Wall Effects in Two-Dimensional Molecular Dynamics Simulations

Thesis for the Degree of
Candidatus Scientiarum

Lance Olav Eastgate

Department of Physics
University of Oslo
November 1998

Abstract

Two-dimensional molecular dynamics simulation is investigated using an original computer program. The wall effects have a central role.

Both statistical theory and a background discussion of molecular dynamics are presented, followed by a more technical description of the actual program. The chronological development of the work is also presented.

A major topic is the discussion of the fluctuations of data obtained through simulation, focusing on kinetic energy fluctuations and the standard error of calculated means. Inaccuracies are caused by both inherent fluctuations in the system and numerical errors. Due to the deterministic behavior of molecular dynamics simulation on short time scales, errors may easily be underestimated. A method is presented where these short correlation times are effectively removed from the error.

In systems connected to heat reservoirs through thermal walls, the correlation time is found to be over 300 times longer than in insulated systems, owing to larger inherent fluctuations. This has implications for the length of the run needed to get statistically acceptable averages.

The investigation of accuracy shows that there are no unique criteria for choosing a best time-step length. There are trade-offs between accuracy and the length of simulation runs, and between keeping the average temperature constant and having short correlation times.

Five different simulation runs are analyzed. After monitoring the approach towards equilibrium, the fluid structure is investigated both locally and through density profiles. The inhomogeneous structure close to the walls is found to be quite similar when insulating and thermal walls are compared.

Finally, it is shown that the use of incorrect velocity distributions in the thermal walls keeps the system from reaching equilibrium. Results deduced using equilibrium assumptions are thus invalidated, as illustrated by non-uniform temperature profiles. Examples of published articles using incorrect distributions are given.

It is concluded that details are important when running computer simulations. Without a thorough knowledge of the background mechanisms, incorrect results easily lead to wrong conclusions.

The correct implementation of thermal walls forms a basis for future hybrid simulations combining hydrodynamics with molecular dynamics simulation.

Preface

This thesis has been written for the degree of Candidatus Scientiarum in the Cooperative Phenomena Group, Department of Physics, University of Oslo. My supervisors have been Professors Jens Feder, Torstein Jøssang and Paul Meakin.

The main theme of this thesis is two-dimensional molecular dynamics and the behavior of fluid near walls. The original motivation for using molecular dynamics was the inability of macroscopic representations to predict the behavior of fluid near boundaries, since the boundary conditions need to be known *a priori*. In the long run, the hope is to combine molecular dynamics with the continuum description, using the former at the boundaries and the latter for the bulk of the fluid.

It soon became evident that a molecular dynamics simulation program needs to be constructed with care. Details of the implementation can easily change the properties of the system, invalidating results. This is perhaps the most important result of this thesis.

Acknowledgments

I would like to thank my supervisors for their good advice and many useful discussions. Although very busy, they took time and were always positive to my requests.

A special thanks goes to Dag Kristian Dysthe. His detailed knowledge of molecular dynamics has been of invaluable help, several times suggesting sorely needed improvements. His enthusiasm and kindness have been a great encouragement.

I am grateful to all my other colleagues in the group for their comments, support and friendship. Their good humor and concern for fellow students make the days brighter. A thanks also goes to Richard Eastgate, my father, for helpful tips and comments.

Finally, I would like to thank my family and friends for bearing with me. Computers and physics give life color; family and friends are what give life meaning.

November, 1998 Lance Olav Eastgate

Contents

Preface **i**

Acknowledgments **i**

1 Introduction **1**

2 Statistical Theory **3**
2.1 Statistical Mechanics 3
2.2 Velocity Distributions 5
2.3 Angular Distribution 9
2.4 Empirical Average 11
2.5 Autocorrelation Function 11

3 Molecular Dynamics **13**
3.1 Molecular Simulation in General 13
3.2 The Model 14
3.3 Integration 15
3.3.1 The Verlet Method 16
3.3.2 The Leap-Frog Method 17
3.4 The Lennard-Jones Potential 17
3.5 Reduced Units 21
3.6 Boundary Conditions 22
3.6.1 Periodic Boundaries 22
3.6.2 Lennard-Jones Walls 23
3.6.3 Thermal Walls 24

4 The Simulation Program **27**
4.1 Outline 27
4.2 Data Structure 28
4.3 Recording Information 29
4.4 Random Sampling 31
4.5 Precalculated Tables 31

5 Development **33**
5.1 Flow in a Tube . . . 33
5.2 Thermal Walls . . . 34
5.3 Moving Particles Out of the Wall Zones . . . 35

6 Fluctuations **37**
6.1 Numerical Errors . . . 38
6.2 Standard Error of the Mean . . . 40
6.3 Autocorrelation . . . 41
6.4 Estimating the Standard Error by Coarse-Graining . . . 42
6.5 "Diffusion" of the Mean . . . 43
6.6 Choosing a Time-step . . . 47

7 Analysis of Simulation Results **49**
7.1 Initialization . . . 49
7.2 Equilibration . . . 50
7.3 Fluid Structure . . . 51
7.3.1 Radial Distribution Function . . . 52
7.3.2 Density Profiles . . . 55
7.4 Temperature Profiles . . . 60

8 Concluding Remarks **69**

Bibliography **71**

Appendices **74**

A Notation **74**

B The Program Listings **78**
B.1 The Main Program . . . 78
B.1.1 MD_main.hh . . . 80
B.1.2 MD_main.cc . . . 80
B.1.3 MD_definitions.hh . . . 84
B.1.4 MD_constants.hh . . . 85
B.1.5 MD_classes.hh . . . 89
B.1.6 MD_classfuncs.cc . . . 92
B.1.7 MD_init.hh . . . 102
B.1.8 MD_init.cc . . . 102
B.1.9 MD_calc.hh . . . 106
B.1.10 MD_calc.cc . . . 107
B.1.11 MD_calc_inline.hh . . . 112
B.1.12 MD_fileoutput.hh . . . 114

B.1.13 MD_fileoutput.cc 116
B.1.14 MD_listtools.hh 121
B.1.15 MD_listtools.cc 121
B.2 Additional Programs 122
B.2.1 norm_pp.cc 122
B.2.2 avg_bins.cc 123
B.2.3 add_bins.cc 124
B.2.4 avg_dist_bins.cc 125
B.2.5 timecorr.cc 126
B.2.6 flyv.cc 127
B.2.7 sample_dist.cc 128

Chapter 1

Introduction

Computer simulation with molecular dynamics is an increasingly used tool for exploring the realm of statistical physics. Analytic calculations are often impossible due to the complex nature of the systems being studied. Molecular dynamics simulation solves equations numerically by propagating a model of a physical system through time.

At first glance, molecular dynamics might seem to be able to solve any physical problem. Unfortunately, there are many limitations. First, there is the problem of making a good physical model of reality. This issue is of general concern, and not limited to molecular dynamics in particular. Another problem is to understand the model by solving the equations that govern it. Some may be solved analytically, while others need to be solved numerically. Molecular dynamics simulation is one such numerical method for solving these equations.

When using molecular dynamics simulation to investigate a model, it can be tempting to believe the numerical result, forgetting about inaccuracies, the limitations of assumptions, or even errors. In this thesis, I try to emphasize the importance of being critical, and that details can be of great significance. Important details can easily be forgotten, and I have even found that some published articles are incorrect due to oversight.

In this thesis, I have used molecular dynamics simulation to investigate the properties of a two-dimensional Lennard-Jones fluid with walls at the boundaries. In particular, I have looked at the interaction between the particles and the walls. I have found that both the assumptions and the implementation of seemingly unimportant details have great impact on some properties of the system, in particular the temperature profiles.

Specifically, the topics in this thesis include the development of my own molecular dynamics program, a discussion of the numerical accuracy of the results, and the effect of thermal walls on the temperature profiles.

Chapter 2 discusses some concepts in statistical mechanics, including an introduction to distributions describing the velocity of fluid particles.

Chapter 3 gives a thorough presentation of molecular dynamics simulation, including a discussion of thermal walls. Chapter 4 consists mostly of technical details in the simulation program I developed in connection with this thesis. A short summary of the chronological development of the work is given in Chapter 5, and it shows how my initial investigations generated the problems I try to answer in this thesis. It was then necessary to include, in Chapter 6, a discussion of accuracy, the choice of time-step length, and the origin and behavior of fluctuations in property averages. Chapter 7 presents the simulation results and analyzes the consequences of various choices of wall models. It includes discussions of equilibration, fluid structure and temperature profiles.

One of the main messages I wish to convey in this thesis, is the importance of paying attention to details. Firstly, I show that the choice of model parameters is a trade-off in accuracy and in the purpose of the investigation. Secondly, it is concluded that the choice of thermal-wall velocity distribution may strongly affect the temperature profile, and thus the outcome of the molecular dynamics study.

Chapter 2

Statistical Theory

This chapter starts by discussing the nature and validity of some aspects of statistical mechanics. Afterwards, a presentation of some statistical distributions and methods that will be needed later in the thesis is given. The distributions will be important in the discussion of thermal walls and temperature profiles.

2.1 Statistical Mechanics

This thesis is concerned with the modeling and simulation of a many-particle system. A brief discussion of the main concepts in thermodynamics and statistical physics is therefore appropriate.

Consider a closed system of N particles interacting classically* through a potential. Microscopically, the system is fully described by Newton's equations of motion. If the positions and momenta at any instant are given, the trajectories of the particles can in principle be calculated for any later time. The system is thus deterministic.

In practice, the task of solving this set of equations becomes impracticable even for small values of N, owing to the complexity of the system. Fortunately, this complexity allows the system to be described macroscopically, using averages of the microscopic values. These average properties give the system a statistical or probabilistic description, in contrast to the deterministic behavior of the microscopic representation.

The use of average properties to describe a system requires that these averages exist, and thus imposes assumptions on the system. Many of the macroscopic properties assume thermal equilibrium. Intuitively, this means that the system has been allowed to settle. Landau & Lifshitz (1989, p. 6) have the following definition of equilibrium: "If a closed macroscopic system is in a state such that in any macroscopic subsystem the macroscopic physical quantities are to a high degree of accuracy equal to their mean values, the

*As opposed to quantum mechanically.

system is said to be in a state of *statistical equilibrium* (or *thermodynamic* or *thermal equilibrium*)."

There exist systems where the average properties remain constant with time, but do not satisfy the above definition of equilibrium. The steady flow of particles through a tube is a good example of this. Such systems will be called *stationary*. Many of the conclusions reached in thermodynamics hold for systems in equilibrium, but not for those in only a stationary state. The following discussion of absolute temperature exemplifies this.

In thermodynamics, the absolute temperature T for a system in equilibrium is defined by

$$\frac{1}{T} = \frac{dS}{dE}, \tag{2.1}$$

where S is the total entropy and E is the total energy (Landau & Lifshitz, 1989, p. 35). The law of equipartition for a monatomic gas of N particles states that

$$\langle K \rangle = \frac{f}{2} N k_B T, \tag{2.2}$$

where $\langle K \rangle$ is the time-averaged kinetic energy, f is the number of translational degrees of freedom, and k_B is Boltzmann's constant. This equation is also deduced under equilibrium assumptions. One thus has a method of calculating the temperature from the kinetic energy of the particles. The reason for raising this topic is that in Chapter 7, the temperature is calculated using this method. In some of the simulated systems the algorithms prevent them from reaching equilibrium. This results in strange behavior of what should have been, under equilibrium conditions, the temperature.

The next section will discuss the Maxwell-Boltzmann velocity distribution, which shows how the kinetic energy is distributed among the particles. This is another example of an equilibrium property.

Sometimes properties such as the absolute temperature are used, even though the system is known not to be in equilibrium. To illustrate, consider two separate equilibrated systems at different temperatures. If they are brought together, the system as a whole is no longer in equilibrium. Does it still make sense to talk about the temperatures in each part?

This leads to the concept of *partial equilibrium*. The time it takes for a part of a system to reach equilibrium is shorter than for the system as a whole. If this difference in *relaxation time* is large, the parts can reach equilibrium separately, even though the total system is in a state of non-equilibrium. This is called partial equilibrium.

Until now, only closed or isolated systems have been considered. Sometimes other systems are more interesting, for example those exchanging heat (canonical) or even particles (grand canonical) with their surroundings. This can be approached by dividing a closed system into two parts, one small and one large. The system under investigation is identified with the small part. The large part is often called the reservoir. A non-isolated system is thus

equivalent to a small part of a closed system in equilibrium with the reservoir.

When using results or properties from thermodynamics and statistical physics, it is important to understand the basic assumptions under which these were deduced or defined. Chapter 7 will illustrate this.

2.2 Velocity Distributions

In classical statistical mechanics, a system of particles in equilibrium can be described by the *distribution function*, $\rho(p, q)$, where p and q are the momenta and positions, respectively, of all the particles in the gas (Landau & Lifshitz, 1989, p. 3). The quantity $\rho(p, q)\, dp\, dq$ is the probability that the system will occupy the points in phase space that lie in the intervals p to $p + dp$ and q to $q + dq$. Here, $dp = dp_1 dp_2 \cdots dp_s$ and $dq = dq_1 dq_2 \cdots dq_s$, where $s = 2N$ in a two-dimensional, N-particle system, when including all particles and dimensions.

Since ρ is a probability distribution (or probability density function, PDF), normalization requires that

$$\int \rho(p, q)\, dp\, dq = 1. \tag{2.3}$$

If a physical quantity A only depends on p and q, its mean can be calculated from the distribution function:

$$\langle A \rangle = \int A(p, q)\rho(p, q)\, dp\, dq. \tag{2.4}$$

The distribution function is often written as (Landau & Lifshitz, 1989, p. 81)

$$\rho(p, q) = Ce^{-\frac{E(p,q)}{k_B T}}, \tag{2.5}$$

where $E(p, q)$ is the energy of the system, k_B is Boltzmann's constant, and T is the absolute temperature. The constant C is given by normalization. The expression applies to a canonical system in equilibrium. Notice that any mean $\langle A \rangle$ based on ρ will only be a function of T, which is assumed to be constant throughout the system.

In a classical system, the energy $E(p, q)$ can always be written as a sum of the kinetic energy $K(p)$ and the potential energy $U(q)$ (Landau & Lifshitz, 1989, p. 82), where K and U are only functions of the momenta and positions, respectively. As a result, the distribution function can be divided into two independent distributions[†], each which can be integrated

[†]Except for some of the most commonly used physical distributions, such as $\rho(p, q)$, PDFs will be named by the convention usually applied in mathematical statistics. The PDF for a random variable X is written $f_X(x)$, and $f_X(x)dx$ is the probability that the random variable X lies in the interval between x and $x + dx$. See for example Larsen & Marx (1986) for more information on random variables and probability density functions.

separately over p and q:

$$\begin{aligned} \rho(p,q) &= f_P(p) f_Q(q) && (2.6)\\ f_P(p) &= C_K e^{-\frac{K(p)}{k_B T}} && (2.7)\\ f_Q(q) &= C_U e^{-\frac{U(q)}{k_B T}}. && (2.8) \end{aligned}$$

By writing the kinetic energy as

$$K(p) = \sum_{i=1}^{s} \frac{1}{2} m v_i^2, \tag{2.9}$$

m being the particle mass and v_i the velocity components of the particles, the distribution function for the momenta, $f_P(p)$, can be written as

$$f_P(p) = f_{V_1,V_2,\ldots}(v_1, v_2, \ldots) = C_K \prod_{i=1}^{s} e^{-\frac{v_i^2}{\alpha^2}} \tag{2.10}$$

with

$$\alpha^2 = \frac{2 k_B T}{m}. \tag{2.11}$$

As can be seen from the above expression, not only are the particles independent, but the different velocity components of each particle are also independent. The distribution function for a single velocity component can thus be written

$$f_{V_i}(v_i) = \frac{1}{\sqrt{\pi}\alpha} e^{-\frac{v_i^2}{\alpha^2}}, \qquad i = 1, 2, \ldots, s, \tag{2.12}$$

where the normalization condition has been used to find the value of the constant C_K. This function is known as the Gaussian distribution (Landau & Lifshitz, 1989, p. 333), and is plotted in Figure 2.1.

To calculate the probability density function (PDF) for V (the magnitude of the velocity) in two dimensions, one must first find the cumulative distribution function (CDF). With $R \equiv \{v_x, v_y \in \mathbb{R} | v_x^2 + v_y^2 \leq v^2\}$, the CDF becomes

$$\begin{aligned} F_V(v) &= P(V \leq v)\\ &= \iint_R f_{V_x}(v_x) f_{V_y}(v_y)\, dv_x\, dv_y\\ &= \int_0^v \int_0^{2\pi} \frac{1}{\alpha^2 \pi} e^{-\frac{u^2}{\alpha^2}} u\, du\, d\theta\\ &= \frac{2}{\alpha^2} \int_0^v u e^{-\frac{u^2}{\alpha^2}}\, du. \end{aligned} \tag{2.13}$$

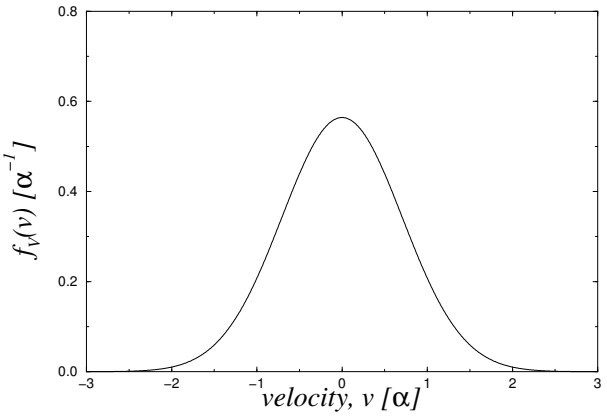

Figure 2.1: *The probability density function (the Gaussian distribution) for the velocities in one direction.*

Here the normal change to polar coordinates was used, with

$$v_x = u\cos\theta \tag{2.14}$$
$$v_y = u\sin\theta. \tag{2.15}$$

The PDF is then found by taking the derivative of $F_V(v)$:

$$\begin{aligned} f_V(v) &= \frac{d}{dv}F_V(v) \\ &= \frac{2}{\alpha^2}ve^{-\frac{v^2}{\alpha^2}}. \end{aligned} \tag{2.16}$$

This is the two-dimensional Maxwellian speed distribution (the three-dimensional distribution is given in Landau & Lifshitz (1989, p. 83)). See Figure 2.2 for a plot of Equation 2.16.

Equation 2.12 applies to the velocities of particles chosen at random at any given time-step. What would the distribution of velocities in the x-direction become if one only considered particles *crossing a vertical*[‡] *line*? Particles that have no speed in this direction will never cross the line, as shown in Figure 2.3, and will therefore not show up in the distribution. Thus Equation 2.12 and Figure 2.1 is clearly *not* correct in this situation.

The PDF for the velocities of particles crossing a vertical line, $\tilde{f}_{V_x}(v_x)$, has an extra factor of v_x. In order to see this, consider for a moment only the velocities in the x-direction. The probability of a particle crossing any given vertical line within a time-step dt is proportional to the distance $v_x dt$ it travels in the x-direction. This distance is proportional to the velocity, since v_x can be considered constant within dt. One can therefore conclude

[‡]Perpendicular to the x-direction.

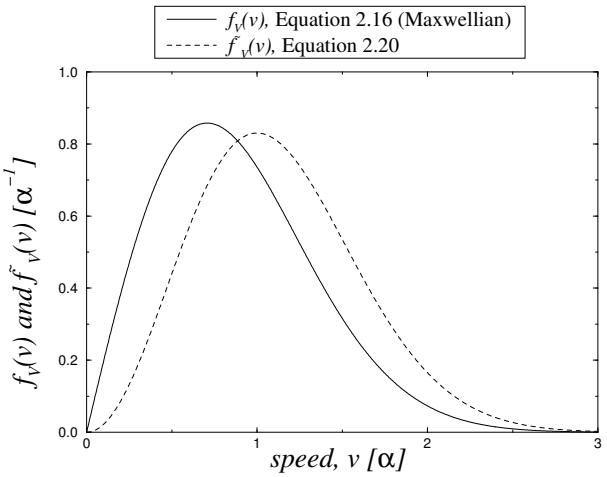

Figure 2.2: *The probability density function for the speed in a two-dimensional system, both for particles picked at random (the Maxwellian distribution) and particles crossing a line.*

that the chance of a random particle crossing a vertical line is proportional to v_x, and the PDF becomes

$$\tilde{f}_{V_x}(v_x) = \frac{2}{\alpha^2} v_x e^{-\frac{v_x^2}{\alpha^2}}. \tag{2.17}$$

The PDF in the y-direction stays the same as before:

$$f_{V_y}(v_y) = \frac{1}{\sqrt{\pi}\alpha} e^{-\frac{v_y^2}{\alpha^2}} \tag{2.18}$$

When finding the CDF for the magnitude of velocities of the particles crossing a vertical line in one direction, say to the right, one must remember that v_x stays positive (or negative).

With $\tilde{R} \equiv \{v_x, v_y \in \mathbb{R} | v_x^2 + v_y^2 \leq v^2 \wedge v_x \geq 0\}$, the CDF becomes[§]

$$\begin{aligned}
\tilde{F}_V(v) &= P(V \leq v) \\
&= \iint_{\tilde{R}} \tilde{f}_{V_x}(v_x) f_{V_y}(v_y)\, dv_x\, dv_y \\
&= \int_0^v \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \frac{2}{\alpha^3\sqrt{\pi}} u \cos\theta e^{-\frac{u^2}{\alpha^2}} u\, du\, d\theta \\
&= \int_0^v \frac{4}{\alpha^3\sqrt{\pi}} u^2 e^{-\frac{u^2}{\alpha^2}}\, du,
\end{aligned} \tag{2.19}$$

[§]See Figure 2.4 for the angular integration limits.

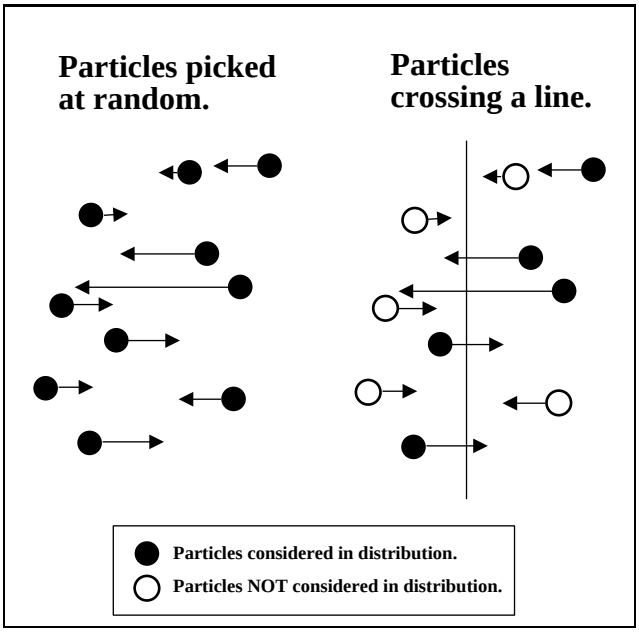

Figure 2.3: *The particles with a high velocity have a greater probability of crossing a line. When choosing particles at random, the velocity has no impact on the selection. Only the velocities in the x-direction are shown.*

giving the PDF for particles crossing a line (see Figure 2.2):

$$\tilde{f}_V(v) = \frac{4}{\alpha^3\sqrt{\pi}} v^2 e^{-\frac{v^2}{\alpha^2}}. \tag{2.20}$$

2.3 Angular Distribution

It is also possible to calculate the PDF for the angle with which the particles will cross a vertical line. Let the random variable Θ be this angle, with $\Theta = 0$ normal to the line (see Figure 2.4).

First, the CDF is calculated:

$$\tilde{F}_\Theta(\theta) = \begin{cases} 0 & \theta < -\frac{\pi}{2} \\ P(-\frac{\pi}{2} \leq \Theta \leq \theta) & -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2} \\ 1 & \frac{\pi}{2} < \theta \end{cases} \tag{2.21}$$

where

$$\begin{aligned} P(-\frac{\pi}{2} \leq \Theta \leq \theta) &= \frac{1}{2} + P(0 \leq \Theta \leq \theta) \\ &= \frac{1}{2} + \int_0^\infty \int_{\frac{v_y}{\tan\theta}}^\infty \tilde{f}_{V_x}(v_x) f_{V_y}(v_y) dv_x dv_y \end{aligned}$$

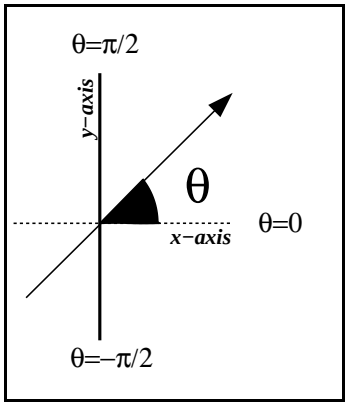

Figure 2.4: *Definition of angle with which a particle crosses a line to the right.*

$$\begin{aligned}
&= \frac{1}{2} + \int_0^\infty \int_{\frac{v_y}{\tan\theta}}^\infty \frac{2}{\alpha^3\sqrt{\pi}} v_x e^{-\frac{v_x^2}{\alpha^2}} e^{-\frac{v_y^2}{\alpha^2}} dv_x dv_y \\
&= \frac{1}{2} + \int_0^\infty \frac{1}{\alpha\sqrt{\pi}} e^{-\frac{v_y^2}{\alpha^2}} \left(\int_{\frac{v_y}{\tan\theta}}^\infty \frac{2}{\alpha^2} v_x e^{-\frac{v_x^2}{\alpha^2}} dv_x \right) dv_y \\
&= \frac{1}{2} + \int_0^\infty \frac{1}{\alpha\sqrt{\pi}} e^{-\frac{v_y^2}{\alpha^2 \sin^2\theta}} dv_y. \qquad (2.22)
\end{aligned}$$

The integration limit $v_y/\tan\theta$ is deduced from the relation $\tan\theta = v_x/v_y$.

The PDF is found by taking the derivative of the CDF $\tilde{F}_\Theta(\theta)$. The following is done under the assumption that the integral and the derivative can commute.

$$\begin{aligned}
\tilde{f}_\Theta(\theta) &= \frac{d}{d\theta}\tilde{F}_\Theta(\theta) \\
&= \frac{2\cos\theta}{\sqrt{\pi}} \int_0^\infty \frac{v_y^2}{\alpha^3 \sin^3\theta} e^{-\frac{v_y^2}{(\alpha\sin\theta)^2}} dv_y \\
&= \frac{2\cos\theta}{\sqrt{\pi}} \frac{\sqrt{\pi}}{4} \\
&= \frac{1}{2}\cos\theta. \qquad (2.23)
\end{aligned}$$

Equation 2.23 is plotted in Figure 2.5.

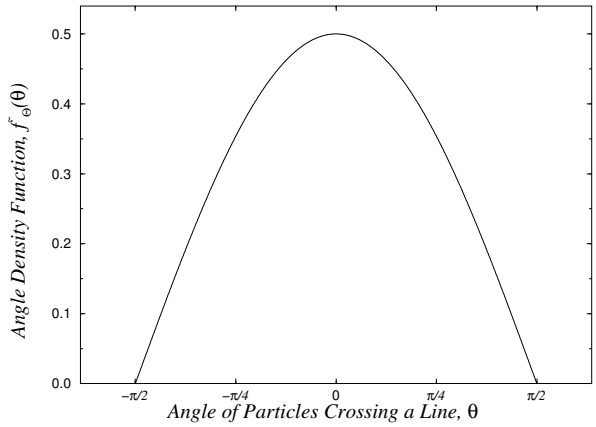

Figure 2.5: *The angular probability density function for particles crossing a line.*

2.4 Empirical Average

The theoretical mean of a PDF is found by using Equation 2.4. The two-dimensional Maxwellian speed distribution (2.16) would thus give

$$\langle v \rangle = \int_0^\infty v f_V(v) dv = \frac{\alpha}{2}\sqrt{\pi}. \tag{2.24}$$

To calculate the empirical average, one could use the sample mean

$$\bar{v} = \frac{1}{N}\sum_{i=1}^{N} |\mathbf{v}_i|. \tag{2.25}$$

Alternatively, one could make a discrete speed distribution $g(u_i)$ of the simulated gas by arranging the particle speeds into B bins, where u_i is the mean speed in the ith bin. The average speed can then be calculated by

$$\bar{v} \approx \sum_{i=1}^{B} u_i g(u_i). \tag{2.26}$$

The values of $\bar{v}$ and $\langle v \rangle$ will grow closer as B and N grow larger.

2.5 Autocorrelation Function

If X and Y are two random variables (which means they both have a probability density function), the correlation coefficient can be defined as (Larsen & Marx, 1986, p. 436)

$$\rho(X,Y) = \frac{E(XY) - E(X)E(Y)}{\sqrt{\mathrm{Var}(X)\mathrm{Var}(Y)}}, \tag{2.27}$$

where Var and E represent the variance and expected value, respectively. If X and Y are totally uncorrelated, $E(XY) = E(X)E(Y)$ and $\rho(X, Y) = 0$. Thus, two independent (uncorrelated) random variables give zero for the correlation coefficient. Note that the opposite is not true in general, since two variables might depend on each other in some way, but still give zero for the coefficient.

On the other hand, if Y is replaced by X, the numerator becomes the variance, and $\rho(X, X) = 1$. This is an example of the fact that two correlated variables will give a higher result for the absolute value of the correlation coefficient.

If $X(t)$ and $Y(t)$ are two time-dependent signals, the time correlation function can be defined as (Haile, 1997, p. 278)

$$C(t) = \lim_{\tau \to \infty} \frac{1}{\tau} \int_0^\tau X(t_0) Y(t_0 + t) dt_0 \tag{2.28}$$

$$= \langle X(t_0) Y(t_0 + t) \rangle. \tag{2.29}$$

This function calculates the correlation coefficient between the two variables for different translations in time. If one sets $X(t) \equiv Y(t)$, $C(t)$ is called the *autocorrelation* function.

An estimate for the autocorrelation of a property $x(t)$ may be calculated from (Haile, 1997, p. 79)

$$C(t) = \frac{\sum_{k=1}^{M} \left(x(t_k) - \langle x \rangle\right) \left(x(t_k + t) - \langle x \rangle\right)}{\sum_{k=1}^{M} \left(x(t_k) - \langle x \rangle\right)^2}. \tag{2.30}$$

Notice that the factors in this equation fluctuate around zero. The value of $C(0)$ will always be unity. In mechanical systems evolving in time, two events become increasingly uncorrelated when separated by a growing value of t. Finally, for large values of t, if the two factors in the sum in the numerator of Equation 2.30 fluctuate independently, $C(t)$ becomes zero.

Chapter 3

Molecular Dynamics

Molecular dynamics simulation is concerned with finding macroscopic properties from microscopic systems. This chapter discusses some of the main concepts in molecular dynamics. The next chapter will then explain more specifically how the algorithms and data structures were implemented.

A short overview of molecular simulation in general is given before presenting the model. Numerical integration of the differential equations governing the motion is then presented, showing how the movement of particles is divided into small time steps. Then comes an explanation of the mechanism through which the particles interact, namely the Lennard-Jones (12,6) potential. The concept of reduced units is introduced, and finally the different kinds of boundary conditions are discussed, including periodic boundaries, insulating Lennard-Jones walls, and thermal walls.

3.1 Molecular Simulation in General

The investigation of a physical system usually proceeds in several steps. First, a theory describing the system is proposed. Such a theory will never describe the real system completely, and is therefore only a model. Once a model is made, one can use it to make predictions about the physical system, either by analytical calculations or by simulations. These predictions should then be compared to experiments. The correspondence between the predictions and the experiments gives an indication of how well the model describes the physical system.

If simulations are used, it is important to realize that the correspondence between the model and the algorithms implemented in the simulations is distinct from that between the model and a real physical system. The main concern of this thesis lies mainly in the former.

The type of simulation to use depends on the model. Some models are based on macroscopic properties, such as density, temperature and pressure. Others have their origin in microscopic properties, such as the interparticle

potential. Molecular dynamics simulation, which is the method used in this thesis, simulates a microscopic model.

Molecular dynamics is not the only way to simulate a model based on microscopic properties. In fact, there is a whole range of methods, starting from Monte Carlo simulation, which is totally stochastic, to molecular dynamics simulation, which is considered deterministic. Monte Carlo simulation tries to find energetically stable configurations for the system by considering random movements in phase space, always accepting them if it leads to a lower total potential energy. Otherwise, a movement is accepted with a certain probability.

In contrast to Monte Carlo simulation, where the system moves in a random (or stochastic) manner, the movement of particles in molecular dynamics is determined by the forces acting on each particle, according to Newton's laws of motion. Thus in principle, two systems with the same initial conditions will always follow the same trajectory in phase space. This is not true for Monte Carlo simulation. One obvious difference between Monte Carlo and molecular dynamics simulations, is the possibility of calculating time-dependent property averages in the latter.

The reason why both simulation methods can be used to calculate property averages, is that both methods are believed to cover the available phase space in a stochastic manner. For molecular dynamics, one has made the assumption that the system is ergodic, that is, time averages and ensemble averages give the same result.

One might object to the use of molecular dynamics due to its seemingly deterministic nature, but a closer inspection shows that there are in fact stochastic processes going on in a molecular dynamics simulation. Firstly, if two systems start with similar but different initial conditions, they will behave similarly at the start, but will diverge after a while and become completely uncorrelated. In addition, a molecular dynamics simulation will introduce small numerical inaccuracies at every time-step, and this will contribute to the stochastic behavior. But even if the simulations had been absolutely accurate, most systems would probably cover the available phase space in a satisfactory manner. This would, however, depend on the initial and boundary conditions.

3.2 The Model

The first task in an investigation is always to identify the problem. In this thesis, one of the goals was to describe and understand the behavior of a two-dimensional model with periodic boundary conditions in one direction, and walls in the other (see Figure 3.1). An interaction potential similar to those found between atoms in inert gases was required. The walls were to exchange energy with the system, since their initial function was to dissipate

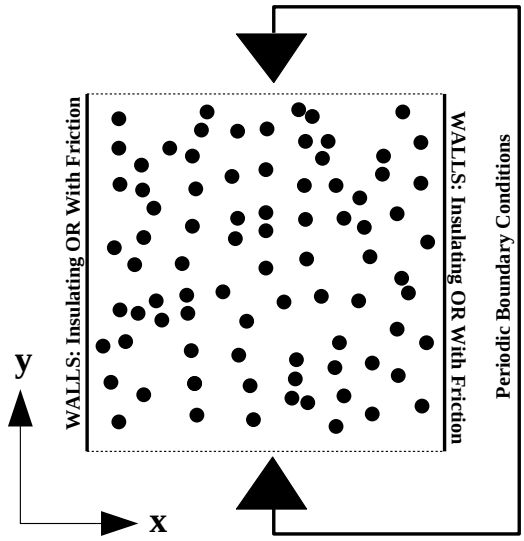

Figure 3.1: *A two-dimensional system with periodic boundary conditions in the y-direction, and vertical walls in the x-direction.*

the heat supplied by a constant gravitational-like force.

Molecular dynamics was chosen as the tool to investigate this model. To simulate the particle interactions, a Lennard-Jones (12,6) potential was used, and thermal walls seemed to be a reasonable way to connect the system to a heat bath.

Although physical considerations have guided many of the choices regarding the simulation program, this model was not designed to describe any particular physical system. Most of the effort was put into investigating the model.

The rest of this chapter describes the most fundamental parts of molecular dynamics simulation.

3.3 Integration

In a molecular dynamics simulation, the positions and velocities are changed according to Newton's laws of motion in small time-steps δt. For each time-step, new positions and velocities must be calculated from the forces that the particles exert on each other and from their positions and velocities at the previous time-step.

Several algorithms have been proposed in the literature. Two of them, Verlet's original method (Verlet, 1967) and the so-called "leap-frog" method (Allen & Tildesley, 1991, p. 78), will be described below. Other methods

exist, for example the velocity form of the Verlet algorithm (Swope, Andersen, Berens & Wilson, 1982), algorithms by Beeman (1976), and predictor-corrector algorithms (see for example Gear (1966) or Allen & Tildesley (1991, p. 82)). Only the leap-frog method was used in the simulations.

3.3.1 The Verlet Method

During each cycle of a simulation, the system progresses a small time-step δt. Given the current and previous position of a particle, $\mathbf{r}(t)$ and $\mathbf{r}(t-\delta t)$, and the current acceleration, $\mathbf{a}(t)$, Verlet proposed that the new position $\mathbf{r}(t+\delta t)$ could be found by

$$\mathbf{r}(t+\delta t) = 2\mathbf{r}(t) - \mathbf{r}(t-\delta t) + (\delta t)^2\mathbf{a}(t). \tag{3.1}$$

The current acceleration is calculated from the force acting on the particles, which in turn depends on the current position (see Section 3.4):

$$\mathbf{a}(t) = \frac{\mathbf{F}\left[\mathbf{r}(t)\right]}{m}. \tag{3.2}$$

Equation 3.1 was found by eliminating the velocities from the Taylor expansions about $\mathbf{r}(t)$:

$$\mathbf{r}(t+\delta t) = \mathbf{r}(t) + \delta t\mathbf{v}(t) + \frac{1}{2}(\delta t)^2\mathbf{a}(t) + \ldots \tag{3.3}$$

$$\mathbf{r}(t-\delta t) = \mathbf{r}(t) - \delta t\mathbf{v}(t) + \frac{1}{2}(\delta t)^2\mathbf{a}(t) - \ldots \tag{3.4}$$

To calculate the kinetic energy, the velocity is needed. Subtracting Equation 3.4 from 3.3 and dividing by $2\delta t$ gives

$$\mathbf{v}(t) = \frac{\mathbf{r}(t+\delta t) - \mathbf{r}(t-\delta t)}{2\delta t}. \tag{3.5}$$

In Equation 3.1, the error is of the order $(\delta t)^4$ because this is the first term that was truncated (the term with $(\delta t)^3$ disappears due to opposite signs in Equations 3.3 and 3.4). The error in Equation 3.5 is of the order $(\delta t)^2$ (and not $(\delta t)^3$, since all the terms are divided by δt). The algorithm is relatively simple, it is time-reversible, and has been shown to conserve energy well, even at long time-steps (Allen & Tildesley, 1991, p. 79). On the negative side, the Verlet method has a tendency to be inaccurate, since it requires the subtraction of one large term from another in order to estimate a small one; in Equation 3.1 the expression $\mathbf{r}(t) - \mathbf{r}(t-\delta t)$, which is the distance moved from the original position $\mathbf{r}(t)$, has two terms which might be quite large. The subtraction of these terms is then added (and thus compared) to the small term $(\delta t)^2\mathbf{a}(t)$.

3.3.2 The Leap-Frog Method

An alternative to the Verlet method is the so-called "leap-frog" method, a modification of the Verlet scheme (see for example Allen & Tildesley (1991, p. 80) and Potter (1973, p. 32)). The name comes from the fact that the algorithm uses velocities that are half a time-step out of phase. When the new positions are calculated, the velocities "leap" over the current time-step. Given the current acceleration and position of a particle, $\mathbf{a}(t)$ and $\mathbf{r}(t)$, and the velocity a half time-step earlier, $\mathbf{v}(t-\frac{\delta t}{2})$, the new position and velocity, $\mathbf{r}(t+\delta t)$ and $\mathbf{v}(t+\frac{\delta t}{2})$, can be calculated:

$$\mathbf{v}(t+\frac{\delta t}{2}) = \mathbf{v}(t-\frac{\delta t}{2}) + \delta t\mathbf{a}(t) \tag{3.6}$$

$$\mathbf{r}(t+\delta t) = \mathbf{r}(t) + \delta t\mathbf{v}(t+\frac{\delta t}{2}). \tag{3.7}$$

To find the kinetic energy, the velocity at time t must be calculated:

$$\mathbf{v}(t) = \frac{\mathbf{v}(t+\frac{\delta t}{2}) + \mathbf{v}(t-\frac{\delta t}{2})}{2}. \tag{3.8}$$

Equation 3.6 is truncated after the term containing the acceleration, giving a truncation error of the order $(\delta t)^2$. It is important that the velocity at time t is used when calculating the kinetic energy, since the total energy is found by adding the kinetic and potential parts. This should ensure that the total energy is constant in an isolated system, if numerical errors are ignored.

In this algorithm, the velocities are directly involved, making any scaling of velocities easier. In addition, two large values are never subtracted to get a small one, as in Verlet's original algorithm.

Figure 3.2 illustrates the two above algorithms.

3.4 The Lennard-Jones Potential

The interparticle potential is one of the main parts of a fluid model. In general, the potential energy of a gas can be written

$$U(\mathbf{r}^N) = \sum_{\text{particles}} u_1(\mathbf{r}_i) + \sum_{\text{pairs}} u_2(\mathbf{r}_i, \mathbf{r}_j) + \sum_{\text{triplets}} u_3(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) + \ldots \tag{3.9}$$

where $\mathbf{r}^N = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$ are the positions of all the particles. The first sum includes any external potential fields, and the other sums are contributions from interactions between pairs, triplets, etc. In all the simulations discussed in this thesis, only external and pair potentials were used, that is, the two first sums in Equation 3.9.

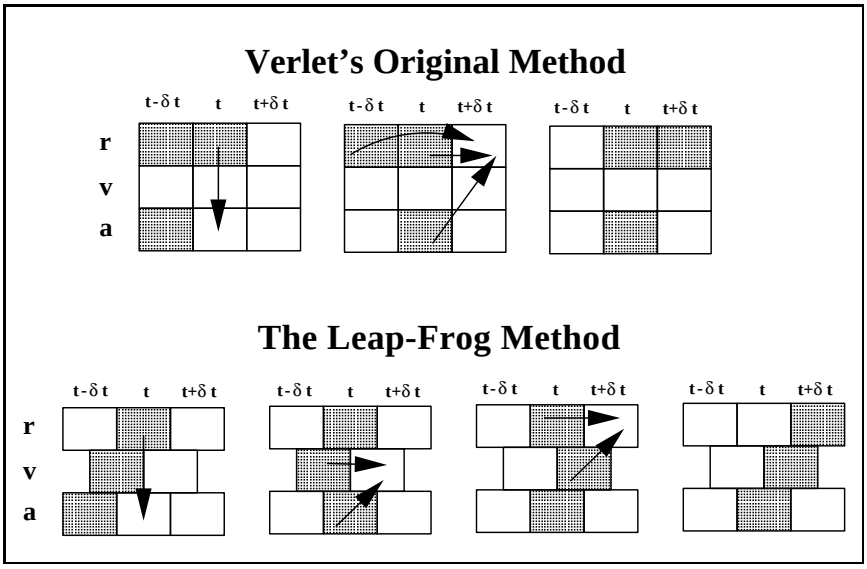

Figure 3.2: *The Verlet and the leap-frog integration algorithms (Allen & Tildesley, 1991, p. 80, Fig. 3.2 (a) and (b)).*

A commonly used pair potential in molecular dynamics simulations is the Lennard-Jones potential (Jones, 1924*a*; Jones, 1924*b*), which Haile (1997, p. 189) presents as

$$u(r) = k\epsilon \left[\left(\frac{\sigma}{r} \right)^n - \left(\frac{\sigma}{r} \right)^m \right]. \tag{3.10}$$

Here σ is the unit of length (see Section 3.5 on reduced units), $-\epsilon$ is the minimum value of $u(r)$, and

$$k = \frac{n}{n-m} \left(\frac{n}{m} \right)^{m/(n-m)}. \tag{3.11}$$

Haile (1997, p. 190) proposes that the common choice of $m = 6$ is due primarily to the leading long-range term $1/r^6$ in London's theory of dispersion (London, 1930). Further, Haile claims that it is popular wisdom that sets $n = 2m = 12$ for the short-range term, due to logic rather than physical justification. The frequently used Lennard-Jones (12,6) potential function thus becomes

$$u(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right]. \tag{3.12}$$

Notice that the potential energy becomes zero when $r = \sigma$. The force acting between two particles at a distance r is given by

$$\begin{aligned} \mathbf{F}(\mathbf{r}) &= -\nabla u(r) \\ &= -\frac{\mathbf{r}}{r} \frac{du(r)}{dr} \end{aligned}$$

$$= \frac{24\epsilon}{\sigma}\left[2\left(\frac{\sigma}{r}\right)^{13} - \left(\frac{\sigma}{r}\right)^{7}\right]\frac{\mathbf{r}}{r}. \tag{3.13}$$

The reason for using only the two first sums in Equation 3.9 (and thus assuming that the potential is pairwise additive) is that the inclusion of any further terms would slow down the simulations drastically. When three or more particles meet simultaneously in reality, the electron clouds will react differently than when only two particles meet, and thus give a different potential function. This applies especially at high densities (Allen & Tildesley, 1991, pp. 7–9), and assuming the potential to be pairwise additive will therefore give wrong results if used naively.

Fortunately, one can approximate the three-body potential by using an *effective* two-body interaction potential. Note that, even though the original two-body potential is temperature independent by definition, the effective potential may vary with temperature and density. When simulating a real gas, it is therefore important to calibrate the potential for the correct temperature and density. The numerical values in Table 3.1 give reasonable results for (three-dimensional) liquid Argon using a Lennard-Jones (12,6) potential (Maitland, Rigby, Smith & Wakeham, 1981). As the discussion of reduced units (Section 3.5) will explain, the numerical values have no qualitative effect.

In practice, the Lennard-Jones potential is truncated to reduce the number of interactions and to avoid particles interacting with themselves when periodic boundary conditions are applied:

$$u(r) = \begin{cases} 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] & r \le r_c \\ 0 & r > r_c \end{cases} \tag{3.14}$$

In these simulations, the cutoff length was set to $r_c = 2.5\sigma$, defining $u(r) \equiv 0$ for $r > r_c$. At this cutoff length, the potential makes an abrupt jump from $u(r_c) = -0.0163\epsilon$ to zero, or 1.63% of the depth of the potential well. Similarly, the force jumps to zero from $F(r_c) = -0.0390\epsilon/\sigma$, which also is 1.63% of *its* minimum value. The truncation causes some inaccuracies in the calculations. In particular, long range corrections should be added if the properties of the real liquid are important (Nicolas, Gubbins, Streett & Tildesley, 1979). In addition, the total energy will not stay constant in an isolated system, but will increase or decrease by a small amount every time two particles cross $r = 2.5\sigma$. The cutoff will not cause a constant increase or decrease in total energy, though, just an increased fluctuation. The truncated potential is plotted in Figure 3.3.

Sometimes it is necessary to remove the jump in the force and potential function, but keep the finite cutoff. This can be achieved by shifting the force by a constant, making the potential go smoothly to zero at the cutoff

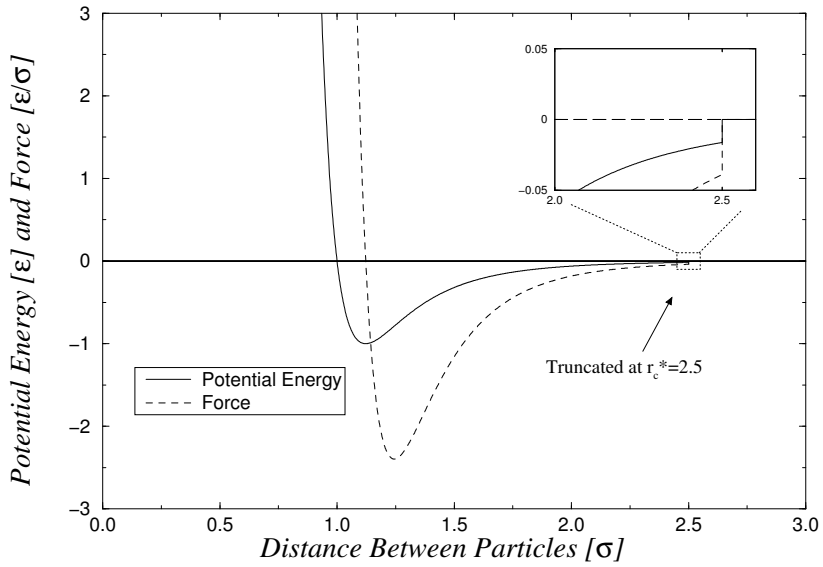

Figure 3.3: *The truncated Lennard-Jones (12,6) pair potential and force.*

length. Mathematically, this can be written

$$\mathbf{F}_s(\mathbf{r}) = \begin{cases} \left(\frac{24\epsilon}{\sigma}\left[2\left(\frac{\sigma}{r}\right)^{13} - \left(\frac{\sigma}{r}\right)^{7}\right] + \Delta F\right)\frac{\mathbf{r}}{r} & r \le r_c \\ \mathbf{0} & r > r_c \end{cases} \tag{3.15}$$

where

$$\begin{aligned} \Delta F &= |\mathbf{F}(\mathbf{r}_c)| \\ &= \left|\left(\frac{du}{dr}\right)_{r_c}\right| \\ &= \left|\frac{24\epsilon}{\sigma}\left[2\left(\frac{\sigma}{r_c}\right)^{13} - \left(\frac{\sigma}{r_c}\right)^{7}\right]\right| \end{aligned} \tag{3.16}$$

Integrating $\mathbf{F}_s$, choosing the constant of integration so that $u_s(r_c) = 0$, the potential becomes

$$u_s(r) = \begin{cases} u(r) - u(r_c) - (r - r_c)\Delta F & r \le r_c \\ 0 & r > r_c \end{cases} \tag{3.17}$$

The shifted-force potential and the corresponding force is plotted in Figure 3.4.

One should be aware that the shifted-force potential function is qualitatively different from the original. If this potential is used to map the properties of a real liquid, corrections should be included (Nicolas et al., 1979).

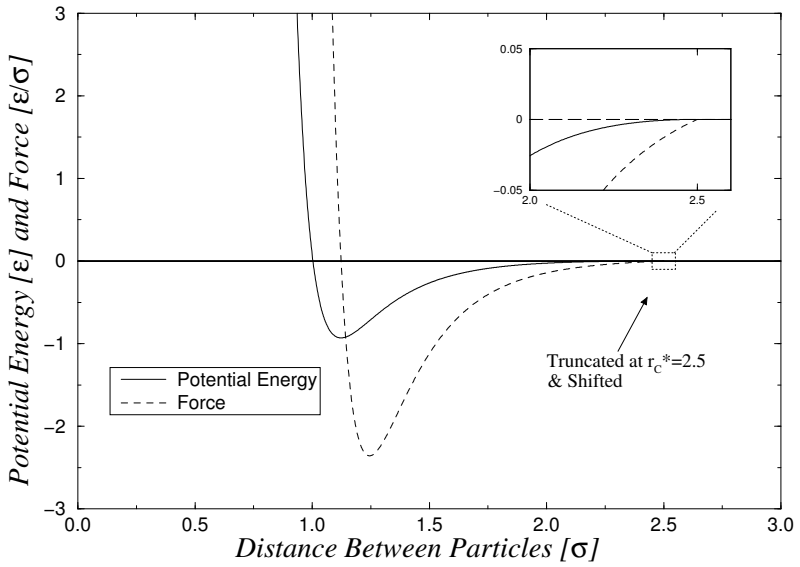

Figure 3.4: *The shifted-force Lennard-Jones (12,6) pair potential and force.*

Another negative aspect with the shifted-force potential is that it takes longer time to calculate during simulation.

In this thesis the shifted-force potential was used in order to reduce the fluctuations. When an isolated system was used to monitor the numerical errors due to truncation and low precision, it was important to keep the total energy as constant as possible (see Chapter 6).

3.5 Reduced Units

When running numerical simulations on a computer, very large or very small numbers often pose a problem. Even though the result of a mathematical operation should have been within an acceptable range, numbers in intermediate steps may be too large or too small, leading to catastrophic round-off or cutoff errors. In physics, an arbitrary choice of units (for example the SI units) often causes this problem, but by representing numerical values as multiples of appropriate physical quantities, the numbers can be made unit-less. By using these *reduced units*, most overflows can be avoided.

In molecular dynamics simulations of Lennard-Jones particles, there exists a standard set of reduced units based on the constants in the expression for the Lennard-Jones potential. Some of these are listed in Table 3.1 (Haile, 1997, p. 199). To convert the numerical value of a physical quantity into reduced units, simply divide the quantity by the conversion factor listed

Quantity	Conversion Factor	SI value for Argon†
Mass	m (mass of one particle)	6.6335×10^{-26} kg
Distance	σ (from LJ-potential)	3.41×10^{-10} m
Energy	ϵ (from LJ-potential)	1.65403×10^{-21} J
Heat Capacity	k_B (Boltzmann constant)	1.38066×10^{-23} JK^{-1}
Time	$\sigma\sqrt{m/\epsilon}$	2.1595×10^{-12} s
Velocity	$\sqrt{\epsilon/m}$	1.57907×10^{2} ms^{-1}
Acceleration	$\epsilon/(\sigma m)$	7.31217×10^{13} ms^{-2}
Force	ϵ/σ	4.85053×10^{-12} N
Temperature	ϵ/k_B	119.8 K
Number density	$1/\sigma^3$ ($1/\sigma^2$ for 2D systems)	2.5220×10^{28} m^{-3} (3D)

†(Maitland et al., 1981)

Table 3.1: *Reduced units in a three-dimensional system. To find the numerical value of a physical quantity in reduced units, simply divide the quantity by the conversion factor listed in the table. (Remember that the simulations discussed in this thesis are two-dimensional.)*

in the table. If reduced units are indicated by an asterisk (*), the distance and potential energy will become $r^* = r/\sigma$ and $u^* = u/\epsilon$, respectively. The Lennard-Jones potential from Equation 3.12 will then take the form

$$u^*(r^*) = 4\left[\left(\frac{1}{r^*}\right)^{12} - \left(\frac{1}{r^*}\right)^{6}\right]. \tag{3.18}$$

The numerical values used for σ and ϵ in the Lennard-Jones potential (3.12) do not affect the simulation results. For the two-dimensional simulations discussed here, all the reduced units from Table 3.1 hold except for the number density (σ^{-2} is used instead). By using standard reduced units, it becomes simpler to compare different simulations, and unnecessary duplicate runs are more easily avoided.

3.6 Boundary Conditions

3.6.1 Periodic Boundaries

The most commonly used boundary conditions are the periodic boundary conditions (PBC). The simulations in this thesis use PBC in the y-direction, and vertical walls in the x-direction (see Figure 3.1). This means that particles near the top of the simulation area feel the presence of the particles at the bottom, just as if they had been right next to each other. One can think of the system as an infinite two-dimensional tube or torus.

The PBC are considered to be the easiest boundary conditions, since they do not favor any particular frames of reference and therefore do not

impose any inhomogeneities on the system. Nevertheless, there exist several problems with the use of PBC. Not only must the interaction range between particles be truncated so that they cannot interact with themselves through the PBC, it is also important to prevent any particle from interacting with another particle twice. This truncation can often lead to errors, especially when considering long-range effects. These errors are more severe for slowly decaying interactions, such as the Coulomb force which goes as $1/r$.

3.6.2 Lennard-Jones Walls

Prior to running the system with heat transfer to the walls, a set of thermally isolating walls was needed to check that energy was conserved everywhere else. Later, they proved useful when comparing the results from the thermal walls.

The easiest way to make the walls insulating, was to use the Lennard-Jones potential*. This was because the potential function had already been implemented for the interparticle interactions.

A shifted-force potential field (Equation 3.17) was therefore inserted on the left and right side of the tube, as shown in Figure 3.5. The algorithm

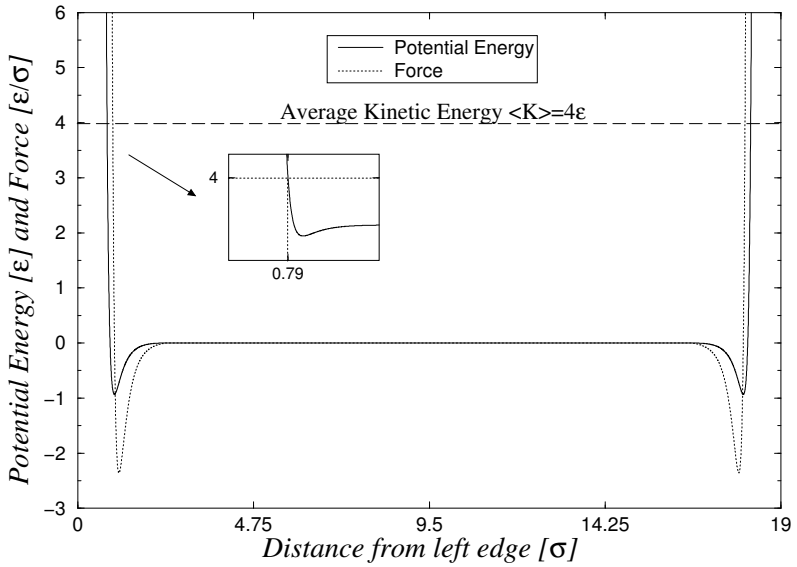

Figure 3.5: *The potential field and force imposed on the system by the Lennard-Jones walls. The force acts only in the x-direction.*

calculated the force on the particles by using their distance normal to the

*A force field that is derived from a continuous potential function ($\mathbf{F}(\mathbf{r}) = -\nabla u(\mathbf{r})$), conserves energy.

wall. Since any forces in the y-direction would cancel, this force only acted in the x-direction, leaving the particle momentum in the y-direction untouched. The force was added vectorially to the interparticle accelerations which had already been calculated. Afterwards the particles were integrated in the usual manner.

Theoretically, the particles could move around in a square area, each side 19σ long, with periodic boundary conditions in the vertical direction. In practice, the horizontal length was reduced to about 17.4σ due to the walls. The Lennard-Jones potential at the walls were translated in the x-direction so that the particle energies at $r = 0.8\sigma$ and $r = 18.2\sigma$ were $u(r) = k_B T$. This translation was done in order to make the bulk density as close as possible to that of the thermal walls (the thermal walls are discussed in the next section). The thermal wall zone boundaries were also placed at $r = 0.8\sigma$ and $r = 18.2\sigma$.

3.6.3 Thermal Walls

The thermal walls were initially implemented to create friction and temperature control. One can also imagine that a particle hitting a granular surface would collide several times with the wall particles before being re-emitted. The re-emitted particle would on average have the same temperature as the wall particles. This gives a physical motivation for using the thermal walls.

In the tube, zones were placed on the left-hand and right-hand sides, extending 0.8σ from the edges. The particles were not directly affected by these zones before they entered them. When a particle crossed a zone line, it was given a new velocity randomly sampled from a velocity distribution. If the particle had not re-entered the tube within the next time-step, it was moved out. Figure 3.6 illustrates how the thermal walls work.

Four different types of thermal walls were used in the main simulations. They are listed in Table 3.2 as wall types $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ for easy reference. They will be explained in more detail below. There are also other thermal walls that were implemented but only used during program development. These will be discussed briefly in the next chapter.

Wall type $\mathcal{A}$ is the one considered to have the "correct" distributions. To be more precise, it uses the distribution of velocities one would find if one measured all the particles crossing a vertical line in the middle of the bulk. The PDFs used in Table 3.2 (Equations 2.20 and 2.23) were calculated in Section 2.2. Measurements from simulations both in the bulk and at the walls seem to support this choice of distributions, and Section 7.4 shows that wall type $\mathcal{A}$ gives the most uniform temperature profiles. Nevertheless, there might be minute modifications that should be made to the PDFs due to a higher average density near the walls.

In Section 7.4 an attempt is made to explain the behavior of the temperature profile when other distributions are used. It thus seemed natural to

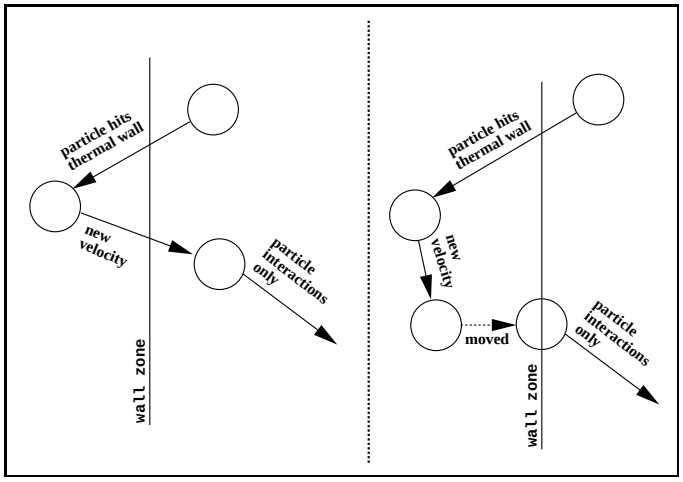

Figure 3.6: *The algorithm used for thermal walls. The figure to the right shows what happens if a particle stays inside the wall zone for more than one time-step.*

exchange either $f_V(v)$ or $f_\Theta(\theta)$ with a delta function. Wall type $\mathcal{B}$ re-emits all the particles normal to the wall, but keeps the distribution of speeds. Wall type $\mathcal{C}$ does the opposite; the particles re-enter the tube at different angles, but with one particular speed v_0. The final wall type, $\mathcal{D}$, is not stochastic at all. All the particles here are re-emitted normal to the wall with a speed of v_0. In the two latter wall types where the speed is constant, the RMS speed v_0 is used, given by

$$\begin{aligned} v_0^2 &= \int_0^\infty v^2 \tilde{f}_V(v)\, dv \\ &= \int_0^\infty v^2 \frac{4}{\alpha^3\sqrt{\pi}} v^2 e^{-\frac{v^2}{\alpha^2}}\, dv \\ &= \frac{3}{2}\alpha^2 \\ &= \frac{3k_BT}{m}. \end{aligned} \tag{3.19}$$

The intention is to keep the average energy output the same as in the first two wall types, and thus obtaining the same temperature.

It should be mentioned that in the implementation of wall type $\mathcal{A}$, the velocities were actually sampled from the Cartesian velocity distributions 2.17 and 2.18:

$$\tilde{f}_{V_x}(v_x) = \tfrac{2}{\alpha^2} v_x e^{-\frac{v_x^2}{\alpha^2}} \quad \text{and} \quad f_{V_y}(v_y) = \tfrac{1}{\sqrt{\pi}\alpha} e^{-\frac{v_y^2}{\alpha^2}}. \tag{3.20}$$

This did not affect the results in any way. If the velocities for wall type $\mathcal{A}$ had been sampled from the distributions listed in Table 3.2, they would

Type	**Description**	**Probability Density Functions**	
		$\tilde{\mathbf{f}}_{\mathbf{V}}(\mathbf{v})$	$\tilde{\mathbf{f}}_{\mathbf{\Theta}}(\theta)$
$\mathcal{A}$	Correct distribution for particles crossing a line.	$\frac{4}{\alpha^3\sqrt{\pi}} v^2 e^{-\frac{v^2}{\alpha^2}}$ 	$\frac{1}{2}\cos\theta$ 
$\mathcal{B}$	Particles re-emitted perpendicular to wall at various speeds.	$\frac{4}{\alpha^3\sqrt{\pi}} v^2 e^{-\frac{v^2}{\alpha^2}}$ 	$\delta(\theta)$ 
$\mathcal{C}$	Particles re-emitted at one particular speed in various directions.	$\delta(v - v_0)$ 	$\frac{1}{2}\cos\theta$ 
$\mathcal{D}$	Particles re-emitted at one particular speed perpendicular to wall.	$\delta(v - v_0)$ 	$\delta(\theta)$ 

Table 3.2: *Different types of thermal walls. The $\delta()$ represents the delta-function.*

still have to be decomposed into their horizontal and vertical components. The reason for using this other implementation was because it had a more intuitive appearance (at least initially), and because it was slightly more efficient to compute.

Chapter 4

The Simulation Program

In this chapter, the programs used to simulate and analyze systems using molecular dynamics will be explained. This includes both the algorithms and the data structure, and some of their limitations. The full listing of all the programs can be found in Appendix B.

The programs include a simulator and several additional algorithms used to analyze the output data. All the programs were written in the C++ language (see for example Stroustrup (1997)). The simulations were run on a Silicon Graphics computer, model O_2, with 128 megabytes of memory, and a MIPS R5000 main processor and floating point coprocessor running at 180 MHz (Silicon Graphics, Inc., 1998). For each of the simulation runs described in Chapter 7, about 6 hours of computation time was needed.

4.1 Outline

The main program simulated particles in a two-dimensional box or tube*. Both periodic boundary conditions and various walls were used. The particles interacted through a Lennard-Jones (12,6) potential.

The main program started off by *initializing* variables such as the particle positions and momenta, and setting up the data structure. In addition some initial data were output to disk. After initialization, the program started the main loop, which consisted of three parts: the calculation of forces, the movement of particles according to Newton's laws, and the recording of results.

During *force calculation*, the program checked each pair of particles that could be close enough to interact. For any such pair, both particles received an acceleration† calculated from their distance of separation. If a particle

*When speaking of the whole simulation area, the term "tube" is used throughout the thesis. The term "box" will hereafter refer to a part of the tube (see Section 4.2)

†Force and acceleration are proportional quantities, the mass of the particle being the proportionality factor, and these terms are therefore used interchangeably (see Equa-

experienced more than one interaction, the accelerations were added vectorially.

The next step was to move all the particles according to the accelerations that were calculated in the previous step. This was done by *integrating* Newton's laws of motion numerically, as explained in Section 3.3. After all the particles had been moved, the program was again ready to calculate the forces between the particles. Thus, the positions of the particles determined the movements, which in turn controlled the new positions.

In the final step of the loop, information was recorded. The positions and momenta of the particles were either output for later analysis, or used directly to calculate velocity distributions, velocity profiles and a particle density profile. Special care was taken during the first cycles of the loop while equilibration was taking place. Only the information needed to monitor the approach to equilibrium was recorded.

After iterating the loop a great number of times, typically ten million, any results calculated during the run were output. After the main simulator had finished, the rest of the data had to be analyzed. The reason why not all the analyzing was done during simulation is that the form of the output may depend on the results. One thus saves computation time, since it is the simulation that is the most computation intensive part of the process.

4.2 Data Structure

The data was grouped into objects to make the structure easier to control. The main data objects were the particles and the walls. In addition, the square area where the simulation took place, the tube, was divided into smaller square boxes. This was done to reduce the number of times the distance between two particles had to be calculated. In the simulations discussed in Chapter 7, there were 144 particles, two walls, and 16 boxes.

Each particle object held the position and the velocity of the particle. These objects were stored in one array and numbered sequentially. The same was done for the boxes[‡], which also were data objects. In addition to keeping the box coordinates, the box objects would point to one of the particles inside the box. This particle would in turn point to another particle. The pointer list would continue thus until all the particles in the box were included. The last particle would point to a NULL pointer. In this way the program could easily process all the particles in any given box (see Figure 4.1).

The pair potential through which the particles interacted was set to zero at the cutoff length and beyond, as described in Section 3.4. After dividing the tube into boxes, each box was assigned a list of all the neighboring boxes. As a starting point, this "neighborhood list" would include all the

tion 3.2).

[‡]A "box" refers to a part of the tube, as described earlier.

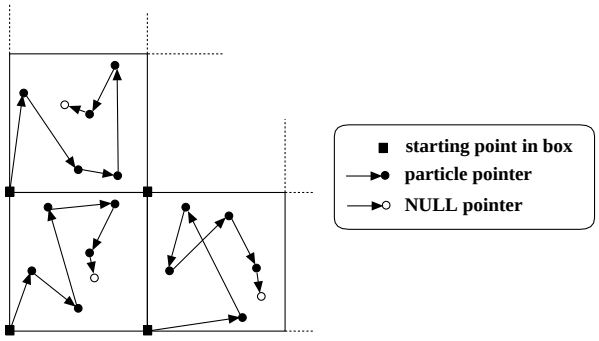

Figure 4.1: *Pointer structure. All the boxes point to a list of the particles inside. The empty circles signify that there are no further particles in that box, and represent the NULL pointers.*

surrounding boxes closer than the cutoff length, thus excluding the boxes where there was no chance of interaction (see Figure 4.2). In addition, lower numbered boxes were removed from the list for higher numbered boxes so that no two boxes pointed to each other. This was to avoid calculating a pair of particle interactions twice.

During the force calculations, the array of boxes was traversed. For each box, the particles within were considered first. Each particle in the pointer list was paired with all the others that followed in the list, but *not* with the preceding ones. Secondly, all the particles in the current box were paired with all the particles in all the boxes that were included in the "neighborhood list". By following this procedure for every box, all the particles within cutoff were paired once, and only once, for each time-step. Since there were only two walls, all the particles were checked against both.

After the particle forces were obtained, the integration was done by running through the array of particles, considering each particle separately.

4.3 Recording Information

In order to interpret and analyze data produced from the simulation, it had to be recorded on disk. The program can be divided into four stages: initialization, equilibration, the main loop and termination.

During initialization, data was only output for error checking and to record the parameters for later reference. This included the preset temperature and density, potential and force graphs, how particles and boxes were connected in the data structure, and precalculated tables used for the thermal walls.

The next stage was equilibration, where the main output was Boltzmann's H-function (see Section 7.2). Sometimes it was instructive to mon-

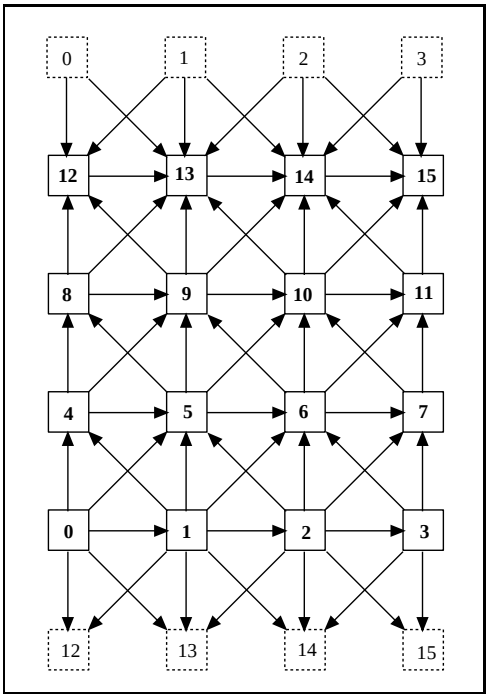

Figure 4.2: *Box neighborhood. How the boxes were connected in the data structure, the "neighborhood list".*

itor the kinetic and total energy as well, but only to check the approach towards equilibrium.

After equilibration, the main part of the simulation started. From the particle point of view, little was changed from the equilibration, but the program started to collect information about the system. The kinetic and total energies were written to disk, but to save disk space, the values were coarsely binned (coarse-grained) first, usually in blocks of 50 or 100. In some simulations, the velocity distributions of the particles that collided with the walls were recorded. The program also included the facility to record other properties, but these were mainly used during the development of the program.

Not all information was immediately recorded on disk. The distribution of velocities, the temperature profiles and the density profiles, and some times also the radial distribution function, were calculated during simulation and output to disk after the main part had finished. Since it could be difficult to determine the number of bins to use, all the profiles and distributions were recorded with 3600 bins. These could then easily be coarse-grained afterwards by almost any binning factor, as desired.

4.4 Random Sampling

It is possible to simulate a stochastic system by taking random samples from a probability density function, that is, picking numerical values for the stochastic variable. There should be a higher probability of selecting a value where the density function is large than where it is close to zero.

One way to sample from a probability density function is to map a uniform distribution on to the inverse cumulative distribution. Take for example the two-dimensional Maxwellian speed distribution, given by Equation 2.16. The cumulative distribution is found by integration:

$$F_V(v) = \int_0^v f_V(u)du = 1 - e^{-\frac{v^2}{\alpha^2}}. \tag{4.1}$$

As long as $f_V(v) > 0$, in this case for $v > 0$, $F_V(v)$ is strictly increasing and one can find the inverse function:

$$v(F) = \alpha\sqrt{-\ln(1-F)}. \tag{4.2}$$

Since $F_V \in [0, 1\rangle$, velocities will be sampled with the correct probabilities if F is chosen randomly from a uniform distribution of length one. The procedure is illustrated in Figure 4.3.

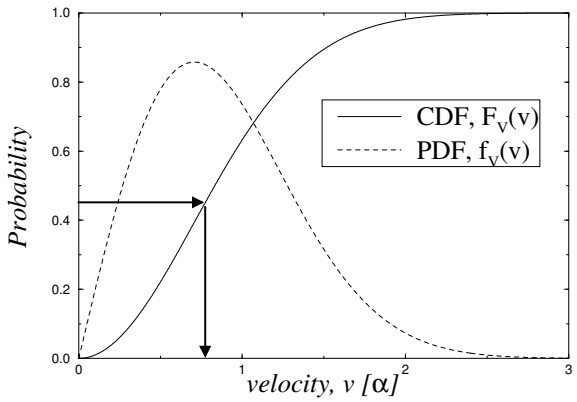

Figure 4.3: *Random sampling. The velocity can be sampled stochastically by uniformly choosing random values in the range* $[0, 1\rangle$ *for the CDF probability (y-axis), and then projecting down onto the velocity axis. The graphs shown in this example are the Maxwellian PDF (Equation 2.16) and its CDF (Equation 4.1).*

4.5 Precalculated Tables

When sampling random velocities for the thermal walls, the inverse cumulative function was needed, as described in the previous section. Unfortu-

nately, the cumulative function cannot always be found analytically, as is the case with both Equation 2.12 and 2.20. It can be found numerically, but such calculations are often more time-consuming.

Instead of calculating the inverse cumulative function every time a particle hit the wall, over $32,000$ values were precalculated and stored in an array. One might argue that this would distort the distribution compared to the double precision used in the floating variables, since only some values $(32,000)$ would be represented. An array containing more than 10^{16} bins would be needed to match the precision of a double floating variable.

Even though the precalculated tables seemed to work satisfactorily with the thermal walls, the problem of low precision in tables should be treated seriously. For a period during the development of the program, precalculated values were used for the interparticle potential function as well. When checking the fluctuations in the total energy in an isolated system (see Chapter 6), the use of these precalculated tables caused an unacceptable low accuracy.

Chapter 5

Development

The intention of this chapter is to give some insight into what was done prior to the main simulations described in Chapter 7. Hopefully, it will give the reader a better understanding of the motivation and an explanation of the reason for the choice of methods and systems used in the final runs. The path to understanding the methodology is far from straight, and I hope this chapter will help others avoid the worst detours.

5.1 Flow in a Tube

My initial task in molecular dynamics was to write a program to simulate a two-dimensional gas flowing through a tube. The tube had periodic boundary conditions in the y-direction, and vertical walls on either side in the x-direction. These walls were initially Lennard-Jones walls (see Section 3.6.2), and were therefore thermally isolating. A constant gravitational-like force in the y-direction was added to initiate and maintain the flow.

To prevent the force from accelerating the particles indefinitely, and to keep the system at a stationary flow, a mechanism for energy dissipation had to be supplied. A natural approach was to introduce friction at the walls, and thermal walls seemed like a good physical candidate. A thermal wall acts as a large heat reservoir by giving all the particles that collide with the wall a new, random velocity weighted by a velocity distribution. This velocity distribution has only one parameter, namely the temperature. If there were no flow (that is, no external constant force), the particles re-emitted from the wall should on average have the same velocity distribution as particles in the bulk of the fluid (see Section 3.6.3 for a more detailed description of thermal walls).

Before running the simulations with the constant force, the system and program code were checked under equilibrium conditions. After some minor corrections everything seemed to work satisfactorily, except for the average temperature and the shape of the temperature profile during simulations

with thermal walls.

In the first simulations with Lennard-Jones walls, both the density and temperature were given by the initial configuration, and no attempts were made to rescale the velocities. Later it became clear that the temperature was extremely high at about $T^* \approx 3000$, but this does not invalidate the qualitative conclusions reached in the initial simulations. One should only be aware that at such high temperatures, the Lennard-Jones particles probably behaved more like hard spheres.

5.2 Thermal Walls

The first set of thermal walls used, re-emitted the particles with angles sampled from a uniform distribution, and the magnitude of the velocities sampled from the Maxwellian speed distribution (Equation 2.16). The uniform distribution was chosen for simplicity, and was not motivated by any particular physical assumption. The temperature of the thermal walls was set to the same value as had been found during the simulations with Lennard-Jones walls. By choosing the same temperature, the systems could be compared, and the same time-step could be used.

The first anomaly found was the unexpected low average temperature at $T^* \approx 2000$. After discussing the problem, I believed I had found the source of the error. As explained in Section 2.2, the velocity distribution of particles crossing a line (or colliding with a wall), is given by Equation 2.20 rather than 2.16. The mean $\langle v^2 \rangle$ of these two distributions are $3\alpha^2/2$ and α^2 respectively (representing the average energy output).

I later found an article by Tehver, Toigo, Koplik & Banavar (1998) where they pointed out the difference between these two distributions. There are several published articles in refereed journals where the wrong distributions have been given, as in the articles by Risso & Cordero (1997) and Du, Li & Kadanoff (1995).

Unfortunately, correcting these distributions did not settle matters. The temperature was measured to be somewhat higher than $T^* \approx 3300$. After looking thoroughly for bugs in the program code, a longer simulation was run to analyze the temperature profiles. Surprisingly they were not flat, as seen in the simulations with the Lennard-Jones walls, but showed a higher temperature close to the walls. It turned out that using an incorrect distribution at the walls caused the temperature profile to become non-uniform. Section 7.4 addresses this phenomenon.

Even though the *speed* distribution was now (at least theoretically) correct, the arbitrary choice of using a uniform distribution for the angles proved to be incorrect when carrying out the calculation. The correct distribution is given by Equation 2.23.

5.3 Moving Particles Out of the Wall Zones

After correcting the thermal wall distributions, new simulations showed a noticeable improvement, but the profiles were still not as straight as for the Lennard-Jones walls. I learned from Tenenbaum, Ciccotti & Gallico (1982) how they always moved the particles that collided with the walls back into the tube after assigning them new velocities. In my previous simulations, any particle entering a wall zone would be assigned a new velocity but not moved. If this particle did not leave the wall zone during the following time-step, it would again be assigned a new velocity. This would continue until it had re-entered the tube. Figure 5.1 illustrates the difference.

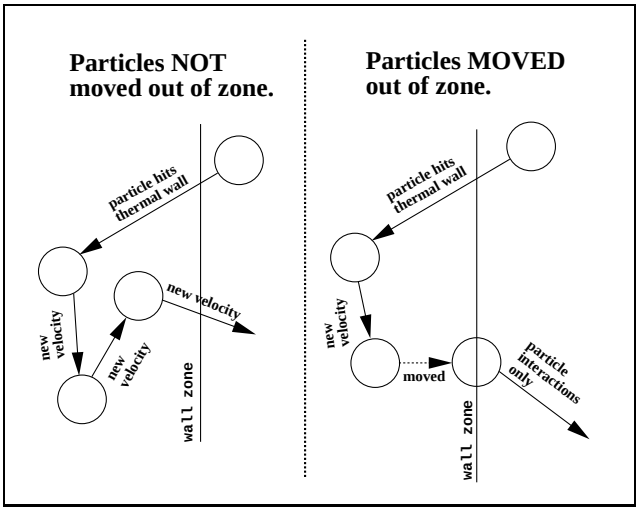

Figure 5.1: *Difference in the algorithms for the thermal walls when they are moved out of the wall zones and when they are not.*

Figure 5.2 shows plots of the distribution of angles of particles leaving the wall zones in simulations where the particles were not forced back into the tube immediately after hitting a thermal wall. The theoretical distributions plotted for comparison shows that on average particles tend to leave the walls at an angle more perpendicular to the wall than intended. The explanation of the discrepancies is that particles that have been assigned velocities almost parallel to the wall have less chance of leaving the wall zones. If the particles are not moved out into the tube, the distribution of the re-emitted particles becomes incorrect.

Moving the particles out of the wall zone does not give a perfect algorithm. Simulations were done using hard reflecting walls, which are equivalent to thermal walls except that the new velocities are not sampled randomly from a distribution. Instead, the old velocity in the direction perpendicular to the wall (which is the x-direction in these simulations) is reversed.

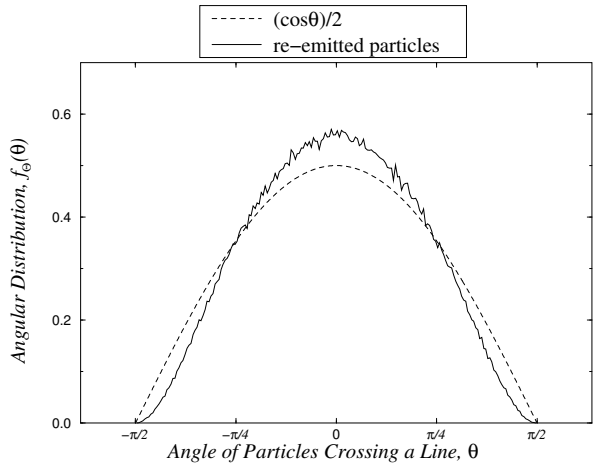

Figure 5.2: *The distribution of the angles with which particles leave thermal walls when they are* not *immediately moved back out into the tube.*

This way the total energy should be conserved. The problem is that by changing the coordinates of a particle, the potential energy of this particle is altered, and the total energy changes. Depending on the temperature and density of the gas, this might cause the total energy to increase. In an insulated system this will eventually lead to instability, halting the simulation program due to particles with extremely high kinetic energy (unless the velocities are rescaled regularly). The bottom line is that by forcing the particles out of the wall zone and back into the tube, one might be adding more energy to the system than intended, whether using hard reflecting or thermal walls. I believe, though, that for thermal walls, these effects are negligible.

Chapter 6

Fluctuations

Most measurements, be it from physical experiments or computer simulations, vary with time to some extent. If a property is measured on two different occasions, the values will most probably differ if enough precision is used. This chapter aims to provide a tool to measure and control such temporal fluctuations in computer simulations.

The fluctuations will be divided into categories according to their origin. Numerical *errors* in the calculations will produce fluctuations similar to the ones present in the total energy of an isolated system. The kinetic energy, on the other hand, will always fluctuate to some extent. The latter will be called *natural* fluctuations.

The next chapter will present and analyze temperature profiles of both an insulated and several temperature-controlled systems, using Lennard-Jones walls and thermal walls, respectively. It is important to know the size of the fluctuations in order to choose the number and length of time-steps, as well as estimating the error present in the final results.

First, the total energy in an isolated system will be used to measure the fluctuations due to numerical errors in the calculation. The discussion is then focused on the kinetic energy as a measure of the natural fluctuations. In the limit of infinitely long simulations, the *average* kinetic energy is supposed to remain constant. This chapter will look at how this limit is approached as well as assessing to what extent such a constant average kinetic energy exists.

The temperature profiles which will be discussed in Chapter 7, show the average kinetic energy in different sections of the tube. Even though some parts of the profiles might fluctuate more than others, the total kinetic energy and its fluctuations should at least be roughly proportional to those of the averages in the temperature profile bins. This is why the total kinetic energy is discussed so thoroughly in this chapter.

6.1 Numerical Errors

The numerical errors will be monitored by looking at the fluctuations in the total energy in an isolated system. In this thesis the terminology applied by Haile (1997, p. 151) will be used, where the numerical errors are divided into truncation errors and round-off errors.

The origin of truncation errors lies in the finite time-step and truncation of the Taylor expansion of Newton's equations of motion. A Taylor series can be written as

$$f(t+\delta t) = f(t) + \frac{df(t)}{dt}\delta t + \frac{1}{2!}\frac{d^2f(t)}{dt^2}(\delta t)^2 + \frac{1}{3!}\frac{d^3f(t)}{dt^3}(\delta t)^3 + \cdots \qquad (6.1)$$

At some point the series must be truncated, and the first missing term determines the truncation error. If, for instance, the second derivative is the first omitted term, the error varies as $(\delta t)^2$. This is exactly what happens for the velocity in the leap-frog integration scheme (Equation 3.6). Clearly the truncation error will grow smaller as the time-step length is reduced, and will vanish as the time-step becomes infinitesimal.

Molecular dynamics simulations are carried out by linearizing the movements into small, finite time-steps. The truncation error can thus be visualized by imagining that particles in one time-step move, say, from a place with a low potential field to a location where the potential energy is high. Clearly a smaller time-step will reduce this jump, and in the limit of an infinitesimal time-step, the particle will feel the potential field as continuous.

Round-off errors are due to the finite precision of the computer program. The program in this thesis used double (64-bit) precision for all the floating-point variables, which corresponds to roughly sixteen significant figures. When the time-steps are long, the truncation errors should dominate, but when the time-step becomes sufficiently small, the precision of the variables will not be able to properly separate one time-step from the next.

Thus on the one hand, the time-steps should be as small as possible to avoid truncation errors. On the other hand, if the time-steps are too short the finite numerical precision of the computer contributes with round-off errors, making the simulation less accurate compared to runs of the same time-span but with longer time-steps. In addition, the smaller the time-step the longer the simulation takes to run.

In Figure 6.1, the time-step δt^* is plotted against the average fluctuations in the total energy. First, the local discrete derivatives of the total energy were found by dividing the difference of consecutive values, $E^*[t_{n+1}] - E^*[t_n]$, by the time-step length δt^*. The root mean square of these derivatives gave the fluctuation values $\langle \frac{\delta E^*}{\Delta t^*} \rangle_{RMS}$ used in the graph:

$$\left\langle \frac{\delta E^*}{\Delta t^*} \right\rangle_{RMS} = \sqrt{\frac{1}{M-1}\sum_{n=1}^{M-1}\left(\frac{E^*[t_{n+1}] - E^*[t_n]}{\delta t^*}\right)^2} \qquad (6.2)$$

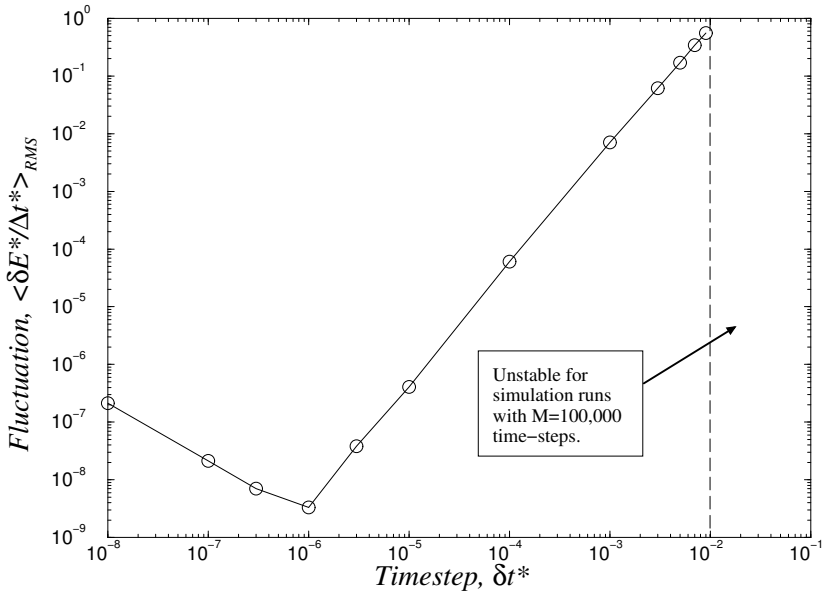

Figure 6.1: *Length of time-step against average fluctuation in energy per time. The fluctuations are relative to time, not to number of time-steps. The rise in fluctuations on the left hand side is due to numerical round-off errors, while the right hand side is inaccuracies from truncation errors. All the points were calculated from the total energy in an isolated system (Lennard-Jones walls) using double precision variables at* $T^* = 4$, $\rho^* \approx 0.4$ *and* $M = 100,000$ *time-steps after equilibration.*

These values indicate the relative numerical fluctuation in total energy between two simulation runs with different time-step lengths δt^* but equal time span Δt^*.

The rise on the right-hand side of the graph reveals how the truncation error affects the fluctuation. Curve-fitting shows that the slope of the graph is approximately two, meaning that the error varies as $(\delta t^*)^2$. This is the same as the truncation error found theoretically for the leap-frog velocity.

The increase on the left hand side is due to numerical round-off error. The slope on this side is approximately -1, or an error varying as $1/\delta t^*$, which seems reasonable since an extra decimal place is needed for every decade.

Looking at Figure 6.1, it seems clear that choosing a time-step at the bottom of the curve would give the best conservation of total energy. On the other hand, choosing a time-step that is too small means unnecessarily long simulation runs. The length of the time-step depends on how long simulations one can run, how much error one can tolerate and the property

in question. This thesis is mainly concerned with temperature profiles.

As mentioned in Section 3.4, the shifted-force potential was preferred to the truncated Lennard-Jones potential in order to reduce fluctuations in the total energy. If the latter had been used, the fluctuations would be greater because the particles would experience a sudden drop or rise in the potential energy every time they passed the cutoff length. The kinetic energy would not have time to adjust to this infinitely rapid change no matter how small the time-step was.

Section 4.5 briefly mentions that the use of precalculated values for the Lennard-Jones potential would also increase the error. Binning the potential function introduces many small "jumps" in the potential energy between each bin. These jumps are similar to the one found at the cutoff length r_c for the truncated Lennard-Jones potential, only smaller. A precalculated Lennard-Jones potential was in fact used during the development of the program, and the result was that the minimum value of Figure 6.1 became too high; there were no acceptable time-step lengths. In this case, the only remedy would be to increase the precision of the variables.

6.2 Standard Error of the Mean

A first approach to measure the normal fluctuations in the kinetic energy, is to look at the unbiased estimator S of the standard deviation of the instantaneous values K_i:

$$S = \frac{1}{M-1} \sum_{i=1}^{M} (K_i - \bar{K})^2. \tag{6.3}$$

Here M is the total number of time-steps after equilibration, and

$$\bar{K} = \frac{1}{M} \sum_{i=1}^{M} K_i \tag{6.4}$$

is the usual average. If the kinetic energy values K_i are independent and sampled from a distribution with a constant average,

$$\lim_{M \to \infty} = \bar{K} = \text{constant}, \tag{6.5}$$

the error in the mean can be estimated by

$$S_{\bar{K}} = \frac{S}{\sqrt{M}}, \tag{6.6}$$

often called the *standard error of the mean**. If an average is calculated for several datasets, the standard error gives an estimate of their standard deviation about the theoretical mean.

*More information on the standard error can be found in for example the books by Larsen & Marx (1986) and Squires (1994).

Once the error in the total kinetic energy is known, one can estimate the order of error in the temperature-profile bins. Given that there exists a true average (that is, the average remains constant), and that the density is uniform[†], the error in the bin values with respect to the "true" temperature profile should be

$$S_{\text{bin}} = S_{\bar{K}}\sqrt{B}, \tag{6.7}$$

where B is the number of bins in the temperature profile. It will be shown later that the average is in fact not constant in an isolated system, but will "diffuse" in a way similar to a random walk due to numerical errors.

6.3 Autocorrelation

The estimate of the standard error (Equation 6.6) assumes that all the instantaneous values K_i are statistically independent. This is not the case for two consecutive values of the kinetic energy in a molecular dynamics simulation, and $S_{\bar{K}}$ becomes incorrect.

The autocorrelation function $C(t)$ in Equation 2.30 measures the correlation between two values separated with a time t. Figure 6.2 shows a plot of the kinetic energies and the corresponding autocorrelation functions for an insulated system with Lennard-Jones walls and a system coupled to a heat bath via thermal walls of type $\mathcal{A}$. The time it takes for two values to become independent, the relaxation time, is for the isolated system $\Delta t^* \approx 0.1$, or about 20 to 50 time-steps, at $\delta t^* = 0.003$. The reason why two values close in time are correlated is that it takes time for the system to reach a significantly different configuration. The relaxation *time* is relatively independent of the length of the time-step δt. Choosing a shorter time-step would require a longer run if the number of independent realizations of the system was to stay constant.

For the system with thermal walls the relaxation time is much longer with $\Delta t^* \approx 30$, or about $10,000$ time-steps, at $\delta t^* = 0.003$. The total energy is no longer constant, and this results in greater fluctuations and a longer relaxation time for the kinetic energy. This is an important result that one should be aware of when simulating systems with thermal walls. It is reasonable to believe that this result holds for non-isolated systems in general; one should at least be aware of the possibility.

Notice the zoom of the autocorrelation function in the lower right graph of Figure 6.2. The shorter relaxation time found in the insulated system is also present here, but is overwhelmed by the thermal fluctuations from the wall.

[†]The density is in fact not uniform at the walls, but it is assumed that this causes minimal effects. These estimates are only supposed to give the order of magnitude of the errors.

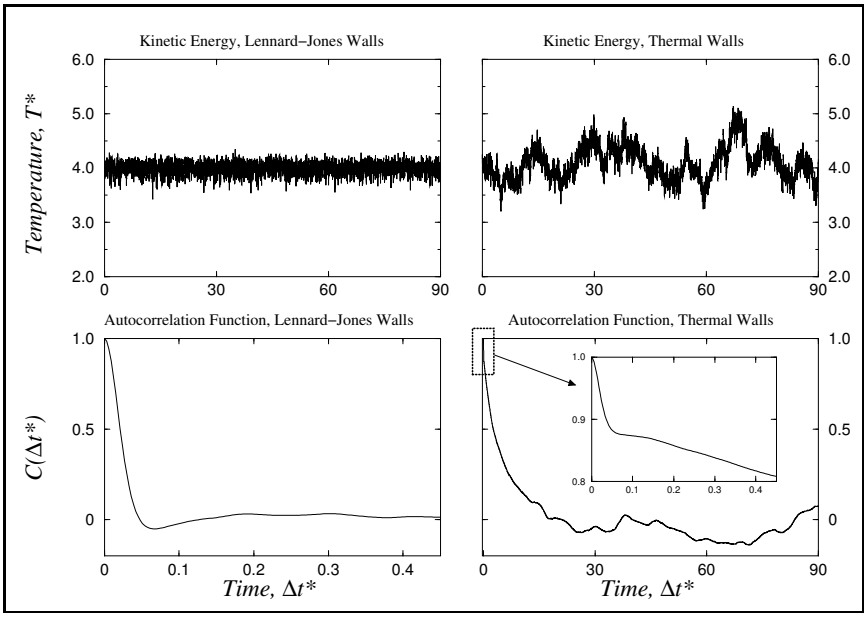

Figure 6.2: *A comparison of the kinetic energy and the corresponding autocorrelation function for a system with Lennard-Jones walls and one with thermal walls type $\mathcal{A}$. Notice the difference in scale on the x-axis of the lower left graph. The simulations were run at $T^* = 4$, $\rho^* \approx 0.4$ and $\delta t^* = 3 \times 10^{-3}$.*

6.4 Estimating the Standard Error by Coarse-Graining

Realizing that the instantaneous values of the kinetic energy K_i are not uncorrelated, invalidating the estimate for the standard error given by Equation 6.6, how does one proceed to find the correct value for $S_{\bar{K}}$?

Flyvbjerg & Petersen (1989) present in their article a *general* method to tackle the problem of correlated values. Consider a property with natural instantaneous fluctuations but a well-defined constant mean, such as the kinetic energy in an isolated system. As explained above, the problem is that consecutive values might be correlated, and Equation 6.6 breaks down.

To overcome this obstacle, Flyvbjerg & Petersen (1989) coarse-grain the sample (in this case all the instantaneous values of the kinetic energy K_i) by averaging pairs of consecutive values (thus halving the number of data points), and calculate the standard error of the mean. They continue thus until there are only two data points left. In the beginning, the standard error will increase, but after some iterations of this coarse-graining, two consecutive points should no longer be correlated and the standard error will stay relatively constant. According to Flyvbjerg & Petersen (1989), the constant value (within error bars) at the plateau represents the "true"

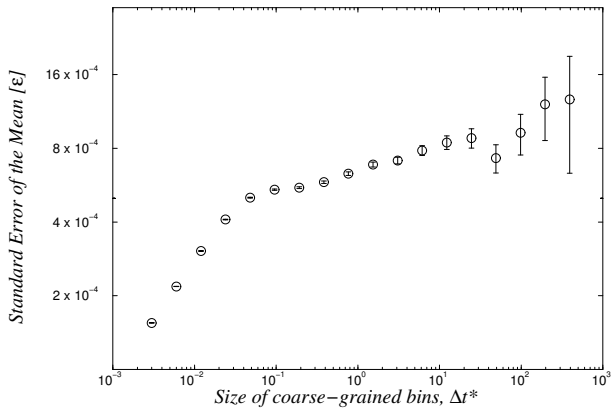

Figure 6.3: *A log-log plot of estimated standard errors of the mean while coarse-graining the instantaneous values of the kinetic energy. According to Flyvbjerg & Petersen (1989), the constant value (within error bars) at the plateau represents the "true" standard error of the mean. The plot was made from a simulation with Lennard-Jones walls,* $M = 500,000$ *time-steps (after equilibration),* $\delta t^* = 3 \times 10^{-3}$, $T^* = 4$ *and* $\rho^* \approx 0.4$.

standard error of the mean. Figures 6.3 and 6.4 show this method applied to the kinetic energy of an insulated system with Lennard-Jones walls and a system with thermal walls, respectively.

There is of course the problem that every time the set is coarse-grained, the number of data points used to calculate the standard error is halved and the error in the standard error increases. A usual estimate for the fractional error in the standard error is[‡] $1/\sqrt{2(n-1)}$, where n is the number of data points involved (Squires, 1994, p. 26).

As explained, after some iterations of coarse-graining, the estimate for the standard error should become constant and remain so within the error bars. If such a value is not reached, Flyvbjerg & Petersen (1989) argue that the simulation has not been run for a sufficient amount of time.

6.5 "Diffusion" of the Mean

This section discusses the effect of cumulative numerical errors. By looking at Figure 6.3, one can see that the plateau is not flat, although the slope has decreased dramatically. Figure 6.5, which is also the kinetic energy in an isolated system but with a slightly larger time-step, shows clearly that the graph reaches a plateau, but then starts to increase again. What is

[‡]It is assumed that the values have a Gaussian distribution, which is the case after some iterations of coarse-graining due to the central limit theorem.

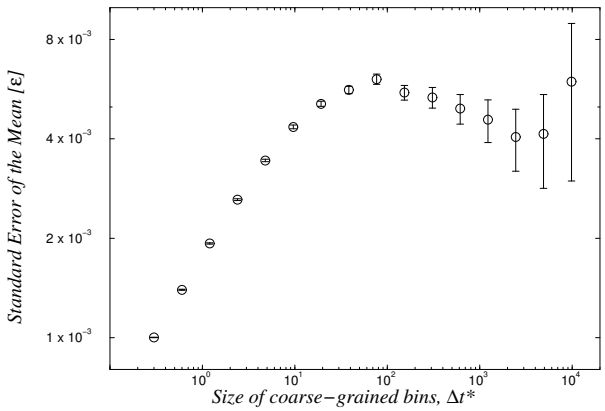

Figure 6.4: *A log-log plot of standard errors of the mean when coarse-graining the kinetic energy of a system with thermal wall type $\mathcal{A}$. The simulation used $M = 10,000,000$, $\delta t^* = 3 \times 10^{-3}$, $T^* = 4$ and $\rho^* \approx 0.4$. The first point is already coarse-grained from 100 values.*

the explanation for this effect? And would the graph perhaps reach a new plateau if a longer simulation was run?

Flyvbjerg & Petersen (1989) do not discuss what happens if no true average value exists. In an isolated system, numerical errors might accumulate, translating the "true" average value a little for every time-step. If this translation were random, the average value would "diffuse" in a way similar to random walk. How would this affect Figure 6.3 and 6.5? During the first coarse-grainings only fluctuations with a short relaxation time are averaged. The estimates for the standard error will therefore behave as before. When the coarse-grained bins become large enough to notice the fluctuations due to the cumulative numerical errors, the standard error will again start to increase. The explanation for the increase at the first iterations and after the plateau is the same: Since two consecutive values are no longer independent, the fluctuations per *time* do not differ much; a coarse-graining halves the number of data-points, so the standard error increases by a factor $1/\sqrt{n}$.

The following computer experiment supports the above explanation. Keep in mind that the method presented by Flyvbjerg & Petersen (1989) is a general one; it is not restricted to fluctuations in the kinetic energy. This also applies to the following discussion of the impact of numerical errors on the estimate of the standard error.

First, several datasets were generated by sampling values from a uniform distribution in the range $[-0.5, 0.5\rangle$. In addition, the accumulation of numerical error, or "diffusion", was modeled by translating the range by a small random number for every sampled value. The range was *not* reset back to the origin for every time-step, but moved relative to its previous po-

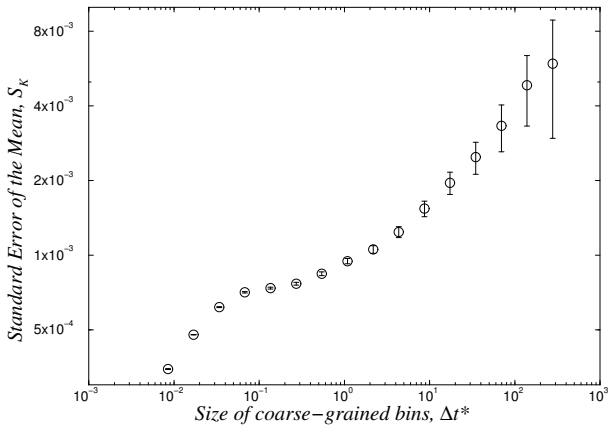

Figure 6.5: *A log-log plot of estimated standard errors of the mean while coarse-graining the instantaneous values of the kinetic energy. Notice that the curve starts to increase again after it has reached the plateau. The plot was made from a simulation with Lennard-Jones walls,* $M = 100,000$ *time-steps (after equilibration),* $\delta t^* = 8.5 \times 10^{-3}$, $T^* = 4$ *and* $\rho^* \approx 0.4$.

sition. For example, if the first value was sampled from the range $[-0.5, 0.5\rangle$, the next value would perhaps be sampled from the range $[-0.51, 0.49\rangle$; the next value from $[-0.505, 0.495\rangle$ etc. This small shift of the range was also sampled from a uniform distribution, but with a much smaller range. The standard errors of the means are plotted in Figure 6.6, where the ranges of the uniform distributions for the shifts are given in the legend (the program that made these datasets is listed in Appendix B as "sample_dist.cc").

All the datasets are initially independent, and should therefore be at the plateau already before any coarse-graining is performed. For the dataset where the average remains constant (no "diffusion"), the graph stays flat within error bars. As the range is allowed to "diffuse" and the existence of a limiting average is removed, the graphs start to increase according to power laws, showing straight lines in the log-log plot. The slope of the graph with the highest "diffusion" is approximately a half, showing that the standard error of the mean grows as the square-root of the simulation length. This behavior is similar to what is observed in Figure 6.5.

This result is significant for the standard error of the mean. Numerical errors causing a "diffusion" of the "true" average kinetic energy will result in an estimated average value $\bar{K}$ that will fluctuate more from simulation to simulation, that is, the average value becomes increasingly inaccurate. It is therefore important to keep the "diffusion" as small as possible.

A popular way of controlling "diffusion" in an isolated system, is to

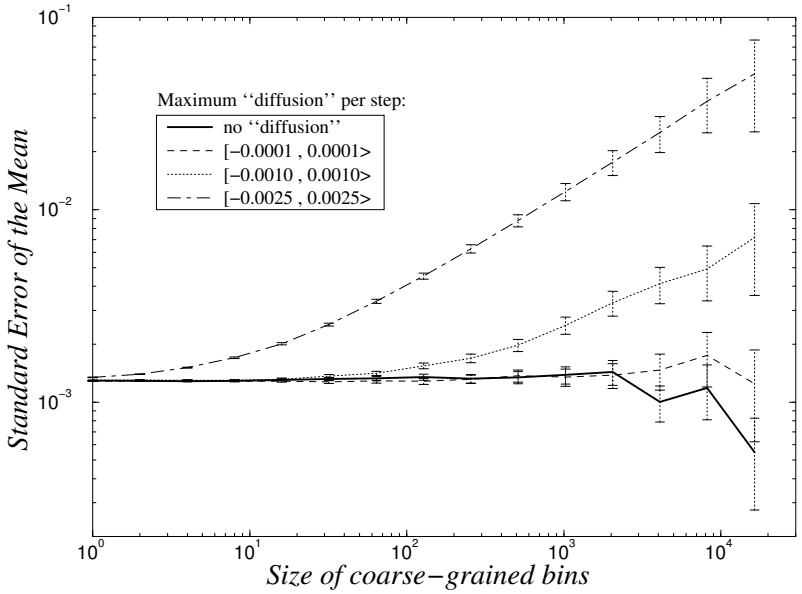

Figure 6.6: *A log-log plot of standard errors of the mean when coarse-graining datasets sampled from a uniform distributions with a range of* $[-0.5, 0.5\rangle$. *"Diffusion" was added by translating the range with a small random number picked from a uniform distribution whose range is given in the legend. All the datasets consist of* $50,000$ *data points. Notice how the standard errors increase when "diffusion" is added. Except for the "diffusion", the data points are sampled independently, and this is why the standard errors have already reached the plateau at the start.*

rescale the velocities of all the particles at regular intervals. At first glance, this might seem like a fool-proof plan, but a rescaling algorithm should be constructed with care. In order to rescale the velocities, one needs to know the current temperature. This is done by averaging the kinetic energy for several time-steps. The problem is that in order to keep the total energy relatively constant (at least better than without rescaling), the kinetic energy should be sampled for quite some time. But that means that the "diffusion" will make this average inaccurate, which implies that the length of the intervals between rescaling is of importance.

The literature displays other perhaps more sensitive methods for keeping the temperature or kinetic-energy mean constant, often called thermostats (see for example Frenkel & Smit (1996, chapter 6)). These thermostats affect the system in a way similar to the thermal walls, making the system canonical. One should be aware, though, that by applying a thermostat, the relaxation time for the kinetic energy will in most cases increase dramatically, as shown in Section 6.3 for the thermal walls. It is also important

to understand that both the rescaling approach and the thermostats affects the "isolated" system so that the total energy no longer is constant.

Numerical errors will not cause this problem in a system with thermal walls. The reason is that the walls define the average value of the kinetic energy. Each time a particle collides with the wall, its velocity is adjusted to give the correct average value. The thermal walls pay a price, though. The number of time-steps needed before reaching a plateau in Figure 6.4 has significantly increased compared to the system with Lennard-Jones walls. The limiting value gives the standard error of the mean. If the value is too high, or the plateau never reached, longer simulations must be run.

6.6 Choosing a Time-step

How should one go about choosing a satisfactory time-step? It can be difficult to find a clear answer in the literature, though there are many discussions of how the time-step affects different simulation models.

First of all, the time-step should never be chosen to the left of the minimum value in Figure 6.1. Second, one should apply the method of coarse-graining presented by Flyvbjerg & Petersen (1989) to the property under investigation. The graph should reach a plateau, indicating that the simulation has run for a sufficiently long *time* to produce independent realizations of the system. If no plateau is reached, the long relaxation time has not allowed the system to fluctuate enough for the proper average to have been calculated. It is also important that the height of the plateau satisfies any requirements one should have for statistical accuracy of the property under investigation.

Thus, to reach the plateau one must have independent realizations of the system, and to increase the statistical accuracy of the mean, the number of independent realizations must be increased. For a system regulated by thermal walls or a thermostat, both these goals can be achieved by increasing the number of time-steps or making the time-steps *longer*, or a combination of the two. One should nevertheless be careful not to make the time-steps too long, since truncation errors might alter details in the trajectories. This is especially important when studying the dynamics of the system.

If the system is isolated, the "diffusion" of the mean can be reduced by making the time-step shorter. Increased statistical accuracy will then require more time-steps, but this might again lead to increased "diffusion". There is a possibility that there exist thermodynamical states where no combination of number and length of time-step would satisfy the statistical accuracy needed. Rescaling the velocities regularly or implementing a thermostat would confine the "diffusion". For the thermostat, the same arguments as those used for the thermal walls would then apply, including the drawbacks such as longer relaxation times and the fact that the system no longer is

isolated. Rescaling the velocities might not increase the relaxation time, but properties other than energy "diffusion" might change, giving undesirable effects, perhaps affecting the average values.

The main conclusion is that there does not exist a unique solution to the problem of finding the "best" time-step length. There are trade-offs between accuracy and the length of simulation runs, and between keeping the average temperature constant and having short relaxation times. Knowing what should be measured and understanding the effects of the different approaches is probably the best advice for obtaining acceptable results.

Chapter 7

Analysis of Simulation Results

The scene is now set for the main simulation runs. This chapter starts out by describing the setup and parameters of these simulations. This is followed by a short description of the equilibration mechanism. After that, a section is dedicated to the fluid structure of the different systems, giving an overview over different methods and explanations available in the literature for both homogeneous and inhomogeneous systems. The temperature profiles constitute perhaps the most exciting part of this thesis. An attempt to explain the non-uniform behavior of these profiles is made in the final part.

7.1 Initialization

Five different systems were simulated; one with Lennard-Jones walls, and four others with the thermal walls listed in Table 3.2. Each system was run for $M = 10,000,000$ time-steps, not including the time needed for equilibration. The time-step was set to $\delta t^* = 0.003$, making the whole simulation last for $\Delta t^* = 30,000$.

In all the five simulations, 144 Lennard-Jones particles were placed in a square grid. To make the structure somewhat random, the particles were displaced a small random amount from their original grid points. They were also given random velocities sampled from a uniform distribution. Compared to the Maxwellian, the uniform distribution was both simpler to implement and allowed the approach towards equilibrium to be monitored (see the next section).

Both the temperature of the thermal walls and the initial temperature for the system with the Lennard-Jones walls was set to $T^* = 4$. In the latter system, the velocities needed to be rescaled (during equilibration) if this temperature was to be reached.

The number density in the bulk was set to $\rho_B^* = 0.4$. One should be

aware, though, that since the walls extended $r^* = 0.79\sigma$ from either side, the actual bulk density was somewhat higher (see Figure 7.5).

The temperature and density were chosen so that the systems were certain to be in the gas phase. Later, an article presenting the phase diagram of the two-dimensional Lennard-Jones gas was found (Barker, Henderson & Abraham, 1981). This phase diagram is shown in Figure 7.1. The chosen phase point ($T^* = 4$ and $\rho^* = 0.4$) is located high above the graph, and lies clearly in the fluid region.

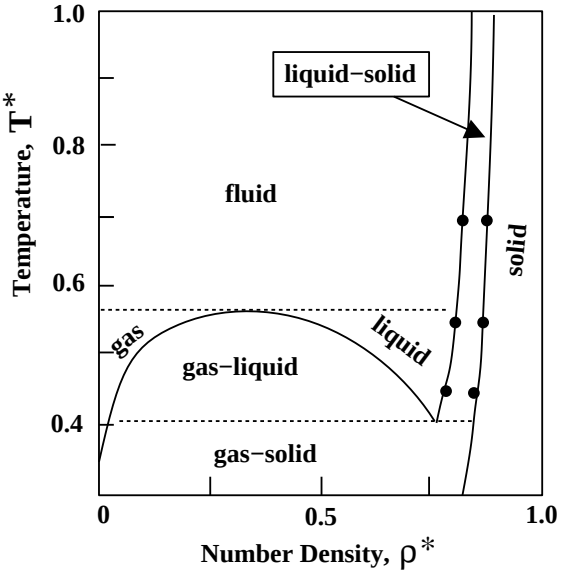

Figure 7.1: *Phase diagram in (ρ^*,T^*) plane for a two-dimensional Lennard-Jones fluid. Solid lines: perturbation theory and self-consistent cell model; Circles: Monte Carlo results. (Barker et al., 1981).*

In addition to the initialization of the physical parameters, the data structures explained in Section 4.2 had to be correctly initialized.

7.2 Equilibration

After initialization, the system needed to reach equilibrium before any measurements could be done. Initially, the gas had neither the right temperature nor the right velocity distribution. By rescaling the velocities, the approach to the preset temperature was speeded up. For the insulated system with Lennard-Jones walls the rescaling was absolutely necessary, since the temperature would remain constant otherwise.

There are several ways to monitor the approach to equilibrium. One of them involves Boltzmann's H-function

$$H(t) = \iint \rho(p,q,t) \ln[\rho(p,q,t)]\, dp\, dq, \tag{7.1}$$

where $\rho(p,q,t)$ is the time-dependent version of the distribution function described in Section 2.2. On average, $H(t)$ will decrease with time until it reaches its minimum value (McQuarrie, 1976, pp. 413–418).

In these simulations, the kinetic portion of the H-function was used to monitor the approach of the velocity distributions towards that of the Maxwellian:

$$\begin{aligned} H_x(t) &= \int_{-\infty}^{\infty} f_{V_x}(v_x) \ln[f_{V_x}(v_x)] dv_x \\ H_y(t) &= \int_{-\infty}^{\infty} f_{V_y}(v_y) \ln[f_{V_y}(v_y)] dv_y \end{aligned} \tag{7.2}$$

Thus by calculating instantaneous values of the H-function, one could see when the velocity distributions reached equilibrium. The theoretical equilibrium value for the H-functions can be obtained by inserting the Maxwellian distributions (Equations 2.12) into Equations 7.2.

To calculate these values from simulation, one must find the velocity distributions f_{V_x} and f_{V_y} by binning the particle velocities. One should be cautious when choosing the bin width. If the bin width is too small, the statistics will become poor. There will be too few particles in each bin, and one would not be able to see a Maxwellian distribution even if there was one there. The resulting value for the H-function would not become negative enough compared to the theoretical result. If the bins are too wide, all the particles will be located in a few bins, and one would not be able to notice any difference in the H-values as time progressed. In addition, the values would become too negative compared to the theoretical result.

Figure 7.2 shows the H_x- and H_y-functions for a system with thermal walls of type $\mathcal{A}$, using 15 bins in the velocity-range $[-2\alpha, 2\alpha]$. The H-values were then coarse-grained, averaging over 300 bins at a time. This was done in order to show more clearly the approach towards the equilibrium value. The straight line is the theoretical result. Note that in this figure, the velocities were never rescaled.

In the main simulation runs, the velocities were rescaled once every 1,000 time-steps until $M = 10,000$ or $t^* = 30$, making the approach to equilibrium quicker. Equilibration without rescaling was then permitted until $M = 100,000$ ($t^* = 300$).

7.3 Fluid Structure

Before examining the temperature profiles in detail, it seems wise to have some knowledge about the structure of the fluid. This applies both to inter-

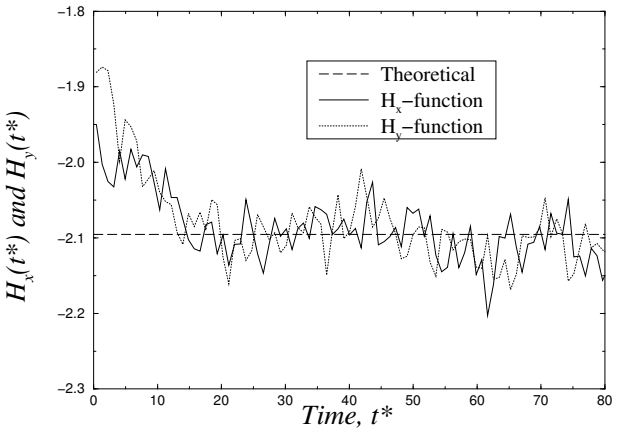

Figure 7.2: *Boltzmann's H_x- and H_y-functions for a system with thermal walls type $\mathcal{A}$, using 15 bins in the range $[-2\alpha, 2\alpha]$ for the velocity distributions. The instantaneous values were coarse-grained by a factor of 300 in order to show more clearly the approach towards the equilibrium value. The straight line is the theoretical result. The velocities were* not *rescaled in this example.*

particle correlation functions, such as the radial distribution function, and the density profile.

Much research has been done on simple homogeneous fluids, including investigations on structure. There are standard methods for calculating values such as the radial distribution function, as can be seen in any standard textbook on liquids or statistical mechanics. Since the advent of fast computers there has been also been substantial investigation into inhomogeneous fluids.

The following sections present the radial distribution function and density profiles for the simulation model used in this thesis. A brief overview over some of the existing literature on the field is also included. No attempt has been made to give an in-depth presentation of the field. Even though unsolved problems remain, all the fundamental theory seems to be well understood.

7.3.1 Radial Distribution Function

The radial distribution function $g(r)$ measures the local structure of the fluid. It can be used to determine if the fluid is in a state closer to a liquid than a gas, or perhaps even a solid. The value $g(r)dr$ is the conditional probability that, given a particle at the origin, a particle will be found in the interval ranging from r to $r + dr$. An intuitive way of regarding $g(r)$ is to imagine that the frame of reference is moved to one specific particle.

The radial distribution function then shows how this particle experiences the local neighborhood on average.

The local density can be found by multiplying $g(r)$ with the number density of the bulk ρ_B. The integration of $\rho_B g(r)dr$ over all distances (that is, the whole volume or area) gives the total number of particles present save one:

$$\begin{aligned} \int \rho_B g(r)\, d\mathbf{r} &= \int \rho_B g(r) 2\pi r\, dr \\ &= N - 1 \end{aligned} \tag{7.3}$$

The "missing" particle comes from the fact that the particle defining the frame of reference is never counted.

For a fluid at low density, $g(r)$ can be calculated theoretically (McQuarrie, 1976, p. 272):

$$\lim_{\rho \to 0} g(r) = exp\left[\frac{-u(r)}{k_B T}\right]. \tag{7.4}$$

Here $u(r)$ is the Lennard-Jones potential given by Equation 3.17. The radial distribution function approaches unity as r increases, showing that the local density becomes equal to the bulk density at large distances. Equation 7.4 is plotted in Figure 7.3.

For a crystalline solid state, which is in a dense regime, $g(r)$ would exhibit clearly defined peaks that would approach delta-functions as the temperature decreased. The particles would thus be clearly structured with strong correlations in space.

For a liquid, $g(r)$ would show significant structure through several layers of particles*. The reason for this layering, is that at $r = 0$ there is always a particle. This particle prevents any others from coming closer than about one radius. Here there will be, on average, a higher density of particles since there are no other particles closer to the origin that will spread them. This higher density will then reduce the chance of a new particle coming close until one has moved out another radius. Looking from a probabilistic point of view, the existence of the particle at the origin is certain. As one proceeds out from the origin, the effect of this particle diminishes, and in the limit the local density becomes uniform and equal to the bulk density.

Figure 7.3 shows the radial distribution function from a simulation with periodic boundary conditions in both directions at approximately the same temperature and density as the other simulations in this chapter. The reason for not using any walls, is that walls make the fluid inhomogeneous. The radial distribution function is designed to measure local structure in the bulk. The next section will discuss the density profiles where the walls are the main concern.

*When talking about "layers", one should think of it in a statistical sense; one would have problems making out these layers by only looking at a single snapshot.

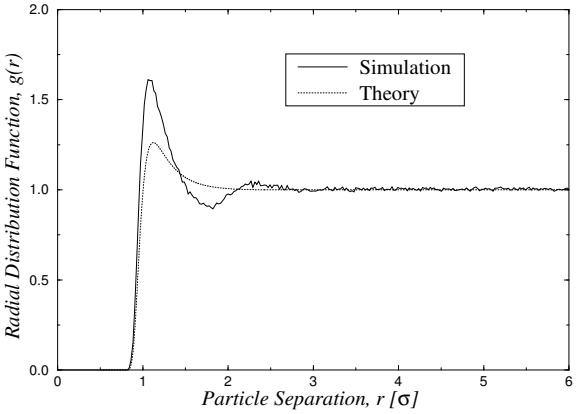

Figure 7.3: *The radial distribution function $g(r)$ for a simulated system, and the theoretical curve in the low-density limit. The theoretical curve is given by Equation 7.4. The other graph was obtained from a simulation with periodic boundary conditions in both directions, $T^* = 4$, $\rho_B^* = 0.4$, $\delta t^* = 3 \times 10^{-3}$ and $M = 100,000$ time-steps.*

One can see from Figure 7.3 that there is no long-range structure in the fluid, and there are no signs of correlation for $r > 3\sigma$. The main peak clearly defines a first layer, but there is only a hint of a second one. Even though the fluid is not a low-density gas, one can conclude that it is far from the critical point, where correlation lengths become very large.

There are several methods described in the literature to calculate the pair correlation function, such as the Kirkwood Integral equation, the Ornstein-Zernike direct correlation function, and pertubation theories (McQuarrie, 1976; Hansen & McDonald, 1976). These methods have mainly been developed for use in homogeneous fluids

An intuitive approach is described in an article from 1967, where Widom proposes that the structure of a homogeneous fluid depends on different parts of the interparticle potential. He suggests that the attractive part of the potential plays an important role on the structure factor when the fluid is close to the critical point. At higher densities, using the triple point as an example, the structure factor is mainly controlled by the repulsive part of the potential. The latter statement suggests the use of a so-called reference fluid, where only the repulsive part of the potential has been kept, and the attractive part is substituted by a uniform background potential field. The use of the reference fluid leads to easier calculations, and it also shows that hard spheres can be used to simulate conditions away from the critical point. Figure 7.4 compares the potential functions $u_{LJ}(r)$, $u_R(r)$ and $u_d(r)$ of the Lennard-Jones fluid, the reference fluid and the hard spheres,

respectively. The reference fluid can in many ways be compared to the interactions proposed by Van der Waals, although his theories were originally aimed at explaining behaviour at the critical point.

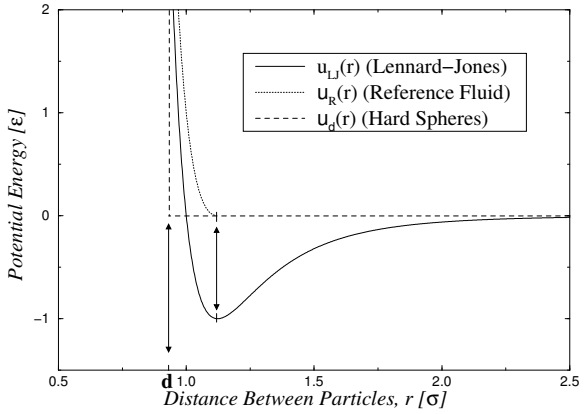

Figure 7.4: *A comparison of the potential functions $u_{LJ}(r)$, $u_R(r)$ and $u_d(r)$ of the Lennard-Jones fluid, the reference fluid and the hard spheres, respectively.*

Chandler, Weeks & Andersen (1983) explain how they have taken Widom's ideas further in what has come to be known as the WCA theory. They do quantitative measurements of the pair correlation function near the triple point of the full Lennard-Jones fluid, the reference fluid and the hard sphere fluid, and show that they are nearly identical, especially at distances greater than the atomic radius.

7.3.2 Density Profiles

Figure 7.5 shows the density profiles of the five simulated systems: The Lennard-Jones walls and the four thermal wall types. When the profiles were calculated during the simulation, each of them had 3600 bins. Afterwards, groups of six data points were averaged (or blocked, or coarse-grained) in order to smooth out the graph and make each new data point more accurate (by reducing the standard error of the mean). Each graph in Figure 7.5 consists therefore of 600 bins.

The most obvious feature is the inhomogeneities at the walls. The density profiles are quite similar to the measured radial distribution function in the previous section. A similar argument can be applied to explain the form of the profiles: No particles are allowed to be inside the wall. As one moves out, a region of high density forms since there are no particles in

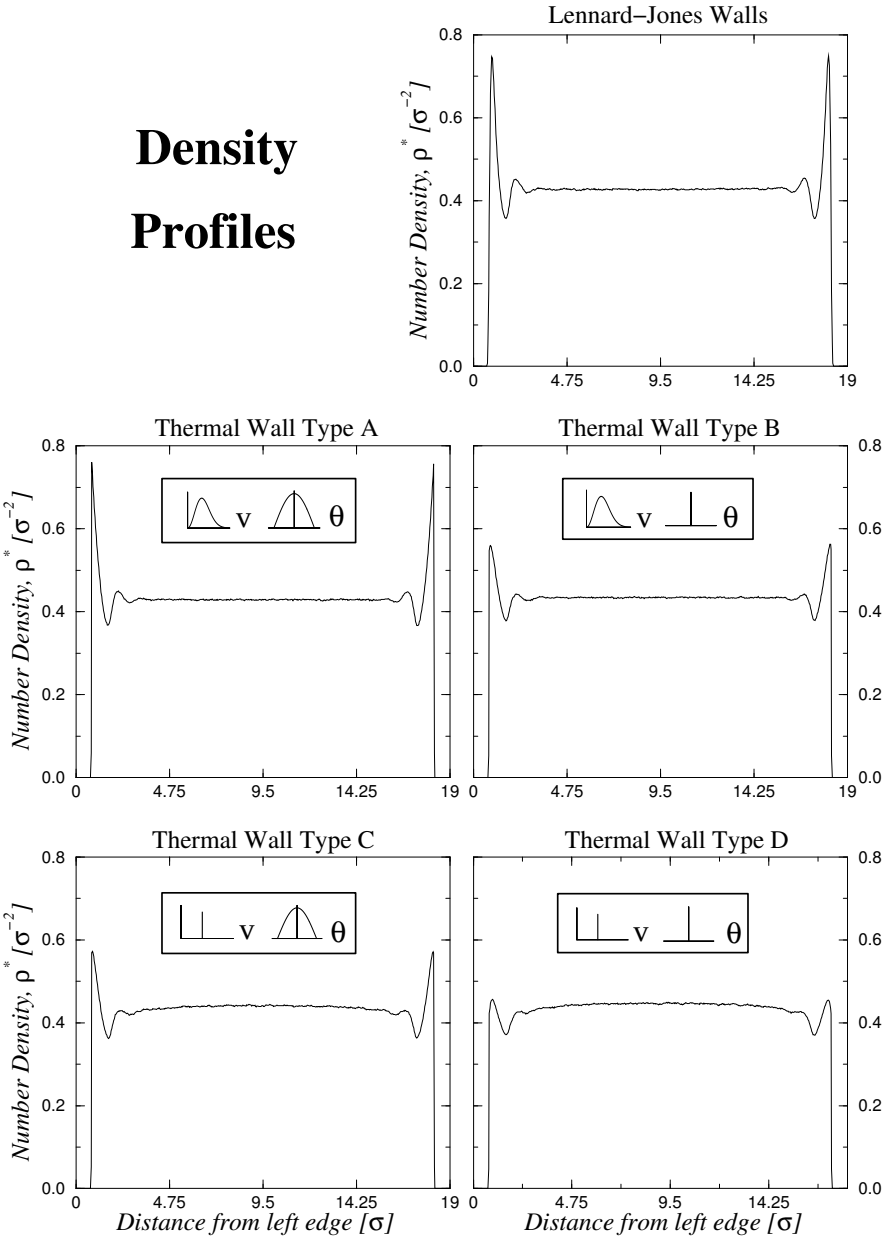

Figure 7.5: *The density profiles of five different systems. The graphs were obtained from simulations where* $T^* = 4$, $\delta t^* = 3 \times 10^{-3}$, *and* $M = 10,000,000$ *time-steps. There are 600 bins in each graph.*

between to spread out this first layer†. After the peak there is a trough due to repulsion from the first layer. Moving from the wall towards the middle, the density profiles show alternating peaks and troughs with diminishing amplitude. The wall effects stop fairly quickly in these systems, since the gas is quite rarified.

Looking closely at the graphs, one can see that the number density is slightly higher than $\rho^* = 0.4$ even though it was initially set to this value if compared to the total volume of the tube. The reason for the higher density is that the walls occupy space in the tube, restricting the space available for the particles. This is seen in Figure 7.5 where the graphs drop abruptly to zero on either side of the tube. For the thermal walls, the wall zone boundary was placed approximately 0.79σ from each side of the tube. For the Lennard-Jones walls, the potential field was adjusted so that particles would have a potential energy of $k_B T$ at this same distance, thus creating tubes with approximately the same areas.

In Figure 7.6, the left side of the density profile belonging to the system with Lennard-Jones walls is compared with the Lennard-Jones potential field. The profile has been inverted in order to make the comparison easier. One can see that the profile drops quite rapidly when the potential energy starts to rise. The (very few) particles that came closest to the edge had a potential energy of over 50ϵ.

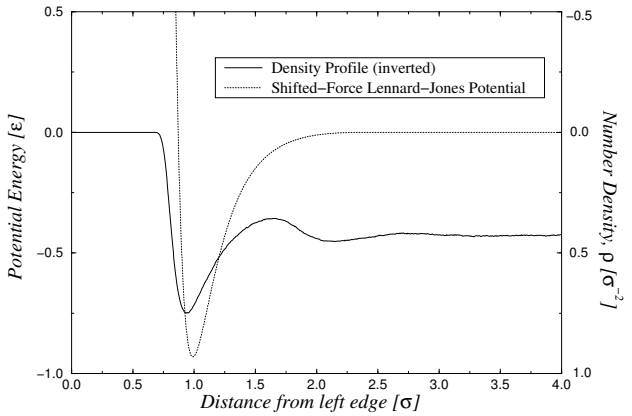

Figure 7.6: *A zoom of the density profile at the edge of a system with Lennard-Jones walls, compared with the potential field. The density profile is inverted for easy comparison.*

Figure 7.7 shows a similar graph for the system with thermal walls of

†Remember that the clearly defined peak, or layer, was calculated from many snapshots. It might be difficult to discern any layering from looking at only one realization, unless the system is in a solid or dense fluid region. The term "layer" is thus used mainly in a statistical sense.

type $\mathcal{A}$, which is believed to be the most correct thermal wall. Here the original density profile with 3600 bins was used to show how abruptly the density changes at the wall zone boundary. The expanded scale on the x-axis shows that the drop is far more rapid than for the Lennard-Jones walls.

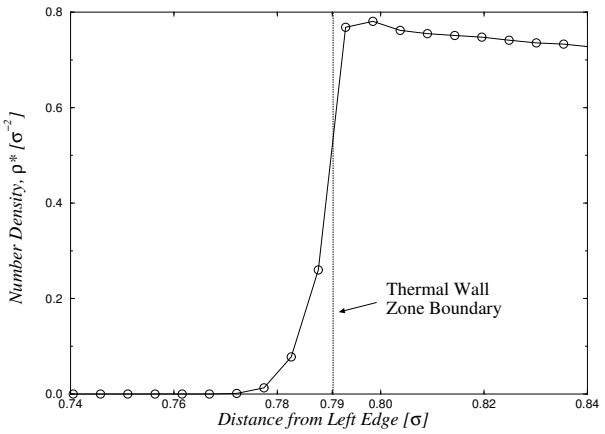

Figure 7.7: *A zoom of the density profile at the edge of a system with thermal walls of type $\mathcal{A}$.*

Comparing the density profiles for the Lennard-Jones walls and the thermal walls of type $\mathcal{A}$ in Figure 7.8, the graphs show a correspondence that is almost surprising. The thermal wall has a sharper peak near the edge, but this is to be expected since the potential field makes the Lennard-Jones wall more diffuse.

Inspecting the density profiles for the three other thermal walls, one notices that the peaks are less prominent. This is especially evident in wall type $\mathcal{D}$. In addition, the graphs of the systems using wall types $\mathcal{C}$ and $\mathcal{D}$ are slightly curved with the maximum at the center of the tube.

In an attempt to explain these features, attention has been given to the average velocity with which particles were re-emitted. At equilibrium, the velocity distribution of particles crossing a line (or re-emitted from a wall) is given by Equation 2.20. The average velocity is

$$\begin{aligned} \langle v \rangle &= \int_0^\infty v \tilde{f}_V(v)\, dv \\ &= \frac{2}{\sqrt{\pi}}\alpha \approx 1.13\alpha, \end{aligned} \tag{7.5}$$

and the RMS speed is (see Equation 3.19)

$$v_{RMS} = \int_0^\infty v^2 \tilde{f}_V(v)\, dv$$

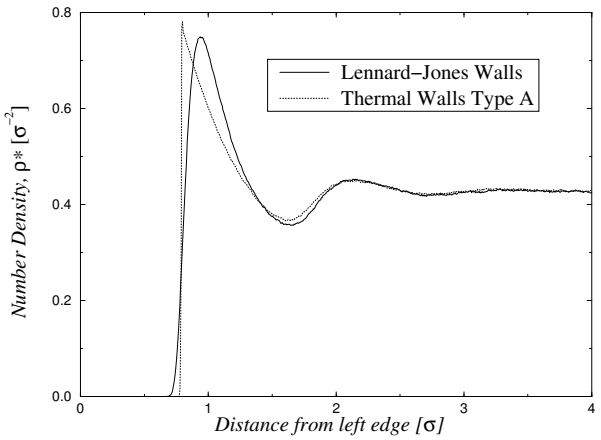

Figure 7.8: *A comparison of the left edge of the density profiles for the Lennard-Jones walls and the thermal walls of type $\mathcal{A}$. The plots were taken from the original profiles containing 3600 bins.*

$$= \sqrt{\frac{3}{2}}\alpha \approx 1.22\alpha. \tag{7.6}$$

Wall types $\mathcal{C}$ and $\mathcal{D}$ re-emitted all the particles at the RMS speed, $v_0 = v_{RMS}$, thus making the kinetic-energy output equal to that of the "correct" thermal wall $\mathcal{A}$. The problem is that, using only one particular speed v_0, the average speed $\langle v \rangle$ for the re-emitted particles in $\mathcal{C}$ and $\mathcal{D}$ becomes equal to the RMS speed:

$$\langle v \rangle = v_0 = v_{RMS}. \tag{7.7}$$

In a "normal" equilibrium system, such as the one with wall type $\mathcal{A}$, the average speed is less than the RMS speed:

$$\langle v \rangle = \frac{2}{\sqrt{\pi}}\alpha < \sqrt{\frac{3}{2}}\alpha = v_{RMS}. \tag{7.8}$$

This means that even though there is no net energy flow, the output of momentum from the walls in $\mathcal{C}$ and $\mathcal{D}$ is too high. In the middle of the tube, where the walls have less influence, the particles are closer to their equilibrium speed distributions, and the average velocity is lower. The curved density profiles are probably caused by the higher average velocity near the walls, driving the particles towards the center of the tube. It is thus believed that higher density in the middle is opposed by the output of a high average velocity at the walls.

The above explanation for the high velocities near the walls is also believed to be the reason for the low peaks in the system using wall type $\mathcal{C}$. In type $\mathcal{B}$, the low peaks are probably due to the fact that all the particles are

re-emitted normal to the walls, directing them further into the tube on average. In the system with walls of type $\mathcal{D}$, both the higher average velocity and the perpendicular re-emission is believed to contribute to the reduced peaks.

In the literature, the structure of a simple fluid near a wall has been calculated (Sullivan & Stell, 1978; Sullivan, Levesque & Weis, 1980) using the wall-particle Ornstein-Zernike integral equations, where several of the closure approximations to so-called hierarchical equations have been compared. Götzelmann, Haase & Dietrich (1996) use similar techniques, and claim to have good agreement with simulation data with some exceptions at high densities.

In uniform fluids, the attractive part of the interparticle potential can be replaced by a uniform background potential field since this part nearly cancels by symmetry. This is not the case for non-uniform fluids, where there is an external field $\phi(\mathbf{r})$. Weeks, Selinger & Broughton (1995) introduce an *effective reference field* (ERF) $\phi_R(\mathbf{r})$ that[‡], when combined with the original reference fluid, takes care of both the attractive part of the full Lennard-Jones potential and $\phi(\mathbf{r})$. By iterating their inhomogeneous force equation

$$\nabla_1[\phi(\mathbf{r}_1) - \phi_R(\mathbf{r}_1)] = -\int d\mathbf{r}_2 \rho_R(\mathbf{r}_2|\mathbf{r}_1;[\phi_R])\nabla_1 u(r_{12}) \tag{7.9}$$

using values from molecular dynamics simulations, they were able to predict density profiles near Lennard-Jones walls[§]. They have also done a similar study (Weeks, Vollmayr & Katsov, 1997) where they had a single stationary Lennard-Jones particle in the middle of the gas instead of a wall. They show that the WSB theory (where the ERF is used) gives better results than the WCA theory, also at lower densities where the WCA theory breaks down. In a recent article, Weeks, Katsov & Vollmayr (1998) calculate the ERF analytically, using linear response theory.

7.4 Temperature Profiles

As mentioned in Chapter 5, the temperature profiles became non-uniform when thermal walls re-emitted particles with a distribution different from that of the incoming particles. To achieve a better understanding of this phenomenon, five simulations with different boundary conditions were run. One of the systems used Lennard-Jones walls, thus conserving the total energy. The four others were the thermal walls of type $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$, as explained in Section 3.6.3. Type $\mathcal{A}$ is supposed to be the thermal wall with the correct distribution, re-emitting the particles with the same distribution of velocities as the particles hitting the wall. Type $\mathcal{B}$ used a constant angle,

[‡]In this particular article, the effective potential was named $\phi_0(\mathbf{r})$.

[§]Their walls used only the repulsive part of the Lennard-Jones potential.

type $\mathcal{C}$ had a constant speed, and type $\mathcal{D}$ combined these two, re-emitting the particle normal to the wall at one particular speed.

The temperature profiles of these five systems is shown in Figure 7.9. As with the density profiles, the temperature profiles were originally sampled in 3600 bins. They were then coarse-grained to increase accuracy and reduce fluctuations. Each graph in Figure 7.9 contains three curves with 400 bins in each curve. The solid line (T) is the ordinary temperature profile whose y-values are, since reduced units are used, equal to the average kinetic energy per particle. The two other curves (T_x and T_y) show the average kinetic energy in the horizontal and vertical directions. The two latter curves actually show twice the kinetic energy in order to make comparison with the temperature profile easy. Note that the "real" temperature profile (T) is in fact just the average of the two other curves.

Using the method of coarse-graining presented by Flyvbjerg & Petersen (1989) explained in Chapter 6, the standard error of the total kinetic means were calculated. Equation 6.7 was then used to estimate the errors in the bin values plotted in Figure 7.9. These values are listed in Table 7.1. Close to the walls, however, where the density profiles in Figure 7.5 are nearly zero, the errors are much higher due to the low number of particles per bin.

Wall Type	$\mathbf{S}_{\bar{\mathbf{K}}}$	$\mathbf{S}_{\text{bin}}$
Lennard-Jones Walls	5×10^{-4}	0.01
Thermal Wall $\mathcal{A}$	6×10^{-3}	0.12
Thermal Wall $\mathcal{B}$	6×10^{-3}	0.12
Thermal Wall $\mathcal{C}$	4×10^{-3}	0.08
Thermal Wall $\mathcal{D}$	4×10^{-3}	0.08

Table 7.1: *The standard errors in the temperature profiles calculated by coarse-graining the total kinetic energies.*

Figure 7.10 shows a close-up of the temperature profile from thermal wall $\mathcal{A}$. The horizontal line in the middle is a linear regression of the profile, and the two other lines show the error given in Table 7.1. The coarse-graining method seems to overestimate the errors to some extent, and can be interpreted as a safe upper limit on the error.

Before examining the temperature profiles of the systems with thermal walls type $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$, an explanation of the peaks on the edges at the Lennard-Jones walls and the thermal wall of type $\mathcal{A}$ will be given.

Figure 7.11 shows a close-up of the left side of the temperature profile for the system with Lennard-Jones walls. For this close-up, a profile containing 1800 bins was used. The potential field of the wall is plotted in the same graph. A logical explanation for the peak is that there were only a few particles that had a high enough kinetic energy to be *able* to reach out that

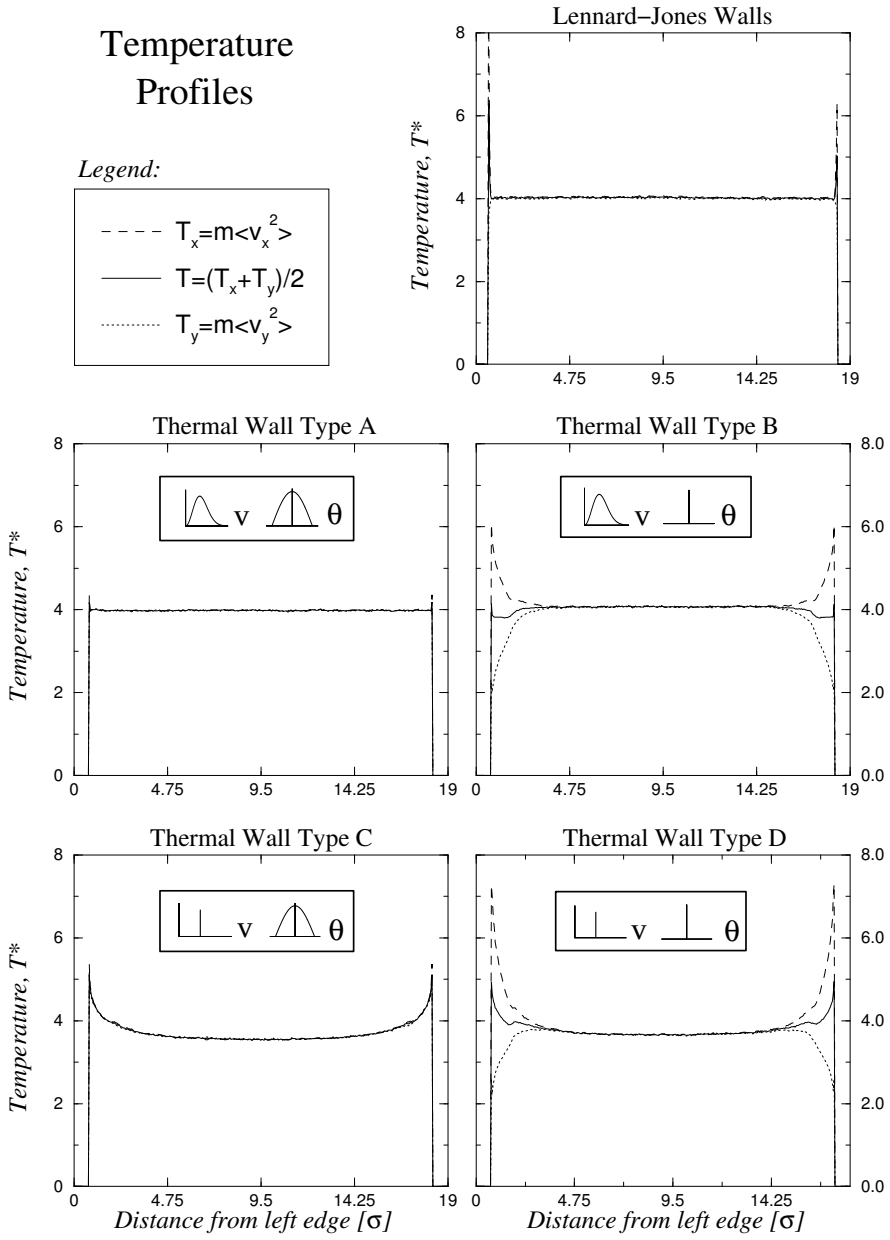

Figure 7.9: *The temperature profiles of the same five systems whose density profiles were shown in Figure 7.5. In addition to the normal temperature profile, the average velocities in the horizontal and vertical directions are also included.*

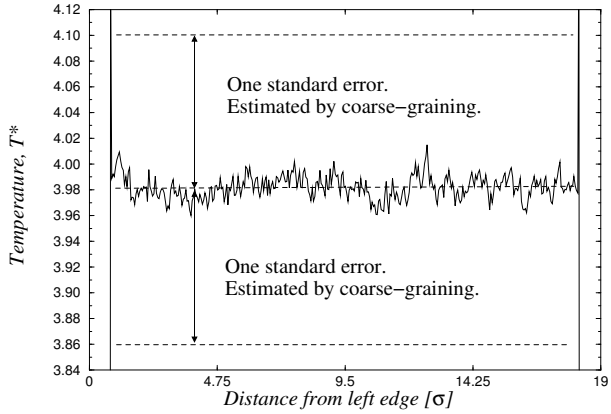

Figure 7.10: *A close-up of the temperature profile for the thermal wall type $\mathcal{A}$. The horizontal lines indicate the error estimated from the coarse-graining method described in Chapter 6.*

far. It was mentioned in the previous section that the particles farthest out had potential energies higher than 50ϵ.

An objection to this hypothesis is that when the particles had arrived thus far, all the kinetic energy would have been changed to potential energy. How could there be any kinetic energy left to form the peak? Consider a particle with a kinetic energy between 10ϵ and 50ϵ heading for the wall. Only a few of the particles that collided with the walls had such high speeds, the others turned back earlier. A high-energy particle, on the other hand, would reach far into the wall, slowing down in the process. Even though it would be slowing down, it would have a higher-than-average velocity on its way, making the temperature profile rise. This last explanation holds for most of the points in the peak save the ones closest to the edge (there are 13 bins to the left of the intersection with the potential field). Here the uncertainty is quite large since only very few particles reached this far, and it is possible that if another simulation was run the top of the peak could have had a significantly higher or lower value.

A peak is also present for the thermal wall type $\mathcal{A}$, and a zoom of the left edge of the temperature profile is shown in Figure 7.12. The reason why the peak in this figure is much higher than in Figure 7.9 is that the temperature profile in the latter figure has been coarse-grained down to 400 bins, as explained earlier. This close-up was taken from a profile with 3600 bins.

The explanation for the thermal wall peak is even simpler than for the Lennard-Jones-walls case. No particle stayed inside the wall zone for more than one time-step. The velocities recorded for the profile were thus the ones

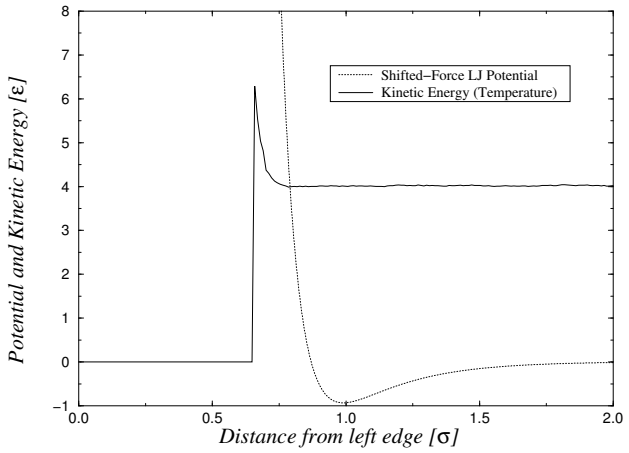

Figure 7.11: *A comparison between the temperature profile of the system with Lennard-Jones walls, and the potential field due to these walls. The reason for the higher temperature at the edges is probably because only particles with a high velocity can reach so far out into the steep part of the Lennard-Jones potential.*

with which the particles entered the zone. The higher the velocity was when they crossed the boundary, the deeper into the zone the particles penetrated on average. Notice the scale on the x-axis of the graph. The peak would diminish if δt^* was made smaller, vanishing as the time-step length reached zero.

To get an idea of how often a particle managed to penetrate deep into the thermal wall zone, the frequency during a $M = 10,000,000$ time-step long simulation has been added to the graph for some of the data points. Even though only a few particles reached far out, this does not mean that the value for this peak has a high uncertainty. Remember that in this case, the kinetic energies and the positions are highly correlated.

Having examined the features in the "normal" temperature profiles (the Lennard-Jones walls and the thermal wall type $\mathcal{A}$), the turn now comes for the three remaining thermal walls. Even though the edge effects just described also applies to these profiles, all three show a non-uniformity in the temperature reaching far beyond the limited area near the walls. Wall type $\mathcal{B}$ and $\mathcal{D}$ both have higher kinetic energies in the horizontal direction close to the wall. The reason is quite obvious; these wall types re-emit all the particles normal to the wall, giving the particles a high velocity in the x-direction, and no velocity at all in the y-direction. For wall type $\mathcal{C}$, the angular distribution distributes the kinetic energy equally between the two directions, and therefore its temperature graphs coincide.

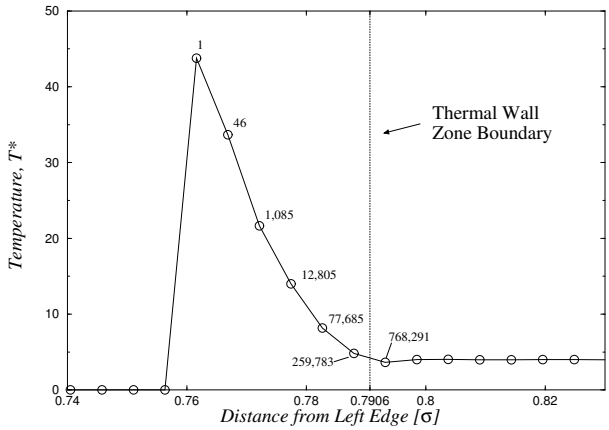

Figure 7.12: *A comparison between the temperature profile of the system with thermal wall type $\mathcal{A}$, and the thermal wall zone boundary. The reason for the higher temperature at the edges is simply that to be able to reach far out, the particles needed a high velocity. The numbers show how many times particles reached as far as the corresponding data points.*

What is the reason for the non-uniformity of the temperature profiles? Even type $\mathcal{C}$, where the distribution among the directions is correct, the profiles show a clearly lower average kinetic energy in the middle. In the following, wall type $\mathcal{C}$ will be discussed for two reasons. It has a more regular form, and more importantly, the profiles are equal for the horizontal and vertical directions (T_x and T_y). This basically reduces the problem from two to one dimensions.

In Section 2.1, the difference between a stationary state and thermal equilibrium was discussed. From the definition of equilibrium given, the non-uniformity of the profile suggests that the system is not in equilibrium.

At wall type $\mathcal{C}$, particles collide with a range of different speeds. The walls then re-emit the particles, but with only one particular speed every time. In the bulk, particles are able to move in a two-dimensional velocity space, represented for example by v_x and v_y, or by v and θ. When particles are re-emitted at only one speed, the speed-dimension in phase space is in effect removed.

If wall type $\mathcal{C}$ were replaced by wall type $\mathcal{A}$, the system would equilibrate to give a uniform temperature profile. System $\mathcal{C}$ can not be said to be in partial equilibrium either[¶], since Figure 7.2 indicates that the relaxation time needed to correctly redistribute the velocities among the particles is about $\Delta t^* = 30$, compared to the length of a run of $\Delta t^* = 30,000$.

If the system is no longer in equilibrium, but only in a stationary state,

[¶]Partial equilibrium is discussed in Section 2.1.

the assumption that the temperature and the average kinetic energy are proportional to each other becomes invalid. It would therefore be more appropriate to refer to the "temperature profiles" as "average kinetic energy profiles".

The second law of thermodynamics states that $\Delta S \geq 0$. Left to itself, a system will try to become as uniform, or disordered, as possible. Asymmetries usually disappear. An incorrect thermal wall, on the other hand, tries to re-establish some of this order. Wall types $\mathcal{C}$ and $\mathcal{D}$ forces all the particles to have only one particular velocity, and in wall types $\mathcal{B}$ and $\mathcal{D}$, the direction is focused. These boundary conditions thus reduce the entropy at the walls, restraining the system from reaching equilibrium.

The profiles, and especially type $\mathcal{C}$, seem almost to decay in a regular fashion. The range with which the walls affect the the bulk properties, meaning the steepness of the decay, is probably linked to the mean free path λ. Every time two particles collide, their energies are redistributed. After enough collisions, the particles have "forgotten" the thermal walls. A short mean free path means frequent collisions, and the particles should "forget" quickly.

The mean free path can be estimated from the number of collisions per length. A hard sphere of diameter d travelling a length L sweeps out an area $L2d$, meaning that it would hit any particle whose center was inside this area. At number density ρ, the number of particles in this area would be $L2d\rho$. The mean free path, or distance travelled per hit, would then become

$$\lambda = \frac{L}{L2d\rho} = \frac{1}{2d\rho}. \tag{7.10}$$

Using $\rho^* = 0.4$ as the number density, and $d^* = 1.8\sigma$ as the diameter (the radius of 0.9σ was chosen because at this distance, the particles have a potential energy of 4ϵ), the mean free path turns out to be about $\lambda^* = 0.7\sigma$. This is short compared to the dimensions of the tube, and means that particles need to crash several times before they lose their "memory" of the walls.

Figure 7.13 show how the velocity distributions change as one moves from the wall towards the middle in a system with wall type $\mathcal{C}$. The distributions are sampled from particles crossing lines at different locations in the tube, but only the particles moving away from the walls are included. The theoretical equilibrium distribution, which is given by Equation 2.20, is plotted for comparison. The theoretical curves have $T^* = 3.59$ rather than $T^* = 4$, since the former is the value of temperature profile $\mathcal{C}$ at the center of the tube (Figure 7.9). In addition, the upper-left graph shows the sampled velocity distribution of the particles colliding with the walls.

The velocity distribution is known at the walls, and one also knows the limiting distribution which is attained in the bulk. If one could find a way of calculating how the distributions change between the wall and the bulk,

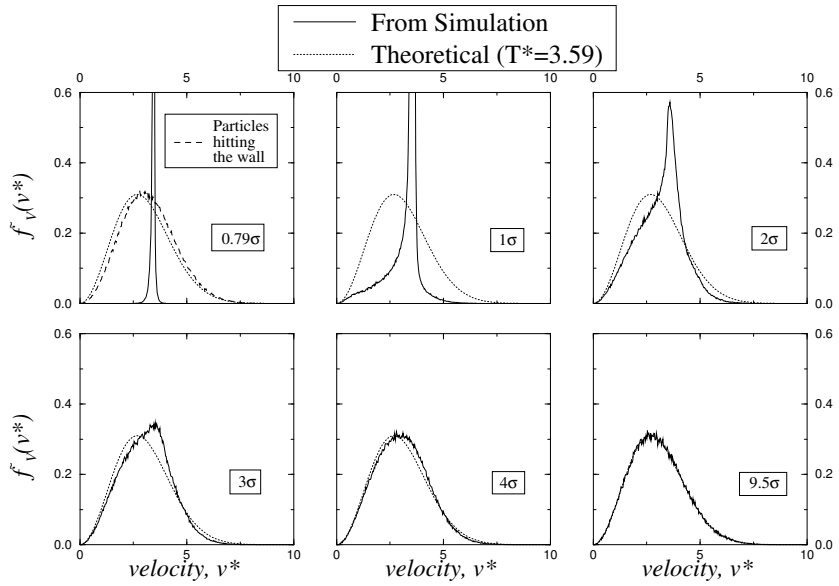

Figure 7.13: *Velocity distributions in wall type $\mathcal{C}$ of particles moving towards the center of the tube, crossing lines at $x = 0.79\sigma$, σ, 2σ, 3σ, 4σ and 9.5σ. The first and last of these lines are the wall zone boundary and the middle of the tube, respectively. The theoretical equilibrium distributions are also plotted. In the upper-left graph, the velocity distribution of the particles colliding with the wall is included.*

one could probably calculate how the average kinetic energy changes as well. A calculation based on the Boltzmann Equation (McQuarrie, 1976, p. 409) might turn out to be fruitful.

Such analytical calculations have not been done, but Figure 7.13 gives a rough description of the change. Moving from the walls towards the middle of tube, the peak decreases and the rest of the distribution grows, "filling out" the theoretical curve. At the same time, the velocity distribution transforms in such a way that the average kinetic energy changes. What controls this transformation? In Section 7.3.2 it was pointed out that for wall types $\mathcal{C}$ and $\mathcal{D}$, the average velocity of the particles leaving the walls should be higher than that found in the bulk. These high-velocity particles would collide with those moving towards the wall, transfering momentum and thus stopping or turning the large number of slowly moving particles (slow meaning below the emitting velocity v_0). The velocity distribution of the particles colliding with the walls, shown in the upper-left graph in Figure 7.13, is thus translated to higher energies. The temperature profiles therefore present a higher average kinetic energy close to the walls.

The irregularity seen in the temperature profile (T) of system $\mathcal{B}$ (notice

the similar irregularity in $\mathcal{D}$) is also caused by a thermal wall re-emitting particles with an incorrect velocity distribution. The explanation for these irregularities will not be pursued any further, but the point is that the results are sensitive to details in the implementation of the thermal-wall algorithms.

Chapter 8

Concluding Remarks

In this thesis I have mainly discussed a molecular dynamics simulation program that I have written "from scratch". During the development of the program there were especially two issues that needed an explanation. The first was what controlled the accuracy of the results, including the time-step length, numerical errors and correlated values. The other was how thermal walls affected the temperature profile.

The investigation of causes for inaccuracies showed that there is no unique "solution" that produces the best result. There are trade-offs, depending on what is important in the current research. When comparing round-off errors and truncation errors, the latter dominated unless poor precision, or in some cases precalculated tables, were used. A method was presented that found the standard error of the mean by coarse-graining, regardless of the initial correlation between values.

An important result concerning thermal walls was the dramatic increase of the relaxation time by more than a factor of 300 compared to an insulated system with Lennard-Jones walls. This has implications on the length of the run needed to get statistically acceptable averages.

Boltzmann's H-function was introduced to monitor the equilibration process and to measure the relaxation time of the velocity distributions due to particle interactions. Further, the fluid structure was investigated, both locally and as profiles. The local radial distribution function supported the phase diagram, showing that the phase point was located in the fluid region far from the critical point. A perhaps more surprising result was how similar the density profiles for the isolating Lennard-Jones walls and the (correctly implemented) thermal walls were. This suggests, at least from a structural point of view, that thermal walls might be a useful method if a heat reservoir is needed.

The section on thermal walls shows that details about how the thermal walls are implemented affects the temperature profiles. Using incorrect velocity distributions might stop the system from reaching equilibrium alto-

gether, and many of the thermodynamic and statistical mechanic definitions and results become invalid. Articles by Risso & Cordero (1997) and Du et al. (1995) are examples where incorrect distributions have been published, and Tehver et al. (1998) mention others.

The reason for the non-uniformity in the profiles of the average kinetic energy for these incorrect thermal walls, is that the thermal walls influence the velocity distribution in the tube. One can also say that the phenomenon is strongly connected to how the particles are restricted in phase space, and the thermal walls with the incorrect distributions lower the entropy close to the walls. The mean free path is probably an important parameter when describing the range of the non-uniformity in the profiles.

Having developed a molecular dynamics simulation program, it can be used to investigate a huge range of physical phenomena; heat flow and viscosity being two simple examples. If the results are to be compared with physical experiments, the model must be extended to three dimensions. In addition, a simulation of molecules rather than simple atoms would require implementation of the particle orientation, and perhaps asymmetric potential fields for particle interactions.

Molecular dynamics simulation is time consuming, and this is probably its worst drawback. A typical simulation describes a physical system lying in the nanosecond and nanometer range. Using a data structure that sections the simulation area into boxes, the speed is increased considerably. To further enhance the efficiency, one could for example use multiple time-steps (Streett, Tildesley & Saville, 1978), determining the length of the time-step from local conditions. This would especially work in simulations with inhomogeneities.

In order to really speed up simulations, one could combine molecular dynamics with simulations based on hydrodynamic equations, such as the Navier-Stokes equation, that describe the fluid macroscopically. Such simulations are often referred to as hybrid models (O'Connell & Thompson, 1995). Macroscopic simulations usually simulate bulk properties well, and sizes can be many orders of magnitude larger. One drawback is that boundary conditions need to be known *a priori*, and if chosen incorrectly, the simulations may fail. The hybrid method uses molecular dynamics to simulate the boundaries microscopically.

The most difficult part of a hybrid simulation is the interface between the molecular dynamics and the continuum part. This interface would need to transfer mass and momentum between the two parts. The thermal walls described in this thesis form a solid basis for understanding the momentum transfer mechanism.

Bibliography

Allen, M. P. & Tildesley, D. J. (1991), *Computer Simulation of Liquids*, Clarendon Press, Oxford.

Barker, J. A., Henderson, D. & Abraham, F. F. (1981), 'Phase diagram of the two-dimensional Lennard-Jones system; evidence for first-order transitions', *Physica* **106A**, 226–238.

Beeman, D. (1976), 'Some multistep methods for use in molecular dynamics calculations', *J. Comp. Phys.* **20**, 130–139.

Chandler, D., Weeks, J. D. & Andersen, H. C. (1983), 'Van der Waals picture of liquids, solids, and phase transformations', *Science* **220**, 787–794.

Du, Y., Li, H. & Kadanoff, L. P. (1995), 'Breakdown of hydrodynamics in a one-dimensional system of inelastic particles', *Phys. Rev. Lett.* **74**(8), 1268–1271.

Flyvbjerg, H. & Petersen, H. G. (1989), 'Error estimates on averages of correlated data', *J. Chem. Phys.* **91**(1), 461–466.

Frenkel, D. & Smit, B. (1996), *Understanding Molecular Simulation: From Algorithms to Applications*, Academic Press.

Gear, C. W. (1966), *The Numerical Integration of Ordinary Differential Equations of Various Orders*, ANL-7126, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439.

Götzelmann, B., Haase, A. & Dietrich, S. (1996), 'Structure factor of hard spheres near a wall', *Phys. Rev. E* **53**(4), 3456–3467.

Haile, J. M. (1997), *Molecular Dynamics Simulation: Elementary Methods*, John Wiley & Sons, Inc., New York.

Hansen, J. P. & McDonald, I. R. (1976), *Theory of Simple Liquids*, Academic Press Inc. (London) Ltd., 24/28 Oval Road, London NW1.

Jones, J. E. (1924*a*), 'On the determination of molecular fields.—I. From the variation of the viscosity of a gas with temperature.', *Proc. Roy. Soc. Lond.* **106A**, 441–462.

Jones, J. E. (1924*b*), 'On the determination of molecular fields.—II. From the equation of state of a gas.', *Proc. Roy. Soc. Lond.* **106A**, 463–477.

Landau, L. D. & Lifshitz, E. M. (1989), *Statistical Physics*, Vol. 5 of *Course of Theoretical Physics*, 3rd edn, Pergamon Press, Oxford.

Larsen, R. J. & Marx, M. L. (1986), *An Introduction to Mathematical Statistics and Its Applications*, 2nd edn, Prentice-Hall, Englewood Cliffs, New Jersey 07632.

London, F. (1930), 'Über einige Eigenschaften und Anwendungen der Molekularkräfte.', *Zeit. Physik. Chem. B* **11**, 222–251.

Maitland, G. C., Rigby, M., Smith, E. B. & Wakeham, W. A. (1981), *Intermolecular Forces: Their Origin and Determination*, Clarendon Press, Oxford.

McQuarrie, D. A. (1976), *Statistical Mechanics*, Harper & Row, New York.

Nicolas, J. J., Gubbins, K. E., Streett, W. B. & Tildesley, D. J. (1979), 'Equation of state for the Lennard-Jones Fluid', *Molecular Physics* **37**(5), 1429–1454.

O'Connell, S. T. & Thompson, P. A. (1995), 'Molecular dynamics — continuum hybrid computations: A tool for studying complex fluid flows', *Phys. Rev. E* **52**(6), R5792–R5795.

Potter, D. (1973), *Computational Physics*, John Wiley & Sons, London.

Risso, D. & Cordero, P. (1997), 'Dilute gas Couette flow: Theory and molecular dynamics simulation', *Phys. Rev. E* **56**(1), 489–498.

Silicon Graphics, Inc. (1998), 'Silicon Graphics — O2 workstation home page', available from: *http://www.sgi.com/o2/* [Accessed 3 Nov 1998].

Squires, G. L. (1994), *Practical Physics*, 3rd edn, Cambridge University Press.

Streett, W. B., Tildesley, D. J. & Saville, G. (1978), 'Multiple time step methods in molecular dynamics', *Mol. Phys.* **35**(3), 639–648.

Stroustrup, B. (1997), *The C++ Programming Language*, 3rd edn, Addison-Wesley, Reading, Massachusetts.

Sullivan, D. E. & Stell, G. (1978), 'Structure of a simple fluid near a wall. I. Structure near a hard wall.', *J. Chem. Phys.* **69**(12), 5450–5457.

Sullivan, D. E., Levesque, D. & Weis, J. J. (1980), 'Structure of a simple fluid near a wall. II. Comparison with Monte Carlo', *J. Chem. Phys.* **72**(2), 1170–1174.

Swope, W. C., Andersen, H. C., Berens, P. H. & Wilson, K. R. (1982), 'A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters', *J. Chem. Phys.* **76**(1), 637–649.

Tehver, R., Toigo, F., Koplik, J. & Banavar, J. R. (1998), 'Thermal walls in computer simulations', *Phys. Rev. E* **57**(1), R17–R20.

Tenenbaum, A., Ciccotti, G. & Gallico, R. (1982), 'Stationary nonequilibrium states by molecular dynamics. Fourier's law.', *Phys. Rev. A* **25**(5), 2778–2787.

Verlet, L. (1967), 'Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules', *Phys. Rev.* **159**(1), 98–103.

Weeks, J. D., Katsov, K. & Vollmayr, K. (1998), 'roles of repulsive and attractive forces in determining the structure of nonuniform liquids: Generalized mean field theory', Unpublished. At the time of submission of this thesis, APS has tentatively scheduled this article for 9 November 1998, *Phys. Rev. Lett.* **81** (19).

Weeks, J. D., Selinger, R. L. B. & Broughton, J. Q. (1995), 'Self-consistent treatment of repulsive and attractive forces in nonuniform liquids', *Phys. Rev. Lett.* **75**(14), 2694–2697.

Weeks, J. D., Vollmayr, K. & Katsov, K. (1997), 'Intermolecular forces and the structure of uniform and nonuniform fluids', *Physica A* **244**, 461–475.

Widom, B. (1967), 'Intermolecular forces and the nature of the liquid state', *Science* **157**(3787), 375–382.

Appendix A

Notation

In this list, the symbol A represents a general function, variable or value.

$\langle A \rangle$	Theoretical mean of $A()$. Used sometimes for empirical averages when the $\bar{A}$ notation is impractical.
$\bar{A}$	Empirical average of the values A_i.
$*, A^*$	The $*$ signifies that A is in reduced units.
α	Defined in Equation 2.11, $\alpha^2 = 2k_BT/m$.
$\delta()$	The delta function. Defined as $\int_{-\infty}^{\infty} A(t)\delta(x-t)\,dt = A(x)$ for a continuous function $A(t)$.
ϵ	Unit of energy, defined by the Lennard-Jones (12,6) potential. See Table 3.1.
λ	Mean free path.
ρ	Number density.
ρ_B	Number density in the bulk.
$\rho_R(\mathbf{r})$	Number density in reference fluid.
$\rho, \rho(p,q)$	Phase space distribution function.
$\rho(X,Y)$	Correlation coefficient between the stochastic variables X and Y.
σ	Unit of length, defined by the Lennard-Jones (12,6) potential. See Table 3.1.
$\phi(\mathbf{r})$	External field.

$\phi_R(\mathbf{r})$	Effective reference field (ERF).
θ, Θ	Angle relative to wall with which a particle crosses a line. See Figure 2.4.
$\mathbf{a}(t)$	Particle acceleration at time t.
$\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$	Thermal wall types. See Table 3.2.
B	Number of bins in a profile or distribution.
C, C_K, C_U	Normalization constants.
$C(t)$	Time correlation function or time autocorrelation function.
d	Particle diameter of a hard sphere.
$E, E(p, q)$	Total energy.
$E(A)$	Expected value of a stochastic variable A.
f	Translational degrees of freedom.
ΔF	The value with which the force is shifted in $\mathbf{F}_s(\mathbf{r})$, the shifted-force Lennard-Jones (12,6) potential.
$\mathbf{F}(t)$	Force acting on a particle at time t.
$\mathbf{F}(\mathbf{r})$	Interparticle Lennard-Jones (12,6) force. $\mathbf{F}(\mathbf{r}) = -\nabla u(\mathbf{r})$.
$\mathbf{F}_s(\mathbf{r})$	The shifted-force Lennard-Jones (12,6) potential.
$f_A(a)$	Probability density function (PDF) for a random variable A.
$F_A(a)$	Cumulative distribution function (CDF) for a random variable A. $F_A(a) = P(A \leq a)$.
$\tilde{f}_\Theta(\theta), \tilde{F}_\Theta(\theta)$	PDF and CDF of angles for particles crossing a vertical line.
$f_V(v), F_V(v)$	Maxwellian PDF and CDF.
$\tilde{f}_V(v), \tilde{F}_V(v)$	PDF and CDF for speed of (equilibrium) particles crossing a vertical line.
$f_{V_x}(v_x), F_{V_x}(v_x)$	PDF (Gaussian) and CDF for the horizontal velocity component.
$\tilde{f}_{V_x}(v_x), \tilde{F}_{V_x}(v_x)$	PDF and CDF for the horizontal velocity component of particles crossing a vertical line.

$f_{V_y}(v_y), F_{V_y}(v_y)$	PDF (Gaussian) and CDF for the vertical velocity component (bulk and crossing a vertical line).
$g()$	Discrete speed distribution.
$g(r)$	Radial distribution function.
$H(t)$	Boltzmann's H-function.
$H_x(t)$	Boltzmann's H-function for the horizontal velocities.
$H_y(t)$	Boltzmann's H-function for the vertical velocities.
k_B	Boltzmann's constant, $k_B = 1.3806\ldots$.
$K(p)$	Total kinetic energy as a function of momenta p.
K_i	Total kinetic energy at timestep i.
m	Particle mass.
M	Number of time-steps.
N	Number of particles.
p	Momenta of all the particles in the gas.
p_i	Momentum component of a particle, $i \in [1, s = 2N]$.
q	Positions of all the particles in the gas. Same as $\mathbf{r}^N$.
q_i	One of the position-coordinates for a particle.
r_c	Cutoff length for the truncated and shifted-force Lennard-Jones (12,6) potential. In this thesis, $r_c = 2.5\sigma$.
$\mathbf{r}_i$	Position of particle i.
$\mathbf{r}^N$	Positions of all the particles in the gas. Same as q.
$\mathbf{r}(t)$	Particle position at time t.
$R, \tilde{R}$	Areas of integration.
s	Number of spatial dimensions multiplied with the number of particles. In this thesis, $s = 2N$.
$S, \Delta S$	Entropy, increase in entropy.
S	Unbiased estimator for the standard deviation.
$S_{\bar{K}}$	Standard error of the mean $\bar{K}$.
S_{bin}	Standard error in profile bin values.
t	Time, or time from the start of a simulation.

t_k	Time at time-step k.
δt	Time-step length in a simulation.
Δt	Time interval, sometimes describing the length of a simulation run.
T	Absolute temperature.
T_x	The average kinetic energy calculated from the horizontal velocity components.
T_y	The average kinetic energy calculated from the vertical velocity components.
$u(r), u_{LJ}(r)$	Lennard-Jones potential (12,6).
$u_s(r)$	Shifted-force Lennard Jones (12,6) potential.
$u_R(r)$	Particle interaction potential function for reference fluid.
$u_d(r)$	Particle interaction potential function for hard spheres of diameter d.
u_i	Mean speed in the ith bin of a discrete speed distribution.
$U(q)$	Total potential energy as a function of positions q.
$v(F)$	The CDF inverse function, $F_V^{-1}(v)$.
$\mathbf{v}(t)$	Particle velocity at time t.
v, V	Speed (magnitude of velocity) of a particle.
v_i, V_i	Velocity component of a particle, $i \in [1, s = 2N]$.
v_x, V_x	Horizontal velocity component.
v_y, V_y	Vertical velocity component.
v_{RMS}	Root mean square velocity.
$Var(A)$	Variance of a stochastic variable A.
x	Horizontal component or direction.
y	Vertical component or direction.

Appendix B

The Program Listings

B.1 The Main Program

A list of the modules in the main program is presented, each with a brief description. This is followed by the complete listings of the modules.

MD_main.hh is the headerfile of MD_main.cc.

MD_main.cc contains main(), and calls the calculating and updating procedures.

MD_definitions.hh contains the definitions and inline (macro) functions, especially the definitions needed for the preprocessor.

MD_constants.hh contains all the global constants. This includes the reduced units.

MD_classes.hh contains the data structure and the distributions.

MD_classfuncs.cc contains functions for updating, normalizing and writing distributions to file. In addition, the constructors for classes defined in MD_classes.hh are located here.

MD_init.hh is the headerfile of MD_init.cc.

MD_init.cc sets up the initial structure, including particles, walls and the linked lists.

MD_calc.hh is the headerfile of MD_calc.cc.

MD_calc.cc calls the force calculation algorithms. It also contains the integration algorithms.

MD_calc_inline.hh contains the innermost force-calculating loop, both between particles, and between particles and walls. These are written inline to improve computer speed.

MD_fileoutput.hh is headerfile of MD_fileoutput.cc.

MD_fileoutput.cc contains most of the routines that save information on disk, except for the distributions.

MD_listtools.hh is the headerfile of MD_listtools.cc.

MD_listtools.cc contains routines for handeling the pointer lists.

B.1.1 MD_main.hh

```
/* MD_main.hh, header file for MD_main.cc */

#ifndef MAIN
#define MAIN

void mainloop(class particle *, class box *, class parameters *);

#endif
```

B.1.2 MD_main.cc

```
/* Molecular Dynamics Simulation Program. */
/* Main Module: MD_main.cc */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "MD_definitions.hh"
#include "MD_constants.hh"
#include "MD_classes.hh"
#include "MD_init.hh"
#include "MD_calc.hh"
#include "MD_fileoutput.hh"
#include "MD_main.hh"


void main()
{
  printf("Program starts...\n");

  long ttt;
  srand48(time(&ttt));

  class box *boxes = new box[no_of_boxes];
  class particle *prt = new particle[no_of_particles];
  class parameters *param = new parameters;

  if ((prt!=NULL) && (boxes!=NULL) && (param!=NULL)){

    int init_OK;

    printf("Initialization starts...\n");
    init_OK=init_variables(prt,boxes,param);
    printf("Initialization finished...\n");

    calculate_forces(prt,boxes,param);

    // Output the simulation constants,
    // i.e. the settings of the run.
    char s[256];
    sprintf(s,"%s%s",DATA_PATH,"constants.dat");
    write_constants_to_file(param,s);

    // Here one can add any output to check that
    // the simulation has been correctly initialized.

#if (SIMULATION==TRUE)
    printf("Mainloop starts...\n");
    if (init_OK) {mainloop(prt,boxes,param);};
#endif

  } else {printf("Memory allocation error (prt and/or boxes) \n");};

  delete [] boxes;
  delete [] prt;
  delete param;
};


void mainloop(class particle prt[], class box boxes[], class parameters *param)
{
  FLOATTYPE elapsed_time=0;
  int abort_now; // Set to TRUE if a particle leaves the tube

#if (REC_HALFKINETIC)
  FILE *fp_leftkin;
  char s_leftkin[256];
  sprintf(s_leftkin,"%s%s",DATA_PATH,"kinetic_left.dat");
  FILE *fp_rightkin;
  char s_rightkin[256];
  sprintf(s_rightkin,"%s%s",DATA_PATH,"kinetic_right.dat");
#endif
```

```
#if (REC_KINETIC)
  FILE *fp_kin;
  char s_kin[256];
  sprintf(s_kin,"%s%s",DATA_PATH,"kinetic.dat");
#endif
#if (REC_POTENTIAL)
  FILE *fp_pot;
  char s_pot[256];
  sprintf(s_pot,"%s%s",DATA_PATH,"potential.dat");
#endif
#if (REC_TOTAL)
  FILE *fp_tot;
  char s_tot[256];
  sprintf(s_tot,"%s%s",DATA_PATH,"total_energy.dat");
#endif
#if (REC_H_FUNCTION)
  FILE *fp_hfunc;
  char s_hfunc[256];
  sprintf(s_hfunc,"%s%s",DATA_PATH,"h_function.dat");
#endif
#if (REC_POT_PART)
  FILE *fp_pp;
  char s_pp[256];
  sprintf(s_pp,"%s%s",DATA_PATH,"pot_part.dat");
#endif
#if (REC_POT_WALL && (WALL_TYPE==LJ))
  FILE *fp_pw;
  char s_pw[256];
  sprintf(s_pw,"%s%s",DATA_PATH,"pot_wall.dat");
#endif
#if (REC_PX)
  FILE *fp_px;
  char s_px[256];
  sprintf(s_px,"%s%s",DATA_PATH,"Px.dat");
#endif
#if (REC_PY)
  FILE *fp_py;
  char s_py[256];
  sprintf(s_py,"%s%s",DATA_PATH,"Py.dat");
#endif
#if (REC_KIN_CHANGE && STOCH_WALLS)
  FILE *fp_kc;
  char s_kc[256];
  sprintf(s_kc,"%s%s",DATA_PATH,"kinetic_change.dat");
#endif
#if (REC_FRAMES)
  FILE *fp_frames;
  char s_frames[256];
  sprintf(s_frames,"%s%s",DATA_PATH,"frames.dat");
#endif
#if (REC_CARTOON)
  FILE *fp_car;
  char s_car[256];
  sprintf(s_car,"%s%s",DATA_PATH,"cartoon.ps");
#endif


  if (REC_ANY
#if (REC_HALFKINETIC)
      && ((fp_leftkin = fopen(s_leftkin,"w")) != NULL)
      && ((fp_rightkin = fopen(s_rightkin,"w")) != NULL)
#endif
#if (REC_KINETIC)
      && ((fp_kin = fopen(s_kin,"w")) != NULL)
#endif
#if (REC_POTENTIAL)
      && ((fp_pot = fopen(s_pot,"w")) != NULL)
#endif
#if (REC_TOTAL)
      && ((fp_tot = fopen(s_tot,"w")) != NULL)
#endif
#if (REC_H_FUNCTION)
      && ((fp_hfunc = fopen(s_hfunc,"w")) != NULL)
#endif
#if (REC_POT_PART)
      && ((fp_pp = fopen(s_pp,"w")) != NULL)
#endif
#if (REC_POT_WALL && (WALL_TYPE==LJ))
      && ((fp_pw = fopen(s_pw,"w")) != NULL)
#endif
#if (REC_KIN_CHANGE && STOCH_WALLS)
      && ((fp_kc = fopen(s_kc,"w")) != NULL)
#endif
#if (REC_PX)
```

```
      && ((fp_px = fopen(s_px,"w")) != NULL)
#endif
#if (REC_PY)
      && ((fp_py = fopen(s_py,"w")) != NULL)
#endif
#if (REC_CARTOON)
      && ((fp_car = fopen(s_car,"w")) != NULL)
#endif
#if (REC_FRAMES)
      && ((fp_frames = fopen(s_frames,"w")) != NULL)
#endif
      ){

#if (REC_CARTOON)
    fprintf(fp_car,"%c!\n/c {1 0 360 arc fill} def\n",'%');
#endif
#if (REC_FRAMES)
    fprintf(fp_frames,"%d %d\n",no_of_particles,1+long((num_of_cycles-start_write_step-1)/data_write_step));
#endif

    for (long i=0;i<num_of_cycles;i++){

      // ***********************************************
      // * HERE COMES THE MAIN CALCULATION             *
      // ***********************************************
#if ((REC_KIN_CHANGE && STOCH_WALLS) || RADIAL_DIST)
      param->current_step=i;
#endif
      abort_now=calculate_next_step(prt, boxes, param);
#if (INIT_RESCALE_VEL)
      if ((!abort_now) && (param->rescale_counter==init_rescale_interval) && (i<init_rescale_stop)) {
        rescale_vel(prt, param, init_temperature);
        printf("Equilibration: rescaling velocity\n");
      };
#endif
#if (REG_RESCALE_VEL)
      if ((!abort_now) && (param->rescale_counter==reg_rescale_interval)) {
        rescale_vel(prt, param, init_temperature);
        printf("Correcting: rescaling velocity at i=%d\n",i);
      };
#endif
      // ***************************************************************************
      // * Here one can save info to file, print values on screen etc.             *
      // ***************************************************************************
      // *                                                                         *

      elapsed_time+=dt;

#if (REC_CRASH_VALS)
      if (i>=start_crash_write) {param->update_crash_dists(prt);};
#endif
#if (REC_FRAMES)
      if (i%data_write_step==0){
        if (i>=start_write_step){write_frame_to_file(fp_frames,prt);};
        printf("Loop number %d.\n",i);
      };
#endif
#if (REC_CARTOON)
      if (i%cartoon_step==0) {
        if ((i>=cartoon_start) && (i<=cartoon_stop)) {write_cartoon_to_file(fp_car,prt);};
      };
#endif

#if (MAKE_DISTRIBUTIONS)
      if (i%dist_step==0){
        if (i>=dist_start){param->update_distributions(prt);};
      };
#endif

#if (RADIAL_DIST)
      if (i%rad_step==0){
        if ((i>=rad_start) && (i<=rad_stop)){param->update_rad_dist(prt);};
      };
#endif

#if (REC_H_FUNCTION)
      if (i%hfunc_step==0){
        if ((i>=hfunc_start) && (i<hfunc_stop)){
          param->update_hfunctions(prt);
          fprintf(fp_hfunc,"%.16g %.16g %.16g %.16g\n",elapsed_time,param->hm,param->hx,param->hy);
        };
      };
#endif
```

```
#if (REC_ENERGY)
      param->E_kinetic_cg+=param->E_kinetic;
      param->E_potential_cg+=param->E_potential;

      if (i%energy_step==0) {
        param->E_kinetic_cg/=(FLOATTYPE)energy_step;
        param->E_potential_cg/=(FLOATTYPE)energy_step;
#if (REC_TOTAL)
        fprintf(fp_tot,"%.16g %.16g\n",elapsed_time,param->E_potential_cg+param->E_kinetic_cg);
#endif
#if (REC_HALFKINETIC)
        fprintf(fp_leftkin,"%.16g %.16g\n",elapsed_time,param->E_kinetic_left);
        fprintf(fp_rightkin,"%.16g %.16g\n",elapsed_time,param->E_kinetic_right);
#endif
#if (REC_KINETIC)
        fprintf(fp_kin,"%.16g %.16g\n",elapsed_time,param->E_kinetic_cg);
#endif
#if (REC_POTENTIAL)
        fprintf(fp_pot,"%.16g %.16g\n",elapsed_time,param->E_potential_cg);
#endif
#if (REC_POT_PART)
        fprintf(fp_pp,"%.16g %.16g\n",elapsed_time,param->E_pot_part);
#endif
#if (REC_POT_WALL && (WALL_TYPE==LJ))
        fprintf(fp_pw,"%.16g %.16g\n",elapsed_time,param->E_pot_wall);
#endif
#if (REC_KIN_CHANGE && STOCH_WALLS)
        fprintf(fp_kc,"%.16g %.16g\n",elapsed_time,param->E_kin_change);
#endif
        param->E_kinetic_cg=0;
        param->E_potential_cg=0;
      };
#endif
#if (REC_MOMENTUM)
        param->Px_avg+=param->Px;
        param->Py_avg+=param->Py;
      if (i%momentum_step==0){
        param->Px_avg/=FLOATTYPE(momentum_step);
        param->Py_avg/=FLOATTYPE(momentum_step);
#if (REC_PX)
        fprintf(fp_px,"%.16g %.16g\n",elapsed_time,param->Px_avg);
#endif
#if (REC_PY)
        fprintf(fp_py,"%.16g %.16g\n",elapsed_time,param->Py_avg);
#endif
        param->Px_avg=0;
        param->Py_avg=0;
      };
#endif

      // *                                                                    *
      // **********************************************************************

      if (abort_now){
        printf("i = %d\n",i);
        break;
      };
    };
  } else {printf("\nUnable to open dat files.\n");};

#if (REC_HALFKINETIC)
  fclose(fp_leftkin);
  fclose(fp_rightkin);
#endif
#if (REC_KINETIC)
  fclose(fp_kin);
#endif
#if (REC_POTENTIAL)
  fclose(fp_pot);
#endif
#if (REC_TOTAL)
  fclose(fp_tot);
#endif
#if (REC_H_FUNCTION)
  fclose(fp_hfunc);
#endif
#if (REC_CARTOON)
  fclose(fp_car);
#endif
#if (REC_FRAMES)
  fclose(fp_frames);
#endif
#if (REC_POT_PART)
  fclose(fp_pp);
```

```
#endif
#if (REC_POT_WALL && (WALL_TYPE==LJ))
  fclose(fp_pw);
#endif
#if (REC_KIN_CHANGE && STOCH_WALLS)
  fclose(fp_kc);
#endif
#if (REC_PX)
  fclose(fp_px);
#endif
#if (REC_PY)
  fclose(fp_py);
#endif
#if (MAKE_DISTRIBUTIONS)
  param->norm_dists();
  param->write_dists_to_file("dist_report.log","v_dist.dat",
                             "Tx_profile.dat","Ty_profile.dat",
                             "T_profile.dat","particle_profile.dat");
#endif
#if (RADIAL_DIST)
  param->norm_rad_dist();
  param->write_rad_dist_to_file("rad_dist.dat");
#endif
#if (MAKE_DISTRIBUTIONS)
  param->norm_crash_dists();
  param->write_crash_dists();
#endif
};
```

B.1.3 MD_definitions.hh

During the simulations described in Chapter 7, only `WALL_TYPE` was changed. The Lennard-Jones walls and wall types $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ are represented in the program by `LJ`, `SDD`, `SCX`, `STC`, `SCC`, respectively. The other wall types were used during the development. The definitions starting with `REC_` were used to turn on and off different output.

```
/* MD_definitions.hh, header file containing definitions */

#ifndef DEFINITIONS
#define DEFINITIONS

#define FALSE 0
#define TRUE 1

#define FLOATTYPE double

// If SIMULATION is FALSE, only initialization and constant checking is done.
#define SIMULATION TRUE

// What kind of shift in LJ potential
#define NONE 0
#define POT 1
#define FORCE 2
#define SHIFTED FORCE

// What to record during simulation
#define DATA_PATH "./"
#define REC_KINETIC TRUE
#define REC_POTENTIAL FALSE
#define REC_TOTAL TRUE
// Record the kinetic energy on the left and right sides independently:
#define REC_HALFKINETIC FALSE
#define REC_H_FUNCTION TRUE
#define REC_POT_PART FALSE
#define REC_KIN_CHANGE FALSE // Only effective if STOCH_WALLS==TRUE
#define REC_POT_WALL FALSE // Only effective if WALL_TYPE==LJ
#define REC_PX FALSE
#define REC_PY FALSE
#define REC_CARTOON FALSE
#define REC_CRASH_VALS TRUE
// REC_FRAMES is now obsolete (replaced by MAKE_DISTRIBUTIONS)
#define REC_FRAMES FALSE
// See end for REC_ANY and REC_ENERGY

#define MAKE_DISTRIBUTIONS TRUE
#define RADIAL_DIST FALSE

// Tube structure
#define H_WALLS FALSE // Horizontal walls
#define V_WALLS TRUE // Vertical walls
```

```
// Walltypes:
// (The way the walls are defined, one can only have one type of wall at any one time.)
#define LJ 1 // Lennard-Jones
#define HR 2 // Hard reflecting
#define SCC 3 // Stochastic, particles are reemitted normal to wall at constant speed
#define SUC 4 // Stochastic, angle chosen from uniform distribution, constant speed
#define SUD 5 // Stochastic, angle chosen from uniform distribution, speed from P(v)~v^2*exp(-v^2/alpha^2)
#define SDD 6 // Stochastic, velocity normal to wall sampled from P(v)~v*exp(-v^2/alpha^2),
              // parallel to wall P(v)~exp(-v^2/alpha^2)
#define STC 7 // Stochastic, angle chosen from cos(theta)/2, v constant.
#define SCX 8 // Stochastic, constant angle (normal to wall), use vx distribution.
#define WALL_TYPE STC

// If FORCE_OUT is TRUE, then a particle cannot stay behing grabdist for more than maximum one timestep.
// If it doesn't get out on its own account, it is moved just in front of grabdist.
#define FORCE_OUT TRUE

// Rescale the velocities initially if the walls don't have this ability.
// (NOTE: Unless testing, this should always be TRUE)
#define INIT_RESCALE_VEL TRUE

// Rescale the velocities at regular intervals.
// Use TRUE and FALSE to turn it on and off for LJ walls
#define REG_RESCALE_VEL (FALSE && (WALL_TYPE==LJ))

// UNI_THETA is TRUE if the angle is chosen from a uniform distribution
#define UNI_THETA ((WALL_TYPE==SUC) || (WALL_TYPE==SUD))

// STOCH_WALL is TRUE if the walls are stochastic.
#define STOCH_WALLS ((WALL_TYPE==SCC) || (UNI_THETA) || (WALL_TYPE==SDD) || (WALL_TYPE==STC) || (WALL_TYPE==SCX))

// A TRIGGER_WALL is a wall that is 'activated' when a particle comes closer than
// a certain distance (e.g. grab_dist).
// A TRIGGER_WALL uses the touch_wall variables.
#define TRIGGER_WALLS ((WALL_TYPE==HR) || (STOCH_WALLS))

// ZONES are used to check when particles cross lines.
// If one wants to FORCE the program to use zones, change the first condition to TRUE.
// When the first condition is FALSE, the zones are turned on if there is
// either TRIGGER_WALLS or REC_CRASH_WALLS, given that the corresponding walls are turned on.
// If the distributions of particles that change zone should be saved, REC_CRASH_VALS MUST be TRUE.
// In addition, particles going in the X-direction are saved if X_ZONES is TRUE, and equiv. with Y_ZONES.
// The reason why X_ZONES and Y_ZONES can be TRUE even if REC_CRASH_VALS is FALSE, is because the TRIGGER_WALLS
// need the zones to function.
#define X_ZONES (FALSE || (V_WALLS && (TRIGGER_WALLS || REC_CRASH_VALS)))
#define Y_ZONES (FALSE || (H_WALLS && (TRIGGER_WALLS || REC_CRASH_VALS)))
#define ZONES (X_ZONES || Y_ZONES)

// Set TRUE for external constant force (e.g. gravity)
#define CONSTANT_FORCE FALSE

// REC_ENERGY is TRUE if any energy is recorded
#define REC_ENERGY (REC_HALFKINETIC || REC_KINETIC || REC_POTENTIAL || REC_TOTAL || REC_POT_PART
|| (REC_KIN_CHANGE && STOCH_WALLS) || (REC_POT_WALL && (WALL_TYPE==LJ)))

#define REC_MOMENTUM (REC_PX || REC_PY)

// REC_ANY is TRUE if there is any recording going on (which means it should be true every time!)
#define REC_ANY (REC_ENERGY || REC_MOMENTUM || REC_H_FUNCTION || REC_CARTOON || REC_CRASH_VALS || REC_FRAMES)

inline double min(double a, double b)
{
  return (a<b ? a:b);
}

inline double max(double a, double b)
{
  return (a>b ? a:b);
}

#endif
```

B.1.4 MD_constants.hh

```
/* MD_constants.hh, header file containing constants */

#ifndef CONSTANTS
#define CONSTANTS

#include <math.h>
#include "MD_definitions.hh"

// First constants concerning the execution of the program
```

```
// For how many cycles the program should run.
const long num_of_cycles=10100001;
#if (INIT_RESCALE_VEL)
// How often the velocities should be rescale in the equilibration period
const long init_rescale_interval=1000;
// How many timesteps the velocities should be rescaled initially.
const long init_rescale_stop=9*init_rescale_interval;
#endif
#if (REG_RESCALE_VEL)
// How often the velocities should be rescaled
const long reg_rescale_interval=5000;
#endif
#if (MAKE_DISTRIBUTIONS)
const long dist_start=100000;
const long dist_step=1; // Should always be 1
#endif
#if (RADIAL_DIST)
const long rad_start=100000;
const long rad_stop=10000000;
const long rad_step=1000;
#endif
#if (REC_FRAMES)
// OBSOLETE
// How many cycles between each time all particles are written to disk.
const long data_write_step=50;
// At which cycle start to make prt_files.
const long start_write_step=10000;
#endif
#if (REC_CRASH_VALS)
// When to begin to write crash values
const long start_crash_write=100000;
#endif
#if (REC_CARTOON)
// Which cycle to start recording cartoon (i.e. ps-file).
const long cartoon_start=100000;
// Which cycle to stop recording cartoon.
const long cartoon_stop=102000;
// How often a frame is saved (1=every frame, 2=every second etc.)
const long cartoon_step=5;
#endif
#if (REC_ENERGY)
// How often the energy is recorded
const long energy_step=50;
#endif
const long momentum_step=100;
#if (REC_H_FUNCTION)
// How often the H-function is calculated
const long hfunc_start=0;
const long hfunc_step=1;
const long hfunc_stop=100000;
#endif

// constants and variables WITH an underscore at the end, are in SI units (i.e. sigma_ is in SI units)
// constants and variables WITHOUT an underscore, are in GENERAL untis (i.e. sigma is in  DU, TU etc. units)

// physical constants
const FLOATTYPE Pi = 3.14159265358;
const FLOATTYPE sixth = 1/(FLOATTYPE)6;
const FLOATTYPE k_ = 1.38066e-23;

// potential energy v=4*epsilon*((sigma/r)^12-(sigma/r)^6)
// force F(vector)=(24*epsilon/r^2)*(2*(sigma/r)^12-(sigma/r)^6)*(x,y,z)
// SI units
const FLOATTYPE sigma_ = 0.341e-9; /* for Argon */
const FLOATTYPE epsilon_ = 119.8*k_; /* for Argon */
const FLOATTYPE m_ = 6.6335e-26; /* for Argon */

// units
// These values are just to compare with SI units.
const FLOATTYPE DU = sigma_; // 1 DU = sigma_ m
const FLOATTYPE TU = sigma_*sqrt(m_/epsilon_); // time Unit
const FLOATTYPE MU = m_; // Mass Unit
const FLOATTYPE EU=epsilon_; // Energy Unit
const FLOATTYPE FU=EU/DU; // Force Unit
const FLOATTYPE VU=sqrt(epsilon_/m_); // Velocity Unit
const FLOATTYPE AU=VU/TU; // Acceleration Unit
const FLOATTYPE TmpU = epsilon_/k_; // Temperature Unit
const FLOATTYPE NDU = 1/(DU*DU); // Number Density Unit

/*
The following values are one due to use of reduced units.
const FLOATTYPE k = 1; // k_*TmpU/EU;
const FLOATTYPE m = 1; // m_ / MU;
```

```
const FLOATTYPE sigma = 1; // sigma_ / DU;
const FLOATTYPE epsilon = 1; // epsilon_ / EU;
*/

// Don't calculate forces/energies for particles farther apart than this distance
const FLOATTYPE r_cutoff=2.5;
const FLOATTYPE r2_cutoff=pow(r_cutoff,2);

// The potential energy jump and force jump at the cutoff distance.
// Both values are negative.
const FLOATTYPE u_cutoff=4*(pow(r_cutoff,-12)-pow(r_cutoff,-6));
const FLOATTYPE f_cutoff=24*(2*pow(r_cutoff,-12)-pow(r_cutoff,-6))/r_cutoff;

const FLOATTYPE r_nopot = 1; // U = 0
const FLOATTYPE r_neutral = r_nopot*pow(2,sixth); // F = 0

// Since there is some areas close to the walls where the particles are not permitted,
// the actual number density will be somewhat higher than given here.
// See the density profile.
const FLOATTYPE number_density = 0.4;

// The timestep
const FLOATTYPE dt = 3e-3;
const FLOATTYPE dt2 = dt*dt;

// other constants

// In the directions where there are no walls,
// no_of_particles_pr_boxlength*no_of_x_boxes > 2*sqrt(number_density*r2_cutoff).
// This is because cutoff<tubelength_x/2
const long no_of_x_boxes = 4;
const long no_of_y_boxes = 4;
const long no_of_boxes = no_of_x_boxes * no_of_y_boxes;

const long no_of_particles_pr_boxlength = 3;
const long no_of_x_particles = no_of_x_boxes*no_of_particles_pr_boxlength;
const long no_of_y_particles = no_of_y_boxes*no_of_particles_pr_boxlength;
const long no_of_particles = no_of_x_particles* no_of_y_particles;
const FLOATTYPE tube_area = no_of_particles / number_density;

const FLOATTYPE boxlength=sqrt(tube_area/(FLOATTYPE)no_of_boxes);
const FLOATTYPE dist_between_part=boxlength/no_of_particles_pr_boxlength;

const FLOATTYPE tubelength_x=no_of_x_boxes*boxlength;
const FLOATTYPE tubelength_y=no_of_y_boxes*boxlength;

#if (H_WALLS)
// tubelength_y>2*horiwall_grabdist
const FLOATTYPE horiwall_grabdist = dist_between_part*0.5;
#endif
#if (V_WALLS)
// tubelength_x>2*vertwall_grabdist
const FLOATTYPE vertwall_grabdist = dist_between_part*0.5;
#endif

// Determines where the LJ-potential is placed on the walls,
// and what the initial temperature should be.
const FLOATTYPE init_temperature = 4;

#if ((WALL_TYPE==SUD) || (WALL_TYPE==SDD) || (WALL_TYPE==SCX))
const FLOATTYPE wall_temperature = init_temperature;
#endif
#if ((WALL_TYPE==SCC) || (WALL_TYPE==SUC) || (WALL_TYPE==STC))
// For a 2D ideal gas, v=sqrt(3kT/m)
const FLOATTYPE v0=sqrt(3*init_temperature);
#endif

// For initialisation, define the maximum velocity in either x or y direction.
const FLOATTYPE max_vx=sqrt(2*init_temperature);
const FLOATTYPE max_vy=max_vx;

#if (REC_H_FUNCTION)
#if (INIT_RESCALE_VEL)
const FLOATTYPE h_temperature = init_temperature;
#elif ((WALL_TYPE==SUD) || (WALL_TYPE==SDD))
const FLOATTYPE h_temperature = wall_temperature;
#elif
const FLOATTYPE h_temperature = 4;
#endif
#endif

#if (CONSTANT_FORCE==TRUE)
const FLOATTYPE constant_force_x=0;
const FLOATTYPE constant_force_y=-(1e17/AU)*m;
```

```
#endif

#if (X_ZONES)
const long num_x_zones=12;
#endif
#if (Y_ZONES)
const long num_y_zones=12;
#endif

// CONSTANTS for the TABLES
#if ((WALL_TYPE==SDD) || (WALL_TYPE==SUD) || (WALL_TYPE==SCX))
// number of bins in table for K_inv, Kv_inv and Kv2_inv.
const long nob=32768;
// The difference in velocity between two adjacent bins in K_inv, Kv_inv and Kv2_inv.
const FLOATTYPE dv=12/(FLOATTYPE)(nob*100); //First number should be approx. max speed
// K_factor, Kv_factor and Kv2_factor determines how large K_inv, Kv_inv and Kv2_inv grows, respectively.
// K_factor=1 makes K_sum=0.5 in the end.
// Kv_factor=1 makes K_sum=1 in the end.
// Kv2_factor=1 makes K_sum=1 in the end.
#if (WALL_TYPE==SDD)
const FLOATTYPE K_factor=0.99999;
const FLOATTYPE Kv_factor=0.99999;
#else
const FLOATTYPE Kv2_factor=0.99999;
#endif
#endif

#if (MAKE_DISTRIBUTIONS)
const long dist_bins=3600;
// if (in a 2D gas) we want a fraction eta to be included in the distribution,
// max_speed=sqrt(alpha^2*ln[1/(1-eta)])
const FLOATTYPE min_speed_v=0;
const FLOATTYPE max_speed_v=12;
const FLOATTYPE binwidth_v=(max_speed_v-min_speed_v)/FLOATTYPE(dist_bins);

const long prof_bins=3600;
const FLOATTYPE binwidth_prof=tubelength_x/FLOATTYPE(prof_bins);
#endif

#if (RADIAL_DIST)
// if rad_length is shorter than the diagonal of the tube,
// g[] might overflow, especially if there are walls on all sides.
const FLOATTYPE rad_length=sqrt(tubelength_x*tubelength_x+tubelength_y*tubelength_y);
const long rad_bins=3600;
const FLOATTYPE binwidth_rad=rad_length/(FLOATTYPE)rad_bins;
#endif

#if (REC_H_FUNCTION)
// If this fvbins is too small, there will be to few bins to see any change in distribution,
// only change in average. Both the theoretical and measured h-value will become too low.

// If fvbins is too large, there will be very few particles in each bin,
// thus there will not be any distribution. The theoretical curve will become
// very accurate, but the measured h-value will become too close to zero (not negative enough).

// 15 seems to work ok (perhaps a little higher (20?)) with 144 particle when
// instantaneous values of h are calculated.
const long fvbins=15;
#endif

#if (REC_CRASH_VALS)
const long num_of_dists=long(num_x_zones/2);
const long upper_zones=num_of_dists+1;

const long crash_bins=3600;

const FLOATTYPE min_crash_x=0;
const FLOATTYPE max_crash_x=tubelength_x;
const FLOATTYPE min_crash_v=0;
const FLOATTYPE max_crash_v=12;
const FLOATTYPE min_crash_v2=0;
const FLOATTYPE max_crash_v2=144;
const FLOATTYPE min_crash_theta=-1.0000001*Pi/2;
const FLOATTYPE max_crash_theta=1.0000001*Pi/2;
const FLOATTYPE min_crash_vx=0;
const FLOATTYPE max_crash_vx=12;
const FLOATTYPE min_crash_vy=-12;
const FLOATTYPE max_crash_vy=12;

const FLOATTYPE theta_dist_binwidth=(max_crash_theta-min_crash_theta)/(FLOATTYPE)crash_bins;
const FLOATTYPE v_dist_binwidth=(max_crash_v-min_crash_v)/(FLOATTYPE)crash_bins;
const FLOATTYPE v2_dist_binwidth=(max_crash_v2-min_crash_v2)/(FLOATTYPE)crash_bins;
const FLOATTYPE vx_dist_binwidth=(max_crash_vx-min_crash_vx)/(FLOATTYPE)crash_bins;
const FLOATTYPE vy_dist_binwidth=(max_crash_vy-min_crash_vy)/(FLOATTYPE)crash_bins;
```

```
#endif

#endif
```

B.1.5 MD_classes.hh

```
/* MD_classes.hh, file containg the classes */

#ifndef CLASSES
#define CLASSES

#include "MD_definitions.hh"
#include "MD_constants.hh"

class coord
{
public:
  double x,y;
};

class velocity
{
public:
  double vx,vy;
};

class acceleration
{
public:
  double ax,ay;
};

class boxlist
{
public:
  long boxnr;
  class boxlist *next_box;
};

class box
{
public:
  // A key; each box has a unique number, the same number as in the array.
  long boxnr;

  class coord bottomleft;
  class coord topright;

  // A pointer to the first particle in the box.
  class particle *first_particle;
  // A pointerlist of boxes with which this box should interact.
  class boxlist *first_box;
  box();
};

class particle
{
public:
  // A key; each particle has a unique number, the same number as in the array.
  long partnr;


  // The coordinate system has positive x-axis to the right,
  // and positive y-axis up.

  class coord r;        // current position
  class velocity ov;    // a halfstep behind v
  class velocity v;     // velocity
  class acceleration a; // acceleration

  // Which horizontal/vertical zone the particle occupies.
  long x_zone,y_zone,old_x_zone,old_y_zone;

#if (UNI_THETA && V_WALLS)
  long touch_vwall;
#endif
#if (UNI_THETA && H_WALLS)
  long touch_hwall;
#endif

  long boxnr;                   // number of box where particle is
  class particle *next_particle; // pointer to next particle in box, last particle points to NULL
  class particle *prev_particle; // pointer to previous particle in box, first particle points to NULL
```

```
  particle();
};

#if (H_WALLS)
class horizontal_wall
{
public:
  // theta is measured ANTICLOCKWISE from positive xaxis (if positive yaxis is up),
  // and theta is the angle of the vector NORMAL to the wall,
  // pointing INTO the box (2D).
  FLOATTYPE y,theta;

#if (WALL_TYPE==LJ)
  // adjust is used to move the potential of the wall
  FLOATTYPE adjust;
#endif

#if ((WALL_TYPE==SUD) || (WALL_TYPE==SDD))
  FLOATTYPE temperature;
#endif
};
#endif

#if (V_WALLS)
class vertical_wall
{
public:
  // See comment on theta, horizontal_wall.
  FLOATTYPE x,theta;

#if (WALL_TYPE==LJ)
  // adjust is used to move the potential of the wall
  FLOATTYPE adjust;
#endif
};
#endif

// This class defines what is to be sent from main to calc
class parameters
{
public:
#if (REC_HALFKINETIC)
  FLOATTYPE E_kinetic_left,E_kinetic_right;
#endif
  FLOATTYPE E_kinetic,E_potential;
  FLOATTYPE E_kinetic_cg,E_potential_cg; // Coarse grained values.
  FLOATTYPE E_pot_part; // Potential energy between particles.
#if (WALL_TYPE==LJ)
  FLOATTYPE E_pot_wall; // Potential energy between particles and LJ-walls.
#endif
#if (STOCH_WALLS && REC_KIN_CHANGE)
  FLOATTYPE E_kin_change; // Change in kinetic energy when hitting wall.
#endif
  FLOATTYPE Px,Py; // Average momentum in x and y directions.
#if (REC_MOMENTUM)
  FLOATTYPE Px_avg,Py_avg; // Averaged over several timesteps.
#endif
#if (INIT_RESCALE_VEL || REG_RESCALE_VEL)
  FLOATTYPE E_kin_rescale;
  long rescale_counter;
#endif

#if (H_WALLS)
  class horizontal_wall h_wall[2];
#endif
#if (V_WALLS)
  class vertical_wall v_wall[2];
#endif

  // x_zone (and y_zone) have the x (and y) coordinates of the line/border to the right (or above) the zone
#if (X_ZONES)
  FLOATTYPE x_zone[num_x_zones];
#endif
#if (Y_ZONES)
  FLOATTYPE y_zone[num_y_zones];
#endif

#if (UNI_THETA)
#if (H_WALLS)
  long touch_hwalls;
#endif
#if (V_WALLS)
  long touch_vwalls;
```

```
#endif
#endif

#if (WALL_TYPE==SDD)
  // P(v) is the distribution of the magnitudes of the velocities
  // (the speed v) of the particles that hit the wall.
  // K(v) is the cumulative distribution, K(v)=int(P(u),u=0..v)
  // v(K)=K_inv(K) is the inverse function of K.
  // By picking values for K from a uniform distribution [0,1>, one gets
  // a random value for the speed picked from the distribtuion P(v)

  // The inverse of the cumulative distribution of speeds in one dimension in a gas.
  // Gives values of v in [-...,+...]
  // K_inv[0] shouldn't really be used, only K_inv[1]...K_inv[nob-1]
  FLOATTYPE K_inv[nob];
  // The inverse of the cumulative distribution of speeds in one dimension when particle comes from a wall.
  // Also the inverse of the cumulative distribution of speeds of a normal two dimensional gas.
  // Gives values of v in [0,+...]
  FLOATTYPE Kv_inv[nob];
#endif

#if ((WALL_TYPE==SUD) || (WALL_TYPE==SCX))
  // The inverse of the cumulative distribution of speeds in two dimensions when particle comes from a wall.
  // Gives values of v in [0,+...]
  FLOATTYPE Kv2_inv[nob];
#endif

#if (MAKE_DISTRIBUTIONS)
private:
  // The distributions
  FLOATTYPE vdist[dist_bins];
  FLOATTYPE v2dist[dist_bins];
  FLOATTYPE Tx_prof[prof_bins]; // A profile of vx^2/2
  FLOATTYPE Tx2_prof[prof_bins]; // Only needed to calculate st.err.for Tx_prof.
  FLOATTYPE Ty_prof[prof_bins]; // A profile of vy^2/2
  FLOATTYPE Ty2_prof[prof_bins]; // Only needed to calculate st.err.for Ty_prof.
  FLOATTYPE T_prof[prof_bins]; // A profile of v^2/2
  FLOATTYPE T2_prof[prof_bins]; // Only needed to calculate st.err.for T_prof.
  FLOATTYPE part_prof[prof_bins]; // Density Profile

  FLOATTYPE max_v;
  long speed_overflow;
  long num_vpart, part_in_part_prof;
public:
  void update_distributions(class particle *);
  void norm_dists();
  void write_dists_to_file(char *, char *, char *, char *, char *, char *);
#endif

#if (RADIAL_DIST)
  FLOATTYPE g[rad_bins];
  void update_rad_dist(class particle *);
  void norm_rad_dist();
  void write_rad_dist_to_file(char *);
#endif

#if (REC_H_FUNCTION)
public:
  FLOATTYPE fv_index[fvbins];
  FLOATTYPE fvx[fvbins];
  FLOATTYPE fvy[fvbins];
  FLOATTYPE fv_maxwell[fvbins];
public:
  FLOATTYPE hm,hx,hy;
  void init_fv();
  void update_hfunctions(class particle *);
#endif

#if ((REC_KIN_CHANGE && STOCH_WALLS) || RADIAL_DIST)
  long current_step;
#endif

#if (REC_CRASH_VALS)
  FLOATTYPE v_emit_dist[num_of_dists][crash_bins];
  FLOATTYPE v2_emit_dist[num_of_dists][crash_bins];
  FLOATTYPE theta_emit_dist[num_of_dists][crash_bins];
  FLOATTYPE vx_emit_dist[num_of_dists][crash_bins];
  FLOATTYPE vy_emit_dist[num_of_dists][crash_bins];

  FLOATTYPE v_crash_dist[num_of_dists][crash_bins];
  FLOATTYPE v2_crash_dist[num_of_dists][crash_bins];
  FLOATTYPE theta_crash_dist[num_of_dists][crash_bins];
  FLOATTYPE vx_crash_dist[num_of_dists][crash_bins];
  FLOATTYPE vy_crash_dist[num_of_dists][crash_bins];
```

```
  long part_in_v_emit_dist[num_of_dists];
  long part_in_v_crash_dist[num_of_dists];
  long part_in_v2_emit_dist[num_of_dists];
  long part_in_v2_crash_dist[num_of_dists];
  long part_in_theta_emit_dist[num_of_dists];
  long part_in_theta_crash_dist[num_of_dists];
  long part_in_vx_emit_dist[num_of_dists];
  long part_in_vx_crash_dist[num_of_dists];
  long part_in_vy_emit_dist[num_of_dists];
  long part_in_vy_crash_dist[num_of_dists];

  long part_not_in_v_emit_dist[num_of_dists];
  long part_not_in_v_crash_dist[num_of_dists];
  long part_not_in_v2_emit_dist[num_of_dists];
  long part_not_in_v2_crash_dist[num_of_dists];
  long part_not_in_theta_emit_dist[num_of_dists];
  long part_not_in_theta_crash_dist[num_of_dists];
  long part_not_in_vx_emit_dist[num_of_dists];
  long part_not_in_vx_crash_dist[num_of_dists];
  long part_not_in_vy_emit_dist[num_of_dists];
  long part_not_in_vy_crash_dist[num_of_dists];

  long total_particles;

  void update_crash_dists(class particle *);
  void norm_crash_dists();
  void write_crash_dists();
#endif

  parameters();
};

#endif
```

B.1.6 MD_classfuncs.cc

```
/* MD_classfuncs.cc, functions in the classes in MD_classes.hh */

#include <stdio.h>
#include "MD_classes.hh"

box::box()
{
  first_particle=NULL;
  first_box=NULL;
};

particle::particle()
{
  prev_particle=NULL;
  next_particle=NULL;
};

parameters::parameters()
{
#if (REC_HALFKINETIC)
  E_kinetic_left=0;
  E_kinetic_right=0;
#endif
  E_kinetic=0;
  E_potential=0;
  E_kinetic_cg=0;
  E_potential_cg=0;
  E_pot_part=0;
#if (WALL_TYPE==LJ)
  E_pot_wall=0;
#endif
#if (STOCH_WALLS && REC_KIN_CHANGE)
  E_kin_change=0;
#endif
  Px=0;
  Py=0;
#if (REC_MOMENTUM)
  Px_avg=0;
  Py_avg=0;
#endif
#if (INIT_RESCALE_VEL || REG_RESCALE_VEL)
  FLOATTYPE E_kin_rescale=0;
  long rescale_counter=0;
#endif

#if (MAKE_DISTRIBUTIONS)
```

```
  long i;

  max_v=min_speed_v;

  speed_overflow=0;

  num_vpart=0;
  part_in_part_prof=0;

  for (i=0;i<dist_bins;i++){
    vdist[i]=0;
  };

  for (i=0;i<prof_bins;i++){
    Tx_prof[i]=0;
    Tx2_prof[i]=0;
    Ty_prof[i]=0;
    Ty2_prof[i]=0;
    T_prof[i]=0;
    T2_prof[i]=0;
    part_prof[i]=0;
  };
#endif

#if (RADIAL_DIST)
  for (i=0;i<rad_bins;i++){
    g[i]=0;
  };
#endif

#if (REC_CRASH_VALS)
  total_particles=0;

  for (long j=0;j<num_of_dists;j++){
    part_in_v_emit_dist[j]=0;
    part_in_v_crash_dist[j]=0;
    part_in_v2_emit_dist[j]=0;
    part_in_v2_crash_dist[j]=0;
    part_in_theta_emit_dist[j]=0;
    part_in_theta_crash_dist[j]=0;
    part_in_vx_emit_dist[j]=0;
    part_in_vx_crash_dist[j]=0;
    part_in_vy_emit_dist[j]=0;
    part_in_vy_crash_dist[j]=0;

    part_not_in_v_emit_dist[j]=0;
    part_not_in_v_crash_dist[j]=0;
    part_not_in_v2_emit_dist[j]=0;
    part_not_in_v2_crash_dist[j]=0;
    part_not_in_theta_emit_dist[j]=0;
    part_not_in_theta_crash_dist[j]=0;
    part_not_in_vx_emit_dist[j]=0;
    part_not_in_vx_crash_dist[j]=0;
    part_not_in_vy_emit_dist[j]=0;
    part_not_in_vy_crash_dist[j]=0;

    for (i=0;i<crash_bins;i++){
      // velocity normal to wall
      // (vx>0 => particle away from wall (inwards or outwards))
      vx_emit_dist[j][i]=0;
      vx_crash_dist[j][i]=0;

      // velocity parallel to wall
      // (vy>0 => particle moving up if wall on left)
      vy_emit_dist[j][i]=0;
      vy_crash_dist[j][i]=0;

      v_emit_dist[j][i]=0;
      v_crash_dist[j][i]=0;

      theta_emit_dist[j][i]=0;
      theta_crash_dist[j][i]=0;

      v2_emit_dist[j][i]=0;
      v2_crash_dist[j][i]=0;
    };
  };
#endif
};

#if (MAKE_DISTRIBUTIONS)
void parameters::update_distributions(class particle prt[])
{
  FLOATTYPE x,y,vx,vy,Tx,Ty,Tx2,Ty2,speed,T,T2;
```

```
  for (long i=0;i<no_of_particles;i++){
    x=prt[i].r.x;
    y=prt[i].r.y;
    vx=prt[i].v.vx;
    vy=prt[i].v.vy;

    Tx=0.5*vx*vx;
    Ty=0.5*vy*vy;
    T=Tx+Ty;
    Tx2=Tx*Tx;
    Ty2=Ty*Ty;
    speed=sqrt(2*T);
    T2=T*T;

    if ((speed>=min_speed_v) && (speed<max_speed_v)){
      vdist[long((speed-min_speed_v)/binwidth_v)]++;
      num_vpart++;
    } else speed_overflow++;

    if ((x>=0) && (x<tubelength_x)){
      Tx_prof[long(x/binwidth_prof)]+=Tx;
      Tx2_prof[long(x/binwidth_prof)]+=Tx2;
      Ty_prof[long(x/binwidth_prof)]+=Ty;
      Ty2_prof[long(x/binwidth_prof)]+=Ty2;
      T_prof[long(x/binwidth_prof)]+=T;
      T2_prof[long(x/binwidth_prof)]+=T2;
      part_prof[long(x/binwidth_prof)]++;
      part_in_part_prof++;
    };
    if (max_v<speed) {max_v=speed;};
  };
};

void parameters::norm_dists()
{
  long i;

  // Normalize the distributions
  for (i=0;i<dist_bins;i++){
    vdist[i]/=(FLOATTYPE)num_vpart*binwidth_v;
  };

  // Averaging the profiles
  for (i=0;i<prof_bins;i++){
    /* part_prof has number of particles per section of the tube */
    if (part_prof[i]!=0){
      // For the standard deviations, see page 241 in Statistics Book.
      // The standard deviation is s^2=(1/(n-1))*sum((v(i)-v_avg)^2)
      // This can be rewritten as s^2=(n*sum(v(i)^2)-sum(v(i))^2)/(n*(n-1))
      // The standard error (or deviation) of the mean is sqrt(Var(V)/n) (but not really s/sqrt(n))

      FLOATTYPE n=part_prof[i];
      Tx_prof[i]/=n;
      Tx2_prof[i]/=n;
      Ty_prof[i]/=n;
      Ty2_prof[i]/=n;
      T_prof[i]/=n; // T_prof has mean (v^2)/2 per particle
      T2_prof[i]/=n; // T2_prof has mean (d^4)/2 per particle (second sample moment of v^2)
    };
  };
};

void parameters::write_dists_to_file(char filename_log[], char filename_v[],
                                     char filename_Txp[], char filename_Typ[],
                                     char filename_Tp[], char filename_p[])
{
  long i;
  char s[256];
  FILE *fp;

  // Write these first things to a separate file!
  sprintf(s,"%s%s",DATA_PATH,filename_log);
  if ((fp = fopen(s,"w")) != NULL){
    fprintf(fp,"Maximum velocity encountered: %.16g\n",max_v);
    fprintf(fp,"Number of particles with speed>max_speed: %d\n",speed_overflow);
    fprintf(fp,"Number of particles in part_prof: %d\n",part_in_part_prof);
    fprintf(fp,"Number of particles NOT in part_prof: %d\n",
            no_of_particles*(1+long((num_of_cycles-dist_start-1)/dist_step))-part_in_part_prof);
    fclose(fp);
    printf("File successfully written to disk: %s\n",s);
  } else {printf("Unable to open file: %s\n",s);};

  // Write distributions to file
  // Use avg_bins.cc to process these files.
```

```
  sprintf(s,"%s%s",DATA_PATH,filename_v);
  if ((fp = fopen(s,"w")) != NULL){
    for (i=0;i<dist_bins;i++){
      fprintf(fp,"%.16g %.16g\n",min_speed_v+(0.5+i)*binwidth_v,vdist[i]);
    };
    fclose(fp);
    printf("File successfully written to disk: %s\n",s);
  } else {printf("Unable to open file: %s\n",s);};

  // Output: x avg(vx^2/2) avg((vx^2/2)^2) no.part.
  sprintf(s,"%s%s",DATA_PATH,filename_Txp);
  if ((fp = fopen(s,"w")) != NULL){
    for (i=0;i<prof_bins;i++){
      fprintf(fp,"%.16g %.16g %.16g %.16g\n",(0.5+i)*binwidth_prof,Tx_prof[i],Tx2_prof[i],part_prof[i]);
    };
    fclose(fp);
    printf("File successfully written to disk: %s\n",s);
  } else {printf("Unable to open file: %s\n",s);};

  // Output: x avg(vy^2/2) avg((vy^2/2)^2) no.part.
  sprintf(s,"%s%s",DATA_PATH,filename_Typ);
  if ((fp = fopen(s,"w")) != NULL){
    for (i=0;i<prof_bins;i++){
      fprintf(fp,"%.16g %.16g %.16g %.16g\n",(0.5+i)*binwidth_prof,Ty_prof[i],Ty2_prof[i],part_prof[i]);
    };
    fclose(fp);
    printf("File successfully written to disk: %s\n",s);
  } else {printf("Unable to open file: %s\n",s);};

  // Output: x avg(v^2/2)(=first moment of v^2/2) avg((v^2/2)^2)(=second moment of v^2/2) no.part.
  sprintf(s,"%s%s",DATA_PATH,filename_Tp);
  if ((fp = fopen(s,"w")) != NULL){
    for (i=0;i<prof_bins;i++){
      fprintf(fp,"%.16g %.16g %.16g %.16g\n",(0.5+i)*binwidth_prof,T_prof[i],T2_prof[i],part_prof[i]);
    };
    fclose(fp);
    printf("File successfully written to disk: %s\n",s);
  } else {printf("Unable to open file: %s\n",s);};


  sprintf(s,"%s%s",DATA_PATH,filename_p);
  if ((fp = fopen(s,"w")) != NULL){
    for (i=0;i<prof_bins;i++){
      fprintf(fp,"%.16g %.16g\n",(0.5+i)*binwidth_prof,part_prof[i]);
    };
    fclose(fp);
    printf("File successfully written to disk: %s\n",s);
  } else {printf("Unable to open file: %s\n",s);};
};
#endif

#if (RADIAL_DIST)
void parameters::update_rad_dist(class particle prt[])
{
  long i,j;

  for (i=0;i<(no_of_particles-1);i++){
    for (j=i+1;j<no_of_particles;j++){

      FLOATTYPE dx,dy,r;

      dx=prt[i].r.x - prt[j].r.x;
#if (V_WALLS==FALSE)
      {
        FLOATTYPE temp;
        if (dx<0){if ((temp=tubelength_x+dx)<-dx) {dx=temp;};}
        else {if ((temp=tubelength_x-dx)<dx) {dx=-temp;};};
      };
#endif

      dy=prt[i].r.y - prt[j].r.y;
#if (H_WALLS==FALSE)
      {
        FLOATTYPE temp;
        if (dy<0) {if ((temp=tubelength_y+dy)<-dy) {dy=temp;};}
        else {if ((temp=tubelength_y-dy)<dy) {dy=-temp;};};
      };
#endif

      r=sqrt(dx*dx + dy*dy);
      // No test is performed to check that the index is in range.
      g[long(r/binwidth_rad)]++;
    };
  };
```

```
};

void parameters::norm_rad_dist()
{
  long i;

  // Averaging the profiles
  FLOATTYPE n;
  for (i=0;i<rad_bins;i++){
    n=(tubelength_x*tubelength_y*(FLOATTYPE)rad_step*(FLOATTYPE)2)/
      (2*Pi*((0.5+i)*binwidth_rad)*binwidth_rad*no_of_particles*(no_of_particles-1)*(1+(rad_stop-rad_start)));
    g[i]*=n;
  };
};

void parameters::write_rad_dist_to_file(char filename[])
{
  long i;
  char s[256];
  FILE *fp;

  sprintf(s,"%s%s",DATA_PATH,filename);
  if ((fp = fopen(s,"w")) != NULL){
    for (i=0;i<rad_bins;i++){
      fprintf(fp,"%.16g %.16g\n",(0.5+i)*binwidth_rad,g[i]);
    };
    fclose(fp);
    printf("File successfully written to disk: %s\n",s);
  } else {printf("Unable to open file: %s\n",s);};
};
#endif

#if (REC_H_FUNCTION)
void parameters::init_fv(){
  long ii;
  // Sample the distribution 2 st.devs. to either side of zero
  // alpha=sqrt(2T) is one st.dev
  const FLOATTYPE alpha=sqrt(2*h_temperature);
  hm=0;
  for (ii=0;ii<fvbins;ii++){
    fv_index[ii]=-2*alpha+4*alpha*(ii/FLOATTYPE(fvbins-1));
    fv_maxwell[ii]=exp(-fv_index[ii]*fv_index[ii]/(alpha*alpha))/(sqrt(Pi)*alpha);
    // fv_maxwell > 0 : if fv_maxwell=0, term gives 0, f wins over log(f)
  };
  FLOATTYPE dv=fv_index[1]-fv_index[0];
  for (ii=0;ii<fvbins;ii++){
    if (fv_maxwell[ii]>0) {
      hm+=fv_maxwell[ii]*log(fv_maxwell[ii])*dv;
    };
  };
};

void parameters::update_hfunctions(class particle prt[]){
  long i;
  FLOATTYPE range = fv_index[fvbins-1]-fv_index[0];
  FLOATTYPE dv=fv_index[1]-fv_index[0];

  for (i=0;i<fvbins;i++){
    fvx[i]=0;
    fvy[i]=0;
  };
  for (i=0;i<no_of_particles;i++){
    if ((prt[i].v.vx>fv_index[0]) && (prt[i].v.vx<fv_index[fvbins-1])){
      fvx[long(fvbins*(prt[i].v.vx-fv_index[0])/range)]++;
    };
    if ((prt[i].v.vy>fv_index[0]) && (prt[i].v.vy<fv_index[fvbins-1])){
      fvy[long(fvbins*(prt[i].v.vy-fv_index[0])/range)]++;
    };
  };
  hx=0; hy=0;
  for (i=0;i<fvbins;i++){
    fvx[i]/=dv*no_of_particles;
    fvy[i]/=dv*no_of_particles;
    if (fvx[i]>0) {
      hx+=fvx[i]*log(fvx[i])*dv;
    };
    if (fvy[i]>0) {
      hy+=fvy[i]*log(fvy[i])*dv;
    };
  };
};
#endif

#if (REC_CRASH_VALS)
```

```
void parameters::update_crash_dists(class particle prt[])
{
  long i;
  FLOATTYPE x,y,vx,vy,v,v2,theta;
  long oxz,xz,oyz,yz;
  long theta_index,v_index,v2_index,vx_index,vy_index;

  for (i=0;i<no_of_particles;i++){
#if ((X_ZONES) && (Y_ZONES==FALSE))
    if (prt[i].x_zone!=prt[i].old_x_zone){
      vx=prt[i].ov.vx;
      vy=prt[i].ov.vy;
      v2=vx*vx+vy*vy;
      v=sqrt(v2);
      x=prt[i].r.x;
      y=prt[i].r.y;
      xz=prt[i].x_zone;
      oxz=prt[i].old_x_zone;
      // By using ov and not v, one looks at the velocity that
      // was used to calculate the new r.

      if ((x>=0) && (x<tubelength_x) && (y>=0) && (y<tubelength_y)){
        // Particle is inside tube
        if ((oxz>=upper_zones) || (xz>=upper_zones)) {
          x=tubelength_x-x;
          vx=-vx;
          oxz=num_x_zones-1-oxz;
          xz=num_x_zones-1-xz;
        };

        if (xz>oxz){
          // Particles are going towards middle of tube: use emit_dists
          if (vx>0) {
            theta=atan(vy/vx);
          }
          else if (vx<0){
            if (vy<0){
              theta=atan(vy/vx)-Pi;
            }
            else {
              theta=atan(vy/vx)+Pi;
            };
          }
          else {
            // vx==0
            if (vy>0) {theta=Pi/2;} else {theta=-Pi/2;};
          };

          theta_index=(long)((theta-min_crash_theta)/theta_dist_binwidth);
          if ((theta_index>=0) && (theta_index<crash_bins)){
            theta_emit_dist[xz-1][theta_index]+=1;
            part_in_theta_emit_dist[xz-1]++;
          }
          else {part_not_in_theta_emit_dist[xz-1]++;};

          v_index=(long)((v-min_crash_v)/v_dist_binwidth);
          if ((v_index>=0) && (v_index<crash_bins)){
            v_emit_dist[xz-1][v_index]+=1;
            part_in_v_emit_dist[xz-1]++;
          }
          else {part_not_in_v_emit_dist[xz-1]++;}

          v2_index=(long)((v2-min_crash_v2)/v2_dist_binwidth);
          if ((v2_index>=0) && (v2_index<crash_bins)){
            v2_emit_dist[xz-1][v2_index]+=1;
            part_in_v2_emit_dist[xz-1]++;
          }
          else {part_not_in_v2_emit_dist[xz-1]++;}

          vx_index=(long)((vx-min_crash_vx)/vx_dist_binwidth);
          if ((vx_index>=0) && (vx_index<crash_bins)){
            vx_emit_dist[xz-1][vx_index]+=1;
            part_in_vx_emit_dist[xz-1]++;
          }
          else {part_not_in_vx_emit_dist[xz-1]++;};

          vy_index=(long)((vy-min_crash_vy)/vy_dist_binwidth);
          if ((vy_index>=0) && (vy_index<crash_bins)){
            vy_emit_dist[xz-1][vy_index]+=1;
            part_in_vy_emit_dist[xz-1]++;
          }
          else {part_not_in_vy_emit_dist[xz-1]++;};
        }
        else if (xz<oxz){
```

```
          vx=-vx;

          if (vx>0) {
            theta=atan(vy/vx);
          }
          else if (vx<0){
            if (vy<0){
              theta=atan(vy/vx)-Pi;
            }
            else {
              theta=atan(vy/vx)+Pi;
            };
          }
          else {
            // vx==0
            if (vy>0) {theta=Pi/2;} else {theta=-Pi/2;};
          };

          theta_index=(long)((theta-min_crash_theta)/theta_dist_binwidth);
          if ((theta_index>=0) && (theta_index<crash_bins)){
            theta_crash_dist[xz][theta_index]+=1;
            part_in_theta_crash_dist[xz]++;
          }
          else {part_not_in_theta_crash_dist[xz]++;};

          v_index=(long)((v-min_crash_v)/v_dist_binwidth);
          if ((v_index>=0) && (v_index<crash_bins)){
            v_crash_dist[xz][v_index]+=1;
            part_in_v_crash_dist[xz]++;
          }
          else {part_not_in_v_crash_dist[xz]++;};

          v2_index=(long)((v2-min_crash_v2)/v2_dist_binwidth);
          if ((v2_index>=0) && (v2_index<crash_bins)){
            v2_crash_dist[xz][v2_index]+=1;
            part_in_v2_crash_dist[xz]++;
          }
          else {part_not_in_v2_crash_dist[xz]++;};

          vx_index=(long)((vx-min_crash_vx)/vx_dist_binwidth);
          if ((vx_index>=0) && (vx_index<crash_bins)){
            vx_crash_dist[xz][vx_index]+=1;
            part_in_vx_crash_dist[xz]++;
          }
          else {part_not_in_vx_crash_dist[xz]++;};

          vy_index=(long)((vy-min_crash_vy)/vy_dist_binwidth);
          if ((vy_index>=0) && (vy_index<crash_bins)){
            vy_crash_dist[xz][vy_index]+=1;
            part_in_vy_crash_dist[xz]++;
          }
          else {part_not_in_vy_crash_dist[xz]++;};
        };
        total_particles++;
      };
    };
#elif ((X_ZONES==FALSE) && (Y_ZONES))
    printf("Not implemented!\n");
#elif (X_ZONES && Y_ZONES)
    printf("Not implemented!\n");
#endif
  };//for (i=1 to num_of_particles)
};

void parameters::norm_crash_dists()
{
  long i,j;

  // If there is a line in the middle, join the two last distributions
  if (num_of_dists%2!=0){
    part_in_vx_emit_dist[num_of_dists-1]+=part_in_vx_crash_dist[num_of_dists-1];
    part_in_vx_crash_dist[num_of_dists-1]=0;

    part_in_vy_emit_dist[num_of_dists-1]+=part_in_vy_crash_dist[num_of_dists-1];
    part_in_vy_crash_dist[num_of_dists-1]=0;

    part_in_v_emit_dist[num_of_dists-1]+=part_in_v_crash_dist[num_of_dists-1];
    part_in_v_crash_dist[num_of_dists-1]=0;

    part_in_theta_emit_dist[num_of_dists-1]+=part_in_theta_crash_dist[num_of_dists-1];
    part_in_theta_crash_dist[num_of_dists-1]=0;

    part_in_v2_emit_dist[num_of_dists-1]+=part_in_v_crash_dist[num_of_dists-1];
    part_in_v2_crash_dist[num_of_dists-1]=0;
```

```
    for (i=0;i<crash_bins;i++){
      vx_emit_dist[num_of_dists-1][i]+=vx_crash_dist[num_of_dists-1][i];
      vx_crash_dist[num_of_dists-1][i]=0;

      vy_emit_dist[num_of_dists-1][i]+=vy_crash_dist[num_of_dists-1][i];
      vy_crash_dist[num_of_dists-1][i]=0;

      v_emit_dist[num_of_dists-1][i]+=v_crash_dist[num_of_dists-1][i];
      v_crash_dist[num_of_dists-1][i]=0;

      theta_emit_dist[num_of_dists-1][i]+=theta_crash_dist[num_of_dists-1][i];
      theta_crash_dist[num_of_dists-1][i]=0;

      v2_emit_dist[num_of_dists-1][i]+=v2_crash_dist[num_of_dists-1][i];
      v2_crash_dist[num_of_dists-1][i]=0;
    };
  };

  // Normalize distributions
  for (j=0;j<num_of_dists;j++){
    for (i=0;i<crash_bins;i++){
      if (part_in_vx_emit_dist[j]!=0)
        {vx_emit_dist[j][i]/=(FLOATTYPE)part_in_vx_emit_dist[j]*vx_dist_binwidth;};
      if (part_in_vx_crash_dist[j]!=0)
        {vx_crash_dist[j][i]/=(FLOATTYPE)part_in_vx_crash_dist[j]*vx_dist_binwidth;};

      if (part_in_vy_emit_dist[j]!=0)
        {vy_emit_dist[j][i]/=(FLOATTYPE)part_in_vy_emit_dist[j]*vy_dist_binwidth;};
      if (part_in_vy_crash_dist[j]!=0)
        {vy_crash_dist[j][i]/=(FLOATTYPE)part_in_vy_crash_dist[j]*vy_dist_binwidth;};

      if (part_in_v_emit_dist[j]!=0)
        {v_emit_dist[j][i]/=(FLOATTYPE)part_in_v_emit_dist[j]*v_dist_binwidth;};
      if (part_in_v_crash_dist[j]!=0)
        {v_crash_dist[j][i]/=(FLOATTYPE)part_in_v_crash_dist[j]*v_dist_binwidth;};

      if (part_in_theta_emit_dist[j]!=0)
        {theta_emit_dist[j][i]/=(FLOATTYPE)part_in_theta_emit_dist[j]*theta_dist_binwidth;};
      if (part_in_theta_crash_dist[j]!=0)
        {theta_crash_dist[j][i]/=(FLOATTYPE)part_in_theta_crash_dist[j]*theta_dist_binwidth;};

      if (part_in_v2_emit_dist[j]!=0)
        {v2_emit_dist[j][i]/=(FLOATTYPE)part_in_v2_emit_dist[j]*v2_dist_binwidth;};
      if (part_in_v2_crash_dist[j]!=0)
        {v2_crash_dist[j][i]/=(FLOATTYPE)part_in_v2_crash_dist[j]*v2_dist_binwidth;};
    };
  };
};

void parameters::write_crash_dists()
{
  // Write some variables to screen
  long i,j;
  long sum_part=0;

  for (j=0;j<num_of_dists;j++){
    printf("part_in_vx_emit_dist[%d]: %d\n",j,part_in_vx_emit_dist[j]);
    printf("part_in_vx_crash_dist[%d]: %d\n",j,part_in_vx_crash_dist[j]);
    sum_part+=part_in_vx_emit_dist[j]+part_in_vx_crash_dist[j];
  };
  printf("total vx: %d\n\n",sum_part);

  sum_part=0;
  for (j=0;j<num_of_dists;j++){
    printf("part_not_in_vx_emit_dist[%d]: %d\n",j,part_not_in_vx_emit_dist[j]);
    printf("part_not_in_vx_crash_dist[%d]: %d\n",j,part_not_in_vx_crash_dist[j]);
    sum_part+=part_not_in_vx_emit_dist[j]+part_not_in_vx_crash_dist[j];
  };
  printf("total not in vx: %d\n\n",sum_part);

  sum_part=0;
  for (j=0;j<num_of_dists;j++){
    printf("part_in_vy_emit_dist[%d]: %d\n",j,part_in_vy_emit_dist[j]);
    printf("part_in_vy_crash_dist[%d]: %d\n",j,part_in_vy_crash_dist[j]);
    sum_part+=part_in_vy_emit_dist[j]+part_in_vy_crash_dist[j];
  };
  printf("total vy: %d\n\n",sum_part);

  sum_part=0;
  for (j=0;j<num_of_dists;j++){
    printf("part_not_in_vy_emit_dist[%d]: %d\n",j,part_not_in_vy_emit_dist[j]);
    printf("part_not_in_vy_crash_dist[%d]: %d\n",j,part_not_in_vy_crash_dist[j]);
```

```
  sum_part+=part_not_in_vy_emit_dist[j]+part_not_in_vy_crash_dist[j];
};
printf("total not in vy: %d\n\n",sum_part);

sum_part=0;
for (j=0;j<num_of_dists;j++){
  printf("part_in_v_emit_dist[%d]: %d\n",j,part_in_v_emit_dist[j]);
  printf("part_in_v_crash_dist[%d]: %d\n",j,part_in_v_crash_dist[j]);
  sum_part+=part_in_v_emit_dist[j]+part_in_v_crash_dist[j];
};
printf("total v: %d\n\n",sum_part);

sum_part=0;
for (j=0;j<num_of_dists;j++){
  printf("part_not_in_v_emit_dist[%d]: %d\n",j,part_not_in_v_emit_dist[j]);
  printf("part_not_in_v_crash_dist[%d]: %d\n",j,part_not_in_v_crash_dist[j]);
  sum_part+=part_not_in_v_emit_dist[j]+part_not_in_v_crash_dist[j];
};
printf("total not in v: %d\n\n",sum_part);

sum_part=0;
for (j=0;j<num_of_dists;j++){
  printf("part_in_theta_emit_dist[%d]: %d\n",j,part_in_theta_emit_dist[j]);
  printf("part_in_theta_crash_dist[%d]: %d\n",j,part_in_theta_crash_dist[j]);
  sum_part+=part_in_theta_emit_dist[j]+part_in_theta_crash_dist[j];
};
printf("total theta: %d\n\n",sum_part);

sum_part=0;
for (j=0;j<num_of_dists;j++){
  printf("part_not_in_theta_emit_dist[%d]: %d\n",j,part_not_in_theta_emit_dist[j]);
  printf("part_not_in_theta_crash_dist[%d]: %d\n",j,part_not_in_theta_crash_dist[j]);
  sum_part+=part_not_in_theta_emit_dist[j]+part_not_in_theta_crash_dist[j];
};
printf("total not in theta: %d\n\n",sum_part);

sum_part=0;
for (j=0;j<num_of_dists;j++){
  printf("part_in_v2_emit_dist[%d]: %d\n",j,part_in_v2_emit_dist[j]);
  printf("part_in_v2_crash_dist[%d]: %d\n",j,part_in_v2_crash_dist[j]);
  sum_part+=part_in_v2_emit_dist[j]+part_in_v2_crash_dist[j];
};
printf("total v2: %d\n\n",sum_part);

sum_part=0;
for (j=0;j<num_of_dists;j++){
  printf("part_not_in_v2_emit_dist[%d]: %d\n",j,part_not_in_v2_emit_dist[j]);
  printf("part_not_in_v2_crash_dist[%d]: %d\n",j,part_not_in_v2_crash_dist[j]);
  sum_part+=part_not_in_v2_emit_dist[j]+part_not_in_v2_crash_dist[j];
};
printf("total not in v2: %d\n\n",sum_part);

printf("Total number of particles: %d\n",total_particles);


// The outfiles
char outfilename_vx_emit_dist[num_of_dists][256];
char outfilename_vy_emit_dist[num_of_dists][256];
char outfilename_v_emit_dist[num_of_dists][256];
char outfilename_theta_emit_dist[num_of_dists][256];
char outfilename_v2_emit_dist[num_of_dists][256];

char outfilename_vx_crash_dist[num_of_dists][256];
char outfilename_vy_crash_dist[num_of_dists][256];
char outfilename_v_crash_dist[num_of_dists][256];
char outfilename_theta_crash_dist[num_of_dists][256];
char outfilename_v2_crash_dist[num_of_dists][256];

FILE *fp;

for (i=0;i<num_of_dists;i++){
  /* Horizontal velocity distribution
     (the velocities from the right wall are negated) */
  sprintf(outfilename_vx_emit_dist[i],"%s%s%d%s",DATA_PATH,"vx_emit_dist",i,".dat");
  sprintf(outfilename_vx_crash_dist[i],"%s%s%d%s",DATA_PATH,"vx_crash_dist",i,".dat");

  /* Vertical velocity distribution
     (the velocities from the upper wall are negated) */
  sprintf(outfilename_vy_emit_dist[i],"%s%s%d%s",DATA_PATH,"vy_emit_dist",i,".dat");
  sprintf(outfilename_vy_crash_dist[i],"%s%s%d%s",DATA_PATH,"vy_crash_dist",i,".dat");


  /* Velocity distribution */
  sprintf(outfilename_v_emit_dist[i],"%s%s%d%s",DATA_PATH,"v_emit_dist",i,".dat");
```

```
  sprintf(outfilename_v_crash_dist[i],"%s%s%d%s",DATA_PATH,"v_crash_dist",i,".dat");

  /* The average angle from the walls.
     0 degrees is normal to the wall,
     +Pi/2 is a quarter circle ccw, -Pi/2 is a quarter circle cw.*/
  // The distributions:
  sprintf(outfilename_theta_emit_dist[i],"%s%s%d%s",DATA_PATH,"theta_emit_dist",i,".dat");
  sprintf(outfilename_theta_crash_dist[i],"%s%s%d%s",DATA_PATH,"theta_crash_dist",i,".dat");

  /* Velocity squared distribution */
  sprintf(outfilename_v2_emit_dist[i],"%s%s%d%s",DATA_PATH,"v2_emit_dist",i,".dat");
  sprintf(outfilename_v2_crash_dist[i],"%s%s%d%s",DATA_PATH,"v2_crash_dist",i,".dat");
};


/* Write distributions to file */
for (j=0;j<num_of_dists;j++){

  if (part_in_vx_emit_dist[j]!=0) {
    if ((fp = fopen(outfilename_vx_emit_dist[j],"w")) != NULL){
      for (i=0;i<crash_bins;i++){
        fprintf(fp,"%g %g\n",min_crash_vx+(0.5+i)*vx_dist_binwidth,vx_emit_dist[j][i]);
      };
      fclose(fp);
      printf("File successfully written to disk: %s\n",outfilename_vx_emit_dist[j]);
    } else {printf("Unable to open file: %s\n",outfilename_vx_emit_dist[j]);};
  };

  if (part_in_vx_crash_dist[j]!=0) {
    if ((fp = fopen(outfilename_vx_crash_dist[j],"w")) != NULL){
      for (i=0;i<crash_bins;i++){
        fprintf(fp,"%g %g\n",min_crash_vx+(0.5+i)*vx_dist_binwidth,vx_crash_dist[j][i]);
      };
      fclose(fp);
      printf("File successfully written to disk: %s\n",outfilename_vx_crash_dist[j]);
    } else {printf("Unable to open file: %s\n",outfilename_vx_crash_dist[j]);};
  };

  if (part_in_vy_emit_dist[j]!=0) {
    if ((fp = fopen(outfilename_vy_emit_dist[j],"w")) != NULL){
      for (i=0;i<crash_bins;i++){
        fprintf(fp,"%g %g\n",min_crash_vy+(0.5+i)*vy_dist_binwidth,vy_emit_dist[j][i]);
      };
      fclose(fp);
      printf("File successfully written to disk: %s\n",outfilename_vy_emit_dist[j]);
    } else {printf("Unable to open file: %s\n",outfilename_vy_emit_dist[j]);};
  };

  if (part_in_vy_crash_dist[j]!=0) {
    if ((fp = fopen(outfilename_vy_crash_dist[j],"w")) != NULL){
      for (i=0;i<crash_bins;i++){
        fprintf(fp,"%g %g\n",min_crash_vy+(0.5+i)*vy_dist_binwidth,vy_crash_dist[j][i]);
      };
      fclose(fp);
      printf("File successfully written to disk: %s\n",outfilename_vy_crash_dist[j]);
    } else {printf("Unable to open file: %s\n",outfilename_vy_crash_dist[j]);};
  };

  if (part_in_v_emit_dist[j]!=0) {
    if ((fp = fopen(outfilename_v_emit_dist[j],"w")) != NULL){
      for (i=0;i<crash_bins;i++){
        fprintf(fp,"%g %g\n",min_crash_v+(0.5+i)*v_dist_binwidth,v_emit_dist[j][i]);
      };
      fclose(fp);
      printf("File successfully written to disk: %s\n",outfilename_v_emit_dist[j]);
    } else {printf("Unable to open file: %s\n",outfilename_v_emit_dist[j]);};
  };

  if (part_in_v_crash_dist[j]!=0) {
    if ((fp = fopen(outfilename_v_crash_dist[j],"w")) != NULL){
      for (i=0;i<crash_bins;i++){
        fprintf(fp,"%g %g\n",min_crash_v+(0.5+i)*v_dist_binwidth,v_crash_dist[j][i]);
      };
      fclose(fp);
      printf("File successfully written to disk: %s\n",outfilename_v_crash_dist[j]);
    } else {printf("Unable to open file: %s\n",outfilename_v_crash_dist[j]);};
  };

  if (part_in_theta_emit_dist[j]!=0) {
    if ((fp = fopen(outfilename_theta_emit_dist[j],"w")) != NULL){
      for (i=0;i<crash_bins;i++){
        fprintf(fp,"%g %g\n",min_crash_theta+(0.5+i)*theta_dist_binwidth,theta_emit_dist[j][i]);
      };
      fclose(fp);
```

```
        printf("File successfully written to disk: %s\n",outfilename_theta_emit_dist[j]);
      } else {printf("Unable to open file: %s\n",outfilename_theta_emit_dist[j]);};
    };

    if (part_in_theta_crash_dist[j]!=0) {
      if ((fp = fopen(outfilename_theta_crash_dist[j],"w")) != NULL){
        for (i=0;i<crash_bins;i++){
          fprintf(fp,"%g %g\n",min_crash_theta+(0.5+i)*theta_dist_binwidth,theta_crash_dist[j][i]);
        };
        fclose(fp);
        printf("File successfully written to disk: %s\n",outfilename_theta_crash_dist[j]);
      } else {printf("Unable to open file: %s\n",outfilename_theta_crash_dist[j]);};
    };

    if (part_in_v2_emit_dist[j]!=0) {
      if ((fp = fopen(outfilename_v2_emit_dist[j],"w")) != NULL){
        for (i=0;i<crash_bins;i++){
          fprintf(fp,"%g %g\n",min_crash_v2+(0.5+i)*v2_dist_binwidth,v2_emit_dist[j][i]);
        };
        fclose(fp);
        printf("File successfully written to disk: %s\n",outfilename_v2_emit_dist[j]);
      } else {printf("Unable to open file: %s\n",outfilename_v2_emit_dist[j]);};
    };

    if (part_in_v2_crash_dist[j]!=0) {
      if ((fp = fopen(outfilename_v2_crash_dist[j],"w")) != NULL){
        for (i=0;i<crash_bins;i++){
          fprintf(fp,"%g %g\n",min_crash_v2+(0.5+i)*v2_dist_binwidth,v2_crash_dist[j][i]);
        };
        fclose(fp);
        printf("File successfully written to disk: %s\n",outfilename_v2_crash_dist[j]);
      } else {printf("Unable to open file: %s\n",outfilename_v2_crash_dist[j]);};
    };
  };
};
#endif
```

B.1.7 MD_init.hh

```
/* MD_init.hh, header file for MD_init.cc */

#ifndef INIT
#define INIT

/*****************************************************************************/
/* GIVE EACH BOX ITS BOUNDARY                                                */
/* MAKE A LIST FOR EACH BOX, CONSISTING OF THE BOXES WITH WHICH IT SHOULD REACT */
/* PREPARE WALLS                                                             */
/* PLACE PARTICLES IN A LATTICE, AND GIVE THEM A RANDOM VELOCITY             */
/* PLACE PARTICLES IN BOXES                                                  */
/*****************************************************************************/
int init_variables(class particle *, class box *, class parameters *);

#endif
```

B.1.8 MD_init.cc

```
/* MD_init.cc, initialize variables */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "MD_definitions.hh"
#include "MD_constants.hh"
#include "MD_classes.hh"
#include "MD_listtools.hh"
#include "MD_calc.hh"
#include "MD_init.hh"

int init_variables(class particle prt[], class box boxes[], class parameters *param)
{
  int init_OK=TRUE;
  long i,j,count;
#if (WALL_TYPE==LJ)
// The place where the LJ-potential is kT
  FLOATTYPE pot_is_kT=pow(2*pow(init_temperature,5)*(sqrt(1+init_temperature)-1),0.166666667)/init_temperature;
#endif

#if (TRIGGER_WALLS)
#if (H_WALLS)
  if (tubelength_y<2*horiwall_grabdist){
    init_OK=FALSE;
```

```
    printf("WARNING: tubelength_y < 2*horiwall_grabdist.\n");
  };
#endif
#if (V_WALLS)
  if (tubelength_x<2*vertwall_grabdist){
    init_OK=FALSE;
    printf("WARNING: tubelength_x < 2*vertwall_grabdist.\n");
  };
#endif
#endif

#if (H_WALLS==FALSE)
  if (no_of_particles_pr_boxlength*no_of_y_boxes<2*sqrt(number_density*r2_cutoff)){
    init_OK=FALSE;
    printf("WARNING: The product no_of_particles_pr_boxlength*no_of_y_boxes is too small.\n");
  };
#endif

#if (V_WALLS==FALSE)
  if (no_of_particles_pr_boxlength*no_of_x_boxes<2*sqrt(number_density*r2_cutoff)){
    init_OK=FALSE;
    printf("WARNING: The product no_of_particles_pr_boxlength*no_of_x_boxes is too small.\n");
  };
#endif


  // PREPARE TABLES
#if (WALL_TYPE==SDD)
  FLOATTYPE v;
  FLOATTYPE K_sum;
  FLOATTYPE threshold;
  const FLOATTYPE alpha2=2*wall_temperature; // alpha squared
  const FLOATTYPE prefact=1/sqrt(alpha2*Pi);
  const FLOATTYPE prefact_v=2/alpha2;

  // Table for normal Maxwellian distribution in one dimension
  count=0; K_sum=0; threshold=0; v=0;
  for (j=0;count<(nob/2);j++){
    K_sum+=FLOATTYPE(prefact*exp(-v*v/alpha2)*dv);
    while ((K_sum>=threshold) && (count<(nob/2))){
      param->K_inv[long(nob/2) + count]=v;
      param->K_inv[long(nob/2) - count]=-v;
      count++;
      threshold=K_factor*(FLOATTYPE)count/(FLOATTYPE)nob;
    };
    v+=dv;
  };
  param->K_inv[0]=param->K_inv[1]; // param->K_inv[0] shouldn't really be used...
  printf("K_inv[1]=%.16g  K_inv[%d]=%.16g  K_sum=%.16g\n",param->K_inv[1],nob-1,param->K_inv[nob-1],K_sum);

  // Table for Maxwell distribution multiplied with v (and normalized)
  count=0; K_sum=0; threshold=0; v=0;
  for (j=0;count<nob;j++){
    K_sum+=(FLOATTYPE)(prefact_v*v*exp(-v*v/alpha2)*dv);
    while ((K_sum>=threshold) && (count<nob)){
      param->Kv_inv[count]=v;
      count++;
      threshold=Kv_factor*(FLOATTYPE)count/(FLOATTYPE)nob;
    };
    v+=dv;
  };
  printf("Kv_inv[0]=%.16g  Kv_inv[%d]=%.16g  Kv_sum=%.16g\n",param->Kv_inv[0],nob-1,param->Kv_inv[nob-1],K_sum);
#endif

#if ((WALL_TYPE==SUD) || (WALL_TYPE==SCX))
  FLOATTYPE v;
  FLOATTYPE K_sum;
  FLOATTYPE threshold;
  const FLOATTYPE alpha2=2*wall_temperature; // alpha squared
  const FLOATTYPE prefact_v2=4/sqrt(pow(alpha2,3)*Pi);

  // Table for Maxwell distribution multiplied with v^2 (and normalized)
  count=0; K_sum=0; threshold=0; v=0;
  for (j=0;count<nob;j++){
    K_sum+=(FLOATTYPE)(prefact_v2*v*v*exp(-v*v/alpha2)*dv);
    while ((K_sum>=threshold) && (count<nob)){
      param->Kv2_inv[count]=v;
      count++;
      threshold=Kv2_factor*(FLOATTYPE)count/(FLOATTYPE)nob;
    };
    v+=dv;
  };
  printf("Kv2_inv[0]=%.16g  Kv2_inv[%d]=%.16g  Kv2_sum=%.16g\n",
         param->Kv2_inv[0],nob-1,param->Kv2_inv[nob-1],K_sum);
```

```
#endif


  // PREPARE BOXES
  // (The program has always used square boxes)
  for (j=0;j<no_of_y_boxes;j++){
    for (i=0;i<no_of_x_boxes;i++){
      count=i+j*no_of_x_boxes;

      // GIVE EACH BOX ITS KEY
      boxes[count].boxnr=count;

      // GIVE EACH BOX ITS BOUNDARY
      boxes[count].bottomleft.x=i*boxlength;
      boxes[count].bottomleft.y=j*boxlength;
      boxes[count].topright.x=(i+1)*boxlength;
      boxes[count].topright.y=(j+1)*boxlength;

      // This line is superfluous if the constructor works
      boxes[count].first_particle=NULL;
    };
  };


  printf("Make boxlist...\n");

  // FOR EACH BOX, MAKE A LIST TO OTHER BOXES IT SHOULD 'REACT' WITH
  // There should be NO boxes that point to EACH OTHER

  char connect_boxes;
  class boxlist *b;

  for (j=0;j<no_of_boxes;j++){
    for (i=0;i<no_of_boxes;i++){
      if (i!=j){
        // still ok to connect
        connect_boxes=TRUE;
        b=boxes[i].first_box;
        while (b!=NULL){
          if (b->boxnr==j) {connect_boxes=FALSE;};
          b=b->next_box;
        };
        if (connect_boxes){
          // still ok to connect
          FLOATTYPE dx=0,dy=0;

          if (i/no_of_x_boxes!=j/no_of_x_boxes){
            // not on the same row

            // lower and upper edge (a and b)
            // of lowermost and uppermost box (1 and 2), respectively.
            FLOATTYPE r1a,r1b,r2a,r2b;

            FLOATTYPE temp;
            r1a=boxes[j].bottomleft.y;
            r1b=boxes[j].topright.y;
            r2a=boxes[i].bottomleft.y;
            r2b=boxes[i].topright.y;
            if (r1a>r2a) {temp=r1a;r1a=r2a;r2a=temp;temp=r1b;r1b=r2b;r2b=temp;}; // make 1 smaller than 2
#if (H_WALLS)
            dy=r2a-r1b;
#else
            dy=min(r2a-r1b,(tubelength_y-r2b)+r1a);
#endif
          };

          if (i%no_of_x_boxes!=j%no_of_x_boxes){
            // not on the same column

            // left and right edge (a and b)
            // of leftmost and rightmost box (1 and 2), respectively.
            FLOATTYPE r1a,r1b,r2a,r2b;

            FLOATTYPE temp;
            r1a=boxes[j].bottomleft.x;
            r1b=boxes[j].topright.x;
            r2a=boxes[i].bottomleft.x;
            r2b=boxes[i].topright.x;
            if (r1a>r2a) {temp=r1a;r1a=r2a;r2a=temp;temp=r1b;r1b=r2b;r2b=temp;}; // make 1 smaller than 2
#if (V_WALLS==TRUE)
            dx=r2a-r1b;
#else
            dx=min(r2a-r1b,(tubelength_x-r2b)+r1a);
#endif
```

```
          };

          if ((dx*dx+dy*dy)<=r2_cutoff) add_box_to_boxlist(&boxes[j],i);
        };
      };
    };
  };

  printf("Prepare walls...\n");

  // PREPARE WALLS

#if (H_WALLS)
  param->h_wall[0].y=0;
  param->h_wall[1].y=tubelength_y;

#if (WALL_TYPE==LJ)
  param->h_wall[0].adjust=-pot_is_kT+horiwall_grabdist;
  param->h_wall[1].adjust=pot_is_kT-horiwall_grabdist;
#endif

  param->h_wall[0].theta=Pi/2;
  param->h_wall[1].theta=3*Pi/2;
#endif


#if (V_WALLS)
  param->v_wall[0].x=0;
  param->v_wall[1].x=tubelength_x;

#if (WALL_TYPE==LJ)
  param->v_wall[0].adjust=-pot_is_kT+vertwall_grabdist;
  param->v_wall[1].adjust=pot_is_kT-vertwall_grabdist;
#endif

  param->v_wall[0].theta=0;
  param->v_wall[1].theta=Pi;
#endif


  printf("Preparing zones...\n");
  // PREPARE ZONES

  // The first zone must be grabdist,
  // the second next-last must be tubelength-grabdist
  // and the last must be tubelength.
#if (X_ZONES)
  param->x_zone[0]=vertwall_grabdist;
  param->x_zone[1]=1;
  param->x_zone[2]=2;
  param->x_zone[3]=3;
  param->x_zone[4]=4;
  param->x_zone[5]=tubelength_x/2;
  param->x_zone[6]=tubelength_x-4;
  param->x_zone[7]=tubelength_x-3;
  param->x_zone[8]=tubelength_x-2;
  param->x_zone[9]=tubelength_x-1;
  param->x_zone[10]=tubelength_x-vertwall_grabdist;
  param->x_zone[11]=tubelength_x;
#endif
#if (Y_ZONES)
  param->y_zone[0]=horiwall_grabdist;
  param->y_zone[1]=1;
  param->y_zone[2]=2;
  param->y_zone[3]=3;
  param->y_zone[4]=4;
  param->y_zone[5]=tubelength_y/2;
  param->y_zone[6]=tubelength_y-4;
  param->y_zone[7]=tubelength_y-3;
  param->y_zone[8]=tubelength_y-2;
  param->y_zone[9]=tubelength_y-1;
  param->y_zone[10]=tubelength_y-horiwall_grabdist;
  param->y_zone[11]=tubelength_y;
#endif

  printf("Preparing particles...\n");
  // PREPARE PARTICLES
  for (j=0;j<no_of_y_particles;j++){
    for (i=0;i<no_of_x_particles;i++){
      count = i+j*no_of_x_particles;
      // GIVE THE PARTICLES THEIR KEY
      prt[count].partnr=count;

      // PLACE PARTICLES IN A LATTICE, GIVE THEM MASS AND A RANDOM VELOCITY
```

```
#if (V_WALLS)
      // Make sure particles are well inside of box, not near walls.
      FLOATTYPE x_space=(tubelength_x-2*vertwall_grabdist)*0.9;
      FLOATTYPE x_disp=x_space/FLOATTYPE(no_of_x_particles);
      prt[count].r.x = (tubelength_x-x_space)/2+(i+0.5*(1+(drand48()-0.5)/FLOATTYPE(100)))*x_disp;
#else
      prt[count].r.x = dist_between_part*(i+0.5*(1+(drand48()-0.5)/FLOATTYPE(100)));
#endif
#if (H_WALLS)
      // Make sure particles are well inside of box, not near walls.
      FLOATTYPE y_space=(tubelength_y-2*horiwall_grabdist);
      FLOATTYPE y_disp=y_space/FLOATTYPE(no_of_y_particles);
      prt[count].r.y = (tubelength_y-y_space)/2 +(j+0.5*(1+(drand48()-0.5)/FLOATTYPE(100)))*y_disp;
#else
      prt[count].r.y = dist_between_part*(j+0.5*(1+(drand48()-0.5)/FLOATTYPE(100)));
#endif

      prt[count].ov.vx = max_vx*(drand48()*2-1);
      prt[count].ov.vy = max_vy*(drand48()*2-1);

      // PLACE PARTICLES IN BOXES
      for (int boxcnt=0;boxcnt<no_of_boxes;boxcnt++){
        if ((prt[count].r.x>boxes[boxcnt].bottomleft.x)
            && (prt[count].r.x<boxes[boxcnt].topright.x)
            && (prt[count].r.y>boxes[boxcnt].bottomleft.y)
            && (prt[count].r.y<boxes[boxcnt].topright.y)){
          put_particle_in_box(&prt[count],&boxes[boxcnt]);
          prt[count].boxnr=boxes[boxcnt].boxnr;
          break;
        };
      };
    };
  };

  printf("Preparing zones...\n");

#if (ZONES)
  // Update the zone variables (do it twice to get it right)
  update_zones(prt,param);
  update_zones(prt,param);
#endif

#if (REC_H_FUNCTION)
  param->init_fv();
#endif
  return(init_OK);
};
```

B.1.9 MD_calc.hh

```
/* MD_calc.hh, header file for MD_calc.cc */

#ifndef CALC
#define CALC

/*********************************************************************/
/* FUNCTION: Calculates force/acceleration by running through the three */
/*           above procedures for all particles and walls.           */
/*           This procedure doesn NOT alter positions or velocities.  */
/* INPUT: Particles, boxes, parameters.                              */
/*********************************************************************/
void calculate_forces(class particle *, class box *, class parameters *);

/*********************************************************************/
/* FUNCTION: Integrates (moves particles) one timestep ahead.        */
/* INPUT: Particles, boxes, parameters, current cycle.               */
/* OUTPUT: 0 : Everything is OK.                                     */
/*         1 : One or more of the particles has escaped from the tube.  */
/*             Abort immediately.                                    */
/*********************************************************************/
char integrate(class particle *, class box *, class parameters *);

#if (ZONES)
/*********************************************************************/
/* FUNCTION: Updates the particles' zone variables, which specify    */
/* which zone they currently occupy.                                 */
/* INPUT: Particles, parameters.                                     */
/*********************************************************************/
void update_zones(class particle *, class parameters *);
#endif

#if (INIT_RESCALE_VEL)
/*********************************************************************/
```

```
/* FUNCTION: Rescale velocities of particles so they match the given    */
/* temperature.                                                         */
/* INPUT: Particles, parameters, new temperature                        */
/************************************************************************/
void rescale_vel(class particle *, class parameters *, FLOATTYPE);
#endif

/************************************************************************/
/* FUNCTION: runs calculate_forces, integrate and update_touchvals      */
/* INPUT: Particles, boxes, parameters, current cycle.                  */
/* OUTPUT: Same as for integration.                                     */
/************************************************************************/
char calculate_next_step(class particle *, class box *, class parameters *);

#endif
```

B.1.10 MD_calc.cc

```
/* MD_calc.cc, module for calculating next step, parameters etc. */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "MD_definitions.hh"
#include "MD_constants.hh"
#include "MD_classes.hh"
#include "MD_listtools.hh"
#include "MD_calc.hh"
#include "MD_calc_inline.hh"


/*****************************************************************************/
/* CALCULATE NEXT STEP (runs calculate_forces and integration)               */
/*****************************************************************************/
char calculate_next_step(class particle prt[], class box boxes[], class parameters *param)
{
  calculate_forces(prt, boxes, param);
  char abort_now=integrate(prt, boxes, param);
#if (ZONES)
  if (!abort_now) {update_zones(prt, param);};
#endif
  return(abort_now);
};


/*****************************************************************************/
/* CALCULATE FORCES (runs the procedures below)                              */
/*****************************************************************************/
void calculate_forces(class particle prt[], class box boxes[], class parameters *param )
{
  long i;

  /* Make accelerations ready for use */
  for (i=0;i<no_of_particles;i++) {
#if (CONSTANT_FORCE)
    // Don't divide by mass, since m=1
    prt[i].a.ax=constant_force_x;
    prt[i].a.ay=constant_force_y;
#else
    prt[i].a.ax=0;
    prt[i].a.ay=0;
#endif
  };

  param->E_pot_part=0;
#if (WALL_TYPE==LJ)
  param->E_pot_wall=0;
#endif

  class particle *p;
  class boxlist *b;

  // Calculate acceleration.
  // I go through every particle.
  // For each particle, I pair it with all the other particles in the same box FURTHER DOWN THE LIST.
  // Then I go through all the boxes on the 'reaction' list,
  // and compare the particle with all the particles in these other boxes.
  // This SHOULD mean that each PAIR is calculated once.

  // After that, all the particles are checked if they interact with the walls.
  // To make this more efficient, I should let the walls point to the boxes that
  // it should react with.
```

```
  for (i=0;i<no_of_particles;i++) {
    // Calculate with all the particles in the same box,
    // but only those FURTHER DOWN THE LIST
    p=prt[i].next_particle;
    while (p!=NULL) {
      calculate_accel_from_particles(prt[i],*p,param);
      p=p->next_particle;
    };

    // Calculate with all the particles in the surrounding boxes
    // (those in the 'reaction' list)
    b=boxes[prt[i].boxnr].first_box;
    while (b!=NULL) {
      p=boxes[b->boxnr].first_particle;
      while (p!=NULL) {
        calculate_accel_from_particles(prt[i],*p,param);
        p=p->next_particle;
      };
      b=b->next_box;
    };

    // Calculate with the walls.
#if (WALL_TYPE==LJ)
#if (H_WALLS)
    for (long jh=0;jh<2;jh++) {
      calculate_accel_from_horizontal_walls(prt[i],param->h_wall[jh],param);
    };
#endif
#if (V_WALLS)
    for (long jv=0;jv<2;jv++) {
      calculate_accel_from_vertical_walls(prt[i],param->v_wall[jv],param);
    };
#endif
#endif
  };

#if (WALL_TYPE==LJ)
  param->E_potential=param->E_pot_wall+param->E_pot_part;
#else
  param->E_potential=param->E_pot_part;
#endif
};



/*****************************************************************************/
/* INTEGRATION                                                               */
/*****************************************************************************/
char integrate(class particle prt[], class box boxes[], class parameters *param)
{
  long i;

  // Go through every particle and find new position.

#if (REC_KIN_CHANGE && STOCH_WALLS)
  param->E_kin_change=0;
#endif

  param->Px=0;
  param->Py=0;

  class velocity nv; // a halfstep in front of v

  // If the particle is behind the grabdist, touch should be set to true.
  // The particle will be integrated, but if it doesn't escape the wall,
  // it is moved (normal to the wall) until it is again inside the box.
  // The particle is thus not allowed to stay behind grabdist for more
  // than one timestep.

  for (i=0;i<no_of_particles;i++) {
#if (WALL_TYPE==HR)
    char touch=FALSE;
#if (V_WALLS)
    if (prt[i].x_zone==0){
      touch=TRUE;
      if (prt[i].ov.vx<0) {prt[i].ov.vx=-prt[i].ov.vx;};
    }
    else if (prt[i].x_zone==(num_x_zones-1)){
      touch=TRUE;
      if (prt[i].ov.vx>0) {prt[i].ov.vx=-prt[i].ov.vx;};
    };
#endif
#if (H_WALLS)
```

```
    if (prt[i].y_zone==0){
      touch=TRUE;
      if (prt[i].ov.vy<0) {prt[i].ov.vy=-prt[i].ov.vy;};
    }
    else if (prt[i].y_zone==(num_y_zones-1)){
      touch=TRUE;
      if (prt[i].ov.vy>0) {prt[i].ov.vy=-prt[i].ov.vy;};
    };
#endif
#elif (STOCH_WALLS)
    FLOATTYPE vx,vy;
    char touch=FALSE;

#if (UNI_THETA)
    FLOATTYPE theta_middle=0;
    FLOATTYPE theta_length=Pi;
    FLOATTYPE theta;

#if (H_WALLS)
    FLOATTYPE theta_h=0;
    if (prt[i].touch_hwall!=-1){
      touch=TRUE;
      theta_h=param->h_wall[prt[i].touch_hwall].theta;
      theta_middle=theta_h;
    };
#endif
#if (V_WALLS)
    FLOATTYPE theta_v=0;
    if (prt[i].touch_vwall!=-1)        {
      touch=TRUE;
      theta_v=param->v_wall[prt[i].touch_vwall].theta;
      theta_middle+=theta_v;
    };
#endif
#if ((H_WALLS) && (V_WALLS))
    if ((prt[i].touch_vwall!=-1) && (prt[i].touch_hwall!=-1)){
      theta_length=abs(theta_h-theta_v);
      if (theta_length>Pi){
        theta_middle+=2*Pi;
        theta_length=2*Pi-theta_length;
      };
      theta_middle/=2;
    };
#endif

    if (touch){
      theta=theta_middle+((drand48()-0.5)*theta_length);
#if (WALL_TYPE==SUC)
      vx=v0*cos(theta);
      vy=v0*sin(theta);
#elif (WALL_TYPE==SUD)
      FLOATTYPE speed=param->Kv2_inv[long(drand48()*nob)];
      vx=speed*cos(theta);
      vy=speed*sin(theta);
#endif
    };

#elif (WALL_TYPE==SCC)
    vx=0; vy=0;
#if (H_WALLS)
    if (prt[i].y_zone==0){touch=TRUE;vy=v0;}
    else if (prt[i].y_zone==(num_y_zones-1)){touch=TRUE;vy=-v0;}
#endif
#if (V_WALLS)
    if (prt[i].x_zone==0){touch=TRUE;vx=v0;}
    else if (prt[i].x_zone==(num_x_zones-1)){touch=TRUE;vx=-v0;}
#endif

#elif (WALL_TYPE==SDD)
#if ((H_WALLS) && (V_WALLS==FALSE))
    // Only HORIZONTAL walls
    if ((prt[i].y_zone==0) || (prt[i].y_zone==(num_y_zones-1))){
      touch=TRUE;
      vx=param->K_inv[1+long(drand48()*(nob-1))];
      if (prt[i].y_zone==0){vy=param->Kv_inv[long(drand48()*nob)];}
      else {vy=-param->Kv_inv[long(drand48()*nob)];};
    };
#elif ((H_WALLS==FALSE) && (V_WALLS))
    // Only VERTICAL walls
    if ((prt[i].x_zone==0) || (prt[i].x_zone==(num_x_zones-1))){
      touch=TRUE;
      if (prt[i].x_zone==0){vx=param->Kv_inv[long(drand48()*nob)];}
      else {vx=-param->Kv_inv[long(drand48()*nob)];};
      vy=param->K_inv[1+long(drand48()*(nob-1))];
```

```
    };
#elif ((H_WALLS) && (V_WALLS))
    // Both HORIZONTAL and VERTICAL walls
    if ((prt[i].x_zone==0) || (prt[i].x_zone==(num_x_zones-1))
        || (prt[i].y_zone==0) || (prt[i].y_zone==(num_y_zones-1))){
      touch=TRUE;
      if (prt[i].y_zone==0){vy=param->Kv_inv[long(drand48()*nob)]; }
      else if (prt[i].y_zone==(num_y_zones-1)){vy=-param->Kv_inv[long(drand48()*nob)];}
      else {vy=param->K_inv[1+long(drand48()*(nob-1))];};

      if (prt[i].x_zone==0){vx=param->Kv_inv[long(drand48()*nob)];}
      else if (prt[i].x_zone==(num_x_zones-1)){vx=-param->Kv_inv[long(drand48()*nob)];}
      else {vx=param->K_inv[1+long(drand48()*(nob-1))];};
    };
#endif

#elif (WALL_TYPE==SCX)
#if (H_WALLS)
    printf("This has not been implemented!\n");
#endif
#if (V_WALLS)
    if ((prt[i].x_zone==0) || (prt[i].x_zone==(num_x_zones-1))){
      touch=TRUE;
      if (prt[i].x_zone==0){vx=param->Kv2_inv[long(drand48()*nob)];}
      else {vx=-param->Kv2_inv[long(drand48()*nob)];};
      vy=0;
    };
#endif

#elif (WALL_TYPE==STC)
#if (H_WALLS)
    printf("This has not been implemented!\n");
#endif
#if (V_WALLS)
    // Reemit particles with angles sampled from cos(theta)/2.
    FLOATTYPE theta=asin(2*drand48()-1);
    if ((prt[i].x_zone==0) || (prt[i].x_zone==(num_x_zones-1))){
      touch=TRUE;
      if (prt[i].x_zone==0){vx=v0*cos(theta);}
      else {vx=-v0*cos(theta);};
      vy=v0*sin(theta);
    };
#endif

#endif

    if (touch){
#if (REC_KIN_CHANGE && STOCH_WALLS)
      if (param->current_step%energy_step==0){
        param->E_kin_change+=(vx*vx+vy*vy)-(prt[i].ov.vx*prt[i].ov.vx+prt[i].ov.vy*prt[i].ov.vy);
      };
#endif
      prt[i].ov.vx=vx;
      prt[i].ov.vy=vy;
    };

#endif // STOCH_WALLS



    ////////////////////////////////////////
    // The main integration algorithm.    //
    ////////////////////////////////////////

    nv.vx=prt[i].ov.vx+dt*prt[i].a.ax;
    nv.vy=prt[i].ov.vy+dt*prt[i].a.ay;

    prt[i].r.x+=dt*nv.vx;
    prt[i].r.y+=dt*nv.vy;

    prt[i].v.vx=(nv.vx+prt[i].ov.vx)/2;
    prt[i].v.vy=(nv.vy+prt[i].ov.vy)/2;

    prt[i].ov=nv;

    // Don't multiply with mass, since m=1
    param->Px+=prt[i].v.vx;
    param->Py+=prt[i].v.vy;

#if (((WALL_TYPE==HR) || STOCH_WALLS) && FORCE_OUT)
    if (touch){
#if (V_WALLS)
      if (prt[i].r.x<=vertwall_grabdist){
        prt[i].r.x=vertwall_grabdist*1.000001;
```

```
      }
      else if (prt[i].r.x>=tubelength_x-vertwall_grabdist){
        prt[i].r.x=tubelength_x-vertwall_grabdist*1.000001;
      };
#endif
#if (H_WALLS)
      if (prt[i].r.y<=horiwall_grabdist){
        prt[i].r.y=horiwall_grabdist*1.000001;
      }
      else if (prt[i].r.y>=tubelength_y-horiwall_grabdist){
        prt[i].r.y=tubelength_y-horiwall_grabdist*1.000001;
      };
#endif
    };
#endif

#if (H_WALLS==FALSE)
    // Periodic boundary conditions in vertical direction
    if (prt[i].r.y<0) {
      prt[i].r.y+=tubelength_y;
      swap_box(&prt[i],&boxes[prt[i].boxnr],&boxes[prt[i].boxnr+no_of_boxes-no_of_x_boxes]);
    };
    if (prt[i].r.y>tubelength_y) {
      prt[i].r.y-=tubelength_y;
      swap_box(&prt[i],&boxes[prt[i].boxnr],&boxes[prt[i].boxnr-no_of_boxes+no_of_x_boxes]);
    };
#endif

#if (V_WALLS==FALSE)
    // Periodic boundary conditions in horizontal direction
    if (prt[i].r.x<0)        {
      prt[i].r.x+=tubelength_x;
      swap_box(&prt[i],&boxes[prt[i].boxnr],&boxes[prt[i].boxnr+no_of_x_boxes-1]);
    };
    if (prt[i].r.x>tubelength_x) {
      prt[i].r.x-=tubelength_x;
      swap_box(&prt[i],&boxes[prt[i].boxnr],&boxes[prt[i].boxnr-no_of_x_boxes+1]);
    };
#endif

    // Checks that no particle has escaped the tube.
    if ((prt[i].r.x<0)
        || (prt[i].r.x>=tubelength_x)
        || (prt[i].r.y<0)
        || (prt[i].r.y>=tubelength_y))        {
      printf("Particle %d outside tube\n",i);
      printf("prt[%d].r.x = %g, prt[%d].r.y = %g\n",i,prt[i].r.x,i,prt[i].r.y);
      printf("prt[%d].v.vx = %g, prt[%d].v.vy = %g\n",i,prt[i].v.vx,i,prt[i].v.vy);
      printf("prt[%d].a.ax = %g, prt[%d].a.ay = %g\n",i,prt[i].a.ax,i,prt[i].a.ay);
      printf("tubelength_x = %g, tubelength_y = %g\n",tubelength_x,tubelength_y);
      return(1); /* something is wrong, abort */
    }
    else {
      // Assuming here that the boxes are numbered from left to right, bottom to top.
      while (prt[i].r.x<boxes[prt[i].boxnr].bottomleft.x)
        {swap_box(&prt[i],&boxes[prt[i].boxnr],&boxes[prt[i].boxnr-1]);};
      while (prt[i].r.x>=boxes[prt[i].boxnr].topright.x)
        {swap_box(&prt[i],&boxes[prt[i].boxnr],&boxes[prt[i].boxnr+1]);};
      while (prt[i].r.y<boxes[prt[i].boxnr].bottomleft.y)
        {swap_box(&prt[i],&boxes[prt[i].boxnr],&boxes[prt[i].boxnr-no_of_x_boxes]);};
      while (prt[i].r.y>=boxes[prt[i].boxnr].topright.y)
        {swap_box(&prt[i],&boxes[prt[i].boxnr],&boxes[prt[i].boxnr+no_of_x_boxes]);};
    };
  };

#if (REC_HALFKINETIC)
  long left=0,right=0;
  for (i=0;i<no_of_particles;i++){
    if (prt[i].r.x>(tubelength_x/(double)2)) {
      param->E_kinetic_right+=prt[i].v.vx*prt[i].v.vx+prt[i].v.vy*prt[i].v.vy;
      right++;
    }
    else{
      param->E_kinetic_left+=prt[i].v.vx*prt[i].v.vx+prt[i].v.vy*prt[i].v.vy;
      left++;
    };
  };
  param->E_kinetic=(param->E_kinetic_left+param->E_kinetic_right)/FLOATTYPE(2*no_of_particles);
  param->E_kinetic_left/=FLOATTYPE(2*left);
  param->E_kinetic_right/=FLOATTYPE(2*right);
#else
  param->E_kinetic=0;
  for (i=0;i<no_of_particles;i++){
    param->E_kinetic+=prt[i].v.vx*prt[i].v.vx+prt[i].v.vy*prt[i].v.vy;
```

```
  };
  param->E_kinetic/=FLOATTYPE(2*no_of_particles);
#endif

  param->E_potential/=FLOATTYPE(no_of_particles);
  param->E_pot_part/=FLOATTYPE(no_of_particles);
#if (WALL_TYPE==LJ)
  param->E_pot_wall/=FLOATTYPE(no_of_particles);
#endif
#if (STOCH_WALLS && REC_KIN_CHANGE)
  param->E_kin_change/=FLOATTYPE(no_of_particles);
#endif
#if ((INIT_RESCALE_VEL) || (REG_RESCALE_VEL))
  param->E_kin_rescale+=param->E_kinetic;
  param->rescale_counter++;
#endif

  param->Px/=FLOATTYPE(no_of_particles);
  param->Py/=FLOATTYPE(no_of_particles);

  return(0); // Everything is OK
};


#if (ZONES)
void update_zones(class particle prt[], class parameters *param)
{
  long i,j;

  for (i=0;i<no_of_particles;i++){
#if (X_ZONES)
    prt[i].old_x_zone=prt[i].x_zone;
    j=0;
    while (prt[i].r.x>param->x_zone[j]){j++;};
    prt[i].x_zone=j;

#if (UNI_THETA && V_WALLS)
    if (j==0) {prt[i].touch_vwall=0;}
    else if (j==(num_x_zones-1)) {prt[i].touch_vwall=1;}
    else {prt[i].touch_vwall=-1;};
#endif
#endif
#if (Y_ZONES)
    prt[i].old_y_zone=prt[i].y_zone;
    j=0;
    while (prt[i].r.y>param->y_zone[j]){j++;};
    prt[i].y_zone=j;

#if (UNI_THETA && H_WALLS)
    if (j==0) {prt[i].touch_hwall=0;}
    else if (j==(num_y_zones-1)) {prt[i].touch_hwall=1;}
    else {prt[i].touch_hwall=-1;};
#endif
#endif
  };
};
#endif

#if (INIT_RESCALE_VEL)
void rescale_vel(class particle prt[], class parameters *param, FLOATTYPE new_temp)
{
  long i;

  // Current Temperature: (2/2)kT=(1/2)mv^2
  // factor=sqrt(new_temp/old_temp)
  // The k and m are left out, since they are one in reduced units.
  // This algorithm also removes momentum in the y-direction.
  // NB: This assumes that there are VERTICAL WALLS,
  // since the momentum is not removed in the x-direction.
  param->E_kin_rescale/=(FLOATTYPE)param->rescale_counter;
  FLOATTYPE factor=sqrt(new_temp/(param->E_kin_rescale));
  printf("Rescaling by a factor %g\n",factor);
  for (i=0;i<no_of_particles;i++){
    prt[i].ov.vx=factor*(prt[i].ov.vx);
    prt[i].ov.vy=factor*(prt[i].ov.vy-param->Py);
  };
  param->rescale_counter=0;
  param->E_kin_rescale=0;
};
#endif
```

B.1.11 MD_calc_inline.hh

```
/* MD_calc_inline.hh, inline functions for force calculations. */
```

```
/*****************************************************************************/
/* CALCULATE ACCELERATION BETWEEN PARTICLES                                  */
/*****************************************************************************/
inline void calculate_accel_from_particles(class particle & p1 , class particle & p2 , class parameters *param)
{
  // For each pair of particles, we check distance (r) to find force.
  FLOATTYPE dx,dy,r2;

  dx=p1.r.x - p2.r.x;
#if (V_WALLS==FALSE)
  {
    FLOATTYPE temp;
    if (dx<0){if ((temp=tubelength_x+dx)<-dx) {dx=temp;};}
    else {if ((temp=tubelength_x-dx)<dx) {dx=-temp;};};
  };
#endif

  dy=p1.r.y - p2.r.y;
#if (H_WALLS==FALSE)
  {
    FLOATTYPE temp;
    if (dy<0) {if ((temp=tubelength_y+dy)<-dy) {dy=temp;};}
    else {if ((temp=tubelength_y-dy)<dy) {dy=-temp;};};
  };
#endif

  r2=dx*dx + dy*dy;

  // U(r)=U_term*(1/r12-1/r6)
  // _F_(r)=(F_term/r2)*(2/r12-1/r6)*(x,y,z),  _F_ means F vector
  // Requires that sigma is 1

  if (r2<r2_cutoff){
    // The particles are in range
    FLOATTYPE r2_inv,r6_inv,r12_inv,main_factor,f_factor,fx,fy;

    r2_inv=1/r2;
    r6_inv=r2_inv*r2_inv*r2_inv;
    r12_inv=r6_inv*r6_inv;

#if (SHIFTED==NONE)
    main_factor=r12_inv-r6_inv;
    param->E_pot_part+=4*main_factor;
    f_factor=24*r2_inv*(main_factor+r12_inv);
#elif (SHIFTED==POT)
    main_factor=r12_inv-r6_inv;
    param->E_pot_part+=4*main_factor-u_cutoff;
    f_factor=24*r2_inv*(main_factor+r12_inv);
#elif (SHIFTED==FORCE)
    main_factor=r12_inv-r6_inv;
    FLOATTYPE r = sqrt(r2);
    param->E_pot_part+=4*main_factor-u_cutoff+(r-r_cutoff)*f_cutoff;
    f_factor=24*r2_inv*(main_factor+r12_inv)-f_cutoff/r;
#else
    printf("Something is wrong in preprocessor statement SHIFTED\n");
#endif


    // fx and fy are the forces on p1
    fx=f_factor*dx;
    fy=f_factor*dy;

    // Don't use mass since m=1
    p1.a.ax+=fx;
    p1.a.ay+=fy;

    p2.a.ax+=-fx;
    p2.a.ay+=-fy;
  };
  // If the particles are not in range, nothing is done.
};

#if ((WALL_TYPE==LJ) && (H_WALLS))
/*****************************************************************************/
/* CALCULATE ACCELERATION FROM HORIZONTAL WALLS                              */
/*****************************************************************************/
inline void calculate_accel_from_horizontal_walls(class particle & p ,
                                                  class horizontal_wall & w ,
                                                  class parameters *param)
{
  // The particle-wall potential is the same as between two particles
  FLOATTYPE dy,r2;
```

```
  dy=p.r.y - (w.y+w.adjust);
  r2=dy*dy;
  if (r2<r2_cutoff) {

    FLOATTYPE r2_inv,r6_inv,r12_inv,main_factor;

    r2_inv=1/r2;
    r6_inv=r2_inv*r2_inv*r2_inv;
    r12_inv=r6_inv*r6_inv;

#if (SHIFTED==NONE)
    main_factor=r12_inv-r6_inv;
    param->E_pot_wall+=4*main_factor;
    p.a.ay+=24*r2_inv*(main_factor+r12_inv)*dy;
#elif (SHIFTED==POT)
    main_factor=r12_inv-r6_inv;
    param->E_pot_wall+=4*main_factor-u_cutoff;
    p.a.ay+=24*r2_inv*(main_factor+r12_inv)*dy;
#elif (SHIFTED==FORCE)
    main_factor=r12_inv-r6_inv;
    FLOATTYPE r = sqrt(r2);
    param->E_pot_wall+=4*main_factor-u_cutoff+(r-r_cutoff)*f_cutoff;
    p.a.ay+=(24*r2_inv*(main_factor+r12_inv)-f_cutoff/r)*dy;
#else
    printf("Something is wrong in preprocessor statement SHIFTED\n");
#endif
  };
  // If the particle is not in range, nothing is done.
};
#endif


#if ((WALL_TYPE==LJ) && (V_WALLS))
/*****************************************************************************/
/* CALCULATE ACCELERATION FROM VERTICAL WALLS                                */
/*****************************************************************************/
inline void calculate_accel_from_vertical_walls(class particle & p ,
                                                class vertical_wall & w ,
                                                class parameters *param){
  // The particle-wall potential is the same as between two particles
  FLOATTYPE dx,r2;

  dx=p.r.x - (w.x+w.adjust);
  r2=dx*dx;
  if (r2<r2_cutoff) {

    FLOATTYPE r2_inv,r6_inv,r12_inv,main_factor;

    r2_inv=1/r2;
    r6_inv=r2_inv*r2_inv*r2_inv;
    r12_inv=r6_inv*r6_inv;

#if (SHIFTED==NONE)
    main_factor=r12_inv-r6_inv;
    param->E_pot_wall+=4*main_factor;
    p.a.ax+=24*r2_inv*(main_factor+r12_inv)*dx;
#elif (SHIFTED==POT)
    main_factor=r12_inv-r6_inv;
    param->E_pot_wall+=4*main_factor-u_cutoff;
    p.a.ax+=24*r2_inv*(main_factor+r12_inv)*dx;
#elif (SHIFTED==FORCE)
    main_factor=r12_inv-r6_inv;
    FLOATTYPE r = sqrt(r2);
    param->E_pot_wall+=4*main_factor-u_cutoff+(r-r_cutoff)*f_cutoff;
    p.a.ax+=(24*r2_inv*(main_factor+r12_inv)-f_cutoff/r)*dx;
#else
    printf("Something is wrong in preprocessor statement SHIFTED\n");
#endif
  };
  // If the particle is not in range, nothing is done.
};
#endif
```

B.1.12 MD_fileoutput.hh

```
/* MD_fileoutput.hh, header file for MD_fileoutput.c */

#ifndef FILEOUTPUT
#define FILEOUTPUT

/*****************************************************************/
/* FUNCTION: Writes values that are constant during simulation to       */
```

```
/*           the file specified.                                    */
/* INPUT: String with filename.                                    */
/*******************************************************************/
void write_constants_to_file(class parameters *, char *);

/*******************************************************************/
/* FUNCTION: Updates a ps file, making a cartoon of moving particles.   */
/* INPUT: Array of particles, string with filename.                */
/*******************************************************************/
void write_cartoon_to_file(FILE * , class particle *);

/*******************************************************************/
/* FUNCTION: Writes the coordinates of the boxes to file.          */
/* INPUT: Array of boxes, string with filename.                    */
/*******************************************************************/
void write_box_coords_to_file(class box * , char *);

/*******************************************************************/
/* FUNCTION: Writes the 'reaction' lists of the boxes to file.     */
/* INPUT: Array of boxes, string with filename.                    */
/*******************************************************************/
void write_box_connections_to_file(class box * , char *);

/*******************************************************************/
/* FUNCTION: Writes which particles are in which box.              */
/* INPUT: Array of boxes, string with filename.                    */
/*******************************************************************/
void write_particle_connections_to_file(class box * , char *);

/*******************************************************************/
/* FUNCTION: Makes two files, the first is the force graph,        */
/*           the second is the potential energy graph.             */
/* INPUT: String with filename (force), string with filename (potential)*/
/*******************************************************************/
void write_force_and_pot_graphs_to_file(class parameters * , char * , char *);

/*******************************************************************/
/* FUNCTION: Writes all the particles' positions and velocities.   */
/*           Values are in reduced units.                          */
/* INPUT: Array of particles, string with filename.                */
/*******************************************************************/
void write_frame_to_file(FILE * , class particle *);

#if (WALL_TYPE==SDD)
/*******************************************************************/
/* FUNCTION: Writes the tables K_inv and Kv_inv.                   */
/*           Values are in reduced units.                          */
/* INPUT: pointer to parameters, strings with filename.            */
/*******************************************************************/
void write_K_inv_to_file(class parameters *, char *);

/*******************************************************************/
/* FUNCTION: Writes the tables K_inv and Kv_inv.                   */
/*           Values are in reduced units.                          */
/* INPUT: pointer to parameters, string with filename.             */
/*******************************************************************/
void write_Kv_inv_to_file(class parameters *, char *);

/*******************************************************************/
/* FUNCTION: Makes distribution of v from K_inv and Kv_inv.        */
/*           Values are in reduced units.                          */
/* INPUT: pointer to parameters, string with filename.             */
/*******************************************************************/
void write_v_sample_to_file(class parameters *, char *);
#endif

#if (WALL_TYPE==SUD)
/*******************************************************************/
/* FUNCTION: Writes the tables K_inv and Kv_inv.                   */
/*           Values are in reduced units.                          */
/* INPUT: pointer to parameters, two strings with filenames.       */
/*******************************************************************/
void write_Kv2_inv_to_file(class parameters *, char *);
#endif

#endif
```

B.1.13 MD_fileoutput.cc

```
/* MD_fileoutput.cc, functions for writing information to file. */

#include <stdio.h>
#include "MD_definitions.hh"
#include "MD_constants.hh"
#include "MD_classes.hh"
#include "MD_calc.hh"
#include "MD_fileoutput.hh"


void write_constants_to_file(class parameters *param , char s[])
{
  // This procedure writes contents of constants to file.
  FILE *fp;

  if ((fp = fopen(s,"w")) != NULL){
    fprintf(fp,"Values that are constant during a simulation.\n");
    fprintf(fp,"---------------------------------------------\n");

    fprintf(fp,"\nDEFINITIONS:\n");
    fprintf(fp,"FALSE:             %d\n",FALSE);
    fprintf(fp,"TRUE:              %d\n",TRUE);
    fprintf(fp,"sizeof(FLOATTYPE):%d\n",sizeof(FLOATTYPE));

    fprintf(fp,"NONE:              %d\n",NONE);
    fprintf(fp,"POT:               %d\n",POT);
    fprintf(fp,"FORCE:             %d\n",FORCE);
    fprintf(fp,"SHIFTED:           %d\n",SHIFTED);

    fprintf(fp,"H_WALLS:           %d\n",H_WALLS);
    fprintf(fp,"V_WALLS:           %d\n",V_WALLS);
    fprintf(fp,"WALL_TYPE:         %d\n",WALL_TYPE);
    fprintf(fp,"UNI_THETA:         %d\n",UNI_THETA);
    fprintf(fp,"STOCH_WALLS:       %d\n",STOCH_WALLS);
    fprintf(fp,"TRIGGER_WALLS:     %d\n",TRIGGER_WALLS);
    fprintf(fp,"CONSTANT_FORCE:    %d\n",CONSTANT_FORCE);
    fprintf(fp,"INIT_RESCALE_VEL: %d\n",INIT_RESCALE_VEL);
    fprintf(fp,"REG_RESCALE_VEL:  %d\n",REG_RESCALE_VEL);
    fprintf(fp,"FORCE_OUT:         %d\n\n",FORCE_OUT);

    fprintf(fp,"DATA_PATH:      %s\n", DATA_PATH);
    fprintf(fp,"REC_HALFKINETIC %d\n", REC_HALFKINETIC);
    fprintf(fp,"REC_KINETIC:    %d\n", REC_KINETIC);
    fprintf(fp,"REC_POTENTIAL:  %d\n", REC_POTENTIAL);
    fprintf(fp,"REC_TOTAL:      %d\n", REC_TOTAL);
    fprintf(fp,"REC_H_FUNCTION: %d\n", REC_H_FUNCTION);
    fprintf(fp,"REC_POT_PART:   %d\n", REC_POT_PART);
    fprintf(fp,"REC_KIN_CHANGE: %d\n", REC_KIN_CHANGE);
    fprintf(fp,"REC_POT_WALL:   %d\n", REC_POT_WALL);
    fprintf(fp,"REC_MOMENTUM:   %d\n", REC_MOMENTUM);
    fprintf(fp,"REC_PX:         %d\n", REC_PX);
    fprintf(fp,"REC_PY:         %d\n", REC_PY);
    fprintf(fp,"REC_CARTOON:    %d\n", REC_CARTOON);
    fprintf(fp,"REC_CRASH_VALS: %d\n", REC_CRASH_VALS);
    fprintf(fp,"REC_FRAMES:     %d\n", REC_FRAMES);
    fprintf(fp,"REC_ENERGY:     %d\n", REC_ENERGY);
    fprintf(fp,"REC_ANY:        %d\n", REC_ANY);
    fprintf(fp,"MAKE_DISTRIBUTIONS: %d\n", MAKE_DISTRIBUTIONS);
    fprintf(fp,"RADIAL_DIST:    %d\n\n", RADIAL_DIST);

    fprintf(fp,"UNITS:\n");
    fprintf(fp,"Distance unit,        DU: %g\n",DU);
    fprintf(fp,"Time unit,            TU: %g\n",TU);
    fprintf(fp,"Mass unit,            MU: %g\n",MU);
    fprintf(fp,"Velocity unit,        VU: %g\n",VU);
    fprintf(fp,"Acceleration unit,    AU: %g\n",AU);
    fprintf(fp,"Force unit,           FU: %g\n",FU);
    fprintf(fp,"Energy unit,          EU: %g\n",EU);
    fprintf(fp,"Temperature unit,   TmpU: %g\n",TmpU);
    fprintf(fp,"Number density unit, NDU: %g\n",NDU);

    fprintf(fp,"\nCONSTANTS (the values below are in the above units):\n");
    fprintf(fp,"Pi:                     %g\n",Pi);

    fprintf(fp,"r_cutoff:    %g\n", r_cutoff);
    fprintf(fp,"r2_cutoff:   %g\n\n", r2_cutoff);

    fprintf(fp,"u_cutoff:    %g\n", u_cutoff);
    fprintf(fp,"f_cutoff:    %g\n\n", f_cutoff);

    fprintf(fp,"r_nopot:    %g\n",r_nopot);
    fprintf(fp,"r_neutral:  %g\n",r_neutral);
```

```
#if ((WALL_TYPE==SUD) || (WALL_TYPE==SDD))
    fprintf(fp,"wall_temperature: %g\n\n",wall_temperature);
#endif
#if ((WALL_TYPE==SCC) || (WALL_TYPE==SUC) || (WALL_TYPE==STC))
    fprintf(fp,"v0:               %g\n\n",v0);
#endif
#if (INIT_RESCALE_VEL)
    fprintf(fp,"init_temperature: %g\n\n",init_temperature);

    fprintf(fp,"init_rescale_interval: %d\n",init_rescale_interval);
    fprintf(fp,"init_rescale_stop:     %d\n",init_rescale_stop);
#endif
#if (REG_RESCALE_VEL)
    fprintf(fp,"reg_rescale_interval:  %d\n",reg_rescale_interval);
#endif

#if (REC_H_FUNCTION)
    fprintf(fp,"hfunc_start: %d\n",hfunc_start);
    fprintf(fp,"hfunc_step:  %d\n",hfunc_step);
    fprintf(fp,"hfunc_stop:  %d\n",hfunc_stop);
#endif

#if ((WALL_TYPE==SDD) || (WALL_TYPE==SUD) || (WALL_TYPE==SCX))
    fprintf(fp,"dv:         %g\n",dv);
#endif

#if (WALL_TYPE==SDD)
    fprintf(fp,"K_factor:   %g\n",K_factor);
    fprintf(fp,"Kv_factor:  %g\n",Kv_factor);
#endif
#if ((WALL_TYPE==SUD) || (WALL_TYPE==SCX))
    fprintf(fp,"Kv2_factor: %g\n",Kv2_factor);
#endif

#if (H_WALLS)
    fprintf(fp,"horiwall_grabdist: %g\n",horiwall_grabdist);
#endif
#if (V_WALLS)
    fprintf(fp,"vertwall_grabdist: %g\n",vertwall_grabdist);
#endif

    fprintf(fp,"\nnumber_density:               %g\n",number_density);
    fprintf(fp,"no_of_particles_pr_boxlength: %d\n",no_of_particles_pr_boxlength);
    fprintf(fp,"tube_area:                    %g\n",tube_area);
    fprintf(fp,"dist_between_part:            %g\n",dist_between_part);
    fprintf(fp,"boxlength:                    %g\n",boxlength);
    fprintf(fp,"dt:                           %g\n",dt);
    fprintf(fp,"dt2:                          %g\n\n",dt2);

    fprintf(fp,"no_of_boxes:       %d\n",no_of_boxes);
    fprintf(fp,"no_of_x_boxes:     %d\n",no_of_x_boxes);
    fprintf(fp,"no_of_y_boxes:     %d\n",no_of_y_boxes);
    fprintf(fp,"tubelength_x:      %g\n",tubelength_x);
    fprintf(fp,"tubelength_y:      %g\n",tubelength_y);
    fprintf(fp,"no_of_particles:   %d\n",no_of_particles);
    fprintf(fp,"no_of_x_particles: %d\n",no_of_x_particles);
    fprintf(fp,"no_of_y_particles: %d\n\n",no_of_y_particles);

#if (CONSTANT_FORCE)
    fprintf(fp,"constant_force_x: %g\n",constant_force_x);
    fprintf(fp,"constant_force_y: %g\n\n",constant_force_y);
#endif

#if (MAKE_DISTRIBUTIONS)
    fprintf(fp,"dist_bins:      %d\n",dist_bins);
    fprintf(fp,"prof_bins:      %d\n",prof_bins);
    fprintf(fp,"binwidth_v:     %g\n",binwidth_v);
    fprintf(fp,"binwidth_prof:  %g\n",binwidth_prof);
    fprintf(fp,"min_speed_v:    %g\n",min_speed_v);
    fprintf(fp,"max_speed_v:    %g\n\n",max_speed_v);
#endif

#if (RADIAL_DIST)
    fprintf(fp,"rad_start:      %d\n",rad_start);
    fprintf(fp,"rad_step:       %d\n",rad_step);
    fprintf(fp,"rad_stop:       %d\n",rad_stop);
    fprintf(fp,"rad_length:     %g\n",rad_length);
    fprintf(fp,"rad_bins:       %d\n",rad_bins);
    fprintf(fp,"binwidth_rad:   %g\n\n",binwidth_rad);
#endif

#if (REC_CRASH_VALS)
    fprintf(fp,"num_of_dists:        %g\n",num_of_dists);
```

```
    fprintf(fp,"upper_zones:         %g\n",upper_zones);
    fprintf(fp,"crash_bins:          %g\n",crash_bins);
    fprintf(fp,"min_crash_x:         %g\n",min_crash_x);
    fprintf(fp,"max_crash_x:         %g\n",max_crash_x);
    fprintf(fp,"min_crash_v:         %g\n",min_crash_v);
    fprintf(fp,"max_crash_v:         %g\n",max_crash_v);
    fprintf(fp,"min_crash_v2:        %g\n",min_crash_v2);
    fprintf(fp,"max_crash_v2:        %g\n",max_crash_v2);
    fprintf(fp,"min_crash_theta:     %g\n",min_crash_theta);
    fprintf(fp,"max_crash_theta:     %g\n",max_crash_theta);
    fprintf(fp,"min_crash_vx:        %g\n",min_crash_vx);
    fprintf(fp,"max_crash_vx:        %g\n",max_crash_vx);
    fprintf(fp,"min_crash_vy:        %g\n",min_crash_vy);
    fprintf(fp,"max_crash_vy:        %g\n",max_crash_vy);
    fprintf(fp,"theta_dist_binwidth: %g\n",theta_dist_binwidth);
    fprintf(fp,"v_dist_binwidth:     %g\n",v_dist_binwidth);
    fprintf(fp,"v2_dist_binwidth:    %g\n",v2_dist_binwidth);
    fprintf(fp,"vx_dist_binwidth:    %g\n",vx_dist_binwidth);
    fprintf(fp,"vy_dist_binwidth:    %g\n",vy_dist_binwidth);
#endif

#if (X_ZONES)
    fprintf(fp,"num_x_zones:       %d\n",num_x_zones);
#endif
#if (Y_ZONES)
    fprintf(fp,"num_y_zones:       %d\n",num_y_zones);
#endif
    fprintf(fp,"num_of_cycles:     %d\n",num_of_cycles);
#if (REC_FRAMES)
    fprintf(fp,"start_write_step:  %d\n",start_write_step);
    fprintf(fp,"data_write_step:   %d\n",data_write_step);
#endif
#if (REC_CRASH_VALS)
    fprintf(fp,"start_crash_write: %d\n",start_crash_write);
#endif
#if (REC_CARTOON)
    fprintf(fp,"cartoon_start:     %d\n",cartoon_start);
    fprintf(fp,"cartoon_stop:      %d\n",cartoon_stop);
    fprintf(fp,"cartoon_step:      %d\n",cartoon_step);
#endif
#if (REC_ENERGY)
    fprintf(fp,"energy_step:       %d\n\n",energy_step);
#endif
#if (REC_MOMENTUM)
    fprintf(fp,"momentum_step:       %d\n\n",momentum_step);
#endif
#if (MAKE_DISTRIBUTIONS)
    fprintf(fp,"dist_start:        %d\n",dist_start);
    fprintf(fp,"dist_step:         %d\n",dist_step);
#endif
    fprintf(fp,"\nCALCULATED VALUES:\n");
    fprintf(fp,"Potential energy per particle: %g\n",param->E_potential);


    fclose(fp);
  }
  else {
    printf("\nUnable to open file: %s !\n",s);
  };
};


void write_cartoon_to_file(FILE *fp , class particle prt[])
{
  // A PS-page is 600 wide.
  const FLOATTYPE max_screen_size=300; // Maximum length (in pixels on screen) of tubelength_y and _x
  const FLOATTYPE faktor=max_screen_size/max(tubelength_x,tubelength_y);

  fprintf(fp,"showpage\n");
  for (int i=0;i<no_of_particles;i++) {
    // "Plots" a dot.
    fprintf(fp,"%g %g c\n",prt[i].r.x*faktor,(prt[i].r.y)*faktor);
  };
};


void write_frame_to_file(FILE *fp , class particle prt[])
{
  // Outputs the particles' positions and velocities.
  int i;
  for (i=0;i<no_of_particles;i++){
    fprintf(fp,"%0.16g %0.16g %0.16g %0.16g\n",prt[i].r.x,prt[i].r.y,prt[i].v.vx,prt[i].v.vy);
  };
};
```

```
#if (WALL_TYPE==SDD)
void write_K_inv_to_file(class parameters *param, char s[])
{
  FILE *fp;
  int i;

  if ((fp = fopen(s,"w")) != NULL){
    for (i=0;i<nob;i++){fprintf(fp,"%d %0.16g\n",i,param->K_inv[i]);};
    fclose(fp);
  }
  else {printf("\nUnable to open file: %s !\n",s);};
};

void write_Kv_inv_to_file(class parameters *param, char s[])
{
  FILE *fp;
  int i;

  if ((fp = fopen(s,"w")) != NULL){
    for (i=0;i<nob;i++){fprintf(fp,"%d %0.16g\n",i,param->Kv_inv[i]);};
    fclose(fp);
  }
  else {printf("\nUnable to open file: %s !\n",s);};
};

#if (MAKE_DISTRIBUTION)
void write_v_sample_to_file(class parameters *param, char s[])
{
  FILE *fp;
  long i;

  FLOATTYPE v_sample[dist_bins];
  FLOATTYPE vx,vy,speed,rnd,fract;
  long speed_overflow=0,num_vpart=0;;

  for (i=0;i<3600;i++){
    v_sample[i]=0;
  };

  for (i=0;i<10000000;i++){
    vx=param->Kv_inv[long(drand48()*nob)];
    vy=param->K_inv[1+long(drand48()*(nob-1))];
    speed=sqrt(vx*vx+vy*vy);

    if ((speed>=min_speed_v) && (speed<max_speed_v)){
      v_sample[long((speed-min_speed_v)/binwidth_v)]++;
      num_vpart++;
    } else speed_overflow++;
  };

  for (i=0;i<dist_bins;i++){
    v_sample[i]/=(FLOATTYPE)num_vpart*binwidth_v;
  };

  printf("Speed overflows: %d\n",speed_overflow);

  if ((fp = fopen(s,"w")) != NULL){
    for (i=0;i<dist_bins;i++){
      fprintf(fp,"%.16g %.16g\n",min_speed_v+(0.5+i)*binwidth_v,v_sample[i]);
    };
    fclose(fp);
    printf("File successfully written to disk: %s\n",s);
  } else {printf("Unable to open file: %s\n",s);};
};
#endif //MAKE_DISTRIBUTIONS
#endif

#if (WALL_TYPE==SUD)
void write_Kv2_inv_to_file(class parameters *param, char s[])
{
  FILE *fp;
  int i;

  if ((fp = fopen(s,"w")) != NULL){
    for (i=0;i<nob;i++){fprintf(fp,"%d %0.16g\n",i,param->Kv2_inv[i]);};
    fclose(fp);
  }
  else {printf("\nUnable to open file: %s !\n",s);};
};
#endif


/*
```

```
// Cannot be used anymore after the force calculations
// were placed in an inline file.
void write_force_and_pot_graphs_to_file(class parameters *param, char s1[] , char s2[])
{
  FILE *fp1,*fp2;
  class particle p1,p2;

  if (((fp1 = fopen(s1,"w")) != NULL) && ((fp2 = fopen(s2,"w")) != NULL)) {
    p1.r.x=0;
    p1.r.y=0;

    p2.r.x=r_cutoff*1.1;
    p2.r.y=0;

    while (p2.r.x>0.5){
      p1.a.ax=0; p1.a.ay=0; p2.a.ax=0; p2.a.ay=0;
      param->E_pot_part=0;
      calculate_accel_from_particles(p1,p2,param);
      // Don't multiply accel. with mass, since m=1
      fprintf(fp1,"%g %g\n",p2.r.x,p2.a.ax);
      fprintf(fp2,"%g %g\n",p2.r.x,param->E_pot_part);
      p2.r.x-=0.0001;
    }
    fclose(fp1);
    fclose(fp2);
  }
  else {printf("\nUnable to open files: %s or %s !\n",s1,s2);};
};
*/

void write_box_coords_to_file(class box boxes[] , char s[])
{
  FILE *fp;

  if ((fp = fopen(s,"w")) != NULL){
    for (int i=0;i<no_of_boxes;i++){
      fprintf(fp,"boxnr. %d: %g %g %g %g \n",
              i,boxes[i].bottomleft.x,boxes[i].bottomleft.y,boxes[i].topright.x,boxes[i].topright.y);
    };
    fclose(fp);
  }
  else {printf("\nUnable to open file: %s !\n",s);};
};

void write_box_connections_to_file(class box boxes[] , char s[])
{
  FILE *fp;
  class boxlist *b;

  if ((fp = fopen(s,"w")) != NULL) {
    for (int i=0;i<no_of_boxes;i++){
      b=boxes[i].first_box;
      fprintf(fp,"%d:\n",i);
      while (b!=NULL) {
        fprintf(fp,"%d\n",b->boxnr);
        b=b->next_box;
      };
      fprintf(fp,"\n");
    };
    fclose(fp);
  }
  else {printf("\nUnable to open file: %s !\n",s);};
};

void write_particle_connections_to_file(class box boxes[], char s[])
{
  FILE *fp;
  class particle *p;

  if ((fp = fopen(s,"w")) != NULL) {
    for (int i=0;i<no_of_boxes;i++){
      fprintf(fp,"Box number %d:\n",i);
      p=boxes[i].first_particle;
      while (p!=NULL){
        fprintf(fp,"%d: %g %g\n",p->partnr,p->r.x,p->r.y);
        p=p->next_particle;
      };
      fprintf(fp,"\n");
    };
    fclose(fp);
  }
  else {printf("\nUnable to open file: %s !\n",s);};
};
```

B.1.14 MD_listtools.hh

```
/* MD_listtools.hh, header file for MD_listtools.cc */
#ifndef LISTTOOLS
#define LISTTOOLS

/*************************************************************************/
/* FUNCTION: Puts a particle in a box at beginning of list.              */
/* INPUT: Pointer to a particle , pointer to a box.                      */
/*************************************************************************/
void put_particle_in_box(class particle * , class box *);

/*************************************************************************/
/* FUNCTION: Moves particle from oldbox to newbox.                       */
/* INPUT: Pointer to a particle , pointer to a oldbox and newbox.        */
/*************************************************************************/
void swap_box(class particle *p , class box *oldbox , class box *newbox);

/*************************************************************************/
/* FUNCTION: Adds a box to the 'reaction' boxlist of another box         */
/*           (at beginning of list).                                     */
/* INPUT: Pointer to box with list, number of box to be inserted.        */
/*************************************************************************/
void add_box_to_boxlist(class box * , long);

#endif
```

B.1.15 MD_listtools.cc

```
/* MD_listtools.cc, tools for manipulating pointer lists */

#include <stdio.h>
#include <stdlib.h>
#include "MD_definitions.hh"
#include "MD_classes.hh"
#include "MD_listtools.hh"

void put_particle_in_box(class particle *p , class box *b)
{
  p->prev_particle=NULL;

  if (b->first_particle==NULL){
    p->next_particle=NULL;
  }
  else {
    p->next_particle=b->first_particle;
    p->next_particle->prev_particle=p;
  };
  b->first_particle=p;
  p->boxnr=b->boxnr;
};

void swap_box(class particle *p , class box *oldbox , class box *newbox)
{
  if (p->prev_particle==NULL) {
    oldbox->first_particle=p->next_particle;
  }
  else {
    p->prev_particle->next_particle=p->next_particle;
  };

  if (p->next_particle!=NULL) {p->next_particle->prev_particle=p->prev_particle;};

  put_particle_in_box(p,newbox);
};


void add_box_to_boxlist(class box *b1 , long b2)
{
  boxlist *b;
  b=new boxlist;
  b->boxnr=b2;

  b->next_box=b1->first_box;
  b1->first_box=b;
};
```

B.2 Additional Programs

A couple of programs were written to handle output from the main simulation. Here follows a list with a brief description of each. The program listings are included afterwards.

norm_pp.cc: Coarse-grains density profiles to the desired number of bins, and rescales the y-values to the real number density ρ^* by using the tube area. The profiles and distributions from the main simulation had 3600 bins.

avg_bins.cc: Same as above, but does not the rescale y-values. Used for the temperature profiles.

add_bins.cc: Same as above, but adds y-values instead of taking the average.

avg_dist_bins.cc: Same as above, and used for distributions. They are normalized after the coarse-graining, making the area under the curve equal to unity.

timecorr.cc calculates the autocorrelation function for a given dataset. It made the two lower graphs in Figure 6.2.

flyv.cc: estimates the standard error of the mean by iteratively coarse-graining a dataset, as presented by Flyvbjerg & Petersen (1989). Figures 6.3, 6.4, 6.5 and 6.6 were made from this program.

sample_dist.cc: made the datasets used in Figure 6.6.

B.2.1 norm_pp.cc

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
  if (argc!=5){
    printf("Reduces # bins by averaging, and at the same time calculates the number density.\n");
    printf("Usage: npp <filename> <tube height> <# ensembles> <reduction factor>\n");
  }
  else {
    char infilename[256];
    char outfilename[256];
    long red_factor;
    FILE *fpin,*fpout;

    double x,y,avg_x,avg_y,tube_height;
    double delta_x;
    long count,frames,num_bins=0;

    sscanf(argv[1],"%s",infilename);
    sscanf(argv[2],"%lg",&tube_height);
    sscanf(argv[3],"%d",&frames);
    sscanf(argv[4],"%d",&red_factor);
    sprintf(outfilename,"%s%c%d",infilename,'.',red_factor);

    if ((fpin=fopen(infilename,"r"))==NULL) {
      printf("Unable to open file for read: %s\n",infilename);
      exit(1);
    };
```

```
    if ((fpout=fopen(outfilename,"w"))==NULL) {
      printf("Unable to open file for write: %s\n",outfilename);
      exit(1);
    };



    printf("Reading file %s, writing to file %s, using a reduction factor of %d and a tubeheight of %g.\n",
           infilename,outfilename,red_factor,tube_height);

    fscanf(fpin,"%lg %lg",&x,&y);
    delta_x=x;
    fscanf(fpin,"%lg %lg",&x,&y);
    delta_x=x-delta_x;
    rewind(fpin);

    avg_x=0; avg_y=0; count=0;
    while (fscanf(fpin,"%lg %lg",&x,&y)!=EOF){
      count++;
      num_bins++;
      avg_x+=x;
      avg_y+=y;
      if (count==red_factor){
        avg_x/=double(red_factor);
        avg_y/=delta_x*double(count)*tube_height*double(frames);
        fprintf(fpout,"%0.16g %0.16g\n",avg_x,avg_y);
        count=0;
        avg_x=0;
        avg_y=0;
      };
    };

    if (count!=0) {
      printf("Warning: Last bin in outputfile was NOT averaged/added from %d bins in inputfile.\n",
             red_factor);
      avg_x/=double(count);
      avg_y/=delta_x*double(count)*tube_height*double(frames);
      fprintf(fpout,"%0.16g %0.16g\n",avg_x,avg_y);
    };

    printf("Total number of bins: %d\n",num_bins);
    fclose(fpin);
    fclose(fpout);
  };
};
```

B.2.2 avg_bins.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void main(int argc, char *argv[])
{
  if (argc!=4){
    printf("Reduces # bins by averaging y components.\n");
    printf("Each line in input file must consist of x avg 2.moment #particles.\n");
    printf("Usage: avb <filename> <# timesteps between indep. ensembles> <reduction factor>\n");
  }
  else {
    char infilename[256];
    char outfilename[256];
    long red_factor,indep;
    FILE *fpin,*fpout;

    double x,m1,m2,n,avg_x,sum_m1,sum_m2,sum_n,avg_m1;
    double s_div_sqrt_n; // Here 'n' is number of independent ensembles.
    long count,num_bins=0;

    sscanf(argv[1],"%s",infilename);
    sscanf(argv[2],"%d",&indep);
    sscanf(argv[3],"%d",&red_factor);
    sprintf(outfilename,"%s%c%d",infilename,'.',red_factor);

    if ((fpin=fopen(infilename,"r"))==NULL) {
      printf("Unable to open file for read: %s\n",infilename);
      exit(1);
    };

    if ((fpout=fopen(outfilename,"w"))==NULL) {
      printf("Unable to open file for write: %s\n",outfilename);
      exit(1);
    };
```

```
    printf("Reading file %s, writing to file %s.\n");
    printf("Using a reduction factor of %d.\n Assuming %d timesteps between independent ensembles.\n",
           infilename,outfilename,red_factor,indep);

    avg_x=0; sum_m1=0; sum_m2=0; sum_n=0; count=0;
    while (fscanf(fpin,"%lg %lg %lg %lg",&x,&m1,&m2,&n)!=EOF){
      count++;
      num_bins++;
      avg_x+=x;
      sum_m1+=n*m1;
      sum_m2+=n*m2;
      sum_n+=n;
      if (count==red_factor){
        avg_x/=double(count);
        if (sum_n!=0){
          avg_m1=sum_m1/sum_n;
          s_div_sqrt_n=sqrt((sum_m2*sum_n-sum_m1*sum_m1)/(sum_n*(sum_n-1)*(sum_n/(double)indep)));
        }
        else {
          avg_m1=0;
          s_div_sqrt_n=0;
        };
        fprintf(fpout,"%.16g %.16g %.16g\n",avg_x,avg_m1,s_div_sqrt_n);
        count=0;
        avg_x=0;
        sum_m1=0;
        sum_m2=0;
        sum_n=0;
      };
    };

    if (count!=0) {
      printf("Warning: Last bin in outputfile was NOT averaged from %d in inputfile bins.\n",red_factor);
      avg_x/=double(count);
      if (sum_n!=0){
        avg_m1=sum_m1/sum_n;
        s_div_sqrt_n=sqrt((sum_m2*sum_n-sum_m1*sum_m1)/(sum_n*(sum_n-1)*(sum_n/(double)indep)));
      }
      else {
        avg_m1=0;
        s_div_sqrt_n=0;
      };
      fprintf(fpout,"%.16g %.16g %.16g\n",avg_x,avg_m1,s_div_sqrt_n);
    };

    printf("Total number of bins: %d\n",num_bins);
    fclose(fpin);
    fclose(fpout);
  };
};
```

B.2.3 add_bins.cc

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
  if ((argc>3) || (argc<2)){
    printf("Reduces # bins by adding the y components.\n");
    printf("Usage: adb <filename> <reduction factor>\n");
  }
  else {
    char infilename[256];
    char outfilename[256];
    long red_factor;
    FILE *fpin,*fpout;

    double x,y,avg_x,add_y;
    long count,num_bins=0;

    sscanf(argv[1],"%s",infilename);
    sscanf(argv[2],"%d",&red_factor);
    sprintf(outfilename,"%s%c%d",infilename,'.',red_factor);

    if ((fpin=fopen(infilename,"r"))==NULL) {
      printf("Unable to open file for read: %s\n",infilename);
      exit(1);
    };
```

```
    if ((fpout=fopen(outfilename,"w"))==NULL) {
      printf("Unable to open file for write: %s\n",outfilename);
      exit(1);
    };



    printf("Reading file %s, writing to file %s, using a reduction factor of %d ...\n",
           infilename,outfilename,red_factor);

    avg_x=0; add_y=0; count=0;
    while (fscanf(fpin,"%lg %lg",&x,&y)!=EOF){
      count++;
      num_bins++;
      avg_x+=x;
      add_y+=y;
      if (count==red_factor){
        avg_x/=double(red_factor);
        fprintf(fpout,"%0.16g %0.16g\n",avg_x,add_y);
        count=0;
        avg_x=0;
        add_y=0;
      };
    };

    if (count!=0) {
      printf("Warning: Last bin in outputfile was NOT averaged/added from %d bins in inputfile.\n",
             red_factor);
      avg_x/=double(count);
      fprintf(fpout,"%0.16g %0.16g\n",avg_x,add_y);
    };

    printf("Total number of bins: %d\n",num_bins);
    fclose(fpin);
    fclose(fpout);
  };
};
```

B.2.4 avg_dist_bins.cc

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
  if ((argc>3) || (argc<2)){
    printf("Reduces # bins by averaging y components in several bins.\n");
    printf("Used for distributions. Normalisation conserved.\n");
    printf("Usage: avdb <filename> <reduction factor>\n");
  }
  else {
    char infilename[256];
    char outfilename[256];
    long red_factor;
    FILE *fpin,*fpout;

    double x,y,avg_x,avg_y;
    long count,num_bins=0;

    sscanf(argv[1],"%s",infilename);
    sscanf(argv[2],"%d",&red_factor);
    sprintf(outfilename,"%s%c%d",infilename,'.',red_factor);

    if ((fpin=fopen(infilename,"r"))==NULL) {
      printf("Unable to open file for read: %s\n",infilename);
      exit(1);
    };

    if ((fpout=fopen(outfilename,"w"))==NULL) {
      printf("Unable to open file for write: %s\n",outfilename);
      exit(1);
    };



    printf("Reading file %s, writing to file %s, using a reduction factor of %d ...\n",
           infilename,outfilename,red_factor);

    avg_x=0; avg_y=0; count=0;
    while (fscanf(fpin,"%lg %lg",&x,&y)!=EOF){
      count++;
      num_bins++;
      avg_x+=x;
```

```
      avg_y+=y;
      if (count==red_factor){
        avg_x/=double(red_factor);
        avg_y/=double(red_factor);
        fprintf(fpout,"%0.16g %0.16g\n",avg_x,avg_y);
        count=0;
        avg_x=0;
        avg_y=0;
      };
    };

    if (count!=0) {
      printf("Warning: Last bin in outputfile was NOT averaged/added from %d bins in inputfile.\n",
             red_factor);
      avg_x/=double(count);
      avg_y/=double(count);
      fprintf(fpout,"%0.16g %0.16g\n",avg_x,avg_y);
    };

    printf("Total number of bins: %d\n",num_bins);
    fclose(fpin);
    fclose(fpout);
  };
};
```

B.2.5 timecorr.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void main(int argc, char *argv[])
{
  if (argc!=4){
    printf("Calculates the time correlation function.\n");
    printf("Usage: tc <filename> <firstbin> <corrbins>\n");
    printf("firstbin=0 reads the whole file.\n");
  }
  else {
    char infilename[256];
    char outfilename[256];

    FILE *fpin,*fpout;

    double time,energy,energy_avg;
    long firstbin,count,i;

    long corrbins,t;


    sscanf(argv[1],"%s",infilename);
    sscanf(argv[2],"%d",&firstbin);
    sscanf(argv[3],"%d",&corrbins);

    double *C = new double[corrbins];

    if ((fpin=fopen(infilename,"r"))==NULL) {
      printf("Unable to open file for read: %s\n",infilename);
      exit(1);
    };

    printf("Reading file %s.\n",infilename);
    printf("First bin: %d\n",firstbin);

    count=0;
    while (fscanf(fpin,"%lg %lg",&time,&energy)!=EOF){count++;};
    printf("Total number of bins in file: %d\n",count);
    double *timearr = new double[count];
    double *energyarr = new double[count];
    rewind(fpin);
    count=0;
    while ((fscanf(fpin,"%lg %lg",&time,&energy)!=EOF) && (count<firstbin)){count++;};
    count=0;
    timearr[count]=time;
    energyarr[count]=energy;
    while (fscanf(fpin,"%lg %lg",&time,&energy)!=EOF){
      count++;
      timearr[count]=time;
      energyarr[count]=energy;
    };
    count++;
    printf("Read %d bins into memory.\n",count);
```

```
    printf("First set of data: time=%g   energy=%g\n",timearr[0],energyarr[0]);
    printf("Last set of data: time=%g   energy=%g\n",timearr[count-1],energyarr[count-1]);

    energy_avg=0;
    for (i=0;i<count;i++){
      energy_avg+=energyarr[i];
    };
    energy_avg/=count;

    printf("Energy average: %g\n",energy_avg);
    printf("\n");
    printf("Calculating Correlation Function\n");

    for (t=0;t<corrbins;t++){C[t]=0;};

    printf("Calculating C[t] for t=");
    for (t=0;t<corrbins;t++){
      if (t%100==0){printf("%d\n",t);};
      for (i=0;i<(count-corrbins);i++){
        C[t]+=(energyarr[i]-energy_avg)*(energyarr[i+t]-energy_avg);
      };
    };

    for (t=1;t<corrbins;t++){C[t]/=C[0];};
    C[0]=1;

    sprintf(outfilename,"timecorr.dat");
    if ((fpout=fopen(outfilename,"w"))==NULL) {
      printf("Unable to open file for write: %s\n",outfilename);
      exit(1);
    };

    for (t=0;t<corrbins;t++){
      fprintf(fpout,"%d %lg\n",t,C[t]);
    };

    fclose(fpout);

    delete [] timearr;
    delete [] energyarr;
    delete [] C;
  };
}
```

B.2.6 flyv.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void main(int argc, char *argv[])
{
  if (argc!=3){
    printf("Gives an estimate on the standard deviations\n using the method ");
    printf("presented in Flyvbjerg et al.s article\n");
    printf("Usage: flyv <filename> <firstbin>\n");
    printf("firstbin=0 reads the whole file.\n");
  }
  else {
    char infilename[256];
    char outfilename[256];

    FILE *fpin,*fpout;

    double x,y,dt;
    long firstbin,count,i;

    sscanf(argv[1],"%s",infilename);
    sscanf(argv[2],"%d",&firstbin);

    if ((fpin=fopen(infilename,"r"))==NULL) {
      printf("Unable to open file for read: %s\n",infilename);
      exit(1);
    };

    printf("Reading file %s.\n",infilename);
    printf("First bin: %d\n",firstbin);

    count=0;
    while (fscanf(fpin,"%lg %lg",&x,&y)!=EOF){count++;};
    printf("Total number of bins in file: %d\n",count);
    if (count==0) {
      printf("Error: File is empty!\n");
```

```
      exit(1);
    };
    if (count<=firstbin) {
      printf("Error: Firstbin exceeds number of bins in file.\n");
      exit(1);
    }
    else {
      count-=firstbin;
      printf("Loading %d bins.\n",count);
    };
    double *x_arr = new double[count];
    double *y_arr = new double[count];
    rewind(fpin);
    count=0;
    while ((fscanf(fpin,"%lg %lg",&x,&y)!=EOF) && (count<firstbin)){count++;};
    count=0;
    x_arr[count]=x;
    y_arr[count]=y;
    while (fscanf(fpin,"%lg %lg",&x,&y)!=EOF){
      count++;
      x_arr[count]=x;
      y_arr[count]=y;
    };
    count++;
    printf("Read %d bins into memory.\n",count);

    dt=x_arr[1]-x_arr[0];
    printf("Timestep: %g\n",dt);

    sprintf(outfilename,"flyv.dat");
    if ((fpout=fopen(outfilename,"w"))==NULL) {
      printf("Unable to open file for write: %s\n",outfilename);
      exit(1);
    };

    printf("Doing calculations.\n");
    double m,s,se;
    long coarsegrain=0;
    while (count>=2){
      m=0;
      for (i=0;i<count;i++){
        m+=y_arr[i];
      };
      m/=(double)count;

      s=0;
      for (i=0;i<count;i++){
        s+=(y_arr[i]-m)*(y_arr[i]-m);
      };
      s/=(double)(count-1)*(double)count;
      s=sqrt(s);

      se=s/sqrt((double)(2*(count-1)));

      dt=x_arr[1]-x_arr[0];

      printf("count=%10d  dt=%10g  m=%10g  s=%10g  se=%10g\n",count,dt,m,s,se);
      fprintf(fpout,"%.16g %.16g %.16g\n",dt,s,se);

      //Coarse Graining:
      count/=2;
      coarsegrain++;
      for (i=0;i<count;i++){
        x_arr[i]=(x_arr[2*i]+x_arr[2*i+1])/(double)2;
        y_arr[i]=(y_arr[2*i]+y_arr[2*i+1])/(double)2;
      };
    };

    printf("\n");

    fclose(fpout);

    delete [] y_arr;
    delete [] x_arr;
  };
}
```

B.2.7 sample_dist.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
```

```
void main(int argc, char *argv[])
{
  if (argc!=4){
    printf("Makes 10000 random values with drift\n");
    printf("Usage: sdi <drift> <length> <range>\n");
    printf("1/<drift> is maximum drift per step\n");
    printf("<drift>=0 means no drift\n");
    printf("<length> is number of values in sample\n");
    printf("<range> is width of short range fluctuations\n");
  }
  else {
    long drift,length;
    double range;
    char outfilename[256];

    sscanf(argv[1],"%d",&drift);
    sscanf(argv[2],"%d",&length);
    sscanf(argv[3],"%lg",&range);
    sprintf(outfilename,"%s%d%c%d%s","samples",drift,'_',length,".dat");

    long ttt;
    double d,a;
    FILE *fp;

    if ((fp = fopen(outfilename,"w")) != NULL){
      srand48(time(&ttt));
      printf("time:%d\n",ttt);

      a=-range*0.5;


      if (drift==0) {
        for (long i=0;i<length;i++){
          d=range*(drand48()-0.5);
          fprintf(fp,"%d %.16g\n",i,d);
        };
      }
      else {
        for (long i=0;i<length;i++){
          d=range*drand48();
          a+=(drand48()-0.5)/(double)drift;
          fprintf(fp,"%d %.16g\n",i,d+a);
        };
      };
      fclose(fp);
    }
    else{
      printf("Unable to open file!\n");
    };
  };
};
```