FLOW OF DILUTE GASES IN COMPLEX NANOPOROUS MEDIA

by

Anders Hafreager

Thesis

for the degree of

Master of Science

Faculty of Mathematics and Natural Sciences
University of Oslo

March 2014

Abstract

Most of the worlds currently accessible hydrocarbon resources are found in tight rocks - rocks with permeabilities in the millidarcy range and with pore sizes in the nanometer range. Tight rocks pose new scientific problems because of the small length-scales involved. Traditional oil plays are found in, for example, sand stone reservoirs with millimeter to micrometer sized pores. For such systems, standard hydrodynamics is a sufficient tool to understand, describe and predict fluid transport. In tight rocks, however, typical pore sizes are in the range of tens to hundreds of nanometers. At this scale, the mean free path - the average distance a particle moves between collisions - may become comparable to the characteristic sizes of the porous medium, and the standard assumption that the fluid can be described as a continuum no longer holds. The mean free path of dilute gases are often tens of nanometers, or higher.

The gas from tight rocks, like shales, is stored inside closed pore networks or adsorbed onto organic matter. In order to extract adequate levels of gas from such rocks, we generate fractures to increase the permeability of the rock. The gas flows from small pore networks with diameters down to a few nanometers. The rate at which the gas flows through such networks is proportional to the permeability of the material - a result known as Darcy's law. Dilute gases in nanoporous media have a non-zero slip velocity which can cause an increase of permeability of a factor 50 compared to what continuum theory predicts. This is an effect known as the Klinkenberg effect. In order to be able to increase gas production rates in a safe way, we need to understand the physics at this scale. This requires models that are valid at these length scales.

In this thesis, we implement two numerical particle models. The first is called Molecular Dynamics. It describes the motion of atoms by using parameterized potentials to compute forces between them. The second model, Direct Simulation Monte Carlo, uses the principles of statistical mechanics allowing larger systems to be studied. Both implementations support arbitrary geometries, and show promising scaling efficiency on massive parallel machines. We use these models to study the Klinkenberg effect and confirm that the Knudsen permeability correction correctly predicts the permeability for systems with pore sizes down to a few nanometers. We also analyze more complicated geometries, and argue that a stochastic version of the Knudsen correction is needed for geometries without a well defined Knudsen number.

Highly efficient custom 3D visualization tools are developed using modern OpenGL techniques such as instanced geometry shaders, billboards and the marching cubes algorithm. Systems with tens of millions of particles can be rendered realtime with decent frame rate, allowing us to study larger systems than what can be done with already existing free software. All software developed during this thesis can serve as tools for further study in the field.

Acknowledgements

My first encounter with physics was in high school with my physics teacher Knut Løvseth. I remember your classes. They were filled with enthusiasm and joy - you showed how much you love physics and science. You are a true inspirer, a great motivator. I am grateful for how you showed me the path to what I love the most - physics.

Later on, at the university, when I attended lectures in the introductory course in mechanics, I met Professor Anders Malthe-Sørenssen, who later became my supervisor. I want to thank you for being another very inspiring person. Every time we talk I get motivated and hungry for learning more, exploring new ideas and connecting the dots in my life. I enjoy every discussion about physics, startups and choices in life in general.

I would also like to thank Svenn-Arne Dragly, whom I have shared endless hours with, having fun learning OpenGL, programming, physics and useless things like stories about prisoners and chests. Our work together does not stop here, we will continue creating amazing stuff.

Learning about Molecular Dynamics and statistical mechanics has been a joyful ride thanks to Kjetil Thøgersen, Jørgen Kjoshagen Trømborg and Camilla Kirkemo Alm. All our discussions about potentials, water, salt, no salt, erroneously salt and cage correlation functions have been great. Also, thank you for all your help with my thesis. I look forward to working with you over the next years.

A big thanks to everyone at the Computational Physics group. You make me look forward to work the next morning, day after day, week after week. And of course, a special thanks to Andreas Nakkerud. All the weird things we have done the past years have been great. Lastly, I want to thank Thea Marcelia Sletten. You are an amazing person.

Oslo, 30 March, 2014
Anders Hafreager

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the later years, the field of computational science has been merged with theoretical physics forming a new field; computational physics. Not only is the physics important, but numerical models, algorithms and code optimizing are important parts of the daily work. A computational physicist often spends most of the time on writing code - a task that can be both frustrating and enjoyable at the same time. The code should do as intended in addition to being both readable and efficient.

A master student starting working on a thesis in computational physics must at some point choose a focus area somewhere in between computer science and physics. The author finds a great pleasure in both fields, but has in the work in this thesis spent most of the time doing code development. But of course, the underlying questions are physical questions whereas the tool we use to answer them is computer science.

In this chapter we first present the project description - written by Prof. Anders Malthe-Sørenssen - in section 1.1. It motivates for the need of having models of gas flow at the nanometer scale. We then briefly explain the process of shale gas extraction in section 1.2 where we introduce the origin of the physical questions the work of this thesis can be used to answer. In section 1.3 the main goals of the work are presented before we in section 1.4 highlight the parts of the work that are new and original, created by the author. We conclude the chapter with section 1.5 that is a brief overview of the structure of the whole thesis.

1.1 Project description

Most of the worlds currently accessible hydrocarbon resources are found in tight rocks - rocks with permeabilities in the millidarcy range and with pore sizes in the nanometer range. The development of technologies for production from tight rocks have changed the energy landscape, making countries such as US self-sufficient with gas and possibly also with oil, and the estimates from the producible reserves of hydrocarbons in tight rocks are continuously increased as new methods are developed and new plays are discovered. However, we are now at a stage where technological and engineering methods have surpassed our basic scientific understanding of production from tight rock systems.

Tight rocks pose new scientific problems because of the small length-scales involved. Traditional oil plays are found in for example sand stone reservoirs with millimeter to micrometer sized pores. For such systems, standard hydrodynamics is a sufficient tool to understand, describe and predict fluid transport, even for multiphase systems. However, in tight rocks, typical pore sizes are in the range of tens to hundreds of nanometers. For such systems, the finite size of the atoms and molecules that make out fluids and surfaces become important: The dielectric properties of water and surface charge distributions, the binding energies of the fluids to the surfaces, and the effects of surface shapes and irregularities on effective surface interactions become important. For example, the usual assumption in fluid mechanics of no-slip boundary conditions may no longer hold, the fluids may behave differently close to surfaces than in bulk, and for smaller pores the surface to volume ratio is larger than for larger systems, and for gases the mean free path may become comparable to the characteristic sizes of the porous medium. These effects introduce challenges in how to describe and model fluid flow and surface reactions in tight rocks.

We have initiated an activity in tight rocks to address tight-rocks-specific effects for enhanced hydrocarbon production and CO2 storage. A part of that initiative requires the development of better models to address fluid flow, both liquids and gases, in tight rocks geometries with a particular focus on shale systems. In this project, we will address fluid flow in tight rocks systems by developing models to address atomistic effects for dilute gases and water in hydrophilic systems. To do this we need different models spanning various length scales. To address the flow of dilute gases in complex geometries on nanometers to micrometer length scales we will develop a method called Direct Simulation Monte Carlo (DSMC), that models a gas through effective particles that collide with other gas particles with stochastic collision rules that conserves momentum and that interacts with surfaces through special reflection rules that can be tuned using for example theoretical, experimental or atomic scale modeling results. Such models have proven useful to address dilute gas flows in regular geometries, such as for tube and channel flows, but we need very general tools to address the complex geometries of tight rocks system as found in experimental, tomographical studies. To supplement the modeling of dilute gases, we will also need methods to address the dynamics and flow of water in small pores using atomic scale models. In that case we need to model specific materials, and we have access to a very good molecular dynamics (MD) model for the interaction between water and silicates that we plan to develop and use to address fluid flow in nanoscale geometries. In this project we will develop both a DSMC and a MD model to address gas and liquid flow in tight rocks system.

1.2 Shale gas extraction

Shale gas is a type of natural gas trapped in shales - tight rocks with pore sizes at the nanometer scale. In these reservoirs, the gas is stored inside already existing pore networks or adsorbed onto organic matter. In order to extract adequate levels of gas from such rocks, we generate fractures to increase the permeability of the rock. Today we usually use horizontal drilling; we drill down from the earth surface until we reach the shale formation, there we continue drilling horizontally inside the shale (see figure 1.1).

The fractures are generated by hydraulic fracturing, a process where millions of liters of water are injected under high pressure, cracking the shales. Proppants - small, usually spherical, ceramic particles - are mixed with water, keeping the fractures open even after the pressure is released. The gas then flows from the nanosized pores into the fracture network through very small channels. Even after the pressure is released, the pressure inside the shale formation remains higher than the surface pressure. This makes the gas flow through the drilling hole to the earth surface. The process of shale gas extraction couples physics at length scales spanning over 10 orders of magnitude. In this thesis, we focus on the smallest scale: the scales at which the gas is produced. We will study flow of dilute gases in nanoporous media.

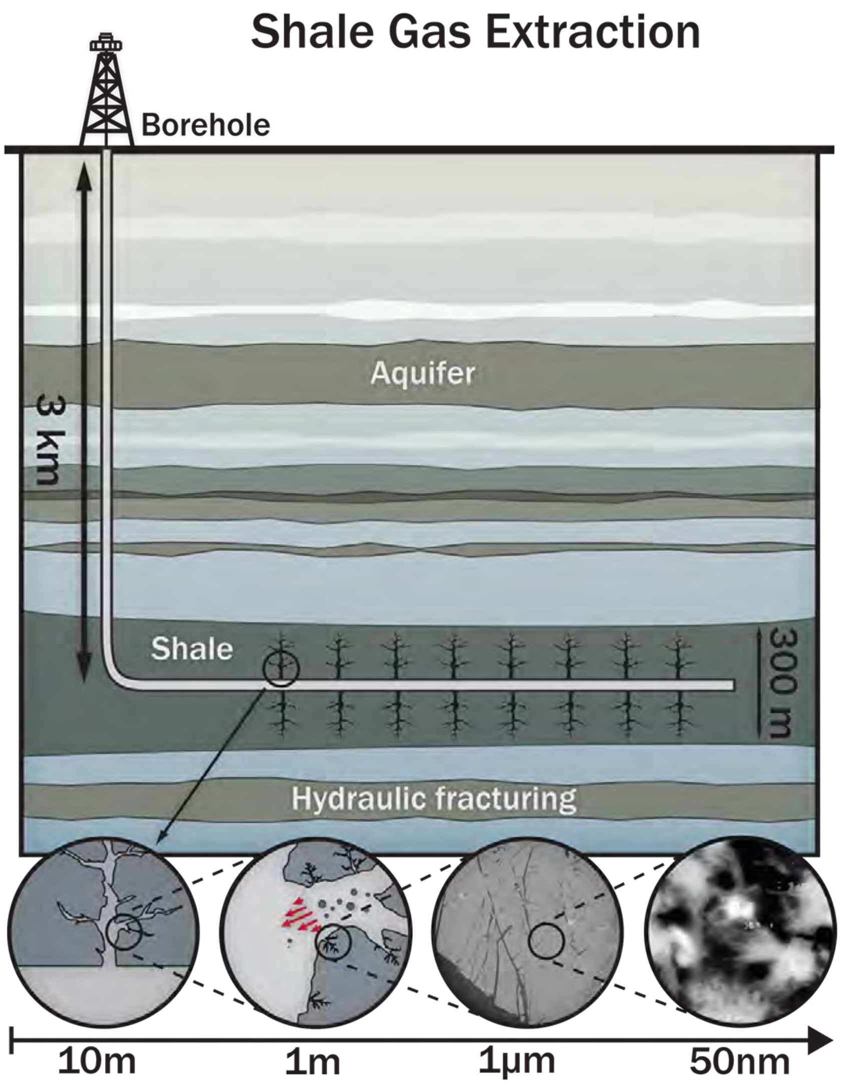

Figure 1.1: Shale gas extraction principles. Hydraulic fracturing cracks the shales, releasing the trapped gas. In this process, physical phenomena on length scales from nanometers to meters may be relevant. The production occurs in nanometer pores, and the gas gathers in the drilling hole through the network of larger fracture. From the drilling hole, the gas flows to the surface. Illustration: Sigve Bøe Skattum.

1.3 Goals

We have chosen the goals of this work so that they will lead to a solid base understanding of how the physics of fluids works at the nanometer scale. On the way, we develop many of the relevant tools. These tools can be used in further studies in the field.

The following were the main goals of this thesis:

a) Develop a three-dimensional parallel DSMC model

The DSMC model has been used for the past 50 years to study flow of dilute gases in systems where the mean free path is of the same order as a characteristic size of the geometry. Having an implementation of this model will allow us to simulate flow in systems with channels so small that continuum mechanics do not longer produce correct physical behavior. The implementation needs to be parallelized for large-scale parallel machines.

b) Develop methods to model arbitrary 3D geometries

Important systems for theoretical purposes are for example cylinders or other simple, mathematically well-described geometries. They can in many cases have analytical solutions and be excellent test cases for a more general numerical model. However, real fracture networks are usually defined by a complicated geometry with no closed form mathematical description. We therefore need to find a way to represent arbitrary geometries allowing us to measure flow properties in more realistic systems.

c) Develop a three-dimensional parallel MD model

Different surface materials may interact very differently with the same fluid. Take for example water. Some surfaces are *hydrophilic*, which means that they attract water, whereas *hydrophobic* materials tend to repel water. This will of course affect how the fluid flows through a system. The DSMC model is a particle model that performs stochastic collisions with collision rules that use parameters depending on the combination of the specific surface material and fluid substance. With a good MD model, we can both study flow in small systems and compute the gas-surface parameters we need in DSMC.

d) Develop custom 3D visualization tools for large particle data sets

Both models produce time trajectories of the particles from a given initial state. The output data is a set of particle positions which can be visualized to learn more about the fluid dynamics. By building such visualization tools from scratch, we can overcome the drawbacks that are in the already available free software (these drawbacks are discussed later) and create custom features that satisfy our needs.

e) Study flow and dynamics of water in simple nanoscale silicates

With an advanced atomic potential in MD, we can study how water flows in nanoscale silicates [25]. Such a potential can for example be used to study how hydroxyl groups on the surface of the silicate affect the water flow. This can also be used to produce gas-surface parameters that enable us to study larger scale systems with the same statistical surface behavior as water in silicates.

f) Investigate how slip velocity affects the permeability in nanoporous media

It is well known from experiments that the measured permeabilities in nanoporous media can be two orders of magnitudes higher than the theoretical predicted values. Klinkenberg explained this effect by the fact that the fluid can have a non-zero velocity near the boundaries [16]. Slip velocity leads to a correction to the theoretically predicted permeability. We will study flow in different nanoporous media to see how well these corrections predicts the permeability in the stochastic DSMC model as well as the MD model for systems with different pore sizes covering two orders of magnitudes.

1.4 My contribution

In every thesis, as in any other scientific work, the foundation of the produced content is results from previous work. It should be clear what new ideas the author has contributed with. This could for example be new theoretical calculations, models, algorithms or tools that has been developed. Since a master thesis is a larger document containing more information than just the new contributions, it might be less obvious which parts that are the unique work of the author. Such a document deserves its own section highlighting these parts.

Both models we have studied in this thesis have been programmed and implemented from scratch. A total of approximately 20000 lines of code have been written in C++ and `Python`. Writing everything from scratch provides a great insight in both models, especially from a numerical perspective, since every detail of the implementation has to be understood. In this section we briefly discuss the contributions by the author. This section is not meant to be an introduction to any of the concepts, so it is assumed that the reader is familiar with the models at the time of reading. If this is not the case, everything in this section should be clear after reading about both models and the visualization program in chapters 4, 5, 7, 8 and 10.

1.4.1 Direct Simulation Monte Carlo

In the DSMC model, we need to represent the geometry of the system (of which the fluid is confined in). This method has to be fast, scalable (for parallelizing) and general so that we can represent an arbitrary geometry. To be able to perform collisions between the surface and the particles, the surface needs to have well defined normal and tangent vectors in every available collision point. The author has developed a voxelation model with a new algorithm to compute the normal and tangent vectors based on the neighboring voxels. This gives a smoother effective surface than just voxel values. We discuss this model in detail in section 5.1.

1.4.2 Molecular Dynamics

The MD code is a standard, but remarkably efficient code using the Lennard-Jones potential. The code structure and parallelization technique is based on what is teached at the University of Southern California [1]. In section 8.4 we discuss that we want to simulate a fluid in an arbitrary geometry with the MD code. The author has developed a simple, but very promising model of a solid that allows a set of atoms to behave as a solid, vibrating about their equilibrium point while interacting with the fluid with realistic atomic forces. In addition, with an applied thermostat on these atoms, we are able to drain the system for energy which is a necessity when we induce flow in the system by adding a constant force as described in subsection 4.5.3. The details of this model is discussed in section 8.4.

[1]See http://cacs.usc.edu/education/cs596/ParallelMD-VG.pdf for details.

1.4.3 Visualization

The free visualization software available, such as VMD and Ovito, are great tools for studying data sets from atomic or molecular models. There are two significant drawbacks the author has noticed; the camera control and performance. In both of these programs, the camera is looking towards a point, usually in the center of the system, whereas the mouse controls the rotation around this point. To be able to study different parts of the system in detail, one could want to control the camera as in a first person shooter game. The author has, together with Svenn-Arne Dragly, developed visualization tools allowing us to visualize up to 100 million atoms simultaneously with decent frame rate with the camera control described above. In chapter 10 we discuss how to make full use of *geometry shaders*, which allow us to render the spheres representing the atoms on the Graphical Processing Unit (GPU).

1.5 The structure of this thesis

This document is arranged in five parts. Chapter 2 opens with a brief discussion about the theory of fluid mechanics. We discuss how the continuum approach in standard hydrodynamics breaks down for dilute gases in nanoporous media, and what the current theory has to offer in predictions of the permeability. The largest subject of this study is the DSMC model in part I. It begins with an introduction to kinetic theory in chapter 3 which we use in 4 when we introduce the DSMC model. The implementation of the model is explained in chapter 5 with the numerical results presented in 6.

In part II, we discuss MD, the second model we have studied. We begin by introducing the theory behind MD in chapter 7 with the implementation in chapter 8. The results are presented in chapter 9. The final part is concerned with our desire of creating a custom 3D visualization tool that we can use to visualize the simulations we perform with both models. We start by a brief introduction to OpenGL in chapter 10 where we explain graphics programming and how the graphics card can be used to draw geometrical models on a computer screen. In chapter 11 we explain how to use advanced shaders in the rendering pipeline to render the time evolution of tens of millions of atoms on the screen. We also discuss the marching cubes algorithm that is used to render the surface defining the geometry in a DSMC simulation.

Chapter 2

Theory of fluids

Most people have an intuition about what a substance is. A substance is just *something*, like water, wood, butter or glass. A metal is a substance. The blood in your body is a substance. All of these are made up of atoms that form molecules in ways giving them very different properties. We know that a substance can exist in different phases, for example plasma, solid, liquid and gas [23], in which the same substance can behave very differently. Liquids and gases are often called fluids because they share the property that they do not have a fixed shape and are often easily deformed. The theory that describes how fluids behave is called fluid mechanics.

We start this chapter with section 2.1 where we discuss the concept of continuum. This leads to the Euler equations and the Navier-Stokes equations in section 2.2. These equations can be used to study the behavior of fluids in motion relative to the material confining the fluid - fluid flow. We then introduce the concept of porous media, a solid material with parts of its volume - the pore space - available for fluids. Fluids can flow through such a material, and the equation describing the flow rate as a result of some pressure gradient is called Darcy's law. One of the parameters in Darcy's law is called permeability. This quantity is discussed in section 2.5.

Pore space with channels at the nanometer scale introduces a distinction between what we call macroflows and microflows. This leads to what is called the breakdown of continuum, which is addressed in section 2.7. The Knudsen number is discussed in section 2.8. It is used to quantify whether or not continuum models can be used, in addition of measuring the importance of slip velocity which has major consequences for the permeability. This is discussed in section 2.9. We then discuss particle models - which are not based on the assumption of continuum - in section 2.10. The last two sections are concerned with corrections of the permeability. These are called Klinkenberg correction and Knudsen correction, and use different models for slip velocity to predict the increase in permeability.

2.1 The continuum

In reality, we know that a fluid is composed of an enormous number of atoms which are separated by mostly empty space. The atoms interact through quantum mechanics that allows molecules to form. Even though this is the true nature (well, every physicist should be careful to say that something really is the *true nature*) of the fluid, it turns out that we can describe it as a continuum, that is, we assume that the mass is continuously distributed in the total volume of the fluid. This also means that macroscopic quantities like temperature, density, energy and the fluid velocity are well defined in *every* point in space. This is great, continuum mathematics is so much easier to work with than discrete mathematics. Calculus tells us that well behaved functions (as we prefer to work with) can both be integrated and differentiated. They also treat us rather nice, they do what we expect them to do.

In physics we strongly believe in conserved quantities. A fluid in motion will internally (far away from the boundaries in the container) have conserved energy, mass and momentum. This can be formulated beautifully with mathematics, and gives rise to what we today know as the Euler equations and the Navier-Stokes equations.

2.2 The Euler equations and the Navier-Stokes equations

With the concept of continuum, we think that every point in space has well defined physical properties like temperature, density and momentum. In 1757, Euler published what's called the *Euler equations* by applying conservation of mass, momentum and energy to a small volume element dV of a fluid. They form a set of differential equations describing how the fluid velocity $\mathbf{u}(\mathbf{r}, t)$ field changes in time and space. The conservation laws can be written as

$$\frac{\partial \rho_m}{\partial t} + \nabla \cdot (\rho_m \mathbf{u}) = 0 \qquad \text{mass} \tag{2.1}$$

$$\frac{\partial \rho_m \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u} \otimes (\rho_m \mathbf{u})) + \nabla P = 0 \qquad \text{momentum} \tag{2.2}$$

$$\frac{\partial E}{\partial t} + \nabla \cdot (\mathbf{u}(E + P)) = 0 \qquad \text{energy,} \tag{2.3}$$

where ρ_m is the mass density, $\mathbf{u}(\mathbf{r}, t)$ is the velocity field, $E(\mathbf{r}, t)$ is the total energy per unit volume, $P(\mathbf{r}, t)$ is the pressure field and $\otimes$ is the tensor product. These can be combined to one vector equation, but its origin, the connection to conservation laws, is more clear when they are separated like this. In the original paper, Euler only derived the first two equations, but the full set is usually referred to as the Euler equations. They describe the motion of fluids with *negligible viscosity*, which is a decent approximation for many fluids.

Some 80 years later, in the 1840s, Sir George Stokes published the Navier-Stokes equations (NSE) which can be seen as an extension of the Euler equations where effects caused by the viscosity of the fluid are included [4]. The Navier-Stokes equations for an incompressible fluid can be written as one vector equation

$$\rho_m \frac{D\mathbf{u}}{Dt} = \rho_m \mathbf{F} - \nabla p + \mu \nabla^2 \mathbf{u} + \mu \nabla(\nabla \cdot \mathbf{u}) \tag{2.4}$$

where $\mathbf{F}$ is an external force (i.e. gravity or an electrostatic field), μ is the viscosity and D/Dt is the material derivative defined as

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla. \tag{2.5}$$

The NSE have quite a few interesting analytically solvable solutions, but for most real systems, the geometry confining the fluid is so complicated that it is usually solved on computers. When solving the NSE, we have to provide boundary conditions to get a unique solution for the system. If we at one end of the container apply some pressure P_0, and some other value P_1 in the other end, we get a flowing fluid since there acts a net force on the fluid. This defines the pressure difference ΔP. We then also specify the velocity at the boundary which often is the no-slip boundary condition, i.e. that the fluid velocity is zero at the container walls. It turns out that for real fluids, this is not always true. In section 2.9 we discuss the effects of slip velocity and how this affects the flow properties of the fluid.

2.3 Flow in porous media

Fluid flow can be defined as fluid in motion relative to its container - the material confining the fluid. A material with parts of its volume available for fluids is called a *porous medium* (see figure 2.1).

We call the larger regions available for fluids *pores* whereas channels connecting these pores are called the *pore network*. All available such space is called the *pore space*. If the porous medium has a total volume V, and the pore space takes up a volume V_p, we define the *porosity* ϕ as

$$\phi = \frac{\text{Pore space volume}}{\text{Total volume}} = \frac{V_p}{V}. \tag{2.6}$$

When a fluid flows through the material, the amount that flows through a surface per unit time is called the *volumetric flow rate* and is usually denoted by Q. This quantity measures how many cubic meters of fluid we can push through a surface orthogonal to the flow direction per unit time. If we increase the pressure difference, we expect a higher flow rate. This is indeed true. In fact, the volumetric flow rate is proportional to the pressure difference.

2.4 Darcy's law

When we apply a pressure difference on each side of a material filled with a fluid, the fluid will start to flow in the direction of lower pressure. In 1856, H. Darcy found a linear relation between the pressure difference and the fluid flow rate. This relation is called Darcy's law and tells us

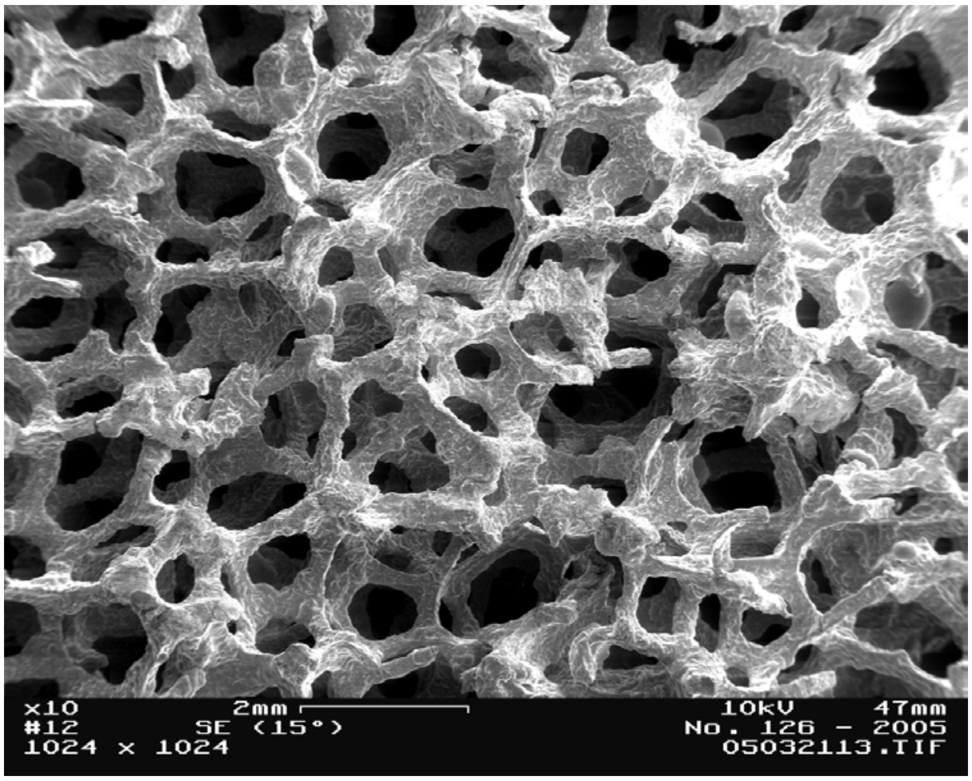

Figure 2.1: An example of a porous medium. Here we see a metal foam - a solid metal with pore space available for a fluid. A nanoporous medium is a porous medium where the size of the pores are at the nanometer scale. Image from http://en.wikipedia.org/wiki/File:Metal_Foam_in_Scanning_Electron_Microscope,_magnification_10x.GIF, accessed 28 March, 2014.

what volumetric flow rate Q we can expect from an *incompressible* fluid through a material of length L, when we apply a pressure difference ΔP, see figure 2.2. The one dimensional version of Darcy's equation is given as

$$u = \frac{Q}{A} = \sigma_D \frac{\Delta P}{L}, \tag{2.7}$$

where u is the *volumetric flux* (volumetric flow rate per unit area), $\Delta P = P_0 - P_L$ is the pressure difference, A is the cross sectional area; the area of the material orthogonal on the flow direction and L is the length of the material in the flow direction. σ_D is the proportionality constant that can be written as

$$\sigma_D = \frac{k}{\mu}, \tag{2.8}$$

where μ is the viscosity and k is the permeability.

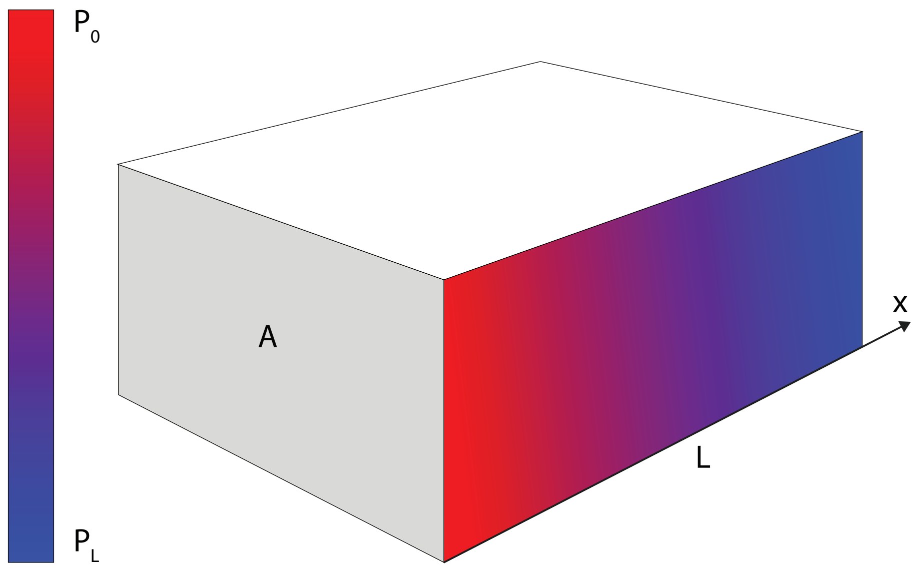

Figure 2.2: A box with volume $V = LA$ with fixed pressure values at $x = 0$ and $x = L$. The volumetric flow rate Q through a cross sectional area A is given by Darcy's law in equation (2.7) .

2.5 Permeability

The motivation of introducing the concept permeability is to separate the proportionality constant into two parts; one that depends on the liquid only, the viscosity μ, and the permeability, a material specific constant k. This means that we in principle can do an experiment with a liquid with known viscosity, say water, and measure the permeability of some material (Darcy studied a sand filled cylinder in his original experiment). Once you know the permeability, you are able to predict the flow rate through the material for *any* other liquid with a well known viscosity. This is of great importance for i.e. the oil industry where they ideally would like to take a sample of the rock in which the oil or gas is confined, measure the permeability with e.g. air, and then use this to predict the recovery rate.

This is of course not completely true in all circumstances. While Darcy originally found the relation as an empiric equation based on experiments, it can be derived from the Navier-Stokes equations. Darcy's law is only correct if the fluid flow satisfies the no-slip condition.

2.6 Macroflows and microflows

In the 1990s H. Bau and J. Zemel performed experiments on microchannel flow in which they found clear deviations from what was expected from the theory[15]. It is useful to introduce the terms *microflows* for flow in geometries where the distance between the channel walls is of order micrometer or smaller, and *macroflows* for larger systems (millimeter and above). Flow at microscales differ from macroscales because of effects that can be classified into four groups

- non-continuum effects,
- surface-dominated effects,
- low Reynolds number effects, and
- multiscale and multiphysics effects.

In this thesis, we focus on the non-continuum effects which are briefly discussed in section 2.7, and the surface-dominated effects as the slip condition described in section 2.9. See [15] for details about the effects of low Reynolds number, multiscale and multiphysics.

2.7 The breakdown of continuum

As we discussed in section 2.1, a fundamental assumption in the NSE is that the space is continuous, but we know that in reality, the mass of the fluid is concentrated in the center of the atoms. We often assume that the mass is uniformly distributed in the volume element of which the conservations laws are applied on. This is known as the *continuum hypothesis* and is invalid when the *mean free path* λ, the average distance a particle moves between collisions, becomes comparable to some characteristic length L in the system, i.e. the diameter of a channel[15]. This is quantified through the *Knudsen number*

$$\mathrm{Kn} = \frac{\lambda}{L}. \tag{2.9}$$

From the kinetic theory we can calculate the mean free path (this is done in section 3.7)

$$\lambda = \frac{m}{\sqrt{2}\pi d^2 \rho_n} = \frac{k_B T}{\sqrt{2}\pi d^2 P}, \tag{2.10}$$

where ρ_n is the number density and m and d are the mass and diameter of the particles. By using the ideal gas law $P = \rho_n k_B T$, we can replace the density with the pressure P and the temperature T. Here k_B is Boltzmann's constant. Molecular Dynamics simulations have shown large fluctuations in temperature and density near the wall in layers a few mean free paths from the surface. Such effects are not reproduced in models based on continuum equations [15]. Another very important effect, as we will discuss in the next subsection, is the non-zero slip velocity of the gas. However, for Knudsen numbers smaller than 10^{-2}, the continuum hypothesis is valid and we can use continuum equations like the NSE and the Euler equations. It turns out that the Knudsen number is very useful.

2.8 Knudsen number

The Knudsen number is the ratio between the mean free path - the average distance a particle moves between colliding with another particle - and a characteristic length in the system. This length could be the radius of a cylinder or the radius of a spherical pore or some other length scale that is representable for the system. In other words, it can be related to how often a particle collides with the surface compared to other particles. A Knudsen number larger than one indicates that a particle travels longer before it collides with another particle than the distance to the surface which means that surface effects are of increasingly importance as the Knudsen number increases.

To get an idea of the length scales where the Knudsen number is about unity, we first need to calculate the mean free path. An oxygen atom in air at room temperature has a mean free path [10]

$$\lambda_{\text{Air}} = 8.0 \times 10^{-8}\ \text{m}, \tag{2.11}$$

whereas the mean free path in water is

$$\lambda_{\text{Water}} = \times 10^{-11}\ \text{m}. \tag{2.12}$$

This means that air in a nanoporous media with typical pore size about 80 nm, the Knudsen number is of order unity and the continuum hypothesis is invalid. Water in the same system has a Knudsen number approximately 1×10^{-4} and continuum models should in principle hold.

2.9 Slip velocity

The usual boundary condition we apply when solving the NSE is the no-slip condition where

$$\mathbf{u}(\mathbf{r}; t) = 0 \qquad \mathbf{r} \in \partial\Omega, \tag{2.13}$$

where $\partial\Omega$ defines the boundary domain. The history of the no-slip condition was studied by Day[9], based on the work of Stokes in the 19th century. Stokes compared theoretical results to experiments for pendulums of different kinds and concluded that

> I shall assume, therefore, as the conditions to be satisfied at the boundaries of the fluid, that the velocity of a fluid particle shall be the same, both in magnitude and direction, as that of the solid particle with which it is in contact. The agreement of the results thus obtained with observation will presently appear to be highly satisfactory. (Stokes, 1901)

In Day's detailed study of the no-slip condition, he says

> Looking back, it appears that the acceptance of a more general no-slip condition was prolonged because of experimental shortcomings, not because of a lack of the ́appropriatétheoretical solutions to fluid flow problems. (Day, 1990)

In other words, the theoretical framework that existed already in the time of Stokes were complete enough to include both slip and no-slip solutions. In fact, Maxwell predicted slip velocity

in a paper already in 1867[17], but the experiments the next 50 years seemed to more or less confirm the no-slip condition.
That a fluid has a slip velocity is rather obvious when reading Klinkenberg's nice argument

> Consider a layer adjacent to the wall which is thinner than the mean free path λ of the gas molecules, so that practically a molecule does not collide with other molecules present in this layer. At a given moment half of the gas molecules in this layer will have a component of velocity moving towards the wall; the other half in the opposite direction. The molecules moving towards the wall have had their last collision somewhere in the flowing mass, and, therefore, will have an average velocity component in the direction of flow different from zero. A part of this average velocity component will be lost in colliding with the wall. Even if the molecules lose it entirely, then still the average velocity component in the direction of flow of all the molecules contained in the layer will amount to half of the average velocity component of the molecules moving towards the wall. The gas in the layer, therefore, will have a finite rate of flow. (L.J. Klinkenberg, 1941)

It is convenient to introduce the concept of *slip length* l_s to be able to quantify the slip velocity. Slip length is defined as the distance into the wall we would have to extrapolate a velocity profile for it to reach zero value, see figure 2.3. Maxwell theory predicts the following relation between

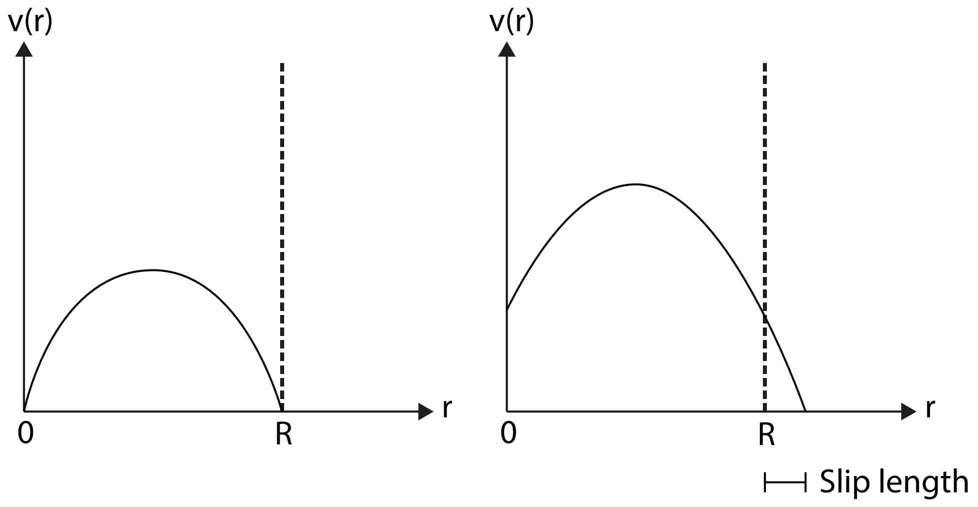

Figure 2.3: Slip length is the distance into the wall we would have to extrapolate a velocity profile for it to reach zero value. We have the no-slip condition on the left, where the slip length is zero whereas we have a non-zero slip length on the right.

the slip length and the mean free path

$$l_s = \alpha\lambda, \tag{2.14}$$

where $\alpha \approx 1.15$ is the slip coefficient [20]. The effects of slip velocity become more apparent when the channel diameter is of the same order as the mean free path. By introducing the dimensionless slip length

$$l_s^* = \frac{l_s}{L} = \alpha \frac{\lambda}{L} = \alpha \mathrm{Kn}, \tag{2.15}$$

we see that the ratio of the slip length to the channel diameter is proportional to the Knudsen number. The actual slip velocity (the average velocity of the molecules right next to the wall) can be written as

$$v_{\text{wall}} = \alpha \lambda \frac{\mathrm{d}v}{\mathrm{d}n}, \tag{2.16}$$

where n is the direction normal on the wall[16]. We call this a *first order* slip model since it is contains only the first derivative of the velocity. Higher order models exists and give corrections to the permeability that are important in nanoporous media, where the channels that contribute to flow are of nanometer scale. This is discussed in section 2.13.

2.10 Particle models

For systems where the continuum hypothesis is invalid, we need other models describing the behavior of the particles in our system. The first idea that might pop our minds might be to study the system at the atomic level. The equations of motion and hence the dynamics of a system can in principle be calculated directly from quantum mechanics by solving Schrödinger's equation. Since this requires calculating the wave function of every atom with complex atomic interactions, the size of the system needs to be very small with today's computers.

An alternative, popular approach is to use a parameterized potential $U(\mathbf{r}^N)$ ($\mathbf{r}^N$ being the positions of all atoms), and calculate the forces through the gradient of U. Newton's equations of motion is then integrated and the dynamics of the system are determined in a classical, deterministic way where important effects from quantum mechanics are embedded in the potential. This method is called *Molecular Dynamics* and is studied in chapter 7. Molecular Dynamics is orders of magnitudes faster than models solving Schrödinger's equation, but it still needs a detailed description of the dynamics of every atom in the system. For many problems, this information is redundant because what's really important is the statistical properties of the system.

In statistical mechanics, we don't need the full information about every single atom. We can then develop models using statistical mechanics and save a lot of computation power compared to Molecular Dynamics. One such model is called Direct Simulation Monte Carlo and is studied in chapter 4. The fundamental equation in this case is the Boltzmann equation, which we derive in section 3.4. In the limit of low Knudsen numbers, these models do of course converge towards the continuum models. It is convenient to classify different flow regimes depending on the Knudsen number.

2.11 Flow regimes

We can divide the entire Knudsen range into different regimes enabling us to get an overview of which equations and models that are valid for which Knudsen numbers. In the low Knudsen number limit, the continuum hypothesis is valid and we can in principle use any of the models we have discussed. The continuum approach is here of course preferable since it more computationally efficient compared to the particle models. At some point (Kn$\geq$ 0.01), the no-slip boundary condition is invalid and we need to incorporate this into the models solving the NSE in order to get accurate solutions. Here starts the slip regime. When the Knudsen number approaches 0.1, we start to see transitional flow where the flow is laminar near the edges and turbulent in the middle of the material, before we at Knudsen numbers larger than 10 have free molecular flow where the particles almost never interact with each other. This is illustrated in figure 2.4 where we have included the regions where different equations are valid.

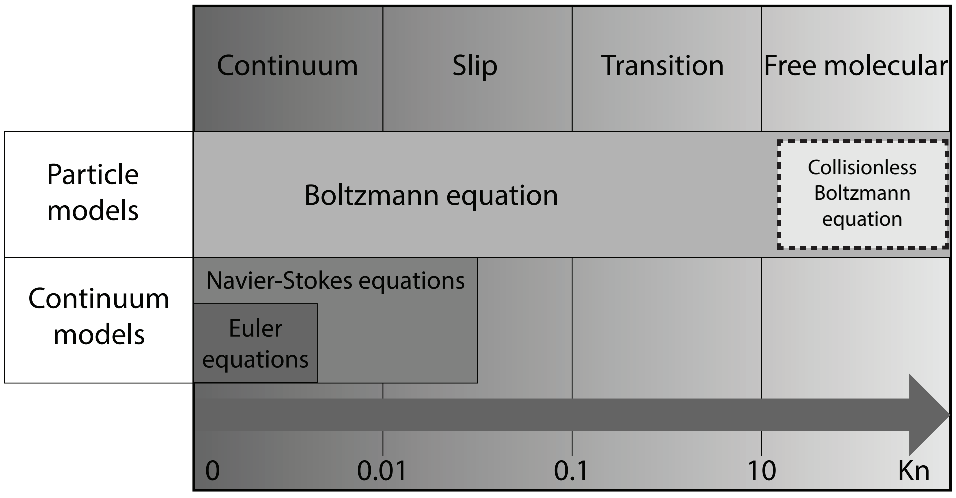

Figure 2.4: The four flow regimes covering the important regions in the Knudsen number range where different flow types appear. In the low Knudsen number limit, the fluid can be assumed to be a continuum and we can use equations like the Euler equation or the NSE. For larger Knudsen numbers, the no-slip boundary condition is invalid and we need a model satisfying slip velocity. We reach the transition flow regime at Kn$\approx$ 0.1 where the continuum models do no longer hold, even with slip boundary conditions. In the high Knudsen regime particle collisions are so rare that it is classified as free molecular flow. In this range, the collisionless Boltzmann equation is valid (see section 3.4).

2.12 The Klinkenberg effect

In 1941, L.J. Klinkenberg published a paper explaining the discrepancies that was found in experiments when measuring the permeability for both liquids and gases in the same material[16]. His discoveries were of great importance in the oil industry

It has become common practice in the oil industry to determine the permeability of core material with dry air; the equipment usually employed for this determination is arranged to operate with the outlet of the sample at or near atmospheric pressure.
(Klinkenberg, 1941)

He introduced a linear scaling function $f_c(\mathrm{Kn})$ that relates what he called the *apparent permeability* k_a - the permeability measured in the lab for a fluid with arbitrary density - to the *absolute permeability* k_∞, the permeability for a liquid in the high density limit. The absolute permeability is a constant dependent only on the porous medium. The relation is given as

$$k_a = f_c k_\infty = \left(1 + 4\alpha\frac{\lambda}{L}\right) k_\infty = (1 + 4\alpha\mathrm{Kn})\, k_\infty, \tag{2.17}$$

where L is the diameter of the channel and $\alpha \approx 1.15$ is the no-slip factor in equation (2.14). Since the mean free path is proportional to the inverse pressure, we can write the scaling function as

$$f_c = \left(1 + \frac{b}{\bar{P}}\right), \tag{2.18}$$

where b is a constant depending on the material. In figure 2.5, we see that the Klinkenberg correction factor predicts a permeability that almost fifty times higher for a gas in a material with Kn = 10. The figure also contains the Knudsen correction factor which is discussed in the next section. The Klinkenberg correction factor is derived using the first order slip velocity in equation (2.16) which is not sufficient for high Knudsen numbers as we will see in section 6.3. Based on higher order slip velocity models, one can derive better permeability corrections.

2.13 Knudsen's correction

Beskok and Karniadakis (1999) developed another correction factor that uses a second order slip velocity model

$$f_c = [1 + \alpha(\mathrm{Kn})\mathrm{Kn}]\left[1 + \frac{4\mathrm{Kn}}{1 + \mathrm{Kn}}\right], \tag{2.19}$$

where $\alpha(\mathrm{Kn})$ is given as[7]

$$\alpha(\mathrm{Kn}) = \frac{\alpha_0}{1 + \frac{A}{\mathrm{Kn}^B}} \tag{2.20}$$

where $A = 0.170$, $B = 0.4348$, and $\alpha_0 = 1.358$. These are fitted parameters based on a large dataset of Loyalka and Hamoodi (1990). We see in figure 2.5 that the Knudsen factor predicts approximately 40% higher correction than the Klinkenberg factor. In chapter 6 we will check the validity of this correction factor for a simple cylinder, and discuss the practical problems we meet when studying flow in complex geometries where the system does not simply have one Knudsen number, but rather a distribution of Knudsen numbers.

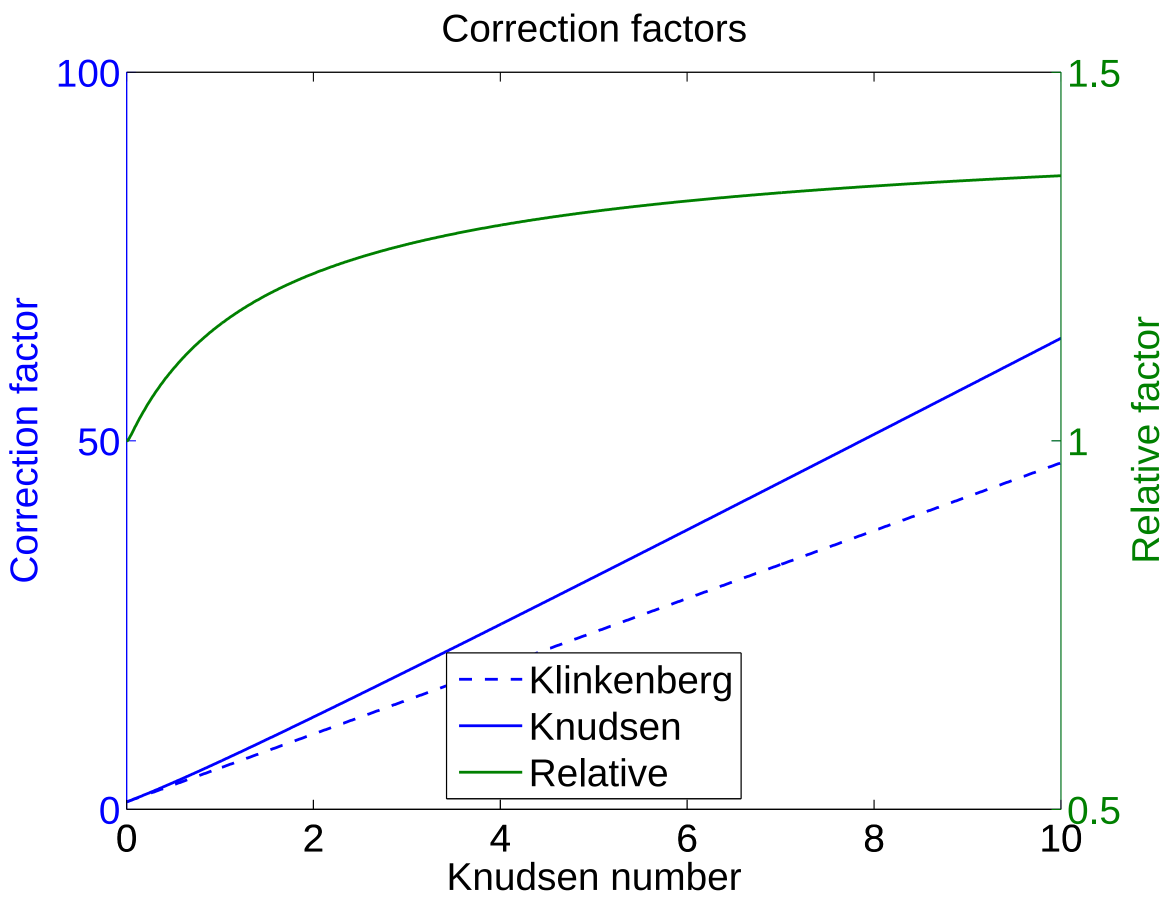

Figure 2.5: A comparison between the Klinkenberg factor and the Knudsen factor shows that slip velocity leads to even higher corrections to the permeability than what the Klinkenberg factor predicts. We see that in the high Knudsen number limit (Kn = 10), we can expect an increase of permeability by a factor 50 compared to a liquid or a high density gas. The line labeled *Relative* shows the ratio between the Knudsen correction and the Klinkenberg correction factors.

Part I

Direct Simulation Monte Carlo

Chapter 3

Kinetic theory

Before we describe the first model, the Direct Simulation Monte Carlo, we need to discuss some theory. The model is based on the kinetic theory of gases, which is a microscopic theory that describes the behavior of gases at the molecular level. A system of N particles is fully described by the $3N$ momentum components combined with the $3N$ spatial coordinates. Together, this forms a $6N$ dimensional phase space where each point represents the state of the system in an ensemble. We start this chapter by introducing the distribution function, the concepts of microstates, macrostates and ensembles in section 3.1, before we in section 3.2 explain how we measure the macroscopic observables which are average values over all the states in an ensemble. Then we have a brief discussion about ergodicity in section 3.3 which is a very important assumption when we start measuring physical quantities in a numerical statistical mechanics model such as Direct Simulation Monte Carlo (chapter 4) and Molecular Dynamics (chapter 7). In section 3.4 we derive the Boltzmann equation which is the fundamental equation that governs the behavior of the distribution function. We then define what an equilibrium state is which we use to derive the Maxwell-Boltzmann velocity distribution in section 3.6. As we might remember, the Knudsen number is an important dimensionless quantity that we use to quantify how important surface and non-continuum effects are. The Knudsen number is the ratio between the mean free path λ and some characteristic length L of the system. In section 3.7 we calculate the mean free path which is used to compute the mean collision time τ_{coll}, which is important to choose a good timestep in the Direct Simulation Monte Carlo model in chapter 4.

3.1 The distribution function

A point $(\mathbf{r}, \mathbf{v})$ in the phase space describes what is called a *microstate* and contains a massive amount of information. Given this point, we would know the position and velocity to *every* particle in the entire system. In a liter of an ideal gas under standard pressure, the number of particles is of order 10^{22} [12], so if each of these $6N$ coordinates were represented as an 8 byte *double* on a computer, we would need more than 10^{11} terabytes of memory just to store all the information. This approach would be very inconvenient and, fortunately, not at all necessary. The really interesting properties in a system are the macroscopic ones, like energy, temperature, pressure, volume, average velocity among others. For example, the total energy in a gas consisting of N particles is calculated as

$$E = \sum_{i=1}^{N} \frac{1}{2} m_i v_i^2 + V(\mathbf{r}),$$

where m_i is the mass of particle i, v_i is its scalar velocity and $V(\mathbf{r})$ is the total potential energy in the system depending on the full $3N$-dimensional spatial coordinate $\mathbf{r}$.
Given a microstate, what happens if we switch two particles, say particle i and j? If particle i had velocity $\mathbf{v}_i = \mathbf{u}$ and particle j had some other velocity $\mathbf{v}_j = \mathbf{w}$, we could quickly swap them so that $\mathbf{v}_i = \mathbf{w}$ and $\mathbf{v}_j = \mathbf{u}$ (theoretically of course, it would be a difficult task in an experiment). If their masses are identical, the total energy of the system would not change, but since *we* know that we switched the two particles, we could want to count this as another microstate. We could in principle paint the particles with different colors, or maybe just label them with their own unique number. However, in a real, mono-atomic gas, we can't really tell the difference if particle i and j secretly agreed to switch places without telling us. If they did so, it would not count as different microstates, the system remains exactly the same. We say that the particles are *indistinguishable.*

If we instead increase the velocity of particle i, we can reduce some another particle j's velocity to keep the total energy constant. Even though the macroscopic property *energy* is unchanged, there is a (theoretically) measurable difference between these two states. The set of all microstates that share the same macroscopic state variables (a *macrostate*) forms an ensemble of systems. A much used ensemble is the microcanonical ensemble (NVE) with a constant number of particles N, constant volume V and constant energy E. Increasing the velocity of particle i while at the same time reducing particle j's velocity just enough to remain the energy unchanged does not change the particle number N, the volume V or the energy. So these two different microstates would both be in the same ensemble.

In a typical system, the number of microstates in a macrostate is so huge that the phase space points can be described by a continuous density function $f(\mathbf{p}, \mathbf{r}, t)$ without losing any important information [19]. The input parameters are the $3N$ momentum components, the $3N$ spatial coordinates plus time. This function is often called a *distribution function*, normalized so that

$$\iint f(\mathbf{p}, \mathbf{r}, t) \mathrm{d}\mathbf{p} \mathrm{d}\mathbf{r} = N, \tag{3.1}$$

where $\mathrm{d}\mathbf{p}\mathrm{d}\mathbf{r} = \mathrm{d}p_1 \mathrm{d}p_2 ... \mathrm{d}r_{3N}$ is the $6N$ dimensional phase space volume element. This density function does not contain the information about the *exact* positions or momenta of the particles, but the *probability* to find the system in a state around a given phase space point. We can then use it to calculate measurable, macroscopic average values.

3.2 Ensemble averages

Given the distribution function f, we can calculate any ensemble average (which will be the measurable, macroscopic properties of the system) by interpreting f as a probability distribution (it needs the factor $1/N$ to be normalized to one) that gives the probability of finding a particle at position $\mathbf{r} + \mathrm{d}\mathbf{r}$ with momentum in the range $\mathbf{p} + \mathrm{d}\mathbf{p}$ at the time t. We can then use the standard expectation value expression to calculate a macroscopic property $\bar{A}$

$$\bar{A}(t) = \frac{1}{N} \iint A(\mathbf{p}, \mathbf{r}, t) f(\mathbf{p}, \mathbf{r}, t) \mathrm{d}\mathbf{p}\mathrm{d}\mathbf{r}. \tag{3.2}$$

This could for example be the total energy

$$\begin{aligned} \bar{E}(t) &= \frac{1}{N} \iint E(\mathbf{p}, \mathbf{r}, t) f(\mathbf{p}, \mathbf{r}, t) \mathrm{d}\mathbf{p}\mathrm{d}\mathbf{r} & (3.3) \\ &= \frac{1}{N} \iint \left(V(\mathbf{r}) + \sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{2m_i} \right) f(\mathbf{p}, \mathbf{r}, t) \mathrm{d}\mathbf{p}\mathrm{d}\mathbf{r}, & (3.4) \end{aligned}$$

where $\mathbf{p}_i$ is the momentum of particle i. Any other quantity of interest can in principle be measured in the same way.

3.3 Ergodicity

The ensemble average calculates the average value of some macroscopic quantity given the distribution function f. Usually, we don't have the distribution function, except in some very simple theoretical calculations. Even then, it might be difficult to compute the integral in equation (3.2). The usual situation when we do numerical statistical mechanics is that we have a way to explore the phase space, hoping that it helps us visit states with probabilities according to the given ensemble. Many Monte Carlo techniques (such as the Metropolis algorithm) allow us to go to a new, random point in the phase space, and calculate the probability of going from the current state to the new state. From this, we can count the number of times we have visited different regions of the phase space and create a histogram or maybe, if we're lucky, fit some existing probability distribution to our data.

Another approach is to let the rules of physics take us around in the phase space, from one state to another, following Newton's equations of motion. Imagine that at $t = 0$, our system is in some microscopic state (a single phase space point $(\mathbf{r}(0), \mathbf{p}(0))$) and at a later time $t = \tau$ has moved to $(\mathbf{r}(\tau), \mathbf{p}(\tau))$. Between these two points, the system has moved through many other points, exploring the phase space. It seems reasonable that in the limit of infinite time, the time evolution should visit the phase space points according to the density given by f. If not, it wouldn't make any sense even talking about f, since it is the time evolution we, as humans, are experiencing in the real physical world.

This is called the ergodicity hypothesis, the assumption that a system following the laws of physics, explores the phase space with the probability of being in a region proportional to the density in that region. The average value of a macroscopic quantity A is then found as

$$\bar{A} = \lim_{t\to\infty} \frac{1}{t} \int_0^t A(\mathbf{r}(t'), \mathbf{p}(t'), t')\mathrm{d}t'. \tag{3.5}$$

3.4 The Boltzmann equation

In the section 3.1, we introduced the distribution function f that describes the density in a $6N$ dimensional phase space. Now, if we know the distribution function at $t = 0$, we could in principle compute any property of the system. But at some later time the distribution function might have changed, unless, of course, $\mathrm{d}f/\mathrm{d}t = 0$ in which the system would be in an equilibrium state. We can, by applying conservation of probability, derive an equation of motion for f. This equation is called the Boltzmann equation and describes how f evolves through time by assuming that any change of probability around a point $(\mathbf{r}, \mathbf{v})$ at the time t must be due to

- flow through a surface in the phase space,
- an external force, or
- internal collisions.

All three will change f in different ways. For simplicity, we will first assume that the particles do not collide and derive the *collisionless Boltzmann equation*. But do not worry, we will add the collision term later and end up with the full Boltzmann equation.

3.4.1 The collisionless Boltzmann equation

Consider the density $f(\mathbf{r}, \mathbf{v}, t)\mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v}$ around the phase space point $(\mathbf{r}, \mathbf{v})$ at the time t. Then, at a time $\mathrm{d}t$ later, if we assume no forces and that the total number of particles has not changed, that *chunk* of density has moved to $(\mathbf{r} + \mathbf{v}\mathrm{d}t, \mathbf{v})$. Conservation of probability states that any change of f within a volume Ω must flow through the boundary $\partial\Omega$

$$\frac{\mathrm{d}}{\mathrm{d}t} \iint_\Omega f \,\mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v} = -\int_{\Omega_v} \mathrm{d}\mathbf{v} \int_{\partial\Omega_r} f(\mathbf{v}\cdot\mathbf{n}_r)\,\mathrm{d}S_r \tag{3.6}$$

$$= -\int_{\Omega_v} \mathrm{d}\mathbf{v} \int_{\Omega_r} \nabla_\mathbf{r}\cdot(f\mathbf{v})\,\mathrm{d}\mathbf{r} = -\iint_\Omega \nabla_\mathbf{r}\cdot(f\mathbf{v})\,\mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v} \tag{3.7}$$

which becomes

$$\frac{\partial f}{\partial t} + \nabla_\mathbf{r}\cdot(f\mathbf{v}) = \frac{\partial f}{\partial t} + \mathbf{v}\cdot\nabla_\mathbf{r} f = 0 \tag{3.8}$$

since $\mathbf{v}$ is independent of $\mathbf{r}$. We can extend this equation by adding the effects of an external force $\mathbf{F}$ that changes the velocity in the same way as the position was changed above (except for the factor $1/m$)

$$\frac{\partial f}{\partial t} + \mathbf{v}\cdot\nabla_\mathbf{r} f + \nabla_\mathbf{v}\cdot\frac{\mathbf{F}}{m} f = 0, \tag{3.9}$$

which we call the collisionless Boltzmann equation. It is a good approximation to describe the dynamics of very dilute gases where intermolecular collisions occur rarely. But we should not ignore collisions between particles, so as promised, we will now see that by treating collisions will appear as an additional term.

3.4.2 The collision operator

We consider a dilute gas so we can assume that only binary collisions occur (we ignore the contribution from collisions between three or more particles at a time). We also assume that the total energy, momentum and mass is conserved in all collisions. Then consider two particles i and j moving towards each other with velocities $\mathbf{v}_i$ and $\mathbf{v}_j$, and relative velocity $\mathbf{v}_{\text{rel}} = \mathbf{v}_i - \mathbf{v}_j$. We define particle i as the *incident* particle and j as the *target* particle. After the collision, the particles will have velocities $\mathbf{v}'_i$ and $\mathbf{v}'_j$ with relative velocity $\mathbf{v}'_{\text{rel}} = \mathbf{v}'_i - \mathbf{v}'_j$.

In order to make the calculations simpler, we change the frame of reference, in which we label the velocities with a tilde so that $\mathbf{v} \to \tilde{\mathbf{v}}$. If we choose the target particle as initial frame of reference, we see that the velocity of the incident particle becomes $\tilde{\mathbf{v}}_i = \mathbf{v}_{\text{rel}}$ and $\tilde{\mathbf{v}}'_i = \mathbf{v}'_{\text{rel}}$. Since momentum is conserved, we know that the relative velocity must remain constant, $|\mathbf{v}_{\text{rel}}| = |\mathbf{v}'_{\text{rel}}|$, during the collision. The direction of $\mathbf{v}'_{\text{rel}}$ is given by the angles ϕ and θ with $\hat{\mathbf{z}}$ along $\mathbf{v}_{\text{rel}}$ and $\phi \in [0, 2\pi], \theta \in [0, \pi]$. We can express $\mathbf{v}'_{\text{rel}}$ as

$$\mathbf{v}'_{\text{rel}} = \mathbf{v}_{\text{rel}} - 2\hat{\mathbf{e}}(\hat{\mathbf{e}} \cdot \mathbf{v}_{\text{rel}}), \tag{3.10}$$

where $\hat{\mathbf{e}}$ is an arbitrary unit vector. If we multiply by $\hat{\mathbf{e}}$, we see that

$$\hat{\mathbf{e}} \cdot \mathbf{v}'_{\text{rel}} = \hat{\mathbf{e}} \cdot \left[\mathbf{v}_{\text{rel}} - 2\hat{\mathbf{e}}(\hat{\mathbf{e}} \cdot \mathbf{v}_{\text{rel}})\right] = -\hat{\mathbf{e}} \cdot \mathbf{v}_{\text{rel}} \tag{3.11}$$

which gives the symmetric relation

$$\mathbf{v}_{\text{rel}} = \mathbf{v}'_{\text{rel}} - 2\hat{\mathbf{e}}(\hat{\mathbf{e}} \cdot \mathbf{v}'_{\text{rel}}). \tag{3.12}$$

The angle between $\mathbf{v}_{\text{rel}}$ and $\mathbf{v}'_{\text{rel}}$ is θ, so

$$\mathbf{v}'_{\text{rel}} \cdot \mathbf{v}_{\text{rel}} = v_{\text{rel}}^2 \cos\theta = v_{\text{rel}}^2 (1 - 2\cos\chi), \tag{3.13}$$

where χ is the angle between $\hat{\mathbf{e}}$ and $\mathbf{v}_{\text{rel}}$, which gives the relation

$$\theta = \pi - 2\chi. \tag{3.14}$$

We now define the solid angle element $\mathrm{d}\Omega = \sin\theta \mathrm{d}\theta \mathrm{d}\phi$ about $\mathbf{v}'_{\text{rel}}$

$$\begin{aligned}
v_{\text{rel}}\mathrm{d}\Omega &= v_{\text{rel}} \sin\theta \mathrm{d}\theta \mathrm{d}\phi = 2v_{\text{rel}} \sin(\pi - 2\chi)\mathrm{d}\chi\mathrm{d}\phi && (3.15)\\
&= 4v_{\text{rel}} \cos\chi \sin\chi \mathrm{d}\chi\mathrm{d}\phi = 4\,|\hat{\mathbf{e}} \cdot \mathbf{v}_{\text{rel}}| \sin\chi \mathrm{d}\chi\mathrm{d}\phi && (3.16)\\
&= 4\,|\hat{\mathbf{e}} \cdot \mathbf{v}_{\text{rel}}|\, \mathrm{d}^2 e, && (3.17)
\end{aligned}$$

where $\mathrm{d}^2 e = \sin\chi \mathrm{d}\chi\mathrm{d}\phi$ is a solid angle element about $\hat{\mathbf{e}}$. In the following, we will calculate the scattering cross section which is the *area* that describes the likelihood of an incident particle being scattered by the target particle. We denote the number density ρ_n and find that the incident flux is $\rho_n v_{\text{rel}}$. The rate $h'_{\text{d}\Omega}$ of scattered particles into $\text{d}\Omega$ is

$$h'_{\text{d}\Omega} = \rho_n v_{\text{rel}} \sigma \text{d}\Omega, \tag{3.18}$$

where the proportionality constant σ is the cross sectional area. We might have several target particles colliding independently of each other which will contribute to the scattering rate. If we have n_t such particles, we obtain the total scattering rate $h_{\text{d}\Omega}$ by multiplying (3.18) by n_t

$$h_{\text{d}\Omega} = n_t \rho_n v_{\text{rel}} \sigma \text{d}\Omega. \tag{3.19}$$

The *differential* cross section σ depends on $\mathbf{v}_{\text{rel}}$ and $\mathbf{v}'_{\text{rel}}$, so we denote it as $\sigma = \sigma(\mathbf{v}_{\text{rel}} \to \mathbf{v}'_{\text{rel}})$, whereas the *total* cross section σ_T is given by integrating over all solid angles

$$\sigma_T = \int \sigma \, \text{d}\Omega. \tag{3.20}$$

We will now look at particles with velocities in the range $[\mathbf{v}, \mathbf{v} + \text{d}\mathbf{v}]$ incident on target particles with velocities in the range $[\mathbf{v}_1, \mathbf{v}_1 + \text{d}\mathbf{v}_1]$. The incident flux is $gf(\mathbf{v}, \mathbf{r})\text{d}\mathbf{v}$ and the number of target particles is $f(v_1, \mathbf{r})\text{d}\mathbf{v}_1\text{d}\mathbf{r}$. The rate at which particles with velocity $\mathbf{v}_1$ are scattered by particles with velocity $\mathbf{v}$ is

$$f(\mathbf{v})f(\mathbf{v}_1)v_{\text{rel}}\sigma \, \text{d}\Omega\text{d}\mathbf{v}\text{d}\mathbf{v}_1\text{d}\mathbf{r} = f(\mathbf{v})f(\mathbf{v}_1)4\left|\hat{\mathbf{e}} \cdot \mathbf{v}_{\text{rel}}\right| \sigma \, \text{d}^2e\text{d}\mathbf{v}\text{d}\mathbf{v}_1\text{d}\mathbf{r}. \tag{3.21}$$

The rate of loss is the rate of which particles in $\text{d}\mathbf{v}\text{d}\mathbf{r}$ are being hit by other particles. We can calculate this by integrating over all solid angles d^2e and incident velocities $\text{d}\mathbf{v}$

$$\text{rate of loss} = \left[\iiint f(\mathbf{r}, \mathbf{v}, t)f(\mathbf{r}, \mathbf{v}_1, t)4\left|\hat{\mathbf{e}} \cdot \mathbf{v}_{\text{rel}}\right| \sigma \, \text{d}^2e\text{d}\mathbf{v}_1\right] \text{d}\mathbf{v}\text{d}\mathbf{r}. \tag{3.22}$$

We also have the inverse event, incident particles with velocity $\mathbf{v}'$ hitting target particles with velocity $\mathbf{v}'_1$ so that the final velocity of the target particles is $\mathbf{v}$. This is calculated with the same idea

$$\text{rate of gain} = \left[\iiint f(\mathbf{r}, \mathbf{v}', t)f(\mathbf{r}, \mathbf{v}'_1, t)4\left|\hat{\mathbf{e}} \cdot \mathbf{v}_{\text{rel}}\right| \sigma \, \text{d}^2e\text{d}\mathbf{v}_1\right] \text{d}\mathbf{v}\text{d}\mathbf{r}, \tag{3.23}$$

since $\left|\hat{\mathbf{e}} \cdot \mathbf{v}_{\text{rel}}\right| = \left|\hat{\mathbf{e}} \cdot \mathbf{v}'_{\text{rel}}\right|$ and $\text{d}\mathbf{v}'\text{d}\mathbf{v}'_1 = \text{d}\mathbf{v}\text{d}\mathbf{v}_1$. The total change in the distribution function is given by the functional $J[f]$

$$J[f] = \iiint 4|\hat{\mathbf{e}} \cdot \mathbf{v}_{\text{rel}}|\sigma[f'f'_1 - ff_1] \, \text{d}\mathbf{v}_1\text{d}^2e \tag{3.24}$$

$$= \iint v_{\text{rel}}\sigma[f'f'_1 - ff_1] \, \text{d}\mathbf{v}_1\text{d}\Omega, \tag{3.25}$$

where $f = f(\mathbf{r}, \mathbf{v}, t)$ and $f_1 = f(\mathbf{r}, \mathbf{v}_1, t)$. The full Boltzmann equation is then given by

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{r}} f + \frac{\mathbf{F}}{m} \cdot \nabla_{\mathbf{v}} f = J[f]. \tag{3.26}$$

In the derivation of the collision operator $J[f]$, we assumed binary collisions only. This is a decent approximation that holds for low densities. By defining the force range D and the dimensionless parameter $\nu = \rho_n D^3$, the Boltzmann equation is valid when ν is small [18]. D^3 defines a volume around a particle in which the forces cannot be neglected. This makes ν the average number of particles within that volume. If that number is small (less than unity) then we can safely neglect collisions between three or more particles.

3.5 H-theorem

Now we have an integro-differential equation describing how the distribution function evolves through time. We will now look at Boltzmann's H-theorem which is a powerful result that gives us all the theoretical tools we need to implement the DSMC method. We will use this to derive what velocity distribution a gas in equilibrium obeys. In addition, we will find the mean free path and the mean collision time which both will be used in the Direct Simulation Monte Carlo method. Let us define the H-function as

$$H(t) = \langle \ln f \rangle = \iint f(\mathbf{r}, \mathbf{v}, t) \ln f(\mathbf{r}, \mathbf{v}, t) \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v}, \tag{3.27}$$

and differenciate with respect to time

$$\frac{\mathrm{d}H}{\mathrm{d}t} = \iint \frac{\partial f}{\partial t} (\ln f) \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v} + \iint \frac{\partial f}{\partial t} \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v} = \iint \frac{\partial f}{\partial t} \ln f \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v}, \tag{3.28}$$

where the last term vanished since the number of particles is conserved

$$\iint \frac{\partial f}{\partial t} \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v} = \frac{\mathrm{d}}{\mathrm{d}t} \iint f \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v} = \frac{\mathrm{d}N}{\mathrm{d}t} = 0. \tag{3.29}$$

We multiply the Boltzmann equation by $\ln f$ and integrate over $\mathbf{r}$ and $\mathbf{v}$

$$\begin{aligned}\iint \ln f \frac{\partial f}{\partial t} \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v} = &- \iint (\ln f) \mathbf{v} \cdot \nabla_{\mathbf{r}} f \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v} - \iint (\ln f) \frac{\mathbf{F}}{m} \cdot \nabla_{\mathbf{v}} f \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v} \\ &+ \iint \ln f J[f] \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v}.\end{aligned} \tag{3.30}$$

The first integral on the right can be integrated by parts

$$\begin{aligned}\iint (\ln f) \mathbf{v} \cdot \nabla_{\mathbf{r}} f \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v} = &\iint f (\ln f)(\mathbf{v} \cdot \hat{\mathbf{n}}) \, \mathrm{d}\Gamma_{\mathbf{r}}\mathrm{d}\mathbf{v} \\ &- \iint f \ln f (\nabla_{\mathbf{r}} \cdot \mathbf{v}) \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v} = 0,\end{aligned} \tag{3.31}$$

where $\mathrm{d}\Gamma_{\mathbf{r}}$ indicates the boundary of the spatial domain. The integral is zero if we assume that f is zero at the boundaries and $\mathbf{v}$ is independent of r. Similarly with the second integral on the right in equation (3.30)

$$\begin{aligned}\frac{1}{m} \iint (\ln f) \mathbf{F} \cdot \nabla_{\mathbf{v}} f \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v} = &\frac{1}{m} \iint f \ln f (\mathbf{F} \cdot \hat{\mathbf{n}}) \, \mathrm{d}\Gamma_{\mathbf{v}}\mathrm{d}\mathbf{r} \\ &- \frac{1}{m} \iint f \ln f (\nabla_{\mathbf{v}} \cdot \mathbf{F}) \, \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{v} = 0,\end{aligned} \tag{3.32}$$

where $\mathrm{d}\Gamma_{\mathbf{v}}$ indicates the boundary of the velocity space. This integral is also zero since f is zero when $\mathbf{v} \to \pm\infty$ and the force is independent of the velocity. By recognizing that the left hand side of equation (3.30) actually is $\mathrm{d}H/\mathrm{d}t$, we end up with

$$\frac{\mathrm{d}H}{\mathrm{d}t} = \iiint \ln f [f' f_1' - f f_1] g\sigma \, \mathrm{d}\Omega \mathrm{d}\mathbf{v}_1 \mathrm{d}\mathbf{v}, \tag{3.33}$$

which can be written as[19]

$$\frac{\mathrm{d}H}{\mathrm{d}t} = \frac{1}{4}\iiint \ln\left[\frac{ff_1}{f'f'_1}\right][f'f'_1 - ff_1]g\sigma\,\mathrm{d}\Omega\mathrm{d}\mathbf{v}_1\mathrm{d}\mathbf{v}. \tag{3.34}$$

The integrand is of the form $-(x-y)\ln(x/y)$ which is negative for $x \neq y$ and zero for $x = y$. This means that

$$\frac{\mathrm{d}H}{\mathrm{d}t} \leq 0, \tag{3.35}$$

which is called the H-theorem. Since $H(t)$ is bounded (see equation (3.27) and remember that f indeed is bounded), in the limit $t \to \infty$, $H(t)$ reaches an equilibrium state as $\mathrm{d}H/\mathrm{d}t = 0$. This in turn means that the integrand in (3.34) must be zero as well which happens if

$$f'f'_1 = ff_1, \tag{3.36}$$

which allows us calculate the equilibrium velocity distribution.

3.6 The Maxwell-Boltzmann distribution

We will now find out which f that satisfies equation (3.36) which we rewrite by taking the logarithm on both sides

$$\ln f' + \ln f'_1 = \ln f + \ln f_1. \tag{3.37}$$

This states that the *sum* of of the logarithm of f is unchanged during a collision. As we required in subsection 3.4.2, energy, momentum and mass are conserved quantities, so $\ln f$ must be a linear combination of these

$$\ln f = \alpha m + \beta\cdot(m\mathbf{v}) - \gamma\frac{mv^2}{2} \tag{3.38}$$

$$= \alpha m + \frac{m}{2}\frac{\beta\cdot\beta}{\gamma} - \frac{m\gamma}{2}\left(\mathbf{v} - \frac{\beta}{\gamma}\right)^2, \tag{3.39}$$

for some real values α, β and γ. We combine all quantities not dependent of $\mathbf{v}$ into one parameter $\ln c$ so that

$$\ln f = \ln c - \frac{m\gamma}{2}\left(\mathbf{v} - \frac{\beta}{\gamma}\right)^2, \tag{3.40}$$

which we can write as

$$f(\mathbf{r},\mathbf{v}) = c\exp\left[-\frac{m\gamma}{2}\left(\mathbf{v} - \frac{\beta}{\gamma}\right)^2\right]. \tag{3.41}$$

By using that

$$\rho_n(\mathbf{r},t) = \int f\,\mathrm{d}\mathbf{v}, \tag{3.42}$$

and

$$\mathbf{v}_0 = \langle \mathbf{v} \rangle = \frac{1}{\rho} \int \mathbf{v} f \, \mathrm{d}\mathbf{v}, \tag{3.43}$$

in addition to the equipartition theorem

$$\frac{3}{2} k_B T = \frac{m}{2} \langle (\mathbf{v} - \mathbf{v}_0)^2 \rangle, \tag{3.44}$$

we can write equation (3.41) as [18]

$$f(\mathbf{r}, \mathbf{v}) = \rho_n \left(\frac{m}{2\pi k_B T} \right)^{3/2} e^{\frac{-m|\mathbf{v}|^2}{2k_B T}}, \tag{3.45}$$

which is the *Maxwell-Boltzmann distribution.* We can integrate out the position part of the distribution to get

$$P(\mathbf{v})\mathrm{d}\mathbf{v} = \left(\frac{m}{2\pi k_B T} \right)^{3/2} e^{\frac{-m|\mathbf{v}|^2}{2k_B T}} \mathrm{d}\mathbf{v}. \tag{3.46}$$

We will use this to validate the DSMC code section 6.1. Now, let us use this to find some interesting properties of the gas. Given a temperature T and the mass of the particles m, what is the average speed of the particles? First, we need to transform the distribution from a vector distribution to a scalar distribution (the magnitude of the velocity might be a more interesting quantity than a given direction). The distribution in equation (3.46) is spherical symmetric, so we should transform to spherical coordinates which gives

$$P(v)\mathrm{d}v = 4\pi \left(\frac{m}{2\pi k_B T} \right)^{3/2} v^2 e^{\frac{-mv^2}{2k_B T}} \mathrm{d}v. \tag{3.47}$$

The average speed of the particles can then be found as

$$\langle v \rangle = \int_0^\infty v P(v) \mathrm{d}v = \frac{2\sqrt{2}}{\sqrt{\pi}} \sqrt{\frac{k_B T}{m}}. \tag{3.48}$$

The particles in a high temperature gas moves faster than particles in a gas with low temperature, as expected. The next important quantity we should compute is the mean free path. We will need that to calculate the Knudsen number from section 2.8.

3.7 Mean free path

The mean free path λ is the average distance a particle travels before it collides with another particle. We imagine a particle with diameter d moving and find its *effective collision area* to be (see figure 3.1)

$$\sigma = \pi(2d)^2. \tag{3.49}$$

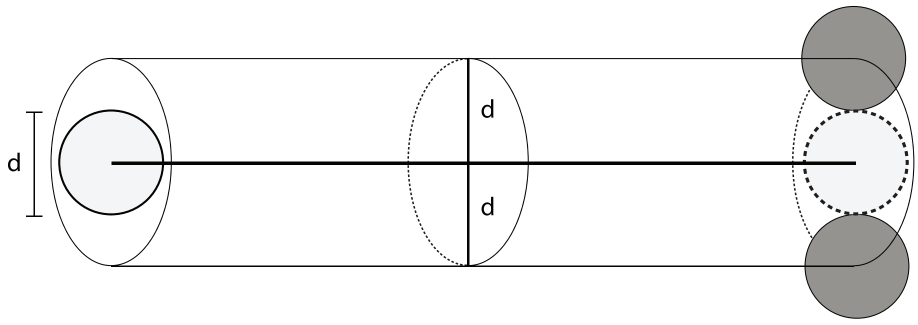

Figure 3.1: A particle with diameter d swipes out a cylinder with diameter $2d$, defining the collision area $A = \pi(2d)^2$ containing all points of which a collision partner can have its center in.

Two particles i and j, with velocities $\mathbf{v}_i$ and $\mathbf{v}_j$, have the relative velocity $\mathbf{v}_{\text{rel}} = \mathbf{v}_i - \mathbf{v}_j$. The norm is given as

$$v_{\text{rel}} = \sqrt{\mathbf{v}_{\text{rel}} \cdot \mathbf{v}_{\text{rel}}} = \sqrt{(\mathbf{v}_i - \mathbf{v}_j)(\mathbf{v}_i - \mathbf{v}_j)} \tag{3.50}$$

$$= \sqrt{\mathbf{v}_i \cdot \mathbf{v}_i - 2\mathbf{v}_i\mathbf{v}_j + \mathbf{v}_j\mathbf{v}_j}, \tag{3.51}$$

from which we can find the average relative velocity by assuming that the velocities are completely random, and hence not correlated, and that the particles have the same mean speed $\langle v \rangle$

$$\langle v_{\text{rel}} \rangle = \sqrt{\mathbf{v}_1^2 + \mathbf{v}_2^2} = \sqrt{2}\langle v \rangle, \tag{3.52}$$

During a time τ, assuming average relative velocity $\sqrt{2}\langle v \rangle$, the total volume sweeped out by particle i is

$$V = \pi d^2 \sqrt{2}\langle v \rangle \tau, \tag{3.53}$$

which in turn gives the number of collisions during such a volume

$$n_{\text{coll}} = V\rho_n = \sqrt{2}\pi d^2 \langle v \rangle \rho_n \tau, \tag{3.54}$$

where ρ_n is the number density. The mean free path is then calculated as the length of the path divided by the number of collisions

$$\lambda = \frac{\langle v \rangle \tau}{\sqrt{2}\pi d^2 \langle v \rangle \rho_n \tau} = \frac{1}{\sqrt{2}\pi d^2 \rho_n}. \tag{3.55}$$

3.8 Mean collision time

From the mean free path, it is easy to calculate the mean collision time. The mean collision time τ_{coll} is simply the average *time* a particle travels before it collides with another particle.

So, if the mean free path λ was average distance a particle will travel before a collision and the average speed of the particles were $\langle v \rangle$, the average time τ_{coll} should be

$$\begin{aligned} \tau_{\text{coll}} &= \frac{\lambda}{\langle v \rangle} = \frac{1}{\sqrt{2}\pi d^2 \rho_n \langle v \rangle} && (3.56) \\ &= \sqrt{\frac{m\pi}{k_B T}} \frac{1}{4\pi d^2 \rho_n}, && (3.57) \end{aligned}$$

where we have used the expression for the average velocity (equation (3.48)).

Chapter 4

Direct Simulation Monte Carlo

We now have the theoretical foundation we need to develop the first numerical model we will use to study flow in nanoporous media. It is called Direct Simulation Monte Carlo (DSMC), and is a stochastic particle model that has showed incredible predictive power for flow in the high Knudsen number regime. The model was developed by G. A. Bird in 1976 and was quickly picked up by engineers working in the field of aerospace. In the upper atmosphere (100 km), the mean free path of air is several meters. For space shuttles, this gives a Knudsen number of order unity since the size of its nose is of order meter [1]. In the later years, the method has been widely used to study microflows which is our main concern in this thesis. In 1992, the model was proved to converge towards a solution of the Boltzmann equation (equation (3.26)) in the limit where the timestep $\Delta t \to 0$ and the number of particles $M \to \infty$ [27].

We start the chapter by introducing the model and its basic philosophy. The model has two two crucial parts, collisions between particles which is discussed in section 4.3, and how the particles interact with the surface. The latter is covered in section 4.2. Another important subject is of course how we measure physical quantities like temperature and energy. This is described in section 4.4. In section 4.5 we have a longer discussion about the pressure and argue that a DSMC gas actually satisfies the ideal gas equation of state. We also derive a relationship between a given pressure difference ΔP and a constant force allowing us to induce flow in the system without needing large gradients in the density or temperature. We then have a brief comment about the numerical stability and how the timestep and collision cell size introduce errors in transport coefficients. We complete the chapter by discussing how we determine whether or not a system has reached a steady state in section 4.7. The implementation of all these steps are explained in detail in chapter 5.

4.1 The model

DSMC is a model that makes full use of statistical mechanics and the kinetic theory of gases. The physical system consists of N atoms of the same type in a box with volume $V = L_x L_y L_z$ (L_i being the length of the box in the i'th dimension) and porosity ϕ (the porosity was the fraction of the total volume available for fluids). The atoms all have mass m and an effective diameter d. Instead of tracking the trajectory every single atom, we simulate M particles, each representing N_{eff} real atoms. We can interpret this approximation as that all atoms in a small region of space around the coordinates of a simulated particle move with approximately the same velocity. The system is periodic in all directions which means that $(L_x, L_y, L_z) = (0, 0, 0)$, the corners of the cube are the same point. If a particle passes through a boundary surface (one of the sides of the box), it will just enter at the opposite side, see figure 4.1. The state of a DSMC simulation

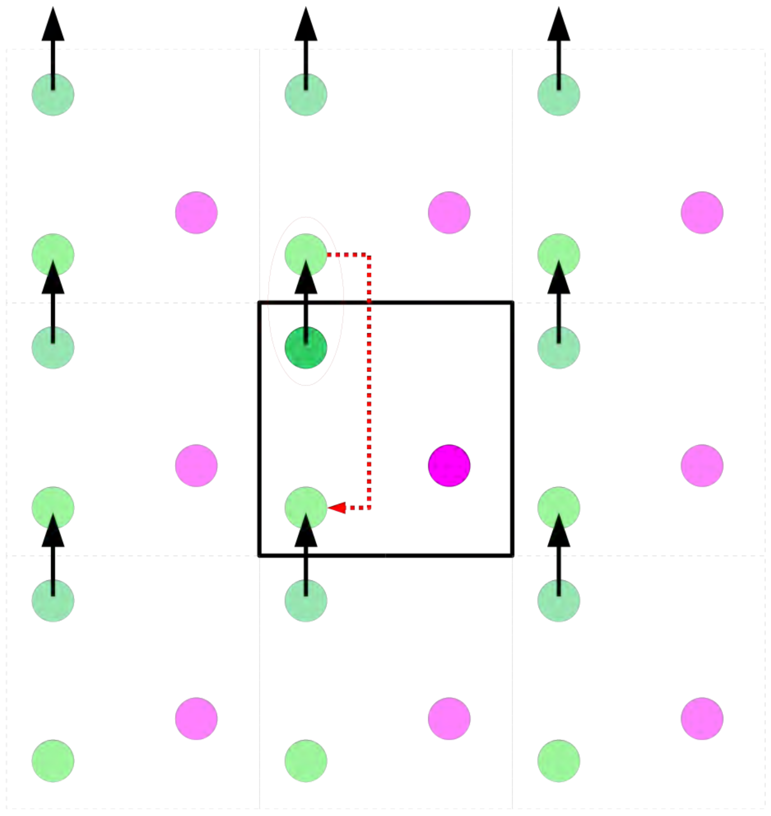

Figure 4.1: Periodic boundary conditions allows particles to fly out of a system by re-entering it in the opposite side. This reduces the amount of finite size system effects. Image from http://en.wikipedia.org/wiki/File:Limiteperiodicite.svg, accessed 16 March, 2014.

is fully described by the $6M$ phase variables, three velocities and three positions per simulated particle.

Since we don't have detailed information about the positions of all the real atoms, we cannot calculate the forces between the particles. Instead, we assume that the gas particles only undergo binary collisions (we neglect collisions between three or more particles), just like we did while deriving the Boltzmann equation in section 3.4. Particles are sorted into collision cells before the collisions are performed in a stochastic manner, where the rate of collisions and post-collision velocities are determined from kinetic theory. We can think that the collision step in the model

is an operator, a stochastic function $\mathcal{C}(\mathbf{r}, \mathbf{v}, \mathcal{G})$, where $\mathbf{r}$ and $\mathbf{v}$ form the phase space point and $\mathcal{G}$ contains all information about the system geometry. Do not worry, this will be clear in a minute.
The equations of motion are integrated by applying the standard Euler method on the positions so that $\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t)\Delta t$ for particle i. The timestep is chosen small enough so that we can split it into two parts; moving and colliding. This is a reasonable assumption as long as the timestep Δt is smaller than the mean collision time τ_{coll} (equation (3.56))

$$\Delta t \leq \tau_{\text{coll}} = \frac{1}{\sqrt{2}\pi d^2 \rho_n \langle v \rangle}, \tag{4.1}$$

since the velocity then does not change during the timestep. If a particle interacts with a boundary during the timestep, some sort of surface interaction rule is applied before the timestep is continued (we allow a particle to collide with the surface several times during a single timestep).

The moving step can also be seen as a stochastic operator, $\mathcal{M}(\mathbf{r}, \mathbf{v}, \mathcal{G})$ since the surface interaction often is a stochastic process. Different surface interaction models are discussed in section 4.2 with a detailed description of the implementation in section 5.1. Statistical properties are sampled at the end of each timestep where the physical quantities are sampled as time averages. A flow chart illustrating the steps of a typical DSMC algorithm is presented in figure 4.2.

4.2 Surface interactions

We remember from section 2.9 that the effects of surface interactions become significant as the pore sizes decrease. For very small pores, the number of atoms near the surface is comparable to total number of atoms. In an atomic model that calculates forces, these effects are already taken care of through the atomic forces, but in DSMC, we need a surface interaction model. In this section, we discuss three different models. The main property of these models is to perform a statistically correct transfer of energy and momentum between the surface and the colliding particles. There are two important parameters that incorporate the differences between gases and surfaces of various types; the normal and tangential accommodation coefficients.

4.2.1 Accommodation coefficients

When a particle with energy E_i hits a surface, some of the energy might be transferred to the wall resulting in an energy change ΔE. On average, we can define the *normal accommodation coefficient*

$$\sigma_n = \frac{E_i - E_r}{E_i - E_w}, \tag{4.2}$$

where E_i is the energy of the incoming particles, E_r is the energy of the outgoing particles and E_w is the energy corresponding to the surface temperature T_w. A thermal accommodation coefficient equal to zero would mean that there is no energy exchange, and we will get the specular wall model described below. $\sigma_n = 1$ on the other hand means that all the reflected particles have

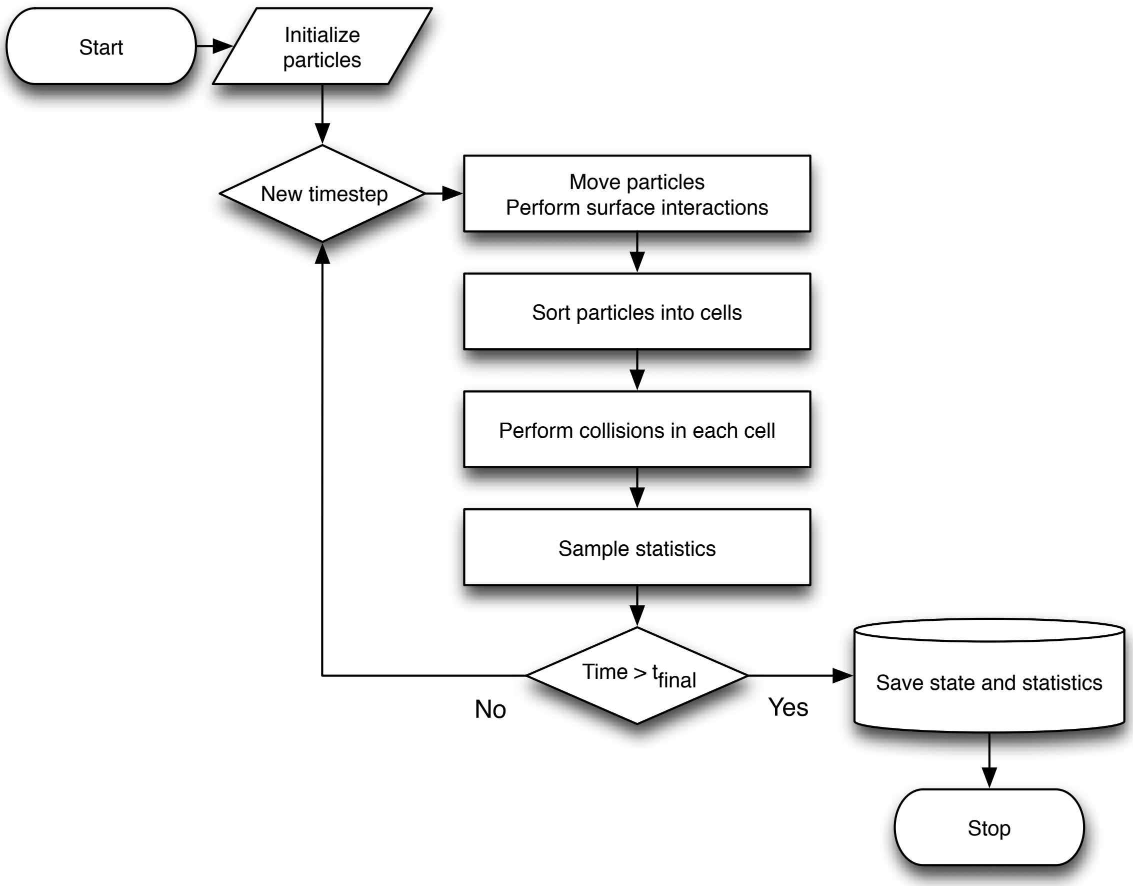
Start
Initialize particles
New timestep
Move particles
Perform surface interactions
Sort particles into cells
Perform collisions in each cell
Sample statistics
Time > t_final
No
Yes
Save state and statistics
Stop

Figure 4.2: Typical steps for a DSMC algorithm.

energies corresponding to the surface temperature. This is what we call the thermal wall (or diffuse reflection[15]), and there is no correlation between the incoming and outgoing velocities. More intricate models may use other values of the accommodation coefficients so the particles *remember* their incoming velocities. We can also define the *tangential momentum accommodation coefficient*

$$\sigma_t = \frac{\tau_i - \tau_r}{\tau_i - \tau_w}, \tag{4.3}$$

where τ_i and τ_r are the incoming and outgoing tangential momentum and τ_w is the momentum of the surface (e.g. a moving surface).

4.2.2 Specular wall

The specular wall behaves just like how a classical mirror reflects light. The colliding particles are reflected so that the normal component of the velocity is reversed while the tangential components remain unchanged. This leads to the famous saying; *the angle of incidence equals the angle of reflection.* Since the magnitude of all momentum components are unchanged, there is no exchange of energy with the wall.

4.2.3 Thermal wall

If we instead think of the wall as a reservoir with a given temperature T_w, we can imagine that the particles go into the wall, collide with the wall atoms for a while, and at some point, return with no correlation with the incoming velocity. In principle, the outgoing particles doesn't have to be the same as those that go into the wall, but as long as there is no net particle flux we might as well assume this for simplicity. Once a particle has hit the surface, we can choose a new, random velocity vector from a distribution so that the reflected particles have an average velocity according to the wall temperature. Since faster particles collide with the wall more often (this would also be true if we got *new* particles from the reservoir), this distribution has to reflect this fact. A distribution that satisfies this property is the *biased* Maxwell-Boltzmann distribution[1]

$$P_n(v_n)\mathrm{d}v_n = \frac{m}{k_B T_w} v_n e^{-\frac{m v_n^2}{2 k_B T_w}} \mathrm{d}v_n, \tag{4.4}$$

for the velocity component normal on the surface and

$$P_t(v_t)\mathrm{d}v_t = \sqrt{\frac{m}{2\pi k_B T_w}} e^{-\frac{m v_t^2}{2 k_B T_w}} \mathrm{d}v_t, \tag{4.5}$$

for the tangential component. Here m is the mass of the particle and k_B is Boltzmann's constant as usual. However, this distribution does not obey detailed balance since the incoming velocity is completely uncorrelated to the outgoing velocity, but it turns out that it provides great theoretical insight given its simple mathematical form. This, in addition to that it is computationally inexpensive (see section 5.2), it is much used in the literature.

4.2.4 The Cercignani-Lampis model

A more realistic model is the Cercignani-Lampis model which can be derived requiring detailed balance and wall isotropy (i.e. given incoming velocity $\mathbf{v}'$ and outgoing velocity $\mathbf{v}$, $P(\mathbf{v}' \to \mathbf{v})$ is unchanged if $\mathbf{v}'$ and $\mathbf{v}$ are rotated with the same angle about the surface normal vector) [8]. The probability of going from an incoming velocity $\mathbf{v}'$ to an outgoing velocity $\mathbf{v}$ is given as

$$\begin{aligned} P(\mathbf{v}' \to \mathbf{v}) = {} & \frac{2\sigma_n \sigma_t (2-\sigma_t)\beta_w^4}{\pi} \\ & \times \exp\Big(-\beta_w^2 \frac{v_n^2 + (1-\sigma_n)(v_n')^2}{\sigma_n} - \beta_w^2 \frac{(v_t - (1-\sigma_t)v_t')^2}{\sigma_t(2-\sigma_t)} \Big) \\ & \times I_0\Big(\beta_w^2 \frac{2\sqrt{1-\sigma_t} v_n v_n'}{\sigma_n} \Big), \end{aligned} \tag{4.6}$$

where v_n and v_t are the normal and tangential components of the velocities, I_0 is the zeroth-order modified Bessel function of the first kind and $\beta_w = (k_B T_w)^{-1}$. σ_n and σ_t are the accommodation coefficients discussed in subsection 4.2.1. We see that the tangential component is a normal distribution with a non-zero mean (the particles remember their incoming velocity), whereas the normal component is more complicated. The normal component distribution is plotted in figure 4.3.

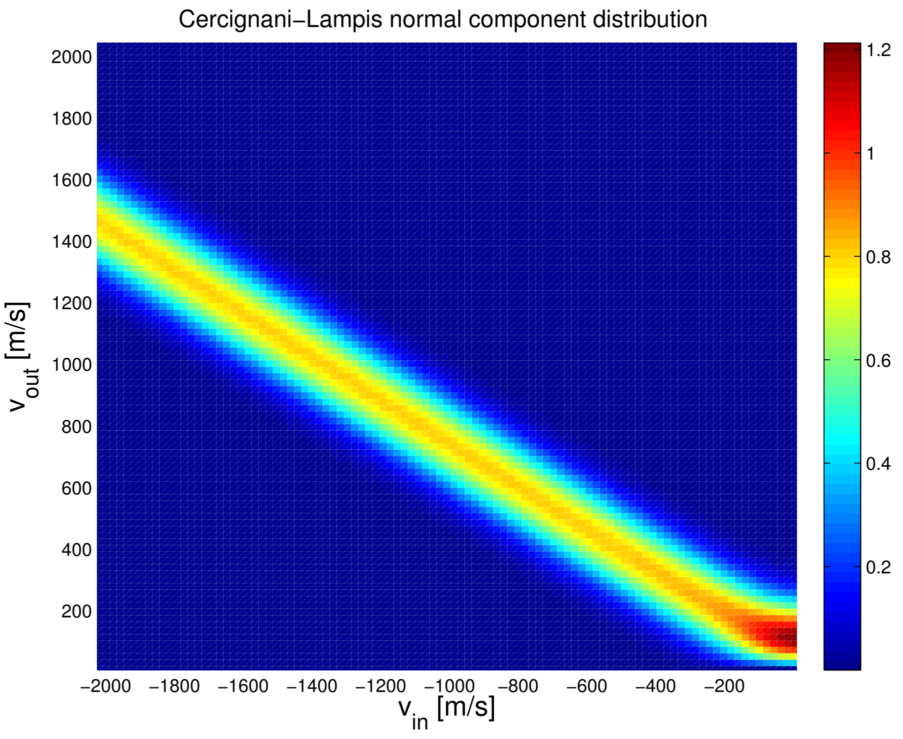

Figure 4.3: The Cercignani-Lampis normal component distribution for $T_w = 100\,\mathrm{K}$, $m = 39.948$ u (argon), $\alpha_n = 0.5$. We see that particles with high velocities are on average reflected with a slightly lower velocity, converging towards the velocity corresponding to the wall temperature T_w. The mean velocity for this temperature is $\langle v \rangle$ =144 m/s.

To draw random numbers from this distribution is orders of magnitudes slower than that of the thermal wall, since it isn't trivial (if even possible) to invert the cumulative distribution function. Instead we must use the von Neumann algorithm which is a accept-reject Monte Carlo algorithm[3]. Our main focus in this thesis is to validate the DSMC model by comparing it to theoretical results of which most have used the thermal wall. The Cercignani-Lampis model is available in the DSMC-code, but due to its computational cost and the low number of comparable theoretical results, we have used the thermal wall in all of our simulations.

4.3 Collisions

In a particle simulation with a continuous force field it is not clear how one would define a *collision event.* If two equal atoms with the same velocity move towards each other, the atoms would at some point reverse their velocities. In this case, one could define the collision event to occur at *the time of which their relative distance is at its minimum*, but other, equally valid, definitions probably exists. It is however clear that a collision should be identified as an event that happens when the atoms are close, i.e. short ranged forces.

As we already have mentioned, we don't operate with forces in the DSMC model. We calculate collision rates from the kinetic theory. In order to do so, we do need to choose an underlying collision model from which we will calculate the collision rates. We have chosen the *hard sphere* model, where each particle is assumed to be a perfect hard sphere with diameter d and mass m. Hard sphere means that two particles with radius R_1 and R_2 will undergo an *fully elastic* collision if their relative distance equals the sum of their radii, see figure 4.4. In DSMC, we will then apply what we could call a stochastic hard sphere collision model, where we use the hard sphere model only to calculate the statistical collision rates.

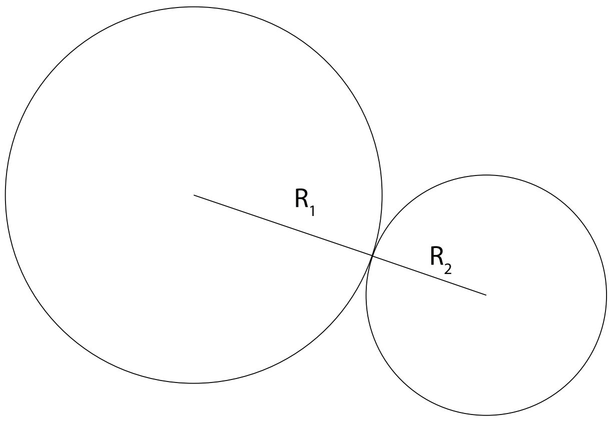

Figure 4.4: The hard sphere collision model. Two particles will collide if their relative distance becomes smaller than $R_1 + R_2$.

Since collisions should occur to nearby particles only, we sort the particles into spatial cells, allowing only particles from the same cell to collide. The dimension of these cells should not exceed the mean free path. If this was the case, two particles displaced by a distance larger than the mean free path could transfer momentum or energy faster than what would happen in a real gas (collisions usually transfer energy and momentum). Note that we allow particles *moving away* from each other to collide, since the simulated particles should not be interpreted as real molecules or atoms. In some sense, they are quasi-particles carrying statistical information only. They can be interpreted as density packets representing the distribution function f from section 3.1. Now that we have chosen a collision model, we should compute the collision rates.

4.3.1 Number of collisions

We will here use similar arguments as we did deriving the mean free path in section 3.7. If we choose two particles, i and j, with relative velocity $\mathbf{v}_r$, each representing N_{eff} real atoms with effection collision area $A = \pi d^2$ (see section 3.7), in a collision cell with volume V_c, the total volume sweeped out during a timestep is

$$V_{\text{sweep}} = N_{\text{eff}} \pi d^2 v_r \Delta t. \tag{4.7}$$

The probability that they will collide is the total sweeped volume V_{sweep} divided by the total cell volume V_c

$$P_{\text{coll}} = \frac{N_{\text{eff}} \pi d^2 v_r \Delta t}{V_c}. \tag{4.8}$$

If a collision cell has N_c particles, a particle has $(N_c - 1)$ potential collision partners. So the total number of collision pairs (we divide by two to prevent double counting of pairs) is $N_c(N_c - 1)/2$ which gives the number of collisions M_{coll}

$$M_{\text{coll}} = \frac{N_c(N_c - 1)}{2} P_{\text{coll}} = \frac{N_c(N_c - 1) N_{\text{eff}} \pi d^2 \langle v_r \rangle \Delta t}{2V_c}, \tag{4.9}$$

where we replaced the relative velocity v_r by the mean value in the cell

$$\langle v_r \rangle = \frac{1}{N_c} \sum_{i>j} |\mathbf{v}_i - \mathbf{v}_j|. \tag{4.10}$$

But computing the mean relative velocity $\langle v \rangle$ in each cell *every timestep* sounds like a horrendous thing to do. It requires to sum over all pairs which is $O(N^2)$, which is exactly what we try to avoid in the first place. But we can do a little trick. Instead, we calculate M_{cand} *candidate pairs* so that

$$\frac{M_{\text{coll}}}{M_{\text{cand}}} = \frac{\langle v_r \rangle}{v_r^{\max}}, \tag{4.11}$$

since the probability of collision is proportional to the relative velocity. Each of these candidates go through an acceptance-rejection process where we pick a uniform random number $\mathcal{R}_1 \in (0, 1)$ and accept the collision if

$$v_r \leq v_r^{\max} \mathcal{R}_1. \tag{4.12}$$

This will only accept $\langle v_r \rangle / v_r^{\max}$ of the candidates and we end up with M_{coll} actual collisions, as desired. The number of candidate pairs is then computed as

$$M_{\text{cand}} = \frac{N_c(N_c - 1) N_{\text{eff}} \pi d^2 v_r^{\max} \Delta t}{2V_c}. \tag{4.13}$$

If we choose $v_r^{\max}$ very high, we will still perform the correct amount of collisions, but the number of rejected collisions would be high and hence the algorithm is inefficient. If a collision pair has a higher relative velocity, we simply update this variable (in that cell).

We should one more time mention that particles moving away from each other can collide. This property has, as we will see in section 4.5.1, an interesting consequence leading to the ideal gas equation of state. One more thing, we haven't figured out how to perform the actual collisions. Until now, we know only how to select collision pairs, so let's calculate the post-collision velocities.

4.3.2 Post-collision velocities

After a collision is accepted, we want to choose new velocities conserving both energy and momentum. We need a total of six equations to determine the post-collision velocities, where four are provided through the conservation laws. Conservation of momentum reveals that the center of mass velocity is unchanged

$$\mathbf{v}_{\text{cm}} = \frac{1}{2}(\mathbf{v}_i + \mathbf{v}_j) = \frac{1}{2}(\mathbf{v}_i^* + \mathbf{v}_j^*) = \mathbf{v}_{\text{cm}}^*, \tag{4.14}$$

where the energy conservation tells us that the relative velocity vector does not change its magnitude

$$v_r = |\mathbf{v}_i - \mathbf{v}_j| = |\mathbf{v}_i^* - \mathbf{v}_j^*| = v_r^*. \tag{4.15}$$

Here we used that the velocities of the particles are uncorrelated so that $\mathbf{v}_i \cdot \mathbf{v}_j = 0$ on average. The two remaining degrees of freedom are determined by choosing the direction of the relative velocity

$$\mathbf{v}_r^* = v_r \left[(\sin\theta\cos\phi)\mathbf{i} + (\sin\theta\sin\phi)\mathbf{j} + (\cos\theta)\mathbf{k}\right], \tag{4.16}$$

where the angles are uniformly distributed over the unit sphere so that all directions for the relative velocity are equally probable. The area element $\mathrm{d}\Omega$ can be written as

$$\mathrm{d}\Omega = \sin\theta\,\mathrm{d}\theta\,\mathrm{d}\phi = -\mathrm{d}\phi\,\mathrm{d}(\cos\theta), \tag{4.17}$$

so we need to choose ϕ and $\cos\theta$ uniformly. This is easy, we simply choose

$$\phi = 2\pi\mathcal{R}_2 \qquad\qquad \cos\theta = 2\mathcal{R}_3 - 1,$$

where $\mathcal{R}_2$ and $\mathcal{R}_3$ are random numbers in the range $(0, 1)$ and calculate $\sin\theta = \sqrt{1 - \cos^2\theta}$. The post-collisions velocities are then found by

$$\mathbf{v}_i^* = \mathbf{v}_{\text{cm}} + \frac{1}{2}\mathbf{v}_r^* \tag{4.18}$$

$$\mathbf{v}_j^* = \mathbf{v}_{\text{cm}} - \frac{1}{2}\mathbf{v}_r^*. \tag{4.19}$$

4.4 Measuring physical quantities

The model, or shall we say, the *simulator* is in principle fully described. The particles move and perform collisions with the surface according to some interaction rule. Then the particles will collide with each other. This process goes on an on until we are satisfied or out of computing time. Within this framework, statistical mechanics happens and particles behave as they should (the model solves the Boltzmann equation). But there is no reason to have a simulator if we're not going to use it to learn physics.

The simulator will take the system from some initial state and guide it through the phase space. This was what the ergodicity hypothesis allows us to do (see section 3.3). We can evolve the system through time and visit the phase space with probabilities equal to those of the ensemble. The DSMC model is inherently stochastic, so any physical quantity should be computed by averaging many instantaneous measurements. We should assume that there will occur gradients of the physical quantities (for example gradients in density and temperature), so we should calculate local values in the collision cells. In a typical collision cell, there will be maybe ten to a hundred particles, so the instantaneous values will fluctuate significantly. But as we know from statistics, if the system is in equilibrium, the fluctuations (here the standard deviation) in e.g. the energy or temperature will decrease as $1/\sqrt{N_m}$ if measured N_m times assuming that the measurements are uncorrelated. The latter requirement can be obtained by measuring every nth timestep. We can measure the correlation between two states through the velocity autocorrelation function given as

$$C_v(t) = \frac{\langle \mathbf{v}(t)\mathbf{v}(0)\rangle_N}{\langle \mathbf{v}(0)\mathbf{v}(0)\rangle_N} \tag{4.20}$$

$$= \frac{1}{N}\frac{\sum_{n=1}^{N} \mathbf{v}_n(t)\cdot\mathbf{v}_n(0)}{\sum_{n=1}^{N} \mathbf{v}_n(0)\cdot\mathbf{v}_n(0)}, \tag{4.21}$$

which is equal to one at $t = 0$, and decays with time as the system becomes more uncorrelated with the initial state. We should then measure physical quantities with a time interval corresponding to the time where the velocity autocorrelation function has become more or less zero. We will now quickly discuss how to measure the physical quantities we will use in our analysis later on.

4.4.1 Energy

The total energy of a system is as usual given by the sum of the kinetic and potential energy. Since we are using the hard sphere model, the potential energy is given as

$$V(\mathbf{r}_1,\mathbf{r}_2) = \begin{cases} 0 & \text{if } |\mathbf{r}_1 - \mathbf{r}_2| > d \\ \infty & \text{if } |\mathbf{r}_1 - \mathbf{r}_2| \le d, \end{cases} \tag{4.22}$$

where collisions will make sure that the relative distance between any particle pair always remains larger than the diameter. The total energy of our entire system will then only be the kinetic energy

$$E = E_k = \sum_{n=1}^{N} \frac{1}{2} m_n v_n^2 \tag{4.23}$$

where m_n is the mass of particle n and v_n is its scalar velocity. An example implementation of how the instantaneous kinetic energy is calculated is given in listing 4.1. Remember that in DSMC, each particle represents a given number of real atoms.

```
double calculate_kinetic_energy(vector<Vector3> &velocities) {
  double kinetic_energy = 0;
  for(int n=0; n<velocities.size(); n++) {
    Vector3 velocity = velocities.at(n);
    kinetic_energy += 0.5*mass*atoms_per_particle*velocity.NormSquared();
  }

  return kinetic_energy;
}
```

Listing 4.1: Calculation of kinetic energy.

Once we have found the kinetic energy, we can easily compute the temperature.

4.4.2 Temperature

The temperature is defined through the equipartition theorem using the three momentum degrees of freedom

$$\langle E_k \rangle = \frac{3}{2} N k_B T, \tag{4.24}$$

where $\langle E_k \rangle$ is the average kinetic energy, N is the number of particles, k_B is Boltzmann's constant and T is the temperature. The only unknown quantity in this equation is the temperature

$$T = \frac{2E_k}{3Nk_B}, \tag{4.25}$$

where we have dropped the average value brackets of the kinetic energy because we use this to define the instantaneous temperature. Note that if the fluid is flowing (the gas has non-zero average velocity), the numerical values of the particle's velocities are higher, which in turn results in higher measured temperatures. But of course, this has to be wrong, the temperature should not depend on the choice of frame of reference. Imagine a bacteria swimming in the flowing fluid, the temperature it feels is proportional to the average kinetic energy compared to the local frame of reference. This indicates that we should define a instantaneous local temperature $T(\mathbf{r}, t)$ which we define as

$$T(\mathbf{r}, t) = \frac{2m}{3k_B} \left[\frac{E(\mathbf{r}, t)}{\rho(\mathbf{r}, t)} - \frac{1}{2} \left(\frac{\mathbf{p}(\mathbf{r}, t)}{\rho(\mathbf{r}, t)} \right)^2 \right], \tag{4.26}$$

where $E(\mathbf{r}, t)$, $\rho(\mathbf{r}, t)$ and $\mathbf{p}(\mathbf{r}, t)$ is the average kinetic energy, density and momentum within some volume around the point $\mathbf{r}$. This is of course still just the equipartition theorem where we measure the kinetic energy in the frame of reference determined by the fluid around the point $\mathbf{r}$. We usually use the collision cells to compute these local values. Now we need to calculate the density.

4.4.3 Density

Here we should comment on another detail, a consequence of our intermolecular collision model. Since there are no forces between the particles, all of them can in principle be at the very same point (remember that we used the hard sphere collision model only to calculate the collision rates, not to detect collisions). This will of course not happen, but it is possible to initiate a state in that configuration. The number density ρ_n in any volume V is easily calculated through

$$\rho_n = \frac{N}{V}, \tag{4.27}$$

where N is the number of atoms in that volume. This enables us to calculate local densities as well as the global density of the system. Again we must not forget that each simulated particle represents N_{eff} real atoms.

4.4.4 Permeability

The permeability k is defined through Darcy's law (equation (2.7)) which we discussed in section 2.4

$$k = \frac{Q\mu L}{A\Delta P}, \tag{4.28}$$

where L is the length of the system in the flow direction, μ is the viscosity, Q is the volumetric flow rate, A is the cross sectional area, $\Delta P = P_0 - P_L$ are the pressures at $x = 0$ and $x = L$. The viscosity can be computed with the kinetic theory [2]

$$\mu = \frac{5}{16d^2}\sqrt{\frac{mk_BT}{\pi}}. \tag{4.29}$$

Measuring the permeability then introduces to problems we need to figure out how to solve. The first is how we measure the volumetric flow rate. As the name indicates, it is a measure of how many units of volume passes through a surface per unit time. In DSMC, we will measure this by counting how many particles that undergo a periodic boundary condition shift in the flow direction, this is the number flow rate. Assuming we have N particles, each representing N_{eff} real atoms in a system with volume V, the volume per particle v is given as

$$v = \frac{V}{NN_{\text{eff}}} = \rho^{-1}. \tag{4.30}$$

The volumetric flow rate Q is then simply the number flow rate multiplied by the volume per particle. The next problem is that the systems we will study are periodic in the flow direction. This implies that the point $x = 0$ actually is the *same point* as $x = L$, which gives $P(x = 0) = P(x = L)$. Hence, the pressure difference is zero, no matter how we measure the pressure. In the next section, we will have a rather comprehensive discussion about pressure and find that a constant acceleration g can be related to a pressure difference ΔP as

$$g = \frac{\Delta P}{m\rho_n\Delta x}, \tag{4.31}$$

where m is the mass of an atom, ρ_n is the number density and Δx is the distance between the two points of the pressure difference, usually the system length L. The permeability is then found as

$$k = \frac{Q\mu}{Agm\rho_n}, \tag{4.32}$$

which is how we will measure the permeability.

4.5 Pressure

Pressure plays an important role in the field of fluid flow. An applied pressure difference (which gives a net force) is usually what induces the flow, in addition to being an important property of the fluid. The discussion about pressure contains both how we *define* the pressure, and we *measure* it in a DSMC simulation. And of course how we induce flow in the simulation. It turns out that DSMC satisfies the ideal gas equation of state, so given constant temperature, the local pressure is proportional to the local density. A pressure difference, or pressure gradient, would then require a similar gradient in the density. This is difficult to obtain in a system with periodic boundary conditions because $\rho(x = 0) = \rho(x = L)$ for a system of length L. Instead we will derive a relation between the pressure gradient and a corresponding constant force which we will use to induce flow in the simulations.

4.5.1 Equation of state

The free *modules* in a DSMC program are the collision operator $\mathcal{C}$ and the move operator $\mathcal{M}$ which fully (stochastically) determine the time evolution of the system. For systems with pairwise interactions (such as hard spheres or the Lennard-Jones potential which we meet in section 7.1), the pressure may be defined as (see appendix C for a derivation)

$$P = \rho_n k_B T + \frac{1}{3V}\left\langle \sum_{i<j} \mathbf{F}(\mathbf{r}_{ij}) \cdot \mathbf{r}_{ij} \right\rangle, \tag{4.33}$$

where the first term is the ideal gas pressure whereas the second term is called the virial of the pressure. Here $\mathbf{F}(\mathbf{r}_{ij})$ is the force between particle i and j, and $\mathbf{r}_{ij}$ is their relative distance. In DSMC we don't have the details about the forces, but we can formulate a similar expression using that the force is the change in momentum per time

$$P = \rho_n k_B T + \frac{1}{3Vt} \sum_{\text{all collisions}} m\Delta\mathbf{v}_{ij} \cdot \mathbf{r}_{ij}, \tag{4.34}$$

where $\Delta\mathbf{v}_{ij}$ is the change of velocity of one of the particles during a collision[13]. In the collision model we have used, there are no correlation between the change in velocity $\Delta\mathbf{v}_{ij}$ and the displacement vector $\mathbf{r}_{ij}$ between the particles

$$\langle \Delta\mathbf{v}_{ij} \cdot \mathbf{r}_{ij} \rangle = 0, \tag{4.35}$$

so the expression for the pressure is reduced to that of an ideal gas

$$P = \rho_n k_B T. \tag{4.36}$$

Since the main focus of this thesis is to study dilute gases where the ideal gas is a good approximation, this collision model is sufficient. For dense gases, or liquids, it is possible to apply collision models that yields other equations of state [13].

4.5.2 Measuring pressure

Since the gas satisfies the ideal gas equation of state, this is of course how we measure the pressure

$$P = \rho_n k_B T, \tag{4.37}$$

since we already know how to calculate the density and the temperature. The local pressure is of course found by using the local values of the density and temperature.

4.5.3 Applying a pressure gradient

In order to induce flow in a system, it is common to apply a pressure gradient. A pressure gradient means that there acts a nonzero net force on any volume element dV in the system. In continuum models like the NSE (see section 2.2), the pressure (and hence the pressure gradients) is incorporated as boundary conditions where pressure is specified at given points. A typical boundary condition is $P(x = 0) = P_0$ and $P(x = L) = P_L$, but as we already mentioned, periodic boundary conditions is a problem since the points are the very same point. Instead we will use ideas from continuum mechanics to relate a given pressure gradient to a constant force which we will apply on all particles in the system. In the literature, this is often called gravity driven flow.

We look at a volume element of size $\Delta V = \Delta x \Delta y \Delta z$ in a channel with a continuous fluid and a pressure gradient in the x-direction, see figure 4.5. The net force acting on the volume element in the $x-$direction is

$$F = P_2 \Delta y \Delta z - P_1 \Delta y \Delta z = \Delta y \Delta z \Delta P, \tag{4.38}$$

where $\Delta P = P_2 - P_1$. We aim to find a constant force $F = mg$ being equivalent to that of the pressure difference. Given an acceleration g, the force is then

$$F = mg = \rho_m \Delta V g. \tag{4.39}$$

We aim to find a force equal to the one from the pressure difference

$$F = \Delta y \Delta z \Delta P = \Delta V \frac{\Delta P}{\Delta x}, \tag{4.40}$$

which gives the relation

$$g = \frac{\Delta P}{\rho_m \Delta x}. \tag{4.41}$$

In simple geometries like a tube, the behavior of the flow for both pressure models should be similar. However, for disordered systems with regions that can *trap* particles, we can expect some effects that will affect the fluid flow in a non-physical way. An example is shown in figure 4.6 where a gas driven by a constant acceleration in the x-direction will be slowed down in the area marked gray. In a gas driven by a real pressure difference, we expect a net force along the channel in all regions. We can now obtain a desired pressure difference through equation (4.41) and apply that acceleration to all particles each timestep.

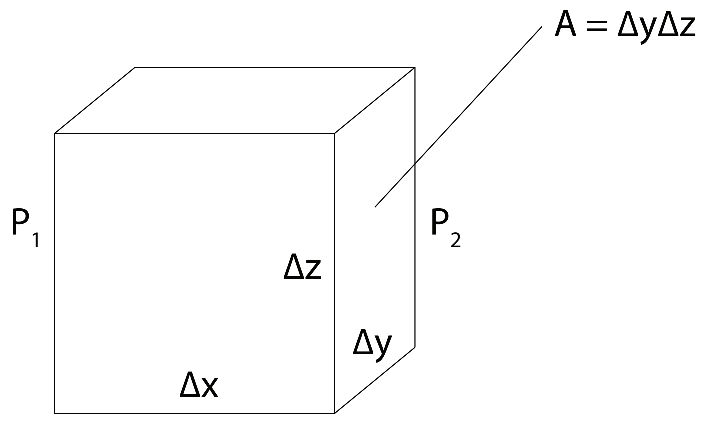

Figure 4.5: The net force acting on the volume element $\Delta V = \Delta x \Delta y \Delta z$ in the $x-$direction is given by the pressure difference times area $A(P_2 - P_1) = \Delta y \Delta z \Delta P$.

4.6 Numerical stability and discretization error

Most numerical methods have a critical stability criterion where the energy or some other property might diverge if the timestep is too large. For example, while solving PDE's with a finite difference scheme, we often encounter the Courant number which is a critical threshold of the ratio of the discretization length of space and time. For the one-dimensional wave equation, this can be expressed as

$$C = \frac{|\dot{x}|_{max} \Delta x}{\Delta t} \leq C_{max}, \tag{4.42}$$

where $|\dot{x}|_{max}$ is the magnitude of the velocity. If the spatial grid has high resolution, small Δx, we need a similarly small timestep Δt.

However, since the DSMC model always conserves energy and momentum (during particle collisions), the method is in principle numerically stable for any timestep. As mentioned in section 4.1, the timestep is split into two parts; moving and colliding. The timestep should therefore be smaller than the mean collision time. Larger timesteps may result in large errors in the transport coefficients (such as viscosity and thermal conductivity)[15]. While the timestep is compared to the mean collision time τ_{coll}, the collision cell size can be seen as the spatial discretization, and be compared to the mean free path λ.

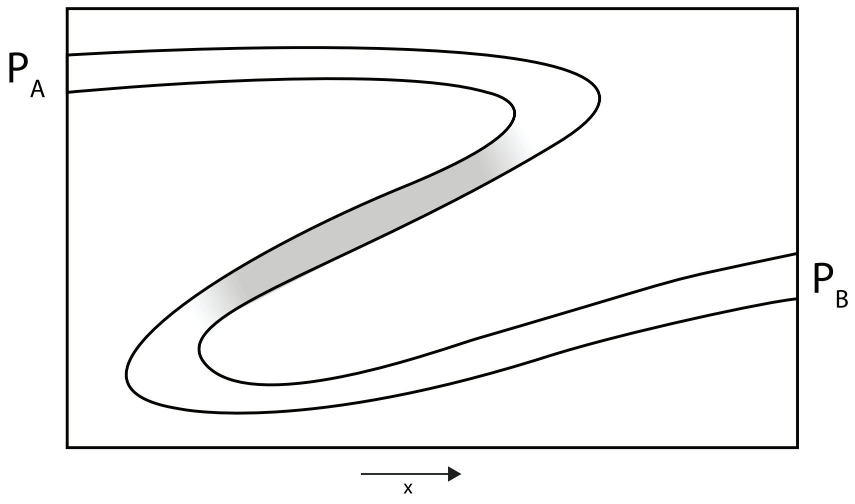

Figure 4.6: Flow induced by a constant acceleration will not reproduce correct flow behavior when the gas in a larger part (marked gray) of the channel flows in the opposite direction of the force. In a *real* pressure-driven flow, the net force on the gas will point along the expected flow direction, also in the gray marked area, whereas the acceleration-driven flow will be slowed down.

4.6.1 Finite cell size

The collision cells allows all particles within a cell to collide with each other. So if the cell size is very large, particles from a hot region (in one corner of the collision cell) may collide with particles in a colder region (maybe in another corner) that are displaced by a large distance. This could enable heat to transfer much faster than it would in a real gas. The cell size L_{cell} should therefore at least be smaller than the mean free path[15]. The viscosity can be calculated from kinetic theory

$$\mu = \frac{5}{16d^2}\sqrt{\frac{mk_BT}{\pi}}, \tag{4.43}$$

which Garcia et al. [2] used to show that the error in the viscosity has a quadratic dependency of the cell size

$$\mu(L_{\text{cell}}) = \frac{5}{16d^2}\sqrt{\frac{mk_BT}{\pi}}\left[1 + \frac{16}{45\pi}\frac{L_{\text{cell}}^2}{\lambda^2}\right]. \tag{4.44}$$

If the length of the collision cells equals the mean free path, we could then expect a 10% error in the viscosity coefficient.

4.6.2 Finite timestep

A large timestep may allow particles to travel through several collision cells during a single timestep. This would allow information to travel faster than in a real gas and also leads to errors in transport coefficients like the viscosity. Hadjiconstantinou [14] derived an expression for the timestep dependency for the viscosity, similar to equation (4.44)

$$\mu = \frac{5}{16d^2}\sqrt{\frac{mk_BT}{\pi}}\left[1 + \frac{16}{75\pi}\frac{(v_m\Delta t)^2}{\lambda^2}\right], \tag{4.45}$$

where $v_m = \sqrt{2k_B/mT}$ is the most probable velocity. We see that the error is proportional to $(v_m\Delta T/\lambda)^2$ which vanishes in the limit $\Delta t \to 0$.

4.7 Reaching a steady state

Since we want to study flow in nanoporous media, before inducing the flow, the fluid is on average obviously at rest. Immediately after we have started applying the constant force that will make the fluid flow, the fluid velocity is still approximately zero. After a certain amount of time, the system will reach a steady state which in its most simple form can be defined as when the time derivative of the *fluid velocity* in any region, the local velocity, is zero. We should not start to sample flow statistics like the permeability until such a state has been reached. However, the system may not be in a steady state even though the average local fluid velocity does not change over time. There are other physical quantities like that may still be changing.

A naive, but simple approach to measure whether or not the fluid velocity has converged is to look at the measured temperature defined in equation (4.25). If the gas temperature starts out at T =300 K before the flow is induced, the measured temperature will increase while the fluid velocity increases. Once the fluid has reached a steady state, the temperature will have converged to some value it will continue fluctuating around. For simplicity, this is how we have determined whether or not the system has reached the steady state. In future development of the code, better methods should be implemented.

Chapter 5

Implementation

In this chapter we go into detail about how the DSMC model is implemented in C++. We assume that the reader is familiar with the programming language. Instead of going through all the classes and their relations, we follow the time line of a simulation, going from the initialization of the system to how the timestep is computed. For convenience, an UML diagram is shown in figure 5.1.

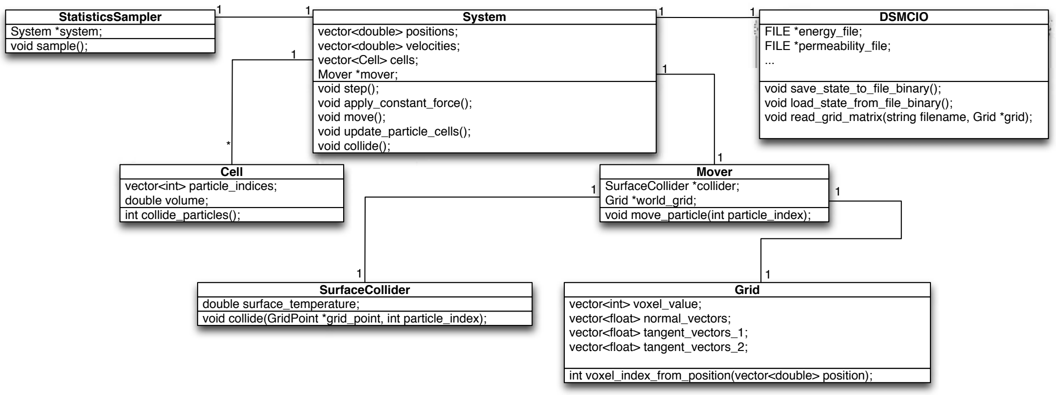
StatisticsSampler
System *system;
void sample();
System
vector<double> positions;
vector<double> velocities;
vector<Cell> cells;
Mover *mover;
void step();
void apply_constant_force();
void move();
void update_particle_cells();
void collide();
DSMCIO
FILE *energy_file;
FILE *permeability_file;
...
void save_state_to_file_binary();
void load_state_from_file_binary();
void read_grid_matrix(string filename, Grid *grid);
Cell
vector<int> particle_indices;
double volume;
int collide_particles();
Mover
SurfaceCollider *collider;
Grid *world_grid;
void move_particle(int particle_index);
SurfaceCollider
double surface_temperature;
void collide(GridPoint *grid_point, int particle_index);
Grid
vector<int> voxel_value;
vector<float> normal_vectors;
vector<float> tangent_vectors_1;
vector<float> tangent_vectors_2;
int voxel_index_from_position(vector<double> position);
1
1
1
*
1
1
1
1
1
1
1
1
1
1

Figure 5.1: A UML-diagram showing how the classes in the DSMC program are related to each other.

The first step of the full program is to create the geometry we want the particles to be confined by. We explain how that is done in section 5.1. Then, we can initialize a system by adding particles randomly inside the geometry. Their velocities should be Maxwell-Boltzmann distributed, as described in section 3.46. The initialization process is explained in section 5.2 before we are ready to perform the timesteps. A timestep is divided into four stages. These stages are

- accelerate particles to induce flow,
- move particles and perform surface interactions,

- update collision cells,
- perform collisions between particles,

which all are discussed in section 5.3. This is everything we need to run a DSMC program in *any* geometry we want. The only thing left to explain is how we have parallelized the code, allowing it to run on many processors on a supercomputer. This final piece is explained in section 5.4. The full code is available, on demand, by contacting the author at anderhaf@fys.uio.no.

5.1 Complex geometries

All the surface interaction models from section 4.2 use the surface normal and tangent vectors to calculate the reflected velocities. These vectors are easy to determine if the system consists of two parallel plates in the xy-plane, or any other mathematically well described geometry. Such systems are interesting as validation test cases, but most real world materials have a more complex geometry without any simple mathematical description. A very much used representation of such geometries is a triangle mesh in which the surface consists of many connected triangles. The triangles have a well defined normal vector and tangent plane which is easy to calculate. With this method, collision detection is done by checking intersection with each triangle and is rather computationally expensive. In this thesis, we have chosen another approach by representing the system as a large, binary three-dimensional matrix consisting of voxels, each having the value *filled* or *empty*. With this model, collision detection is done by a quick memory lookup to check if the voxel corresponding to the position of a particle is filled or not. In this section we discuss how to create such a matrix, how to identify the surface voxels and how we calculate the normal and tangent vectors.

5.1.1 Binary representation

Any system geometry is fully described by a three dimensional matrix with dimensions $m \times n \times l$. Each matrix element represents a voxel in the physical space, and can take values 0 or 1. A value of one means that the voxel is filled, whereas zero means the voxel is empty. Since the particles only will collide directly with the voxels defining the surface, these are given the value 2. The idea is best explained with an example.

5.1.2 Example - a cylinder

We will now show how to create a cylinder with radius r with this model. The idea is very simple, we just loop through every voxel in the system and check whether or not the voxel should be marked as filled or empty. We define that no matter how many voxels we have, the radius r is given in the range $[0, 0.5]$ so that the system is a $1 \times 1 \times 1$ cube. The algorithm should be easy to understand from the code example in listing 5.1.

```
void create_cylinder(double radius) {
  float voxel_size_x = 1.0 / num_voxels_x;
  float voxel_size_y = 1.0 / num_voxels_y;

  double cylinder_center_x = voxel_size_x * (num_voxels_x / 2.0);
  double cylinder_center_y = voxel_size_y * (num_voxels_y / 2.0);

  for(int i=0; i<num_voxels_x; i++) {
      for(int j=0; j<num_voxels_y; j++) {
          for(int k=0; k<num_voxels_z; k++) {
              double x = i/double(num_voxels_x) + voxel_size_x/2.0;
              double y = j/double(num_voxels_y) + voxel_size_y/2.0;
              bool is_solid = true;

              double dx = (x - cylinder_center_x);
              double dy = (y - cylinder_center_y);
              double dr2 = dx*dx + dy*dy;

              if(dr2 < radius*radius) {
                  is_wall = false;
              }

              int index = i*ny*nz + j*nz + k;

              if(is_wall) vertices[index] = 1;
              else vertices[index] = 0;
          }
      }
  }

  calculate_normals_tangents_and_inner_points();
}
```

Listing 5.1: An example showing how to create a cylinder.

Notice the last line there, the function *calculate_normals_tangents_and_inner_points()*. After all voxels are marked as filled or empty, we need to identify the surface voxels and calculate their tangent and normal vectors.

5.1.3 Identifying the surface voxels

A voxel that is filled, but not part of the surface, will be completely surrounded by filled voxels. We *define* the surface voxels as *the filled voxels that have less than 26 neighboring filled voxels*. The algorithm can be implemented like in listing 5.2 (we also need to take care of the periodic boundary conditions, but the idea is best illustrated without that complication).

```
bool is_surface(int voxel_index_i, int voxel_index_j, int voxel_index_k) {
  for(int i=-1;i<=1;i++) {
      for(int j=-1;j<=1;j++) {
      for(int k=-1;k<=1;k++) {
            // Skip self
            if(i == j == k == 0) continue;

                    if(world_matrix[voxel_index_i + i][voxel_index_j + j][
                        voxel_index_k + k] == 0) {
                      // This neighbour is empty, hence a surface voxel
                      return true;
                    }
                }
            }
        }

        return false;
    }
```

Listing 5.2: An example showing how to identify the surface voxels. The world_matrix contains all the voxel values (zeros and ones).

This has to be done for every voxel in the system, but only once per geometry. When we have identified all surface voxels, we need to calculate the normal and tangent vectors for each of them. Once this is done, we can save and use the geometry in a simulation.

5.1.4 Calculating normal and tangent vectors

Each surface voxel is a cube with six faces, each having a normal vector pointing out from the cube. Each face then defines the tangent plane orthogonal to this normal vector. When a particle collides with a surface voxel, we can calculate which face the particle passed through and use those vectors to calculate the reflected velocity. However, in this thesis, we have developed a new way of describing the surface vectors. We will use the neighboring voxels so that the normal vector of a surface voxel contains information of the curvature of the surface in order to have more realistic collisions.

The idea is simpler to illustrate for a two dimensional square, but the concept can easily be generalized to three dimensions. A square consisting of 9 pixels has a geometric center $\mathbf{r}_{\mathrm{gc}}$, plus a center of mass $\mathbf{r}_{\mathrm{cm}}$ which can be defined by the values, the mass, of the voxels

$$\mathbf{r}_{\mathrm{cm}} = \sum_{i,j} \mathbf{r}_{ij} m_{ij}, \tag{5.1}$$

where $m_{ij} \in \{1, 0\}$ and $\mathbf{r}_{ij}$ is the coordinate of the voxel with $\mathbf{r}_{\mathrm{gc}}$ as the origin. We *define* the normal vector to be the normalized local center of mass vector

$$\mathbf{n} = \frac{\mathbf{r}_{\mathrm{cm}}}{|\mathbf{r}_{\mathrm{cm}}|}. \tag{5.2}$$

In figure 5.2, we have computed the normal vector for four different voxel configurations where we see that the direction of the normal vectors seems reasonable. In the three dimensional case, we find the normal vectors in exactly the same way, but by using the 26 neighbors. The only thing we need to calculate are the tangent vectors. These can be found by first choosing

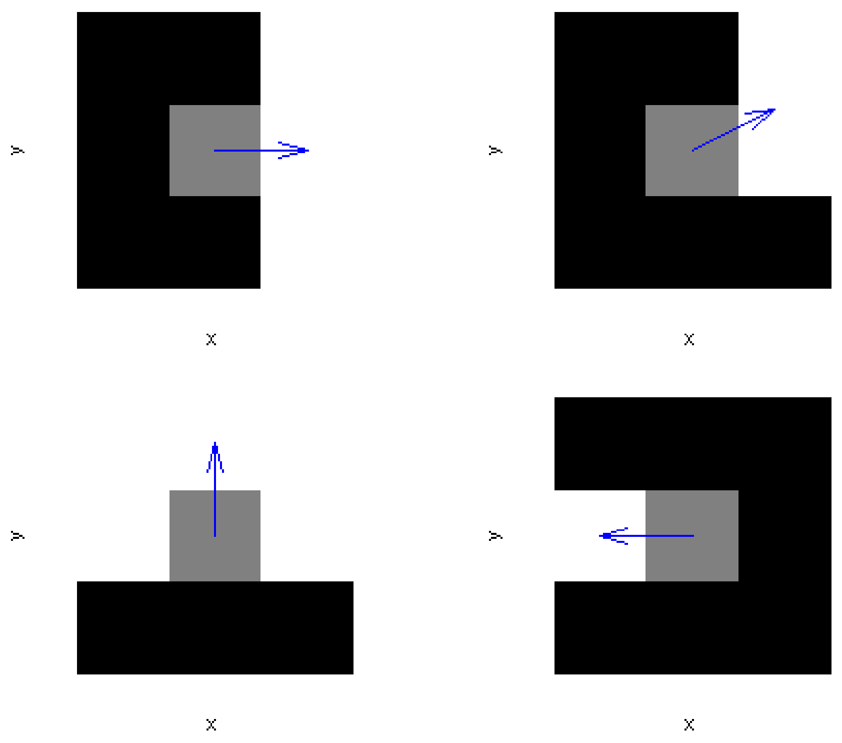

Figure 5.2: Four different pixels configurations in a 3×3 grid. The filled pixels are marked black whereas the empty pixels are white. We compute the normal vector of the center surface pixels (marked gray) based in its neighbors following equation (5.2). We see that this algorithm generates normal vectors that behave as expected.

a random vector $\mathbf{v}$ and apply the Gram-Schmidt process making it orthogonal on the normal vector so that

$$\tilde{\mathbf{t}}_1 = \mathbf{v} - (\mathbf{n} \cdot \mathbf{v})\mathbf{n}, \tag{5.3}$$

which gives the normalized tangent vectors

$$\mathbf{t}_1 = \frac{\tilde{\mathbf{t}}_1}{|\tilde{\mathbf{t}}_1|} \tag{5.4}$$

$$\mathbf{t}_2 = \mathbf{n} \times \mathbf{t}_1. \tag{5.5}$$

The porosity ϕ is found by counting the number of empty voxels divided by the total number of voxels

$$\phi = \frac{N_{\text{empty}}}{N_{\text{voxels}}}. \tag{5.6}$$

5.2 System initialization

Now that we know how the geometry is represented, we are ready to discuss the DSMC simulator itself. When we want to run a simulation in a given geometry, we need to decide a few things. We need to specify

- the density (ρ_n),
- the temperature (T_0),
- the physical system size (L_x, L_y, L_z), and
- the number of atoms each simulated particle represents (N_{eff}).

These are needed input parameters that will affect the initialization process. The first step when the program starts is to load the geometry from disk. We then create the collision cells, before we add all the particles to the system.

5.2.1 The geometry

We remember that everything we need to know about the geometry are the voxel matrix of size $N_x N_y N_z$, a normal vector and two tangent vectors per voxel. This data is saved in a class `Grid` where the geometry data is saved in four variables and lookup functions as shown in listing

```
typedef enum {
    voxel_type_empty = 0,
    voxel_type_wall = 1,
    voxel_type_boundary = 2
} voxel_type;

class Grid
{
private:
  int nx, ny, nz; // Number of voxels
  vector<unsigned char> voxels;
  vector<Vector3> normals;
  vector<Vector3> tangents1;
  vector<Vector3> tangents2;
public:
  unsigned char &get_voxel(const int &i, const int &j, const int &k) {
      return voxels[i*ny*nz + j*nz + k];
  }

  unsigned char &get_voxel(Vector3 &position) {
    int i = nx * (position.x / system_size.x);
    int j = ny * (position.y / system_size.y);
    int k = nz * (position.z / system_size.z);

      return get_voxel(i,j,k);
  }

  // The other three are similar
}
```

Listing 5.3: A Grid class example. This class contains everything we need to know about the geometry.

The first part defines the different values a voxel can have, empty, wall and boundary. The voxel values and associated surface vectors are stored in `std::vector` objects in a linear form (an array with one index). This means that given the three voxel coordinates (i, j, k), they are mapped onto one index as shown in the code example.

5.2.2 Collision cells

As we remember from section 4.3, we will perform collision between particles that are in the same collision cell only. We haven't really talked about what such a collision cell *is* yet, except that its size should be smaller than one third of the mean free path. The simplest way to create these cells is to just divide the total system volume into smaller boxes of equal size. Then each of these cells should have control over which particles that are inside the volume it represents (or owns if you like). Then each timestep, a number of collisions is performed in each cell. This number was found to be (equation (4.9))

$$M_{\text{coll}} = \frac{N_c(N_c - 1)N_{\text{eff}}\pi d^2\langle v_r\rangle\Delta t}{2V_c}.$$

We see that one of the factors is the cell volume V_c. But some of the cells may have large parts of their volume unavailable for fluids, they have a *local porosity*. This is okay, we just need to loop through all of the voxels within a cell and compute the local porosity. An example of how this can be done is shown in listing 5.4.

```
void create_cells() {
    for(int k=0;k<grid.nz;k++) {
        int c_z = float(k)/grid->nz*cells_z;
        for(int j=0;j<grid.ny;j++) {
            int c_y = float(j)/grid->ny*cells_y;

            for(int i=0;i<grid.nx;i++) {
                int c_x = float(i)/grid->nx*cells_x;
                // Find the one-dimensional cell index
                int cell_index = cell_index_from_ijk(c_x,c_y,c_z);
                // Count both the total number of voxels
                // and the number of empty voxels
                Cell &cell = cells.at(cell_index);
                cell.total_voxels++;
                cell.empty_voxels += world_grid->get_voxel(i,j,k)<
                    voxel_type_wall;
            }
        }
    }

    for(int i=0; i<cells.size(); i++) {
      Cell &cell = cells.at(i);
      double cell_porosity = cell.empty_voxels / cell.total_voxels;
      cell.porosity = cell_porosity; // Set porosity
      cell.volume *= cell_porosity;  // Update volume
    }
}
```

Listing 5.4: Example code showing how to find porosity and volume of the collision cells.

5.2.3 Particles

Assuming that we have created the grid object, filled in the voxel array and created all the collision cells, we are ready to create all the particles. As mentioned earlier in this section, we need to specify the system size (L_x, L_y, L_z). The total system volume is then of course found as $V_{\text{system}} = L_x L_y L_z$. But we are going to study a system with a given porosity, the number of volume available to the fluid divided by the total volume. The porosity was found in equation (5.6) by counting all the empty voxels. The volume available for fluid is then $V = \phi V_{\text{system}} = \phi L_x L_y L_z$, which combined with the density ρ_n gives us the total number of *atoms* in the system

$$N_{\text{atoms}} = \rho_n V. \tag{5.7}$$

But we are going to create M particles, each representing N_{eff} real atoms, so the total number of particles in our system becomes

$$M = \frac{N_{\text{atoms}}}{N_{\text{eff}}} = \frac{\rho_n V}{N_{\text{eff}}}. \tag{5.8}$$

Each of these particles is assigned a random position in the physical space. If the particle is placed inside a wall, then we find a new, random position until it is safely placed inside the available pore space. Each particle gets a velocity according to the input temperature T_0 where each velocity component is a normal distribution with standard deviation $\sigma_v = \sqrt{k_B T_0/m}$. Once we have found a position for the particle, we need to add it to the collision cell corresponding to its position. An example code showing how this is done is found in listing 5.5.

```
void initialize_particles() {
  double system_volume = system_size.x * system_size.y * system_size.z;
  int num_particles = density*volume*porosity / num_atoms_per_particle;
  double velocity_standard_deviation = sqrt(boltzmann_constant*temperature
      / mass);
  for(int index=0; index<num_particles; index++) {
    // First, assign velocities from the Maxwell-Boltzmann distribution
    velocities.at(index).x = rnd.nextGauss() * velocity_standard_deviation;
    velocities.at(index).y = rnd.nextGauss() * velocity_standard_deviation;
    velocities.at(index).z = rnd.nextGauss() * velocity_standard_deviation;
```

```
        find_position(index);

        // Find the collision cell and add the particle
        int cell_index = cell_index_from_position( positions.at(index) );
        Cell &cell = cells.at(cell_index);
            cell.add_particle(index);
    }

    void find_position(const int &index) {
        // Assume that the particle is inside the wall
        // until proven otherwise
          bool is_inside_wall = true;
          Vector3 &position = positions.at(index);

          while(is_inside_wall) {
              position.x = system_size.x*rnd->next_double();
              position.y = system_size.y*rnd->next_double();
              position.z = system_size.z*rnd->next_double();

              // Check if this voxel has value equal to voxel_type_wall or
                  voxel_type_surface
              is_inside_wall = world_grid->get_voxel(position)>=voxel_type_wall
                  ;
          }
    }
}
```

Listing 5.5: Particle initialization.

This method will uniformly distribute all the particles in the available pore space. We are now ready to perform the timesteps.

5.3 Timestep

The idea of a timestep is to evolve the system a small amount of time Δt. During this process, the particles are first accelerated before they are moved according to their velocity. If the particles collide with the surface, we will pick a new velocity before they finish the timestep (which may contain many surface collisions). When they have reached their final position, we update the collision cells before particle collisions are performed in each cell. We will now explain each of these stages.

5.3.1 Acceleration

The magnitude of the acceleration is determined by the choice of pressure difference in equation (4.41). The velocity is determined using forward Euler

$$\mathbf{v}_i(t+\Delta t) = \mathbf{v}_i(t) + \mathbf{a}\Delta t, \tag{5.9}$$

for every particle i.

5.3.2 Moving and surface interactions

All our particles now have some velocity, and we can in principle integrate the position with Euler's method

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t)\Delta t. \tag{5.10}$$

But if a particle is very close to the surface, and has a velocity towards it, it might fly into the wall during the timestep. It might even move through a boundary voxel and land into the *inner wall.* We don't want to perform collisions with an inner wall voxel, because it does not have well defined normal and tangent vectors. Instead, we have to back trace and identify which boundary voxel the particle hits first. It is done by a recursive bisection method.

Assume that we have moved our particle from an empty voxel into an inner wall voxel using the whole timestep Δt, see figure 5.3. We then move it back half a timestep to see whether or not it now is in a 1) empty voxel, 2) boundary voxel or 3) wall voxel. If it still is in an inner wall voxel, we move back *another* $\Delta t/4$ (and so on, halfing the distance each step) until we either find the boundary voxel or hit an empty voxel again. If we while moving back hit an empty voxel, we accept the move. We haven't used the full timestep Δt, but a smaller amount of time, τ. This means that we can continue the timestep that now is a bit shorter, $\Delta t - \tau$. If the particle at any point in the algorithm hits the boundary voxel, we're almost done. We then have to compute the exact time δt the particle will use before it collides with the surface of the boundary voxel (the boundary voxel has six faces of which the particle can hit). After the particle is moved to that point, we choose a new, random velocity based on the normal and tanget vectors of that boundary voxel.

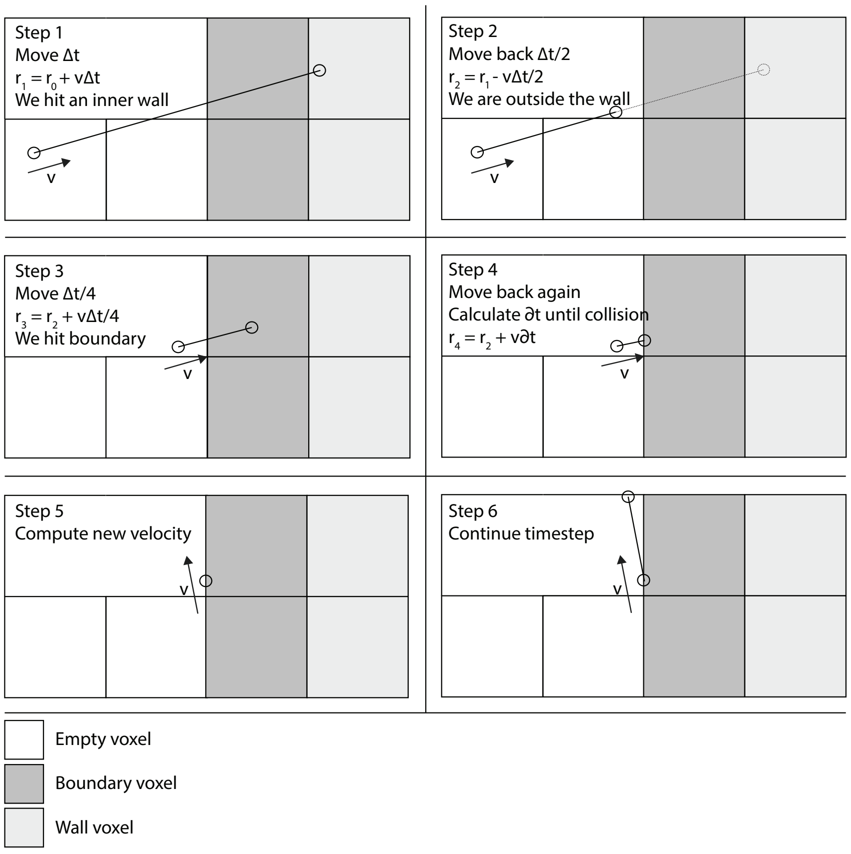

Figure 5.3: The collision detection algorithm. A particle first moves the full timestep Δt (step 1) from an empty voxel to an inner wall voxel. It is then moved back half a timestep $\Delta t/2$ (step 2) to see whether or not it is still in the wall. In step 3 we have accepted step 2 and move the particle another $\Delta t/4$. Now it is in a boundary voxel. We then move back (step 4) to the previous position and compute the amount of time δt it is until collision with the surface of that boundary voxel. In step 5 we choose a new velocity based on the normal and tangent vectors that boundary voxel has. Then we continue the timestep. This whole process may happen several times during one timestep.

The code performing this recursive collision process is found in listing 5.6. This function is called for every particle in the system.

```
void move_particle(int &index, double dt, Random &rnd) {
    double tau = dt; // Time left in the timestep
    Vector3 &position = positions.at(index);
    Vector3 &velocity = velocities.at(index);
    unsigned char &voxel_value = grid.get_voxel(position);
    do_move(position, velocity, tau);
    voxel_value = grid.get_voxel(position);
    // We now have three possible outcomes
    if(voxel_value >= voxel_type_wall) {
        // We hit a wall. First, move back to find boundary
        while(voxel_value != voxel_type_boundary) {
            if(voxel_value == voxel_type_wall) {
                do_move(position, velocity, -tau/2); // Move back half the
                    timestep
                tau /= 2;
                voxel_value = grid.get_voxel(position);
            } else {
                // We have now used tau of the total timestep dt
                dt -= tau;
                // We hit an empty voxel, continue timestep if
                // there is anything left in timestep
                if(dt > 1e-5 && depth < 10) {
                    move_particle(index,dt,rnd);
                    return;
                }
            }
        }
        // This is a boundary voxel
        unsigned char collision_voxel_index = voxel_index;
        while(voxel_value == voxel_type_boundary) {
            collision_voxel_index = voxel_index;
            do_move(position, velocity, -tau); // Move back
            // Compute time until collision with voxel boundary surface
            tau = grid.get_time_until_collision(position, velocity, tau,
                collision_voxel_index);
            do_move(position, velocity, tau);
            voxel_value = grid.get_voxel(position);
        }
        dt -= tau;
        Vector3 &normal = grid.get_normal(collision_voxel_index);
        Vector3 &tangent1 = grid.get_tangent1(collision_voxel_index);
        Vector3 &tangent2 = grid.get_tangent2(collision_voxel_index);
        surface_collider.collide(rnd, velocity, normal, tangent1, tangent2)
            ;
    } else dt = 0; // Didn't hit any surface during the timestep

    if(dt > 1e-5) {
        move_particle(index,dt,rnd);
    }
}
```

Listing 5.6: The collision detection algorithm.

5.3.3 Update collision cells

Now that all the particles have new positions, we need to update the collision cells so that they are aware of the changes. We loop through all particles and see if they have moved into a new collision cell. It is easily understood by a code example, see listing 5.7.

```
void update_collision_cells() {
  for(int index=0; index<num_particles;index++) {
      Cell &old_cell = cell_currently_containing_particle(n);
      Cell &new_cell = cell_that_should_contain_particle(n);
```

```
        if(old_cell.index != new_cell.index) {
            old_cell.remove_particle(index);
            new_cell.add_particle(index);
        }
    }
}
```

Listing 5.7: Updating the collision cell particle lists.

5.3.4 Perform particle collision

The final part of the timestep is to perform collisions within each collision cell. We assume that each cell knows how many collision partner candidates it should try to collide. Each candidate pair is chosen randomly (a particle cannot, of course, collide with itself) and we calculate the relative velocity. This is compared to the maximum relative velocity in that cell (as described in section 4.3) to decide whether or not the pair should collide. If we do get a collision, new random velocities are chosen (conserving both energy and momentum). Again, we illustrate the implementation with a code example, see listing 5.8.

```
void collide(Random &rnd) {
  // Loop over the candidate collision pairs
    for(int collision=0; collision<collision_pairs; collision++ ) {
    // Pick two particles at random
        int index_0 = (int)(rnd.next_double()*num_particles);
        int index_1 = ((int)(index_0+1+rnd.next_double()*(num_particles-1))
            ) % num_particles;

        // These indices are local indices in this cell. Find the global
            indices.
        index_0 = global_particle_indices[index_0];
        index_1 = global_particle_indices[index_1];

    // Calculate pair's relative speed
        Vector3 &v0 = velocities.at(index_0);
        Vector3 &v1 = velocities.at(index_1);
        double v_rel = (v0 - v1).length();

        // Update if new maximum relative velocity
        if( v_rel > vr_max ) {
            vr_max = v_rel;
        }

    // Accept or reject candidate pair according to relative speed
        if( v_rel > rnd.next_double()*vr_max ) {
            collide_particles(v0, v1, v_rel, rnd);
    }
  }
}

void collide_particles(Vector3 &v0, Vector3 &v1, Random &rnd) {
    Vector3 v_center_of_mass = 0.5*(v0 + v1);
    double v_rel = (v1 - v2).length();

    double cos_theta = 1.0 - 2.0*rnd.next_double();
    double sin_theta = sqrt(1.0 - cos_theta*cos_theta);
    double phi = 2*M_PI*rnd.next_double();

    Vector3 relative_velocity(1,1,1)*v_rel;
    relative_velocity.x *= cos_theta;
    relative_velocity.y *= sin_theta*cos(phi);
    relative_velocity.z *= sin_theta*sin(phi);
    v0 = v_center_of_mass + 0.5*relative_velocity;
    v1 = v_center_of_mass - 0.5*relative_velocity;
}
```

Listing 5.8: Example code showing how to perform collisions.

5.3.5 Final comments

We have now discussed the implementation and explained the details about every step in the algorithm. The sampling of statistics is trivial and is done as explained in section 4.4. After the simulation is completed, the state containing all positions and velocities of the particles is saved to disk so we can continue the simulation at a later time, or, as we will discuss in chapter 11, visualize the particles and their trajectories. Now we just need to explain how the parallelization is implemented.

5.4 Parallelization

The parallelization complicates parts of the implementation steps described in section 5.3. For example, the file containing the geometry information is split into several files, one per processor. Each processor will then only know about a subset of the whole system geometry. To simplify this reading, we will only explain the basic philosophy of how the code is parallelized.

Each collision cell is completely independent of the other cells. We can then divide the spatial domain into sub domains, each fully controlled by one processor. Each processor is responsible for executing the timestep for every particle in the corresponding volume. The processors will usually contain many collision cells as illustrated in figure 5.4. We will use the terms *processor*, *node* and *CPU* interchangeably. Given that the processors have knowledge about the particles living in its volume, the only thing we have to take care of is when particles move from one processor to another. We have used (MPI) for the communication between processors, and assume that the reader is familiar with how MPI works.

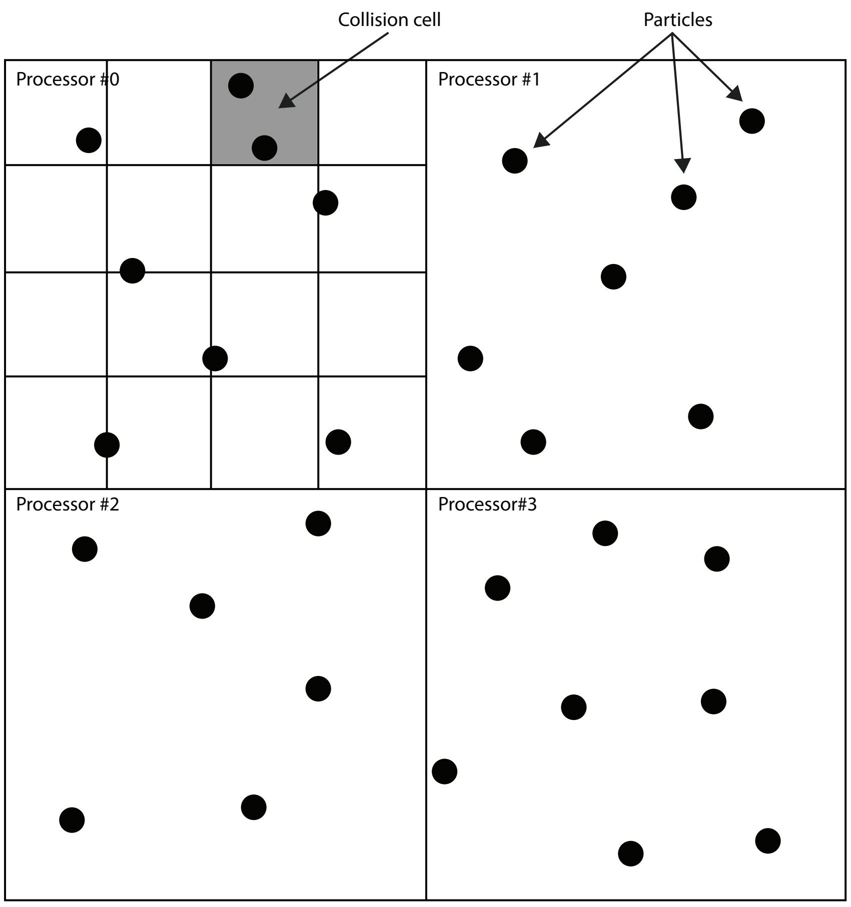

Figure 5.4: Illustration of how the spatial domain can be divided into four sub domains, each controlled by a processor. Each processor contains many particles that are placed in several collision cells (marked grey).

5.4.1 Topological structure

The processors are divided into a three dimensional grid with (P_x, P_y, P_z) being the number of CPU's in each dimension, yielding a total of $P = P_xP_yP_z$ processors. We can then use the grid coordinates (p_x, p_y, p_z) to uniquely label the processors as shown in figure 5.5. When starting a program with MPI, each process is provided a unique identification number p in the range $[0, P-1]$ for P processors. This can be mapped to the 3-dimensional grid coordinates through

$$\begin{aligned} p_x(p) &= \frac{p}{P_yP_z} \\ p_y(p) &= \frac{p}{P_z} \bmod P_y \\ p_z(p) &= p \bmod P_z \end{aligned} \tag{5.11}$$

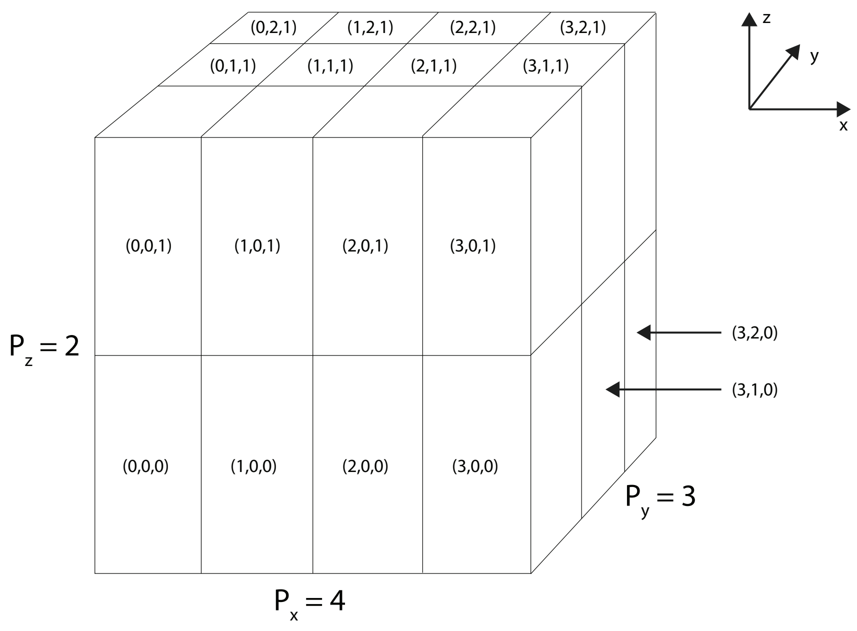

Figure 5.5: Processor labeling in a 3-dimensional grid. Each processor is uniquely identified through its coordinate (p_x, p_y, p_z).

whereas the inverted mapping is

$$p(p_x, p_y, p_z) = p_x P_z P_y + p_y P_z + p_z. \tag{5.12}$$

With the processor id p given, it is easy to determine which sub volume this processor should control. If the system is of size L_i in the i'th dimension, we can find the *node length* $L_i^{\text{node}} \equiv l_i = L_i/P_i$. A processor with coordinates (p_x, p_y, p_z) will control all particles with coordinates in the range

$$\begin{aligned} x &\in [p_x l_x, (p_x+1)l_x\rangle \\ y &\in [p_y l_y, (p_y+1)l_y\rangle \\ z &\in [p_z l_z, (p_z+1)l_z\rangle. \end{aligned} \tag{5.13}$$

Since the collision cells are independent of each other, collisions will happen in parallel where each processor loops through all of its cells colliding the particles as described in section 5.3.

5.4.2 Exchanging particles

During a timestep, a particle can move from one processor to one of the 26 neighboring nodes (the middle node in a $3 \times 3 \times 3$ grid has a 26 neighbors). After each timestep, all processors loop through their particles to find the ones having moved out of the processor's spatial domain. This process is illustrated in listing 5.9.

```
double mpi_move() {
  for(int n=0; n<num_particles_local; n++) {
    int node_id = topology->index_from_particle_index(n);
    if(node_id != myid) {
      // Particle belongs to another node
    }
  }
}
```

Listing 5.9: Detecting which particles moved out of a processor's spatial domain.

In principle, there are 26 potential receiving nodes, so each node needs to be able to send information to all of them. The easiest way to implement this is to let each processor directly communicate with all of its neighbors. However, this approach requires a lot more communication time than actually needed.

If we instead send information about these particles through a 3-step process, we can reduce the number of neighboring nodes from 26 to 6. This idea is best illustrated in two dimensions, but is easily generalized to the three-dimensional case, see figure 5.6. An analysis of the parallelization is studied in subsection 6.2 where we compare how well the program performance scales with increasing number of processors.

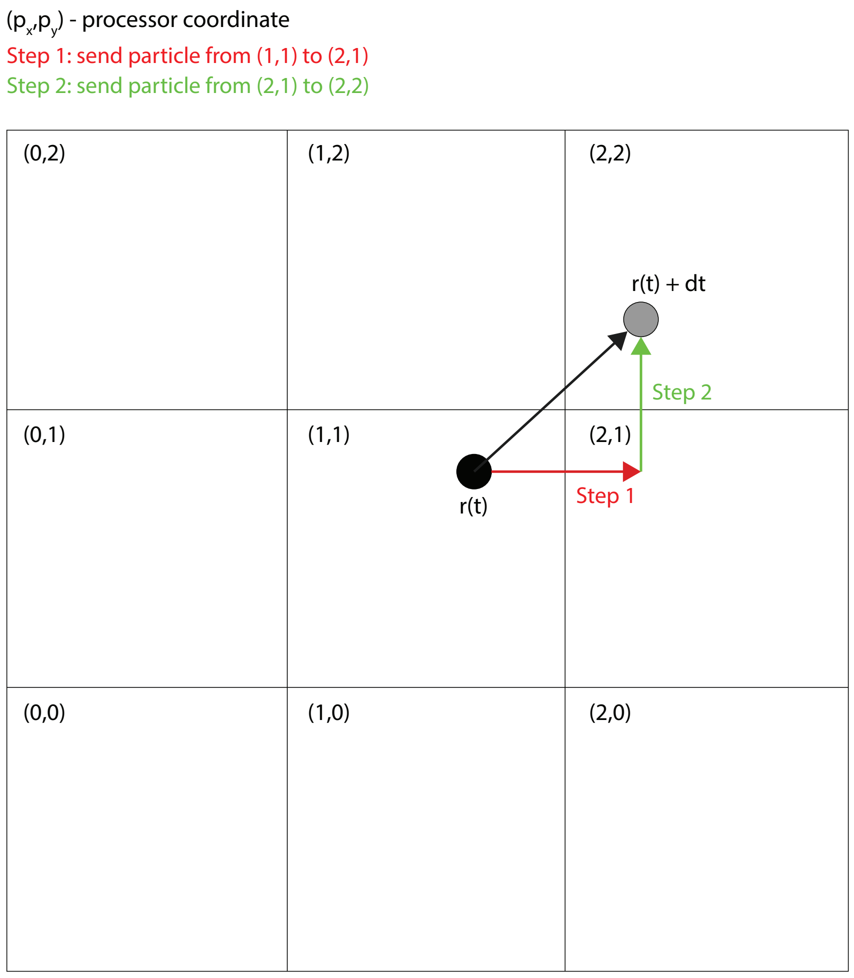
(p_x,p_y) - processor coordinate
Step 1: send particle from (1,1) to (2,1)
Step 2: send particle from (2,1) to (2,2)
(0,2)
(1,2)
(2,2)
r(t) + dt
Step 2
(0,1)
(1,1)
(2,1)
r(t)
Step 1
(0,0)
(1,0)
(2,0)

Figure 5.6: The middle node (1,1) has 8 neighbors it needs to communicate with. Each node only needs to communicate with its nearest neighbors (4 in two dimensions, 6 in three dimensions), because the nearest neighbors can work as intermediate information carriers. A particle that moves from processor (1,1) to (2,2) will in step 1 be sent to (2,1), then in step 2 be sent to (2,2).

Chapter 6

Results

Every time a scientist creates or implements a model, it is important to verify that the model is giving correct results. Or maybe we should rephrase; it is important to verify that the model is *doing as intended.* It is not always clear what correct results *means.* As George E.P. Box said

Essentially, all models are wrong, but some are useful. (George E.P. Box, 1987)

A good model should be in agreement with already existing theories (at least those we believe are reliable). Take special relativity for example, in the limit of low velocities, it reproduces the results of Newtonian mechanics (e.g. the effects time dilation and length contraction vanishes). In this chapter we will first validate the model by comparing the outcome of the model to established theoretical results in section 6.1. In section 6.2 we discuss how well the performance of the implementation scales for an increasing number of processors. We then, in section 6.3, analyze the Knudsen correction for the permeability for flow in a cylinder for various Knudsen numbers and cylinder radii. We conclude the chapter by studying a more complicated geometry, randomly packed spheres, in section 6.4. In all simulations we have done, the mass m was chosen to be 6.63×10^{-26} kg and the diameter d is 3.62×10^{-10} m. These are values describing argon, meaning the gas in the system is an argon gas. The timestep is approximately 1 ps in all simulations. This is smaller than the mean collision time for gases with density less than 4.0×10^{27} m^{-3} (equation (3.56)).

6.1 Code validation

We validate the DSMC implementation by comparing numerical results to those of statistical mechanics and the kinetic theory. First we let the particles start in some non-equilibrium state where all the velocity components are $\pm v_0$ to confirm that the collision model results in the Maxwell-Boltzmann velocity distribution. The velocity v_0 corresponds to some initial temperature T_0 that defines the standard deviation in the Maxwell-Boltzmann distribution. Then we study the Poisuille flow (two infinite parallel plates) and compare the spatial velocity distribution $v(x)$ - x being the distance from the lower plate - to numerical solutions to the linearized Boltzmann equation.

6.1.1 Maxwell-Boltzmann velocity distribution

We showed in section 3.6 that the particles in a gas in equilibrium will have velocities according to the Maxwell-Boltzmann distribution. We have run a simulation with 10^5 simulated particles with initial velocities given for each particle n as

$$\begin{aligned}\dot{x}_n &= v_0(1-2(n \bmod 2))\\ \dot{y}_n &= v_0(1-2((n+1) \bmod 2))\\ \dot{z}_n &= \dot{x}_n,\end{aligned} \tag{6.1}$$

where v_0 is some velocity chosen so that $T \approx 300K$. There are no walls the particles can exchange energy with, so the total energy is conserved. The reason we have chosen this initial velocity distribution is that the net momentum will be zero if N is an even number (each particle's velocity components is canceled out by the next particle). This is is of course not the Maxwell-Boltzmann distribution - we are not in an equilibrium state. We expect that after some time, the system will equilibrate and approach the Maxwell-Boltzmann distribution which is given as (equation (3.45))

$$P(v_i)\mathrm{d}v_i = \sqrt{\frac{m}{2\pi k_B T}}e^{\frac{-mv_i^2}{2k_BT}}\mathrm{d}v_i, \tag{6.2}$$

where $i \in \{x, y, z\}$ indicates the velocity component. We ran the simulation for 10^5 timesteps, and as we see in figure 6.1, the collision algorithm reproduces this distribution perfectly. The temperature that was measured to be $T = 298.5$ K.

6.1.2 Poiseuille velocity profile

The Poiseuille flow through a long channel is a standard and fundamental problem that is widely studied in the gas dynamics literature. The system consists of two infinite parallel plates, displaced by a distance h in the y-direction. A pressure gradient is applied in the z-direction by a constant acceleration g corresponding to equation 4.41. The channel is periodic in the x and $z-$direction, emulating a large system with negligible boundary effects. The gas particles collide with the plates through the thermal wall model, so the reflected particles will have zero tangential velocity on average. In the continuum limit, $\mathrm{Kn} \to 0$, we expect the velocity profile to approach the parabolic solution [4]

$$v_z(y) = \nabla P \frac{1}{2\mu} y(y-h). \tag{6.3}$$

However, in the transition flow regime, $0.1 \leq \mathrm{Kn} \leq 10$, we expect a non zero slip velocity[20]. The Knudsen number will affect the velocity distribution through the fact that a high Knudsen number means a larger mean free path, hence fewer inter-molecular collisions. A low collision

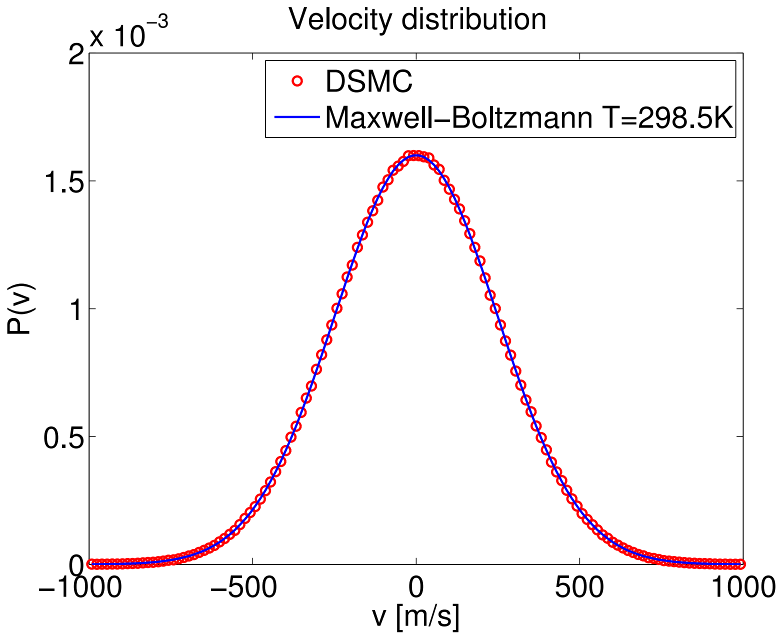

Figure 6.1: The system clearly reaches the Maxwell-Boltzmann distribution when the initial velocity distribution follows equation 6.1. Here the final temperature was measured to T =298.5 K which reproduces the expected distribution perfectly.

rate will make the surface effects propagate slower through the system resulting in a higher slip velocity. Ohwada et al. [22] studied the Poiseuille flow for hard-sphere molecules with a wide range of Knudsen numbers by numerically solving the linearized Boltzmann equation. By using non-dimensionalized units, the result is not dependent on the pressure difference or the temperature. The only assumption is that the pressure gradient is so small that the Boltzmann equation can be linearized around the equilibrium state. To set up as system like this is a simple task with the DSMC algorithm, and is a good test case to validate both the surface interaction model and the inter-molecular collision model.

We chose a cubic system with side length 1.0 μm with a solid plate at y =0 μm and y =1.0 μm. We ran six simulations with different densities, corresponding to different Knudsen numbers in the range $[0.1, 11.0]$. We want to vary the Knudsen number which was defined as

$$\mathrm{Kn} = \frac{\lambda}{L} = \frac{1}{\sqrt{2}\pi d^2 \rho_n L}. \tag{6.4}$$

We have substituted the mean free path from equation (3.55) in the last expression so that we can choose the Knudsen number through the density

$$\rho_n(\mathrm{Kn}) = \frac{1}{\sqrt{2}\pi d^2 \mathrm{Kn} L}. \tag{6.5}$$

In all runs, we chose the number of real atoms per particle $N_{\text{eff}} = 10$. Assuming ideal gas pressure P_0, an applied pressure difference corresponding to $0.1P_0$ was chosen to induce the flow. In figure

6.2 we see that the velocity profiles produced from the DSMC model is in excellent agreement with the solutions obtained by Ohwada et al. for all Knudsen numbers. We also see that as the Knudsen number increases, the slip velocity also increases, which is what we expect, see section 2.9.

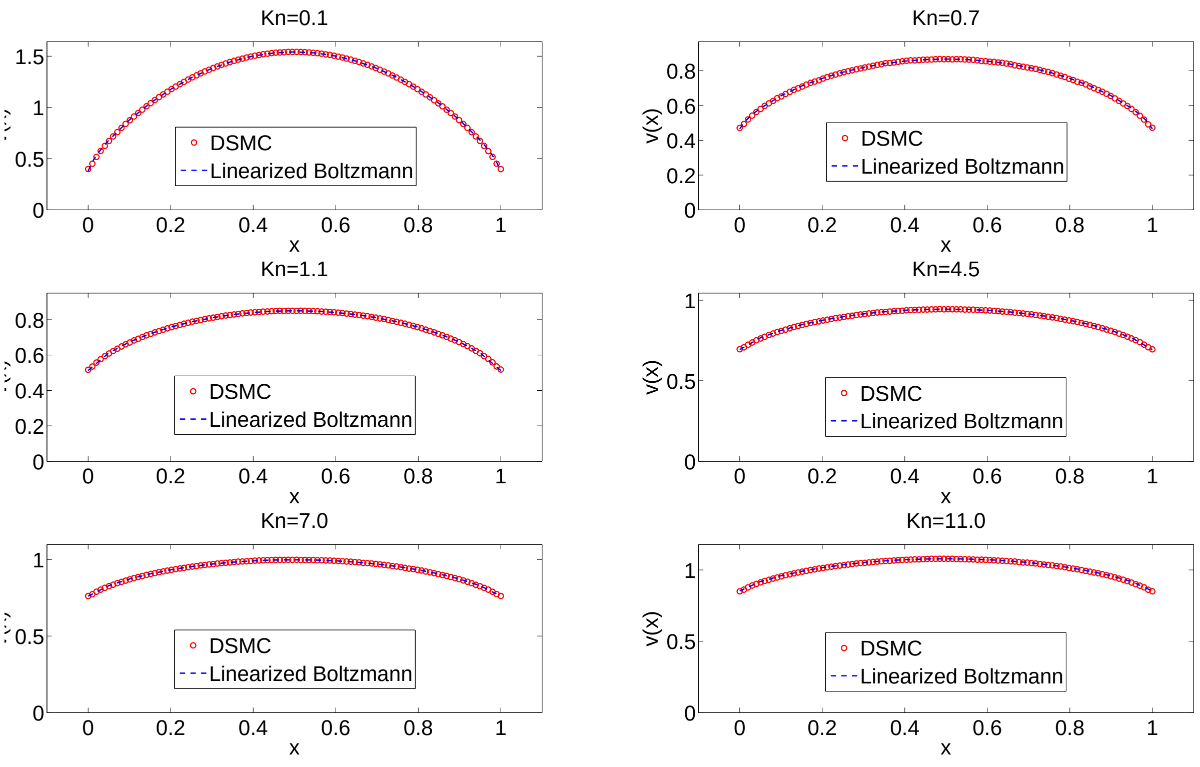

Figure 6.2: Non-dimensionalized velocity distribution for flow between infinite parallel thermal plates for different Knudsen numbers over two orders of magnitude. Each run is compared to the linearized Boltzmann solution of Ohwada et al. [22] which shows that the algorithm produces correctly the behavior in all regions, near the wall and internally in the channel. An applied pressure difference $\Delta P = P_A - P_B$ with $P_A = 1.1 P_B$ is applied. The initial temperature is T=300 K. We see how the slip velocity becomes larger for increasing Knudsen numbers, as expected from the discussion in section 2.9.

6.2 Parallelization performance

No matter how many processors we distribute the work load to, the total computation time needed to perform a simulation is obviously not reduced. There are still as many collisions as before as the physical problem is identical. The idea is to do many calculations at the same time so that the *real* time we actually wait is reduced. Parallelizing comes with a cost, as we need more logic to allow the processors communicate with each other (exchanging particles and waiting for each other to finish each time step). As the number of processors increases, the total computation time *per processor* is reduced, but the time spent on communication often increases. We will measure what's called *parallel scalability* which indicates how efficient a program is when the number of processors is increased. There are two different kinds of scalability - weak and strong scaling

- strong scaling is how the computation time changes with an increased number of processors on a fixed system size, whereas the
- weak scaling is how the computation time changes with an increased number of processors on a fixed system size *per processor*.

6.2.1 Strong scaling

To see how the program efficiency scales with a fixed system size while increasing the number of processors is important if we want to study a specific system (a given system size and geometry, e.g. scanned from real data), but we would like to reduce the simulation run time. With a fixed system size, the total number of particles per CPU is reduced while increasing the total number of processors. We define the *strong scaling efficiency* η_s as

$$\eta_s = \frac{t_1}{N t_N}, \tag{6.6}$$

where t_1 is the total run time using one processor and t_N is the total run time using N processors. We see that $\eta_s \in (0,1)$ since Nt_N is the ideal scaling without any communication overhead. In this benchmark, we have run the simulation in an empty geometry - there are no surfaces to collide with.

The physical system is a cube with side length L =1.0 µm with a density $\rho_n = 1.0 \times 10^{27}$ m^{-3} which gives a total of one billion atoms. We choose that one DSMC particle represents 50 atoms, yielding a total of 20 million particles in the whole system. The benchmark was performed for 10000 timesteps ($\Delta t = 0.005$) with 2^N processors from 1 CPU to 512 CPUs yielding a good estimate of how efficient the program scales for a relatively large number of processors. In table 6.1 we have the scaling results with additional information like the number of intermolecular collisions per processor. This indicates the amount of computation each processor has to perform.

The strong scaling efficiency is plotted together with the weak scaling against the number of processors in figure 6.3. We see that the program scales great with only 30% overhead for 512 processors. The weird behavior we see at 128 processors, where the efficiency *increases* to 0.9, is probably due to random luck on the supercomputer where all nodes are physically close to each other. In order to get better statistics, this benchmark should be run several times.

N_{CPU}	$N_{\text{particles}}/N_{\text{CPU}}$	$N_{\text{collisions}}/N_{\text{CPU}}$	t_n [s]	η_s
1	2.0×10^7	1.16×10^{11}	90802 s	1.0
8	2.5×10^6	1.45×10^{10}	15372 s	0.74
16	1.25×10^6	7.25×10^9	6752 s	0.84
32	6.3×10^5	3.62×10^9	3805 s	0.75
64	3.1×10^5	1.81×10^9	1706 s	0.83
128	1.6×10^5	9.06×10^8	785 s	0.90
256	7.8×10^4	4.53×10^8	415 s	0.85
512	3.9×10^4	2.27×10^8	250 s	0.71

Table 6.1: Benchmark results showing the strong scaling efficiency η_s for the DSMC program. We see that the program scales great with only 30% overhead for 512 processors. The weird behavior we see at 128 processors, where the efficiency *increases* to 0.9, is probably due to random luck on the supercomputer where all nodes are physically close to each other. In order to get better statistics, this benchmark should be run several times.

6.2.2 Weak scaling

Another important scaling problem appears when we want to maximize the simulated system size. If we keep a constant system size *per CPU*, and increase the number of processors, the limitation of how big we efficiently can create the system is controlled by the weak scaling. We then introduce the *weak scaling efficiency* η_w defined as

$$\eta_w = \frac{t_1}{t_N}, \tag{6.7}$$

where again t_1 is the total run time using one processor and t_N is the run time using N processors. If the algorithm scales perfectly, the total run time would remain constant while increasing the number of processors (each CPU is ideally independent), but we expect some overhead. This implies that the range for η_w also is between zero and one. We have run the same geometry as for the strong scaling, but each processor controls a volume of 1 μm^3 so that the largest system is 512 μm^3. Keeping the same density, but using 500 atoms per particle, gives a total of 2 million particles per processor. In table 6.2 and figure 6.3, we see the results for the weak scaling efficiency simulation. The weak scaling also shows promising results with a reasonable low overhead.

N_{CPU}	$N_{\text{particles}}$	$N_{\text{collisions}}$	t_N	η_w
1	2.00×10^6	5.83×10^9	3450 s	1.0
2	4.00×10^6	1.16×10^{10}	3640 s	0.95
4	8.00×10^6	2.33×10^{10}	5190 s	0.67
8	1.60×10^7	4.66×10^{10}	4700 s	0.73
16	3.20×10^7	9.31×10^{10}	6620 s	0.52
32	6.40×10^7	1.86×10^{11}	5470 s	0.63
64	1.28×10^8	3.73×10^{11}	5360 s	0.64
128	2.56×10^8	7.45×10^{11}	5760 s	0.60
256	5.12×10^8	1.49×10^{12}	5430 s	0.64
512	1.02×10^9	2.98×10^{12}	5450 s	0.63

Table 6.2: Benchmark results showing the weak scaling efficiency η_w for the DSMC program.

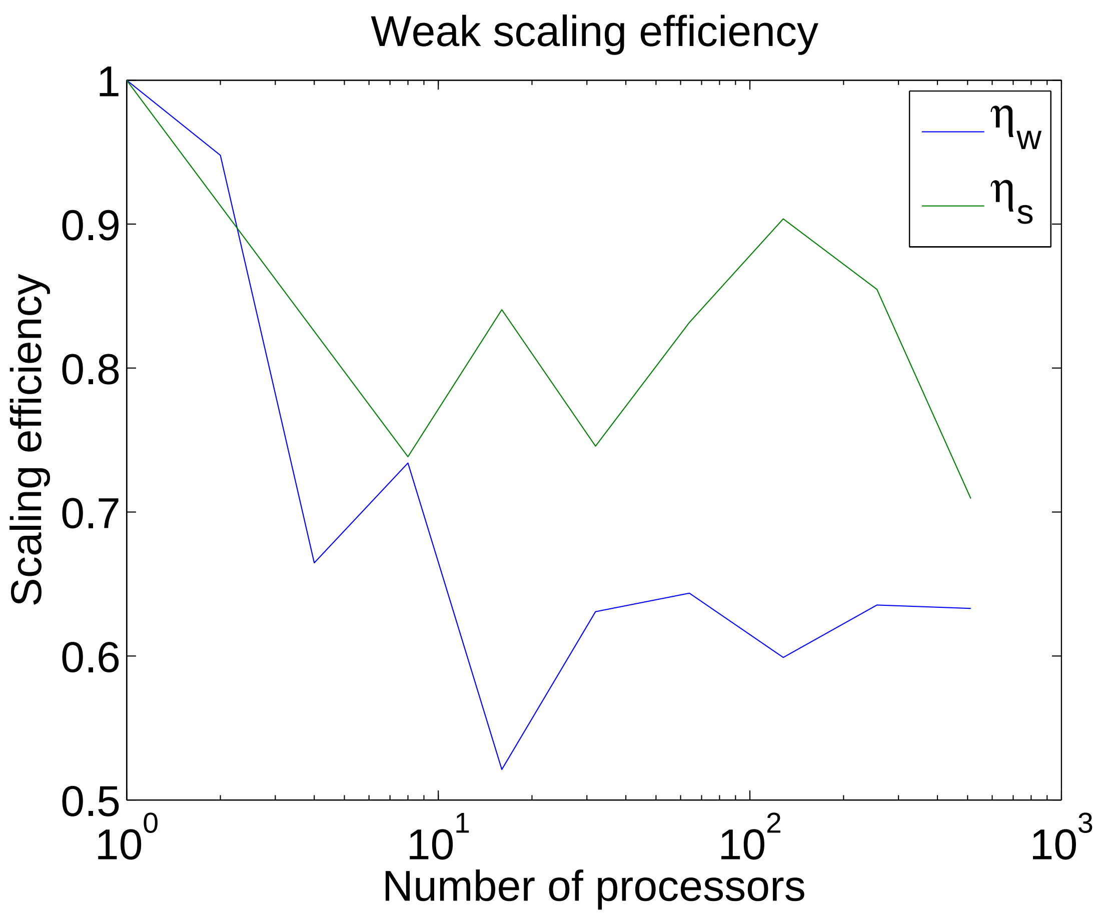

Figure 6.3: The weak and strong scaling efficiency, η_w and η_s, as a function of the number of processors N_{CPU}. We see that at 512 processors, both efficiencies are reduced to 60-70% because of the MPI communication overhead. Despite the bad statistics, it seems like the efficiency has more or less converged to a satisfactory level.

6.3 Results for simple geometries

In this section, we study flow in a simple geometry where the permeability is known from theory. The expression for the permeability is only valid for small Knudsen numbers (which we called the absolute permeability; the permeability for fluids in the continuum limit), so it is a perfect test case for the Knudsen correction factor f_c in equation (2.19).

6.3.1 Flow in a cylinder, varying Knudsen number

We have induced flow in a cylinder with radius 0.45 μm with an applied acceleration corresponding to a pressure difference $\Delta P = 0.1 P_0$, where P_0 is the ideal gas pressure at 300 K. While varying the density, we adjusted N_{eff} so that the total number of simulated particles was approximately one million. We expect an apparent permeability satisfying the Knudsen correction

$$k_a = k_\infty f_c = k_\infty [1 + \alpha(\text{Kn})\text{Kn}] \left[1 + \frac{4\text{Kn}}{1 + \text{Kn}}\right]. \tag{6.8}$$

The analytical absolute permeability for a cylinder with radius r is given by[15]

$$k_\infty = \frac{r^2}{8}, \tag{6.9}$$

which gives the following prediction for the apparent permeability

$$k_a = [1 + \alpha(\text{Kn})\text{Kn}] \left[1 + \frac{4\text{Kn}}{1 + \text{Kn}}\right] \frac{r^2}{8}. \tag{6.10}$$

After 200k timesteps with sampling, the permeability was calculated according to equation 4.32. In figure 6.4 we have plotted the measured permeability as a function of Knudsen number with the Knudsen corrected analytical solution. The left figure has the Knudsen numbers on a logarithmic x-axis to emphasize the lower Knudsen numbers. We see that the measured permeability is slightly lower than predicted for the higher Knudsen numbers. This is probably an effect from the voxelization of the cylinder where the effective radius is slightly smaller than desired. By reducing the radius by a factor 0.99 in equation (6.10), the measured permeability perfectly overlaps the analytical solution.

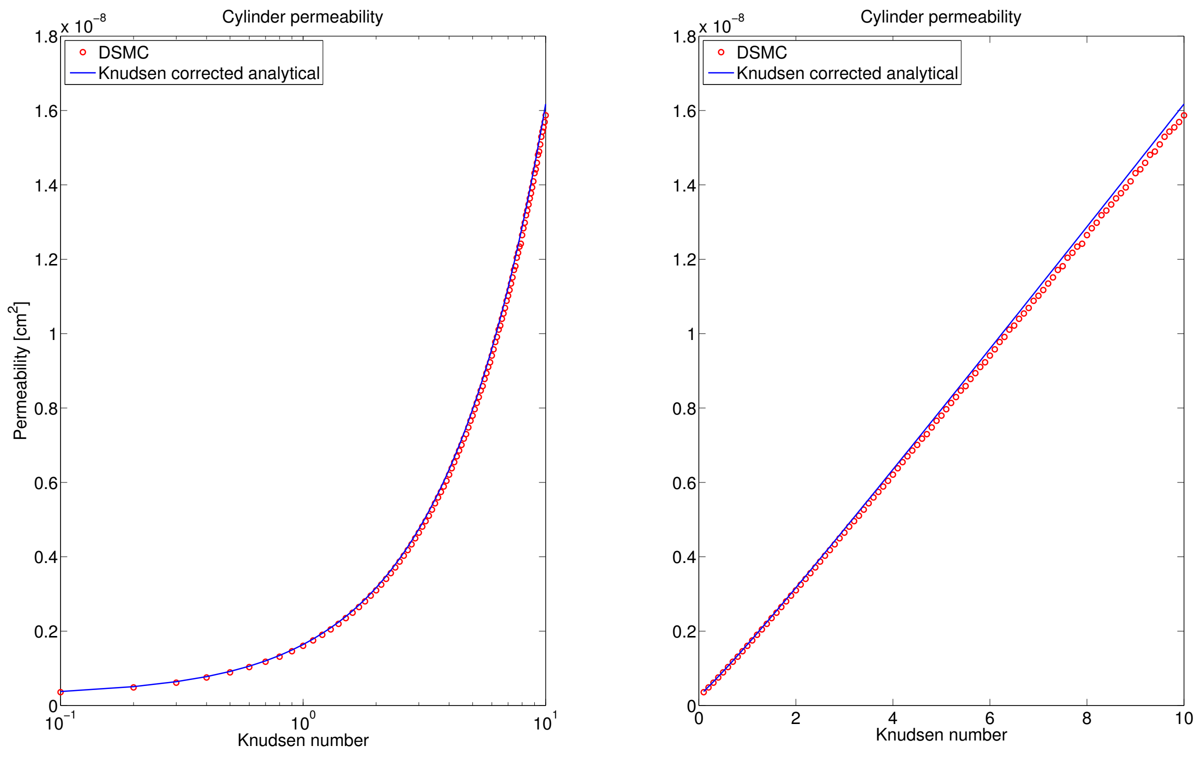

Figure 6.4: Permeability as a function of Knudsen number for a cylinder with radius 0.45 µm and length 1 µm with an applied pressure difference $\Delta P = 0.1P_0$, P_0 being the ideal gas pressure. We control the Knudsen number by varying the density. The blue line is the Knudsen corrected analytical solution in equation (6.10). We see a slightly lower permeability than the analytical solution, which probably is an effect from the voxelization of the cylinder where the effective radius is slightly smaller than desired. By reducing the radius by a factor 0.99 in equation (6.10), the measured permeability perfectly overlaps the analytical solution.

6.3.2 Flow in a cylinder, varying radius

If we instead keep the Knudsen number constant (Kn = 1.0), we can vary the radius to verify equation (6.9). We have studied radii in the range 0.1 µm to 0.45 µm with the same pressure difference as in the previous simulation ($\Delta P = 0.1P_0$). In figure 6.5 we have plotted the measured permeability as a function of cylinder radius. The straight line confirms the quadratic dependency in equation (6.9).

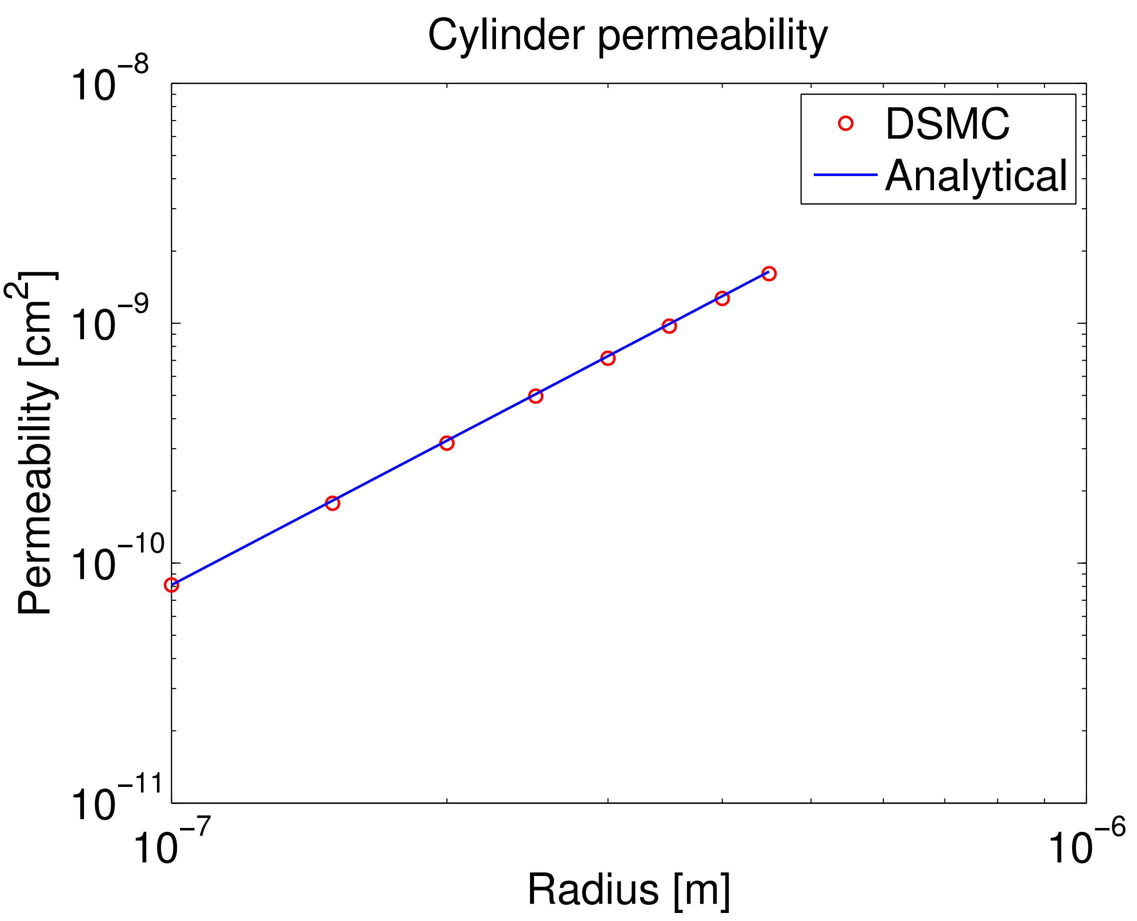

Figure 6.5: Logarithmic plot of the permeability for different cylinders with radii in the range 0.1 µm to 0.45 µm with an applied pressure difference $\Delta P = 0.1P_0$, P_0 being the ideal gas pressure. The blue line is the Knudsen corrected analytical solution from [15]. The permeability scales with the radius as r^2.

6.4 Results for complicated geometries

We have now seen that the Knudsen correction factor works well for systems with a well defined Knudsen number. It does however need *one* Knudsen number to be able to predict the permeability, but for more complex geometries, there is rather a distribution of Knudsen numbers than a single number. It could work as a lower and an upper limit of the permeabilities for two different input Knudsen numbers, but they could possibly differ by an order of magnitude. Another approach could be to find the average distance $\langle L \rangle$ to the surface and use that distance to estimate the Knudsen number as

$$\mathrm{Kn}^* = \frac{\lambda}{\langle L \rangle}. \tag{6.11}$$

In this section we will study a system consisting of randomly packed spheres that does not have a well defined Knudsen number. The spheres may overlap, so their positions are completely independent of each other. The distribution and expectation value of distances to sphere surfaces is derived in appendix A also providing a suggestion to the estimated Knudsen number $\mathrm{Kn}(r,\phi)^*$ for packed spheres of radius r and porosity ϕ. First we discuss the Carman-Kozeny equation which is an analytical expression for the permeability for packed spheres. Then we discuss how the simulation is run and discuss the result.

6.4.1 The Carman-Kozeny equation

In 1927, Kozeny proposed an equation predicting the (absolute) permeability of packed spheres of radius r forming a system with porosity ϕ given as

$$k_\infty = \frac{r^2}{9K}\frac{\phi^3}{(1-\phi)^2}, \tag{6.12}$$

where K is the Kozeny constant which is experimentally measured to be around five for equally size spheres[5]. This result has been verified to predict permeabilities in many macro scale experiments since its discovery. However, at the nanometer scale, we expect deviations due to high Knudsen numbers. Using the Knudsen correction factor with the estimated Knudsen number $f_c(\mathrm{Kn}(r,\phi)^*)$ (equations (2.19) and (A.21)), we expect the permeability to be

$$k_a(r) = f_c\left[\mathrm{Kn}(r,\phi)^*\right]\frac{r^2}{9K}\frac{\phi^3}{(1-\phi)^2}. \tag{6.13}$$

6.4.2 The simulation and results

We ran the simulation for spheres with radii from 0.01 µm to 0.08 µm while keeping approximately constant porosity, $\phi \approx 0.3$. This was done by adding spheres until the desired porosity was obtained. For each radius, ten geometries were created with different seeds yielding better statistics. The permeability for each sphere radius is then averaged over the ten different geometries. In figure 6.6, we have plotted the permeabilities measured in the simulations with the average value of the ten runs per sphere radius. We have also plotted the Knudsen corrected Carman-Kozeny permeability which gives a pretty good estimate for the permeability.

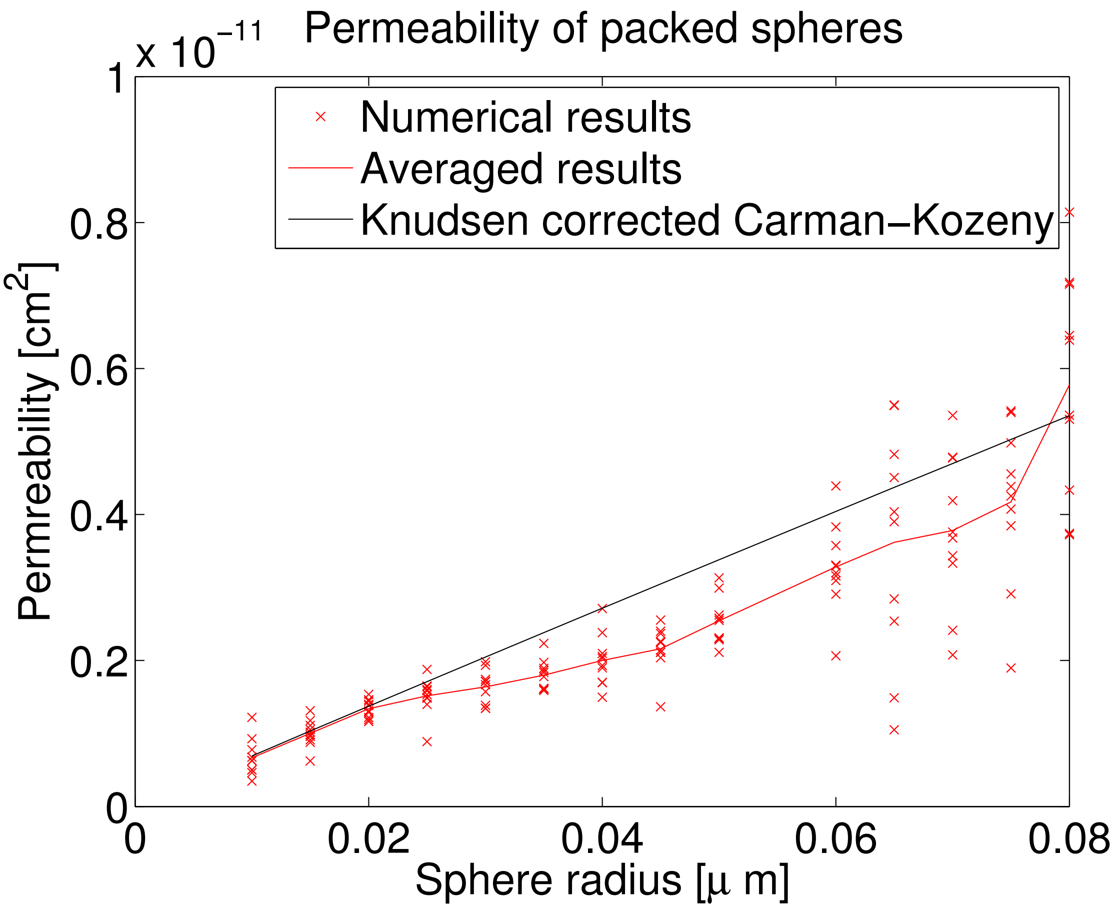

Figure 6.6: Permeability for different sphere radii in a system consisting of packed spheres. The red dots show the numerical results whereas the red line shows the average value for each sphere radius. The black line is the Knudsen corrected Carman-Kozeny expression for the permeability with the estimated Knudsen number (equation (A.21)). We see that due to the large statistical spread in the sphere configurations, the spread in permeability is also quite large. While the Knudsen corrected Carman-Kozeny expression does give a good estimate of the permeability, it is clear when the spread in Knudsen numbers is large, it is not sufficient with one single average value.

Part II

Molecular Dynamics

Chapter 7

Theory

Molecular Dynamics is another numerical model that describes the behavior of liquids, gases and solids at the finest scale of any classical model. We can study the dynamics of single atoms and how they interact with each other forming molecules and larger objects like advanced pore networks. The idea is simple and has been used since the time of Sir Isaac Newton in the 17th century when he formulated his laws of motion. With the knowledge of the relevant forces between the atoms, we can solve Newton's equations and calculate their dynamics.

We open this chapter by a short introduction to the model in section 7.1 before we explain how forces are calculated with the Lennard-Jones potential in section 7.2. With computed forces, we can integrate the atoms with Newton's laws following an time integration scheme which we discuss in section 7.3. The last section is concerned with how we keep a constant temperature using a thermostat. Physical quantities are measured in the same way as we did in DSMC in section 4.4.

7.1 The model

The state of a Molecular Dynamics system is fully described by seven variables per atom; three positions, three velocities plus the atom type. The phase variables are evolved through the laws of motion. It is here common, as in DSMC (see section 4.1), to apply periodic boundary conditions. Applied periodic boundary conditions in all directions implies constant volume (unless, of course, we rescale the system size which can be done). The atomic forces are calculated as the gradient of a chosen potential that can differ quite a lot depending on the requirements of the model. A noble gas like Argon can be modeled with a simple potential called the Lennard-Jones potential which will be discussed below. With this potential, one can calculate equilibrium thermodynamic properties that are in good agreement with experimental values [26]. System statistics are sampled as ensemble averages through ergodicity over large times. A typical Molecular Dynamics algorithm is illustrated in figure 7.1.

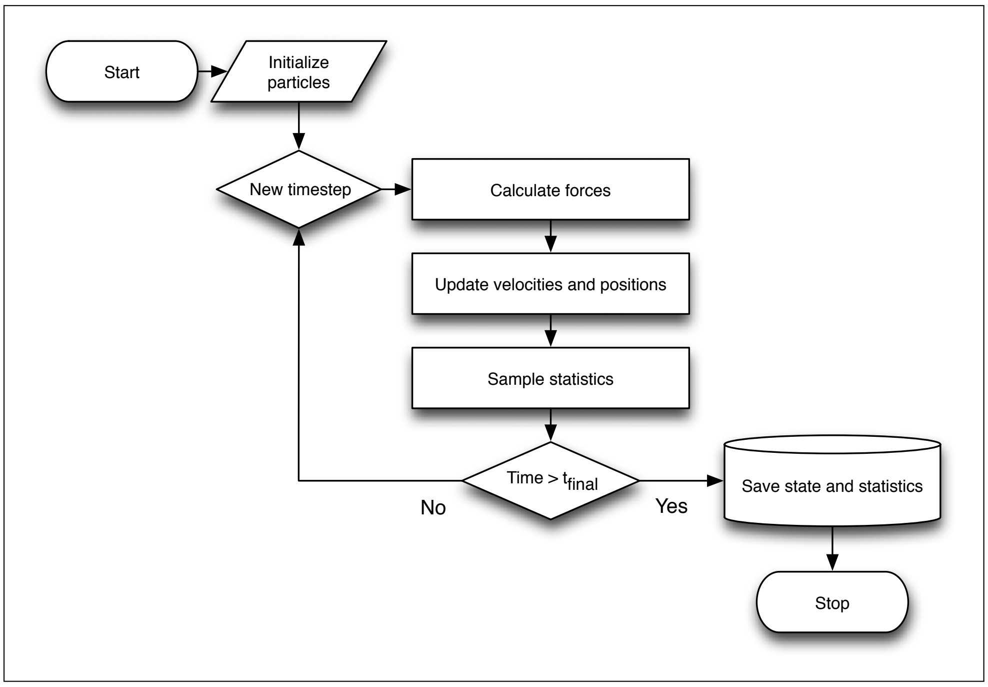

Figure 7.1: Flow chart illustrating a typical Molecular Dynamics algorithm.

7.2 Force calculation - potentials

In general, the potential energy is given as a function of the configuration given by the atom positions

$$U(\mathbf{r}) = \sum_{i<j} U_2(r_{ij}) + \sum_{i<j<k} U_r(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) + ..., \tag{7.1}$$

where $\mathbf{r}$ is the phase space point, U_n is a function of the positions of n atoms, r_{ij} is the relative distance between atom i and j, $\mathbf{r}_i$ is the position of atom i. Advanced potentials such as ReaxFF can even have 5-atom contributions to the energy[24]. The reason for this is simple; when three atoms are close to each other, the electron configuration might be different than if there were only two atoms. These effects play a large role in forming molecules, such as water. Numerically, the calculation of forces is the most expensive part of the whole program, so for simple systems or educationally purposes it might be sufficient to use the Lennard-Jones potential.

7.2.1 The Lennard-Jones potential

We often see this potential referred to as the 6-12 potential, due to its functional form. It is a function that inherently carries the two main properties of atomic forces; the short-ranged Pauli repulsion and the long-ranged van der Waals force. The potential is only between pairs of atoms

$$U_{LJ}(r_{ij}) = 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right], \tag{7.2}$$

where r_{ij} is the relative distance between atom i and j, ϵ and σ are coupling constants giving the depth of the potential well and the distance where the potential is zero. The force is given as the gradient of this potential yielding

$$\mathbf{F}_{LJ}(\mathbf{r_{ij}}) = -\nabla U_{LJ}(r_{ij}) = -24\epsilon \left[2\left(\frac{\sigma^{12}}{r_{ij}^{13}}\right) - \left(\frac{\sigma^{6}}{r_{ij}^{7}}\right)\right] \mathbf{u}_{ij}, \tag{7.3}$$

where $\mathbf{u}_{ij}$ is the unit vector pointing from atom i to atom j. In this thesis, we will only study the LJ-potential, so we simplify the notation by choosing $\mathbf{F}_{LJ} = \mathbf{F}$ and $U_{LJ} = U$. The form of the potential is shown in figure 7.2. With this potential, we can in principle calculate the

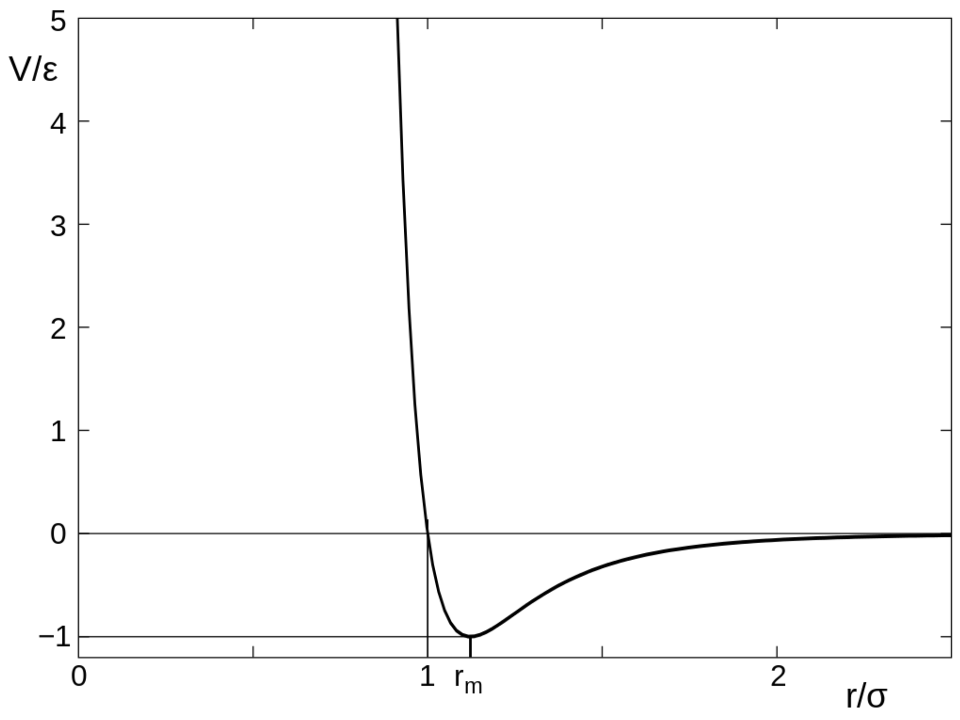

Figure 7.2: The Lennard-Jones potential as a function of relative distance r_{ij} between two atoms. (Image from http://en.wikipedia.org/wiki/File:12-6-Lennard-Jones-Potential.svg, accessed 18 March, 2014.)

forces between atoms that are 10 meters apart. We already know the answer though, it should be very, very close to zero. Atoms being far away from each other do not interact. This distance is actually not very large. If we insert $r_{ij} = 3.0\sigma$ into equation (7.3), and choose units so that $\sigma = \epsilon = 1.0$, we get $F(r = 3.0) \approx 0.01$ (compared to the maximum attraction near the potential well $F(r = 1.24) \approx 2.4$, more than 100 times larger). The force decreases slowly towards zero for larger displacements. We can exploit this property and only compute forces between atoms displaced by a distance smaller than some cut-off radius.

7.2.2 Cut-off radius

In principle, we have to sum over all pairs in the system which for N atoms is $O(N^2)$. This calculation can be reduced to $O(N)$ by realizing that the gradient of the potential (hence the force) is nearly zero at $r \approx 2.5\sigma$. We now introduce the cut-off radius r_{cut}, which we choose to be $r_{\text{cut}} = 2.5\sigma$. The force between a pair of atoms $F(r_{ij})$ is then written as

$$\mathbf{F}(\mathbf{r}_{ij}) = \begin{cases} -24\epsilon \left[2\left(\frac{\sigma^{12}}{r_{ij}^{13}}\right) - \left(\frac{\sigma^6}{r_{ij}^7}\right)\right] \mathbf{u}_{ij} & \text{if } r_{ij} \leq r_{\text{cut}} \\ 0 & \text{if } r_{ij} > r_{\text{cut}}. \end{cases} \tag{7.4}$$

This means that we don't have to compute the forces between atoms that are more than r_{cut} apart. So *locally*, within this radius r_{cut}, the number of pairs we need to sum over is proportional to ρ_n^2, but if we double the system size, most of the extra atoms do not feel the forces from the atoms already in the system since they are separated by more than r_{cut}. So the calculation of forces is $O(N)$.

We need to be careful here, because the potential energy is calculated with the same rules at the forces. So an atom just outside r_{cut} will have zero potential energy whereas an atom just *inside* will have a non-zero energy. This can lead to erroneous fluctuations in the energy. We will fix that by shifting the potential energy so that $U(r_{\text{cut}})$ is zero. The potential energy is then

$$U(r_{ij}) = \begin{cases} U_{LJ}(r_{ij}) - U(r_{\text{cut}}) & \leq r_{\text{cut}} \\ 0 & \text{if } r_{ij} > r_{\text{cut}}, \end{cases} \tag{7.5}$$

where $U_{LJ}(r_{ij})$ is the same as in equation 7.2.

7.3 Time integration

The idea of doing a Molecular Dynamics simulation is to start from some initial state $(\mathbf{r}_0, \mathbf{v}_0)$ and compute the time evolution of atoms following Newton's equations with the defined forces. We will use this time evolution to sample statistics from states in the phase space, assuming that the time evolution will guide the system through the phase space with probabilities according to the statistical ensemble. This was the assumption of ergodicity. In a standard Molecular Dynamics simulation, we want to sample states from the microcanonical ensemble (NVE) where the energy, volume and number of atoms are conserved. Our choice of time integrator should therefore conserve energy as good as possible. A good choice is the Velocity Verlet algorithm, which is known to be both fast and conserve energy quite well[11].

The Velocity Verlet algorithm requires the knowledge of the positions, velocities and instantaneous accelerations of the atoms. The latter is found through the force on an atom $\mathbf{a}_i = \mathbf{F}_i/m_i$, so it requires the storage of $9N$ variables in a system with N atoms. We have derived the Velocity Verlet algorithm using the Liouville formulation of time evolution in appendix B. The integration steps are

$$\mathbf{v}(t+\Delta t/2) = \mathbf{v}(t) + \frac{\mathbf{F}(t)}{m}\frac{\Delta t}{2} \tag{7.6}$$

$$\mathbf{r}(t+\Delta t) = \mathbf{r}(t) + \mathbf{v}(t+\Delta t/2)\Delta t \tag{7.7}$$

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t+\Delta t/2) + \frac{\mathbf{F}(t+\Delta t)}{m}\frac{\Delta t}{2}, \tag{7.8}$$

which is done each timestep for every atom in the system. We call the first step a *half kick*, because it integrates the velocity with forward Euler with half a timestep. The position is then also integrated with forward Euler with the full timestep Δt before we do another half kick.

7.4 Thermostat

When N atoms move around, their temperature are defined as in DSMC, through the equipartition theorem

$$T = \frac{2E_k}{3Nk_B}, \tag{7.9}$$

where E_k is the instantaneous kinetic energy. But what if we would like another temperature? If we want to increase the temperature, we see that increasing the kinetic energy would do the job. This is in fact a decent way to control the temperature. We say that when we push the system towards a given temperature, we apply a thermostat. A much used thermostat is the Berendsen thermostat. Assume that the system has a temperature T, and we would like to push the system towards a new temperature T_0. We can think of this as a heat bath in contact with the system. The Berendsen thermostat is defined in a way so that

$$\frac{\mathrm{d}T}{\mathrm{d}t} = \frac{T_0 - T}{\tau}, \tag{7.10}$$

for some time constant τ which essentially determines how fast the system reaches the desired temperature. A simple algorithm obeying the above convergence rate is by multiplying the velocities of all atoms by a factor

$$\gamma = \sqrt{1 + \frac{\Delta t}{\tau}\left(\frac{T_0}{T} - 1\right)}, \tag{7.11}$$

where Δt is the timestep. We see that if the temperature T is *higher* than the temperature of the heat bath T_0, γ becomes smaller than one, reducing the velocities of all the atoms. If the temperature is lower than desired, γ is larger than one. While this is a efficient way to gain a desired temperature, it does affect the system in a non-physical way.

Without a thermostat, the states of the system live in the microcanonical ensemble where the number of atoms, volume and energy is constant. By applying a thermostat, the energy is clearly not conserved and it is tempting to say that we instead have a constant temperature which is the canonical ensemble. The Berendsen thermostat might produce states not only from this ensemble, so it should be used with care.

Chapter 8

Implementation

In this chapter we will go through the implementation of the MD code in the same way we did with DSMC in chapter 5. We discuss the stages from the beginning of the simulation and introduce implementation techniques when they appear in the process. We here assume that we want to simulate N atoms in a box of physical size $L_x \times L_y \times L_z$ using the Lennard-Jones potential. Our choice of units is described in appendix E.

We start with section 8.1 by briefly explaining how the code is parallelized, since this affects many of the implementation choices we have made. The steps in the actual program can then be explained. We start system initialization in section 8.2 where we describe the FCC lattice and how we choose the initial velocities of the atoms. The timestep consists of several stages that are explained in section 8.3. In section 8.4 we introduce a model of a solid allowing us to study flow in complicated geometries.

8.1 Parallelization

The parallelization technique is very similar to what we did in DSMC in section 5.4. The spatial domain is divided into $P_x \times P_y \times P_z$ smaller domains (P_i being the number of processors in the ith dimension). Each such domain is then of size $l_x \times l_y \times l_z$ where $l_i = L_i/P_i$. The processor arrangement and labeling is done as shown in figure 8.1, where a processor with coordinates (p_x, p_y, p_z) controls atoms with coordinates in the range

$$\begin{aligned} x &\in [p_x l_x, (p_x+1)l_x\rangle \\ y &\in [p_y l_y, (p_y+1)l_y\rangle \\ z &\in [p_z l_z, (p_z+1)l_z\rangle. \end{aligned} \tag{8.1}$$

We represent the coordinates of an atom on processor (p_x, p_y, p_z) in the local coordinate system where the CPU's origin $\mathbf{p}_{\text{origin}}$ defines the coordinate system. An atom with global coordinates $\mathbf{r}$ will then have local coordinates $\mathbf{r}'$

$$\mathbf{r}' = \mathbf{r} - \mathbf{p}_{\text{origin}}. \tag{8.2}$$

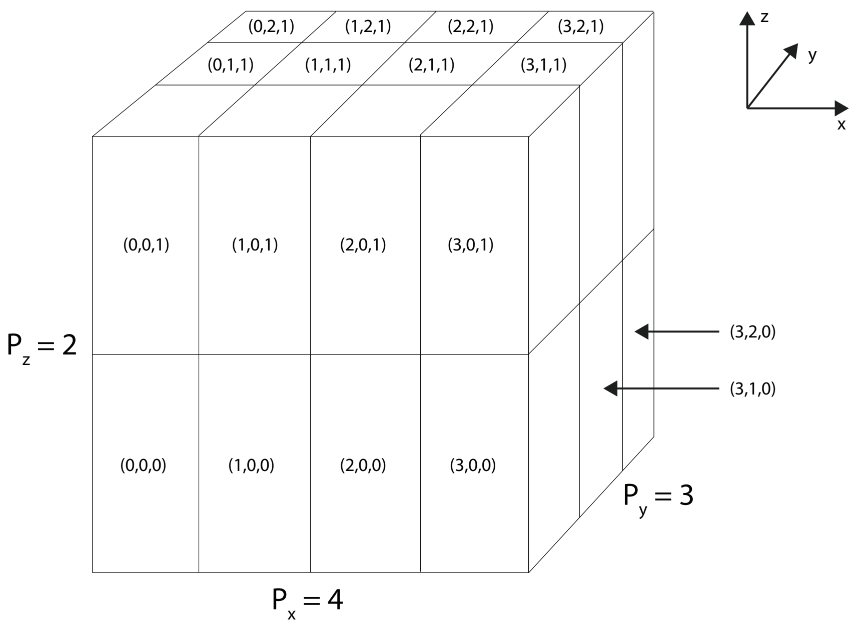

Figure 8.1: Processor labeling in a 3-dimensional grid. Each processor is uniquely identified through its coordinate (p_x, p_y, p_z).

We can then easily detect if an atom has moved outside its processor's domain by checking if any of the local coordinate components are outside the range $[0, l_i)$ for dimension i. If an atom has moved to another processor, it has moved to one of the 26 neighboring processors. We will use the same 3-step process as described in subsection 5.4.2 which is best illustrated with the 2-dimensional example in figure 8.2. We only need to know about the 6 nearest neighbors on each processor. We notice a neat detail here, periodic boundary conditions are automatically taken care of. If we again look at figure 8.1, we have 4 processors in the x-direction. The processor to the *right* of $(3, 0, 1)$ is $(0, 0, 1)$ if we use periodic boundary conditions. But since we use the local coordinates on each processor, when an atom moves to another processor, its coordinates must be *shifted* so its has correct local coordinates on the new processor. Each processor has one *shift vector* per neighbor, containing the transformation it needs to apply on the atom's coordinates before it moved. If the atom moved through the system boundary (periodic boundary conditions), the shift vector must of course reflect this.

8.2 System initialization

We are now ready to initialize the MD system. We assume that we will simulate an system with volume

$$V = L_x L_y L_z, \tag{8.3}$$

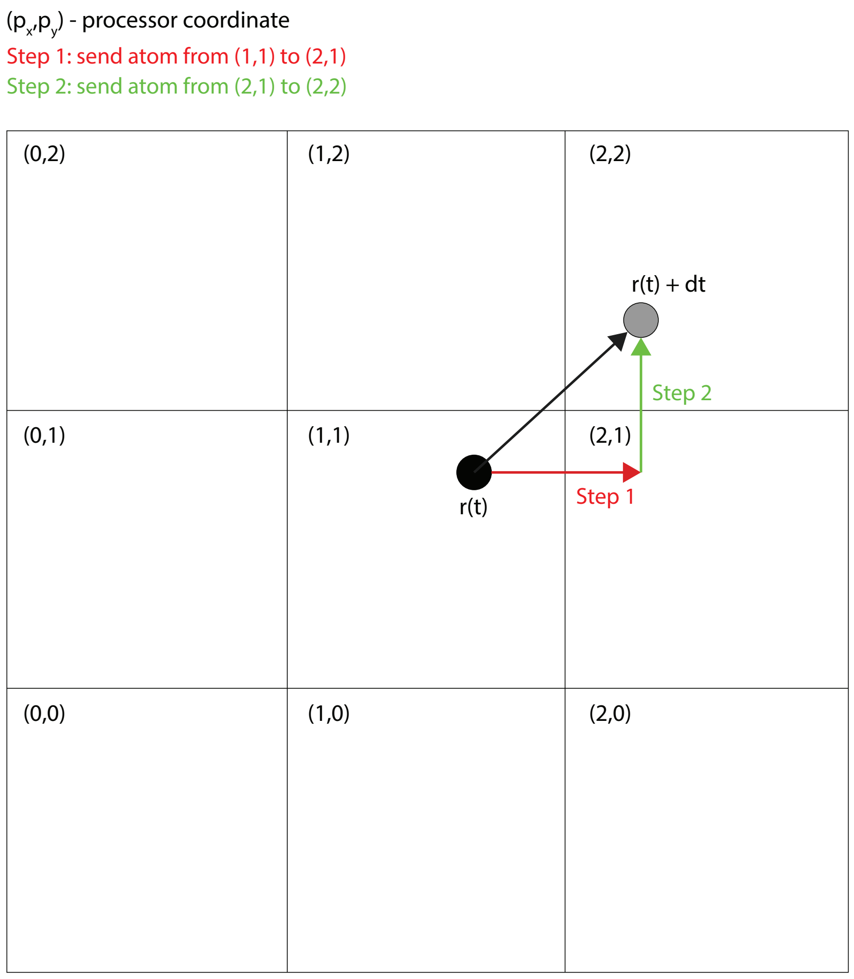
(p_x,p_y) - processor coordinate
Step 1: send atom from (1,1) to (2,1)
Step 2: send atom from (2,1) to (2,2)
(0,2)
(1,2)
(2,2)
r(t) + dt
Step 2
(0,1)
(1,1)
(2,1)
r(t)
Step 1
(0,0)
(1,0)
(2,0)

Figure 8.2: The middle node (1,1) has 8 neighbors it needs to communicate with. Each node only needs to communicate with its nearest neighbors (4 in two dimensions, 6 in three dimensions), because the nearest neighbors can work as intermediate information carriers. An atom moving from processor (1,1) to (2,2) will in step 1 be sent to (2,1), then in step 2 be sent to (2,2).

where L_i is the system size in the ith dimension. The system contains N argon atoms and is performed on

$$P = P_x P_y P_z \tag{8.4}$$

processors, P_i being the number of processors in the ith dimension. Each processor controls a volume

$$V_p = l_x l_y l_z, \tag{8.5}$$

where $l_i = L_i/P_i$ is the node length. The origin of processor (p_x, p_y, p_z) is given as

$$\mathbf{p}_{\text{origin}}(p_x, p_y, p_z) = p_x l_x \hat{i} + p_y l_y \hat{j} + p_z l_z \hat{k}. \tag{8.6}$$

The initialization process consists of creating the atoms, place them at some position and give them velocities according to the Maxwell-Boltzmann distribution (equation (3.46)). We will initially place the atoms on an face-centered cubic lattice (FCC lattice).

8.2.1 FCC lattice

The FCC lattice is a lattice structure where we place atoms on all 8 corners of a cube in addition to all the faces, hence the name. In figure 8.3 we see how the atoms are placed on the FCC lattice. An FCC lattice can be constructed from a unit cell consisting of four atoms with local

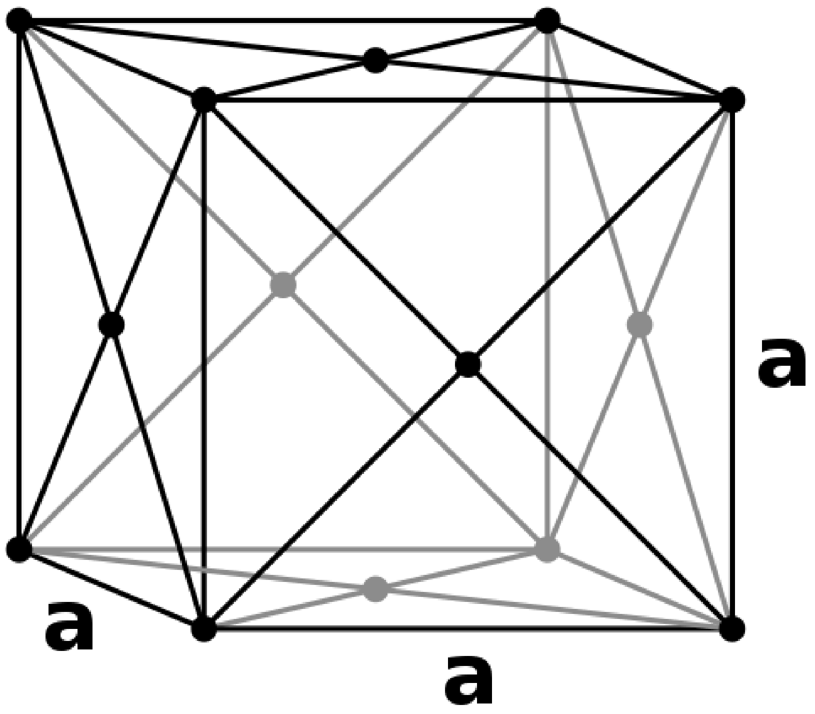

Figure 8.3: The face-centered cubic lattice. There are atoms on all eight corners of a cube, in addition to the six faces. The length a is called the lattice constant. (Image from http://en.wikipedia.org/wiki/File:Lattice_face_centered_cubic.svg, accessed 19 March, 2014.)

positions (relative to the origin of the unit cell)

$$\mathbf{r}_1 = 0\hat{\mathbf{i}} + 0\hat{\mathbf{j}} + 0\hat{\mathbf{k}} \tag{8.7}$$

$$\mathbf{r}_2 = \frac{a}{2}\hat{\mathbf{i}} + \frac{a}{2}\hat{\mathbf{j}} + 0\hat{\mathbf{k}} \tag{8.8}$$

$$\mathbf{r}_3 = 0\hat{\mathbf{i}} + \frac{a}{2}\hat{\mathbf{j}} + \frac{a}{2}\hat{\mathbf{k}} \tag{8.9}$$

$$\mathbf{r}_4 = \frac{a}{2}\hat{\mathbf{i}} + 0\hat{\mathbf{j}} + \frac{a}{2}\hat{\mathbf{k}}, \tag{8.10}$$

where a is the lattice constant, the length of the unit cell. By adding several such unit cells, each with origin determined by the unit cell coordinate (m, n, l)

$$\mathbf{r}_{\text{unit cell}} = ma\hat{\mathbf{i}} + na\hat{\mathbf{j}} + la\hat{\mathbf{k}}, \tag{8.11}$$

we can create a system of arbitrary size. Each atom is assigned a random velocity according to the Maxwell-Boltzmann distribution. The total number of atoms in the system is based on how many unit cells we create. If we label the number of unit cells *per processor* in the ith dimension $N_c^{(i)}$, the total number of unit cells is

$$N_c = N_c^{(x)} N_c^{(y)} N_c^{(z)} P, \tag{8.12}$$

which gives the total number of atoms (each unit cell contains 4 atoms)

$$N = 4N_c. \tag{8.13}$$

The system size is then

$$L_x = aN_c^{(x)} P_x \tag{8.14}$$
$$L_y = aN_c^{(y)} P_y \tag{8.15}$$
$$L_z = aN_c^{(z)} P_z, \tag{8.16}$$

yielding a total volume $V = a^3 N_c$. In listing 8.1, we have shown how this is implemented in C++.

```cpp
void create_fcc_lattice() {
    double xCell[4] = {0, 0.5, 0.5, 0};
    double yCell[4] = {0, 0.5, 0, 0.5};
    double zCell[4] = {0, 0, 0.5, 0.5};

    int index = 0;
    double velocity_standard_deviation = sqrt(boltzmann_constant*
        temperature/mass);
    for(int x = 0; x < unit_cells_per_cpu_x; x++) {
        for(int y = 0; y < unit_cells_per_cpu_y; y++) {
            for(int z = 0; z < unit_cells_per_cpu_z; z++) {
                // Loop over the 4 atoms in this unit cell
                for(int k = 0; k < 4; k++) {
                    positions.at(index).x = (x+xCell[k]) * FCC_a;
                    positions.at(index).y = (y+yCell[k]) * FCC_a;
                    positions.at(index).z = (z+zCell[k]) * FCC_a;
                    velocities.at(index).x = rnd.nextGauss()*
                        velocity_standard_deviation;
                    velocities.at(index).y = rnd.nextGauss()*
                        velocity_standard_deviation;
                    velocities.at(index).z = rnd.nextGauss()*
                        velocity_standard_deviation;
                    index++; // Increase atom counter
                }
            }
        }
    }
}
```

Listing 8.1: Code example showing how to create an FCC lattice on one of the processors.

This is the whole initialization process. The atoms are where they should be, they have a statistically correct velocity and we are ready to integrate forward in time.

8.3 Timestep

We are using the Velocity Verlet algorithm as we discussed in section 7.3 and derived in appendix B. The timestep calculation starts by what we called a half kick, where the velocities are integrated half a timestep with forward Euler. But to be able to do so, we need to have calculated the forces. This part is the most technical in the whole algorithm, because of two things. We need to find a way to implement the cut-off radius so that we don't loop over too many atom pairs, and each processor needs to have information about the atoms on its neighboring atoms (there are forces between atoms living on different CPUs). We will first discuss how duplicates of atoms, so-called ghost atoms, are copied between nodes before we explain how we use cell lists to compute forces between nearby atoms only.

8.3.1 Ghost atoms

Assume now that we are going to calculate the forces between the atoms on some node with coordinates (p_x, p_y, p_z). The atoms near the boundary will of course feel the presence of the atoms on the next processor (if they are sufficiently near). The node will then, from each of its 26 neighbors, get a copy of the atoms that are less than r_{cut} from the node boundary. In figure 8.4, this is illustrated in a two-dimensional example.

Since we don't calculate forces between atoms displaced by more than the cut-off radius, we actually compute all the forces we want. This process is done for every processor, every timestep, so that all nodes have all the information they need to compute the forces (and potential energy).

8.3.2 Cell lists

We will now sort the atoms on a node into cells, similar to the collision cells in DSMC (see section 5.3). Each processor has atoms with coordinates in the range $[0, l_i)$, l_i being the node length in the ith dimension, and ghost atoms with coordinates $(-r_{\text{cut}}, 0) \cup [l_i, l_i + r_{\text{cut}})$. We choose the cells to be about the same size as the cut-off radius, which gives the number of such cells N_f (force cells) in the ith dimension

$$N_f = \lceil l_i / r_{\text{cut}} \rceil + 2, \tag{8.17}$$

where we have added two extra *ghost cells* containing the ghost atoms, $\lceil a \rceil$ is the ceiling function of a, that is, the smallest integer number not larger than a. The reason we use the ceiling function is so the length of the node exactly matches the combined length of the cells (minus the two extra ghost cells). The size of the cells is then at least r_{cut} so that the atoms within a cell will not feel the presence of atoms from any other cells than the 26 nearest neighbors. In figure 8.5

we have illustrated the idea in a two-dimensional system. We see the total spatial domain of a processor with the copied ghost atoms (in the gray area), divided into cells of size r_{cut}. The yellow cell will compute forces between all of its atoms and the atoms in the green cells. This is done for every inner cell (the ghost atoms are computed on another processor).

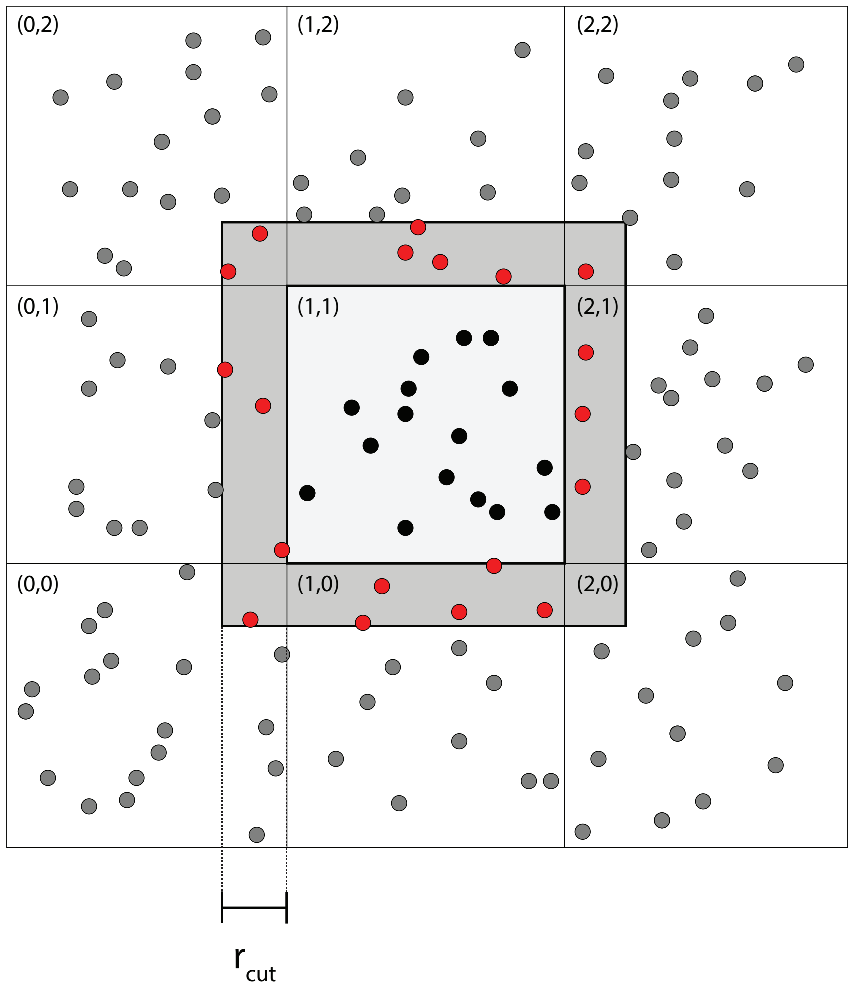

Figure 8.4: The middle processor with coordinates (1,1) in a two-dimensional system receives a copy of all the atoms less than r_{cut} from the boundary (ghost atoms, marked red) from the neighboring eight processors. The gray area is the region with the ghost atoms. The black dots are the atoms on node $(1, 1)$, gray dots are atoms that do not contribute to the forces of the black atoms.

8.3.3 Calculation of forces

The atoms are now sorted into cells. We then loop over all cells, and for each cell, loop over the 26 neighboring cells. With each cell pair, loop over all pairs of atoms and calculate forces between them if their relative distance is smaller than r_{cut}. In the inner scope of the loop, we now have two atoms i and j with positions $\mathbf{r}_i$ and $\mathbf{r}_j$. Given their relative distance $\mathbf{r}_{ij} = (\Delta x, \Delta y, \Delta z)$, we are ready to compute the force between them. The Lennard-Jones force is given as (for $r_{ij} \leq r_{\text{cut}}$)

$$\mathbf{F}(\mathbf{r}_{ij}) = -24\epsilon \left[2\left(\frac{\sigma^{12}}{r_{ij}^{13}}\right) - \left(\frac{\sigma^6}{r_{ij}^7}\right) \right] \mathbf{u}_{ij}, \tag{8.18}$$

and if we choose the so-called MD units ($\sigma = 1.0, \epsilon = 1.0$, see appendix E), the x-component of the force is given as

$$F_x(r_{ij}) = -24 \left[\left(\frac{2}{r_{ij}^{12}}\right) - \left(\frac{1}{r_{ij}^6}\right) \right] \frac{\Delta x}{r_{ij}^2}, \tag{8.19}$$

where Δx is the $x-$component of $\mathbf{r}_{ij}$. Notice that we have factored out $1/r_{ij}^2$. This is arithmetical convenient for the implementation, because, as we see in equation (8.19), we need r_{ij}^{-6} and r_{ij}^{-12}, which both are easily calculated from r_{ij}^{-2} as powers.

First, we skip atoms displaced by a distance larger than r_{cut} and compute the square of the relative distance

$$r_{ij}^2 = |\mathbf{r}_i - \mathbf{r}_j|^2 . \tag{8.20}$$

Its inverse is gives us the higher powers

$$r_{ij}^{-2} = \frac{1}{r_{ij}^2} \tag{8.21}$$

$$r_{ij}^{-6} = \left(r_{ij}^{-2}\right)^3 \tag{8.22}$$

$$r_{ij}^{-12} = \left(r_{ij}^{-6}\right)^2 . \tag{8.23}$$

In listing 8.2 we have shown the code that calculates the Lennard-Jones force. We see that Newton's third law is implemented by skipping a pair of atoms if `atom_index_0<atom_index_1`, avoiding calculating that pair twice.

```
void apply_lennard_jones() {
    // Loop through all local cells (not including ghosts)
    for (int cell_index_x=1; cell_index_x<=num_cells.x; cell_index_x++) {
    for (int cell_index_y=1; cell_index_y<=num_cells.y; cell_index_y++) {
    for (int cell_index_z=1; cell_index_z<=num_cells.z; cell_index_z++) {
    // Index of this cell
    cell_index = cell_index_from_ijk(cell_index_x, cell_index_y,
        cell_index_z);
    // Loop through all neighbors (including ghosts) of this cell.
    for (int i=cell_index_x-1; i<=cell_index_x+1; i++) {
    for (int j=cell_index_y-1; j<=cell_index_y+1; j++) {
    for (int k=cell_index_z-1; k<=cell_index_z+1; k++) {
    // Index of neighbor cell
    cell_index_2 = cell_index_from_ijk(i,j,k);
    // Head is pointing to the first atom index in a cell
    int atom_index_0 = head_atoms[cell_index];

    while (atom_index_0 != EMPTY) { // The last atom in a cell points at
        the EMPTY value
    int atom_index_1 = head_atoms[cell_index_2];
    while (atom_index_1 != EMPTY) {
        if(atom_index_0 < atom_index_1) { // Newton's 3rd law
            double dx = positions.at(atom_index_0).x-positions.at(
                atom_index_1).x;
            double dy = positions.at(atom_index_0).y-positions.at(
                atom_index_1).y;
            double dz = positions.at(atom_index_0).z-positions.at(
                atom_index_1).z;

            double dr2 = dx*dx + dy*dy + dz*dz;

            if (dr2<r_cut_squared) {
                double dr2_inverse = 1.0/dr2;
                double dr6_inverse = dr2_inverse*dr2_inverse*dr2_inverse;
                double dr12_inverse = dr6_inverse*dr6_inverse;
                double force = 24*(2*dr12_inverse-dr6_inverse)*dr2_inverse;

                accelerations.at(atom_index_0).x += force*mass_inverse*dx;
                accelerations.at(atom_index_0).y += force*mass_inverse*dy;
                accelerations.at(atom_index_0).z += force*mass_inverse*dz;

                accelerations.at(atom_index_1).x -= force*mass_inverse*dx;
                accelerations.at(atom_index_1).y -= force*mass_inverse*dy;
                accelerations.at(atom_index_1).z -= force*mass_inverse*dz;
            }
        }
        atom_index_1 = linked_list_of_atoms[atom_index_1]; // Next atom
    }
    atom_index_0 = linked_list_of_atoms[atom_index_0]; // Next atom
    }}}}}}
}
```

Listing 8.2: Implementation of the Lennard-Jones force. The code loops over all cells and their neighbors computing the forces between all the atoms in neighboring cells.

8.3.4 The timestep

Now that we can calculate the forces, we can summarize the whole timestep. Assuming that the forces are already calculated, we start by the half kick, integrating the velocity half a timestep. Then we move the atoms by integrating the positions with forward Euler a whole timestep Δt.

Since some atoms now may have changed which processor they live on, we have to move atoms with `MPI`. Once this is done, every processor is ready to calculate the forces, but in order to do so, we need the ghost atoms on every node. It is time to reset the accelerations from the last timestep, before we apply the constant force we added to induce flow in the system. After this is done, we can calculate the forces and do the final half kick, which is the last stage of the timestep. In listing 8.3 we show how the `step` function is implemented, simply calling functions performing all the stages we have now explained.

```
void step() {
    half_kick();
    move();
    mpi_move();
    mpi_copy();
    reset_accelerations();
    apply_constant_force();
    apply_lennard_jones();
    half_kick();
}
```

Listing 8.3: Code showing the stages during a timestep.

8.4 Complex geometries

As we did with DSMC, we want to study flow in arbitrary geometries. To be able to do this, we need to create a model that satisfies some properties we already have in DSMC. The flowing fluid needs to

- be confined in a subset of the total volume,
- have realistic interactions with the solid wall, and
- have energy drained by the solid.

The first requirement is not completely strict, but most of the flowing fluid should be in the free volume. The reason why we need to drain the energy is that in order to induce fluid flow, we apply a constant force that increases the total energy of the system. We want to reach an equilibrium where the average rate of drained energy exactly matches the energy added by the applied force. We assume that the geometry is described as a boolean function $G : \mathbb{R}^3 \to \{1, 0\}$ that fully determines whether a point in space is part of the solid or the free volume. A convenient representation is the voxelization described in section 5.1.1.

8.4.1 A naive approach

To illustrate the basic idea, we will discuss the simplest model satisfying only the first two requirements. Given a molecular dynamics state, we can loop through the positions $\mathbf{r}_i$ of each atom i, and mark the atoms as *frozen* if $G(\mathbf{r}_i) = 1$ (which means that this point is part of the solid). Atoms marked as frozen will not move at all, but all forces are calculated normally. One way of interpreting the non-moving frozen atoms is that they have infinite mass. The total energy is still conserved in the system. An implementation is illustrated in listing 8.4.

```
bool point_is_a_solid(Vector3 &position) {
  int voxel_index_i = position.x / system_length.x * num_voxels.x;
  int voxel_index_j = position.y / system_length.y * num_voxels.y;
  int voxel_index_k = position.z / system_length.z * num_voxels.z;

  // The world matrix is a binary matrix
  return world_matrix[voxel_index_i, voxel_index_j, voxel_index_k];
}

void mark_frozen_atoms() {
  for(int i=0; i<number_of_atoms; i++) {
    Vector3 &position = positions.at(i);
    if(point_is_a_solid(position)) {
      atom_types.at(i) = FROZEN;
    }
  }
}

void move() {
  for(int i=0; i<number_of_atoms; i++) {
    if(atom_types.at(i) != FROZEN) {
      // Only move non-frozen atoms
      positions.at(i).x += velocities.at(i).x*dt;
      positions.at(i).y += velocities.at(i).y*dt;
      positions.at(i).z += velocities.at(i).z*dt;
    }
  }
}
```

Listing 8.4: Example code showing how to mark atoms within a solid.

If the atoms forming the solid are dense enough, very few atoms will be inside the wall, so the first requirement is satisfied. The interaction between the solid and the flowing fluid is as realistic as the potential is, so the only requirement not satisfied is the drainage of the energy.

8.4.2 A simple model of a solid

We can improve the solid model by adding a harmonic oscillator potential to all the frozen atoms. Instead of freezing them completely, we allow them to vibrate around their equilibrium position $\mathbf{q}$ which is the initial position of the simulation. The force on atom i in the solid is then

$$F^{(s)}(\mathbf{r}_i) = -k(\mathbf{r}_i - \mathbf{q}_i) - \sum_{j\neq i} 24\epsilon \left[2\left(\frac{\sigma^{12}}{r_{ij}^{13}}\right) - \left(\frac{\sigma^6}{r_{ij}^7}\right)\right] \mathbf{u}_{ij}, \tag{8.24}$$

where k is the strength of the oscillator, $\mathbf{q}_i$ is the equilibrium position for atom i and the last part is the Lennard-Jones potential described in section 7.2. The energy is of course still conserved with this model, but we can now apply a thermostat on the solid atoms making them act as a large reservoir trying to keep the temperature at some wall temperature T_w. With this model, we can study flow in any geometry with a behavior near that of the DSMC model. An implementation is shown in listing 8.5.

```
void apply_constant_force() {
    for(n=0;n<number_of_atoms;n++) {
        if(atom_type.at(n) != FROZEN) velocities.at(n).at(flow_direction)
           += acceleration*dt;
    }
}

void apply_harmonic_oscillator() {
    double spring_constant = 1000.0;
    for(n=0; n<number_of_atoms; n++) {
        if(atom_type.at(n) == FROZEN) {
            double dx = positions.at(n).x - initial_positions.at(n).x;
            double dy = positions.at(n).y - initial_positions.at(n).y;
            double dz = positions.at(n).z - initial_positions.at(n).z;
            accelerations.at(n).x += -spring_constant*dx / mass;
            accelerations.at(n).y += -spring_constant*dy / mass;
            accelerations.at(n).z += -spring_constant*dz / mass;
        }
    }
}

void step() {
    half_kick();
    move();
    mpi_move();
    mpi_copy();
    reset_accelerations();
    apply_constant_force();
    apply_lennard_jones();
    apply_harmonic_oscillator();
    half_kick();
}
```

Listing 8.5: Implementation of the harmonic oscillator model of a solid.

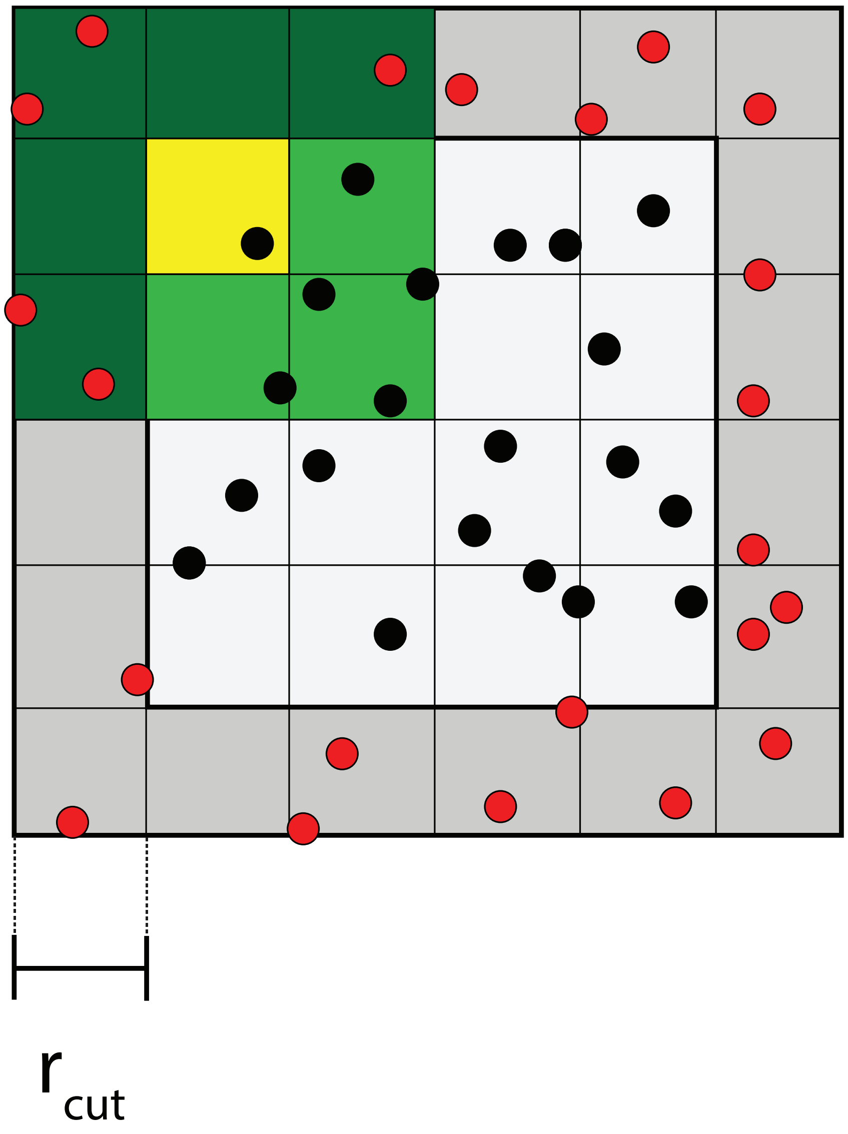

Figure 8.5: The spatial domain for a processor with the added ghost atoms in the gray area. The domain is divided into cells of size r_{cut} so that the yellow cell only needs to compute forces between atoms in the cell and the neighboring 8 green cells. The processor will only process the cells (the ghost atoms are computed on another processor).

Chapter 9

Results

In this chapter we discuss the results of our MD simulations. Here we start by a scaling performance benchmark in section 9.1 before we in section 9.2 move on to a validation of the Knudsen correction we confirmed with the DSMC model. Most of the concepts are equivalent as in DSMC in section 6.2, so the reader is is assumed to have read that section.

9.1 Parallelization performance

The parallelization scheme we have used in MD is very similar to the one we used in DSMC. In section 6.2, we measured both the strong and the weak scaling efficiency (η_s and η_w) which measure two different ways of scaling the program. We will not repeat the discussion about these except present the result for both scaling for the MD program.

9.1.1 Strong scaling

The strong scaling efficiency η_s tells us how well the program scales if we keep the system size constant while increasing the number of processors. The strong scaling efficiency was defined as

$$\eta_s = \frac{t_1}{N t_N}, \tag{9.1}$$

where t_1 is the total run time using one processor and t_N is the total run time using N processors. In this benchmark, we chose a system consisting of $48 \times 48 \times 48 = 110592$ unit cells which gives a total of 442368 atoms. When we increase the number of processors, these 48 unit cells per dimension are distributed on the processors. The timestep was $\Delta t = 0.02$ and the initial temperature was approximately T_0 =100 K so that the final gas temperature was around T =60 K. This fall in temperature is explained by the fact that the FCC lattice is the potential minimum, so with a non-zero temperature, some of the initial kinetic energy will be converted to potential energy. We run the program with the number of processors in the range 1 to 512.

In table 9.1 we have presented the results for this benchmark which is summarized in figure 9.1. We see that when going from one to two processors, we get a more than ideal speedup with $\eta_s(N_{\text{CPU}} = 2) = 1.17$ which can be explained by the CPU cache. When a processor compute with a value stored at some memory address, it will first look in the three levels of cache to see if the value of the memory address is cached there. Cached values are much faster available for computation than those only in the memory. When going from one processor to two, a larger part of the positions array (which is used in the force calculation) can be cached, hence a more than double speedup is obtainable. When using a larger number of processors, the MPI communication time starts increasing so that the total time increases per CPU.

N_{CPU}	$N_{\text{atoms}}/N_{\text{CPU}}$	t_N	η_s
1	442368	29196 s	1.00
2	221184	12471 s	1.17
4	110592	6435 s	1.13
8	55296	3469 s	1.05
16	27648	1926 s	0.95
32	13824	1021 s	0.89
64	6912	715 s	0.64
128	3456	375 s	0.61
256	1728	220 s	0.52
512	864	150 s	0.38

Table 9.1: Benchmark results showing the strong scaling efficiency η_s for the MD program. We see that when going from one to two processors, we get a more than ideal speedup with $\eta_s(N_{\text{CPU}} = 2) = 1.17$ which can be explained by the CPU cache. When a processor compute with a value stored at some memory address, it will first look in the three levels of cache to see if the value of the memory address is cached there. Cached values are much faster available for computation than those only in the memory. When going from one processor to two, a larger part of the positions array (which is used in the force calculation) can be cached, hence a more than double speedup is obtainable. When using a larger number of processors, the MPI communication time starts increasing so that the total time increases per CPU.

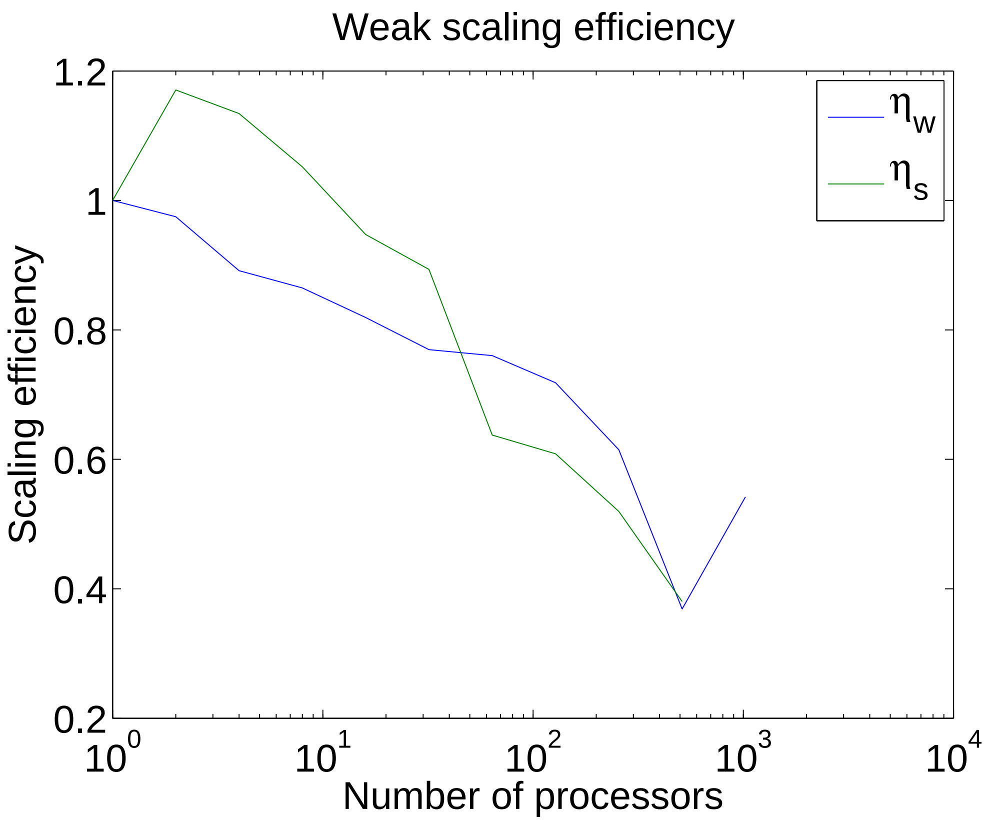

Figure 9.1: Benchmark results showing the strong and the weak scaling efficiency, η_s and η_w, for the MD program. We see that when going from one to two processors, we get a more than ideal speedup with $\eta_s(N_{\mathrm{CPU}} = 2) = 1.17$ which can be explained by the CPU cache. When a processor compute with a value stored at some memory address, it will first look in the three levels of cache to see if the value of the memory address is cached there. Cached values are much faster available for computation than those only in the memory. When going from one processor to two, a larger part of the positions array (which is used in the force calculation) can be cached, hence a more than double speedup is obtainable. When using a larger number of processors, the MPI communication time starts increasing so that the total time increases per CPU. The weak scaling shows similar scaling in the whole range of processors. In order to reduce the statistical noise, several benchmarks should be run and averaged.

9.1.2 Weak scaling

If we increase the number of processors, but keep the nubmer of atoms per processor constant, we can use the weak scaling efficiency η_w to see how the program scales in this case. The weak scaling efficiency is defined as

$$\eta_w = \frac{t_1}{t_N}, \tag{9.2}$$

where again t_1 is the total run time using one processor and t_N is the run time using N processors. In this benchmark, we chose $10 \times 10 \times 10 = 1000$ unit cells per processor yielding a total of 4000 atoms per CPU. The timestep here as well is $\Delta t = 0.02$ with the same initial temperature as in the strong scaling so that the final temperature is approximately T =60 K. In table 9.2 and figure 9.1, we have presented the results for the weak scaling.

N_{CPU}	N_{atoms}	t_N	η_w
1	4000	1246 s	1.00
2	8000	1278 s	0.97
4	16000	1398 s	0.89
8	32000	1441 s	0.86
16	64000	1521 s	0.82
32	128000	1620 s	0.77
64	256000	1639 s	0.76
128	512000	1735 s	0.72
256	1024000	2027 s	0.61
512	2048000	3379 s	0.37

Table 9.2: Benchmark results showing the weak scaling efficiency η_w for the MD program.

9.2 Flow in a cylinder, varying Knudsen number

We have used the MD program to simulate flow in a cylinder with a fixed radius R, just like we did in section 6.3 with DSMC. We will measure the permeability to see how well the Knudsen correction factor (see section 2.13) predicts the permeability for very small pores (here a cylinder) with an atomic model. The cylinder was created with the solid model we described in section 8.4.2. Since the solid now consists of atoms (in DSMC it was just a scalar field defining the surface), we should create the cylinder carefully. We have prepared the system with the following steps

- Heat the system 2000 timesteps, T =300 K
- Thermalize the system 2000 timesteps
- Heat the system 2000 timesteps, T =300 K
- Thermalize the system 2000 timesteps
- Create cylinder (explained below)
- Reduce density (explained below).

By first heating the system, we let the system melt from a solid state to a liquid state. This allows the system to become more random than in the initial lattice. Once we create the cylinder, we apply a harmonic oscillator potential on the atoms in the cylinder so they more or less stay in their initial position. We have here chosen a system consisting of $64 \times 64 \times 64 = 262144$ unit cells which gives a total of 1048576 atoms to begin with. This in turn yields a system with size $L_i = 336.64$ in the ith dimension. By choosing the cylinder radius to be $r = 0.45L_i$ and the flow in the z-direction, we mark all atoms within a distance r from the center (in the xy-plane) as gas atoms, and atoms outside r as solid atoms. But the cut-off radius was chosen to be $r_{\text{cut}} = 2.5\sigma$ (see section 7.2.2), so the gas atoms inside the cylinder will not feel the presence of the atoms outside $r + 2.5\sigma$ directly. To save computation time, we have removed all atoms outside this radius. Such a cylinder is shown in figure 9.2 where the yellow atoms are the solid wall whereas the green atoms are the gas. Once we have the cylinder, we can choose the density yielding a

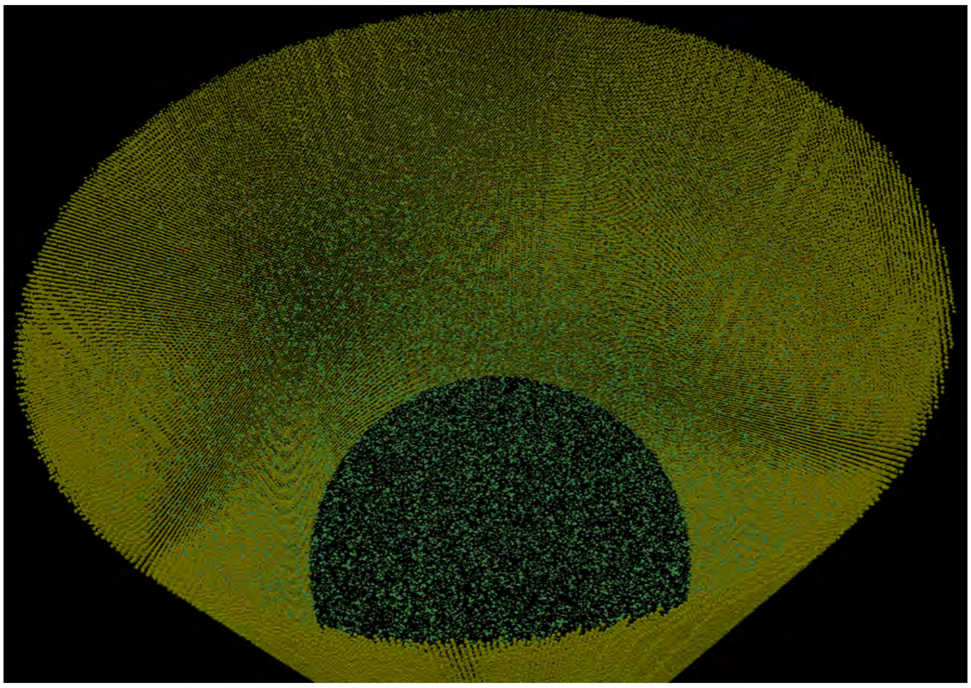

Figure 9.2: The final cylinder we used to simulate gas to validate the Knudsen correction factor for the permeability (see section 2.13). Here the yellow atoms are the solid wall (with a harmonic oscillator potential on each atom, keeping the atoms in place), and the green atoms are the gas. Since we have used a cut-off radius $r_{\text{cut}} = 2.5\sigma$, we have removed the solid atoms outside the radius $r + 2.5\sigma$. This visualization is done with the tool discussed and developed in chapter 11.

desired Knudsen number

$$\rho_n(\text{Kn}) = \frac{1}{\sqrt{2}\pi d^2 \text{Kn} L}, \tag{9.3}$$

where we have used d =3.62 Å as we did in DSMC. The flow is induced in the same way as we did in DSMC (explained in section 4.5), where we can, given a pressure difference ΔP, accelerate the atoms inside the cylinder according to equation (4.41)

$$g = \frac{\Delta P}{\rho_m \Delta x}.$$

Here, Δx is the system length in the flow direction, L_z. We have chosen a pressure difference equal to $0.2P_0$, where $P_0 = \rho_n k_B T$ being the ideal gas pressure. We can then for each Knudsen number induce flow and measure the permeability. The system reached an equilibrium state before we sampled for 500000 timesteps. In figure 9.3, we have plotted the measured permeabilities for different Knudsen numbers with the Knudsen corrected analytical solution

$$k_a = [1 + \alpha(\text{Kn})\text{Kn}] \left[1 + \frac{4\text{Kn}}{1 + \text{Kn}}\right] \frac{r^2}{8}. \tag{9.4}$$

The left figure, using a logarithmic x-axis shows that the permeabilities in the lower range of the Knudsen numbers are predicted very by the theory. The right figure shows that the permeabilities in the whole range, two orders of magnitudes, follows the expression in equation (9.4). For the high Knudsen numbers, we see an increase in the statistical noise which is explained by the fact that a high Knudsen number is obtained by a low density which gives a low number of atoms. In order to get better statistics, we would need to run the simulation for a longer time.

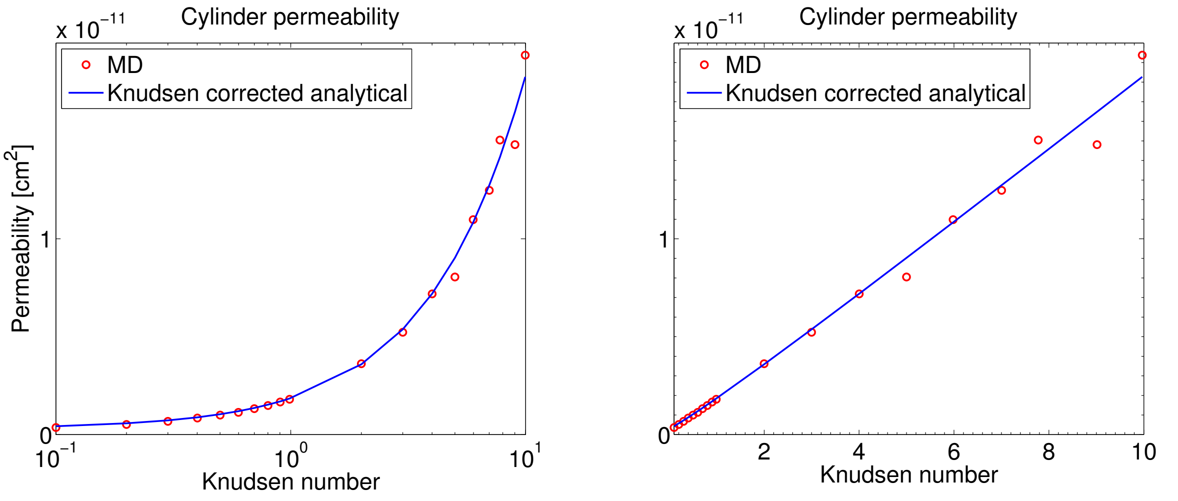

Figure 9.3: Permeabilities for Knudsen numbers in the range 0.1 to 10.0 in a cylinder with radius r =151 Å. The left figure has a logarithmic $x-$axis to emphasize the good prediction in the lower Knudsen range. The right figure shows that the MD code produces results according to the Knudsen corrected permeability for a cylinder in equation (9.4) in the entire range. The increased statistical noise is explained by that for large Knudsen numbers, the number of gas atoms is low.

Part III

Visualization

Whenever scientists perform an experiment in the lab or on a computer, they need to study the information outcome of the experiment. This could be the positions of some particles, the temperature of a gas or maybe the color of a fluid. Once we have this information, we need to represent it in a way that is useful for understanding the results. We often put numbers in a table or plot them as a graph. This is an extremely powerful ways to look at data and we can learn a lot by seeing how two or more variables are dependent of each other. It is convenient to introduce a general term, *visualization*, which is just a visual representation of information (maybe not *any* visual representation. Numbers in a table might be excluded.) One could say that a world map is a representation of the geometrical information describing how the continents are connected. The graph of some data is a relation between two or more variables. The output result from any computer simulation or experiment is information that can be visualized in some way.

There already exists a lot of software that can be used to visualize data in different ways. For particle simulations one can for example use *VMD*[1] or *Ovito*[2]. Both of these are great tools allowing us to visualize the time trajectories of atoms or particles. There are two drawbacks that have motivated the author to write a visualization tool from scratch.

In both VMD and Ovito, the way we navigate with the camera is inspired by other 3d software like 3ds Max, Maya and Blender where the camera is pointing towards a point whereas the mouse controls the position of the camera on the surface of a sphere centered in point. If we want to follow the movement of a single particle, this way of controlling the camera is inconvenient. We would like to control the camera in the way a space ship is controlled in a game, like in a first person shooter game. Here the mouse controls the direction the camera is looking and the camera position is controlled by the keyboard.

In addition, there are performance drawbacks with both programs where we have noticed that medium sized datasets (number of particles $\approx 1 \times 10^6$) lead to a frame rate that makes it very difficult to navigate around in the system. Also, the geometry of the DSMC code cannot be visualized in VMD or Ovito since these are particle visualizers only and the geometry is represented as a scalar field (the voxels from section 5.1). To be able to solve these problems, we have implemented our own visualizer using OpenGL. In this chapter, we first give a brief introduction to OpenGL in section 10.1 explaining its basic ideas. We go through the concepts of vertices, primitives and how colors and textures are linearly interpolated from the values of the vertices. Vertex Buffer Objects are briefly explained in section 10.2 before we go through the sequence of shaders in the rendering pipeline in section 10.3. Note that this is not a complete introduction of OpenGL. Only the basic concepts that are required to understand how we have made the visualization tools are covered.

[1]http://www.ks.uiuc.edu/Research/vmd/
[2]http://www.ovito.org/

Chapter 10

OpenGL

10.1 What is OpenGL?

OpenGL (Open Graphics Library) is an open source graphics library providing an application programming interface (API) to communicate with a graphics processing unit (GPU) in order to get hardware-accelerated *rendering*. Rendering is the process of generating an image from *models* in an *environment*. These models contain information about the geometry and associated textures. A model is typically represented by a set of points, triangles or some other set of *primitives*. Before discussing the model, we should explain what a primitive is.

10.1.1 Primitives

In OpenGL, the primitive decides how to interpret a set of vertices. The same set of vertices may be interpreted, hence rendered, very differently on the screen. Three points can be used to draw three single dots, a triangle or a part of a rectangle among other different geometries. To illustrate this idea, in figure 10.1, the four vertices $\{(0,0),(0,1),(1,1),(1,0)\}$ have been rendered with the primitives *GL_ POINTS*, *GL_ LINES*, *GL_ TRIANGLES*, *GL_ QUADS* and *GL_ TRIANGLE_ STRIP*. The rendered objects are of course very different depending on the primitive. When using for example the *GL_ TRIANGLES*, OpenGL interprets groups of three vertices as one triangle. In the case of *GL_ QUADS*, it will of course use groups of four vertices to define the renderable object. All of the primitives (except *GL_ POINTS* and *GL_ LINES*) form one or more two dimensional surfaces that are colored from either the interpolated values between the vertices (which have a defined color) or from a texture map.

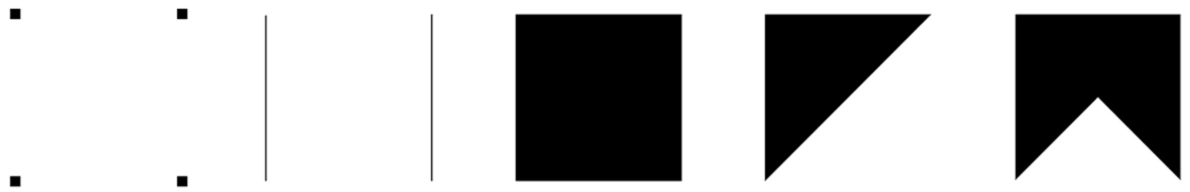

Figure 10.1: The vertices $\{(0,0),(0,1),(1,1),(1,0)\}$ have been rendered as the primitives (from left) *GL_POINTS*, *GL_LINES*, *GL_TRIANGLES*, *GL_QUADS* and *GL_TRIANGLE_STRIP*. We see that the final rendered geometrical objects are quite different for the different primitives.

10.1.2 Color interpolation

When creating a primitive consisting of N vertices, we can color each vertex $\mathbf{r}_i$ with an RGBA vector

$$\mathbf{c}_i = (r_i, g_i, b_i, \alpha_i) \tag{10.1}$$

where the components are the red, green, blue and alpha values for vertex i. The first three components define the color whereas the last component is the opacity. Since we only specify the color for the vertices, there are a lot of points in between them that do not have a defined color value. OpenGL assigns colors to these inner points by *linearly interpolating* the color values from the vertices. As an example, let us say we have a triangle with three vertices $\mathbf{p}_a, \mathbf{p}_b$ and $\mathbf{p}_c$. Any point $\mathbf{p}$ *in* this triangle can be uniquely specified by using *barycentric coordinates* which is a set of three numbers (a,b,c) in the range $[0,1]$, normalized so that $a+b+c=1$ [21]. Once we have these coordinates, the point $\mathbf{p}$ in the global coordinate system (in which the vertices $\mathbf{p}_i$ are defined) is found as

$$\mathbf{p} = a\mathbf{p}_a + b\mathbf{p}_b + c\mathbf{p}_c. \tag{10.2}$$

The barycentric coordinates are found through

$$a = \frac{A(\mathbf{p},\mathbf{p}_b,\mathbf{p}_c)}{A(\mathbf{p}_a,\mathbf{p}_b,\mathbf{p}_c)}, \qquad b = \frac{A(\mathbf{p},\mathbf{p}_a,\mathbf{p}_c)}{A(\mathbf{p}_a,\mathbf{p}_b,\mathbf{p}_c)}, \qquad c = \frac{A(\mathbf{p},\mathbf{p}_a,\mathbf{p}_b)}{A(\mathbf{p}_a,\mathbf{p}_b,\mathbf{p}_c)}, \tag{10.3}$$

where $A(\mathbf{a},\mathbf{b},\mathbf{c})$ is the area formed by the three vertices $\mathbf{a},\mathbf{b}$ and $\mathbf{c}$. If we want to find the color of a point $\mathbf{p}$ inside the triangle, We use the barycentric coordinates as the *weights* to linearly interpolate the colors specified for the vertices

$$\mathbf{c}(\mathbf{p}) = a\mathbf{c}_a + b\mathbf{c}_b + c\mathbf{c}_c, \tag{10.4}$$

where $\mathbf{c}_i$ is the color we gave the vertex at $\mathbf{p}_i$. In figure 10.2 we see how the colors are interpolated from the three values at the triangle vertices.

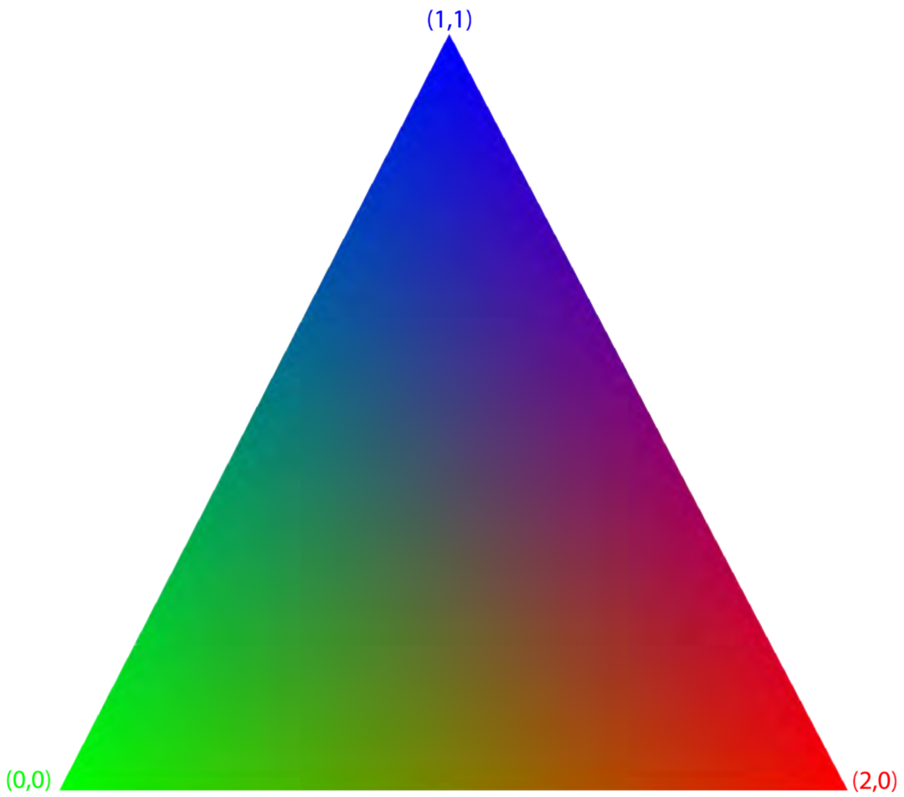

Figure 10.2: The three vertices $\{(0,0),(1,1),(2,0)\}$ colored green, blue and red, forms a triangle where the inner points of the triangle is colored by the interpolation between the three vertices.

10.1.3 Textures

Instead of using the colors specified per vertex, we can upload a texture (for example an image) to the graphics card so that the GPU can use that image as the source of how to color the pixels (this can actually be combined with the color values as we soon will see). A texture consists of $m \times n$ pixels where each pixel has a color $\mathbf{c}_t(i,j)$. Each vertex i of a triangle is assigned a *local texture coordinate*

$$\mathbf{t}_i = (t_x, t_y), \tag{10.5}$$

where $t_x, t_y \in [0,1]$. This is done so the pixel coordinates are independent of the size of the texture. We have a simple mapping to the *global texture coordinates* (the actual pixel coordinates in the image) $\mathbf{t}^* = (t_x^*, t_y^*)$ where

$$\begin{aligned} t_x^* &= m \cdot t_x \\ t_y^* &= n \cdot t_y, \end{aligned} \tag{10.6}$$

where we have added the asterisk on the global coordinates since we will mostly use the local ones. If a pixel with local coordinate $\mathbf{t}$ in the texture has a color value $\mathbf{c}_t(\mathbf{t})$, we can find the color $\mathbf{c}(\mathbf{p})$ of a point $\mathbf{p}$ within the triangle by interpolating the local texture coordinates just like we did with the colors. Remember that the point $\mathbf{p}$ has barycentric coordinates (a, b, c), which gives the interpolated local texture coordinates $\mathbf{t}$

$$\mathbf{t}(\mathbf{p}) = a\mathbf{t}_a + b\mathbf{t}_b + c\mathbf{t}_c, \tag{10.7}$$

where $\mathbf{t}_i$ is the local texture coordinate assigned to vertex i. The color of this point $\mathbf{p}$ is then simply

$$\mathbf{c}(\mathbf{p}) = \mathbf{c}_t[\mathbf{t}(\mathbf{p})]. \tag{10.8}$$

In figure 10.3 we have illustrated how the texture interpolation works. In the left figure, the texture is mapped with texture coordinates corresponding to the coordinates of the triangle vertices. In the figure to the right we have a skewed mapping making the image look skewed. We can of course combine both these methods (colors and textures) by applying both a texture

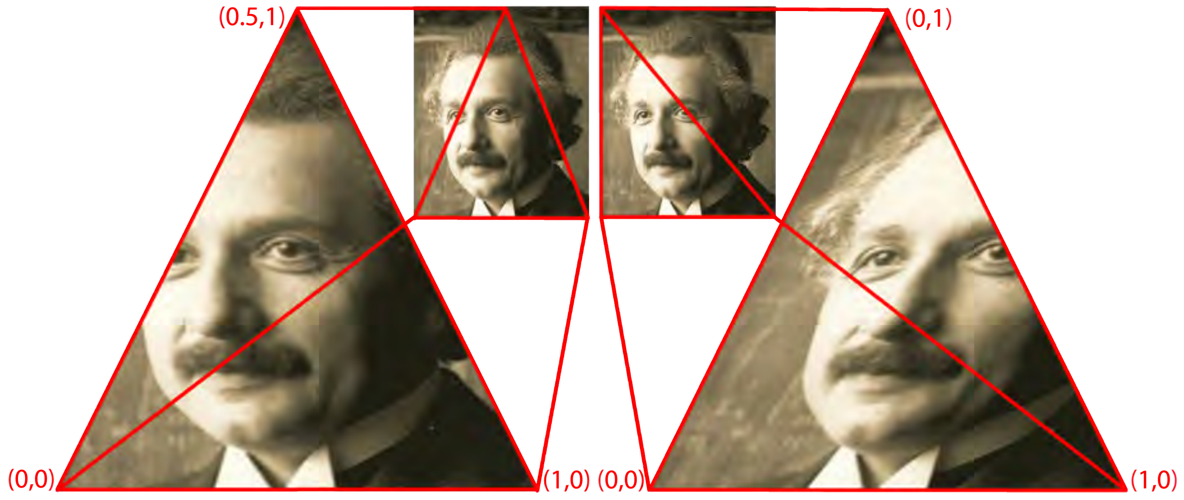

Figure 10.3: Showing how texture interpolation works. Left: the three vertices $\{(0,0),(1,1),(2,0)\}$ are assigned the local texture coordinates $\{(0,0),(0.5,1),(1,0)\}$ where we see how the texture are transformed onto the triangle . Right: the three vertices $\{(0,0),(1,1),(2,0)\}$ are assigned the local texture coordinates $\{(0,0),(0,1),(1,0)\}$ where we see that the texture on the triangle is skewed because of the transformation. Note that the coordinates in the figure at the vertices are the texture coordinates, not the coordinates of the vertices.

coordinate *and* a color value to each vertex. OpenGL will then render an image where the rendered color $\mathbf{C}$ becomes

$$\mathbf{C}(\mathbf{p}) = \mathbf{c}_t[\mathbf{t}(\mathbf{p})] \odot \mathbf{c}(\mathbf{p}), \tag{10.9}$$

where $\odot$ is the element-wise multiplication operator defined as

$$(\mathbf{a} \odot \mathbf{b})_i = a_i \cdot b_i, \tag{10.10}$$

where $\mathbf{a}, \mathbf{b} \in \mathbb{R}^N$ and d_i is the i'th component of vector $\mathbf{d}$. In figure 10.4 we have combined both the color and the texture giving a triangle with the same image of Albert Einstein colored in the same way as the triangle in figure 10.2.

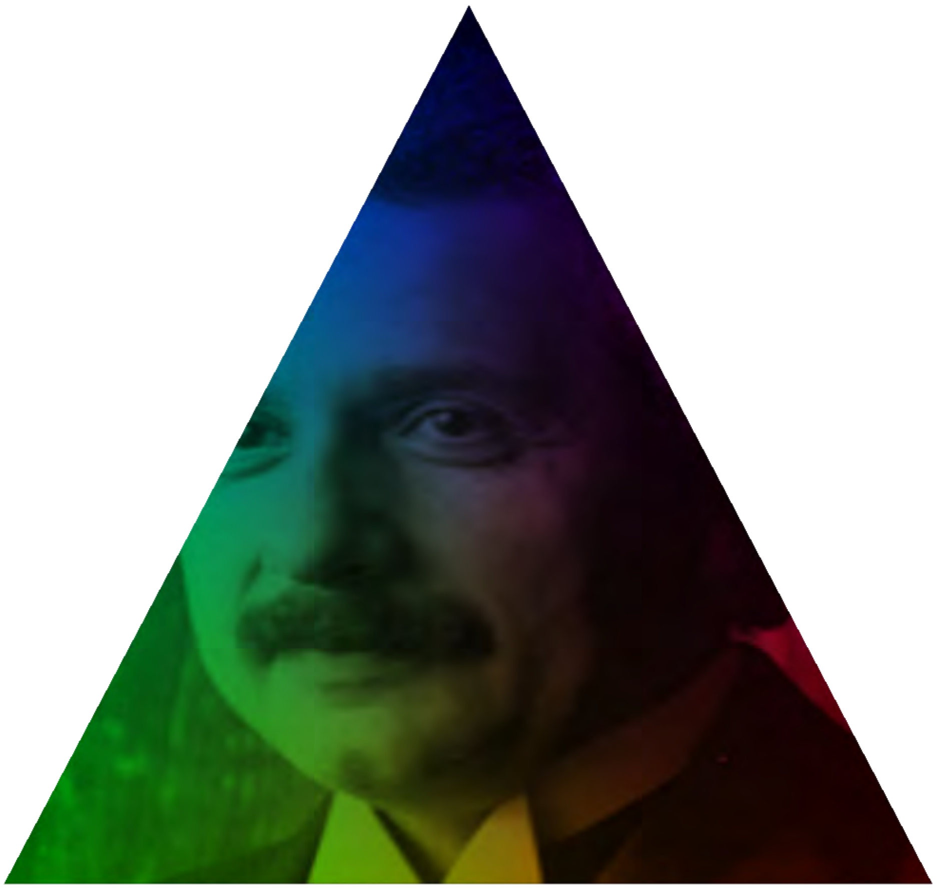

Figure 10.4: We can combine both colors and textures to color points within a triangle according to equation (10.9). Here we have used the same colors as in figure 10.2 and the texture of Albert Einstein in figure 10.3.

10.1.4 Model

A model of an object contains all the information fully defining how a geometric object looks like without any effects from the environment, such as light. The model is fully described by a set $\mathcal{M}$ containing primitive objects $\mathcal{P}$, each having

- a vertex array,
- the primitive type,
- a color array,
- a texture id,
- a local texture coordinate array, and

- a normal vector array.

We have discussed the vertices, the primitive, the color and texture coordinate arrays (remember, one color and one texture coordinate per vertex). We also need a *texture id* (which is just an int variable) allowing us to tell the GPU which texture it should apply to this primitive object. In addition, we can assign a *normal vector* to each vertex which can be used to create realistic lighting and other effects. It's time for an example of a model.

We now want a model of a die. A die is a cube with six faces. The model $\mathcal{M}$ would then contain six primitive objects $\mathcal{P}_i$, each having an array of four vertices. The primitive type would be *GL_ QUADS* where the color of course could be your favorite color. We need to upload six textures (one face has one dot, another face has two dots etc) which gives us six different texture id's. The local texture coordinate array is simply the four corners of the unit square whereas the normal vectors should point outwards from the cube. In figure 10.5 we have used this technique to draw a red die.

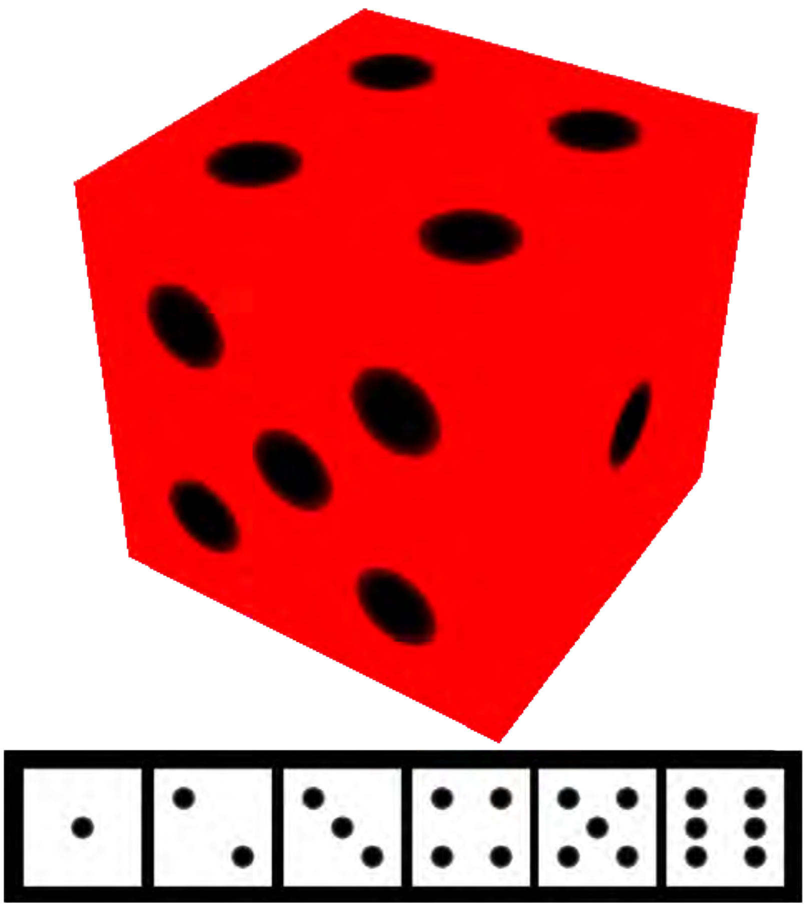

Figure 10.5: Drawing a die using the technique described in subsection 10.1.4. The color was chosen to be red whereas each of the six faces were assigned one of the textures in the black box.

10.2 Vertex Buffer Objects

A Vertex Buffer Object (VBO) is a feature in OpenGL that allows the user to upload an array of vertices to the graphics card which can be used for very fast rendering. This could for example be a set of positions, colors or normal vectors. Once this data is uploaded to the GPU, we can render the model described by the VBO in very few function calls. We ask the GPU to give us a buffer identifier. Then we tell the GPU to allocate enough memory to store our vertices before we copy the data from the main memory to the graphics card. The data is now available on the graphics card.

10.3 Rendering pipeline

Now that we know what a VBO is, we are ready to render the objects described by a set of VBOs on the screen. First we activate the data we want to render by telling the graphics card (of course by using the OpenGL API) which VBO identifier represents the positions, colors, normal vectors and texture coordinates. Then, we ask the GPU to render the activated data as the primitive the vertices represent (we know if the vertices should be rendered as triangles, quads or lines).

We can now make full use of the parallel architecture of the GPU. On a high end NVIDIA graphics card[1], there are often more than 2000 cores available to run small programs called *shaders* that process the input data from vertices to the final image rendered on the screen. The shaders are written in a programming language called GLSL[2] and are compiled on the GPU runtime. The full job of processing all the vertices calculating the final color values for all the pixels are distributed on these cores. In this section we discuss the different rendering steps in what's called the *rendering pipeline* that are relevant to develop the visualization tools in this thesis. In figure 10.6 we have illustrated the rendering pipeline containing the stages that we have used.

10.3.1 Ins, outs and uniforms

As we have already mentioned, the shaders are small programs run in parallel. Each shader gets some input data which are called *input*-variables. These are specific values for this *instance* of the shader, for example the position vertex for a particle. The shader then usually calculates *something* (a common task in the first shader stage is to convert a vertex from the model space to the projection space), before it sends a variable to the next step in the pipeline. The variables going out of a shader is of course called an *output*-variable. Just to be sure we understand; the next shader gets this *output*-variable as an *input*-variable for further processing.

[1]Such as the NVIDIA GTX Titan. At the time of writing, it costs about 8000 NOK on http://www.komplett.no.

[2]GLSL (OpenGL Shading Language) is a programming language with syntax similar to C. It provides a set of linear algebra operations that are heavily used in computer graphics.

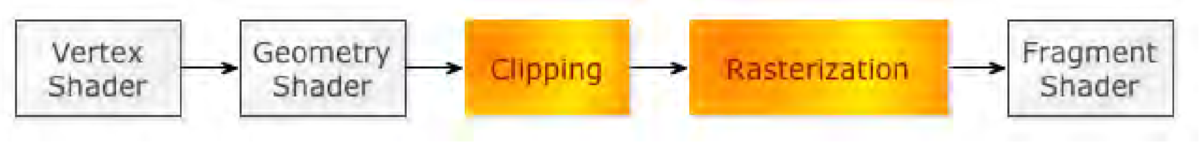

Figure 10.6: Diagram of the OpenGL rendering pipeline. The gray boxes are shader steps (the shader programs can be written by the user). Each vertex is processed in the vertex shader before it is sent to the geometry shader that can create zero or more vertices based on the input vertex. Each vertex is then analyzed in the clipping program that removes vertices that cannot be rendered. The rasterization program creates a two dimensional image of every triangle so that the fragment shader can merge everything to color values per pixel on the screen. Not that the full pipeline contains more steps, but these are the relevant stages for understanding what our visualization tool does.

But there are of course some variables that are constant through the whole rendering process. If all the particles have the same color, we don't need to specify the color per atom (this would use a lot of extra bandwidth on the GPU). Variables like this should be sent to the GPU as so-called *uniform*-variables. They are available in all instances of every shader stage.

10.3.2 Vertex shader

The vertex shader is executed once per vertex in the input VBOs. It specifies which input vertex that is to be interpreted as the position, color, normal vector and/or texture coordinate if they are specified. The input position vertex is usually in the model space which are local coordinates for this specific object. A typical vertex shader applies the Model View Projection matrix on the vertex, transforming it from the model space to the projection space which often is assumed in the later pipeline stages. The latter may be done on the geometry shader instead if a three dimensional object is created at that stage.

10.3.3 Geometry shader

Each vertex from the vertex shader is a part of a primitive (such as *GL_POINTS* or *GL_TRIANGLES*). A geometry shader takes a primitive (i.e. a set of vertices, each processed by the vertex shader) as input and outputs zero or more primitives that may be of another type than the input primitive. A typical use is to describe a geometrical object (this could be a sphere or a tube) on the geometry shader so that the input primitive is just one single vertex; the position. This significantly reduces the memory and bandwidth usage on the GPU which usually gives great performance improvements.

The geometry shader can also be run in *instancing mode* which means that the shader program is run a given number of time per vertex with an invocation id given. If a particle system obeys periodic boundary conditions, we can for example use the geometry shader instancing to add 26 periodic copies of the system making the system look larger than it is.

10.3.4 Clipping

After the geometry shader has decided which primitives we want rendered, some of the vertices may not be visible on the screen. They can be behind the camera, too far to the right or in some way outside the camera view. When we in the final stages (the fragment shader) are going to decide colors on every pixels, we don't need to compute primitives that does not contribute to the final image. This is called clipping and is a very simple process to perform in the projection space. All vertices outside the clip volume (the volume that will be rendered) will be discarded and not computed at the rasterization stage.

10.3.5 Rasterization

The rasterization program will for each primitive determine which pixels that are a part of the primitive. Each active pixel will have interpolated colors and texture coordinates as described in subsection 10.1.3 before we reach the fragment shader, the final stage in the rendering pipeline.

10.3.6 Fragment shader

The fragment shader is executed one time *per pixel* for each of the primitives that contributes to that pixel. If the primitives are transparent, the final color value on a pixel will be a combination of all the primitives. The fragment shader gets interpolated color values and texture coordinates (like we explained in section 10.1.2) that are used to decide the actual color of that pixel. This is the last step we cover in the OpenGL pipeline. The final color values are then written to a *framebuffer* which finally is the image that is shown on the screen on the computer.

Chapter 11

Advanced particle visualizer

With our newly acquired knowledge about OpenGL and learned how we can use the API to render objects on the screen, we have everything we need to develop our own visualization tools that can handle datasets from both MD and DSMC. As we now should be well aware of, the state of a system with N particles is described by the $3N$ particle positions and the $3N$ velocity components. If we save this information every timestep of a simulation, we can use it to render a time series, an animation of the trajectories of all the particles. We will render the particles as spheres, but are going to cheat a bit. An actual sphere rendered in OpenGL would need to be composed of many triangles forming the spherical shape. To be able to render a smooth sphere, we would need more than 100 triangles *per sphere* as we will see in section 11.1. We will apply a trick used in computer games for years. Instead of rendering spheres, we use something called *billboards*, which is a rectangle with an *image* (a texture) of a sphere, always pointing towards the camera. In section 11.1, we explain how we effectively can create and render billboards with the geometry shader on the GPU. If the particle system has periodic symmetry (both MD and DSMC use periodic boundary conditions), we can also use the geometry shader to render copies of the system, making the illusion that the system is larger than it really is.

We did a performance test to see the amount of particles it is possible to visualize with this method. The benchmark measure is the number of frames per second the program can manage to render. This is done in section 11.1.3. When we visualize a dataset from a DSMC simulation, we should, in addition to the particle positions, render the surface geometry (which, as we remember from section 5.1, is a voxelized scalar field). We will then be able to see the surface the particles collide with which will make it easier to understand their behavior. In section 11.2, we discuss the so-called *marching cubes* algorithm, which allows us to create a set of renderable triangles from the iso-surface of a scalar field (the points where the scalar field values intersect some value).

11.1 Billboards

We could in principle create a spherical shape by connecting triangles or other primitives as illustrated in figure 11.1. The figure contains two spheres with different numbers of primitives. We need a substantial number of primitives to get a shape that looks like a sphere (200 in the left sphere in the figure), but we can cheat a bit, by instead rendering something that *looks* like a sphere A billboard is, as the name suggests, a rectangle filled with a texture, always pointing

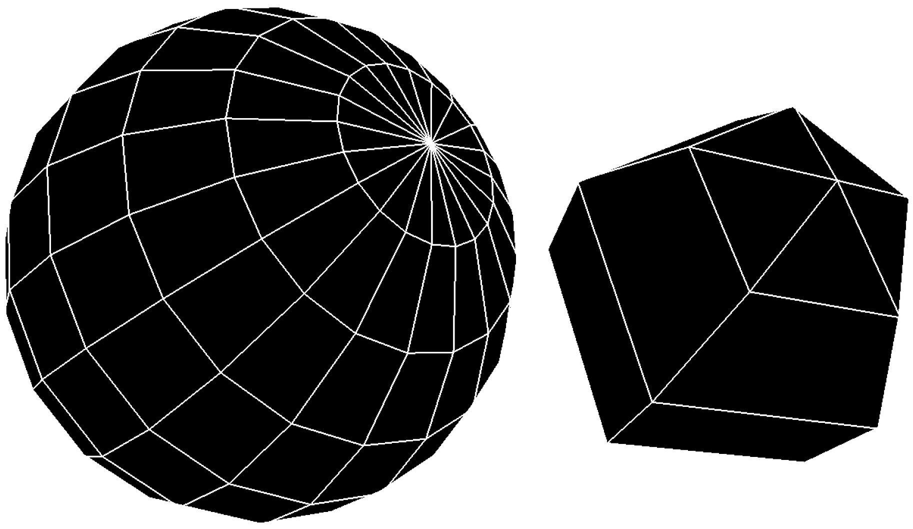
Figure 11.1: A sphere can be created by connecting triangles or other primitives. The sphere to the left is made of 200 primitives whereas the one to the right is made of 25 primitives. It is clear that we need a certain number, more than 25, of primitives before we are convinced that the object should represent a sphere.

towards the camera. The texture is going to be a circle (a sphere does indeed look like a circle when rendered on the screen anyway). It is quite easy to render such a rectangle with OpenGL, we already have a primitive called *GL_ QUADS* which is exactly what we need. The only thing we need to do is provide four vertices, the corners in the rectangle, and give each vertex a texture coordinate. The four vertices $\mathbf{v}_i$ can be calculated from one single vertex $\mathbf{r}$, the particle position, by

$$\begin{aligned}
\mathbf{v}_1 &= \mathbf{r} + (-\Delta x, -\Delta y) \qquad (11.1)\\
\mathbf{v}_2 &= \mathbf{r} + (-\Delta x, \Delta y)\\
\mathbf{v}_3 &= \mathbf{r} + (\Delta x, \Delta y)\\
\mathbf{v}_4 &= \mathbf{r} + (\Delta x, -\Delta y),
\end{aligned}$$

where $(L_x = 2\Delta x, L_y = 2\Delta y)$ is the size of the billboard, see figure 11.2. This is already what Ovito does[1], but we are going to improve this even more. Instead of computing the four vertices

[1]See the source code at http://www.ovito.org/index.php/download2

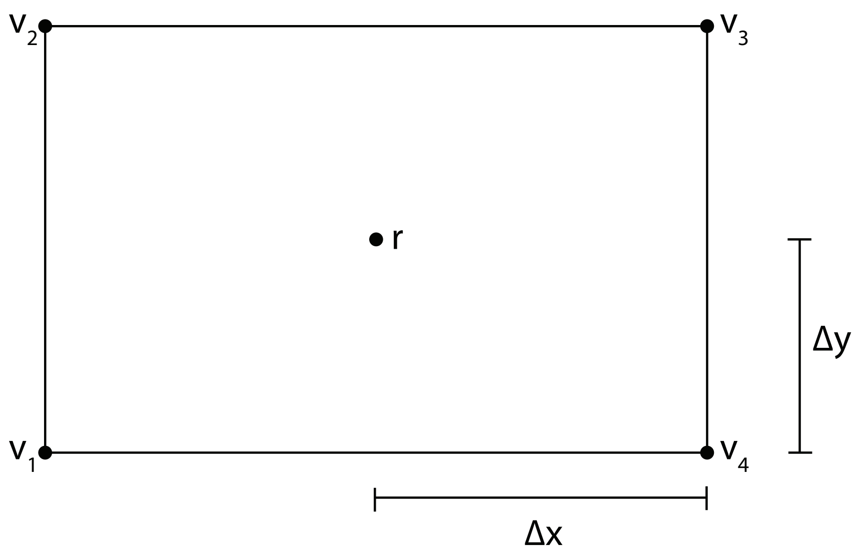

Figure 11.2: A billboard is made of four vertices $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ and $\mathbf{v}_4$ that can be calculated from one vertex $\mathbf{r}$ (equation (11.1)).

and uploading these as *GL_QUADS* to the GPU, we will only upload the positions of the particles, and exploit the geometry shader to create the billboard vertices on the GPU.

Before we start visualizing the data, all timesteps of a simulation are uploaded as VBO's on the GPU so that we don't need to upload data every frame. The VBO is rendered as *GL_POINTS* so each vertex represents the center position of a particle. We will now follow a single vertex through the pipeline

11.1.1 The pipeline

The VBO can now be seen as one OpenGL model containing the positions of all the particles. The vertices are then in the *model space*. Every single vertex starts its life in the pipeline by going into the vertex shader. The vertex shader is pretty simple in this case, it just transforms the input vertex from the model space to the projection space as we can see in listing 11.1.

```
#version 330\n"
uniform mat4 qt_ModelViewProjectionMatrix;
in vec4 qt_Vertex;
void main(void)
{
    gl_Position = qt_ModelViewProjectionMatrix * qt_Vertex;
}
```

Listing 11.1: billboardVertexShader.glsl

This position is then sent to the geometry shader as an instance of the *GL_POINTS* primitive. The geometry shader will, from that single vertex (the position, now in the projection space), create and emit *four* vertices that are displaced from the input vertex as explained in equation (11.1) and figure 11.2. The output primitive is here the *GL_TRIANGLE_STRIP* which will create two triangles with vertices $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$ and $\{\mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4\}$. The size of the billboard is available as a uniform (as explained in subsection 10.3.1) in the geometry shader. If we want to visualize particles with different sizes (e.g. two atom types with different atomic radii), we just render different VBO's with the corresponding size. We want the four vertices to form a rectangle pointing towards the camera, so the operation in equation (11.1) should be done in the view space (here the z-direction is orthogonal on the camera plane which forms the xy-plane). Since the position vertex now is in the projection space, we should transform the four displacement vectors to the projection space by applying the *qt_ProjectionMatrix* before we add them together. We then set the texture coordinate (the local coordinate in the texture image) on each vertex and emit it with the *EmitVertex()* function. The algorithm might be easier to understand with a code example, see listing 11.2.

```
#version 400
layout( points ) in;
layout( triangle_strip , max_vertices = 4 ) out;
uniform mat4 qt_ProjectionMatrix;
uniform vec2 size;
out vec2 texCoord;

void main(void) {
    vec4 pos = gl_in[0].gl_Position;
    gl_Position = pos + qt_ProjectionMatrix*vec4(-size.x, -size.y, 0.0,
        0.0);
    texCoord = vec2(0.0, 0.0);
    EmitVertex();
    gl_Position = pos + qt_ProjectionMatrix*vec4(-size.x, size.y, 0.0, 0.0)
        ;
    texCoord = vec2(0.0, 1.0);
    EmitVertex();
    gl_Position = pos + qt_ProjectionMatrix*vec4(size.x, -size.y, 0.0, 0.0)
        ;
    texCoord = vec2(1.0, 0.0);
    EmitVertex();
    gl_Position = pos + qt_ProjectionMatrix*vec4(size.x, size.y, 0.0, 0.0);
    texCoord = vec2(1.0, 1.0);
    EmitVertex();
    EndPrimitive();
}
```

Listing 11.2: billboardGeometryShader.glsl

This is done with every single position vertex in the VBO (once per particle per frame) and all the output primitives from the geometry shader will be rasterized, clipped and texture coordinates are interpolated into the fragment shader that is run once per pixel per visible primitive. Before we discuss the fragment shader, we should take a look at the texture that each billboard will have.

We didn't lie before, we will use a circle that actually is a real sphere rendered in the 3D modeling program Blender[2]. With some lighting, we get this nice shiny effect making the ̈spheres ̈look more interesting. This texture is shown in figure 11.3, where we also see that all the pixels outside the circles are transparent. This is very important so that we can discard these contributions in the fragment shader. The fragment shader will then get an interpolated texture coordinate as

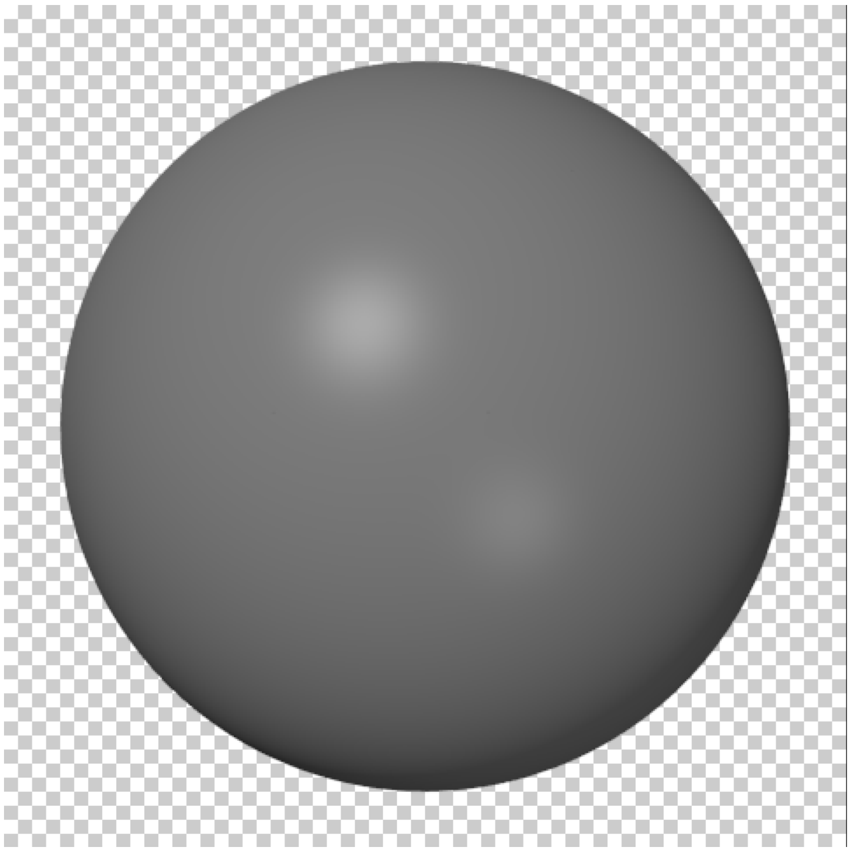

Figure 11.3: The texture used on the billboards is an image of a real sphere rendered with Blender. Notice that the pixels outside the circle are transparent. This allows us to discard these texture pixels in the fragment shader.

input, which we will use to look up what RGBA value this pixel gets from the texture. If the alpha value (the opacity) is smaller than one, it means that the pixel contribution comes from a texture pixel outside the circle. In that case we just discard it. We almost forgot one last detail, the *color* of the particles. Again we assume that all particles have the same color (if not, we just use multiple VBO's per timestep), so as we did with the billboard size variable, the color is available as a uniform. The final pixel color is then found by equation (10.9)

$$\mathbf{C}(\mathbf{p}) = \mathbf{c}_t[\mathbf{t}(\mathbf{p})] \odot \mathbf{c}(\mathbf{p}).$$

The fragment shader code is shown in listing 11.3 with the final rendering result in figure 11.4. In this rendered image, we also added light effects making atoms that are far away from the

[2]Available from http://www.blender.org/.

camera appear darker. This increases the feeling of depth which clarifies the pores.

```
#version 330
uniform vec4 color;
uniform sampler2D qt_Texture0;
in vec2 texCoord;
out vec4 MyFragColor;

void main(void) {
    MyFragColor  = texture2D(qt_Texture0, texCoord.st);
    MyFragColor  = MyFragColor * color;
    if(MyFragColor.a < 0.9999) {
        discard;
    }
}
```

Listing 11.3: billboardFragmentShader.glsl

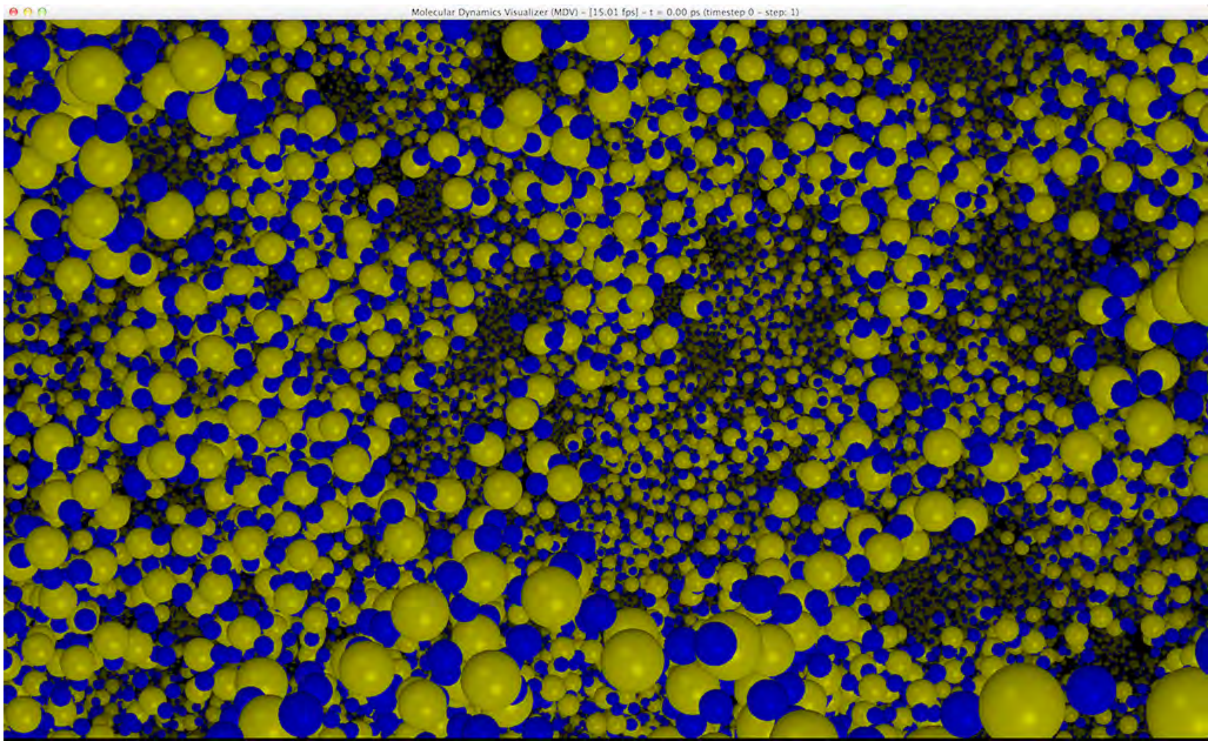
Molecular Dynamics Visualizer (MDV) - [15.01 fps] - t = 0.00 ps (timestep 0 - step: 1)

Figure 11.4: The final rendered image with an MD simulation using billboards. In the rendering, light effects are added to increase the feeling of depth, clarifying the pore structure. The atoms form a nanoporous SiO_2 system simulated with a MD code developed at the University of Southern California.

11.1.2 Periodic copies

We used the geometry shader to create a *GL_ TRIANGLE_ STRIP* creating the billboard from one single vertex (the particle position). Then, one may ask, why not use the geometry shader to create *several* billboards, adding periodic copies of every particle in the system? This makes sense if the simulation operates with periodic boundary conditions so we have periodic symmetry on all directions. So, if the system is a box of size $L_x \times L_y \times L_z$, we can create 26 copies of the whole system giving a larger box of size $3L_x \times 3L_y \times 3L_z$, making the impression that the system is larger. This provides an important effect near the boundaries of the system, see figure 11.5. We will use a feature in OpenGL called geometry shader instancing to add all the copies. With

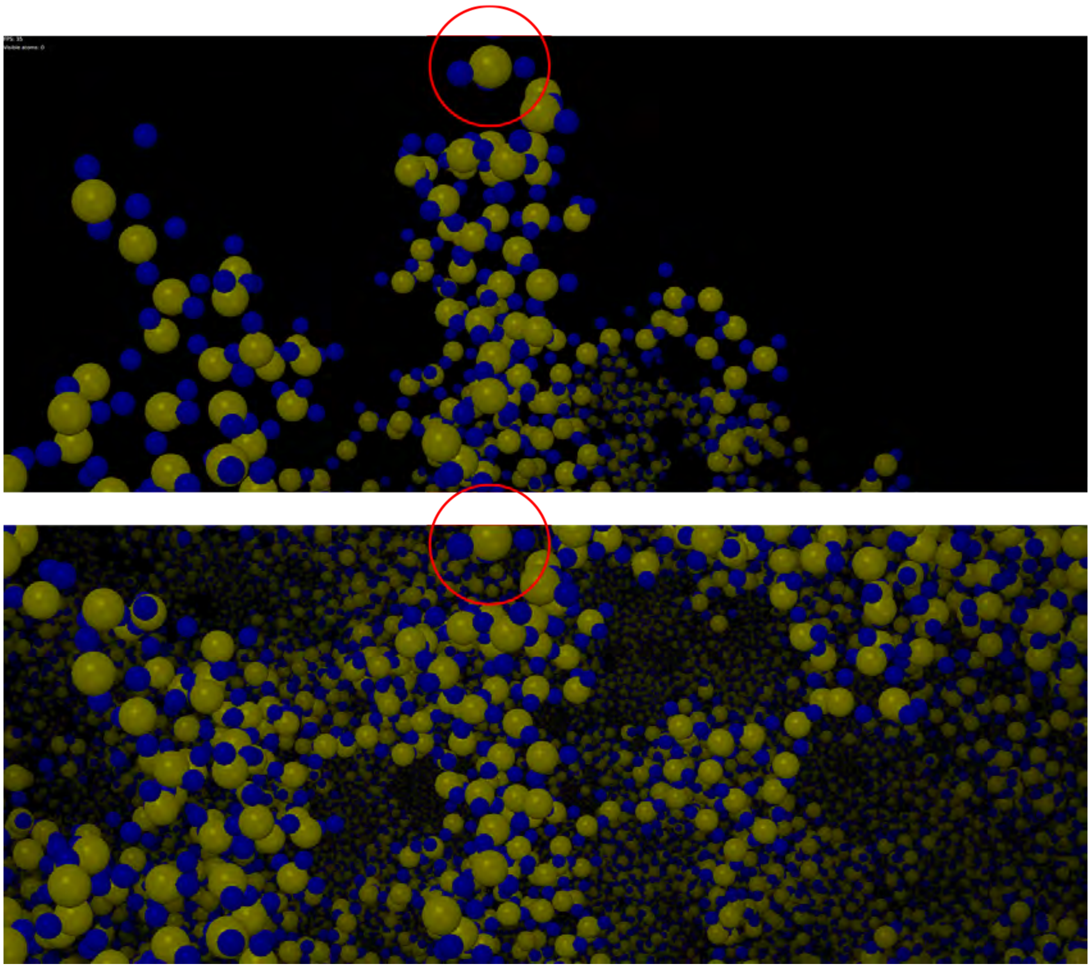

Figure 11.5: Adding periodic copies of the system provides an important effect when looking at atoms near the boundary. The upper image shows a visualization with no periodic copies of the atoms. This shows that the system clearly has a finite size. In the lower image, by adding 26 periodic copies of the whole system, it is not obvious that we are at the edge of the system.

instancing, the geometry shader is executed 27 times per input vertex. We can use the variable *gl_ InvocationID* to identify which of the 27 instances we are running. The system coordinate

$(S_x, S_y, S_z), S_i \in \{-1, 0, 1\}$ for invocation id i is calculated as

$$S_x = (i \bmod 3) - 1 \tag{11.2}$$
$$S_y = (\lceil i/3 \rceil \bmod 3) - 1 \tag{11.3}$$
$$S_z = (\lceil i/9 \rceil) - 1, \tag{11.4}$$

where $\lceil a \rceil$ is the ceiling function. One instance of the geometry shader should then render a billboard displaced with

$$\mathbf{d} = (L_x \cdot S_x, L_y \cdot S_y, L_z \cdot S_z), \tag{11.5}$$

where the system size (L_x, L_y, L_z) is available as a uniform. The displacement $\mathbf{d}$ is given in the model space, so we must transform it the projection space before we add it to the position vertex (which, as we remember, already is in the projection space). The full geometry shader code using instancing to add periodic copies is found in listing 11.4.

```
#version 400
layout(invocations=27) in;
layout(points) in;
layout(triangle_strip, max_vertices = 4) out;
uniform mat4 qt_ProjectionMatrix;
uniform mat4 qt_ModelViewProjectionMatrix;
uniform vec2 size;
uniform vec3 systemSize;
out vec2 texCoord;

void main(void) {
    int x = gl_InvocationID % 3 - 1;
    int y = (gl_InvocationID/3) % 3-1;
    int z = (gl_InvocationID/9)-1;

    vec4 displacement = vec4(systemSize.x*x,systemSize.y*y,systemSize.z*z,0);
    vec4 pos = gl_in[0].gl_Position + qt_ModelViewProjectionMatrix *
        displacement;
    gl_Position = pos + qt_ProjectionMatrix*vec4(-size.x, -size.y, 0.0, 0.0);
    texCoord = vec2(0.0, 0.0);
    EmitVertex();
    gl_Position = pos + qt_ProjectionMatrix*vec4(-size.x, size.y, 0.0, 0.0);
    texCoord = vec2(0.0, 1.0);
    EmitVertex();
    gl_Position = pos + qt_ProjectionMatrix*vec4(size.x, -size.y, 0.0, 0.0);
    texCoord = vec2(1.0, 0.0);
    EmitVertex();
    gl_Position = pos + qt_ProjectionMatrix*vec4(size.x, size.y, 0.0, 0.0);
    texCoord = vec2(1.0, 1.0);
    EmitVertex();
    EndPrimitive();
}
```

Listing 11.4: billboardGeometryShaderWithPeriodicCopies.glsl

11.1.3 Rendering benchmark

A typical benchmark for a visualization program is the number of frames per second (FPS) the program is able to render a given amount of primitives. In this benchmark, we have used the multibillboard class[3] that has implemented the shader pipeline explained above. It renders a box with N billboard spheres and adds 26 copies in the geometry shader making the total number of visible spheres $N_{\text{spheres}} = 27N$. In table 11.1 we have shown a representable subset of the data showing the performance of the billboard class. We have run the program with the number of visible spheres in the range a few thousand to more than one billion. In figure 11.6 we have plotted the data from table 11.1 with more data points near the sudden drops of frame rate. We assume that these drops can be explained by meeting different limits on the graphics card. It seems to happen at specific number of vertices emitted by the geometry shader, so our guess would be the bandwidth on the GPU. We have not used any shader profiling tools, so we cannot draw any conclusion without further investigation.

N_{spheres}	FPS
3,375	62.09 ± 3.236
27,000	61.80 ± 3.149
729,000	62.03 ± 3.088
5,832,000	62.28 ± 3.061
19,034,163	30.51 ± 1.601
27,000,000	30.36 ± 1.436
45,499,293	20.81 ± 2.276
70,957,944	15.15 ± 0.879
110,592,000	12.14 ± 2.180
132,651,000	9.99 ± 1.066
287,496,000	4.78 ± 0.8532
421,875,000	3.29 ± 0.4548
1,061,208,000	1.40 ± 0.2386

Table 11.1: Benchmark showing the number of frames per second (FPS) the billboard class is able to render with the number of billboard spheres from a few thousand to more than one billion visible spheres. The benchmark is performed on an NVIDIA GTX Titan.

The visualization tool for an MD state does not need anything else than what we now have presented. The timesteps containing the positions of all the atoms are uploaded to the graphics card as VBOs and the atoms are rendered as points into the rendering pipeline where the billboards are created in the geometry shader. This same rendering technique can of course be used to render the particles in a DSMC simulation, but we want to render the complex geometry that the particles interact with. We will use what's called marching cubes for this.

[3]Source code available from http://github.com/ComputationalPhysics/compphys-qt3d-additions.

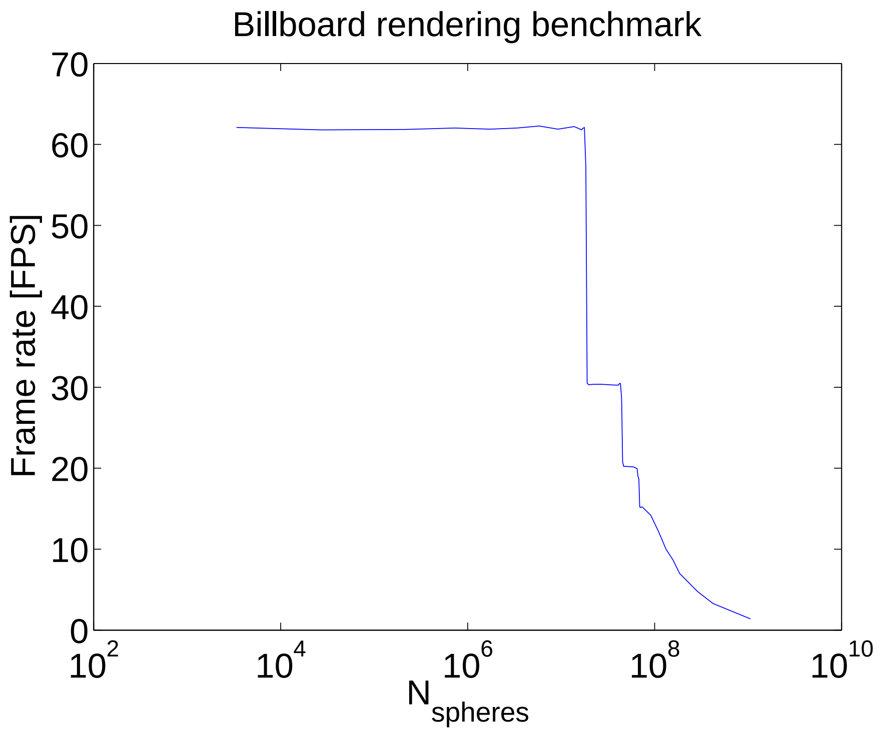

Figure 11.6: Benchmark showing the number of frames per second (FPS) the billboard class is able to render with the number of billboard spheres from a few thousand to more than one billion visible spheres. The benchmark is performed on an NVIDIA GTX Titan. The sudden drops in FPS at specific number of spheres are probably explained by reaching the bandwidth limit in the different shader stages.

11.2 Marching cubes

The geometry in DSMC is represented as voxels as we discussed in section 5.1. They form a scalar field with values zero for empty voxels and non-zero for filled voxels. The surface of the geometry is where the values of the scalar field *changes* from zero to a non-zero value. This defines the *iso-surface*, i.e. a surface where all the values on one side are zero, and non-zero on the other side. If we find a way to visualize this iso-surface, it will coincide with where the DSMC particles collide. We then need to create some primitives (in this case triangles) that are connected to each other, forming a the full iso-surface. Such an algorithm exists and is called *marching cubes.*

Marching cubes is an algorithm used to generate a set of connected triangles from the iso-surface of a scalar field. The method was presented in a paper published in 1987 and has been widely used in medical visualizations of CT and MRI scans [28]. Assuming that the scalar field is discretized in space, each point - vertex in our case - has a value larger or larger than some chosen iso-value. Given a cube consisting of eight of these vertices, there exists $2^8 = 256$ unique combinations (each vertex has two possibilities). Because of symmetries (a cube with only one vertex being larger than the iso-value has through rotations 8 different configurations that really are the same configuration), this set can be reduced to 15. In figure 11.7, we see the 15 unique configurations and the corresponding triangles in each configuration. For a given cube, the 8

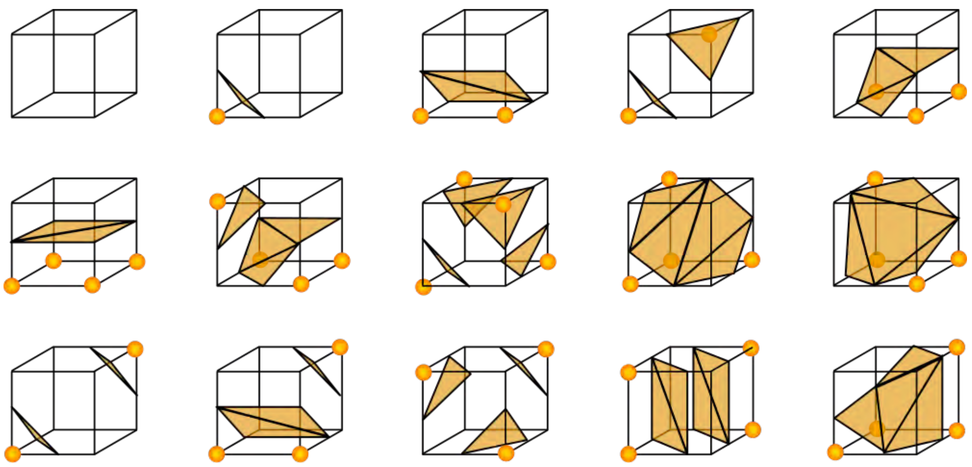

Figure 11.7: A cube consisting of eight vertices. Given that each vertex can have a value smaller or larger than some given iso-value, the cube has has $2^8 = 256$ different combinations. This number can, as we see here, be reduced to 15 due to symmetries. An iso-surface on a scalar field can then be generated as renderable triangles with this technique. Image from http://en.wikipedia.org/wiki/File:MarchingCubes.svg, accessed 20 March, 2014.

vertex values (one or zero) can be represented as the bits in an 8-bit integer. The final value of this integer is then the index of a precomputed table containing a list of all triangles needed for that configuration.

The original authors thought the 15 combinations would be enough, but it turns out that with these 15 configurations, there exists cases where the surface gets holes - it is not topologically correct. This problem was solved in 1995 by using a larger set with 33 unique configurations which spans out the full configuration space [6]. We have used a precomputed table with 256 configurations made from this basis of 33 elements. Each vertex is part of at least one triangle. We can then compute the normal vector per vertex as an average of the normal vectors of all triangles it is a part of. This enables the fragment shader to get interpolated normal vectors per vertex giving beautiful, smooth shadows as we can see in figure 11.8. Surfaces having a normal vector pointing towards the camera have maximum light intensity. All other surfaces will have reduced light intensity proportional to the dot product between the normalized camera-to-object vector and the normal vector.

Since the geometry in DSMC model is described by a scalar field with values zero, one and two, we can choose iso-value of one and use the marching cube algorithm to generate triangles we can render on the GPU. The set of triangles is uploaded to the graphics card as a VBO and is rendered with a simple draw call. In figure 11.8, we have rendered the packed spheres we studied in section 6.4 with the marching cubes algorithm. Here we used $256 \times 256 \times 256$ voxels obeying periodic symmetry in all directions. Two overlapping spheres will have a smooth, shared surface as we see in the top of the figure.

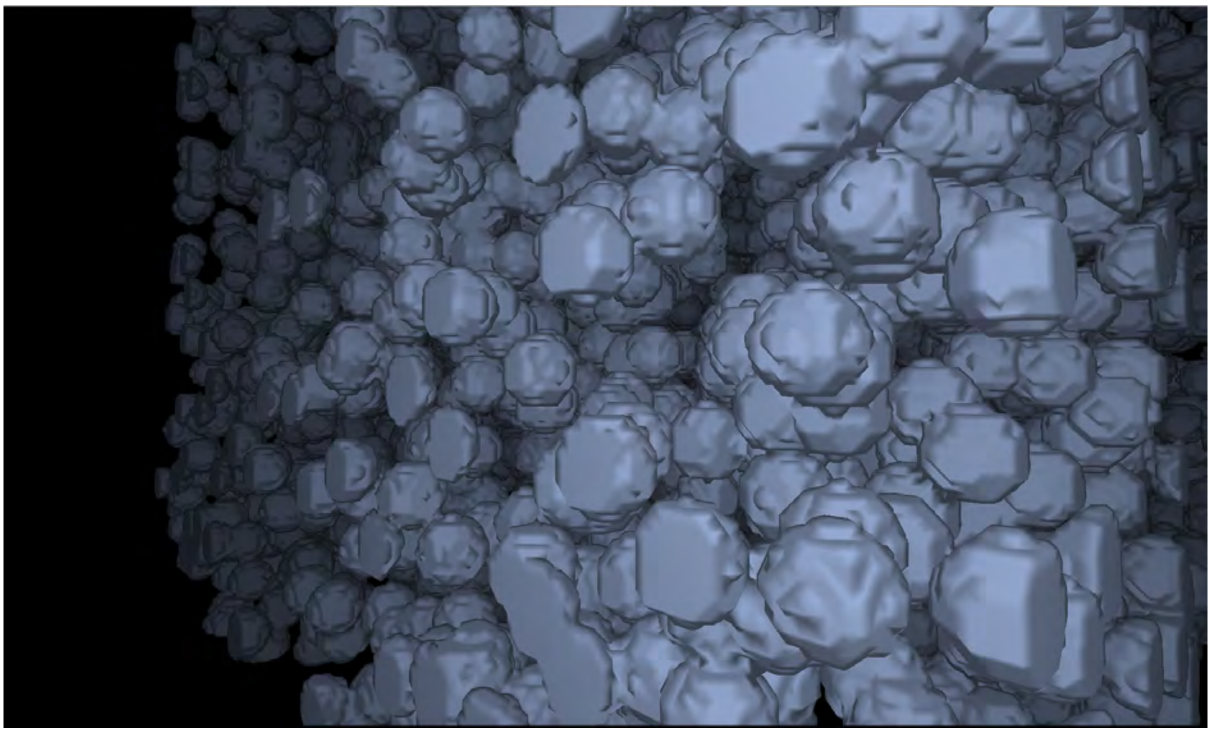

Figure 11.8: Randomly packed spheres in DSMC visualized with the marching cubes algorithm. The geometry is made up by $256 \times 256 \times 256$ voxels obeying periodic symmetry in all directions. In the top of the figure, we see two how marching cubes elegantly renders two overlapping spheres. Normal vectors are computed for each vertex by averaging the normal vectors of all the triangles it is a part of. This enables the fragment shader to get interpolated normal vectors per vertex giving beautiful, smooth shadows.

11.3 Gallery

A picture is worth a thousand words. In this section we will show some screen shots from the visualization tool we have made. The DSMC visualizer can both render the state of the system while it integrates the system forward in time - a live visualization. This makes it easier to find interesting regions in the system, or is great as a show-off case.

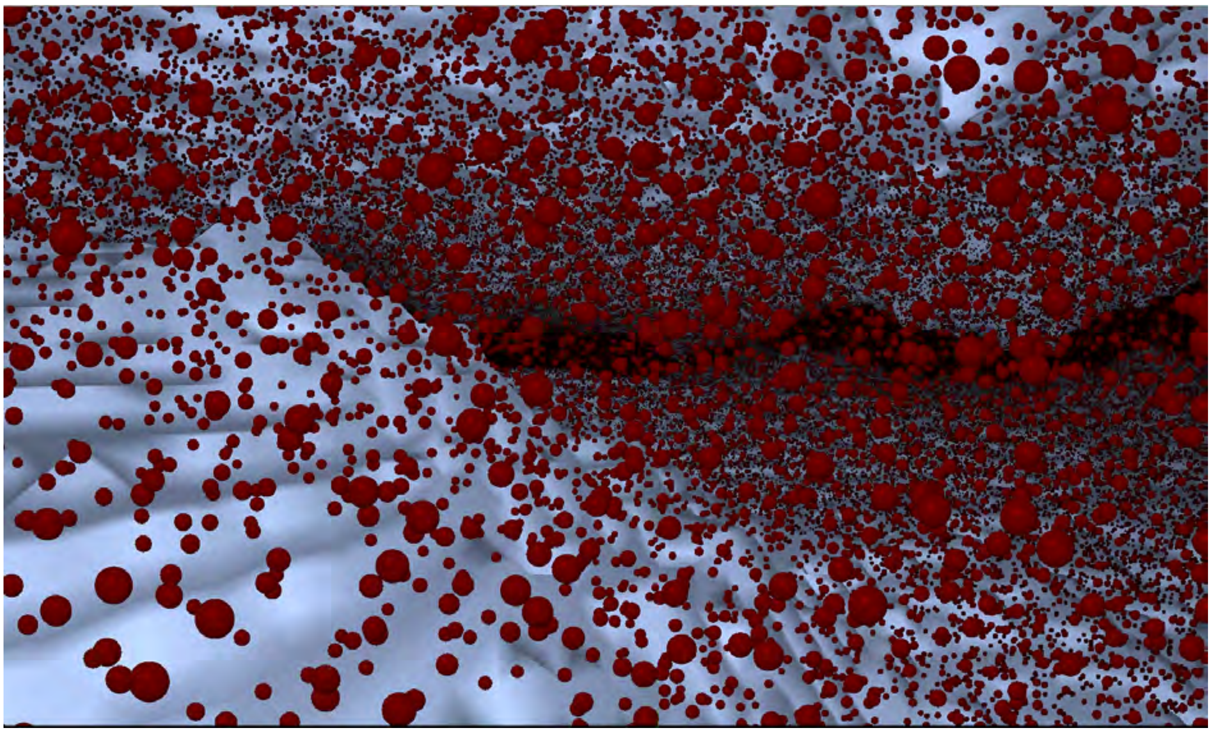

Figure 11.9: Here we see a live simulation on a 2013 Macbook Pro. One million DSMC particles in a fracture created with the diamond-square algorithm (a thanks to Filip Sund who implemented this algorithm). We use billboards to render the particles and the marching cubes algorithm to create a smooth surface from the voxelized scalar field. With a good frame rate it is easy to study flow in any region of the system.

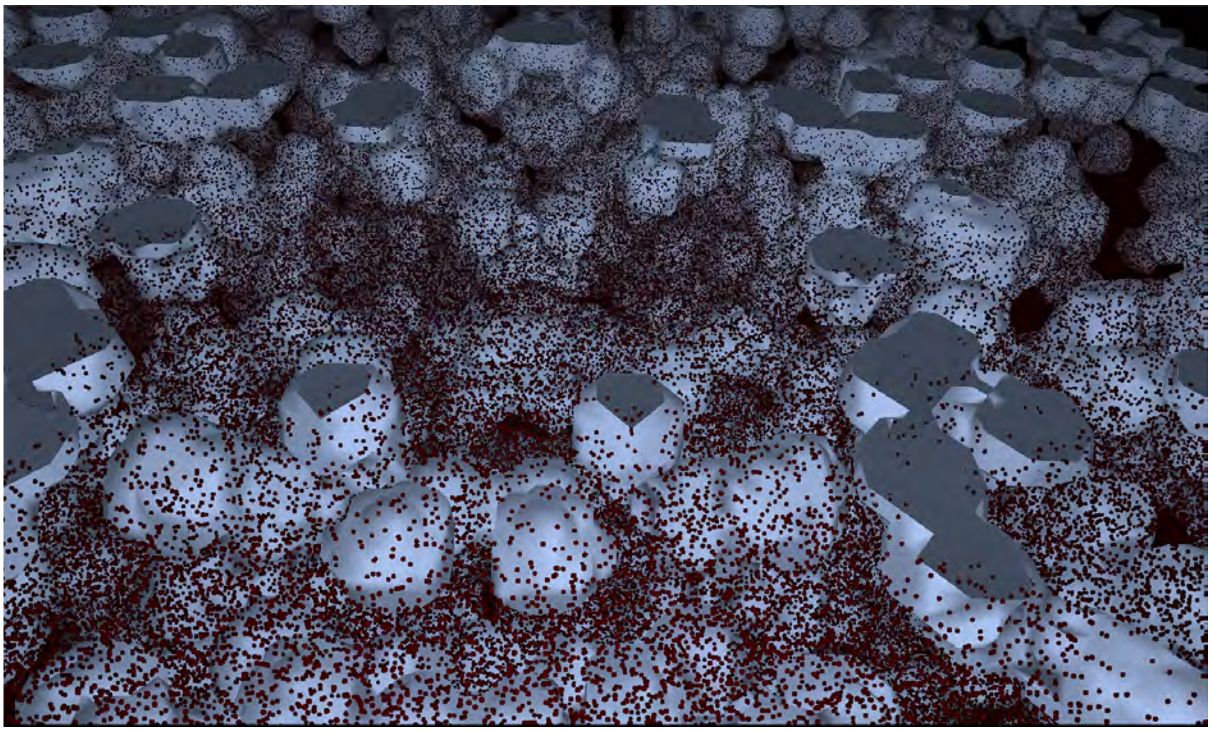

Figure 11.10: This is a live simulation of four million DSMC particles in a system consisting of packed spheres using the same rendering technique as in figure 11.9. The camera is placed outside the system where we clearly see some of the spheres being cut in half because of periodic boundary conditions.

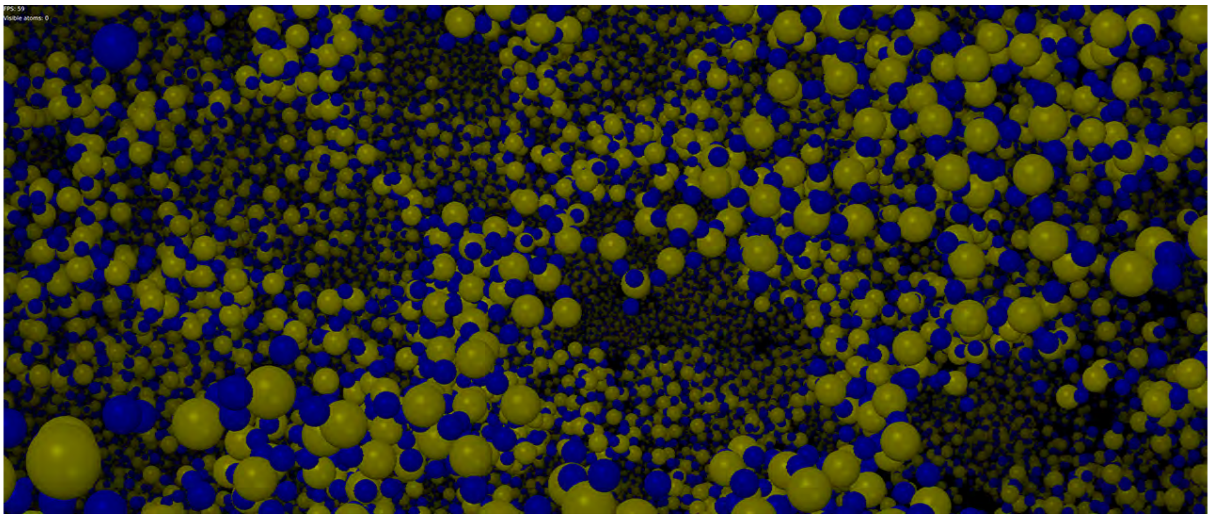

Figure 11.11: Here we see a nanoporous silicate simulated with an MD code developed at the University of Southern California. The system was created by preparing the system in the β-cristobalite state. It was then heated to 4500 K, stretched (increasing the system size) before it was cooled down, quenched, to 300 K making this beautiful pore network. The pores were then filled with water. The total system consists of approximately 400,000 atoms, including water as shown in figure 11.12.

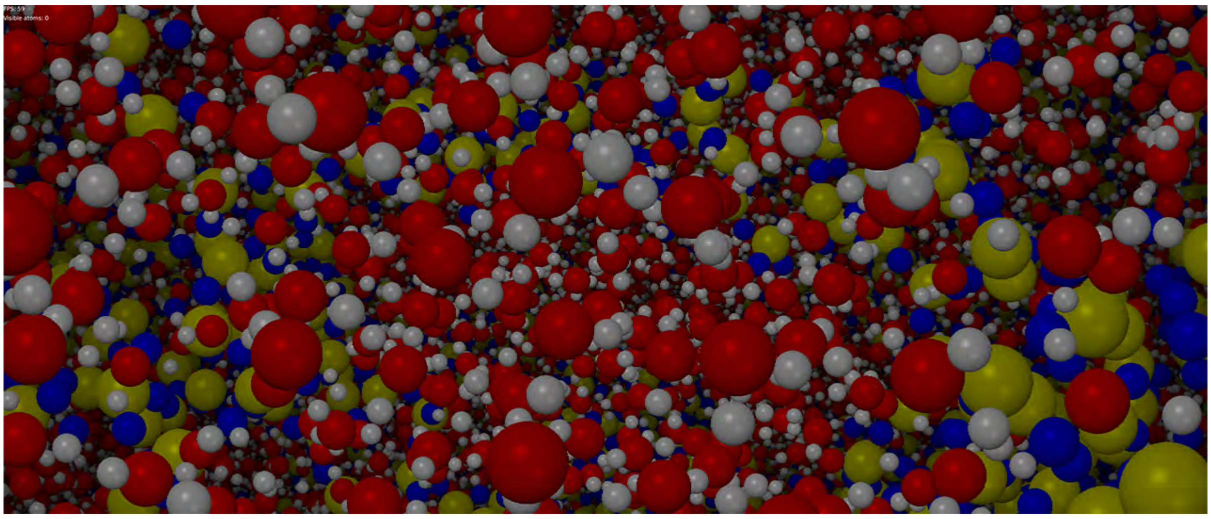

Figure 11.12: This is the same system as in figure 11.11, but with the water visible. We clearly see the water molecules form with one oxygen and two hydrogen atoms. With a non-static picture, with time evolution, we see the hydrogen atoms vibrate and even change which water molecule it is a part of.

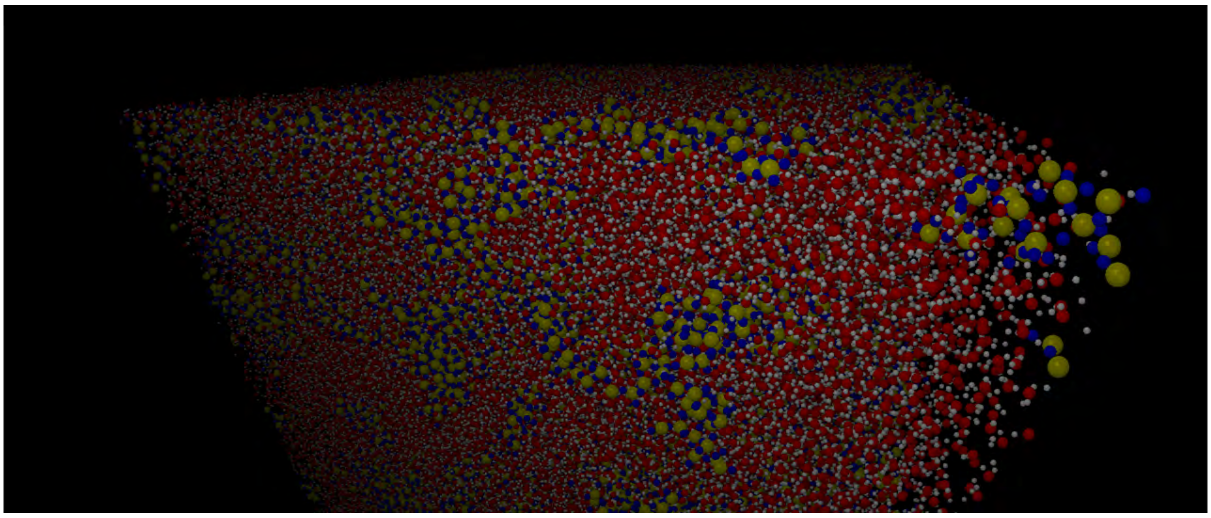

Figure 11.13: Again the same system as in figures 11.11 and 11.12, but with the camera outside the system.

Part IV

Conclusion and discussion

Chapter 12

Summary and conclusion

12.1 Summary

After having explained and discussed all models and the results we have obtained during our work, it is convenient to briefly summarize the thesis. The main physical problem we discussed in the beginning was shale gas extraction. In the fracturing process, the gas is released from within the shales through small pore networks, small channels with diameter from a few nanometers and up. Standard hydrodynamics breaks down at this scale because slip velocity and non-continuum effects, so we needed a model enabling us to study gas dynamics at this scale. In chapter 3 we briefly discussed statistical mechanics and kinetic theory, and used this to derive the Boltzmann equation, the mean free path and the mean collision time. These results were used to justify the DSMC model in chapter 4. We explained how the model is implemented in chapter 5. With the parallel implementation and the voxel-based representation of arbitrary geometries, we managed to achieve the first two goals $a)$ and $b)$ in section 1.3.

In chapter 6, we discussed the simulations with both model validation, performance tests and permeability analysis. The implementation shows a promising scaling performance up to at least 512 processors in addition to computing correct velocity profiles for a large range of Knudsen numbers compared to work of others. We confirmed that the Knudsen correction from section 2.13 predicted the permeability very well for cylinders, a geometry with a well defined Knudsen number. In the case of packed spheres, the Knudsen number is less obvious how to define due to statistical spread in the geometrical formation of the spheres, we found that an expectation value of the pore size could be used to find what we called an estimated Knudsen number (see appendix A). We were able to predict the permeability to the correct order of magnitude, but the geometrical statistical spread was reflected through a spread in the final measured permeability. The ratio between the largest and the smallest permeabilities was a factor two or more.
We then moved on to discuss the second model we have studied, Molecular Dynamics, with an introduction to the model in chapter 7. The numerical implementation was explained in detail in chapter 8. We also introduced a new model for simulating fluids confined by a solid that interacts with realistic atomic forces and remains being a solid with the initial geometry by adding a harmonic oscillator potential on the atoms. The results in chapter 9 showed that

the parallel implementation scaled satisfactory up to at least one thousand processors. We ran the MD program to measure the permeability in a cylinder for different Knudsen numbers, just like we did with DSMC. By simulating a similar system, but with a model based on different physical fundamentals, this is a good way to validate the Knudsen correction for both different length scales and a large range of Knudsen numbers. In MD, the cylinder was 30 times smaller than in DSMC (15 nm vs 450 nm), with results confirming that the Knudsen correction works very well also for pores as small as 15 nanometers. Combining the results from both models, the goals *c*) and *f*) were achieved. We will get back to the Knudsen correction in the discussion section below. We did not manage to implement the water-silica potential in MD as we wanted to do in goal *e*). This would allow us to use MD to study flow in more realistic systems than with the Lennard-Jones potential.
In the last part of the thesis, the custom 3D visualization tool, we first gave an introduction to OpenGL and its purpose in chapter 10. We discussed the basics of graphics programming as well as the more advanced, important parts of the rendering pipeline that allowed us to develop an extremely efficient visualization program to visualize large particle data sets with tens of millions of atoms rendered real-time with a decent frame rate. This was possible by using a combination of billboards and instanced geometry shaders, both explained in chapter 11. We concluded the part with a final show-off gallery in section 11.3. With this, we have achieved the last goal *d*) from section 1.3.

12.2 Discussion

We have used two fundamentally different models to study flow in similar systems. MD is an atomic model computing forces between every atom whereas DSMC is a stochastic particle model based on statistical mechanics. Studying a physical problem this way has a great strength; when two models with different assumptions agree on the results, we have, to a greater extent, reasons to trust them than if it was just one model. But in any model, we make assumptions, and these may produce wrong answers. The main reason to use DSMC rather than MD is of course the reduced computational cost. This points out an important fact; the more physics you include in a model, the more computational expensive it is. This leads to the fundamental problem we always meet before choosing a model; what is the physical problem, and what are the relevant phenomena?

A complete model of shale gas extraction should include physics describing both the gas production in the nanometer pores as well as the large scale flow from the fractures into the drilling hole. Today we do not have any good models covering this huge range in length scales and time scales. On the larger scale, we use continuum mechanics whereas the smaller scale requires models like MD or DSMC. On the smallest scale there are several alternatives to these two. Other popular methods are the lattice Boltzmann (LB), dissipative particle dynamics (DPD), which both can simulate larger systems for lager times. But again, a faster model usually includes less physics. A comparison between DSMC and these models would be interesting.

Both models we have studied have indeed shown promising results, calculating permeabilities in nanopores in agreement with the Knudsen correction, but there might be other important effects we have not included in the models. Such effects are probably not incorporated in the Knudsen correction either, but could become evident in an experiment. Validation of a model is not necessarily a validation of the assumptions. For example, if the surfaces inside the pores have a non-zero net charge, it could significantly affect the fluid flow and the permeability. We can incorporate this in MD by using a more advanced potential. The model we used to create a solid in MD used a harmonic oscillator potential to keep the atoms at approximately the same position as they started, keeping the geometrical form of the solid. While this model includes realistic atomic forces and vibrations inside the solid, deformations and fracturing is now impossible since the atoms are forced in the original formation.

It is also assumed that the gas is trapped both inside already existing pores as well as adsorbed onto the organic material inside the shales. The rate of desorption from the organic material may require additional models releasing the gas from the organic material to the fracture network. These are just a few, but important remarks about the limitations of the models. It is needless to say that they are not complete in any sense. Even if they *were* complete, if they calculated both gas production and flow from the smallest pores into the fracture network, we would not be even close to a complete model of a shale reservoir. The coupling between different length scales - multi scale physics - is still an unsolved problem in physical science.

From the theoretical perspective, we used the Knudsen correction to predict permeabilities in geometries with a closed form solution for the absolute permeability at the no-slip scale. If we work with a geometry without a known absolute permeability, we can measure this in the high density limit, and still use the Knudsen correction to predict permeabilities for dilute gases. The correction factor is a function of the Knudsen number, which is a well defined quantity in simple geometries like a cylinder. However, for more complicated geometries like the packed spheres, the Knudsen number should rather be seen as a distribution than one single number. It is possible, as we discuss in appendix A, to calculate a distribution of Knudsen numbers providing both the mean value and the standard deviation which may be used to develop a *stochastic* Knudsen correction based on the original model. By obtaining statistical information about the geometry of a real shale reservoir, such a theory could in principle enable us to do an economical risk analysis based on the permeability distribution.

12.3 Future work

No matter how much work we do, new ideas pop our minds faster than we manage to complete the old ones. During the past months, through everything we have learned, both ideas about extensions to the models and applications have emerged. First of all, as we discussed in section 4.7, our analysis of whether or not the system has reached a steady state is inadequate. New, automatic methods should be incorporated by, in several regions (for example by using the collision cells in DSMC), looking at expectation values and the variance of physical quantities such as momentum, energy and temperature. Even with a more advanced analysis of if the steady state is reached or not, reaching a steady state may take a long time. In the benchmark we performed and discussed in section 6.2, 512 processors used 545 seconds to proceed 500 timesteps with one billion particles. If the system requires 100k timesteps to reach a steady state, this would require 15k cpu hours just to reach the steady state. With DSMC, it may be possible to use a smaller number of particles (that is, increase the number of effective atoms per simulated particle) while the system reaches a steady state. Then, before the sampling, increase the total number of particles by distributing a number of particles with the velocities and densities computed from the previous simulation, equilibrating this state before starting the sampling.

A detailed study of how the Cercignani-Lampis collision model from section 4.2 affects the permeability would also be a interesting and important work. Here one could for example also use the Lennard-Jones MD model to fire single atoms towards a surface and gather a distribution to compare with the Cercignani-Lampis model.

With the voxelization model of the surface we developed in section 5.1 a few questions quickly arise. What are the effects of the discretization of the geometry? A real, smooth surface does indeed look different than a jagged surface composed by voxels. With this representation, the effective surface area is increased which could affect the flow. A detailed study of these effects should be carried out. Such a representation also requires a lot of memory as the memory requirement scales as $O(N^3)$. To save memory, we can use a sparse voxel octree representation. If a larger group of voxels all share the same value, only one, larger voxel is saved in memory, representing all the smaller voxels.

Since the matrix already is on a discretized grid and the net momentum transfer from the particles is calculated during collisions, it is possible to extend the model to include deformation and cracking which are important processes inside the shales. Production from organic material can also be included by including a diffusion solver on the matrix that can desorb and adsorb gas particles.

For a large number of processors, the amount of work each processor is assigned may not be equal to all the others. This means that a processor that is computing only half the work than another processor, it will spend 50% of its time waiting for the other processors to finish. The solution to this is known as load balancing. An analysis of the system prior to the simulation may distribute the amount of work not by dividing the system into equal *total volumes*, but equal *pore volumes* in which the particles are. The amount of work is proportional to the number of particles a processor has. A last, but very interesting task we would like to do is simulate gas through a nanoporous media from a real material. Such data can be obtained with a technique called focused ion beam scanning electron microscopy (FIB-SEM).

Part V

Appendices

Appendix A

Knudsen number of a packed spheres system

If we want to use the Knudsen correction factor (equation (2.19)) to predict permeabilities in a nanoporous system consisting of packed spheres, we need to derive some statistical properties of the system giving us enough information to calculate the expected Knudsen number. Now, consider a volume V consisting of N points (sphere centers) placed randomly and independently in the system. Note that this allows spheres with radius r to overlap. The density of points is of course given by

$$n = \frac{N}{V}. \tag{A.1}$$

The probability of placing a sphere center in a volume v is v/V, yielding the probability of *not* placing a sphere center in that same volume $(1 - v/V)$. If we now place N such points, the probability $P_0(N)$ of not having placed *any* points in that volume is given by

$$P_0(N) = \left[1 - \frac{nv}{N}\right]^N, \tag{A.2}$$

where we have used that $v/V = nv/nV = nv/N$ using equation (A.1). In the limit where $N \to \infty$ and $V \to \infty$, keeping the density n and v constant, the probability approaches

$$P_0 = \lim_{N\to\infty} \left[1 - \frac{nv}{N}\right]^N = \exp(-nv). \tag{A.3}$$

We can use this to compute the probability of finding *no* sphere centers within a distance l from a point

$$P_0(l) = \exp\left(-\frac{4n\pi}{3} l^3\right), \tag{A.4}$$

where we just used the volume of a sphere $v = 4/3\pi l^3$. This is the cumulative probability distribution of finding *no* sphere centers within a distance l, so the inverse problem, the probability of finding *at least one* sphere center at some distance $x < l$ is now easy to calculate

$$P(x < l) = 1 - \exp\left(-\frac{4n\pi}{3}l^3\right). \tag{A.5}$$

This is also a cumulative distribution function which we can differentiate to find the probability distribution of distances l

$$p(l) = 4\pi n l^2 \exp\left(-\frac{4n\pi}{3}l^3\right). \tag{A.6}$$

$p(l)\mathrm{d}l$ is the probability of finding a sphere center within the range $[l, l+\mathrm{d}l)$. In this calculation, we are only interested in the distribution of distances l *given that we are not inside any spheres.* So we define a new distribution function

$$q(l) = \begin{cases} 0 & \text{if } l < r, \\ Mp(l) & \text{if } l \geq r, \end{cases} \tag{A.7}$$

where M is the normalization constant for $q(l)$ (the area must be less than one now since we removed all the points from zero to r). We find M by integrating

$$1 = M\int_0^\infty q(l)\mathrm{d}l = M\int_r^\infty p(l)\mathrm{d}l \tag{A.8}$$

$$= M4\pi n\int_r^\infty l^2 \exp\left(-\frac{4n\pi}{3}l^3\right)\mathrm{d}l \tag{A.9}$$

$$= M\exp\left(-\frac{4n\pi}{3}r^3\right), \tag{A.10}$$

which gives $M = \exp\left(\frac{4n\pi}{3}r^3\right)$. Avoiding the points being inside the spheres is of course the same as choosing only the points that are in the pore space. The probability of randomly choosing such a point is actually the porosity ϕ (remember, the porosity is pore space volume divided by total volume). It is found by using equation (A.4)

$$\phi = \exp\left(-\frac{4n\pi}{3}r^3\right), \tag{A.11}$$

which we recognize as M^{-1}. We can then rewrite $q(l)$

$$q(l) = \begin{cases} 0 & \text{if } l < r, \\ 4n\pi\phi^{-1}l^2\exp\left(-\frac{4n\pi}{3}l^3\right) & \text{if } l \geq r. \end{cases} \tag{A.12}$$

Now it would be interesting to find the average distance to the sphere centers. It is found by calculating

$$\langle l\rangle = \int_r^\infty lq(l)\mathrm{d}l = \frac{4n\pi}{\phi}\int_r^\infty l^3\exp\left(-\frac{4n\pi}{3}l^3\right)\mathrm{d}l \tag{A.13}$$

$$= \sqrt[3]{\frac{3}{4n\pi}}\frac{1}{\phi}\Gamma\left(\frac{4}{3}, \frac{4n\pi}{3}r^3\right), \tag{A.14}$$

where $\Gamma(a, x)$ is the incomplete gamma function defined as

$$\Gamma(a, x) = \int_x^\infty t^{a-1} \exp(-t) \mathrm{d}t. \tag{A.15}$$

We can use equation (A.11) to replace the dependency of the number density n with the porosity ϕ and sphere radius r by solving for n

$$n = \frac{3}{4\pi r^3} \ln \phi^{-1}, \tag{A.16}$$

and then insert this into equation (A.14) to obtain

$$\langle l(r, \phi) \rangle = \frac{r}{\phi} \left[\ln \phi^{-1}\right]^{-\frac{1}{3}} \Gamma\left(\frac{4}{3}, \ln \phi^{-1}\right). \tag{A.17}$$

What we really want is the distances to the *surfaces* of the spheres, $d = (l - r)$ which is easy to find

$$\langle d(r, \phi) \rangle = \langle l(r, \phi) \rangle - r = \frac{r}{\phi} \left[\ln \phi^{-1}\right]^{-\frac{1}{3}} \Gamma\left(\frac{4}{3}, \ln \phi^{-1}\right) - r \tag{A.18}$$

$$= r \left[\phi^{-1} \left[\ln \phi^{-1}\right]^{-\frac{1}{3}} \Gamma\left(\frac{4}{3}, \ln \phi^{-1}\right) - 1\right]. \tag{A.19}$$

In the expression for the Knudsen number ($\mathrm{Kn} = \lambda/L$), we will use this quantity instead of L as our best estimate, or average value, of the channel diameter. We then define the estimated Knudsen number $\mathrm{Kn}(r, \phi)^*$ as

$$\mathrm{Kn}(r, \phi)^* = \frac{\lambda}{\langle l(r, \phi) \rangle} \tag{A.20}$$

$$= \frac{\lambda}{r \left[\phi^{-1} \left[\ln \phi^{-1}\right]^{-\frac{1}{3}} \Gamma\left(\frac{4}{3}, \ln \phi^{-1}\right) - 1\right]}. \tag{A.21}$$

Appendix B

The Liouville operator and time integrators

The derivation of time integrator schemes are usually done in a mathematical sense using Taylor expansions. Given a Taylor expansion $f(x+h) = f(x) + hf'(x) + h^2/2f''(x) + ...$, we often truncate at some term, yielding a truncation error $O(h^n)$. In physics, there are other properties of a system of which the error doesn't scale as the truncation error. A typical example is the energy of a particle system. Two different time integrators may have the same truncation error, but have very different long term behavior if one has a drift in the energy whereas the other doesn't.

There are other properties that might be difficult to measure through a mathematical analysis of the Taylor expansion that may be more important than the truncation error itself. In the Hamiltonian formulation of classical mechanics, we obtain the equations of motion through the energy operator $\mathbf{H} = \mathbf{T} + \mathbf{V}$, where $\mathbf{T}$ and $\mathbf{V}$ are operators measuring the kinetic and potential energy. Using the physical description of a system can help developing better integration schemes.

In this chapter we will address the equations of motion by using the Liouville operator to derive a way to create time integration schemes. We start by looking at the phase space coordinates and define the Liouville operator in section B.1. We then define the time evolution operator and split the Liouville operator into two operators; one operator acting on the positions and one acting on the momenta. These operators do not commute, so we use the Trotter identity to introduce time discretization and derive the Velocity Verlet algorithm, which is the one used in the Molecular Dynamics code. We then do an analysis of the mathematical error to find the local error (which is the same as the truncation error) in section B.3. This derivation is done as in [11].

B.1 Liouville operator

The physical system consists of N particles, each having three positions and three momenta defining the phase space point $(\mathbf{r},\mathbf{p})$. Now assume some function of these variables $f(\mathbf{r}(t),\mathbf{p}(t)) = f(t)$ (the function is indirectly a function of time) that has the time derivative

$$\dot{f}(t) = \dot{\mathbf{r}}\frac{\partial f(t)}{\partial \mathbf{r}} + \dot{\mathbf{p}}\frac{\partial f(t)}{\partial \mathbf{p}} \equiv i\hat{\mathbf{L}}f(t), \tag{B.1}$$

where we have defined the Liouville operator

$$i\hat{\mathbf{L}} = \dot{\mathbf{r}}\frac{\partial}{\partial \mathbf{r}} + \dot{\mathbf{p}}\frac{\partial}{\partial \mathbf{p}}. \tag{B.2}$$

This allows us to define the time evolution operator $\hat{\mathcal{U}}(t)$

$$f(t) = \hat{\mathcal{U}}(t)f(0) = e^{i\hat{\mathbf{L}}t}f(0), \tag{B.3}$$

which is easily verified

$$\dot{f}(t) = i\hat{\mathbf{L}}\left[e^{i\hat{\mathbf{L}}t}f(0)\right] = i\hat{\mathbf{L}}f(t). \tag{B.4}$$

If we now split the Liouville operator into two parts

$$i\hat{\mathbf{L}} = i\hat{\mathbf{L}}_r + i\hat{\mathbf{L}}_p, \tag{B.5}$$

so that

$$i\hat{\mathbf{L}}_r = \dot{\mathbf{r}(0)}\frac{\partial}{\partial \mathbf{r}}. \tag{B.6}$$

Let us see what this operator can do if we insert it into equation (B.3) and expand the exponential

$$\begin{aligned} f(t) &= \exp\left(i\hat{\mathbf{L}}_r t\right) f(0) && \text{(B.7)}\\ &= \exp\left(\dot{\mathbf{r}}t\frac{\partial}{\partial \mathbf{r}}\right) f(0) && \text{(B.8)}\\ &= \sum_{n=0}^{\infty}\frac{(\dot{\mathbf{r}}(0)t)^n}{n!}\frac{\partial^n}{\partial \mathbf{r}^n}f(0) && \text{(B.9)}\\ &= f\left[\left(\mathbf{r}(0)+\dot{\mathbf{r}}(0)t\right),\mathbf{p}(0)\right]. && \text{(B.10)} \end{aligned}$$

It does just what we expect it to do, it is a displacement operator, moving the points in the phase space according to their time derivative. The momentum Liouville operator does of course exactly the same, so that by applying the total time evolution operator, we do indeed get

$$f(t) = e^{i\hat{\mathbf{L}}t}f(0) = f\left[\left(\mathbf{r}(0)+\dot{\mathbf{r}}(0)t\right),\left(\mathbf{p}(0)+\dot{\mathbf{p}}(0)t\right)\right]. \tag{B.11}$$

If we were to use this in a simulation, we normally do not apply the full operator at the same time, we might first treat the positions, then the momenta. So ideally, we would want to first apply one operator, then the next one

$$e^{i\hat{\mathbf{L}}} = e^{i\hat{\mathbf{L}}_p + i\hat{\mathbf{L}}_r} \neq e^{i\hat{\mathbf{L}}_p}e^{i\hat{\mathbf{L}}_r}, \tag{B.12}$$

but this is not the case since the operators do necessarily commute. However, for two operators **A** and **B**, we can use the *Trotter identity*[11]

$$e^{A+B} = \lim_{N\to\infty} \left(e^{A/2M} e^{B/M} e^{A/2M} \right)^N , \tag{B.13}$$

which can be truncated

$$e^{A+B} = \left(e^{A/2N} e^{B/N} e^{A/2N} \right)^N e^{O(1/N^2)}. \tag{B.14}$$

This can be used to derive different time integrator schemes. We will now derive the Velocity Verlet scheme which is used in the Molecular Dynamics code.

B.2 Derivation of the Velocity Verlet algorithm

The truncated Trotter identity is neat, we can replace A with the $i\hat{\mathbf{L}}_p$ and B with $i\hat{\mathbf{L}}_r$

$$\frac{i\hat{\mathbf{L}}_p}{M} \equiv \Delta t \dot{\mathbf{p}}(0) \frac{\partial}{\partial \mathbf{p}} \tag{B.15}$$

$$\frac{i\hat{\mathbf{L}}_r}{M} \equiv \Delta t \dot{\mathbf{r}}(0) \frac{\partial}{\partial \mathbf{r}} \tag{B.16}$$

defining the timestep $\Delta t = t/N$. The truncated time evolution operator now reads

$$\hat{\mathcal{U}}(t) = \left(e^{i\hat{\mathbf{L}}_p \Delta t/2} e^{i\hat{\mathbf{L}}_r \Delta t} e^{i\hat{\mathbf{L}}_p \Delta t/2} \right)^N \tag{B.17}$$

where we identify one timestep iteration

$$\hat{\mathcal{U}}(\Delta t) = e^{i\hat{\mathbf{L}}_p \Delta t/2} e^{i\hat{\mathbf{L}}_r \Delta t} e^{i\hat{\mathbf{L}}_p \Delta t/2}. \tag{B.18}$$

If we now apply this operator on $f(0)$, we first apply the rightmost operator

$$e^{i\hat{\mathbf{L}}_p \Delta t/2} f\left[\mathbf{p}(0), \mathbf{r}(0)\right] = f\left\{ \mathbf{r}(0), \left[\mathbf{p}(0) + \frac{\Delta t}{2} \dot{\mathbf{p}}(0) \right] \right\}, \tag{B.19}$$

before applying $\exp(i\hat{\mathbf{L}}_r t)$

$$e^{i\hat{\mathbf{L}}_r t} f\left\{ \mathbf{r}(0), \left[\mathbf{p}(0) + \frac{\Delta t}{2} \dot{\mathbf{p}}(0) \right] \right\} \tag{B.20}$$

$$= f\left\{ \left[\mathbf{r}(0) + \Delta t \dot{\mathbf{r}}(\Delta t/2) \right], \left[\mathbf{p}(0) + \frac{\Delta t}{2} \dot{\mathbf{p}}(0) \right] \right\}. \tag{B.21}$$

Our last operator gives the final result

$$f\left\{ \left[\mathbf{r}(0) + \Delta t \dot{\mathbf{r}}(\Delta t/2) \right], \left[\mathbf{p}(0) + \frac{\Delta t}{2} \dot{\mathbf{p}}(0) + \frac{\Delta t}{2} \dot{\mathbf{p}}(\Delta t) \right] \right\}. \tag{B.22}$$

These steps can be summarized as one usually does with time integrators

$$\mathbf{v}(\Delta t/2) = \mathbf{v}(0) + \frac{\mathbf{F}(0)}{m}\frac{\Delta t}{2} \tag{B.23}$$

$$\mathbf{r}(\Delta t) = \mathbf{r}(0) + \mathbf{v}(\Delta t/2)\Delta t \tag{B.24}$$

$$\mathbf{v}(\Delta t) = \mathbf{v}(\Delta t/2) + \frac{\mathbf{F}(\Delta t)}{m}\frac{\Delta t}{2}, \tag{B.25}$$

where we have replaced $\dot{\mathbf{p}}$ with the equivalent $(\mathbf{F}/m)$ and $\dot{\mathbf{r}}$ with $\mathbf{v}$ which is valid if the forces can be calculated from the position. These steps are called the Velocity Verlet algorithm and has, as we now will see, an error $O(\Delta t^3)$.

B.3 Truncation error

During one timestep iteration, we approximate the Liouville operator

$$e^{i\hat{\mathbf{L}}\Delta t} \approx e^{i\hat{\mathbf{L}}_p\Delta t/2}e^{i\hat{\mathbf{L}}_r\Delta t}e^{i\hat{\mathbf{L}}_p\Delta t/2} = e^{i\hat{\mathbf{L}}\Delta t+\hat{\epsilon}}, \tag{B.26}$$

where we have introduced the *error operator* $\hat{\epsilon}$ that represents our truncation error. These are linear operators on which we can use the Campbell-Baker-Hausdorff expansion of general, non-commuting linear operators

$$e^{\lambda\hat{\mathbf{A}}}e^{\lambda\hat{\mathbf{B}}} = e^{\lambda\hat{\mathbf{A}}+\lambda\hat{\mathbf{B}}+\frac{\lambda^2}{2}[\hat{\mathbf{A}},\hat{\mathbf{B}}]+\frac{\lambda^3}{12}[\hat{\mathbf{A}},[\hat{\mathbf{A}},\hat{\mathbf{B}}]]+\frac{\lambda^3}{12}[\hat{\mathbf{B}},[\hat{\mathbf{A}},\hat{\mathbf{B}}]]+\dots} \tag{B.27}$$

together with

$$e^{\hat{\mathbf{A}}}e^{\hat{\mathbf{B}}} = e^{\hat{\mathbf{B}}+[\hat{\mathbf{A}},\hat{\mathbf{B}}]+\frac{1}{2!}[\hat{\mathbf{A}},[\hat{\mathbf{A}},\hat{\mathbf{B}}]]+\frac{1}{3!}[\hat{\mathbf{A}},[\hat{\mathbf{A}},[\hat{\mathbf{A}},\hat{\mathbf{B}}]]]+\dots}e^{\hat{\mathbf{A}}} \tag{B.28}$$

to find the leading term in $\hat{\epsilon}$

$$-(\Delta t)^3\left(\frac{1}{24}[i\hat{\mathbf{L}}_r,[i\hat{\mathbf{L}}_r,i\hat{\mathbf{L}}_p]] + \frac{1}{12}[i\hat{\mathbf{L}}_p,[i\hat{\mathbf{L}}_r,i\hat{\mathbf{L}}_p]]\right), \tag{B.29}$$

where we see that Δt^3 is the global truncation error in the Velocity Verlet algorithm after n timesteps. The local truncation error is then Δt^4.

Appendix C

Derivation of pressure in Molecular Dynamics

A substance in a Molecular Dynamics simulation does not generally satisfy the ideal gas equation of state. The pressure has the general form

$$P = k_B T \sum_{m=1}^{\infty} \rho_n^m B_m(T), \tag{C.1}$$

where the functions $B_m(t)$ are called the virial coefficients with $B_1(T) = 1$, yielding the ideal gas pressure [23]. We will now derive an expression for the pressure of this form by using Clausius' virial function. Assume that we have the positions of all atoms, $\mathbf{r}$, and define

$$W(\mathbf{r}) = \sum_{n=1}^{N} \mathbf{r}_n \cdot \mathbf{F}_n^{\mathrm{TOT}}, \tag{C.2}$$

where $\mathbf{F}_n^{\mathrm{TOT}}$ is the total force acting on atom n, including external forces. We assume equilibrium, so that the kinetic energy has reached an approximately constant value (it will of course fluctuate with standard deviation $1/\sqrt{N}$ as usual). We measure the statistical average of W by computing (using $\mathbf{F} = m\mathbf{a} = m\ddot{\mathbf{r}}$)

$$\langle W \rangle = \lim_{t\to\infty} \frac{1}{t} \int_0^t d\tau \sum_{n=1}^{N} m_n \mathbf{r}_n(\tau) \cdot \ddot{\mathbf{r}}_n(\tau). \tag{C.3}$$

Integrating by parts gives

$$\langle W \rangle = -\lim_{t\to\infty} \frac{1}{t} \int_0^t d\tau \sum_{i=1}^{N} m_i |\dot{\mathbf{r}}_i(\tau)|^2 = -2\langle E_k \rangle = -3Nk_bT, \tag{C.4}$$

by using equipartition. Now, assume that the atoms live inside a parallelepipedic container of size L_x, L_y, L_z with hard walls (they don't move) and origo in one of its corners. If we divide the force into external and interatomic forces, $\mathbf{F}_n^{\mathrm{TOT}} = \mathbf{F}_n + \mathbf{F}_n^{\mathrm{EXT}}$, and assume that the external forces are forces from the container (no gravity or electric fields), we can calculate W^{EXT}. The atoms near the walls apply a pressure on the wall $P = F/A$. As an example, we look at all the atoms that are near the wall located at $x = L_x$. The virial function gives

$$W_x^{\mathrm{EXT}} = \sum_{n=1}^{N_x} \mathbf{r}_n \cdot \mathbf{F}_n^{\mathrm{EXT}}, \tag{C.5}$$

where n now sums over all atoms that are near the container wall at $x = L_x$. The position vectors are $\mathbf{r}_n = (L_x, y_n, z_n)$ (for different y_n and z_n) and the force has only a component normal to the wall $F_n^{\mathrm{EXT}} = 1/N_x(-PL_yL_z, 0, 0)$. We then get

$$W_x^{\mathrm{EXT}} = -L_xPL_yL_z = -PV, \tag{C.6}$$

and by doing the same for the other dimensions, we get

$$W = -3PV. \tag{C.7}$$

Inserting this result into (C.4) yields

$$\left\langle \sum_{n=1}^{N} \mathbf{r}_n \cdot \mathbf{F}_n \right\rangle - 3PV = -3Nk_bT \tag{C.8}$$

which can be rearranged to

$$PV = Nk_bT - \frac{1}{3} \left\langle \sum_{n=1}^{N} \mathbf{r}_n \cdot \mathbf{F}_n \right\rangle. \tag{C.9}$$

Using this result, we can define the pressure

$$P = \rho_n k_b T - \frac{1}{3V} \left\langle \sum_{n=1}^{N} \mathbf{r}_n \cdot \mathbf{F}_n \right\rangle, \tag{C.10}$$

where ρ_n is the number density.

Appendix D

Workflow and Python scripting

When we are running simulations to study flow in nanoporous media, one single experiment consists of more than just starting the program and waiting for the result. A typical simulation scheme could be

1. Prepare geometry
2. Initialize system
3. Reach a steady state
4. Sample statistics.

Some of these steps aren't even performed by the same C++ program, so we have developed a scripting framework in Python wrapping all the functionality from the different C++ programs into one scripting framework. This simplifies the workflow all the way from the very idea of some physical problem to running the simulation to study it. In this appendix we will discuss the Python framework and show the workflow of how to study problem, from idea to simulation, using DSMC. First, we quickly explain the basic principles of the Python framework.

D.1 The Python framework

The framework is a Python class with a lot of convenience methods built-in. Its main job is to compile the C++ programs and simplify the generation of config files that are used by the C++ DSMC program. The class is called DSMC and is best explained by a code example, see listing D.1. This allows us to quickly create a script running a full simulation consisting on several steps in a short amount of time.

```
class DSMC:
def __init__(self, compiler = "mpic++", dt=0.001, nx=1, ny=1, nz=1):
    # Physical parameters
    self.unit_converter = DSMC_unit_converter(self)
    self.temperature = unit_converter.temperature_from_si(T=300)
    self.density = unit_converter.density_from_si(rho=1.0e25)
    self.diam = unit_converter.length_from_si(3.62e-10)
    self.mass = unit_converter.mass_from_si(6.63e-26)

    self.dt = dt
    self.compiler = compiler
    self.nx = nx
    self.ny = ny
    self.nz = nz

    self.timesteps = 10000
    self.surface_interaction = "thermal"
    # ...

  def save_state(self, path):
    # ...

  def load_state(self, path):
    # ...

  def prepare_new_system(self):
    # ...

  def apply_pressure_gradient_percentage(self, factor_of_ideal_gas_pressure
      ):
    # ...

  def create_config_file(self):
    original_file = open("program/000_config_files/dsmc.ini,'r');
    output_file   = open('dsmc.ini','w');

    for line in original_file:
      line = line.replace('__dt__',str(self.dt) )
      line = line.replace('__density__',str(self.density) )
      # ...
      output_file.writelines(line)
    # ...

  def compile(self):
    # ...

  def run(self):
    # ...
}
```

Listing D.1: dsmcconfig.py. This program illustrates how the framework script is created. A lot of convenience functions are built-in so that running a simulation is a simple process. The create_config_file() function replaces the string __value__ in a dummy config file with the real value stored in the class.

D.2 From idea to simulation

Create the geometry

The DSMC program package we have developed is an excellent tool to study flow of dilute gases in complex nanoporous media. So what we usually do is to come up with the idea of some interesting geometry to study. This could either be a real sample dataset, a mathematical description or some other creative way to create the geometry. An example could be the packed spheres discussed in section 6.4 where each sphere is placed out randomly inside the volume defined by the $N_x \times N_y \times N_z$ voxel grid. All the voxels within a given distance (the sphere radius) from the sphere center are marked as solid voxels. Any other geometry can made with this technique with the code in the *complexgeometry.cpp* file.

Initialize system

Once we have the geometry of the system we would like to study, we need to insert gas particles inside the system. The gas could in principle have different densities based on position or other properties, but in this thesis we are placing the particles uniformly in the pore volume, i.e. in voxels marked as empty. We choose velocities from the Maxwell-Boltzmann distribution (equation (3.45)) so that the gas has the temperature we want.

Reach a steady state

Now that we have our gas particles inside the (possibly) complex geometry, we want to apply a pressure gradient inducing flow. In the beginning, all the particles are (on average) at rest, so we should wait until we have reached a steady state before we start sampling statistics. Methods to determine whether or not a system has reached a steady state was discussed in section 4.7.

Sample statistics

If we assume that the system finally has reached a steady state, we can start measuring the physical quantities discussed in section 4.4. The main focus in this thesis has been to study the permeability for different geometries.

D.3 The run script - example.py

Finally, after having defined all the steps in our scheme, we can gather everything into one single Python script. This example script shows how to perform all the steps discussed above.

```python
from dsmcconfig import *
from dsmc_unit_converter import *
from dsmc_geometry_config import *
nx = 2 # Number of processors in x-direction
ny = 2 # Number of processors in y-direction
nz = 2 # Number of processors in z-direction
program = DSMC(dt=0.001, nx=nx, ny=ny, nz=nz)
dsmc = program.compile(name="main")
geometry = DSMC_geometry(program)
unit_converter = DSMC_unit_converter(program)
```

```
# Save geometry files to this directory
geometry.binary_output_folder = "../geometries/example_geometry/"
program.world = geometry.binary_output_folder

# Set the dimensionality of the voxel grid
geometry.num_voxels_x = 128
geometry.num_voxels_y = 128
geometry.num_voxels_z = 128

#geometry.create_cylinders(radius=0.05, num_cylinders_per_dimension = 4)
geometry.create_packed_spheres(radius = 0.02, spheres_type = 1,
    wanted_porosity = 0.5)

# Specify the array size (maximum number of particles)
program.max_molecules_per_node = 1.5e6

# Set system size (in micro meters)
program.Lx = 1
program.Ly = 1
program.Lz = 1

# Select density (Kn = mean_free_path / length)
program.density = unit_converter.density_from_knudsen_number(
    knudsen_number=1.0, length=program.Ly)

# Prepare a new simulation, delete old files
program.reset()

# Select how many atoms each particle represents
program.atoms_per_molecule = 1

# Set the number of collision cells
program.set_number_of_cells(geometry, particles_per_cell=100)

# Initialize gas particles
program.prepare_new_system()
program.run(dsmc)

# Apply pressure gradient
program.apply_pressure_gradient_percentage(factor=2.0)

# Select low statistics measuring interval
program.statistics_interval = 10000
# Select num timesteps to reach steady state
program.timesteps = 500000

program.create_config_file()
program.run(dsmc)

# Select high statistics measuring interval
program.statistics_interval = 100
# Select num timesteps to sample
program.timesteps = 100000

program.create_config_file()
program.run(dsmc)
```

```
}
```

Listing D.2: example.py

Appendix E

Physical units

The choice of units does of course not change the physical behavior. It is only a scaling of the the physical quantities so that their numerical values become more convenient. For example, it is common to choose a unit of length L_0 so that one unit of length is a typical value in the physical system. There are however only a few *free* choices of units of which the other units follow from. In this appendix, we discuss the choices of units in both the MD and DSMC simulations and derive the other units.

E.1 Choice of units in MD

Here we use the so-called MD units. A convenient consequence of these units are that the parameters and masses in the Lennard-Jones force can be factored out which gives simpler calculations. The units we choose are

$$\begin{aligned}\text{Length } L_0 &= 3.405 \times 10^{-10} \text{ m}, && \text{(E.1)}\\ \text{Mass } m_0 &= 1.66 \times 10^{-27} \text{ kg}, && \text{(E.2)}\\ \text{Energy } E_0 &= 1.65 \times 10^{-21} \text{ J}, && \text{(E.3)}\\ \text{Temperature } T_0 &= 119.6 \text{ K} && \text{(E.4)}\end{aligned}$$

E.2 Choice of units in DSMC

In DSMC, we use the same initial units as in MD, but with another unit of length since the systems normally are a few orders of magnitude larger. Here we use

$$\text{Length } L_0 = 1.0 \times 10^{-6} \text{ m}, \tag{E.5}$$

$$\text{Mass } m_0 = 1.66053886 \times 10^{-27} \text{ kg}, \tag{E.6}$$

$$\text{Energy } E_0 = 1.65088 \times 10^{-21} \text{ J}, \tag{E.7}$$

$$\text{Temperature } T_0 = 119.6 \text{ K} \tag{E.8}$$

Note that the mass, energy and the Boltzmann constant are equal in both MD and DSMC.

E.3 Derivation of the other units

The other units can be derived from the four basis units through relations like $E = mc^2$ and $P = F/A$. Together, these physical formulas form a set of equations that can be solved for each physical quantity. We find the time unit t_0 through $E = mc^2$

$$E0 = m_0 \frac{L_0^2}{t_0^2} \tag{E.9}$$

$$t_0 = L_0 \sqrt{\frac{m_0}{E_0}}. \tag{E.10}$$

The force unit F_0 is found by using Newton's second law

$$F_0 = \frac{m_0 L_0}{t_0^2} = \frac{E_0}{L_0} \tag{E.11}$$

We find the temperature T_0 by using that we chose the Boltzmann constant to be 1.0, which gives

$$T_0 = \frac{E_0}{k_B}. \tag{E.12}$$

The pressure is found through $P = F/A$

$$P_0 = \frac{F_0}{L_0^2} = \frac{E_0}{L_0^3}. \tag{E.13}$$

Now we have all all the conversion factors between SI units and MD/DSMC units. The programs are written so that all input values and internal variables are in the MD/DSMC units, but we have a simple unit converter script that can transform any physical value both to and from SI units. The conversion script can be found in listing E.1.

```
from dsmcconfig import *
from math import sqrt, pi
```

```
class DSMC_unit_converter:
  def __init__(self, dsmc):
    self.dsmc = dsmc
    self.m0 = 1.66053886e-27  # si
    self.L0 = 1e-6            # si
    self.E0 = 1.65088e-21     # si
    self.kb = 1.3806503e-23   # si

    self.t0 = self.L0*sqrt(self.m0/self.E0)
    self.F0 = self.E0/self.L0
    self.T0 = self.E0/self.kb
    self.P0 = self.m0/(self.t0**2*self.L0)
    self.v0 = self.L0/self.t0
    self.a0 = self.v0/self.t0
    self.visc0 = self.P0*self.t0
    self.diff0 = self.L0**2/self.t0
    self.perm0 = self.L0**2
    self.number_density0 = 1.0/(self.L0**3)

  def pressure_to_si(self, P):
    return P*self.P0

  def pressure_from_si(self, P):
    return P/self.P0

  def temperature_to_si(self, T):
    return T*self.T0

  def temperature_from_si(self, T):
    return T/self.T0

  # All the other physical quantities can be calculated like this.
```

Listing E.1: dsmcUnitConverter.py

Bibliography

[1] Francis J Alexander and Alejandro L Garcia. “The direct simulation Monte Carlo method”. In: *Computers in Physics* 11.6 (1997), p. 588.

[2] Francis J Alexander, Alejandro L Garcia, and Berni J Alder. “Cell size dependence of transport coefficients in stochastic particle algorithms”. In: *Physics of Fluids* 10 (1998), p. 1540.

[3] Michael P Allen and Dominic J Tildesley. *Computer simulation of liquids*. Oxford University Press, 1989.

[4] George K Batchelor. *An introduction to fluid dynamics*. Cambridge University Press, 2000.

[5] PC Carman. “Fluid flow through granular beds”. In: *Transactions-Institution of Chemical Engineeres* 15 (1937), pp. 150–166.

[6] Evgeni V Chernyaev. “Marching cubes 33: Construction of topologically correct isosurfaces”. In: *Report CN/95-17* (1995).

[7] Faruk Civan. “Effective correlation of apparent gas permeability in tight porous media”. In: *Transport in Porous Media* 82.2 (2010), pp. 375–384.

[8] TG Cowling. “On the Cercignani-Lampis formula for gas-surface interactions”. In: *Journal of Physics D: Applied Physics* 7.6 (1974), p. 781.

[9] Michael A Day. “The no-slip condition of fluid dynamics”. In: *Erkenntnis* 33.3 (1990), pp. 285–296.

[10] Mark W Denny. *Air and water: the biology and physics of life’s media*. Princeton University Press, 1993.

[11] Daan Frenkel and Berend Smit. *Understanding molecular simulation: from algorithms to applications*. Academic Press, 2001.

[12] Alejandro L Garcia. *Numerical methods for physics*. 2nd edition. Prentice Hall, New Jersey, 2000.

[13] Alejandro L Garcia and Florence Baras. “Direct simulation Monte Carlo: Novel applications and new extensions”. In: *Proceedings of the 3rd Workshop on Modelling of Chemical Reaction Systems, Heidelberg*. 1997.

[14] Nicolas G Hadjiconstantinou. “Analysis of discretization in the direct simulation Monte Carlo”. In: *Physics of Fluids* 12 (2000), p. 2634.

[15] George E Karniadakis, Ali Beðskèok, and Narayana Rao Aluru. *Microflows and nanoflows*. Vol. 29. Springer, 2005.

[16] LJ Klinkenberg. “The permeability of porous media to liquids and gases”. In: *Drilling and production practice* (1941).

[17] James C Maxwell. “On stresses in rarified gases arising from inequalities of temperature”. In: *Philosophical Transactions of the Royal Society of London* 170 (1879), pp. 231–256.

[18] James A McLennan. *Introduction to nonequilibrium statistical mechanics*. Prentice Hall, Englewood Cliffs, 1989.

[19] Donald Allan McQuarrie et al. *Statistical thermodynamics*. Harper & Row New York, 1973.

[20] David L Morris, Lawrence Hannon, and Alejandro L Garcia. “Slip length in a dilute gas”. In: *Physical Review A* 46.8 (1992), p. 5279.

[21] Leech Jon Munshi Aaftab. *OpenGL ES Common Profile Specification Version 2.0.25 (Full Specification)*. 2010. URL: http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf.

[22] Taku Ohwada, Yoshio Sone, and Kazuo Aoki. “Numerical analysis of the Poiseuille and thermal transpiration flows between two parallel plates on the basis of the Boltzmann equation for hard-sphere molecules”. In: *Physics of Fluids A: Fluid Dynamics* 1 (1989), p. 2042.

[23] Finn Ravndal and Eirik Grude Flekkøy. *Statistical Physics - a second course*. Lecture notes in statistical physics. 2014.

[24] Adri CT Van Duin et al. “ReaxFF: a reactive force field for hydrocarbons”. In: *The Journal of Physical Chemistry A* 105.41 (2001), pp. 9396–9409.

[25] Priya Vashishta et al. “Interaction potential for SiO2: a molecular-dynamics study of structural correlations”. In: *Physical Review B* 41.17 (1990), p. 12197.

[26] Loup Verlet. “Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules”. In: *Physical Review* 159.1 (1967), p. 98.

[27] Wolfgang Wagner. “A convergence proof for Bird’s direct simulation Monte Carlo method for the Boltzmann equation”. In: *Journal of Statistical Physics* 66.3-4 (1992), pp. 1011–1044.

[28] Wikipedia. *Marching cubes — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-March-2014]. 2014. URL: http://en.wikipedia.org/wiki/Marching_cubes.