
    delete boundaryCondition;
    delete j;
    delete rhoBar;
    delete lattice;
}

int main(int argc, char* argv[])
{
    plbInit(&argc, &argv);
    global::directories().setOutputDir(outputDir);

    // The try-catch blocks catch exceptions in case an error occurs,
    // and terminate the program properly with a nice error message.

    // 1. Read command-line parameter: the input file name.
    string xmlFileName;



    try {
        global::argv(1).read(xmlFileName);
    }
    catch (PlbIOException& exception) {
        pcout << "Wrong parameters; the syntax is: "
              << (std::string)global::argv(0) << " input-file.xml" <<↩
                  std::endl;
        return -1;
    }

    // 2. Read input parameters from the XML file.
    try {
        param = Param(xmlFileName);
    }
    catch (PlbIOException& exception) {
        pcout << exception.what() << std::endl;
        return -1;
    }

    // 3. Execute the main program.
    try {
        runProgram();
    }
    catch (PlbIOException& exception) {
        pcout << exception.what() << std::endl;
        return -1;
    }
}
```

References

Brownlee, R., Gorban, A., & Levesley, J. (2007). Stability and stabilization of the lattice Boltzmann method. *Physical Review*(75).

Cheng, Y., & Zhang, H. (2011, June). A viscosity counteracting approach in the lattice Boltzmann BGK model for low viscosity flow: Preliminary verification. *Computers and Mathematics with Applications*, *61*(12). doi: 10.1016/j.camwa.2010.11.031

Chopard, B., & Droz, M. (1998). *Cellular Automata Modeling of Physical Systems.* Cambridge, England: Cambridge University Press.

d'Humières, D., Ginzburg, I., Krafczyk, M., Lallemand, P., & Luo, L.-S. (2002). Multiple-relaxation-time lattice Boltzmann models in three dimensions. *The Royal Society*. doi: 10.1098/rsta.2001.0955

Gorban, A., & Packwood, D. (2012). *Enhancement of the Stability of Lattice Boltzmann Methods By Dissipation Control.* Retrieved 14th of December, 2015, from `https://www.ma.utexas.edu/mp_arc/c/13/13-20.pdf`

Guo, Z., & Shu, C. (2013). Lattice Boltzmann Method and its Applications in Engineering. *Advances in Computational Fluid Dynamics*(9859).

Hänel, D. (2004). *Molekulare Gasdynamik.* Berlin, Germany: Springer-Verlag.

Keating, B. R. (2011). *Methods for stabilizing high Reynolds number Lattice Boltzmann simulations.* Ann Arbor, MI: ProQuest LLC.

Lallemand, P., & Luo, L.-S. (2000, April). Theory of the Lattice Boltzmann Method: Dispersion, Dissipation, Isotropy, Galilean Invariance, and Stability. *ICASE*(17).

Latt, J. (2008, April). *Choice of units in lattice Boltzmann simulations.* Retrieved 30th of November, 2015, from `http://wiki.palabos.org/_media/howtos:lbunits.pdf`

Mohamad, A. A. (2011). *Lattice Boltzmann Method: Fundamentals and Engineering Applications with Computer Codes.* London, England: Springer.

Morrison, F. A. (2013). *An Introduction to Fluid Mechanics.* New York, USA: Cambridge University Press.

NASA. (2015, November). *What is Drag?* Retrieved 30th of November, 2015, from `https://www.grc.nasa.gov/www/k-12/airplane/drag1.html`

Palabos. (2015, November). *What is Palabos?* Retrieved 30th of November, 2015, from `http://www.palabos.org/documentation/userguide/introduction.html#what-is-palabos`

Succi, S. (2001). *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond.* Oxford, England: Oxford University Press.

Sukop, M. C., & Daniel T. Thorne, J. (2005). *Lattice Boltzmann Modeling: An Introduction for Geoscientists and Engineers.* Berlin, Germany: Springer.

Viggen, E. M. (2009). *The Lattice Boltzmann Method with Applications in Acoustics* (Master's thesis). Norwegian University of Science and Technology, Trondheim, Norway.

Viggen, E. M. (2014). *The lattice Boltzmann method: Fundamentals and acoustics* (PhD thesis). Norwegian University of Science and Technology, Trondheim, Norway.

Wolf-Gladrow, D. A. (2000). *Lattice-Gas Cellular Automata and lattice Boltzmann Models*. Berlin, Germany: Springer-Verlag.