This is to certify that the

dissertation entitled

Computational Methods to Simulate Large Classical Particle Systems with Applications to Fluids and Microporous Materials

presented by

TIBOR F. NAGY

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in PHYSICS

S.D. Mahanti

Major professor
S.D. Mahanti

Date 07/14/2000

COMPUTATIONAL METHODS TO SIMULATE LARGE CLASSICAL PARTICLE SYSTEMS WITH APPLICATIONS TO FLUIDS AND MICROPOROUS MATERIALS

By

Tibor F. Nagy

A DISSERTATION

Submitted to

Michigan State University

in partial fulfillment of the requirements

for the Degree of

DOCTOR OF PHILOSOPHY

Department of Physics and Astronomy

2000

ABSTRACT

COMPUTATIONAL METHODS TO SIMULATE LARGE CLASSICAL PARTICLE SYSTEMS WITH APPLICATIONS TO FLUIDS AND MICROPOROUS MATERIALS

By

Tibor F. Nagy

A completely discrete – discrete space, time and energy – computational model and its implementation is presented here as an alternative to continuous type simulations such as Molecular Dynamics and continuum Monte Carlo methods. The final goal is to reduce the amount of the calculation per particle in order to increase the timescale of the simulation and the size of the particle system simulated. The computation is performed on a lattice with discrete particles in discrete states.

The fundamental particle simulation algorithms are studied. The well known neighbor-list or Verlet-list method is speeded up from quadratic to linear size dependence. This algorithm is applicable to continuous simulations also, it is lattice independent. Then the cell-list method is adapted to discrete lattices, using efficient bit manipulations. A field-representation method is developed as well as a generalization of the Lattice-Gas method.

The discrete computational techniques and other programming methods for today's advanced computer architectures are discussed in detail with their implementations in Fortran. The high performance of the whole method is mainly due to these techniques. Performance data are presented for the different methods.

The pair distribution function and density profiles inside parallel slits are calculated for different fluid densities. The lattice effects are successfully removed from the pair distribution function. The results agree well with continuum results for large slit widths. New results on layering for very narrow slit widths are presented.

Finally a lattice-based geometrical method is presented to characterize the structure of void space inside zeolites. This method is generic, it can be applied to arbitrary atomic and geometrical systems as well.

"The purpose of computing is insight, not numbers."
(R.W. Hamming)

Acknowledgements

I would like to express my gratitude to my thesis advisor, S. D. Mahanti for his help, guidance, unlimited patience and financial support. I would like to thank D. Tománek for his supervision and support as well.

I also thank my thesis committee members, Jerzy Borysowicz, James L. Dye, Julius S. Kovacs and Gerald Pollack for their help during my studies and for their careful reading of the manuscript.

Special thanks are given to Phillip Duxbury, Michael Thorpe, Raymond Brock, James Linnemann, Thomas J. Pinnavaia for letting me use their own or their departmental computers. I have used them a lot.

I thank everybody at the Physics Department: the professors, the secretaries, the computational staff and my officemates. It was very pleasant to be a graduate student here. I would like to thank the students of my Computational Physics class where, as an experiment, I covered some of the topics presented here. I have learned a lot by teaching these methods.

Last but not least I thank my families – both the American and the Hungarian – and my wife Stephanie for their continuous love and support.

The financial support from the Physics Department, the College of Natural Sciences, the National Science Foundation and the Center for Fundamental Materials Research is also acknowledged.

Contents

List of Figures

List of Tables

Chapter 1

Overview

In this chapter we will briefly review currently existing computational techniques to simulate the behavior of a given collection of classical particles. These techniques cover a broad scale from the microscopic atomic to the macroscopic continuum level depending on how the ensembles of particles or the physical substances are treated in the problem. At the microscopic scale the two most important methods - Molecular Dynamics (MD) and Monte Carlo (MC) - will be briefly discussed. At the macroscopic end Computational Fluid Dynamics (CFD) will be mentioned. This field itself represents a very large area in science and engineering, its literature fills up sections in libraries. It is of course impossible to describe it in a subsection of a thesis, its fundamental methodology will be mentioned just to orient ourselves among the different methods. The Lattice-Gas (LG) method will be then described as a particle based method giving correct macroscopic behavior very successfully and efficiently.

The gap between the micro and macro world is still wide and it is still waiting to be covered by new methods and techniques. Recently the expressions "bridging the landscapes", "mesoscopic simulators" can be very often heard at scientific meetings. The power of computers increases constantly every year. Unfortunately our desire to reach larger simulation sizes on longer time scales increases as well, keeping our

computers constantly busy and leaving plenty of room to develop new and faster simulation methods.

1.1 Molecular Dynamics

The Molecular Dynamics method was developed and successfully used already in the very early days of electronic computers [1], and it is still one of the most fundamental techniques to simulate particle systems ranging from the atomic level in condensed matter physics to the galactic scale in astronomy. It is basically the numerical solution of the classical equation of motion, Newton's second law:

$$m_i \frac{d^2\vec{r}_i}{dt^2} = \vec{F}_i,$$

where m_i is the mass of the ith particle, $\vec{r}_i$ is the position and $\vec{F}_i$ is the force acting on the particle caused by the other particles of the ensemble and possible external fields. This force must be known by the simulator in order to proceed with the iteration from moment to moment. If the number of particles is N, this equation of motion represents $3N$ coupled, ordinary, second order differential equations. The analytical solution of this problem is not feasible for large N, where large means anything greater than two. The numerical method to solve these differential equations is the *finite difference* method: the time is discretized, it proceeds in increments of δt. If the positions, velocities, accelerations and other information describing the system are known at a given moment of time t, then these quantities can be approximated at a later time $(t + \delta t)$ by their Taylor expansions around time t:

$$\begin{aligned}
\vec{r}(t+\delta t) &= \vec{r}(t) + \delta t\vec{v}(t) + \frac{1}{2}\delta t^2\vec{a}(t) + \frac{1}{6}\delta t^3\vec{b}(t) + \cdots \\
\vec{v}(t+\delta t) &= \vec{v}(t) + \delta t\vec{a}(t) + \frac{1}{2}\delta t^2\vec{b}(t) + \cdots \\
\vec{a}(t+\delta t) &= \vec{a}(t) + \delta t\vec{b}(t) + \cdots \\
\vec{b}(t+\delta t) &= \vec{b}(t) + \cdots
\end{aligned}$$

to a certain degree of accuracy. This accuracy is controlled by the size of the time interval δt, the number of terms used in the expansion and the integration-scheme which generates the quantities of the future from the quantities of the present and the past. The quantity $\vec{b}$ in the expansion above denotes the third time derivative of the position. The Taylor-expansion does not necessarily have to go this far nor does it need to stop here. Different integration schemes might use different numbers of terms from the expansions, or might combine the same terms differently.

One of the simplest and probably the most widely used scheme was developed by Verlet [2]. Let us take the *future* and the *past* Taylor expansions of the positions keeping the first three terms:

$$\begin{aligned} \vec{r}(t+\delta t) &= \vec{r}(t) + \delta t\vec{v}(t) + \frac{1}{2}\delta t^2\vec{a}(t) + \cdots, \\ \vec{r}(t-\delta t) &= \vec{r}(t) - \delta t\vec{v}(t) + \frac{1}{2}\delta t^2\vec{a}(t) + \cdots. \end{aligned}$$

By adding together these two equations, after rearranging we obtain:

$$\vec{r}(t+\delta t) = 2\vec{r}(t) - \vec{r}(t-\delta t) + \delta t^2\vec{a}(t). \tag{1.1}$$

As we can see, the future position is calulated in this scheme based on the present and the past position and the present acceleration. When simulating a particle ensemble on the computer this calculation is performed for each particle. Then the particles are moved to their new position. At the new positions the forces acting on the particles are calculated with the knowledge of the particle-particle force function and the external field, if that is present. From the forces the accelerations at these new positions are determined by $\vec{a}_i = \vec{F}_i/m_i$. Now we have all the information to take a new step in time, the iteration cycle is therefore completed. The velocity is not present in the Verlet-scheme, as we can see. If velocity dependent quantities are needed to be calculated – such as kinetic energy –, then the velocities can be obtained by:

$$\vec{v}(t) = \frac{\vec{r}(t+\delta t) - \vec{r}(t-\delta t)}{2\delta t}. \tag{1.2}$$

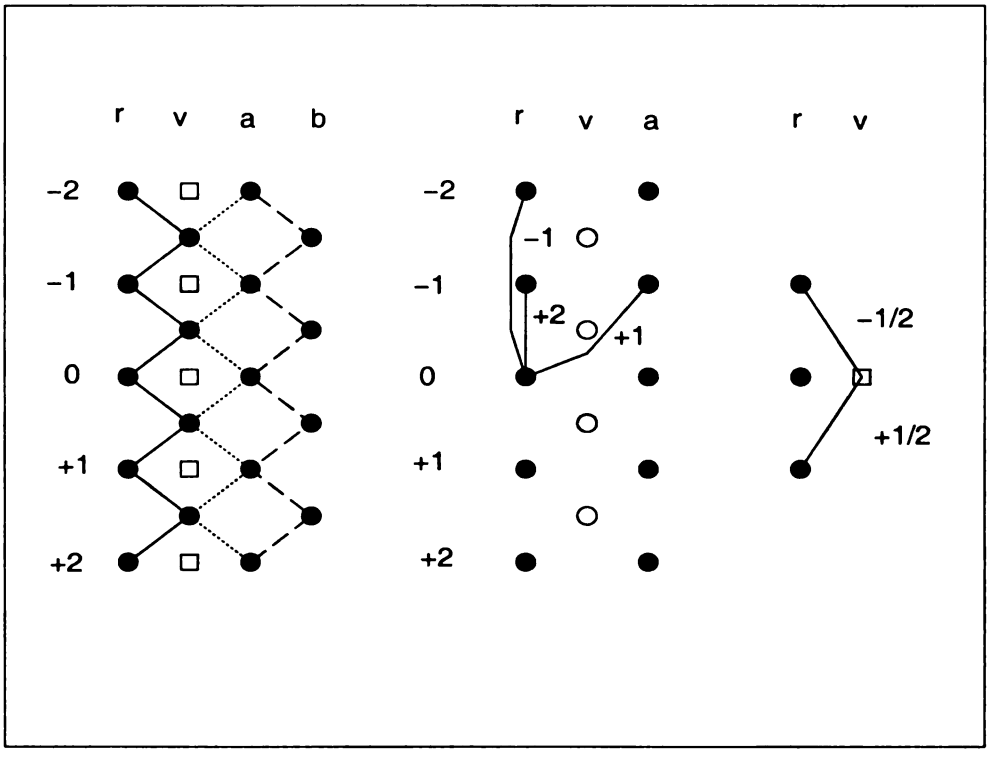

Figure 1.1: Verlet scheme represented on the lozenge diagram. The skeleton of the diagram is on the left: position, velocity, acceleration and third time derivative of the position. The Verlet scheme is in the middle. The velocity scheme is on the right.

Unfortunately the main advantage of the Verlet-scheme – its simplicity – causes the method to be not very stable numerically: a small term – $\delta t^2 \vec{a}$ – is added to the difference of two large terms – the positions. Large numbers of other, more sophisticated and more complicated schemes were developed in the past decades in order to give more precise and numerically more stable results. Obviously the price we have to pay for this higher accuracy is more computing time; therefore a careful balance must always be found between the quality of results and the calculation time.

The different integration schemes can be best understood and graphically visualized with the so called *lozenge* diagram. [3] See Figure 1.1. The framework of the diagram is shown on the left. The different columns represent positions, velocities, accelerations and the third derivatives of the positions at consecutive moments in

time. The time axis is vertical and it points downward. The present is labeled by zero in the figure. Every second column is shifted by half of the timestep δt. The quantities in these columns – marked by full circles – represent values in the middle of the time intervals. Calculating a physical quantity at a given point in a column using left neighbors means differentiation with respect to time: $v_{1/2} = (r_1 - r_0)/\delta t$, using right neighbors means integration in time: $r_1 - r_0 = \delta t v_{1/2}$. The diagram in the middle represents the Verlet integration scheme given by equation (1.1). The numbers next to the edges of the graph give the weights of the terms in the formula. Moving from the acceleration column to the coordinate column means integration in time twice resulting in a δt^2 factor. The velocity elements – the second column – are represented by empty circles because the velocity is not present in this scheme. The rightmost diagram corresponds to the equation (1.2). This formula calculates the velocity at the endpoint of a time interval using two next nearest neighbor positions. These types of velocity points are represented by empty squares on the lozenge diagram. Alternatively it is possible to calculate the velocities in the middle of the intervals using two nearest neighbor positions. This graphical representation helps us to understand and remember this and other integration schemes.

We need to be extremely cautious when we program integration schemes on computers. Floating point arithmetic is not associative contrary to the common assumption. When adding quantities very different in size – as in the Verlet scheme – small terms can easily get lost next to the large ones. The smaller a term is, the more computation it usually requires to be determined. Losing these terms is not just a waste of effort, but it can also modify the physical results of our simulations significantly. As a rule of thumb for coding, we should always group together variables holding quantities similar in magnitude before performing operations on them in order to minimize the loss of precision. Other simple but important rules, methods and integration schemes are described in [4] – [8]. At the present time Molecular

Dynamics simulations of atomic systems typically use $10^{-14} - 10^{-15}$ second for the timestep δt, perform 1 to 10 million iterations per particle reaching the timescales of nanoseconds. The size of the simulated system largely depends on the computer in use. The largest supercomputer simulations today can contain 100 million particles. The peak performance of these simulations is in the range of 10–20,000 atomiterations per second per processor [9] - [12]. This performance is still not sufficient to study atomic or molecular systems on long diffusive timescales.

1.2 Monte Carlo method

Monte Carlo type methods were used long before the development of electronic computers, the roots of these techniques date to centuries back. Ingenious mechanical devices - for example the Galton gasket - were used to study randomness and distributions of events. Random numbers generated by real physical phenomena were tabulated in tables and used by researchers in different fields. These tables are accessible even today in electronic or other forms readable by computers [13]. An analog mechanical Monte Carlo trolley was designed by Fermi [14] to simulate the diffusion of neutrons in nuclear reactors and other devices. With the advent of digital computers the Monte Carlo method became one of the most important simulation methods in science. We discuss random number generation by computers in a later chapter; here we describe the physical aspects of the method.

When we simulate a set of particles using Molecular Dynamics, we follow the time evolution of the particles. As we monitor them in time, we collect information about the behavior of the ensemble and we can "measure" physically important quantities, for example kinetic energy. These observables are our main goals in computer simulations. However we can employ other averaging methods. The different configurations of the particle system do not necessarily have to be consecutive in time reflecting

the true time evolution, they can be randomly generated independent snapshots of the system. This kind of averaging method is perfectly appropriate, if we are interested only in configurational properties in equilibrum, such as spatial distributions but not the time dependent dynamics. The "Monte Carlo method" is therefore the generic name for any of these types of techniques, sampling the configurational space randomly. Traditionally only the configurational space was sampled, however more advanced Monte Carlo methods incorporated momentum also as we will see in the case of the Lattice-Gas model. In this model the real space configurations visited after each other are not independent anymore, they represent a probabilistic time evolution of the ensemble. The "time" however is not the real physical time but the series of timesteps taken by the simulator. The relation between the real time and the simulation time is usually not known. This is generally the major drawback of any Monte Carlo method. The equivalence of the Molecular Dynamics type time averaging and the Monte Carlo type ensemble averaging – either configurational or phase space averaging – is one of the most fundamental questions in statistical and computational physics. Usually this equivalence holds true, but there are exceptions.

When should we use Monte Carlo methods instead of Molecular Dynamics? When for example the microscopic dynamics of the system is not known, or it is known but computationally expensive to simulate it. Or we just want to simulate random processes. The question of the true randomness in Nature versus our limited knowledge is rather philosophical, we do not address it here.

The first and most referred application of the Monte Carlo technique on digital computers was performed by Metropolis *et al.* [15]. Within the framework of the Monte Carlo method the acceptance probability of a configuration was derived and used successfully for the first time. Since then this special case of the Monte Carlo technique is called the Metropolis algorithm or Monte Carlo importance sampling, or because of its wide use, mistakenly simply Monte Carlo. We should keep in mind

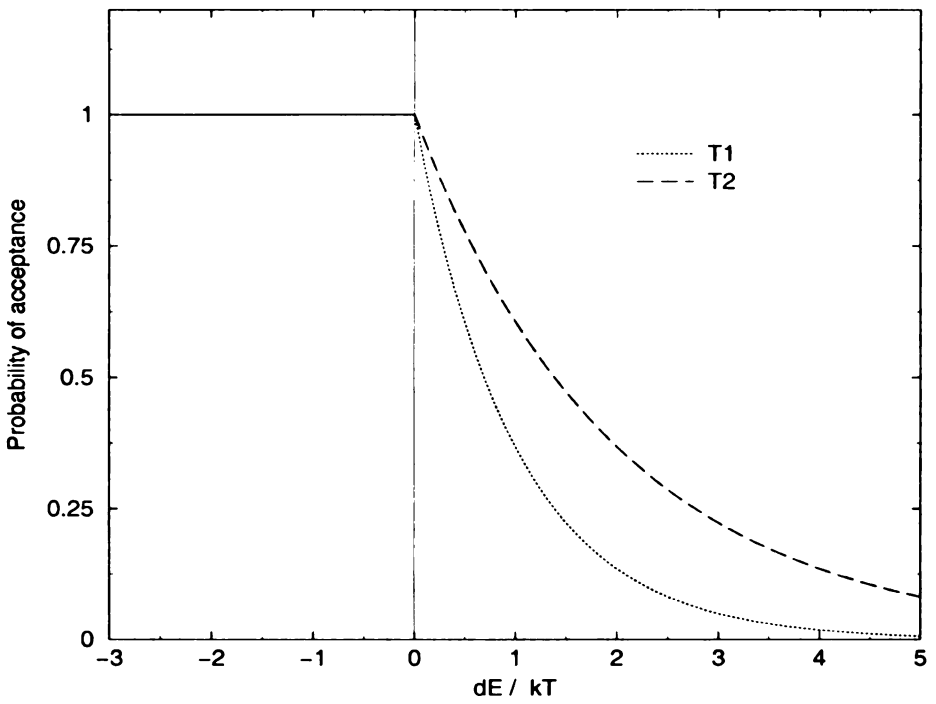

Figure 1.2: The Metropolis acceptance function at two different temperatures T_1 and T_2, where T_1 is the lower one.

that any method using random sampling can be called a Monte Carlo method. It can sample only the configuration space, but in the general case it might sample the full phase space also. The Metropolis algorithm is just one special recipe to accept or reject a newly generated state or configuration based on its Boltzmann weight relative to the old state or configuration. If the energy difference between the new trial and the old configuration is ΔE, then the new trial is accepted with a probability of $exp(-\Delta E/kT)$, where k is the Boltzmann constant, T is the temperature. If ΔE is negative, the new state is always accepted. This acceptance function is shown in Figure 1.2. for two different temperatures T_1 and T_2, where $T_1 < T_2$. If the temperature is low, then the acceptance probability is also low. At zero temperature only those steps are accepted which decrease or at least do not increase the energy. At infinitely high temperature every step is accepted independently of the increase or decrease in

energy. The heart of the Metropolis algorithm is that a new state is generated from the old one by a small random modification. This gives it its efficiency. Generating a completely new configuration each step would require large amount of calculation, although certain types of simulations might need this kind of sampling. This step by step simulation of the particle system is stochastic as opposed to the completely deterministic path in the case of Molecular Dynamics. If this evolution simulated by the computer is interpreted carefully, it can give information about the true time evolution of the system, but we always have to remember the probabilistic nature of the Monte Carlo methods when studying the real physical world. Furthermore very often physically forbidden steps are taken in Monte Carlo simulations, for example a particle is removed from inside the bulk of a fluid. This is clearly impossible in the real world. However we should not attribute physical meaning to one single event like this, useful information about observable quantities are collected through statistical averaging. Nonphysical steps in a Monte Carlo simulation usually help to reach equilibrum quicker than in a fully realistic simulation. On the other hand incorrect use of these steps might drive the whole simulation to a wrong track. There are no general rules, only computer experiments can decide about the validity of a given type of simulation step.

1.3 Computational Fluid Dynamics

In the framework of Computational Fluid Dynamics the gaseous or liquid substance in the question is described by continuous scalar and vector fields such as density, temperature, pressure, velocity. In the general case these physical quantities may change in time as well. Depending on the exact physical properties of the fluid – compressibility, viscosity – and the external forces and boundary conditions, very different physical laws and their equations must be applied. General treatment of the

problem is practically impossible. The most important class of equations in this area of physics is the Navier-Stokes equations. In the case of incompressible, viscous flow it takes the following form:

$$\frac{\partial \vec{v}}{\partial t} + (\vec{v}, \nabla)\vec{v} = -\frac{1}{\rho}\nabla p + \nu \Delta \vec{v}.$$

Because of the incompressibility the continuity equation will be:

$$\nabla \vec{v} = 0,$$

where $\vec{v} = \vec{v}(\vec{r}, t)$ is the velocity vector field, $p = p(\vec{r}, t)$ the pressure, $\rho = \rho(\vec{r}, t)$ the density scalar fields, ν the viscosity of the fluid. The formal scalar product $(\vec{v}, \nabla)$ is the differential operator

$$v_x \frac{\partial}{\partial x} + v_y \frac{\partial}{\partial y} + v_z \frac{\partial}{\partial z}$$

acting on every component of $\vec{v}$. Analytical solution is possible only for very special geometries and boundary conditions. In engineering, enormous amounts of computational power are spent in solving these equations numerically, and a huge amounts of manpower are invested in the development of these numerical methods. The majority of these methods are basically *finite difference* type numerical techniques. In real engineering problems the shape of the space where the physical quantities are calculated is usually arbitrary and lacks any symmetry. Orthogonal – cartesian, cylyndrical, spherical – discretization of the space is usually rather impractical; nonregular meshes adapted to the geometry of these problems are used instead. The difference operators might become very complicated on these grids, but still significant computational gain can be achieved compared to the orthogonal lattices. The techniques of using nonregular meshes or grids are commonly called *finite element* methods, causing general confusion with the name of finite difference methods. These finite element methods represent a very large field in numerical engineering today [16]; the detailed discussion is impossible here.

The Navier-Stokes equations have been solved for a large number of porous material geometries with fluids inside them under a broad range of physical conditions [17]. One major question is the physical validity of these solutions at different length scales. Computational Fluid Dynamics is definitely applicable down to the 1 mm – 100 micron range, but it surely fails at the 100 nanometer scales, where the world becomes truly atomistic. No simple answer can be given in the range between these two limits, fluid properties and external conditions can modify the physical behavior significantly. This intermediate length scale is not accessible to the methods described above, Molecular Dynamics cannot reach it efficiently on today's computers, and continuum fluid dynamics is not valid here. The Lattice-Gas method was developed in the effort to bring particle based simulation techniques and continuum methods closer.

1.4 Lattice-Gas method

The early history of the Lattice-Gas and other related discrete methods cannot be traced back in time precisely. One of the simplest version of this model was constructed and studied in the seventies by Hardy, de Pazzis and Pomeau [18], and it has been called the HPP model since then. In this model, pointlike particles move on a two dimensional *square* lattice, jumping one lattice unit in one time step in a synchronized way with each other. When two particles collide frontally they scatter away from each other perpendicular to their original direction. When they collide at a right angle, they proceed in their original direction without interaction. After long investigation of this model, it turned out that it does not recover the results of the Navier-Stokes equation perfectly, and the simulation results depend on the orientation of the lattice. After these discouraging results the model was abandoned and forgotten for about a decade. Due to the work by Wolfram [19] in the field of

Cellular Automata research, the Lattice-Gas model was rediscovered in 1986 by Frish et al. [20], but this time on a *triangular* lattice. This version of the model became known by the names Hexagonal Lattice-Gas (HLG) or Lattice-Gas Cellular Automata (LGCA). The underlying triangular lattice plays a very important role: due to the hexagonal symmetry, the Navier-Stokes equation is recovered correctly from the microscopic collision rules of the particles. Furthermore the results of the simulations are independent of the orientation of the lattice. Due to the correct physical predictions by the model and the high computational efficiency, it suddenly became extremely popular.

Let us briefly review the Lattice-Gas model at the microscopic level. The pointlike particles take one step of a lattice unit at every timestep simultaneously. At any moment no two particles moving in the same directions are allowed to occupy the same lattice site: no two particles are allowed to occupy the same phase space point. Particles traveling in different directions however are allowed to be at the same site. There are six different directions at every lattice site pointing to the six nearest neighbors of that site, therefore a maximum of six particles can occupy each site. The 64 possible occupation states are shown in Figure 1.3. There are 64 possible site states, because every direction can be in two states: it can contain a particle or it can be empty, hence $2^6 = 64$. For practical reasons the directions are labeled counterclockwise from 0 to 5, with the zeroth direction pointing to three o'clock. The binary representation of the state table from Figure 1.3 is shown here:

000000	000001	000010	000011	000100	000101	000110	000111
001000	001001	001010	001011	001100	001101	001110	001111
010000	010001	010010	010011	010100	010101	010110	010111
011000	011001	011010	011011	011100	011101	011110	011111
100000	100001	100010	100011	100100	100101	100110	100111
101000	101001	101010	101011	101100	101101	101110	101111

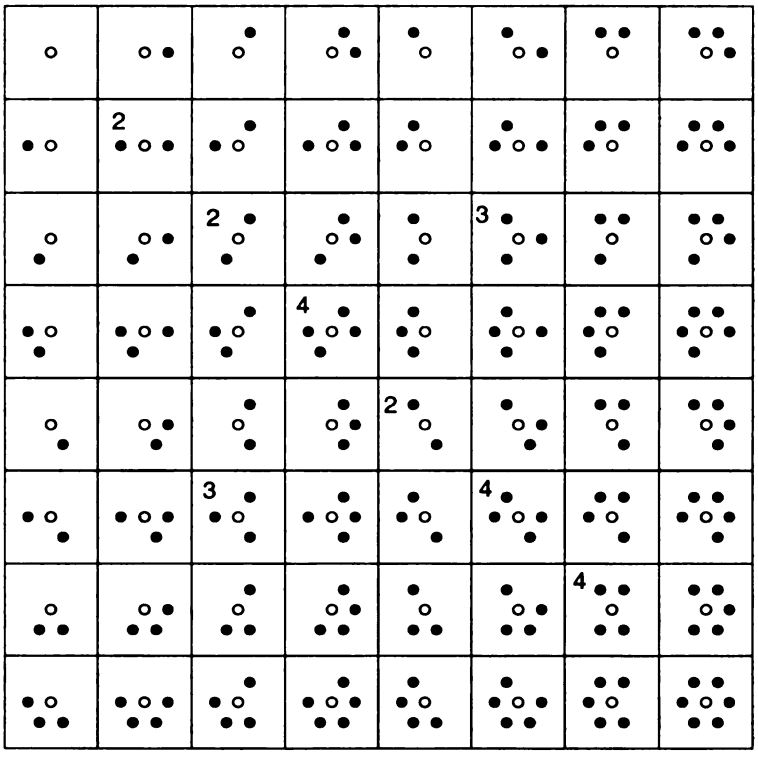

Figure 1.3: The 64 possible states of a site in the Lattice-Gas model. The empty circles represent the site itself, the full circles represent the particles moving out from this site. The state in the top left corner represents the completely empty site, the bottom right corner represents the completely full site, occupied by six particles moving out in the six possible directions.

110000	110001	110010	110011	110100	110101	110110	110111
111000	111001	111010	111011	111100	111101	111110	111111

The lowest – rightmost – bit is the zeroth bit. This bit represents a particle moving in the zeroth direction. If this bit is 1, then there is a particle moving in this direction, if this bit is 0, then there is no particle – or in other words there is a hole – moving in this direction. The decimal representation of these numbers is naturally:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	.	.	.	.	22	23
24	.	.	.	.	.	.	31
32	.	.	.	.	.	.	39
40	41	.	.	.	.	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

From the binary representation we can see that a lattice site is represented by six bits. The Lattice-Gas simulation is performed by manipulating these bits. Every iteration cycle is executed in two steps as follows:

1. Move the particles in their original direction by one step.
2. Resolve the particle collisions.

The collisions are resolved using the same state table shown in Figure 1.3. As we see some of the states in the table are labeled by arabic numbers. Three two-body collisions are labeled by 2, two three-body collisions are labeled by 3, and three four-body collisions are labeled by 4. The other states are not labeled. The collisions are resolved in the following way: if a state results in a particular label, then it is switched to another labeled state of the same arabic number with equal probability in the case when there is a choice. For example in the case of a frontal two-body collision

– labeled by 2 – the particles scatter with a 60 degree right twist or with a 60 degree left twist with 50-50 percent probability. In the case of the symmetric three-body collisions – labeled by 3 in the table – the state is always switched to the only other symmetric three-body state, meaning that the particles simply bounce back in the directions from which they originally came from. The four-body collisions – labeled by 4 – are completely analogous to the two body collisions, the probability of the left or right twist is 50-50 percent. All the other unlabeled states remain unchanged during the collision resolution, the particles in these interactions simply do not scatter at all. Mass – the number of particles – and momentum is conserved in each type of collision. It is easy to see that the state table together with the transition rules – and therefore the whole Lattice-Gas method itself – remain invariant under the exchange of the particles and the holes.

Several modifications and extra features have been added to the basic Lattice-Gas method during the past years [21], [22]. The power of these methods lies in their computational simplicity from the viewpoint of the computer: the language of physics is translated into the language of bit operations and Boolean logic, where computers work the best. One of the most efficient Fortran implementations of the simple Lattice-Gas model and its performance is given in [23]. As the extra features are introduced into the model, the elegant simplicity and therefore the computational efficiency easily gets lost.

1.5 Continuous versus discrete methods

In the previous sections we have reviewed the most important types of simulation techniques. Computationally all these techniques can be divided into two classes based on the underlying arithmetic used in the simulations: continuous or discrete, – in the language of the computers: floating point or integer type. Molecular Dynam-

ics and Computational Fluid Dynamics are traditionally continuous methods. Monte Carlo is a generic framework algorithm; it can drive both continuous or discrete simulations. Lattice-Gas is a completely discrete method. Historically the largest class of mathematical methods used in physics is the class of differential equations, for example Newton's equation of motion, the Maxwell equations, the Schrodinger equation. These equations contain continuous space, time and variables. When we solve these equations numerically on the computer, we often forget about the conflict between the continuous nature of these models and the discreteness of our machines. The floating point numbers and operations offered by the digital computers are just approximations of the real quantities and functions we need. For example a simple conversion is already problematic between decimal and binary number systems, because these two number systems are *incompatible.* Two number systems are called incompatible, if their base numbers have prime factor(s) not present in both base numbers. In our case $10 = 2 \times 5$, therefore 5 is a prime factor present in only one of the bases. Due to this fact, some of the decimal numbers with finite length would be equal to infinitely long periodically repeating binary numbers. The decimal number 0.1 is equal to the $0.000\overline{1100}$ binary number with the last four digits repeating infinitely. The opposite direction, binary to decimal conversion however will always produce finite sequences from finite sequences of digits. Because in our computer we have only finite amount of storage to hold the bits, this infinitely long binary sequence will be cut at some point. To demonstrate this, let us run the following program:

```
X = 0.005
WRITE(6,*) X
```

After storing the decimal number `0.005` in variable `X`, and writing it back to the screen, we get the not very surprising result: `4.9999999E-3`. The number `0.005` is of course just one of the many decimal numbers producing this phenomena. Interestingly not every fractional decimal number behaves like this. For example the

decimal number 2.1 is returned correctly by the program above, altough it is not representable in binary form using finite number of bits. On every computer there is a large set of decimal numbers which will not be recovered correctly after storing and simply retrieving it. The situation gets worse when we perform operations – addition, multiplication – on these numbers. After one single multiplication we might already start losing precision. The code

```
X = 2.1
Y = X*X
WRITE(6,*) X,Y
```

will produce `2.100000` and `4.409999` as output. When we solve differential equations or other continuous mathematical models, we constantly struggle with the numerical error trying to reduce it.

There are special purpose analog computers and devices – for example operational amplifiers – operating on continuous quantities, but they are far less flexible and universal than our digital programmable computers. There are theoretical models for continuous computation for example the UMRAM (Unit Multiplication Random Access Machine) [24], where the access, multiplication and addition of infinitely precise quantities are performed at unit cost. However this model still has to be implemented in hardware, if that is physically feasible at all.

There are completely discrete – discrete space, discrete time and discrete state – models: in mathematics for example the Cellular Automata models, in physics the spin models such as the Ising model. These two particlular classes of models are fundamentally equivalent to each other, they are just the result of two slightly different approaches – the approach of the mathematicians or computer scientists, and the physicists – to describe the same reality. This class of models can be exactly implemented on today's digital computers without the loss of any precision. Furthermore the computational efficiency of these discrete simulations is far better

than the efficiency of the continuous simulations, because the underlying arithmetic is integer type instead of floating point. The fundamental floating-point operations – addition and multiplication – can take 3 to 5 times more time on a typical computer today than an integer operation. With additional programming techniques – look-up tables, bit manipulation – the performance of discrete simulations can be increased even further.

Can we apply these efficient discrete techniques in the field of atomistic fluid simulations using lattice resolutions finer than in the Lattice-Gas? This thesis hopefully will give a convincing *yes* answer to this question. However we do not assume or claim any discreteness of the physical world at the level of atoms and molecules. We just simply apply discrete simulation and numerical techniques learned from other fields [25] in order to increase computational performance in this field.

Chapter 2

Modeling fluids

2.1 Introduction

In this chapter we introduce the simplest version of our model. The underlying lattice and the particles will be defined first. Along with the physical results the computational implementation will also be also explained. Computer codes are still very often neglected in scientific papers. There are only a few books written for physicists by physicists [4] - [8]. In these books the emphasis is naturally placed on the physics, not on the programming aspects. Scientists - with the obvious exception of computer scientists - usually do not have deep formal education in programming. Good coding is supposed to be picked up when doing *real* science. Graduate students are often puzzled by other people's code or struggling with the mysterious behavior of not very well-written and documented programs. It is not easy to turn this mystery and misery to mastery. In the hope that someday somebody will read this work and find an idea or a solution to a programming problem, several code fragments will be included in the text with explanations.

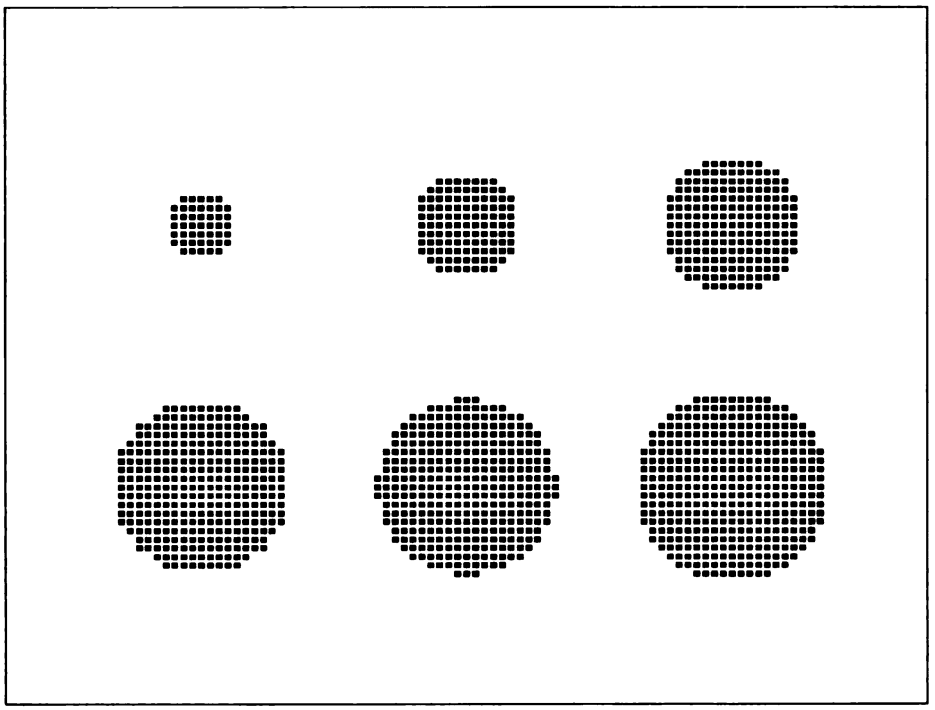

Figure 2.1: Circular objects with different radii on a two dimensional square lattice. The radii are 4.0, 6.0, 8.0, 10.0, 10.1 and 10.9 lattice units respectively.

2.2 The basic model

The simplest form of our model consist of a two-dimensional square lattice and particles on this lattice. The particles are not simple points anymore as in the Lattice-Gas method, but they have structure. The simplest structured object in this world is a circle with a given radius R measured in the lattice units a. Figure 2.1 shows six "circular" objects with the following radii: 4.0, 6.0, 8.0, 10.0, 10.1, 10.9. Notice that the radius is not necessarily an integer number: diagonal points - points that are on the edge of the circle between the horizontal and vertical diameters - can join or leave the circle when the radius is changed by less than a lattice unit. These circles – especially the small ones – are not very smooth, they have ragged edges. One can say they are not circular shaped at all. How can we expect to obtain smooth results from a rough model like this? This is one of the most critical questions in the applicability

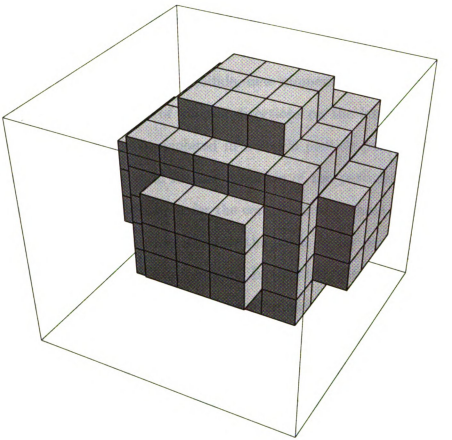

Figure 2.2: A three dimensional spherical object with radius 3.5 lattice units on a simple cubic lattice.

of the whole model, we return to it later when discussing the physical results. Besides the square lattice, triangular lattice will also be used in two dimensions and simple cubic lattice in three dimensions. A "spherical" object with radius 3.5 on the simple cubic lattice is shown in Figure 2.2. Due to the finite size of the particles, a collision can be non-central with a nonzero impact parameter. The momentum conservation can be satisfied on a discrete lattice only in the case of special initial conditions and impact parameters. For example, when two particles collide horizontally or vertically on the square lattice with an impact parameter exactly equal to half of the particles' radius. However an arbitrary collision would produce momentum vectors generally not representable on the lattice. Different lattices might have a different number of

special cases when the momentum might be conserved, but no lattice can satisfy the momentum conservation in the case of every possible collision.

In our model, when two particles collide, new momentum vectors are generated completely *randomly* from a set of allowed directions on the lattice. Therefore the momentum conservation will be violated in a single particle-particle collision, but it might remain statistically valid for the ensemble of the particles, or for the time average of one particle's collisions, or for the combination of these two averages.

We use two different schemes to simulate the paths of the particles between two collisions: random walk and ballistic motion. In the case of random walk a new direction is generated randomly at every timestep for each particle. A newly generated direction is statistically independent of a previous direction for any given particle. When two particles collide with each other during their walk, the originally chosen step by the initiating particle will be rejected as a geometrically forbidden step and this particle will not move at this timestep. The direction generated at the next timestep might coincide with the previous direction which was just rejected. In this case it will be rejected again, unless the other particle moved away in the meantime. In the case of ballistic motion the particles will remember their momentum direction between two timesteps. They will take their memorized step if it is allowed geometrically. However if a collision happens with another particle or with some geometrical constraint, the colliding particle stays at rest for one timestep and a new direction is generated and stored in the particle's memory to use in the next timestep. In the simplest case of this model the different momentum directions are statistically independent and have the same probability. This applies for both the random walk and the ballistic motion. Nonequal probability weights for the different momentum directions can be introduced into the model also.

Small example simulations are shown in Figure 2.3 for square and triangular lattices and for random walk and ballistic motion.

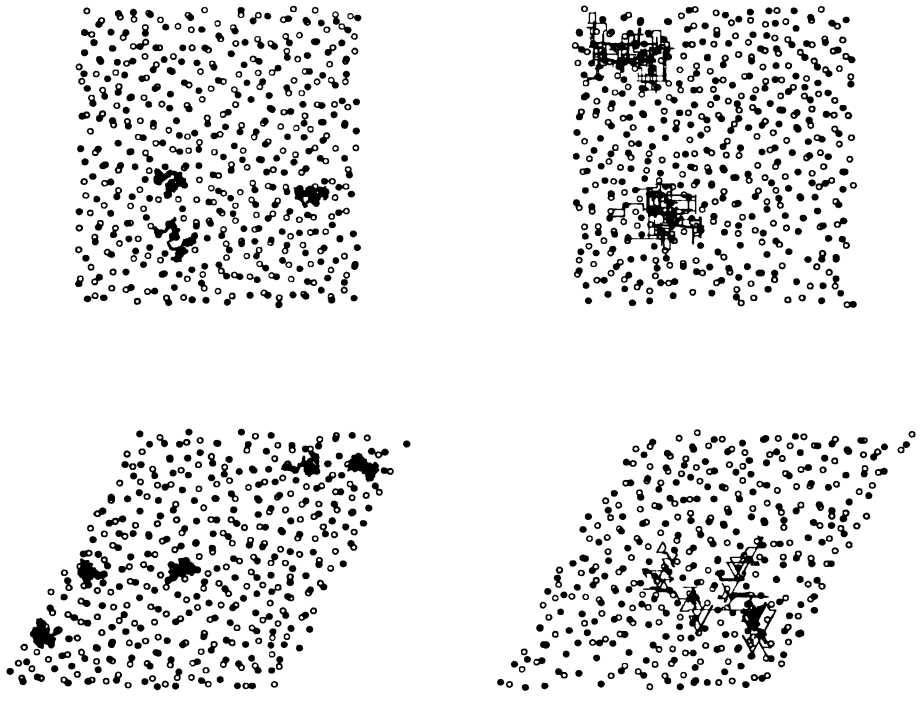

Figure 2.3: Two dimensional example simulations for square and triangular lattices. The underlying lattices are 512 by 512, the radius of the particles is 12.0 lattice units. The particles are shown with a smaller radius for clarity. The full circles represent the initial positions of the particles, the empty circles represent the final positions. Every particle took 5000 steps in each simulation by random walk in the left two figures and by ballistic motion in the two right figures. The number of particles is 250 in the top two and 200 in the bottom two figures. The paths of several particles are shown by continuous lines.

The word "timestep" always refers to the ensemble timestep, if not stated otherwise. In every ensemble timestep each particle is updated once. The initial particle configurations are generated randomly. Because of this random initial condition the ensemble update is performed in the same order on the particles at every timestep without introducing specific pattern into the motion. Random permutations of the particles between updates might be required only in the case of special geometric or pattern forming initial conditions.

2.3 Computational digression

Before implementing our model, we are going to discuss three computational subjects which will play an important roles in the performance of the implementation. These are the following: the order of indices when handling multidimensional arrays, binary arithmetics and random number generation. These topics are still not very well known to physicists despite their impact on the performance or the reliability of the simulation. These "tricks" can be useful in other areas in computational physics.

2.3.1 Order of indices

Large two and three dimensional arrays will have to be handled in our simulations. When manipulating these arrays, we use nested loops with different depths. Does the order of indices matter in these loops? To be concrete let us ask the following question: Which one of the following program fragments will be executed faster, version A or version B? (Let us assume that a two dimensional matrix A has to be filled by a numerical constant π.)

```
C Version A
      DO I=1,1000
        DO J=1,1000
```

```
          A(I,J) = 3.1415926  !first index is outermost
        ENDDO
      ENDDO
C Version B
      DO I=1,1000
        DO J=1,1000
          A(J,I) = 3.1415926  !first index is innermost
        ENDDO
      ENDDO
```

Our first answer is that they will need the same amount of time to finish, because the response time of the semiconductor memory device in a given machine is always the same independently of the location of the requested memory element. This is why it is called Random Access Memory (RAM), emphasizing that we can access its contents in any random order, the answer will always take the same time. (This is in contrast with the linearity of magnetic tapes for example, where the access time strongly depends on the position of the information on the tape.)

However a gap opened up between the speed of the Central Processing Units (CPUs) and the RAM memories. Typical CPUs today run at 300-800 MHz, memory chips on the other hand only at 133-100 MHz or less. This is due to the very intensive development on the CPU market. It became clear already in the mid-eighties, that the communication speed between the CPU and the memory is the crucial factor in the performance of a computer, not the raw frequency of the CPU. This phenomenon became known as the von Neumann bottleneck: the processing unit is not fed fast enough, it spends most of its time waiting for data from the memory. The majority of computers today are designed based on John von Neumann's concept [26], namely one active device – the CPU – is acting on one passive device – the memory. These are the type of machines sitting on almost every desk, and they suffer from this narrow

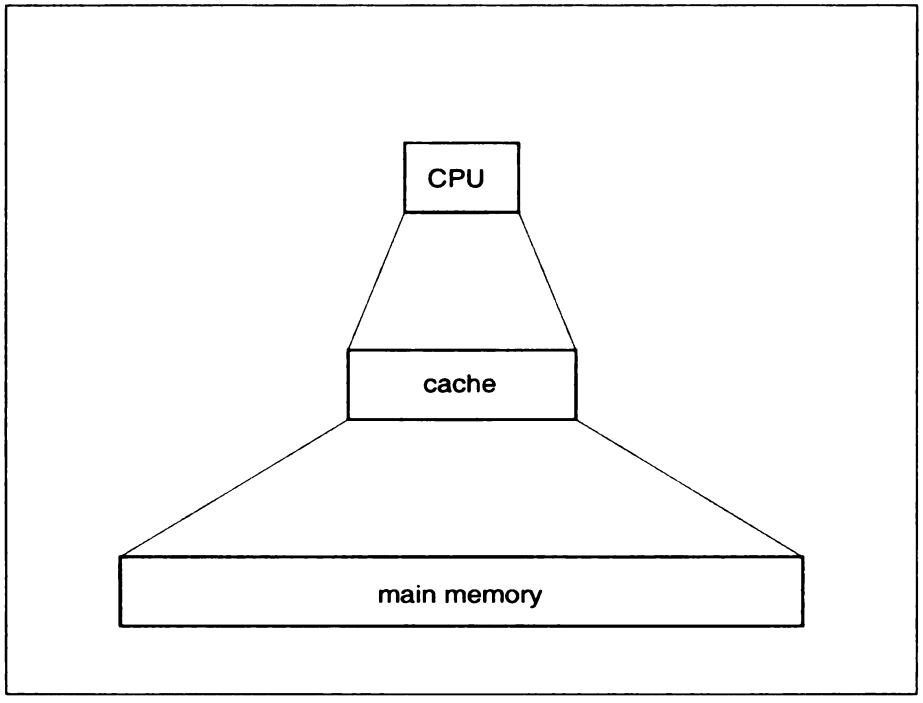

Figure 2.4: Intermediate cache memory between the CPU and the main memory. Only one level of cache is shown, but usually more than one is implemented in today's hardware. The other levels would function in the same way.

bandwidth constraint. There exist machines based on different concepts: massively parallel machines [27] or other exotic not CPU based constructions free from the bottleneck. These machines are ironically called "non-von" machines. Ironically because the theory of Cellular Automata used in the construction of the majority of these parallel machines was developed by von Neumann himself and Stanislaw Ulam [14, 28].

The solution to this bottleneck problem was the introduction of intermediate memory, the so called *cache* memory between the CPU and the main memory. See Figure 2.4. The cache runs at a faster speed than the main memory, but because of its fast speed, it is also more expensive. Due to the higher price it is smaller in capacity than the main memory. Information commutes in chunks between the

memory and the cache. The exact size of this unit largely depends on the exact type of the hardware and it is called by several different names at different levels of the hardware: line, block, page, but they all function similarly. As the CPU processes data or executes program code, the currently used block is transferred from the main memory to the cache giving faster access for the CPU. Transferring a block – which is a continuous series of data words – from the memory to the cache is performed faster than moving these units individually. From the cache these units are ready to be accessed individually by the CPU, but at the faster speed of the cache. This is the main idea behind memory caching. Unused inactive blocks are moved back from the cache to the main memory to give room for other blocks.

Today's personal computers have two levels of cache, workstations usually have three. The level closest to the CPU – called level one (L1) – is integrated on the CPU chip itself, and runs at the same speed. The size of the L1 cache is typically 8-64 Kbyte, the L2 is 1-2 Mbyte, but significant differences may exist depending on the exact design. This hierarchy of intermediate memories is able to speed up the execution of a program significantly, if the program accesses the data more linearly in the memory. In this case a block already in the cache contains the next piece of data. A new block must be fetched from the memory only when the boundary of the block has been reached. A program of this linear access type is called *cache-friendly.* The cache can contain several different blocks originally not necessarily consecutive in the memory. If the consecutive memory accesses are completely random in time a new block has to be transferred each time, resulting no gain in performance with respect to the cacheless design. A program of this type misses the cache often, giving high miss to hit ratio. This complicated memory managing process is usually controlled by hardware only, no intervention is required or even possible by the programmer [30, 31]. However with a little extra care we can make our programs cache-friendly and therefore more efficient. But how?

Turning back to the original matrix filling question, the answer is that version B will be executed about 3-5 times faster than version A depending on the frequency ratio of the CPU and the main memory in the machine. Why is version B the winner? When FORTRAN was designed, engineers had a thorough education in linear algebra. A matrix is usually pictured as a collection of column vectors, so it was natural to store a matrix as a linear sequence of the consecutive columns in the one-dimensional memory. Vertically neighboring elements in the matrix are stored next to each other in the memory, horizontal neighbors on the other hand can be far from each other depending on the size of the matrix. This storage scheme is called *column-major* ordering. In version B of our program the first index – the row index – runs faster, because this is the index of the inner loop. The memory access is therefore linear cache-friendly, resulting in good performance. See Figure 2.5. In version A matrix elements far from each other in memory are accessed, a block already in the cache will be missed in each new access, forcing one to bring a different block into the cache, not utilizing the faster speed of the cache. When coding array manipulation in FORTRAN, we should remember the principle of triple-F: "in Fortran the First index runs Fastest", therefore the first index must be the index of the innermost loop. However there are certain operations, for example the transposition of a matrix, when the indices will always be in the efficient order on one side of the equation but in the inefficient order on the other side, no matter which order we use:

```
C Matrix transposition
      DO I=1,1000
        DO J=1,1000
          A(J,I) = B(I,J)  !Left side OK, right inefficient
          or
          A(I,J) = B(J,I)  !Right side OK, left inefficient
        ENDDO
```

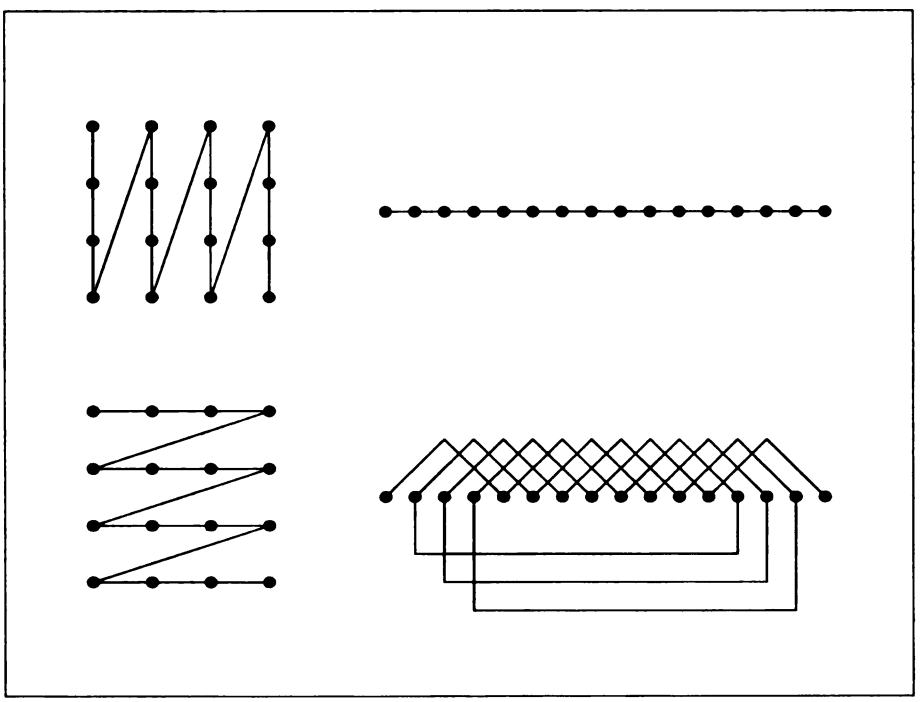

Figure 2.5: Accessing elements of a two dimensional matrix which was stored using column-major ordering. Top: sequential, cache-friendly and therefore fast access, equivalent to version B of the code. Bottom: nonsequential, cache-unfriendly slow access, equivalent to version A.

```
ENDDO
```

There are advanced transposing and ordering algorithms developed for operations like this, but they are beyond our scope [32]. Other important well known and well tuned algorithms - like the Fast Fourier Transformation - can be redesigned for this new hardware with cache memory, achieving even higher performance [33]. Simple fundamental concepts and techniques are explained in [34] - [37].

More modern programming languages like PASCAL, C and others developed after FORTRAN had chosen the row-major ordering to store arrays. The reason behind this decision was the ignorance of linear algebra by modern time programmers. When they write nested loops they simply pick the alphabetical order for both the loop indices and the array indices, which gives the inefficient memory access in the case of the FORTRAN type column-major storage. The switch of the ordering fixes this problem, but introduces other difficulties in multi-language projects. When a matrix is passed between subroutines or functions written in languages using different storage order, the matrix appears transposed. Because transposing is exactly what we want to avoid, the same algorithms must be coded in a transposed way in the two languages. This will make interlanguage translation - either automatic or manual - difficult and error prone.

2.3.2 Binary arithmetic

Manipulating numbers at the binary level is generally regarded as the lowest and dirtiest level of activity even by programmers. Physicists and mathematicians try to program the computer at a high abstract level, not thinking too much about the cost and the efficiency of the calculation. ("Mathematicians don't understand the cost of operations." D. Knuth. By mathematicians he refers to everybody who is not a computer scientist.) This tendency is perfectly normal, if the calculation takes much less time than programming the calculation itself. On the other hand if the simulation

lasts much longer than the programming, then it is worthwile to investigate faster techniques, even if they are not obvious to us, but are more efficient for the computers.

Changing the underlying arithmetic of a simulation from floating point to integer can yield significant speed-up, as we mentioned earlier. A processor typically performs an integer operation (addition, subtraction, bit manipulation) in one clock-cycle, while floating point addition or subtraction takes two, three or even more cycles. Integer multiplication usually requires longer time than addition. Similarly, floating point multiplication takes more time than floating point addition, and floating point division is slower than floating point multiplication. Integer division or modulo calculation however might take even longer than floating point operations, and the execution time is strongly dependent on the operands. This operation therefore should be avoided, whenever is possible. Integer division is usually not even implemented in modern CPU's instruction set, due to the complexity of this operation. Either software function is used instead, or it is performed with floating point arithmetic. An alternative to modulo for a special case will be given later in this section.

Now we will review how the integer numbers are represented in binary form and how to operate on these numbers. We will use these methods when we implement periodic boundary condition in our simulation or when we sort the particles into cells inside the simulation box. The examples will be given using 8 bit numbers but only for brevity. These methods will work analogously if the underlying numbers consist of 16, 32 or more bits.

If only positive numbers are needed the values between 0 and 255 can be stored in eight bits:

```
11111111 = 255  largest representable number
11111110 = 254
...
10000000 = 128
```

```
01111111 = 127
...
00000001 =   1
00000000 =   0  zero
```

The first scheme to represent negative numbers beside positive ones was developed very early for the first computers and it was called "one's complement" representation. The negative pair of a given positive number is generated by complementing all the bits to their opposite values. For example the pair of +9 and −9 looks like this:

```
00001001 = +9
11110110 = -9.
```

As we can, see complementation works in the opposite direction the same way: +9 can be generated from −9 by flipping every bit in the binary form. The only problem with this scheme is that it contains two zeros: the +0 and the −0:

```
00000000 = +0
11111111 = -0.
```

To correct this deficiency the "two's complement" method was constructed by a little modification of the one's complement: after flipping the bits an extra 1 is added to the result. The zero becomes a unique number, the fix point of this transformation:

```
00000000  original zero
11111111  after bitflip
100000000 adding 1 to the result.
```

Altough the highest – ninth – bit became 1, it is simply thrown away, because we can have only eight bits in a byte. When we use 16 or 32 bit long numbers this extra bit is ignored similarly in the process. It is possible to catch this bit, but we do not need it for our purposes. Our +9 and −9 example will look like this now:

```
00001001 = +9
11110111 = -9.
```

Interestingly this scheme works in both directions also. The best way to imagine the two's complement representation is the following: let us cut the list of signless positive numbers from 0 to 255 into half at the midpoint between 127 and 128. Slide the upper half (128-255) under the zero keeping the original orientation of the list. The original upper half becomes the lower negative half:

```
01111111 = +127
01111110 = +126
...
00000010 =   +2
00000001 =   +1
00000000 =    0  zero
11111111 =   -1  (was 255)
11111110 =   -2  (was 254)
...
10000001 = -127  (was 129)
10000000 = -128  (was 128)
```

As we can see there are 128 negative and 127 positive numbers. The highest (leftmost) bit became the sign bit: it is 0 for the positive numbers, and it is 1 for the negative numbers. Generally, if we have n bits, then $(2^{n-1} - 1)$ positive and (2^{n-1}) negative numbers can be represented around the zero. When using 16 bits, the representable interval is from $-32,768$ to $+32,767$, when using 32 bits it is from $-2,147,483,648$ to $+2,147,483,647$. The general purpose computers and programming languages today always use the two's complement representation. Complementation is defined for nonbinary number-systems also. In this case the general name of one's complement

is *radix-minus-one* complement or *diminished* radix complement, the generalization of two's complement is called *radix* complement or *true* complement.

One more binary operation must be understood before our physical application, namely masking bits out from a number. If for example only the five lowest bits are needed from an eight bit long number, then the following bitwise AND operation will give the required result:

```
     01101101  source
     00011111  mask
AND  --------
     00001101  result
```

The original data is AND-ed with the mask which lets through only those bits from the source where the maskbits are equal to 1. The other bits from the source, where the mask is 0, are set to zero independently of their original value. The bits at these positions were *masked out.*

Now let us turn to our physical application: a randomly walking particle moves on a one-dimensional discrete lattice of 32 sites. How can we implement periodic boundary condition? Let us label the lattice points starting from zero: 0,1,2,...31. When the walker takes a step, the new position must be checked, and if it is outside of the allowed range, it must be mapped back at the opposite end of the lattice. The following integer variables and codefragment will perform this operation:

```
NS = 32              !size of the lattice defined
MS = NS-1            !mask for bitwise operation
...                  !some code can be here
NEWX = ...           !new position is generated
IX = IAND(NEWX,MS)   !periodic boundary condition applied
...                  !more code can come here
```

The function **IAND** is the name of the bitwise AND operator in FORTRAN. How does this whole method work? If the walker wants to leave the simulation box on the right side with a coordinate value 32, the

```
        00100000  =  32  NEWX   new coordinate
        00011111  =  31  MS     mask
AND     --------
        00000000  =   0  IX     corrected coordinate
```

operation will bring it back on the left side with a coordinate of zero. The indexing of the lattice points starts at zero as the leftmost lattice site. When the walker tries to leave the box on the left with a coordinate value of -1, the same AND operation as before will map it back to the allowed range on the right side:

```
        11111111  =  -1  NEWX   new coordinate
        00011111  =  31  MS     mask
AND     --------
        00011111  =  31  IX     corrected coordinate
```

This method works correctly even if the walker is allowed to jump longer than one lattice constant when trying to leave the box: it will be brought back on the opposite side of the box to the proper position. Mathematically this AND operation between the mask - equal to $(2^n - 1)$ - and an arbitrary operand - the source - is a modulo calculation: it calculates the remainder of the source when divided by 2^n simply keeping the lower n bits and deleting all the higher ones. This is a faster way to calculate remainder than integer division, unfortunately it can be applied only when the divider is integer power of two. These examples were given using 8 bit arithmetic, but again the method works when using 16 or 32 bit arithmetic. In the case of two or three dimensional simulation this periodic boundary correction must be performed on each coordinate independently. The simulation box does not necessarily have

to be square or cubic shaped, different dimensions can have different lengths. The only requirement for the lengths is that they must be exact powers of two, and the indexing must start from zero. The mask must be the length of the box minus one in every dimension. The other traditionally used methods – if-then coordinate checking, modulo arithmetic, look-up table – do not impose this size restriction on the box, but they need significantly more time to execute compared to the bitwise AND which is performed in one machine cycle. Furthermore when a Fourier transformation of the simulation box is needed, power-of-two sizes will be clearly advantageous.

One more technique will be explained in this section; namely how to sort particles into cells. The simulation box is often divided into smaller units called cells. The particles are sorted into the cells based on their coordinates. This classification is required when collecting spatial distribution data or when using the cell-list method to simulate the system. (The cell-list method will be described in detail later.) The question is the same as before: how to do it quickly? The answer is the same again: by using bit manipulation. In the previous example the size of the simulation box was 32. Let us divide this box into 4 cells of length 8 each. For better understanding let us pick more than one particle with random coordinates from inside the box:

```
10:110  =  22  coordinate  #1
00:110  =   6  coordinate  #2
10:101  =  21  coordinate  #3
```

The binary form of the coordinates are divided into two parts by the colon: a lower and a higher part. This colon serves as a delimiter only for clarity, it does not hold place and it does not have any mathematical function. The higher part tells which cell the coordinate is from, the lower part tells the relative position within that cell. There are 2 high bits giving $2^2 = 4$ cells, and there are 3 low bits allowing $2^3 = 8$ positions inside a given cell. The total number of bits is 5, resulting in $2^5 = 32$ possible positions, which was our original initial condition. The high part of the first and the

third coordinate is the same, namely binary 10 , meaning that these coordinates are from the same cell, the second cell. The high part of the second coordinate is 00, this particle is in the zeroeth cell. The low part of the first and the second coordinate is equal - 110 - : these two particles occupy the same relative position - the sixth position - within two different cells. Generally the form of a number in any number system - not only in binary - is a pathway: the highest digit tells us which main cell the number is in. In the case of the binary number system we can choose between two cells: left or right, zero or one. In the case of decimal system we can choose from ten cells, from 0 to 9. The consecutive lower digits tell the position within the previous cells with more and more precision. This process is shown in Figure 2.6. For clarity only three bits are used instead of five as in the numerical example above.

By now we know it is straightforward to determine the lower part of a coordinate, we just have to mask out the higher bits and keep the lower ones by using the proper mask:

```
    10:110  =  22  coordinate #1
    00:111  =   7  mask
AND ------
    00:110  =   6  relative coordinate inside the cell
```

To extract the higher part of the coordinate we need a different treatment, because a simple masking will leave the high bits in their original position, giving incorrect result:

```
    10:110  =  22  coordinate #1
    11:000  =  24  mask
AND ------
    10:000  =  16  wrong result
```

To fix this problem, we have to shift the final result to the right by three places.

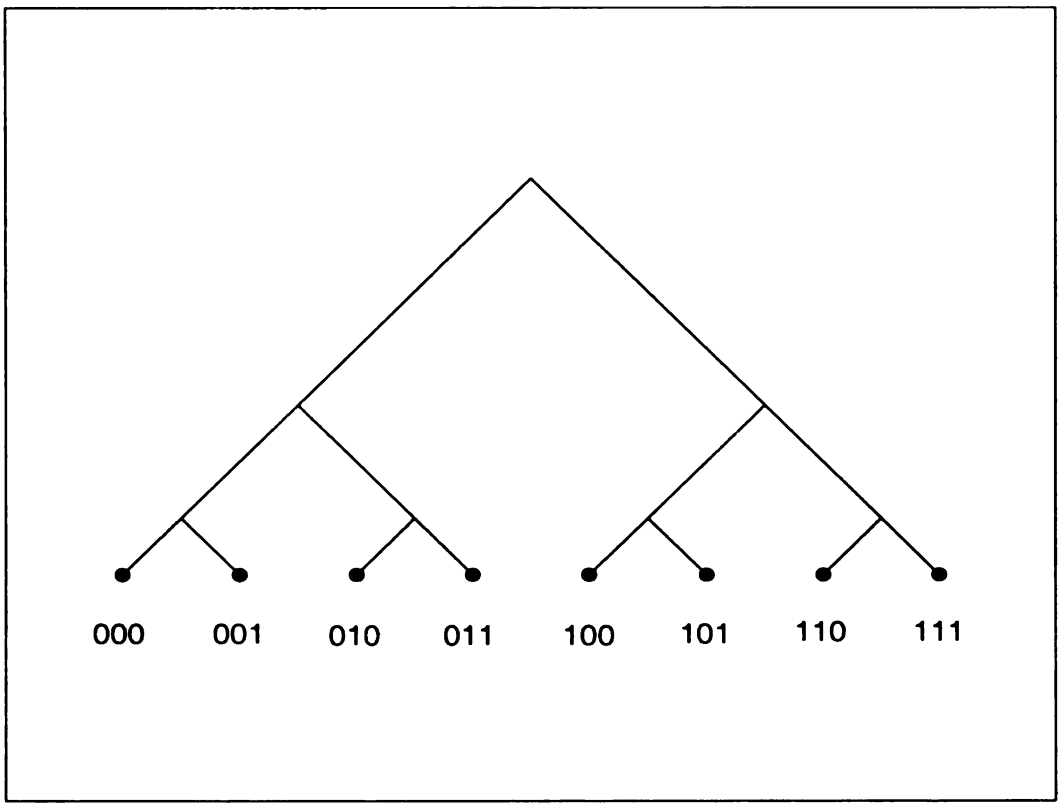

Figure 2.6: The binary tree for three bits. Every three digit binary number at the bottom is a path on the tree. A 0 bit means left, a 1 bit means right at the nodes where the tree splits into smaller branches. The highest bit of the numbers tells to which half the number belongs. The highest two bits locate the quarter, the highest three bits the eighth.

When shifting a binary number in either right or left direction the bits leaving the number will be dropped (shifted out), bits entering at the other end will be filled with zeros. Therefore there is no need for the initial AND operation shown above at all, the bits not needed will be deleted during this shift:

```
        10110  =  22  coordinate #1
SHIFT#1 01011  =  11
SHIFT#2 00101  =   5
SHIFT#3 00010  =   2  correct cell number.
```

These three shifts above can be written in one FORTRAN statement, and they will be executed in one step in one CPU-cycle:

```
IC = ISHFT(IX,-3)
```

The `ISHFT` function needs two arguments: the variable to be shifted – which is our original coordinate `IX` – and the number of shifts to be carried out. The shift is signed, negative shifts move the bits to right, to the direction of lower places. Positive shifts move the bits to left, to the direction of higher places. The incoming new bits have zero value, the bits shifted out are lost. Several other types of shifts exist on different CPUs, but this is the only type we will need. This kind of shift is usually called a *logical* shift. The logical shift is nothing but dividing or multiplying by 2^n where n is the number of shifts. The lower bits lost during the right-shift give the reminder in the division. These bits were not lost. We kept them when we masked them out before the shift, as we did in this example.

The techniques described in this section are simple and very powerful. Contrary to the common belief, bitwise operators are implemented in FORTRAN, and these techniques are portable between platforms. Strictly speaking bit operators are not part of the FORTRAN-77 standard, but after the Department of Defense recommended inclusion into the language, they were implemented in a standard way. With the

FORTRAN 90/95 version of the language these operators became a standard part of the language.

2.3.3 Random number generation

Proper generation and correct use of random numbers are still very often in the center of scientific arguments:

"My random number generator is faster than yours!"

"Yes, but my generator produces better random numbers!"

Can we generate real randomness with a completely deterministic computer? The answer is clearly *no*, we can only mimic randomness.

The first generator was proposed by von Neumann [4, 39] and used first by Metropolis [15]. This generator is a simple "square and extract" method: it squares a four digit decimal integer resulting in an eight digit number and then it extracts the middle four digits. This generator turned out to be rather poor, and failed many important statistical tests since its first application. However the final physical results produced using this generator were correct in [15], and are still correct. The question of randomness and the generation of it by deterministic computers turned out to be one of the hardest questions in mathematics. How well can we mimic randomness with a simple algorithm like this? Von Neumann's own answer was not very promising: "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." [39]. Several more sophisticated algorithms were developed to generate pseudo random numbers since von Neumann's generator. The most widely used is definitely the *linear congruental* method. Let us generate a series of integer numbers in the following recursive way:

$$x_{n+1} = (ax_n + c) \bmod m, \tag{2.1}$$

where m is the modulus, a is the multiplier, c is the increment. The very first

starting value x_0 is often called *seed*. The parameter m must be large in order to give a long series of numbers. It is usually 2^{32} on today's computers to make the modulo calculation efficient as we learned in the previous section. The critical parameter is a. If it is chosen carefully, then the generator will walk through every number between 0 and $(2^{32} - 1)$ before it starts to repeat itself, and the series of the numbers will mimic randomness well. The most important criteria to determine a good value for a are given in [39]. The generators on Digital and Sun workstations currently use $a = 69069$. The value of c is not very critical, it is usually chosen to be 1 in order to prevent the generator from getting trapped inside a loop that generates zeros infinitely. Several generators that use different parameters are given with their limitations and capabilities in [13].

The starting value x_0 simply tells the generator where to start its walk through the numbers. In practice the generated integer numbers are mapped linearly into the $[0, 1)$ interval – inclusive on the left, exclusive on the right – by a floating point division with 2^{32}. This final result is given back by the function through its name:

```
X = RAN(NSEED)
```

The floating point variable `X` will contain the resulting random number. At the same time the integer seed is also modified. The modification of the argument by the function is called *side effect*. This is the way the generator function remembers the last random number between two function calls. It is not advisable to change the seed at every call, but it is perfectly legal. However the "never change the seed after it is initialized" rule is not true either. If our program is done with one simulation, it can start a new one in the same run with a new seed, so the two simulations become logically independent. To understand the generator better, we will now abuse it with the following program by forcing it to take the $0, 1, 2, \ldots 5$ seeds systematically:

```
DO I=0,5
```

```
      NSEED = I
      X     = RAN(NSEED)
      WRITE(6,*) I,X,NSEED
    ENDDO
```

Notice that variable I is not passed directly to the function **RAN** because that would modify it as a side effect. Modification of the loop variable inside the loop is possible, but not wise. The program will give the output:

```
0   0.0000000            1
1   0.0000160        69070
2   0.0000321       138139
3   0.0000482       207208
4   0.0000643       276277
5   0.0000804       345346
```

From this we can recognize that the multiplier $a = 69069$, the increment $c = 1$, and the floating point random number is generated from the integer one by a division with 2^{32}. If we do not force the generator as above, but instead we let it keep its seed between calls, and at the same time we convert the integer random numbers to binary, we will be able to see the pattern even more clearly. The program

```
    NSEED = 0
    DO I=0,5
      X = RAN(NSEED)
      WRITE(6,*) I,X,NSEED
    ENDDO
```

will generate the output:

```
0   0.0000000            1   00000000000000000000000000000001
```

```
 1   0.0000160          69070   00000000000000010000110111001110
 2   0.1107408      475628535   00011100010110011000001111110111
 3   0.7630801    -1017563188   11000011010110010011011111001100
 4   0.1799780      772999773   00101110000100110000101001011101
 5   0.9028781     -417135238   11100111001000110000010101111010
```

(The code above is not complete; the binary conversion is not included for clarity.) The lowest – rightmost – bits are not random at all, they alternate periodically. This is due to eq. (2.1): from an odd number an even is generated, and from an even an odd is generated. Therefore the other commonly used "always initialize the generator with a *large odd* seed" is not true either. Even if we start with zero seed as above, only the first one or two numbers will not be random, which cannot be very dangerous in a large simulation. If we plan to use the integer random number at the level of bits, then the lowest ones should be avoided. For this special purpose there are generators that produce random bits at every place [13]. However these bit generators can be used efficiently only if the interval we need to cover is an exact power of two. Some parameters such as the size of the simulation box will always be exact power of two in our simulations, but others – for example the number of particles, or the number of allowed particle steps on the lattice – might be arbitrary integers. Therefore a linear "stretching" of the $[0, 1)$ interval is used: the floating-point random number is multiplied by a proper constant. This is the only floating point operation currently in the core of the simulations. According to the specifications, the generators are not allowed to give 1.0 as a result. However, depending on the exact properties of the hardware and the software, some generators may function incorrectly. The **RANO** generator from [13] on the DEC Alpha processor using the FORTRAN-77 compiler produces 1.0 once in about every $30 - 50$ million calls. It is like getting a seven when rolling a six sided dice! If we are lucky, our simulation program stops with a fatal error message. If we are not so lucky, it will keep running, producing strange or

unphysical results. The other generators: **RAN1**, **RAN2** from [13] and the **RAN** intrinsic function built into FORTRAN-77 on the Alpha processors have worked correctly so far.

When performing serious calculations, not only the seed, but also the generator must be changed periodically. If the results are invariant under these changes, the Monte-Carlo computational side of the simulation is correct. Conceptual or physical problems however may still exist. Changing the generators requires editing and recompiling of the code. But the seed should never be "wired into" the code as is commonly practiced. If a simulation is performed using the same physical parameters – size of the box, number of particles – and the same seed, the results of different runs will be exactly identical, leading to wrong conclusions. The best practice is to use the following or similar input data in our parameter files:

```
T          GENSEEDFLAG    T:generate your own, F:read next line
-23986     NSEED          manual seed
```

This is a snip from an actual parameter file read by the simulation programs. The input parameters are on the left, the names of the variables are in the middle, comments are on the right. If the flag is true – `T` –, the program generates its own seed from the internal clock. (This procedure is not explained here, but the source code to build a seed from the clock is given in the Appendix.) If it is false – `F` –, it uses the manual seed given in the other line. In both cases the initial seed is then written back to a log file for debugging purposes. When the results are strange or unexpected, the seed from the log file is fed back to the simulation along with the unmodified physical parameters, to reproduce the results. If the problem is reproducible, we have a chance to catch it.

We used the methodology described here in each simulation. The generators were periodically rotated. No unexplained computational behavior was found in the final results with the exeption of the illegal value of 1.0 returned by the **RAN0** generator as

mentioned. Therefore this generator was not used.

2.4 Implementation

Based on the computational techniques discussed in the previous chapter we will now implement several methods in practice needed for our simulations. These methods range from simple tricks applicable only for special situations to generally usable methods. In the codefragments the implicit type definition – which is the default in FORTRAN – is always turned off by the statement `IMPLICIT NONE`. If not stated otherwise, the variables are assumed to be `INTEGER`, independently from their first letter. Only very few `REAL` variables and functions are used, these will be clearly specified. If there are line numbers on the right side of the program listings, then they are used only in the explanations, they are not part of the code.

2.4.1 Coordinate based methods generally

When simulating a particle system, the minimum amount of information we need is the set of positions defined by two or three coordinates depending on the dimensionality of the problem:

$$\vec{r_i} = (x_i, y_i, (z_i)),$$

where i goes from 1 to N, to the number of particles. Depending on the exact type of the simulation, velocities or accelerations might also be required. In our model velocities are used in addition to coordinates in the ballistic type simulation, and only coordinates in the random walk type simulation. If the particles are different in size or have extra features, the information describing them must also be listed. The interaction – either in the form of force or energy – is calculated based on the coordinates. Velocity dependent interactions are also possible, but they are not very common. In condensed matter physics the interactions are short range, meaning

that particles interact only if they are close to each other, typically a few particle sizes away. This would suggest that only a fairly small number of interactions per particle must be calculated by the computer. However, using only the coordinates, we cannot tell if two particle are within the interaction range or not before actually calculating the distance between them. This turns out to be the central problem of all simulation algorithms: altough the linear list of the coordinates describes the particle configuration unambigously, it is not a suitable data structure for efficient simulation. Algorithms taht use the coordinate list usually perform only at $O(N^2)$, *i.e.* the calculation time depends quadratically on the size of the physical system simulated, because at each step every particle's distance is calculated from every other particle in order to decide if they interact or not.

By allocating extra storage for additional – and therefore redundant – information it is possible to speed up this basic algorithm significantly. Two important methods were developed for this purpose: the neighbor-list method by Verlet [2] and the cell-list method [38]. In both of these techniques, lists are constructed with particles within a safely large distance range. In the neighbor-list method these lists are attached to the particles, in the cell-list method to the cells. (The cells are small regular periodic regions within the simulation box.) With the aid of these lists the simulation steps become $O(N)$ in both cases: the interaction for each particle is calculated by using the particles on the lists only. However these lists have to be kept consistent, they have to be refreshed periodically as the system evolves. Refreshing the neighbor-lists unfortunately needs $O(N^2)$ calculation traditionally, leaving this method in the $O(N^2)$ class. The cell-lists on the other hand can be updated in $O(N)$ time, making this method to a truly linear algorithm. In the next section a new algorithm is presented, where the neighbor-lists are refreshed quickly, *i.e.* in $O(N)$ time, rendering this method to be a true linear algorithm also. In the section after that we show how to generate the cell-lists on a discrete lattice using bit manipulation. After that

another new method, the field-representation is presented. This method is based on the Lattice-Gas model, it uses a lattice to represent the particles, but at a higher resolution. Finally the most important computational tricks used on the triangular lattice are explained.

Let us now implement the fundamental data structures we need in our simulations. As we mentioned earlier the coordinates of the particles are necessary:

```
INTEGER   LATOM
PARAMETER (LATOM=99999)
INTEGER   X(0:LATOM),Y(0:LATOM)
INTEGER   NATOM
```

In three dimensions the third coordinates will also be declared. The parameter **LATOM** controls the amount of memory to be allocated, the variable **NATOM** contains the actual number of atoms given by the user in the parameter file. Indexing starts from zero for reasons explained later. Parameters controlling memory usage start with letter **L**, actual variables characterizing the simulation start with **N**. The size of the simulation box is given by the **LS-NS** pair:

```
INTEGER   LS
PARAMETER (LS=4095)
INTEGER   NS
```

where the content of **NS** is provided by the user, and it must be an exact power of two, for example 1024, 2048 or at most 4096 in the example above. The bitmask for the periodic boundary condition calculation is always generated by:

```
MS = NS - 1
```

The letter **S** refers to the word *Size* in the examples above. If the simulatin box is not square or cubic shaped, separate variables are needed in each dimension: **NX**, **NY**,

NZ. They all have to be exact powers of two. The allowed directions – either for the case of the random walk or for the ballistic flight – are defined by:

```
INTEGER   NDIR
INTEGER   DIRX(0:7),DIRY(0:7)
```

where NDIR holds the number of allowed directions, DIRX and DIRY are the components. All of these variables are initialized from a parameter file supplied by the user. This section of the parameter file usually looks like:

```
4          NDIR   number of directions
 1  0      Right
-1  0      Left
 0  1      Up
 0 -1      Down
```

In three dimensions there is a third component, and the indexing goes from 0 to 25 in order to accomodate nearest, next nearest and second nearest jumps on the simple cubic lattice. One of the few REAL entities is the random number generator function:

```
REAL      RAN
INTEGER   NSEED
```

The generator can be either the built in generator or it can be taken from for example [13]. Its seed is declared after it, and it is initalized from the system clock or from the parameter file as discussed in the previous section, but it is not "wired into" the source code. These are the most important variables necessary for our simulation. With them we can already understand the central driver of the random walk type simulation:

```
MATOM = NATOM - 1
... some code can be here ...
```

```
DO J=0,MATOM
  WD   = INT(NDIR*RAN(NSEED))
  NEWX = IAND((X(J) + DIRX(WD)),MS)
  NEWY = IAND((Y(J) + DIRY(WD)),MS)
  ...
  rest of the simulation comes here
  ...
ENDDO
```

For each atom a *Working Direction* WD is generated randomly from the set of allowed directions. This is one of the several places, where we can save an extra addition, because the indexing of the directions starts from zero and not one. If we started from one, we would need:

```
WD = INT(NDIR+RAN(NSEED)) + 1
```

This painful pedantry seems to be unnecessary, but this part of the simulation is executed most frequently. Any extra operation here can waste a large amount of CPU time. The new position is generated from the old position and the displacement. The periodic boundary condition correction is performed by the IAND function as we can see. Other programming details will be given in the next sections.

2.4.2 A fast neighbor-list method

The traditional neighbor-list method was developed by Verlet [2] to speed up the basic "every particle compared to every particle" version of Molecular Dynamics simulations. If the maximum distance of particle-particle interaction is r_c – usually called the *cut-off* radius –, then a safely larger radius r_l is defined by $r_l = r_c + \Delta r$, where Δr is a positive term – usually called *skin* – controlled by the user. Then a list is generated and stored for each particle containing the index of all other particles

within the r_l list-radius. The simulation is performed using these neighbor-lists: when the interaction of a particle with its environment is determined, only the particles on the list are included in the calculation. If the interaction is short ranged – which is usually true in many simulations – then the physics is not affected by the exclusion of the far particles. However as the simulated system evolves, particles from outside of the list-radius r_l might get inside the interaction radius r_c. The danger is clear: an intruder particle not present on the neighbor-list crosses the safety zone Δr, and it can come close enough to interact with the particle owning the list. The simulator does not include the intruder in the interaction calculation, because it was not on the neighbor-list originally. The simulation at this point goes wrong, calculating an incorrect interaction. To prevent this, the neighbor-lists must be updated periodically at a given frquency. This refresh frequency f_r must be chosen according to the Δr safety zone. If Δr is large, we do not need to refresh the lists very frequently. However a large list-radius will result in long neighbor-lists, requiring larger memory storage. If Δr is small, the lists are short, but we have to update them frequently. There is no specific rule about chosing Δr and f_r, they must be fine tuned when running computer experiments.

Altough the neighbor-list method gives a significant speed-up, the whole method remains $O(N^2)$, because the lists are generated from scratch comparing every particle with every other one. If for example the lists are updated at every hundredth simulation step for a large particle system, then the neighbor-list method will be about a hundred times faster than the basic method, but it will still scale as $O(N^2)$ with the system size.

Can we refresh the neighbor-lists in a smarter and faster way? The answer is *yes*. The intruder particles do not come from very far, they are second or third neighbors, therefore they are on the neighbor-lists of the neighboring particles. This process is shown in Figure 2.7. On the left side one particular configuration of three particles

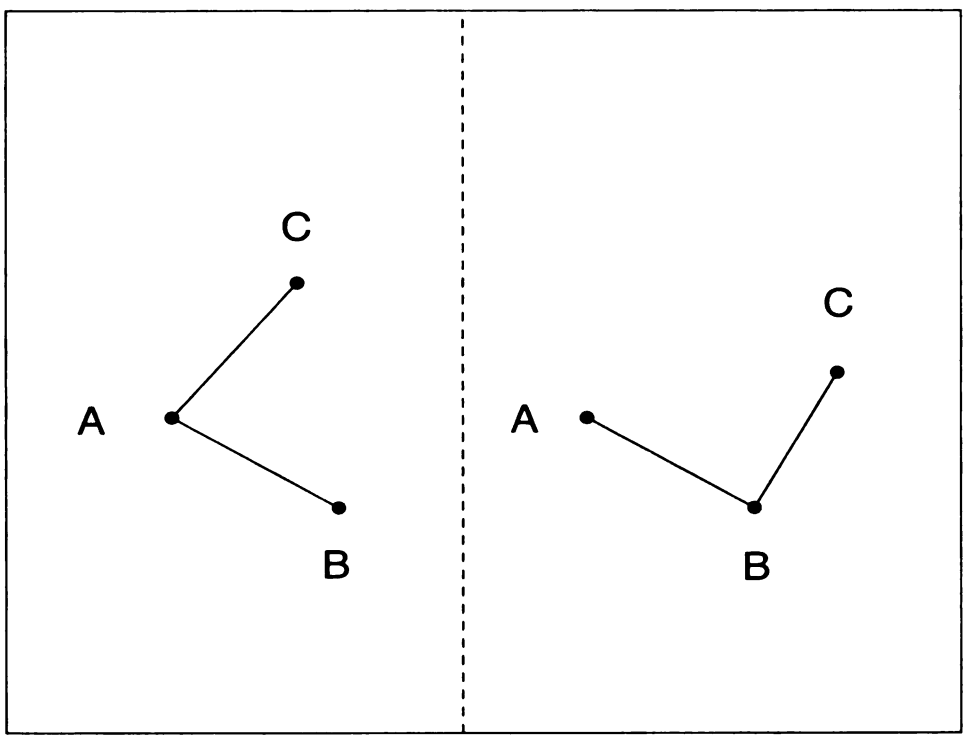

Figure 2.7: Particle C approaches particle B through particle A. The left panel shows the configuration before, the right panel after the simulation step. The solid lines represent the physical particle-particle interactions. Particles A and B remain within each other's interaction range, the particle C switches from the range of A to that of B. Updating of the lists is assumed to be due after the simulation step shown. The lists before the step describe the physical connections on the left. After the update, the lists must reflect the configuration on the right.

– A, B and C – is shown, on the right the same particles after one simulation step. Particles A and B did not move much, but particle C moved to right significantly in this example. How can we then quickly refresh the neighbor-lists? With *two* consecutive steps for each particle:

1. Scan the particles on the neighbor-list of the central particle. Keep only those which are still inside the list radius.

2. Scan the particles on the neighbor-lists of the particles on the central neighbor-list. Keep those which are inside the list radius.

The second step is nothing, but checking the next nearest neighbors on the connectivity graph. Unfortunately, the first step might modify the central neighbor-list, and therefore the conectivity graph as well, so in the second step not every particle is checked, some of them might have already been disconnected. We can understand this situation with the help of Figure 2.7. Before the step – on the left side – both particle B and C interact with A, therefore both particles are on A's list. B and C however are far enough from each other, they are not listed on each other's lists. After the step – on the right – particle C moved away from the neighborhood of A to the neighborhood of B. The connections on the figure represent the *physical* interactions. The graph connections are described by the neighbor-lists. They must contain the physical interaction connections, in order for the simulation to be correct. In optimal case the two graphs – the physical and the logical – coincide. Let us try to construct the logical graph representing the physical situation on the right from the graph on the left. On the left the physical and the logical graphs coincide, the simulation is correct up to this point. When the list of particle A is examined after the simulation step is taken, both B and C are on it. B stayed close, however C moved away, therefore C will be removed. Neither B nor C has neighbors other than A – in this example –, so no scan is performed for next nearest neighbors. When the list of B is scanned, only A is found on it, and it stays on it. The scan for the next nearest neighbors of B

cannot find C, because C was already removed from A's list. We have lost C in the logical space! In the physical space on the other hand C moved close enough to B to interact with it, therefore the simulation is not valid physically anymore. Remember, we want to avoid exhaustive – every particle against every particle – check, because we want to speed up our $O(N^2)$ algorithm to $O(N)$.

How can we fix the problem above? By using two copies of neighbor-lists for each particle: a *source* copy and a *target* copy of the lists. The new target lists are constructed based on the old source lists. No modification – for example disconnection – is made on the source lists during the construction! When the target lists are completed, the two lists switch their role: the target becomes the source that the simulation is based on. They are periodically alternated during the whole simulation. Altough the memory requirement for the lists is doubled now, this algorithm scales linearly with the size of the particle system. The performance of this algorithm will be given in the next chapter. The code is rather long and complicated for this method; we describe it only in words here. The scan in the first step – the scan of the central particle's own list – is straightforward: only the particles inside the list radius will make it to the new target list. The scans in the second step – the scans of the lists of the neighboring particles – are more complicated. Not only the geometrical condition must be checked, but also the database of the lists must remain consistent by not allowing a particle to enter into its own list or the same particle twice into any given list. The danger of the self-pointing particle comes from the following source. If particle B is on the list of particle A, then particle A is on the list of particle B as well: the neighbor's neighbor-list will always contain the particle currently under test. It is forbidden to enter a particle into its own list, because that will lock the particle in its current position – a particle's distance from itself is always zero. The energy calculation might become incorrect also using this zero distance. The source of a double entry is that a common neighbor of two particles will be on the list of

both of the particles, therefore the same particle found in the first scan will be found in the scans in the second step. A double entry does not cause particle clash, but the interaction energy will be calculated incorrectly: twice for the particle present on a list twice. By rejecting the double entries the lists remain logically consistent, and the simulation keeps running correctly.

2.4.3 Cell-list method by bitmanipulation

In the cell-list or domain-decomposition method [38] the simulation box is divided into smaller units called cells or domains. Then each particle is classified into a cell based on the particle's coordinates. For every cell a list is constructed containing all the particles within that cell. Obviously, a particle cannot be listed in more than one cell's list. The classification of the whole particle system is an $O(N)$ procedure, because it takes the same amount of time to classify any one of the particles, and every particle must be classified only once. The lists are associated with the cells, not with the particles as in the neighbor-list method. When the construction of the lists is finished, the simulation can start. The interaction of a particle with the other particles is calculated as follows: based on the particle's coordinates the cell, that it belongs to is determined. The particles on the list of this central cell and the surrounding cells are the candidates to interact with our original particle. In two dimensions 9, in three dimensions 27 cell-lists are scanned. Just as in the neighbor-list method, the cell-lists must be refreshed at a given frequency. The frequency must be chosen so that between two updates of the lists no particle can travel a distance equivalent to the side length of a cell. As we can see, both the list updates, and the simulation steps need $O(N)$ time, making the cell-list method a true linear method.

Now let us implement the cell-list method on our discrete lattice using bit manipulation. Let us assume we have a 1024 by 1024 sized discrete simulation box in two dimensions, and it is divided into 16 cells. The size of one cell is therefore 256 by 256

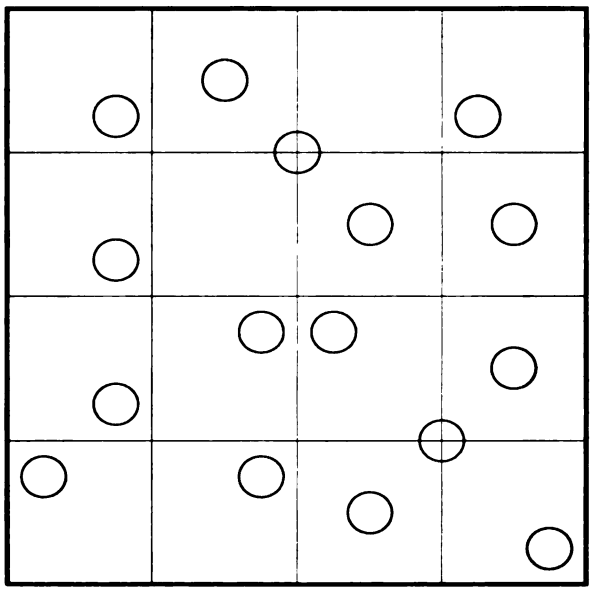

Figure 2.8: A two dimensional square shaped simulation box with randomly distributed particles. The box is divided into 16 cells. The side length of the box is assumed to be 1024, therefore the side of a cell is 256. Both the box size and the cell size must be exact powers of two in order to use bit manipulation in the cell list method. One complete coordinate is stored in 10 bits. The two highest bits locate the cells inside the box, the low eight bits give the coordinate within a cell.

lattice unit. See Figure 2.8. We will need the following data structures to implement the cell-list method:

```
INTEGER     NS                          01
INTEGER     NLOBS                       02
INTEGER     LCL                         03
PARAMETER   (LCL=50)                    04
INTEGER     LCLS                        05
PARAMETER   (LCLS=31)                   06
INTEGER     CL(0:LCL,0:LCLS,0:LCLS)     07
```

where **NS** holds the actual side length of the square shaped simulation box. **NLOBS** is

the *Number of LOw Bits in the Size* characterizing the cell. These variables receive their value from the parameter file given by the user, but for the sake of this example we initialize them inside the code according to the sizes shown in Figure 2.8:

```
NS    = 1024
NLOBS = 8
```

The size of the box is 1024, the size of the cell is 256. We need 8 bits to locate a lattice site within a cell, because $2^8 = 256$. In other words the variable `NLOBS` must be the base 2 logarithm of the cell size. A total of 10 bits is needed to locate a site inside the whole box – $2^{10} = 1024$ –, therefore the two high bits will locate the position of the cells within the simulation box. The parameter `LCL` containing 50 is the *Length of the Cell-Lists*. This length must be large enough to accomodate every particle inside a given cell. It depends on the size of the cells, the size of the particles and the density of the fluid. Furthermore it must be safely large to handle possible density fluctuations as well. Traditionally the cell-list method used *linked lists* [4, 32]. In that method only one pointer is associated with every cell and every particle. The cell's pointer contains the *head*, the label of the first particle found in that cell. Then the first particle points to the second, the second to the third and so on. The *tail*, the last particle points to a stop signal, a number which would not be a valid particle label for example -1. The advantage of this method is that it uses only the minimum required amount of memory, there is no need for overestimation against density fluctuations. The disadvantage is that the memory access of the linked lists is random in the memory of the computer, therefore it is not cache-friendly. The parameter `LCLS` containing 31 – an exact power of two minus one – gives the number of cell-lists to allocate in a given dimension. The three dimensional array `CL` is the *Cell List* itself. This array can be quite large in real simulations, in this example it is only 51 by 32 by 32 by 4 bytes, which is 208896 bytes in total. Because the elements of one list are accessed right after each other, the element index is the first one to

give efficient sequential memory access. The second and third indeces are the cell indeces. With one more important variable `MCLS = NCLS-1`, and a few work and loop variables - `ZX,ZY,WI,I,J` - the cell-list is generated with the following code:

```
DO J=0,MCLS                                    01
  DO I=0,MCLS                                  02
    CL(0,I,J) = 0                              03
  ENDDO                                        04
ENDDO                                          05
DO I=0,MATOM                                   06
  ZX = ISHFT(X(I),-NLOBS)                      07
  ZY = ISHFT(Y(I),-NLOBS)                      08
  CL(0,ZX,ZY) = CL(0,ZX,ZY) + 1                09
  WI = CL(0,ZX,ZY)                             10
  CL(WI,ZX,ZY) = I                             11
ENDDO                                          12
```

In the lines 1-5 the length of every cell-list is cleared to zero. In the lines 6-12 the cell-lists are constructed. The important lines are 7 and 8. The x and y coordinates of the ith particle are bit shifted to the right: the low 8 bits are shifted out, only the high 2 bits are kept in the lowest two places. These high bits of the coordinates identify the cell where the particle resides. Then the length of this cell-list is incremented by one in line 9 to register this new particle. This just incremented length value gives the next available empty storage place in the list. A working index `WI` is generated in line 10 to point to this next empty place. In line 11 the label of the ith particle - `I` - is stored at this place. Line 12 closes the loop over the atom index. In the case of continuous type simulation the cell-index calculation needs two or three floating-point operations per dimension. Here we did it with *one* bit shift, which takes one CPU cycle. The bit shift operators are extensively used inside the simulation loop also,

giving good performance. Performance data of this method will be given together with the other methods' performance.

2.4.4 Field-representation

What other alternatives do we have to represent and simulate particles besides the coordinates of the particles? In the Lattice-Gas method pointlike particles move and interact on the lattice. These particles are represented by bits *i.e.* numbers stored on the lattice. The dynamics of the system is simulated by manipulating these numbers according to certain rules as we saw. It is important to notice that in the Lattice-Gas method, the coordinates of the particles are not stored explicitly. Instead the points of an abstract discretized space are *flagged*, when they are occupied by particles. Furthermore there are no neighbor-lists or cell-lists to keep track of interactions. The collisions between particles are detected right away by checking the content of the targeted site on the lattice. The main computational advantage of this method is the simplicity of the operations when simulating the particles and detecting their interactions. The disadvantage is the large amount of memory required to store the simulated physical region. The disadvantage of the method from the physics point of view is that the solutions given by it are continuum type like the solutions to the Navier-Stokes equations. The method is particle based, but the results do not reflect this particle nature back. This contradiction is only apparent. The results are smooth; because the particles are pointlike on the scale of the lattice, they cannot be resolved by the lattice.

Let us therefore take a step further by allowing our particles to have structure seen by the lattice. Let the lattice points which are part of a particle have a value of 1, the background points outside the particle have a value of 0. See Figures 2.1 and 2.2. We can refer to these latter points as *vacuum* points. The vacuum points are not shown on the figures at all. On Figure 2.1 the black points form a "circular"

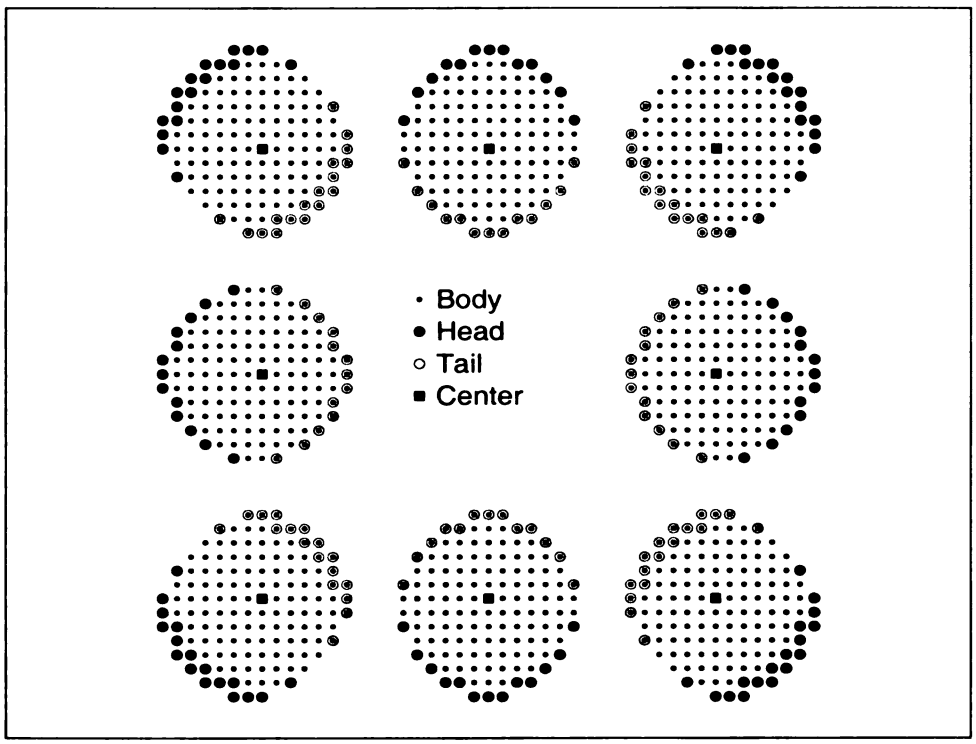

Figure 2.9: The eight possible movements of a circle on a two dimensional square lattice: two horizontal, two vertical and four diagonal. The values of the lattice sites marked by *Head* are incremented, the *Tail* sites are decremented to simulate particle motion. The value of the *Body* points remain the same. The *Center* of the particles are also marked. The radius of the circle is 6.1 lattice units.

step-function. When a particle moves on the lattice, it carries this step function with itself. The updating mechanism of this step function is rather simple, it is shown in Figure 2.9. There are four nearest neighbor – up, down, left, right – and four next nearest neighbor diagonal movements on the square lattice. When a particle – a circular shaped step function – is moved in a particular direction, only the lattice points in the direction of motion must be updated. The new lattice points joining the object are called *Head* points, the old points leaving the object are called *Tail* points, as shown on the figure. The values of the lattice sites at the Head points are incremented by 1, at the Tail points are decremented by 1, when the particle takes the

step. Every other points - the *Body* points - of the object are intact in this process, the lattice values are not modified at these points. The three dimensional case is perfectly analogous: lattice sites in the direction of the step are incremented, on the other side are decremented. The sites to be modified are the sites on the surface of the sphere.

Let us now connect the radius of our physical particles with the radius of this circular shaped step function on the lattice in the following way: if the radius of the physical particle is R, then the radius of the step function will be $2R$. What does this circle with radius $2R$ represent? It is the area forbidden for the centers of the other particles, the so called *excluded area* in two dimensions. In three dimensions it is a sphere with radius $2R$, and it is called *excluded volume.* By using the excluded area, the collision of two two dimensional particles is depicted in Figure 2.10. When two particles come close to each other, their excluded area starts to intersect. At this intersection the values of the lattice sites become 2. When the distance between the two centers is exactly $2R$, then the two objects are in contact, they are not allowed to come closer. In the excluded area picture the centers are not allowed to move to lattice sites with a value greater than *one.* In this two-body collision this value is 2. In the collision of more than two objects the intersection of the excluded areas has a value equal to the number of colliding objects. In two dimensions this can be maximum 4. It is important to notice that, as in the Lattice-Gas method, there are no lists of particle coordinates to keep track of interactions. Collisions between particles are detected directly from the lattice by checking the value of the lattice site targeted by the center of the initiating particle. The coordinates of the particles are stored as a single linear list only. We will use the coordinates to collect spatial distribution data for example, but the simulation is *not based* on particle-particle distance calculation. The other important property of this method is that the original shapes of the particles are restored when the particles leave each other after the collision. This is true for

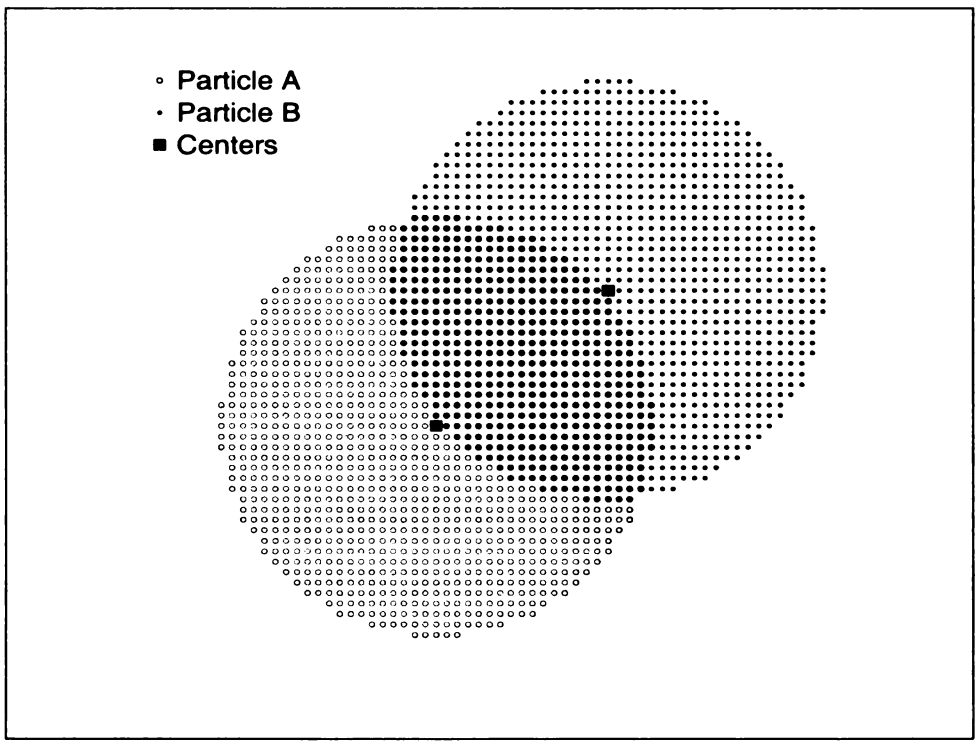

Figure 2.10: Collision of two particles on a two-dimensional square lattice. The radius of the particles is 10.1, the radius of the excluded area is 20.2. The centers are not allowed to penetrate the intersection of the two circles, where the value of the lattice sites were incremented to two.

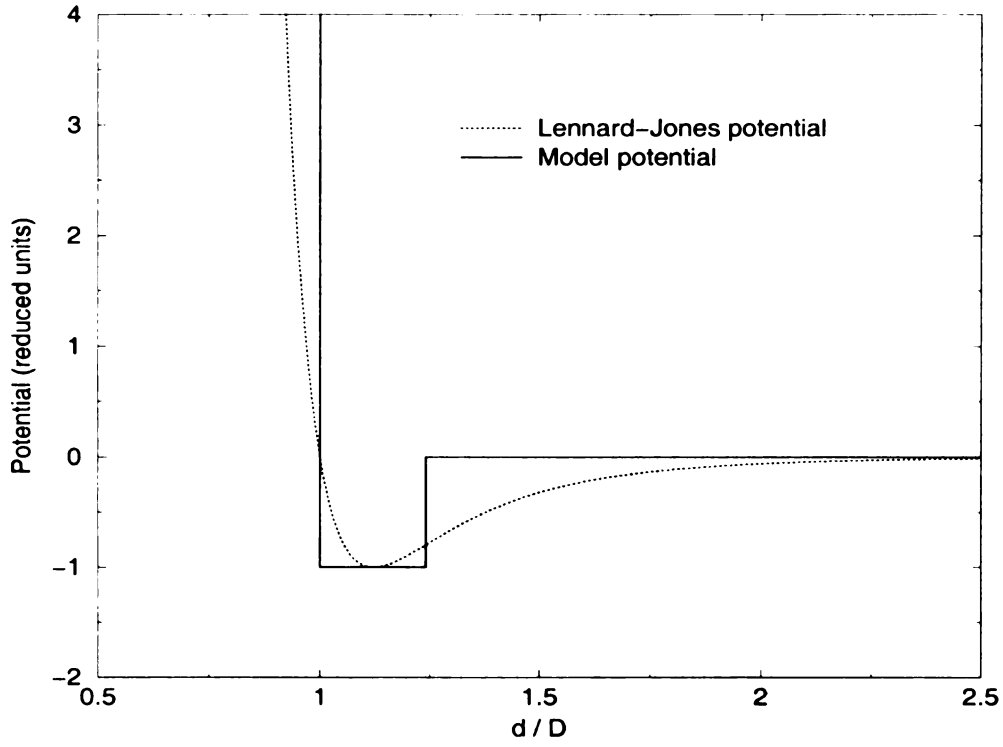

Figure 2.11: Modeling Lennard-Jones potential by a square shaped attractive well. The depth of the well is unity, the width is a variable parameter. The width is set to $\sqrt[6]{\frac{26}{7}} = 1.2444$ of the diameter on the graph, where the attractive force is the strongest. Generally it can be set to any other value in the simulation.

particle-obstacle collision also.

How can we model more complex interactions, for example

$$U(r) = 4\epsilon \left(\frac{D^{12}}{d^{12}} - \frac{D^6}{d^6} \right),$$

Lennard-Jones type interactions within this representation? First, let us model the Lennard-Jones interaction with a square shaped attractive potential. See Figure 2.11. The depth of the well is fixed to one in units of reduced potential energy, the width is controlled by us. On the figure the width is set to $\sqrt[6]{\frac{26}{7}} = 1.2444$ of the diameter. This is the distance where the Lennard-Jones attraction force is the strongest. The interaction between particles in this representation is shown in Figure 2.12. The particle on the right is inside the potential well of the central particle. It is allowed

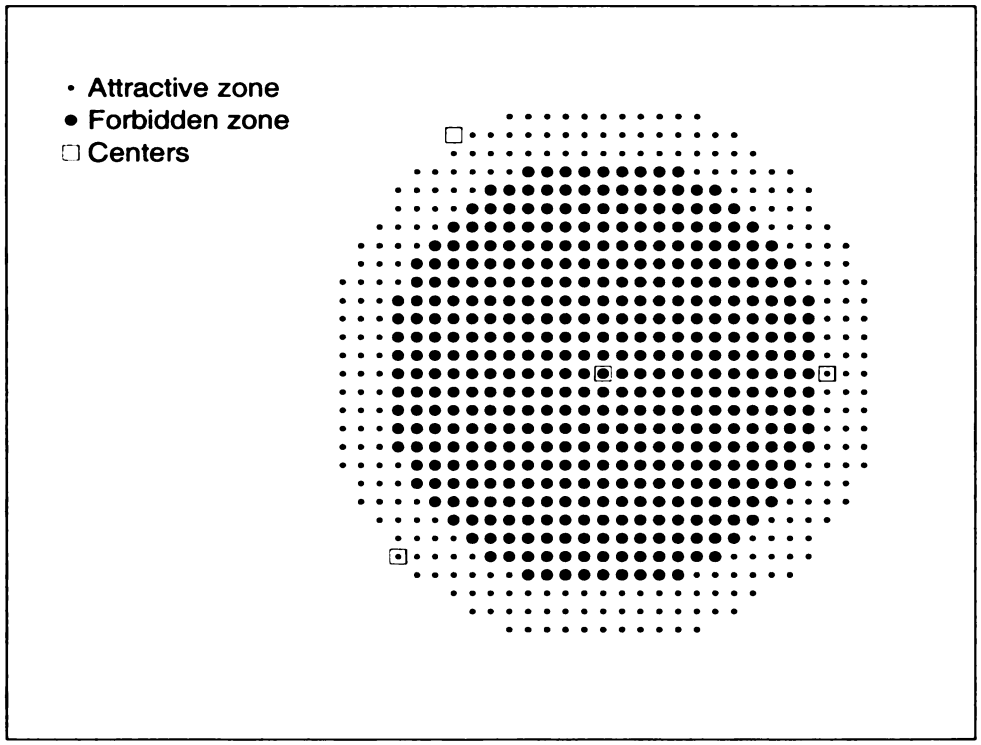

Figure 2.12: The particle in the center interacts with three other particles. The forbidden excluded area and the attractive zone is shown only for the central particle for the sake of clarity, but the other three particles have the same zones around them also. The centers are represented by empty squares. The radius of the excluded area is 12.0 lattice unit, the range of the attraction is 15.0 to approximate the attraction range of the Lennard-Jones potential.

to move freely up, down and right, but it is not allowed to move to left because of the hard core repulsion of the central particle. The particle on the left and up can take any steps. If it moves down or right it will fall inside the attractive well lowering the total energy. The particle on the left and down is just inside the potential well. It can move freely to right and up. When it tries to move to left or down its step will be accepted with the $exp(-1/T)$ Metropolis probability. The Boltzmann constant k is set to 1 for computational simplicity. The depth of the potential well is also 1. The strength of the interaction is therefore controlled by the temperature only, but without any loss of generality. For computational efficiency the Metropolis acceptance function is precalculated and tabulated for the actual temperature before the simulation begins. Due to the discreteness of the energy, these probability values can be easily looked up. This technique is commonly used in simulations of spin models. For continuous type simulations the Boltzmann factor has to be calculated during the run. Because the exponential function is one of the biggest consumer of the CPU time, we save large amounts of computation compared to continuous type models [40].

How can we implement the field-representation method computationally? First, we need the underlying lattice. In two dimensions:

```
INTEGER   LS
PARAMETER (LS=8191)
INTEGER*1 F(0:LS,0:LS)
INTEGER   NS
```

allocates an 8192 by 8192 square lattice `F` – called *Field* – with one byte per site, totaling 64 megabytes. The maximum value of the field is 4 when four particles touch each other in a close packed manner. Therefore 3 bits per site would be enough to store these values. However we might need to represent other geometrical objects on the lattice beside the particles, so it is useful to have extra bits. In three dimensions we need to store even higher values for intersections. We could break up a byte

among several lattice sites, but that would make the programming extremely complex. Therefore even if some memory is wasted, we use eight bits per lattice site in both two and three dimensions. In three dimensions usually 256^3 sized lattices were used, but 512 by 512 by 256 size was often reached in the case of pure hard core interaction. The allocation of the three dimensional lattice is analogous to the two dimensional one shown above. This lattice stores the excluded area or volume of the particles. Another lattice is allocated with the same size to store the interaction field of the particles and possible external potentials. These two lattices – the geometry and the interaction lattices – are kept separately and used differently. When a particle moves, first the geometrical lattice is checked for conflicts. If the step is not allowed – collision with another particle or with some obstacle – no further action is taken, the simulation takes the next particle. If the step is allowed geometrically, then the integer energy difference is calculated between the new and the old positions, and the step is taken with the Metropolis acceptance probability looked up from the table.

The shape of the particles, the head and tail points shown in Figure 2.9 are stored in tables. The potential field of the particle, the head and tail points for this field stored and handled analogously. Why do we waste large amount of memory to store the underlying lattices, the shapes of the particles and the interactions? Can't we simply use coordinate based methods? We can, of course. But the field-representation method will be able to handle non-circular particle or interaction shapes, and arbitrary background geometry or potential easily, if necessary. Non-circular shapes will be used in the next section. Furthermore, when random initial configurations of the fluid are generated, the field-representation method gives superior performance compared to the coordinate based methods. The performance of the method will be presented in the next chapter.

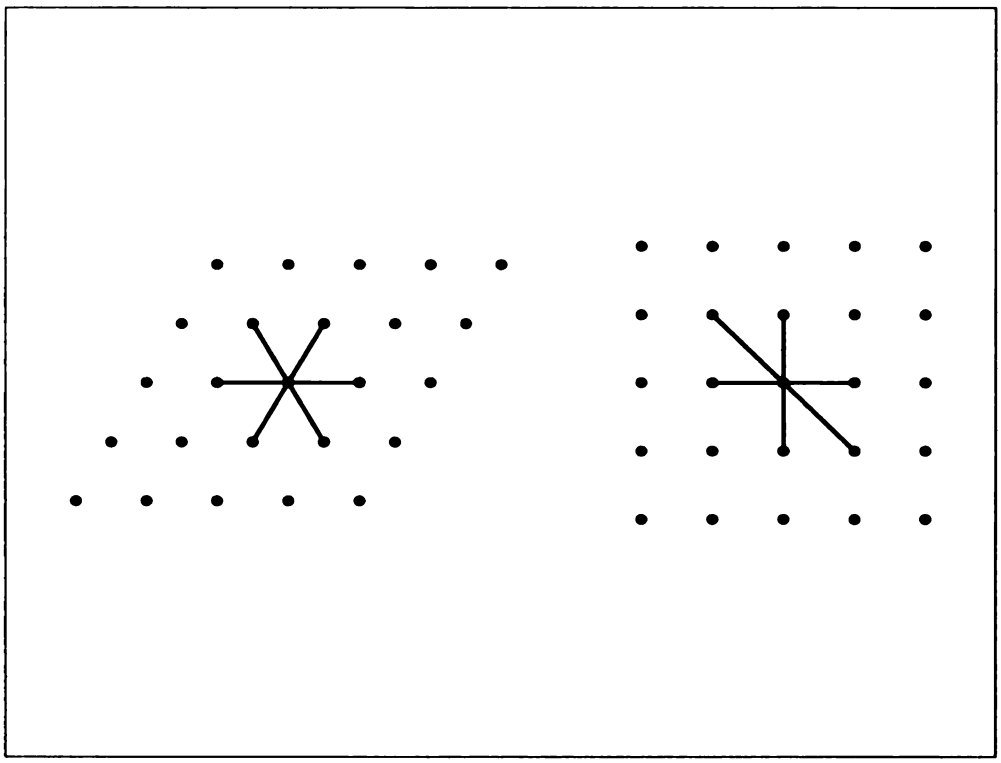

Figure 2.13: The triangular lattice on the left is sheared to the square lattice on the right. The nearest neighbors on the triangular lattice are connected to the central site to show the shape of the neighborhood before and after the transformation.

2.4.5 Techniques for the triangular lattice

The triangular lattice has advantages over the square lattice as we have seen in the case of the Lattice-Gas. On the other hand the triangular lattice is often treated in simulations as the naughty stepbrother of the square lattice. What is the simplest way to represent the triangular lattice and perform calculations on it? The triangular lattice can be sheared to form a square lattice. See Figure 2.13. The hexagonal neighborhood on the triangular lattice becomes a four nearest plus two opposite next nearest neighbor connectivity. Circles on the triangular lattice become ellipses on the square lattice. See Figure 2.14. If we use the field-representation method discussed in the previous section to simulate particles on the triangular lattice, we just have to use the elliptical particle shapes and the two additional next nearest neighbor steps. The

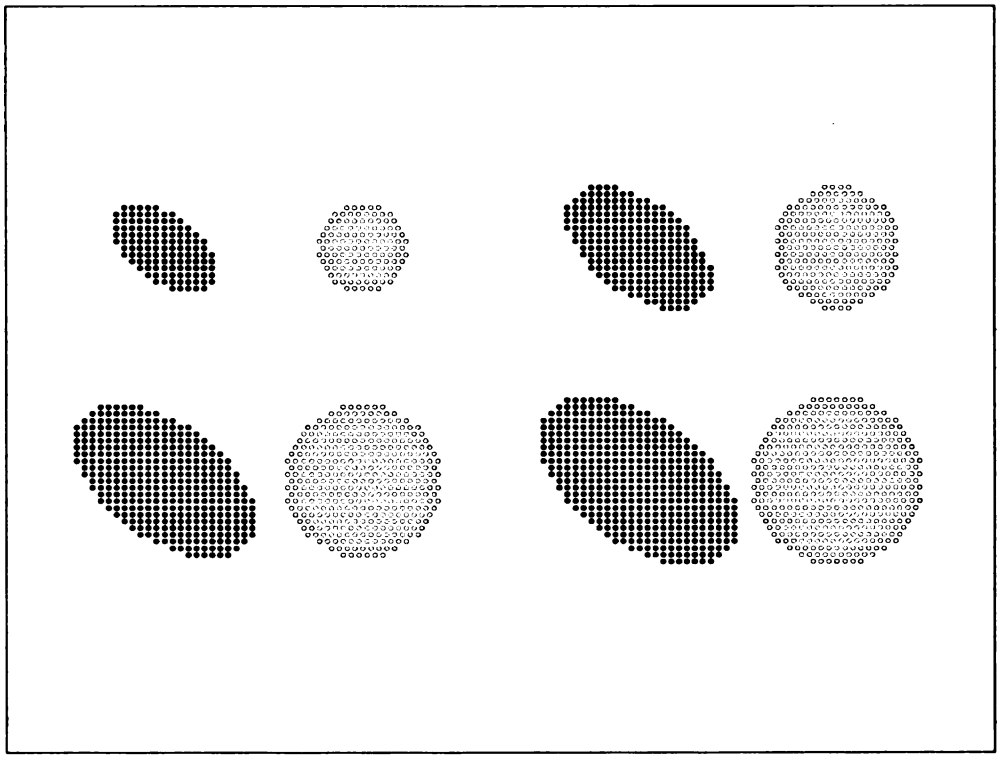

Figure 2.14: Four circles on the triangular lattice are depicted by small empty circles. The radii of the circles are 6.0, 8.0, 10.0, 10.9 lattice units respectively. Then the underlying triangular lattice is sheared to a square lattice. The circles are transformed to ellipses shown by filled circles.

program originally developed for the square lattice will work fine for the triangular lattice; modifications are required only at the data level: number of allowed steps, step directions, shapetables, head and tail points.

When using coordinate based methods on the triangular lattice, the underlying arithmetic is switched from integer to floating point type, because one of the lattice vectors has non-integer components:

$$\begin{aligned} \vec{l_1} &= (1,0), \\ \vec{l_2} &= \left(\frac{1}{2}, \frac{\sqrt{3}}{2}\right). \end{aligned}$$

In particle simulations usually *distance squares* are calculated and distance calculations are avoided whenever it is possible due to the large cost of the square root calculation. The distance square on the triangular lattice is:

$$ds^2 = dx^2 + dxdy + dy^2.$$

Therefore it can be calculated from the *integer* indices on the lattice without using the non-integer components of lattice vectors at all. To be precise, the distance square has this nice form exactly because of the special components of the second lattice vector. Hence we need very little modification in our programs to switch from a square lattice to a triangular one. Whenever there is a distance square calculation like:

```
D2 = DX*DX + DY*DY
```

in our program, it should be modified to:

```
D2 = DX*DX + DX*DY + DY*DY
```

or to:

```
D2 = DX*(DX + DY) + DY*DY
```

to be more efficient by saving one multiplication. Remember, all the variables are integer type in the examples above.

Another difference between square and triangular lattices we discuss is the *minimum image convention.* In particle simulations periodic boundary conditions are used in most of the cases. The simulation box is surrounded by its own copies in every direction, and every particle has images in those boxes. When we determine the interactions between the particles, every particle is assumed to interact with the other particles only once and at the shortest distance. When the difference vectors:

$$\vec{d}_{ij} = \vec{r}_i - \vec{r}_j$$

are calculated, the shortest one has to be found for every pair of particles. To understand this procedure, we place the origin – the bottom left corner – of our simulation box at $\vec{r}_j$, the position of the jth particle. See the left side of Figure 2.15. Due to the periodic boundary condition, the other three corners will be the images of the origin and therefore $\vec{r}_j$ as well. The ith particle can be anywhere inside the box. Depending which quarter it resides in, it will be closest to the corner of that quarter. Do we need to calculate the distances from the corners in order to decide which one is the closest? Fortunately not. Due to the rectangular shape of the simulation box, we perform the minimum image correction on the coordinates of the difference vector first, and then we need to calculate the distance – which will be the shortest – only once. The distance correction function is shown in Figure 2.16 for a box of size 16. Altough the calculation of this function on the fly is not very complicated, it is precalculated and stored in a look-up table for efficiency:

```
INTEGER DCTBL(-LS:LS)
NSP2 = NS / 2
DO I = -NSP2,NSP2
  DCTBL(I) = I
```

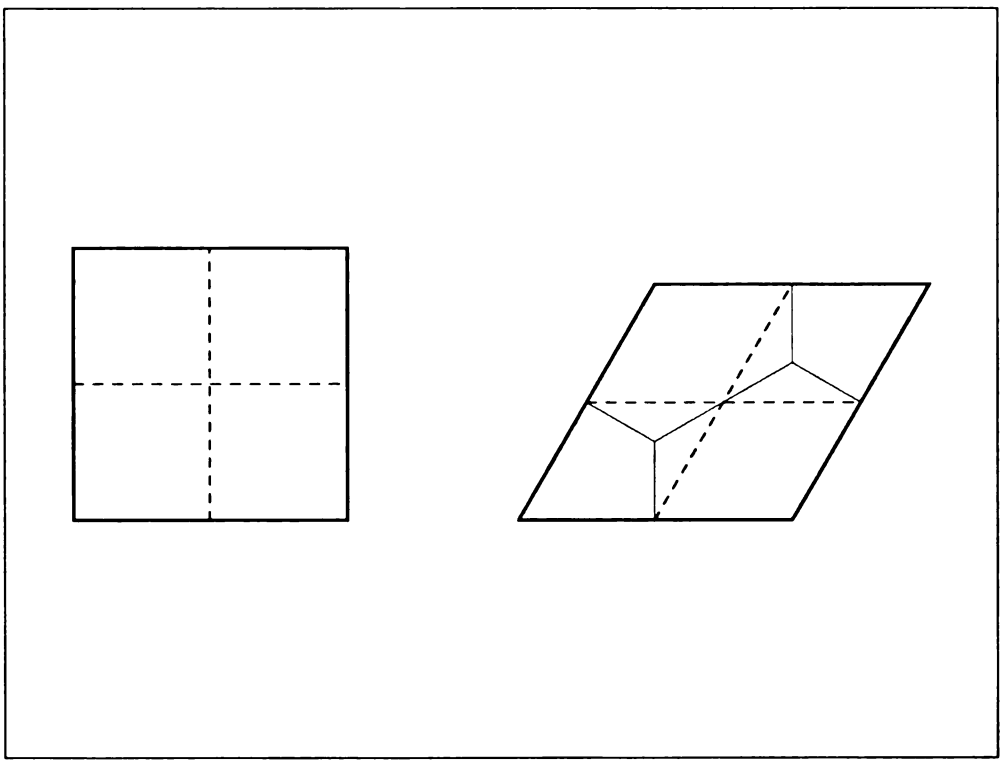

Figure 2.15: Minimum image convention for square and triangular simulation boxes. On the left, the square box is divided into four quarter squares. Particles in a given quarter are closest to the corner of that quarter. The square lattice on the left is sheared to triangular one on the right. The dashed lines correspond to the boundaries between the quaters from the square lattice. The continuous lines separate the correct honeycomb shaped regions – the Wigner-Seitz cells – closest to the corners.

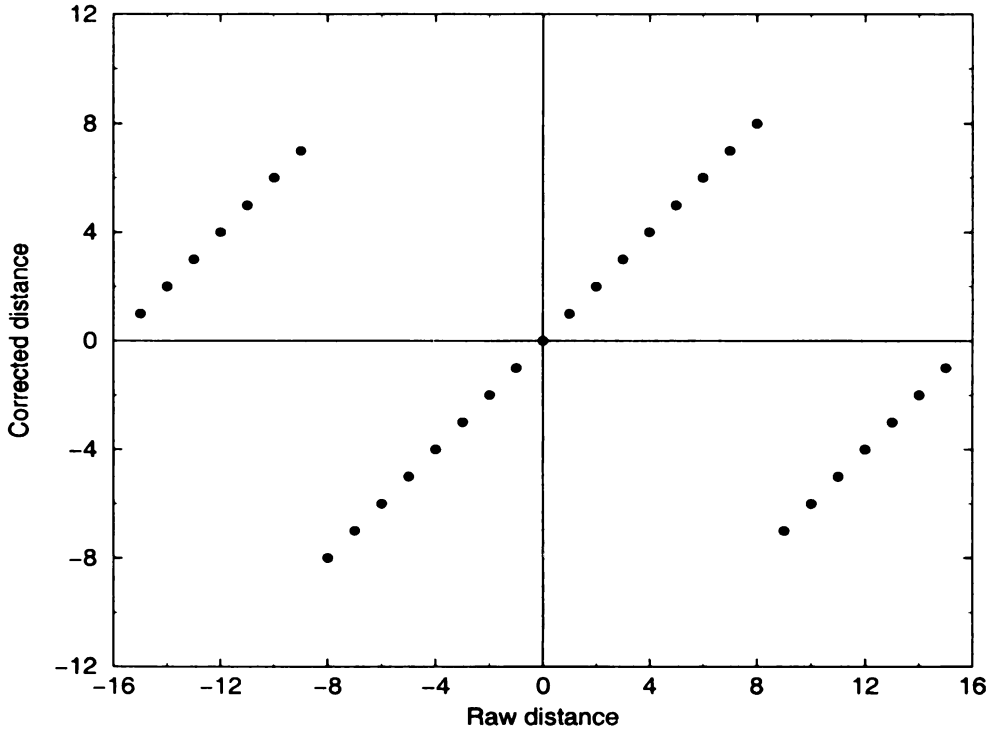

Figure 2.16: The distance correction function for a square simulation box of size 16. The largest distance is 8, half of the size of the box due to the periodic boundary condition. The function is precalculated and tabulated for computational efficiency.

```
ENDDO
DO I = (NSP2+1),MS
  DCTBL( I) = I - NS
  DCTBL(-I) = NS - I
ENDDO
```

Using this *Distance Correction Table*, the calculation of the shortest distance between the ith and jth particle is:

```
DX = DCTBL(X(I) - X(J))
DY = DCTBL(Y(I) - Y(J))
D2 = DX*DX + DY*DY
```

If the shape of the simulation box is not square but rectangular, different distance correction tables must be set up and used for the different sides.

Will this distance correction method work for triangular lattice too? Unfortunately not perfectly, only approximately. As we can see on the right side of Figure 2.15, the dashed lines sheared from the square shaped box on the left do not coincide with the continuous lines separating the honeycomb shaped regions, where the distance is minimal to the corners of the simulation box. For the correct calculation – corresponding to the honeycomb shaped Wigner-Seitz cell – the four different distances from each corner must be calculated in order to decide which one is the shortest:

$$\begin{aligned} d_1^2 &= x^2 + xy + y^2, \\ d_2^2 &= d_1^2 + n_x(n_x - 2x - y), \\ d_3^2 &= d_1^2 + n_y(n_y - 2y - x), \\ d_4^2 &= d_2^2 + d_3^2 + n_x n_y, \end{aligned}$$

where n_x and n_y are the sizes of the box in general case. This is a rather complicated calculation compared to the distance correction in the case of the square lattice.

However, if the simulation box is large enough compared to the range of interaction between the particles, the missclassification of the particles around the dashed and the continuous lines does not make any difference from the physics point of view. Therefore the image correction method for the square box can be used for the rhombus box on the triangular lattice.

2.5 Results

In this section we study the behavior of interacting many-particle systems by applying our simulation methods to different situations. Performance data will be given first, then specific physical results will be presented. The codes were executed mainly on two computers – *rigid3* and *rigid4* – on our CMPT cluster. Both of these machines are DEC Alpha computers with a Digital Unix operating system and a FORTRAN-77 compiler installed on them. Rigid3 operates at 266 MHz, rigid4 at 433 MHz. Execution times presented in the next sections were usually measured on rigid4. To convert these times for rigid3, we can simply multiply the measured CPU times by a factor of 1.6, the frequency ratio of the two machines, because they are very similar in their hardware architecture. Smaller test runs were executed on several other machines, mainly on Silicon Graphics and Sun computers in the Chemistry and Physics departments, but no extensive performance data were recorded on these computers.

2.5.1 Simulation performance

In our simulations we need to generate random initial configurations of fluid particles and then simulate them efficiently. In this section we study the behavior and performance of coordinate and field-representation methods. Figure 2.17 shows a configuration of 80,000 particles randomly dropped to a square simulation box of size

8192 by 8192. Figure 2.18 shows the bottom left 1024 by 1024 corner of the same configuration containing about 1250 particles. The radius of the particles is 12 lattice units. Figure 2.19 shows a configuration of 11,111 particles in a 512 by 512 by 256 simulation box, and Figure 2.20 shows a configuration of 2692 particles in a 256^3 box. The radius of the particles in the three dimensional simulations was 8 lattice units. How can we generate large configurations like these efficiently? We can use coordinate based methods with neighbor lists. Unfortunately when we try to drop a new particle into the simulation box, this new particle must be checked against *every* other particles already in the box, because the new particle does not have a neighbor-list yet. No matter which neighbor-list scheme – the basic or the fast – is used to refresh the neighbor-lists, we have to construct the list for the new particle first. Therefore this method will scale *quadratically* with the number of particles. However it becomes even slower, when a higher fluid density is reached, because it is more and more probable that the new particle will clash with another one already in the box. In this case we will try to place the new particle at a different position. The execution times for configurations of different sizes are shown in Table 2.1. Eight different particle systems were generated with sizes from 10 to 75 thousands inside an 8192 by 8192 box by two different methods: neighbor-list and field-representation. Why is the neighbor-list based method so slow relative to the field-representation method? The neighbor-list method performs as $O(N^2)$ at low densities as we just discussed. The field-representation method however needs only $O(N)$ time in the same density range, and runs faster initially as well. In the field-representation the excluded area of the particles alrady in the box are stored on the lattice. When a new particle comes at a random lattice position, only this position must be checked. If the value of the targeted lattice site is zero – vacuum point –, then the new particle is allowed to occupy this site. If the value is greater than zero, the new particle clashes with one or more particles, therefore we need to look for another position.

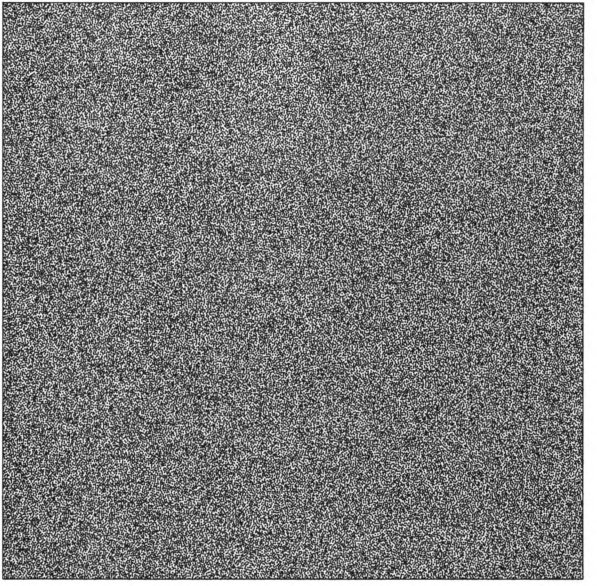

Figure 2.17: Random configuration of a two dimensional fluid containig 80,000 circular particles. The radius of a particle is 12 lattice units. The underlying lattice is an 8192 by 8192 square lattice. This configuration was generated in 45 seconds. The area density of the fluid is 0.539.

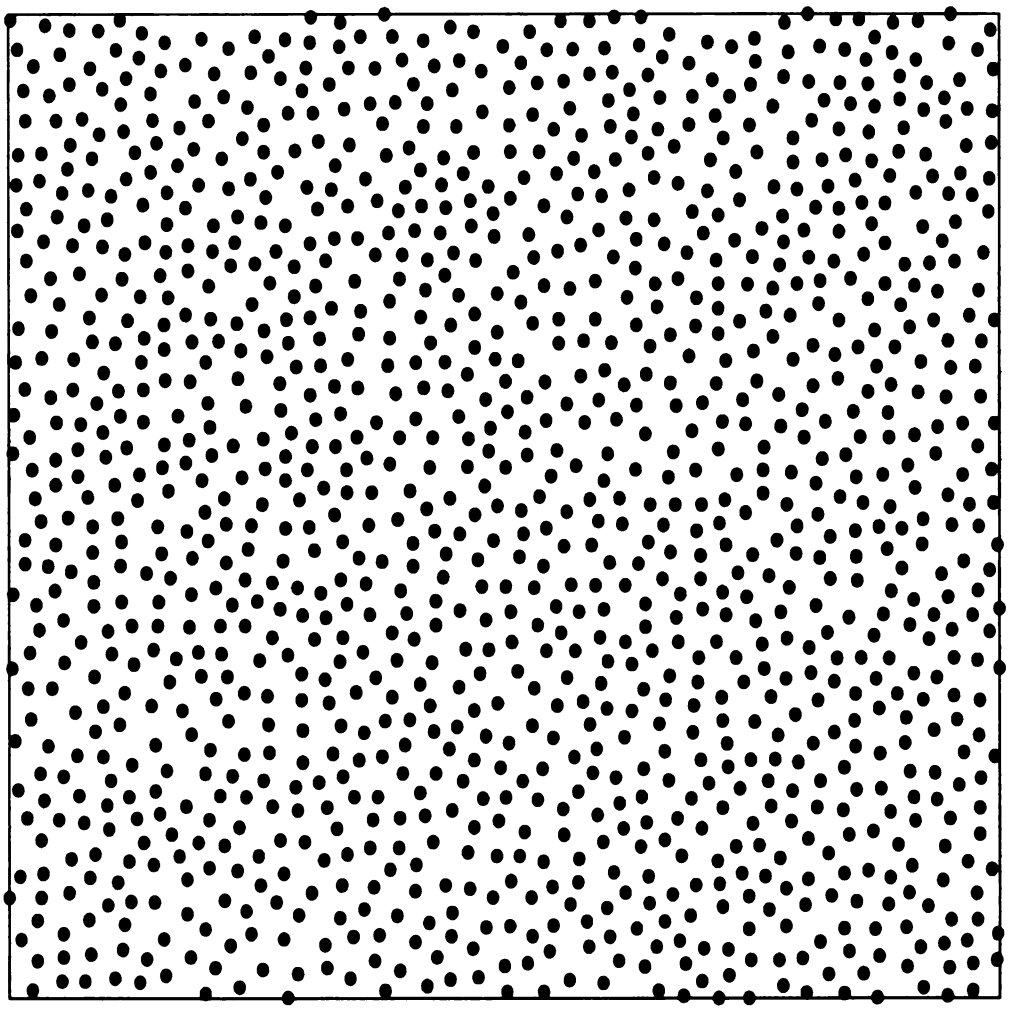

Figure 2.18: One 64th of the configuration from the previous figure. The size of the lattice is 1024 by 1024, and the number of the particles is about 1250. The particles are shown with a radius smaller than their original radius for better view. At this density many of the particles are in contact with each other.

Figure 2.19: Random configuration of a three dimensional fluid containig 11,111 spherical particles. The radius of a particle is 8 lattice units. The particles are shown with a smaller radius – 6 lattice units – for better view. The underlying lattice is a 512 by 512 by 256 cubic lattice. The volume density is 0.355. This configuration was generated in 33 seconds.

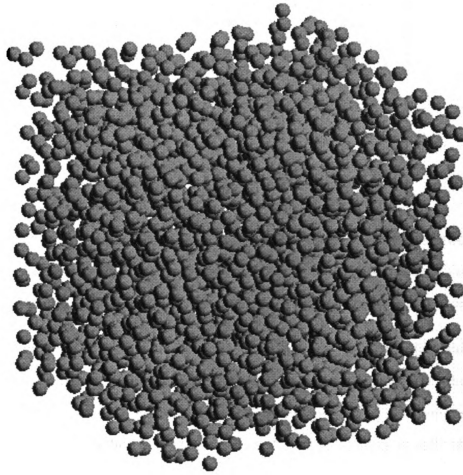

Figure 2.20: Random configuration of a three dimensional fluid containig 2,692 spherical particles. The radius of a particle is 8 lattice units. The particles are shown with a smaller radius – 6 lattice units – for better view. The underlying lattice is a 256^3 cubic lattice. The volume density is 0.344. This configuration is independent of the one on the previous figure. This system was generated in 5.5 seconds.

System size	Neighbor-list (s)	Field-repr. (s)
10,000	11.9	2.0
20,000	49.7	3.0
30,000	118.7	4.0
40,000	230.8	5.1
50,000	423.9	6.1
60,000	826.7	7.3
70,000	2484.6	8.8
75,000	7436.0	11.1

Table 2.1: Two dimensional random initial configuration generation by two different methods: neighbor-list and field-representation. The number of particles ranges from 10 to 75 thousand. The CPU times were measured on rigid4 and are given in seconds.

The collision check is done in a *unit* time in the field method, therefore this method scales *linearly* with the number of particles at low densities. However, this method will also slow down as the density of the fluid increases. Another advantage of the field-representation method is that the lattice sites can be systematically checked for availability, therefore we can know *exactly* when a configuration is jammed *i.e.* cannot accept more particles. The data from Table 2.1 is depicted in Figure 2.21. The cost we pay for this high performance is, of course, a large amount of memory. In the field method the 8192 by 8192 lattice is stored in 64 megabytes, one byte per site. On the other hand the neighbor-lists of the coordinate based method fit in less than 10 megabytes easily even for the largest particle systems.

It is possible to use the cell-list method when generating initial conditions. This method would be somewhere between the field-representation and the neighbor-list method. It would scale linearly with the number of particles, but it would not be as fast as the field method, because the collision checks would be much more complicated.

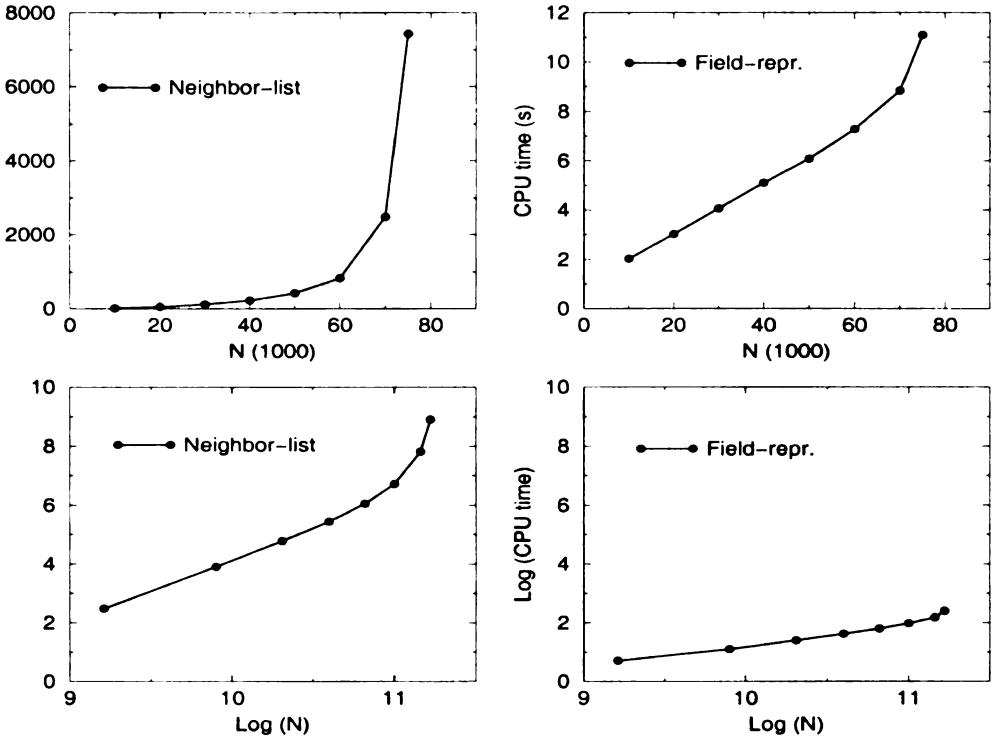

Figure 2.21: CPU times of neighbor-list and field-representation methods as a function of system size. The top two graphs are linear on both axes but with different vertical scale, both measured in seconds. The bottom two graphs are log-log plots of the top ones. The graphs in the left column show the times for the neighbor-list method. The graphs for the field method are on the right. The neighbor-list method requires $O(N^2)$, the field method $O(N)$ time at low densities. At higher densities both methods slow down.

The configuration in Figure 2.17 was generated by the field method in only 45 seconds. The neighbor-list method however was not able to produce this configuration within a time limit, the job was killed after 24 hours without finishing. The two simulations used the *same* physical parameters and the *same* seed. Altough they are two different algorithms, they use the random numbers in the *same* way, therefore they were working on the exact *same* configuration. This is an important detail, because this density was close to the highest achievable density, where configurations start to become impossible to reach. The field-representation method has successfully placed all 80,000 particles, therefore this particular configuration definitely exist.

The area density of the configuration in Figure 2.17 is $\rho_A = 0.539$, the volume densities of the configurations in Figure 2.19 and 2.20 are $\rho_V = 0.355$ and 0.344 respectively. These densities are close to the so called *jamming coverage* Θ_J in two and three dimensions. This problem is known as the *parking problem* or *Random Sequential Adsorption* (RSA) [41, 42]. Objects with well defined shape and size – for example identical circles in two, spheres in three dimensions – are dropped in a box. A new object cannot intersect the old ones already in the box. Several other versions and modifications of the problem were developed with applications ranging from mathematics to chemistry [43]. The jamming coverage for circles in two dimensions is $\Theta_{J,2D} = 0.5472 \pm 0.0002$, and for spheres in three dimensions $\Theta_{J,3D} = 0.384 \pm 0.001$. To reach higher densities of liquid states, the particle configuration must be compressed externally or some other artificial technique should be applied.

After the initial configuration is generated, we have to simulate the ensemble. We can use any of the algorithms described in the previous sections. The performance of these algorithms for two dimensional particle systems with hard core repulsion is shown in Table 2.2. The performance of the traditional neighbor-list method drops rather quickly. The other methods maintain an almost constant performance as the number of particles increases. There is a slow decrease however. It is due to the

System size	NL	FNL	CL	FR
1,100	393	397	216	48
4,400	210	386	210	48
17,600	70	344	204	48
70,400	20	337	199	47

Table 2.2: Performance of the different simulation algorithms in two dimensions. Four different particle systems were simulated by each method. The number of the particles is shown in the left column. The radius of the particles was 12.0 lattice units. The size of the simulation boxes were 1024^2, 2048^2, 4096^2 and 8192^2 respectively. The area density was kept constant at 0.474. The four methods are: Neighbor-List (NL), Fast Neighbor-List (FNL), Cell-List (CL) and Field-Representation (FR). The numbers give the performance in 1000 atomsteps per second on the rigid4 machine.

fact that as the system size incereses, the data structure describing it is more spread in the computer memory. Consecutive memory accesses are farther from each other, therefore the memory caching mechanism operates less and less efficiently. This effect is even more pronounced, when the radius dependence of the performance is studied for the field-representation method. If the radius of the particles and both sides of the simulation box are halved, while the total number of the particles and hence the density remains the same, the preformance should double, because the head and tail points of the circular step function are on the perimeter, therefore their number is also halved. Half as many operations are needed to be performed on the lattice, therefore it should take half as much time. However since the simulation box is also smaller, the lattice sites in consecutive columns are closer to each other in the memory. The memory accesses will be more local, the caching will work more efficiently. See Table 2.3. As we can see, when the radius is halved, the preformance increases to more than twice, and the same simulation runs faster in a smaller piece of memory. The

Radius	1024^2	8192^2
12.0	62	48
6.0	254	151
3.0	688	579

Table 2.3: Performance of the field-representation method as a function of particle radius in two dimensions. The radius of the particles is in the left column. 1100 particles with radius 12.0 were simulated in a 1024^2 simulation box. Then the radius and both sides of the box was halved to maintain the same density of the particle system. The simulations needed a 1024^2 lattice or smaller, but they were carried out in a corner of 1024^2 and 8192^2 lattices allocated in the memory to study the preformance of the simulation on the size of the memory used. The performance data is given in the units of 1000 atomsteps per second.

conclusion is therefore the following: for optimal performance only the necessary amount of memory should be allocated.

In three dimensions the coordinate based methods slow down by a factor of 3 to 5 due to the increased number of neighbors in three dimensions. The field-representation method slows down even more, because the number of head and tail points now lie on the surface of the spherical particles, proportional in number to the square of the radius. However the field method remains the only efficient way to generate random initial configurations of fluids. Therefore a combination of these two methods are used: the initial configurations are produced by the field-representation method, and then they are read by the coordiante based methods for further simulation.

2.5.2 Pair-distribution function of hard core systems

The liquid phase is probably the most interesting phase structurally compared to the gas or solid phases [44]. It is clearly distinct from them in its Pair Distribution Function (PDF). This function describes the probability of finding two particles at a given distance from each other. In two dimensions:

$$2\pi r \rho g(r) dr,$$

in three dimensions:

$$4\pi r^2 \rho g(r) dr$$

is the probability to find a particle in the $(r, r + dr)$ distance interval from another particle, where ρ is the number density of the particles: N/A in two and N/V in three dimensions. In simulations the pair distribution function is usually constructed by using the form:

$$g(\vec{r}) = \frac{2V}{N^2} \langle \sum_{i<j} \delta(\vec{r} - \vec{r}_{ij}) \rangle.$$

Every particle-particle distance is measured once ($i < j$) and stored in the distribution. The $\langle\rangle$ averaging can be time or ensemble averaging. The factor in the front gives the proper normalization. When the physical system is isotropic – like the bulk of a liquid –, the pair distribution function $g(\vec{r})$ will depend only on the particle-particle distance but not the orientation, therefore it simplifies from $g(\vec{r})$ to $g(r)$, where $r = |\vec{r}|$. According to the precise nomenclature $g(\vec{r})$ should be called the pair distribution function, and $g(r)$ the radial distribution function. However, very often the acronym PDF is used for both functions. In this work only isotropic systems are studied; we will refer to $g(r)$ as PDF, even if it is not perfectly correct.

Now let us compute the pair distribution function for a simple system: 1100 atoms are placed on a 1024 by 1024 square lattice with radius 12.0 lattice units. They perform a random walk and they interact with each other via hard core repulsion. Every atom takes 1000 steps before the data collection starts. Then the distribution

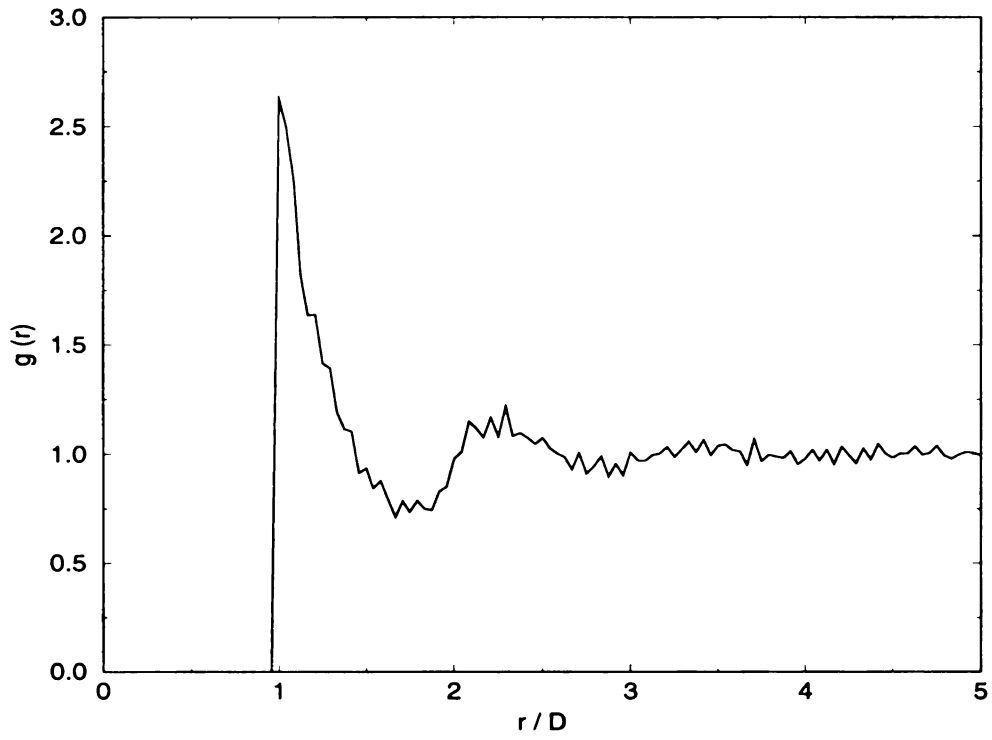

Figure 2.22: The pair distribution function for a two dimensional hard core particle system. The size of the simulation box is 1024 by 1024 and it contains 1100 particles with radius of 12.0 lattice units. The interparticle distance r is normalized with the particle diameter D. The area density of the liquid is 0.474. The data were collected for 5000 time steps after 1000 equilibration time steps. The main wavy features can be seen well, but the function is rather noisy.

is collected for 5000 steps. The final result is shown in Figure 2.22. Altough the main wavy nature of the pair distribution function clearly shows up, it looks pretty noisy. Maybe the particle system was not large enough, or the data collection period was not long enough. It turns out that increasing the system size or the simulation time does not remove, or does not even reduce the noise: it stays persistently in size and in pattern! What kind of noise is this then, and how can we remove it? It is not random noise, it is a lattice effect. When the PDF is collected, the particles are counted in circular or spherical shells. The underlying square, triangular or cubic lattice is not

Shell	Square	Triangular	Cubic
(0,1)	1	1	1
(1,2)	8	12	26
(2,3)	16	18	66
(3,4)	20	24	158
(4,5)	24	30	234
(5,6)	40	36	410
(6,7)	36	48	470
(7,8)	48	66	738
(8,9)	56	60	866
(9,10)	56	66	1170

Table 2.4: The number of lattice points inside the first ten shells for square, triangular and cubic lattices. The shells (5,6) and (6,7) for the square lattice are shown in Figure 2.23. The fluctuations in two dimensions can be so large, that the functions are not even monotone.

compatible with the symmetry of the polar or spherical coordinate systems. Therefore the number of lattice points within a shell is not exactly proportional to the area or the volume of the shells. Some shells are under, some shells are overrepresented. See Figure 2.23. By counting the lattice points inside the shells, the noise can be calculated and removed. The number of lattice points inside the shells for our three different lattices are shown in Table 2.4. If a lattice point is exactly on the boundary, then it is counted into the outer shell. However this convention is arbitrary, it just has to be used consistently. For example the points with coordinates (3,4) and (4,3) in the first quadrant of the square lattice and their negative combinations are *exactly* on the boundary in Figure 2.23 because the group of integer numbers (3,4,5) is a Pythagorean triplet. The triangular and the cubic lattices have other combinations of

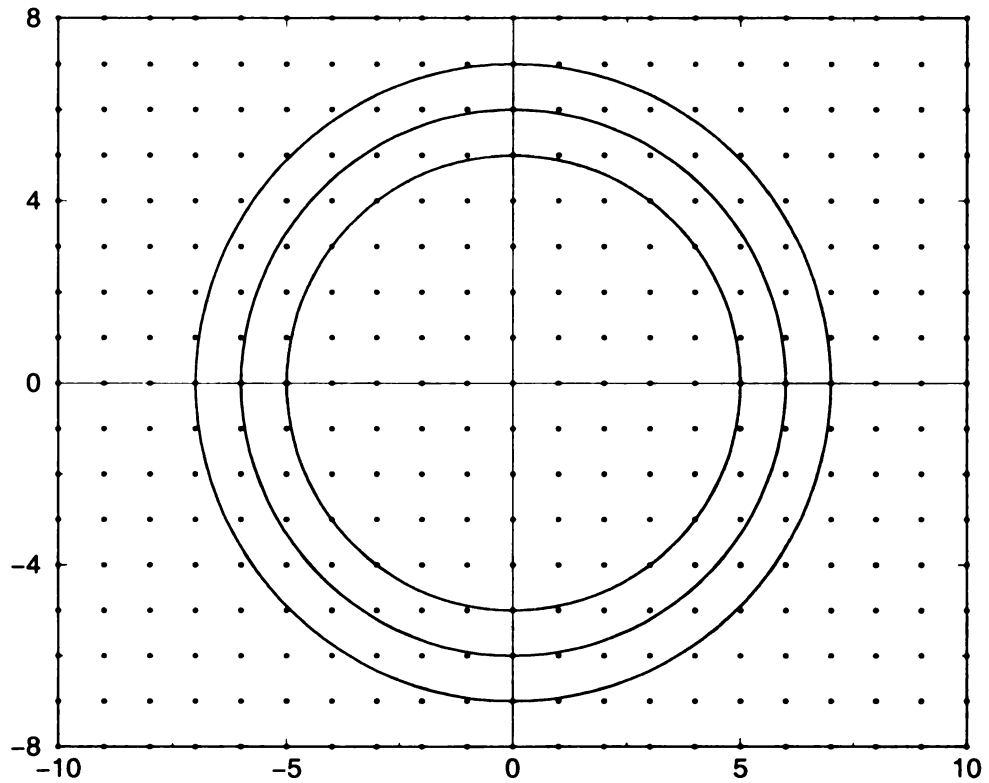

Figure 2.23: Incompatibility between the square lattice and the polar coordinate system. Two circular shells are shown (5,6) and (6,7). The number of lattice points within a shell only approximates the area of the shells, but is not exactly proportional to them. Some shells are underrepresented, some shells are overrepresented. For example there are 40 lattice points in shell (5,6) and only 36 in shell (6,7), even though the area of the latter shell is larger. This fluctuation is the origin of the noise on the PDF.

numbers. The total number of points inside a circle (sphere) with radius r ultimately approximates the area (volume) of the circle (sphere) with an uncertainty ϵ. The radius dependence of this uncertainty for the square lattice was studied by Gauss and Sierpinski [45]. Gauss showed that $\epsilon = k_1 r$, Sierpinski proved $\epsilon = k_2 r^{2/3}$, where k_1 and k_2 are constants, and r is the radius of the circle. We definitely do not want to reduce the error by increasing the radius of the particles, because that would require even larger lattices. To remove the noise, we calculate the ratio between the number of lattice points in a shell and the *exact* area or volume of the shell. The exact formulas of:

$$\begin{aligned} \pi\left((r+1)^2 - r^2\right) &= \pi\left(2r+1\right), \\ \frac{4\pi}{3}\left((r+1)^3 - r^3\right) &= \frac{4\pi}{3}\left(3r^2+3r+1\right) \end{aligned}$$

must be used for the area and volume elements instead of the $2\pi r dr$ and $4\pi r^2 dr$, because $dr = 1$ is not negligible compared to the radii of smaller shells. These ratios for the first fifty shells are shown in Figure 2.24. If we divide our noisy pair distribution function from Figure 2.22 with the corresponding lattice measure function from this figure, the noise disappears. The smooth PDF is shown in Figure 2.25. It looks really smooth, but how does it compare to results from continuum simulations? Numerical data of continuous Monte Carlo simulations were taken from [46], and plotted together with our simulations at four different densities in Figure 2.26. They agree very well, but the PDF from the lattice is systematically under the continuum values for the first few points. This can be understood qualitatively: the particles on the lattice can be in close contact with each other at the minimum distance $2R$ only in special orientations, for example exactly on top of each other vertically, next to each other horizontally, or in the directions of the Pythagorean triplets, if those exist for that radius. Particles in the continuum case on the other hand can touch each other at the closest $2R$ distance in any orientation. Therefore the value of the PDF is smaller

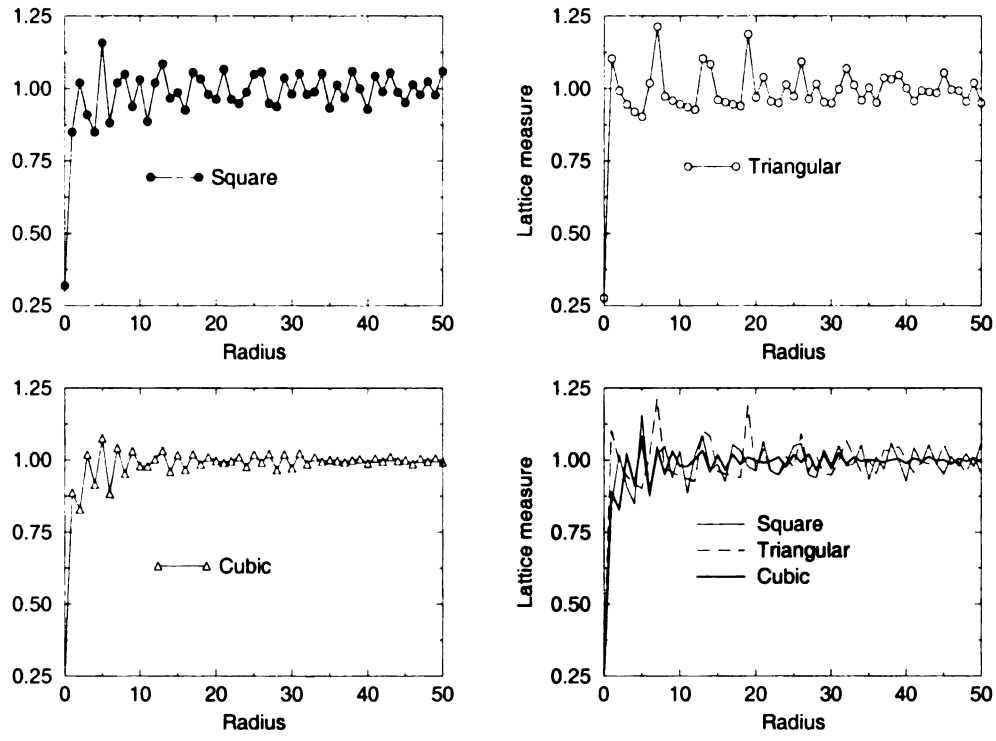

Figure 2.24: The ratios of the areas and volumes measured by the discrete lattices with respect to the exact continuum areas and volumes of the shells for the first fifty shells. The three different lattices are shown separately on the first three graphs and then together on the last graph. Interestingly the function for the triangular lattice is not smoother than for the square lattice as our intuition would suggest. The function for the cubic lattice is clearly smoother than the square one, probably because of the third dimension.

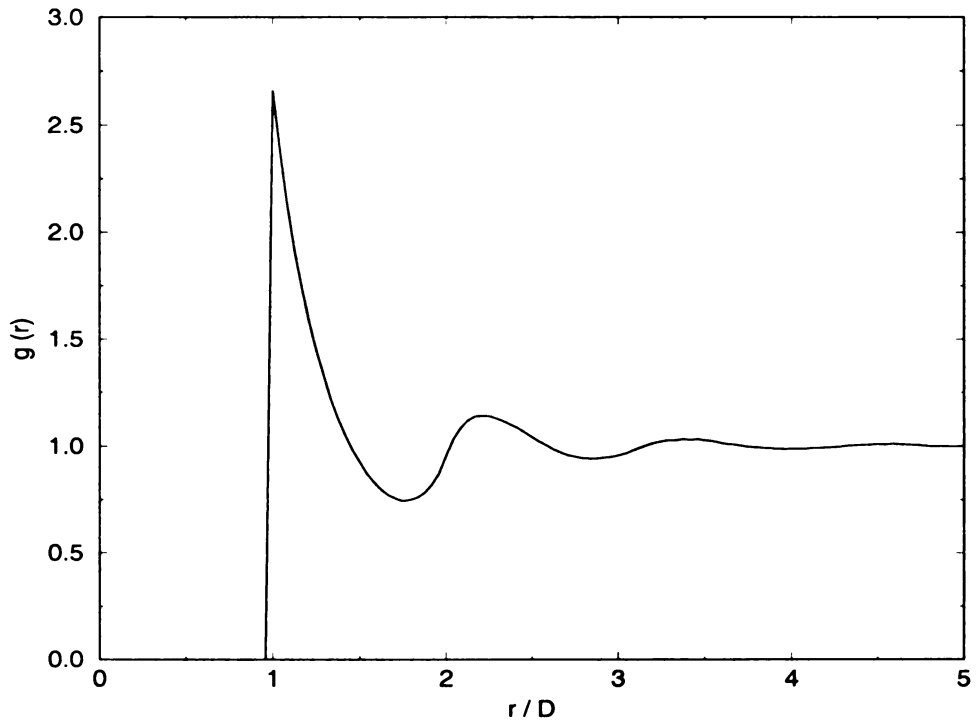

Figure 2.25: The smooth pair distribution function after the noise from lattice effects has been removed. The PDF from Figure 2.22 was divided by the lattice measure function of the square lattice from Figure 2.24. For the simulation parameters see Figure 2.22.

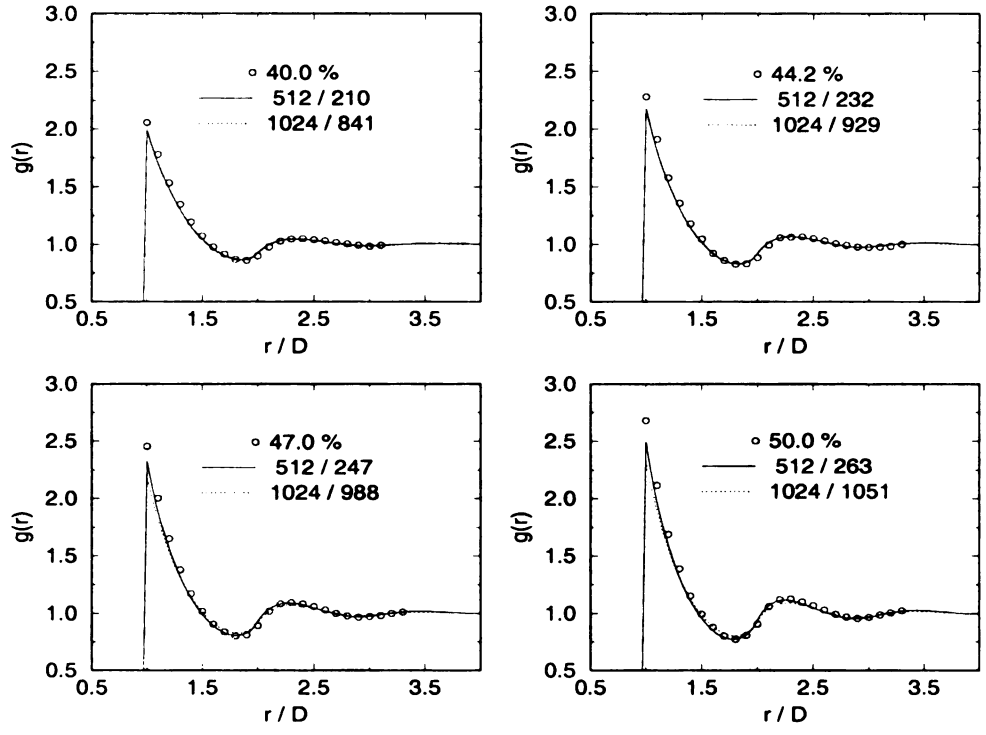

Figure 2.26: Comparison of the pair distribution functions obtained by continuous and discrete Monte Carlo simulations at four different densities. The data of the continuous simulations were taken from [46] and they are plotted as empty circles. The densities are given in percents of the hexagonal close packed density in two dimensions: $\rho_{A,HCP} = \pi/(2\sqrt{3}) = 0.9069$. The discrete simulations were performed on 512^2 and 1024^2 lattices with the number of particles shown in the graphs. The radius of the particles was 12.0 lattice units, and the particles took 5000 data collection steps after 1000 steps were discarded.

at short distances in the discrete case due to the limited number of positions on the lattice.

The following question can also be raised: does the pair distribution function within the lattice model depend on the underlying lattice or the exact type of random walk? The answer to this question is shown in Figure 2.27. The result is not surprising, the PDF is independent of the geometry of the lattice and the exact type of the randomwalk.

Finally the computational technique of the PDF collection is described briefly. When the distribution function is collected in a simulation, it is stored as a function of particle-particle *distance-square* and not distance. Because the distance square is an integer number even on the triangular lattice – as we have seen in the previous chapter –, the storage of the function is straightforward. The function is converted back to the distance space only at the very end of the simulation, when the final results are prepared for output. This way we can save significant amount of CPU time by not calculating the square root of the distance squares inside the simulation loop. The price we pay for this high performance is again large amount of extra memory. For example in a square shaped simulation box with side a the largest possible distance between two particles is $\frac{\sqrt{2}}{2}a$ according to the minimum image convention. The square of this distance is $\frac{a^2}{2}$. Therefore in the case of a 1024 by 1024 box we need to use a linear array of size 524,288 instead of a short array of about 724 elements. Furthermore the long array in the distance square space will be sparse, because a large portion of the integer numbers cannot be written as a sum of two squares, therefore no PDF events will ever be recorded at those positions. The final conclusion is the same again: we can save either CPU time or memory, but not both.

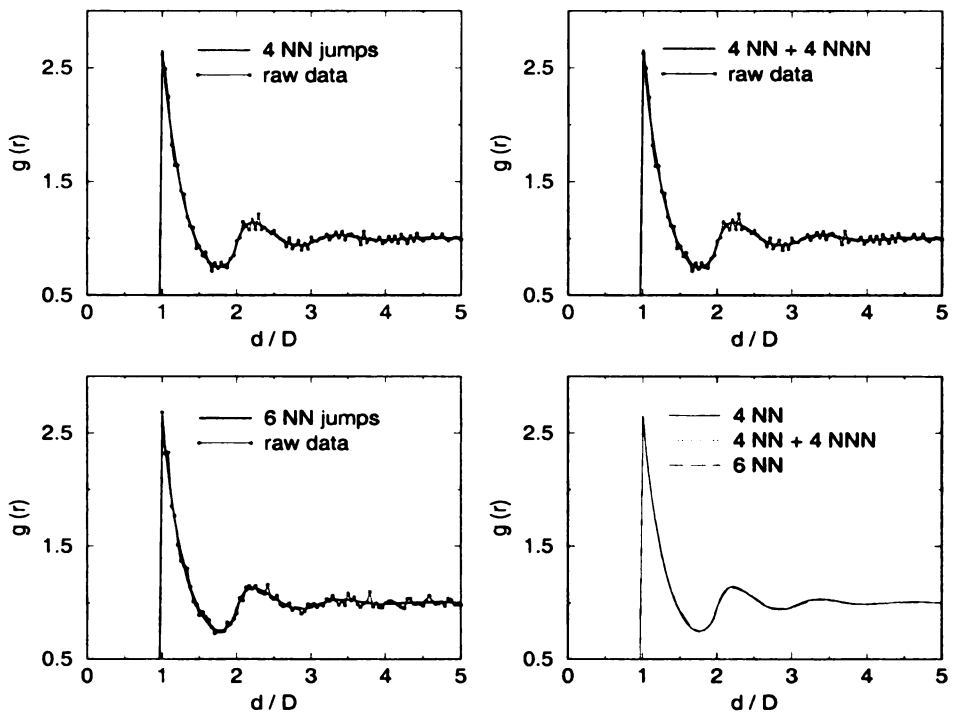

Figure 2.27: The pair distribution functions for different lattices – square and triangular – and different random walks on the square lattice. The first two graphs are for the square lattice with four nearest neighbor (4NN) and four nearest neighbor plus four next nearest neighbor (4NN + 4NNN) jumps. The third graph shows the PDF on triangular lattice with six nearest neighbor (6NN) jumps. The three graphs are shown together on the last graph. The curves are almost exactly on top of each other. The radius of the particle is 12.0 lattice units and the area density is 0.474 for each case. The first three graphs show the raw noisy PDF also. The last graph contains only the smooth denoised functions.

2.5.3 Density profile inside narrow slits

One of the simplest geometrical constraints we can place our particles in is a parallel slit. Either in two or in three dimensions one of the particles' coordinates is restricted to an interval. When a particle attempts to cross the restricting boundary, it is simply not allowed to take the step and it is not brought back at the other end of the interval either, as it would be in the case of periodic boundary condition. The structure of the liquid is strongly influenced by the presence of the wall. This geometrical setup has been studied extensively by computer simulations [47] - [53], and by analytic and semi-analytic methods such as density functional methods [54] - [57]. It is the testbed for any new method, therefore we will test our method on this geometry too.

The two parallel walls are set up at a distance of 11 particle diameters from each other both in two and three dimensions. The particles are placed randomly between the walls at different densities. Due to the finite nonzero diameter of the particles, the width of the area or volume allowed for the centers of the particles is narrower than the true slit width by one particle diameter. The local particle density across the slit is measured in the simulations by collecting the particle positions. It is straightforward and efficient to collect events into a histogram on the underlying discrete lattice. The results for two dimensions are shown in Figure 2.28, for three dimensions in Figure 2.29. Traditionally the reduced density is used in the literature on these plots. The reduced density in two dimensions is:

$$\rho_{R,2D} = \rho_N D^2 = \frac{4}{\pi}\rho_A,$$

in three dimesnions it is:

$$\rho_{R,3D} = \rho_N D^3 = \frac{6}{\pi}\rho_V,$$

where $\rho_N = N/A$ or N/V is the number density, ρ_A and ρ_V are the area and volume densities respectively. The reduced density is 1, when the circular (spherical) particles form a square (simple cubic) lattice in two (three) dimensions exactly touching each

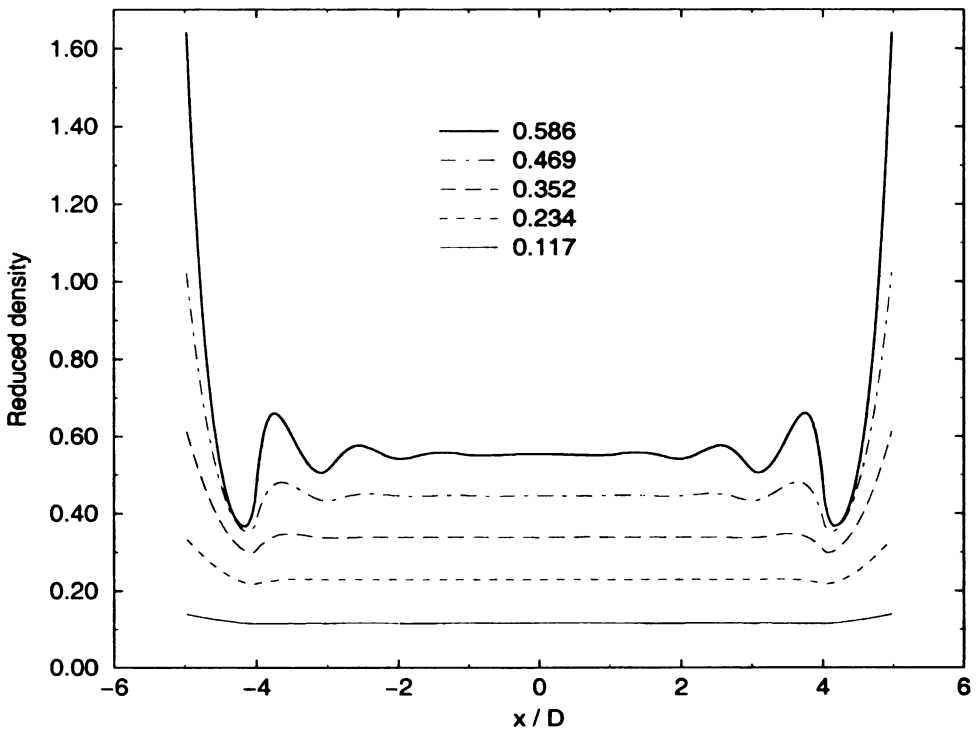

Figure 2.28: Density of a two dimensional liquid inside a parallel slit. The distance between the walls is 11 particle diameters (D), therefore the centers of the particles are restricted to the [-5,+5] interval. As the density increases, the particles start to form layers next to the walls. The average reduced densities of the ensembles are given on the graph. At the lowest density 800, at the highest density 4000 particles were simulated for 1 million steps each. The radius of the particles was 12.0 lattice units, the width of the allowed region was 240 lattice units.

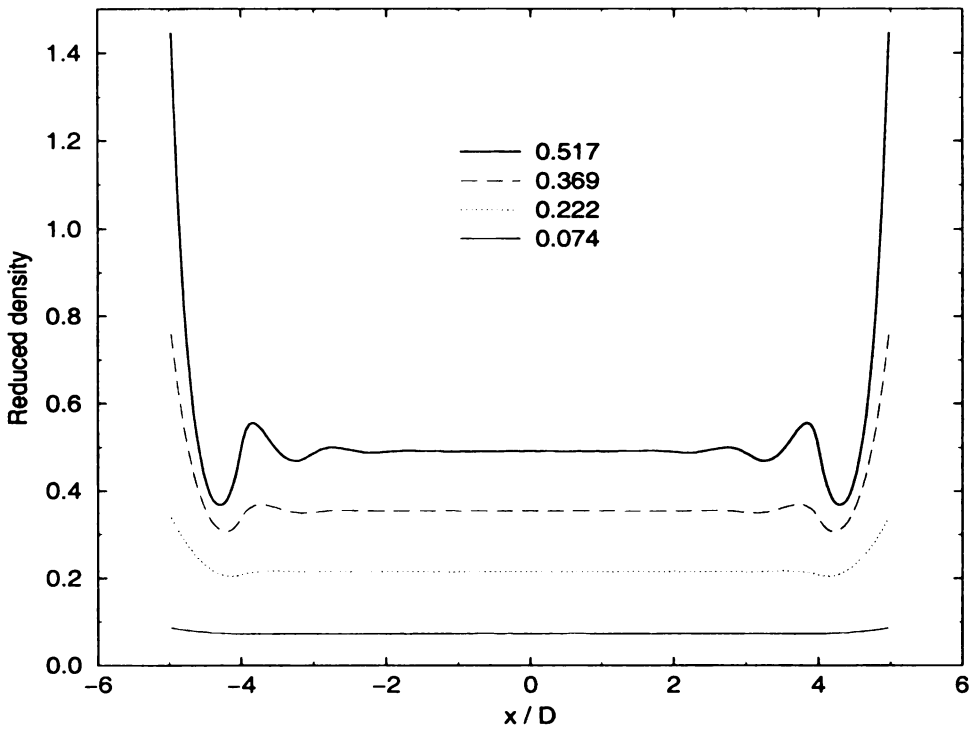

Figure 2.29: Density of a three dimensional liquid inside a parallel slit. The distance between the walls is 11 particle diameters (D), the same as in the previous figure. The same layering effect can be seen when the density is increased. The average reduced densities of the ensembles are given on the graph. At the lowest density 400, at the highest density 2800 particles were simulated for 1 million steps each. The radius of the particles was 11.0 lattice units, the width of the allowed region was 220 lattice units.

other. In these reduced density units the highest close packed densities both in two and three dimensions correspond to a number greater than one.

From the figures we can see the layering of the particles next to the wall. These particles are pushed against the wall by the other particles. We can clearly observe three layers of particles at the highest densities shown in the figures. In the middle of the slit the densities are nearly flat. What happens if the walls are moved closer to each other, while the particle density is kept at the same level? First, the flat middle part of the density function will shrink. When it disappears, the two wavy sides start to interact with each other, deviating slowly from the universal limiting shape. This process is shown in Figure 2.30. When the slit gets even more narrow, the shape of the density function becomes very sensitive to changes in width. Figure 2.31 shows the density function, when the width is between 2.5 and 5.0. The order of the graphs is now reversed, the width is increased from the narrow to the wide, and the functions are centered around the middle of the slit to see the layer formation better. The widest function on this figure – 5.0 particle diameters – corresponds exactly to the shortest function from the previous figure. There are three layers inside a slit 2.5 diamater wide, and there are five five layers inside a slit 5.0 diamaters wide. How does the third layer form in the middle? Let us take another step further to even narrower slits. Figure 2.32 shows the detailed mechanism of layer formation from a width smaller than the particle radius to the width of 2.5 diameter, at which the middle layer is fully developed. When the slit is very narrow, the density profile is flat. The particles can bump into each other only from the front or the back. When the slit gets a little wider, the particles start to push each other sideways against the wall: the density function bends up. With further increase in width, the round shaped valley profile sharpens, and suddenly the middle layer develops. It is just a little bump first, but then it develops to a big peak in the middle. Figure 2.33 shows the development of the middle layer at the finest scale reachable at this lattice

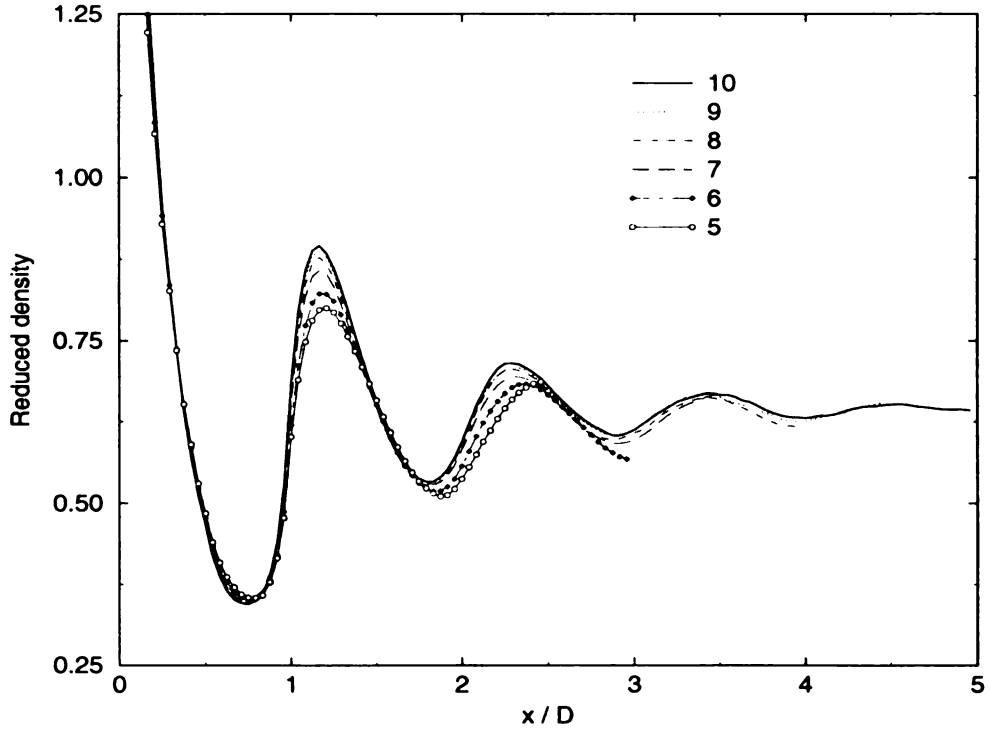

Figure 2.30: Decreasing the width of the slit in two dimensions, while keeping the particle density constant at $\rho_R = 0.688 \pm 0.002$. As the slit walls come closer, thes shape of the density function start to change significantly. The left wall is at position $x = -0.5$, the first layer of particles is formed at $x = 0$. Only the left half of the functions are shown due to the symmetry. The numbers on the graph represent the width allowed for the particle centers measured in diameters. The function has the same shape from width 10 to 7, at widths 6 and 5 it starts to change its shape.

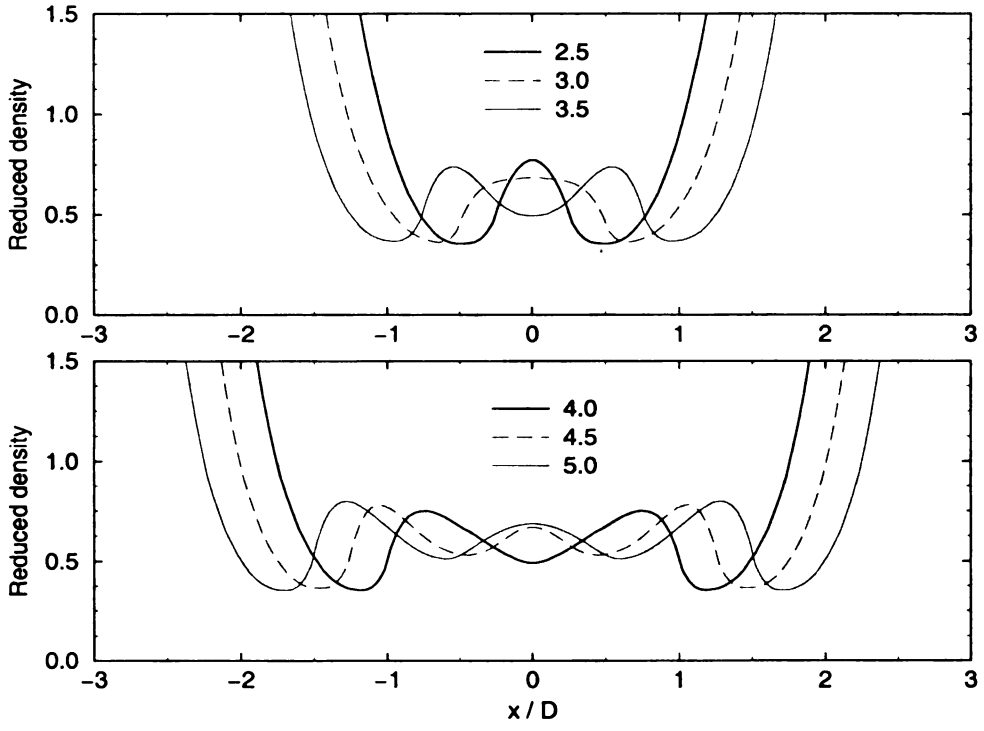

Figure 2.31: Layer formation inside a two dimensional slit when the width is increased from 2.5 to 5.0 particle diameters. The density is the same as in the previous figure, but with a larger uncertainty, because the particle system is smaller: $\rho_R = 0.688 \pm 0.005$. The middle layer at width 2.5 widens first at width 3.0, then splits into two peaks at 3.5. The smallest system contained 585 particles, the largest 1178. The data collection was 100,000 timestep long per particle.

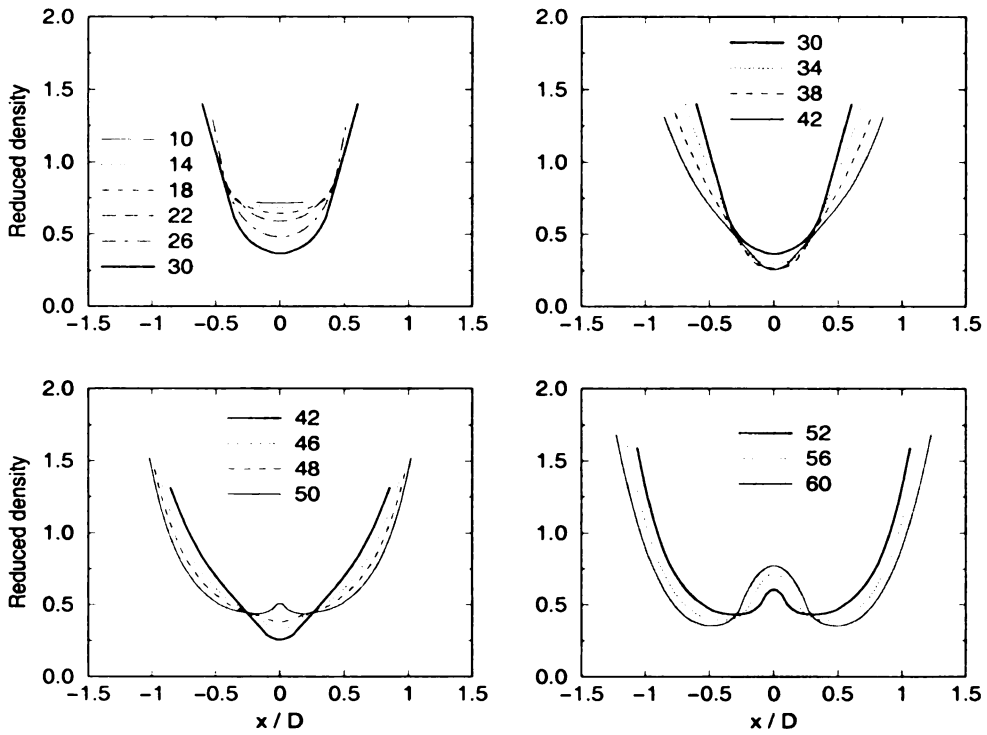

Figure 2.32: The detailed mechanism of particle layer formation beginning at a width narrower than a particle radius to 2.5 particle diameter. The density is the same as in the prevoius two figures: $\rho_R = 0.68 \pm 0.02$. The narrowest system contained 102, the widest 585 particles, taking 100,000 steps each. The numbers on the graphs give the slit width in lattice units, but the unit on the x axis is still particle diameter as before. The particle radius was 12.0 lattice units.

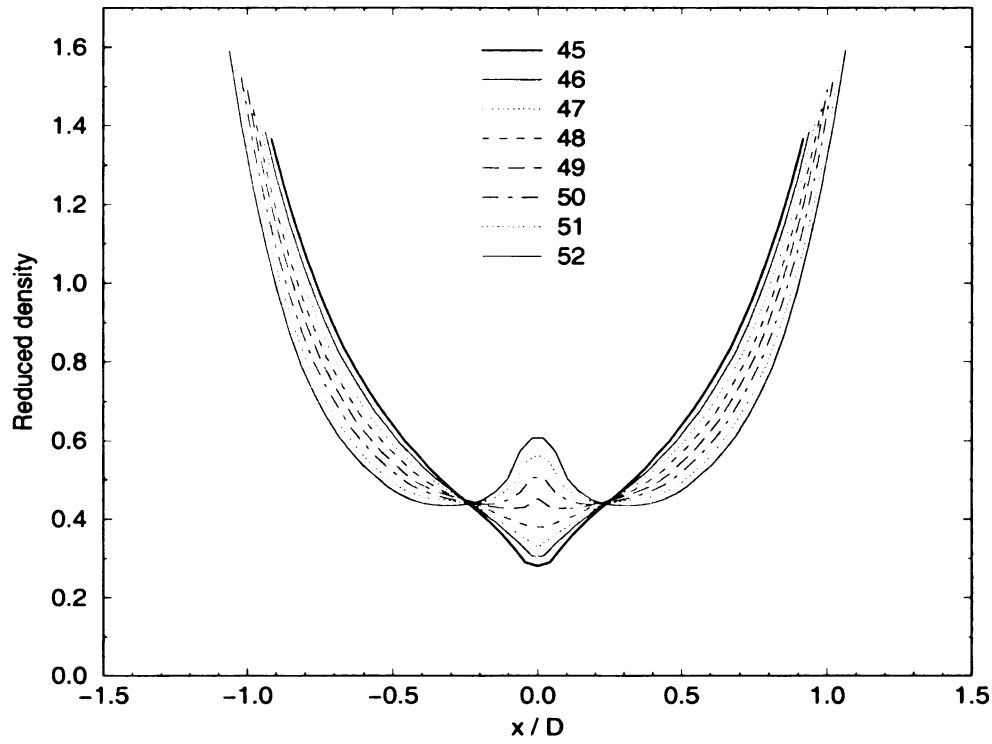

Figure 2.33: The development of the middle layer at the finest scale, when the slit width is changed by one lattice unit at a time. The particle radius is 12.0 lattice units, the density is the same as in the previous figure: $\rho_R = 0.68 \pm 0.02$. The early formation of the third layer can be clearly observed.

resolution.

The density profiles for wide slits presented in this section are in good agreement with the results of continuous type Monte-Carlo simulations [47] – [49]. However, no simulation results are known to us for very narrow slits with fine stepsizes [50].

The density profiles did not need any additional noise removal procedure as in the case of pair distribution function, because the slit walls were always parallel to the underlying lattice. However, density profiles collected in cylindrical or spherical pores would contain noise from lattice effects and would need to be corrected with the technique used for the PDF.

Chapter 3

Modeling zeolites

3.1 Introduction

Many commercial and noncommercial computer packages exist that permit one to visualize molecules and crystal structures by wireframe, stick, ball-and-stick models, and space-filling Corey-Pauling-Koltun (CPK) representations [58]. For Zeolites and other porous materials such as vycors, aerogels and recently discovered nanoporous materials such as MCM-41 the geometry of the pore space plays a very important role in the physics and chemistry of the system [59] – [61]. The *solvent contact* and *solvent accessible surfaces* can be constructed with the *rolling sphere* method [58]. Many commercial computer programs use this algorithm for molecules, in which the sites that are accessible to spheres of a specified radius are displayed. We present a method here which is much simpler and more efficient for a large collection of atoms such as crystals and other complex systems [62]. The mathematics of the method is discussed in the next section. Then the visualization process is described generally. After that four different zeolites are presented as examples, illustrated by a series of pictures generated by the visualization software. The examples are the following: Linde type-L zeolite as a one-dimensional system [63]; MCM-22 as a two-dimensional

one [64, 65]; ZSM-5 [66] and Zeolite-Y [67] as three-dimensional pore structures. At the end simple statistical analysis is applied to extract more information about the example systems. In the last section, the results are summarized and information is given about the availability of the program package. It should be noted at this point that originally this algorithm and the code was developed in a different scientific area – in the channel structure of *electrides* – and it is already in everyday use in that field [68] – [76].

3.2 The mathematical basis of the method

A given number of atoms are at known fixed positions in space:

$$\vec{c_i} = (x_i, y_i, z_i),$$

and every atom has a van der Waals radius r_i which is also known. The index i goes from 1 to N, the total number of atoms. Now we ask the following questions: Where are the empty regions - regions not occupied by atoms - in this structure and how to construct a three-dimensional picture to visualize them? What is the measure of *emptiness*? Let a mathematical grid be placed – for simplicity and other practical reasons a cubic grid – at the positions:

$$\vec{g_j} = (u_j, v_j, w_j)$$

over the space occupied by the atoms, in which $j = 1, 2, ..., m$ and m is the number of grid points. See Figure 3.1. At every grid point the distance between the mathematical grid point and the *closest* atomic surface is calculated and this distance is assigned to the grid point. The atomic surface is defined by a sphere with the radius of the atom in question, in other words we are using a hard sphere atomic model. Bonded atoms then appear as intersecting spheres. Mathematically the value assigned to each

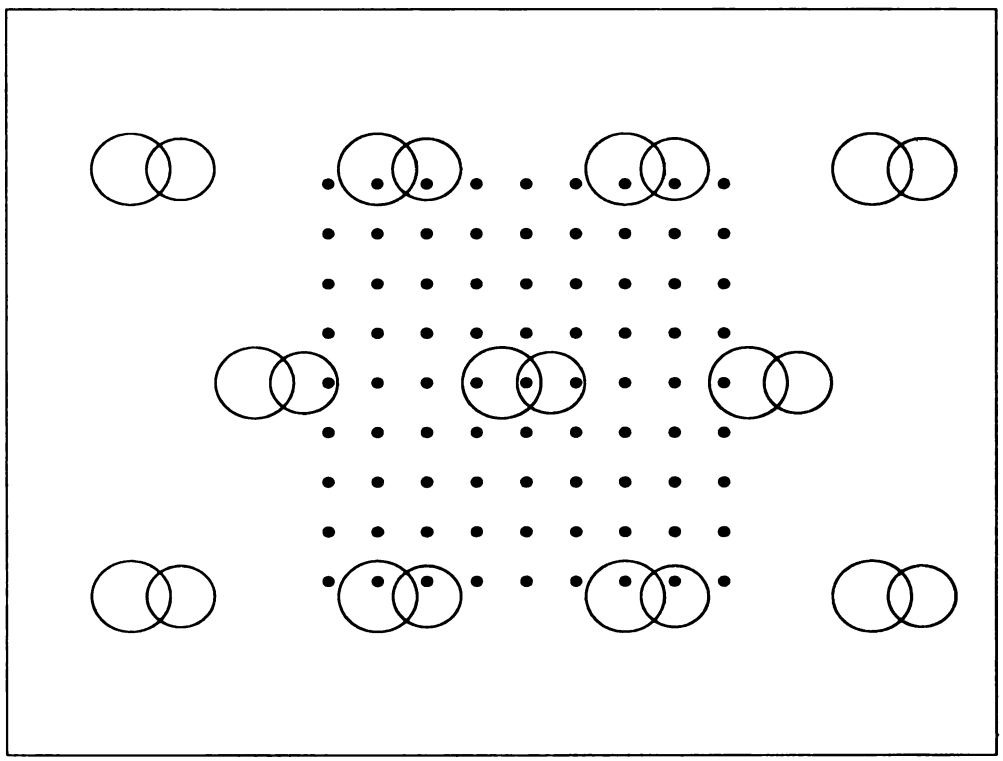

Figure 3.1: Sampling the space of an imaginary two dimensional crystal. The large empty circles represent the atoms, the small filled dots form the mathematical grid. The grid must be completely inside the crystal in order to avoid edge effects.

grid point is thus the following distance:

$$d_j = min\{\sqrt{(x_i - u_j)^2 + (y_i - v_j)^2 + (z_i - w_j)^2} - r_i\},$$

for all i. If a grid point happens to be inside the van der Waals radius of an atom, then simply zero is assigned to this grid point. As it can be seen, this procedure is nothing but placing the largest possible *test sphere* at every grid point without intersecting any atomic surface. This calculation provides a *scalar field :* a real number is assigned to every grid point in the space. The value of this scalar field ranges from zero – grid points inside or exactly on the atomic surfaces – to the largest possible distance found in the atomic structure – grid points at the center or close to the center of the largest cavities.

In computational geometry this problem belongs to the class called *motion planning* or the *piano mover's problem* [77], where a polygon shaped object has to be moved through a labyrinth of geometrical obstacles. Our method is one possible solution to the special case of spherical piano and spherical obstacles, applied to the case of atomic structures.

3.3 The visualization process

Commercial visualization programs – for example Application Visualization System (AVS) or Explorer – allow visualization of three dimensional scalar fields by drawing an *isosurface* [78, 79] through the points which have the same value of the scalar. Three dimensional contour plots are also a possible way to visualize scalar fields. The *level* of the isosurface, which in the present case is the radius of the test sphere, is controlled by the user. By changing this level, the scalar field can be scanned through, the cavity and channel structure seen by test spheres with different radii can be visualized. Alternatively the three dimensional isosurface can be imagined as intersecting front of the spherical waves starting from the atomic surfaces and

propagating outwards at a constant velocity [80]. By rotating the isosurface on the computer screen, the channel and cavity structures can be viewed from different directions.

To reduce the size of the data files, the real-valued scalar field is mapped linearly into the 0-255 integer interval, using only one byte for every grid point. The visualization programs mentioned above prefer this data format by default. If the scalar field has its largest value equal to 5-10 Å (which is typical for zeolites), then the error introduced by this fieldvalue-discretization is 0.02-0.04 Å. Due to the discreteness of the mathematical grid – the field is sampled at discrete positions – additional error is introduced, which can be much larger than this one. In a typical calculation, as seen in the next section, usually a 41 by 41 by 41 cubic mathematical grid is placed on a 36 Å by 36 Å by 36 Å real physical volume, although this may be increased to an 81 by 81 by 81 grid if desired. Therefore the former choice yields a distance between two gridpoints of 0.9 Å, and 0.45 Å for the latter. This will be the maximum error superimposed on the channel and cavity diameters.

The isosurfaces might be slightly rough for the visualization programs right after the calculations. The aesthetic quality of the pictures can be improved significantly without much lost information by using a smoothing procedure. The smoothing procedure is a three dimensional second order averaging with the following weight factors:

$$\frac{1}{64}\left[\begin{pmatrix}1&2&1\\2&4&2\\1&2&1\end{pmatrix},\begin{pmatrix}2&4&2\\4&8&4\\2&4&2\end{pmatrix},\begin{pmatrix}1&2&1\\2&4&2\\1&2&1\end{pmatrix}\right].$$

This three dimensional array of weights is the three dimensional tensor product of the weight vector $\frac{1}{4}(1,2,1)$ used for one dimensional smoothing [13]. More sophisticated smoothing weights or procedures are possible but were not necessary for our purposes. The new smoothed value of the scalar field associated with a given grid point depends

on its old value and the values of its 26 neighbors. The effect of this smoothing is shown in Figures 3.2 and 3.3. Figure 3.2 shows the original data field; Figure 3.3 displays the data field after two consecutive smoothings. The complete description of these Figures is given in the next section. The original six-fold symmetry around the longitudinal axis can be seen on the unsmoothed structure. But this symmetry is not very clear from the picture, because the underlying mathematical cubic grid used by the visualization program is not compatible with the hexagonal symmetry of the crystal. The smoothing procedure "cleans up" the picture, it smooths the sharp edges and peaks, and it deletes the small fragments. But, at the same time it results in some loss of information, as occurs with every kind of smoothing. The optimal number of applications of this smoothing on the data field is one or two according to our experience. The smoothing procedure is applied twice in every figure except Figure 3.2. The error introduced by the smoothing depends on the actual shape of the scalar field itself. To eliminate the smoothing error, the original scalar field may be used without any smoothing for precise calculations, and then the smoothened field for visualization only. By increasing the number of grid points this error and the discretization error can be reduced also. The price one pays for the smaller error is, as usual, increased computer time and bigger data files. The total error coming from the three different sources - discretization of space, discretization of field and smoothing - can be as large as 10-15 percent compared to the channel diameters in the Atlas of Zeolite Structure Types [81]. However the main purpose of the visualization method is to help understand qualitatively the channel structures in zeolites. More quantitative data can be obtained by special purpose programs as we will see.

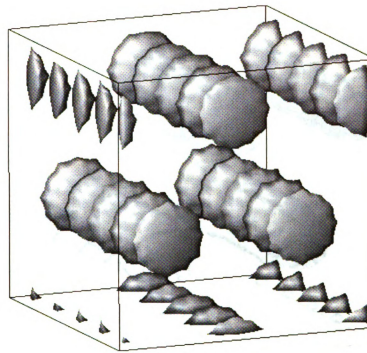

Figure 3.2: Perspective view of the channels in Linde type-L zeolite, seen by Argon atoms (r=1.7 Å). The volume with strange shape shown in gray is the volume available to the *center* of the test spheres with a given radius, in this case corresponding to an Argon atom. No smoothing has been apllied to the data field. The edge length of the cubic shaped sample volume is 35.9 Å. The crystal structure has hexagonal symmetry, therefore the cubic volume shown here is not compatible with the symmetry of the crystal. Three complete channels and a half one can be seen. The small fragments are from channels that extend outside of the sampled volume.

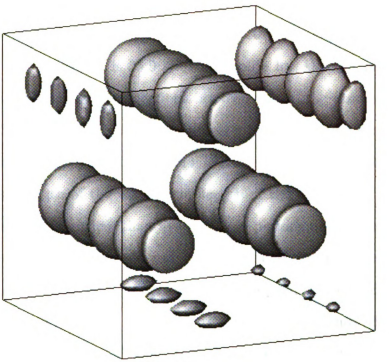

Figure 3.3: The channels in Linde type-L zeolite, seen by Argon atoms (r=1.7 Å) after two smoothings. The sharp edges, peaks and some small fragments are deleted due to this smoothing. The smoothing procedure is very useful for examining the main channels and cavities without the fine structures, but it introduces some error, because it modifies the field sligthly. This modification is evident in the figure: the smoothed channels have a slightly smaller diameter than the unsmoothed ones at the same level. Experience indicates that this change is not more than 10%.

3.4 Example structures

In the following four examples we have used 1.32 Å for the van der Waals radius of Oxygen and 1.82 Å for Silicon. The first example is the Linde type-L Zeolite [63], shown in Figures 3.2 - 3.4. The unit cell parameters are the following: $a = 18.46$ Å; $b = 18.46$ Å; $c = 7.51$ Å; $\alpha = 90°$; $\beta = 90°$; $\gamma = 120°$. The edge length of the cubic shaped volume on the figures is 35.9 Å. This Zeolite has straight parallel channels along the z axis without interconnections between them: the pore structure is one dimensional. The shape of each channel is almost cylindrical (it has six-fold symmetry around its axis as mentioned in the previous section) with a wavy profile along the axis. The repeat length of the wave is the length of the unit cell in the z direction which is 7.51 Å. Somewhat less than five complete periods are shown in the figures. There are three complete channels and one cut in half, because the cubic volume shown in the figures is not compatible with the hexagonal unit cell of the crystal. It should be emphasized here once again that the figures do not show the shape of the channels, but rather the locus of points that the *center* of a sphere of the designated radius can sample. The largest sphere which can be fit into the widest part of the channel has a radius of 4.8 Å, and the sphere which can be fit into the narrowest part has a radius of 3.5 Å. The latter radius is the critical one: a sphere with this radius can diffuse through the system, but a bigger one cannot. A comparison of Figures 3.3 and 3.4 shows the difference in effective porosity: the center of the larger Xenon atoms sample a smaller volume than the smaller Argon atoms.

The second example, the MCM-22 zeolite [64], is shown in Figures 3.5 - 3.8. The unit cell parameters are the following: $a = 14.11$ Å; $b = 14.11$ Å; $c = 24.88$ Å; $\alpha = 90°$; $\beta = 90°$; $\gamma = 120°$. The edge length of the cube is 30.4 Å. This structure has two different, non intersecting two-dimensional channel systems. One of the channel systems forms a triangular lattice, the other one a honeycomb lattice. There is no link between the two channel systems. The triangular lattice has large cavities about

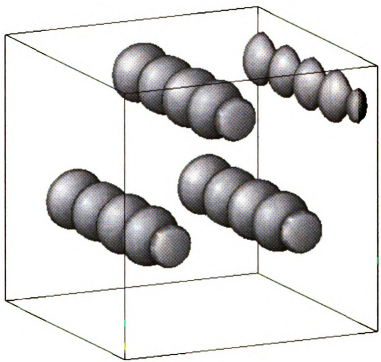

Figure 3.4: The channels in Linde type-L zeolite as seen by Xenon atoms (r=2.2 Å). Comparing this figure with the previous one, it is shown that Xenon atoms see a narrower channel structure than Argon atoms.

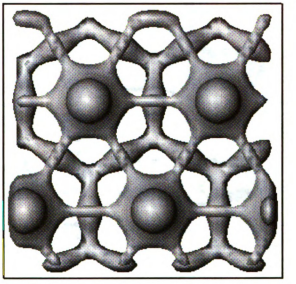

Figure 3.5: Top view of the channel structure of the MCM-22 zeolite viewed along the z-axis. The level is at the Argon atom radius (r=1.7 Å); the edge length of the cube is 30.4 Å. One of the channel systems forms a two-dimensional triangular lattice with big cavities at the hexagonal sites. The second channel system is a two-dimensional honeycomb lattice.

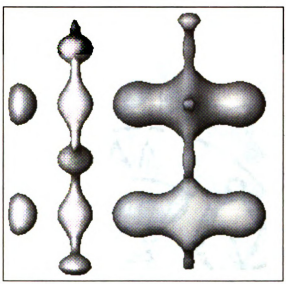

Figure 3.6: Side view of the channel structure of the MCM-22 zeolite viewed along the direction of one of the a axes. The level is the Argon radius (r=1.7 Å). This view shows that there is no appreciable connection between the two channel systems. The length of the large cavities in the triangular lattice is 20 Å, the diameter is about 8.2 Å. The "triangular lattice - honeycomb lattice" pattern repeats itself in the z direction. The two small hemispherical fragments are parts of the large cavity from the next triangular lattice.

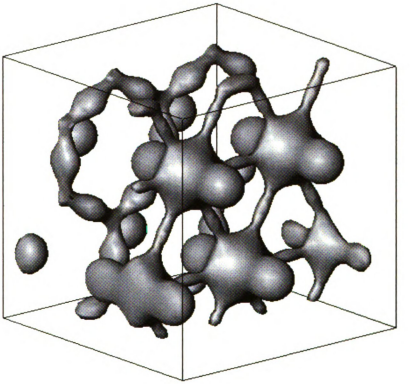

Figure 3.7: Perspective view of the channel structure of the MCM-22 zeolite viewed at the level of the Argon radius. This view provides a good illustration of the relative positions of the two channel structures in the three dimensional space.

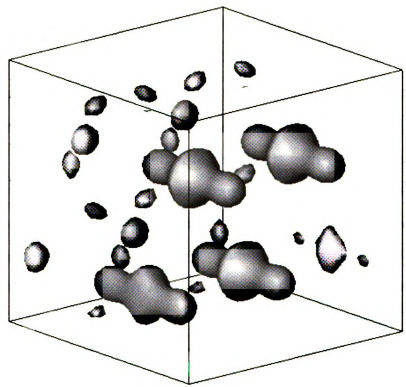

Figure 3.8: The same perspective view as the previous figure, but now at the level of the Xenon radius (r=2.2 Å). Clearly, both channel systems became disconnected; the Xenon atom is too large to diffuse through this zeolite.

20 Å long and 8.2 Å in diameter. The honeycomb lattice does not have similar features. The "triangular lattice - honeycomb lattice" pattern repeats itself in the z direction. Argon atoms could diffuse through both channel systems. The triangular lattice connections close at a test sphere radius of 1.8 Å, the honeycomb at 1.9 Å. Both channel systems are closed for Xenon atoms.

The third example is the well known ZSM-5 zeolite [66]. The unit cell is rectangular with three different side lengths: $a = 20.022$ Å; $b = 19.899$ Å; $c = 13.383$ Å. Three side views and two perspective views are shown in Figures 3.9 - 3.13. The edge length of the cube is 35.9 Å. The straight channels in the direction of the b axis are the major channels in this structure with a diameter of 4.5 Å. The secondary, wavy-shaped channels in the direction of the a axis with a diameter of 4.3 Å are slightly narrower. Altough there are no straight direct channels in the direction of the c axis, particles can diffuse slowly in this direction using the major and the secondary channels alternately. The side views and the first perspective view show the channel structure experienced by Argon atoms. The second perspective view, Figure 3.13, shows the structure at the level of the Xenon radius. From this figure it can be seen that the Xenon atoms could not pass through the narrower secondary channels, and they are at the threshold of going through the wider major channels. It would be simple to calculate the threshold more precisely by increasing the resolution of the sampling grid or sampling a smaller volume of the crystal.

The last example is the Zeolite-Y structure [67], shown in Figures 3.14 - 3.15. This zeolite has a cubic unit cell: $a = 25.03$ Å. The edge length of the sample cube is 35.5 Å. The tetrahedral-shaped major cavities form a diamond lattice. The diameter of these cavities is 11.2 Å and they are connected to each other through channels with a radius of 3.3 Å. There is a secondary cavity in this structure with radius 2.2 Å, but access is provided to these cavities only through very narrow channels of radius 0.6 Å. Zeolite-Y is probably the most thoroughly studied zeolite in laboratory

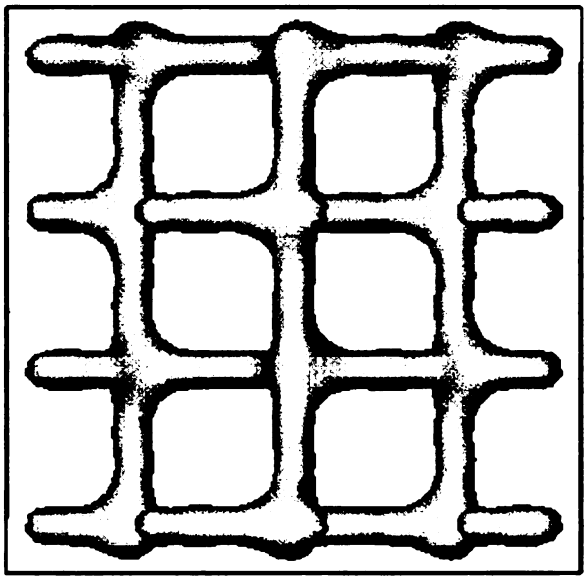

Figure 3.9: The channels in zeolite ZSM-5 viewed along the direction of the *c*-axis at the level of the Argon radius (r=1.7 Å). The edge length of the cube is 35.9 Å. The vertical channels, in the direction of the *b*-axis, are the major channels in the structure with diameters of 4.5 Å. The horizontal secondary channels, in the direction of the *a*-axis, with diameters of 4.3 Å are slightly narrower.

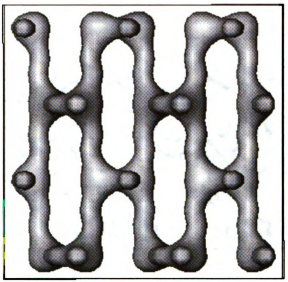

Figure 3.10: ZSM-5 viewed along the a-axis at the level of Argon radius. On this view the major channels are still vertical – direction b – , but now the structure is viewed along the direction of the secondary channels.

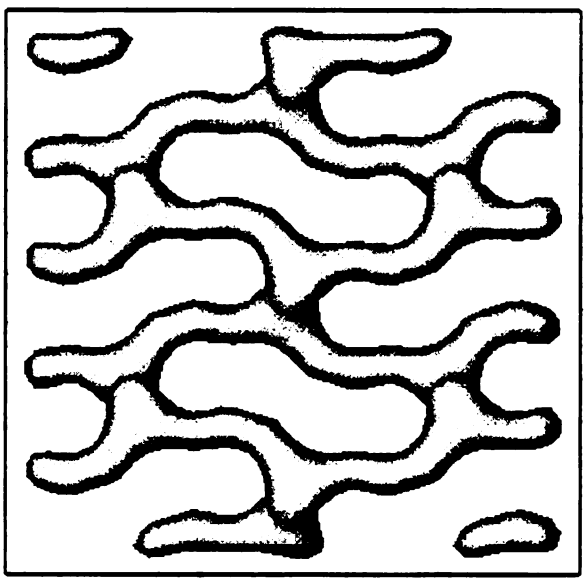

Figure 3.11: ZSM-5 viewed along the *b*-axis, along the direction of the major channels. The horizontal wavy shaped channels are the secondary channels in the direction of the *a* axis. Although there are no straight direct channels in the direction of the *c*-axis, particles can diffuse slowly in this direction by using the major and the secondary channels.

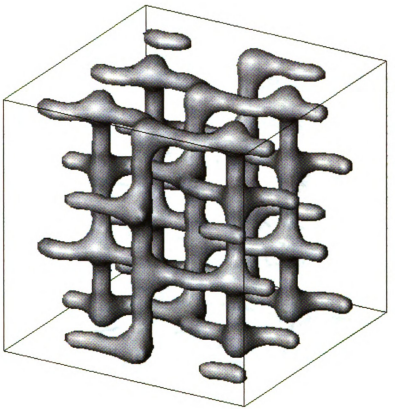

Figure 3.12: A perspective view of ZSM-5 at the level of the Argon radius (r=1.7 Å) that shows the complex three dimensional channel network. Rotation of pictures such as this provide information about the connectivity, that is not easily available with traditional molecular or crystal visualization tools.

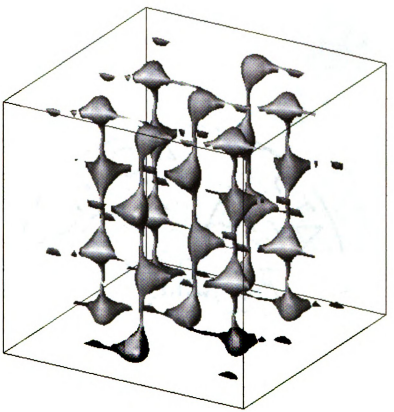

Figure 3.13: A perspective view of ZSM-5 at the level of the Xenon radius (r=2.2 Å). The secondary channels are already closed at this level, and the major channels are barely large enough to permit passage of Xenon atoms.

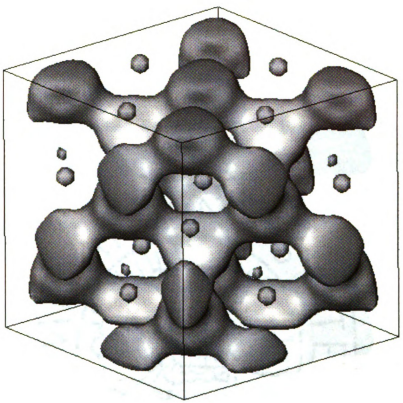

Figure 3.14: The channels and cavities in the Zeolite-Y structure at the level of the Argon radius (r=1.7 Å). The major cavities form a diamond lattice. The radius of the largest test sphere which can be fit into these tetrahedral-shaped cavities is 5.6 Å. The major cavities are connected to each other through channels with a radius of 3.3 Å. The secondary spherical shaped cavities have a radius of 2.2 Å. Access is provided to these cavities only through narrow channels with a radius of 0.6 Å. These channels cannot be seen at the level of the Argon radius.

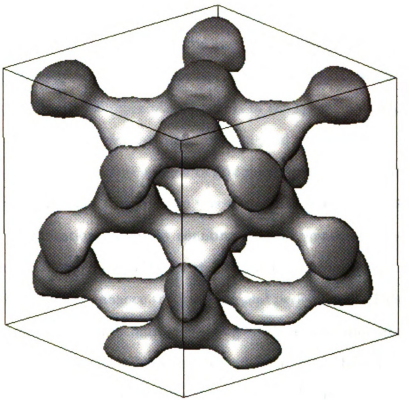

Figure 3.15: The channels and cavities in the Zeolite-Y structure at the level of the Xenon radius (r=2.2 Å). The diamond structure can be seen more clearly at this higher level. The secondary spherical cavities have just disappeared, because they have about the same radius as Xenon atoms.

experiments and in computer simulations due to its large cavity and channel sizes and three dimensional structure [82] - [86].

3.5 Statistical Analysis

Besides using this procedure for visualization, the scalar field can be processed using statistical analysis to gain more information about the geometry of the pore structure. The probability density function $n(r)$ can be calculated straightforwardly. The $n(r)dr$ product gives the relative number of grid points in the space at which a test sphere with radius r_t, $(r < r_t < r + dr)$ can be fit into the solid while touching the atoms. In other words $n(r)dr$ is the relative number of grid points exactly at distance r from the atomic surfaces. Originally, this function was computed directly from the scalar field calculated for visualization, simply by counting the grid points with the same values and then building a histogram to see the distribution. Later an independent method was developed in order to enhance the precision and performance. This method uses only one unit cell - the smallest unit containing all the information about the whole crystal - with periodic boundary condition instead of the cubic shaped sample volume. Changing the size of the sample cube leads to slightly different statistics, because the cube is not compatible with the symmetry of the crystal. Furthermore, the grid points used in the statistical ananlysis to sample the space are generated at random positions inside the unit cell instead of via a specific cubic pattern as in the visualization. The results of this calculation are shown in Figure 3.16. For every structure the density function goes to zero at the radius of the largest possible test sphere R_{max}, which is 4.8 Å for the Linde type-L Zeolite, 4.6 Å for MCM-22 structure, 3.3 Å for ZSM-5 and 5.6 Å for Zeolite-Y. The total area under the probability density function *i.e.* the integral from zero to the R_{max} radius gives the total porosity of a given structure: 0.57 for Linde, 0.56 for MCM-22, 0.53 for ZSM-5 and 0.66 for Zeolite-Y. The total

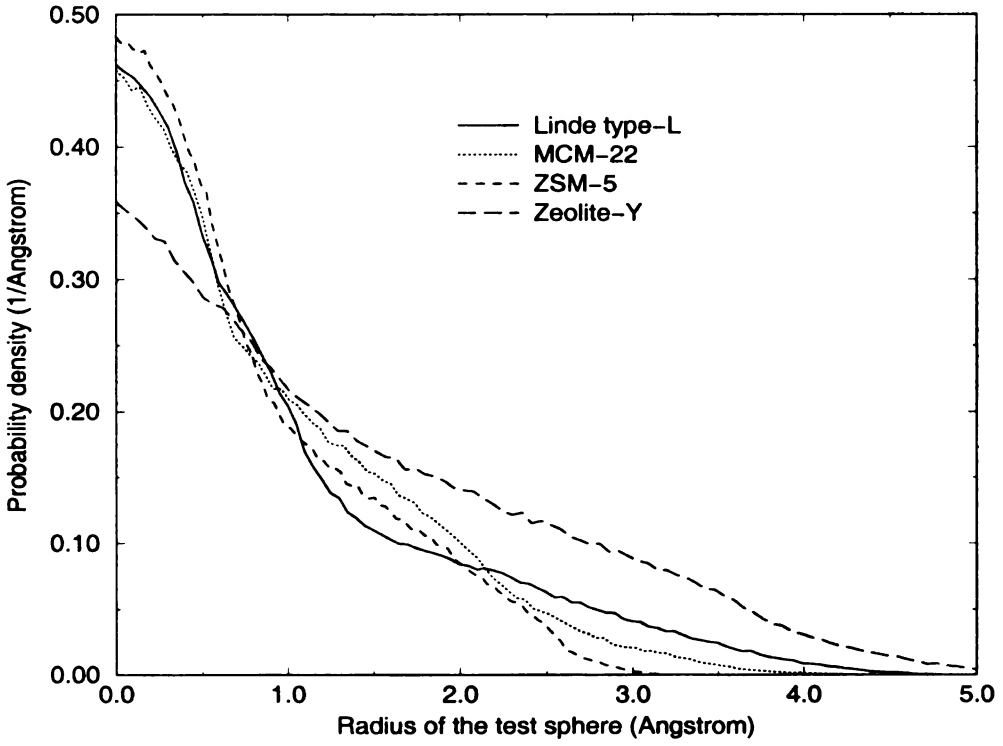

Figure 3.16: The porosity probability density functions for the four different zeolites. These functions were calculated by generating 1 million probing positions randomly inside the unit cell. The density functions go to zero at R_{max}, the radius of the largest cavity found in the crystal (Linde type-L: 4.8 Å; MCM-22: 4.6 Å; ZSM-5: 3.3 Å; Zeolite-Y: 5.6 Å). The integral of the density functions from r_t, the radius of a given test sphere to R_{max}, the radius of the largest test sphere that can be fitted into the crystal gives the effective porosity functions shown in the next figure.

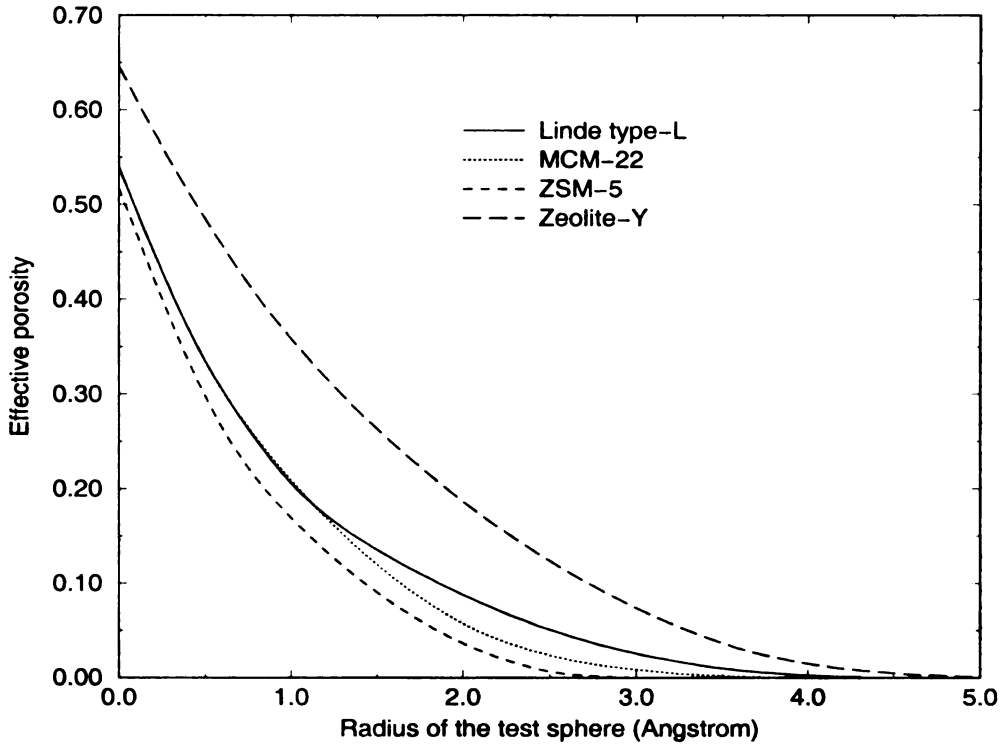

Figure 3.17: The effective porosity functions for the four different zeolites. The curves start from the maximum porosity values for a test sphere of 0.0 Å radius (Linde type-L: 57%; MCM-22: 56%; ZSM-5: 54%; Zeolite-Y: 66%), then decrease as the radius of the probing particle increases and finally go to zero at R_{max}, the radius of the largest spherical cavity found in the crystal (Linde type-L: 4.8 Å; MCM-22: 4.6 Å; ZSM-5: 3.3 Å; Zeolite-Y: 5.6 Å). These function were computed by numerical integration of the density functions.

porosity of a crystal is the ratio of the empty volume - volume not occupied by atoms - and the total volume of the crystal. Integration of the density function from r_t - a given test sphere radius in the range between zero and R_{max} - to R_{max} gives the *effective porosity* function. These functions are shown in Figure 3.17 for our example zeolites. The effective porosity has the following meaning: it is the relative volume in the crystal where a test sphere with radius r_t is allowed to be placed. These are the regions which are at a distance of at least r_t from the atomic surfaces. Obviously

a larger atom sees smaller free space inside the cavities and channels than a smaller atom, therefore the effective porosity is reduced for a larger atom. For example Figure 3.17 shows that an Argon atom (r=1.7 Å) sees only 10-20% effective porosity in the four Zeolites instead of the total void of 53-66%.

The effective porosity function contains more information about a crystal than the simple value of porosity - the ratio of void and total volume -, which is just one value of the whole function at $r = 0$. However it should be noted that the construction of the effective porosity function is purely mathematical. The test spheres are dropped into the crystal artificially from outside through an extra dimension. In reality a diffusing atom has to travel through narrow channel to reach large cavities. Even if the channels are perfectly straight, at intersections they might form cavities with radius larger than the radius of the channels. These cavities will always be *unreachable* from outside for larger particles even if they can accomodate them. Therefore the real physical porosity function decays faster than the mathematical porosity function. This mathematical porosity function is an upper limit. A more realistic construction of the function would require the correct treatment of accessibility as in [87]. These percolative threshold sizes of the channels can be determined - even if not very precisely - from the visualization process by changing the isosurface level.

3.6 Summary

We have presented a geometrical method here that allows us to visualize channels and cavities in zeolites and other porous structures by using computers, as exemplified by pictures of different zeolites. The pictures can help not only to understand better the connectivities and the structure of pore space, but also to give a simple yet quantitative description about pore sizes and threshold sizes of the diffusing particles. The program package to generate the data fields for visualization from a crystal

structures (documentation, installation instructions, Fortran source files and example structures) is available for free usage from:

`ftp://argus.cem.msu.edu/pub/dye/voids`

by anonymous file transfer, and technical support is available from the author. The package is currently used at several universities and at the NEC Laboratory in Princeton [88].

Chapter 4

Summary and conclusion

A class of discrete computational models was developed and implemented as an alternative to continuous type simulations such as Molecular Dynamics and continuum Monte-Carlo methods.

Four different algorithms were used at the driver level: traditional neighbor-list, a fast neighbor-list, cell-list and field-representation. The fast neighbor-list method designed in this work is a sophisticated extension of the traditional neighbor-list method. It fixes the major deficiency – the quadratic dependence of the calculation time on the size of the system simulated – of the neighbor-list method resulting in a true linear algorithm. This algorithm is independent of the underlying lattice; therefore it can be used in continuous type simulations as well. The cell-list method implemented here is a discrete version of the widely used continuous cell-list or domain decomposition algorithm. It uses simple bit manipulation very efficiently to sort the particles into the cells. The field-representation method is a generalization of the Lattice-Gas method: the particles in the Lattice-Gas simulation are pointlike; in the field-representation they have their own structure. These algorithms give different performance depending on the exact situations, but they all outperform the currently fastest Molecular Dynamics and other continuous type simulations. This is mainly

due to the integer arithmetic employed.

Three different lattices – square, triangular and cubic – were implemented and studied. The particles can follow two different traveling methods: random walk or ballistic motion. After collisions the new directions can be generated in an isotropic or in a biased way. The underlying lattice can be utilized to represent arbitrary geometrical constraints or external potential field or both at the same time. The particles can interact with each other through hard core repulsion only or with some model potential. Several physical properties can be collected in the simulations such as density profiles or pair distribution functions. When combining all these possibilities together the total number of combinations is quite large, it is in the range of several hundreds. Obviously not all of them gives a physically sensible model. Thirty of the most fundamental cases have been implemented in Fortran and used successfully for this work. Many other cases are in the experimental stage. The length of an average module is about one thousand lines.

The most important discrete computational techniques were discussed in detail. These techniques are still not very widely known and used. They are relatively simple and straightforward to implement in Fortran in a portable way as was shown. The power of the simulation methods in this work is mainly due to these techniques.

Due to the discreteness of the underlying lattice, certain physical results must be corrected in order to recover the continuum case. These lattice effect were successfully removed from the pair distribution function. The density profile results for the hard core fluid inside a parallel slit did not need extra correction because of the geometrical compatibility of the lattice and the slit. Detailed results with fine stepsize were given for narrow slits.

A lattice based geometrical method was developed to characterize the pore structure inside zeolites. This method is universal; it can be applied to arbitrary atomic systems. Originally this method was developed to visualize channel structures in elec-

trides. It is in every day use in this field, producing cavity and channel maps for these materials. These maps are very similar in shape and character to electron density maps which still require the power of supercomputers to be obtained. Even though the two maps are not exactly the same, the geometrical maps help to understand the physics of electrides. In the case of zeolites the geometrical maps are exactly what we need to see in order to understand their physical properties. This geometrical method was the first step in the development of this family of lattice based discrete techniques.

Ultimately the particles can be placed on the lattices describing the geometry of real physical enviromnents such as zeolites. The method is powerful enough to overcome certain limitations of Molecular Dynamics. With additional features such as chemical bonds for example, the applicability of the method widens even more. The new features however must be introduced with care in order not to lose too much of the computational efficiency.

The emphasis was placed on the computational methodology and the algorithms for the following reasons. Developing, implementing and testing new methods is always demanding. These methods now can help other studies. Many of these methods can be directly applied to other lattice based simulations, for example percolation problems and spin systems.

Without investigating the fundamental computational methods, it is difficult to reach large system sizes and long times even on today's powerful computers.

Appendix A

Source code for one module

In this appendix the source code for one of the modules is presented as an example, along with the parameter and status files. Each module has a unique name. There is a jobfile associated with each module with the same name and with the `.job` extension. The names of the jobs to be performed are placed in the job-file. The job-file can contain maximum 49 jobnames with maximum 30 characters each, one jobname per line without empty lines. The structure of the jobfile is obvious, it is not shown here. Each module reads its own jobfile at the very beginning of the run and then executes them one by one. The extension `.par` is added to each jobname, and the simulation parameters are read from that parameter file. If the parameter file does not exist or it is misplaced, the module stops with a fatal error. There is no procedure built into the modules to handle these types of situations with tolerance. (The error handling routine would be more complicated than the whole main Monte-Carlo loop.) The parameter files are slightly different for each module, but they are pretty similar in structure. The most important parameters – size of the simulation box, number of particles, particle radius – are always present in the parameter files. The parameter file for this module is shown below. The numbers, logical values and strings are read from the first column by the program module – everything else is ignored. We can

use this area for our comments and reminders as the example show.

```
1024        NX              X-Size of the lattice
1024        NY              Y-Size of the lattice
1100        NATOM           Number of atoms
12.0        RAD             Hardcore radius
1000        TSTEP           Number of ENSEMBLE steps
T           GENSEEDFLAG     T:gen your own seed, F:read next number
-23459      NSEED           Manual seed for RNG
T           GENCONFFLAG     T:gen your own config, F:read it from file
1024n1100r12.xy
1           IRS             Number of config-repeats (1,2,4,8)
1           CTRY            Constant for squeze trial
T           WRITECONFFLAG Output configuration to file (T:Yes,F:No)
8           NDIR            Number of directions
 1  0                       Right
 0  1                       Up
-1  0                       Left
 0 -1                       Down
 1  1                       Right-Up
 1 -1                       Right-Down
-1  1                       Left-Up
-1 -1                       Left-Down
```

After the parameters are read from the parameter file, they are echoed back to a log-file, to make sure each parameter has the correct value. The log-file has the name of the job with the `.log` extension. The log-file looks very similar to the parameter file; it is not shown here.

During the simulation a status file is created containing important information about the simulation run. The name of the status file is created from the name of the job with the `.sts` extension. The most important initial simulation parameters are repeated to this status file again. Then other important simulation results – particle densities, collision ratios – are written into this file. The number of particles successfully placed in the simulation box is not necessarily equal to the number given in the parameter file initially. If the planned density is too high, the program might not be able to place all the particles. The status file gives the number of particles finally placed, and the correct densities are calculated based on this number. The status file is shown here for this module.

```
--------------------------------
Program  : H2PIA
--------------------------------
Fundamental prameters
NX       :       1024
NY       :       1024
NATOM    :       1100
RAD      :   12.00000000
TSTEP    :       1000
--------------------------------
Densities
Numb dens:     0.00104904
Redu dens:     0.60424805
Area dens:     0.47457528
--------------------------------
Initial configuration distances
2*RAD    :   24.00000000
MINDO    :   24.00000000
MIND2O   :        576
--------------------------------
Final configuration distances
2*RAD    :   24.00000000
MINDO    :   24.00000000
MIND2O   :        576
--------------------------------
Final collisions
PCOL     :       169664
Coll rat%:    15.42400
--------------------------------
```

Every module generates result-files depending on what type of data was collected in the simulation. This example module performs only the simulation without collecting anything, so only the initial and final configurations of the particle system are written to files. These files simply contain the coordinates of the particles, they are not shown here.

Finally the Fortran source code for this module is given here. It is meant to be a well commented, easy-to-read code, but it probably still has many obscure points. The author would be happy to hear any suggestions, or to give detailed explanations. The source code of the other modules are also available. This module simulates hard core particles walking randomly in two dimensions using the field-representation method

with periodic boundary condition and isotropic direction generation. Many of the computational techniques discussed in this work can be found in the source code.

```
C-----------------------------------------------------------------------
C File: h2pia.for
C-----------------------------------------------------------------------
C h: hard core particles in Field-Representation
C 2: on a 2D square lattice
C p: with Periodic boundary condition
C i: Isotropic (=nonbiased) random walk
C a: version A: simulation only
C-----------------------------------------------------------------------
C Update: in every ENSEMBLE time step each atom takes its own
C step in the SAME order in every cycle.
C-----------------------------------------------------------------------
      PROGRAM    H2PIA
C-----------------------------------------------------------------------
      IMPLICIT   NONE

      INTEGER    LX,LY              !Sizes for memory-allocation
      PARAMETER  (LX=8191,LY=8191) !(255,511,1023,2047,4095,8191)
      INTEGER*1  F(0:LX,0:LY)       !F: particle cores and media walls
      INTEGER    NX,NY              !Actual sizes (N<=(L+1) must be)

      INTEGER    LATOM            !Number of atoms MINUS ONE
      PARAMETER  (LATOM=999999)   !to reserve in memory
      INTEGER    X(0:LATOM)       !X coordinates of atoms
      INTEGER    Y(0:LATOM)       !Y coordinates of atoms
      INTEGER    NATOM            !Actual number of atoms
                                  !(NATOM<=(LATOM+1) must be)

      REAL*4     RAD              !Radius of a particle
      INTEGER    TSTEP            !Num of ENSEMBLE timesteps to take

      LOGICAL    GENSEEDFLAG      !T:generate our own, F:read from file
      INTEGER    NSEED            !Seed for RNG

      LOGICAL    GENCONFFLAG      !T:generate our own config, F:read it
      CHARACTER  CONFFILENAME*30 !Name of the file where config is
      INTEGER    LENGCFN          !Length of the name above

      INTEGER    IRS         !How many times to repeat the configuration
                             !in both X and Y direction. (1,2,4,8,...)

      INTEGER    CTRY        !Trial Constant for initial state

      LOGICAL    WRITECONFFLAG   !T:write configuration F:don't write
```

```
      INTEGER    NUMBJOBS
      CHARACTER  NAMEJOBS(49)*30
      INTEGER    LENGJOBS(49)

C Randomwalk vectors
      INTEGER    NDIR            !Number of directions (1-8)
      INTEGER    DIRX(0:7)       !Direction table X component
      INTEGER    DIRY(0:7)       !Direction table Y component

C Hard core shapes
      INTEGER    NPNT            !Number of points in the shapetable
      INTEGER    SHPTBLX(8000)   !Table of the X shapecoordinates
      INTEGER    SHPTBLY(8000)   !Table of the Y shapecoordinates

C Hard core deltashapes
      INTEGER    NDP(0:7)        !Number of +1 delta elements
      INTEGER    DXP(125,0:7)    !Delta = +1, X coord
      INTEGER    DYP(125,0:7)    !Delta = +1, Y coord
      INTEGER    NDM(0:7)        !Number of -1 delta elements
      INTEGER    DXM(125,0:7)    !Delta = -1, X coord
      INTEGER    DYM(125,0:7)    !Delta = -1, Y coord

      INTEGER    I
C-----------------------------------------------------------------------
C Read jobfile
      CALL RDJOBNAMES(NUMBJOBS,NAMEJOBS,LENGJOBS)

C Execute jobs one by one
      DO I=1,NUMBJOBS

C Read parameter file
        CALL  READPAR(NAMEJOBS(I),LENGJOBS(I),NX,NY,NATOM,RAD,
     #  TSTEP,GENSEEDFLAG,NSEED,
     #  GENCONFFLAG,CONFFILENAME,LENGCFN,IRS,
     #  CTRY,WRITECONFFLAG,NDIR,DIRX,DIRY)

C Generate your own seed only when the flag says so
        IF (GENSEEDFLAG) THEN
          CALL GENSEED(NSEED)
        ENDIF

C Write log file for double check
        CALL WRITEPAR(NAMEJOBS(I),LENGJOBS(I),NX,NY,NATOM,RAD,
     #  TSTEP,GENSEEDFLAG,NSEED,
     #  GENCONFFLAG,CONFFILENAME,LENGCFN,IRS,
     #  CTRY,WRITECONFFLAG,NDIR,DIRX,DIRY)
```

```
C Generate 2D circular step functions
        CALL GEN2DCIR(2.0*RAD,NPNT,SHPTBLX,SHPTBLY)

C Generate Head and Tail points
        CALL GEN2DDEL(NPNT,SHPTBLX,SHPTBLY,NDIR,DIRX,DIRY,
     #  NDP,DXP,DYP,NDM,DXM,DYM)

C Core: main Monte Carlo routine
        CALL MCCORE(LX,LY,F,NX,NY,LATOM,X,Y,NATOM,RAD,TSTEP,
     #  NSEED,GENCONFFLAG,CONFFILENAME,LENGCFN,WRITECONFFLAG,
     #  IRS,CTRY,NDIR,DIRX,DIRY,NPNT,SHPTBLX,SHPTBLY,
     #  NDP,DXP,DYP,NDM,DXM,DYM,NAMEJOBS(I),LENGJOBS(I))

      ENDDO !I
C-----------------------------------------------------------------------
      STOP
      END
C-----------------------------------------------------------------------
      SUBROUTINE MCCORE(LX,LY,F,NX,NY,LATOM,X,Y,NATOM,RAD,TSTEP,
     #NSEED,GENCONFFLAG,CONFFILENAME,LENGCFN,WRITECONFFLAG,
     #IRS,CTRY,NDIR,DIRX,DIRY,NPNT,SHPTBLX,SHPTBLY,
     #NDP,DXP,DYP,NDM,DXM,DYM,NAME,LENG)
C-----------------------------------------------------------------------
C Main Monte Carlo routine.
C-----------------------------------------------------------------------
      IMPLICIT   NONE
C Input-Output:
      INTEGER    LX,LY          !Sizes for memory-allocation
      INTEGER*1  F(0:LX,0:LY)   !F: particle cores and media walls
      INTEGER    NX,NY          !Actual sizes (N<=(L+1) must be)

      INTEGER    LATOM          !Number of atoms MINUS ONE to allocate
      INTEGER    X(0:LATOM)     !X coordinates of atoms
      INTEGER    Y(0:LATOM)     !Y coordinates of atoms
      INTEGER    NATOM          !Actual number of atoms
                                !NATOM =< (LATOM+1) must be

      REAL       RAD            !Radius of the atom
      INTEGER    TSTEP          !Number of ENSEMBLE timesteps

      INTEGER    NSEED          !Seed for RNG

      LOGICAL    GENCONFFLAG     !T:generate our own config, F:read it
      CHARACTER  CONFFILENAME*30 !Name of the file where config is
      INTEGER    LENGCFN         !Length of the name above
```

```
      LOGICAL    WRITECONFFLAG   !T:write configuration F:don't write

      INTEGER    IRS         !How many times to repeat the configuration
                             !in both X and Y direction. (1,2,4,8,...)

      INTEGER    CTRY        !Trial Constant for initial state

C Randomwalk vectors
      INTEGER    NDIR            !Number of directions (1-8)
      INTEGER    DIRX(0:7)       !Direction table X component
      INTEGER    DIRY(0:7)       !Direction table Y component

C Hard core shapes
      INTEGER    NPNT            !Number of points in the shapetable
      INTEGER    SHPTBLX(8000)   !Table of the X shapecoordinates
      INTEGER    SHPTBLY(8000)   !Table of the Y shapecoordinates

C Hard core deltashapes
      INTEGER    NDP(0:7)        !Number of +1 delta elements
      INTEGER    DXP(125,0:7)    !Delta = +1, X coord
      INTEGER    DYP(125,0:7)    !Delta = +1, Y coord
      INTEGER    NDM(0:7)        !Number of -1 delta elements
      INTEGER    DXM(125,0:7)    !Delta = -1, X coord
      INTEGER    DYM(125,0:7)    !Delta = -1, Y coord

      CHARACTER  NAME*30
      INTEGER    LENG

C Functions:
      INTEGER    IAND
      REAL*4     RAN

C Local variables:
      INTEGER    MX,MY
      INTEGER    PCOL
      INTEGER    I,J,K,L
      INTEGER    WI,WJ,WD,JNS
      INTEGER    OLDX,OLDY,NEWX,NEWY,ZX,ZY
      INTEGER    D2MIN,MIND2O
      INTEGER    MAXTRY,ITRY
      INTEGER    LDATA,LSTAT
      REAL*4     PI,RW
      CHARACTER  WNAME*30
C-----------------------------------------------------------------------
C Init important variables
      PCOL  = 0          !Particle-particle collision counter
      LDATA = 10         !Logical unit for DATA files
```

```
      LSTAT = 11        !Logical unit for STATUS file
      PI    = 3.1415926
      D2MIN = INT((2.0*RAD)**2) !Min allowed distance square
C-----------------------------------------------------------------------
      MX = NX - 1     !Masks for bitwise operations
      MY = NY - 1
C-----------------------------------------------------------------------
C Clear main array just in case. (The devil never sleeps.)
      DO J=0,MY
        DO I=0,MX
          F(I,J) = 0
        ENDDO !I
      ENDDO !J
C-----------------------------------------------------------------------
C Initial configuration.
      IF (GENCONFFLAG) THEN
C Let's generate our OWN random initial configuration.
C (Try to set up NATOM piece of particles in the field without clash.)
        MAXTRY = CTRY*NATOM
        ITRY   = 0
        DO I=0,(NATOM-1)
  10      IF (ITRY.GT.MAXTRY) THEN
C The field is full enough, let's finish
            NATOM = I !Number of successfully placed atoms
            GOTO 20
          ENDIF
C Generate a random position
          NEWX = INT(NX*RAN(NSEED))
          NEWY = INT(NY*RAN(NSEED))
C Is it allowed to occupy it?
          IF (F(NEWX,NEWY).EQ.0) THEN
C YES: let's put the atom down
            ITRY = 0      !Reset trial counter
            X(I) = NEWX   !Store X coord
            Y(I) = NEWY   !Store Y coord
            DO J=1,NPNT   !Put the atom into the field
              ZX = IAND((NEWX+SHPTBLX(J)),MX)
              ZY = IAND((NEWY+SHPTBLY(J)),MY)
              F(ZX,ZY) = F(ZX,ZY) + 1
            ENDDO !J
          ELSE
C NO: try again, but let's count the trial
            ITRY = ITRY + 1
            GOTO 10
          ENDIF
        ENDDO !I
 20     CONTINUE
```

```
      ELSE
C-----------------------------------------------------------------------
C Let's read the configuration from an external file.
        WNAME = CONFFILENAME(1:LENGCFN)
        OPEN(UNIT=LDATA,FILE=WNAME,STATUS='UNKNOWN')
        I = 0
 56     READ(LDATA,*,ERR=57,END=57) X(I),Y(I)
        I = I+1
        GOTO 56
 57     CLOSE(LDATA)
        NATOM = I
C Let's generate field based on the particle coordinates
        DO I=0,(NATOM-1)
          DO J=1,NPNT
            ZX = IAND((X(I)+SHPTBLX(J)),MX)
            ZY = IAND((Y(I)+SHPTBLY(J)),MY)
            F(ZX,ZY) = F(ZX,ZY) + 1
          ENDDO
        ENDDO
      ENDIF
C-----------------------------------------------------------------------
C Let's repeat the configuration and then correct the simulation
C box parameters accordingly.
      IF (IRS.GT.1) THEN
C Repeat coordinates in X direction first
        DO J=1,(IRS-1)
          WJ  = J*NATOM
          JNS = J*NX
          DO I=0,(NATOM-1)
            WI = I + WJ
            X(WI) = X(I) + JNS
            Y(WI) = Y(I)
          ENDDO !I
        ENDDO !J
C Repeat field in X direction
        DO J=1,(IRS-1)
          JNS = J*NX
          DO K=0,MY
            DO L=0,MX
              F((JNS+L),K) = F(L,K)
            ENDDO !L
          ENDDO !K
        ENDDO !J
C Correct the number of atoms accordingly
        NATOM = IRS*NATOM
C Correct the size in X after the repeat
        NX = IRS*NX
```

```
        MX = NX - 1
C Repeat coordinates in Y direction
        DO J=1,(IRS-1)
          WJ  = J*NATOM
          JNS = J*NY
          DO I=0,(NATOM-1)
            WI = I + WJ
            X(WI) = X(I)
            Y(WI) = Y(I) + JNS
          ENDDO !I
        ENDDO !J
C Repeat field in Y direction
        DO J=1,(IRS-1)
          JNS = J*NY
          DO K=0,MY
            DO L=0,MX
              F(L,(JNS+K)) = F(L,K)
            ENDDO !L
          ENDDO !K
        ENDDO !J
C Correct the number of atoms accordingly
        NATOM = IRS*NATOM
C Correct the size in Y after the repeat
        NY = IRS*NY
        MY = NY - 1
      ENDIF
C-----------------------------------------------------------------------
C Calculate the minimum particle-particle distance for status report.
C Calculates only for open boundary condition!
C O(N^2) process, but paranoia is paranoia.
      MIND2O = NX*NX + NY*NY
      DO I=0,(NATOM-2)
        OLDX = X(I)
        OLDY = Y(I)
        DO J=(I+1),(NATOM-1)
          ZX = OLDX-X(J)
          ZY = OLDY-Y(J)
          MIND2O = MIN(MIND2O,(ZX*ZX + ZY*ZY))
        ENDDO
      ENDDO
C-----------------------------------------------------------------------
C Open status file and write important parameters
      WNAME = NAME(1:LENG)//'.sts'
      OPEN(UNIT=LSTAT,FILE=WNAME,STATUS='UNKNOWN')
 8    FORMAT(1X,A,I10)
 9    FORMAT(1X,A,F14.8)
      WRITE(LSTAT,*) 'Program  : H2PIA'
```

```
      WRITE(LSTAT,*) '---------------------------'
      WRITE(LSTAT,*) 'Fundamental prameters'
      WRITE(LSTAT,8) 'NX       : ',NX
      WRITE(LSTAT,8) 'NY       : ',NY
      WRITE(LSTAT,8) 'NATOM    : ',NATOM
      WRITE(LSTAT,9) 'RAD      : ',RAD
      WRITE(LSTAT,8) 'TSTEP    : ',TSTEP
      WRITE(LSTAT,*) '---------------------------'
      WRITE(LSTAT,*) 'Densities'
      RW = FLOAT(NATOM)/FLOAT(NX)/FLOAT(NY)
      WRITE(LSTAT,9) 'Numb dens: ',RW
      WRITE(LSTAT,9) 'Redu dens: ',4.0*(RAD**2)*RW
      WRITE(LSTAT,9) 'Area dens: ',PI*(RAD**2)*RW
      WRITE(LSTAT,*) '---------------------------'
      WRITE(LSTAT,*) 'Initial configuration distances'
      IF (MIND2O.LT.D2MIN) THEN
        WRITE(LSTAT,*) '***************** WARNING !!! Particle clash.'
      ENDIF
      WRITE(LSTAT,9) '2*RAD    : ',2.0*RAD
      WRITE(LSTAT,9) 'MINDO    : ',SQRT(FLOAT(MIND2O))
      WRITE(LSTAT,8) 'MIND2O   : ',MIND2O
C-------------------------------------------------------------------------
C Output: Initial configuration if it is requested.
      IF (WRITECONFFLAG) THEN
        WNAME = NAME(1:LENG)//'.xyini'
        OPEN(UNIT=LDATA,FILE=WNAME,STATUS='UNKNOWN')
        DO I=0,(NATOM-1)
          WRITE(LDATA,'(I4,1X,I4)') X(I),Y(I)
        ENDDO
        CLOSE(LDATA)
      ENDIF
C-------------------------------------------------------------------------
C Main Monte-Carlo loop starts here.
C-------------------------------------------------------------------------
C Init particle-particle collision counter
      PCOL = 0
C Main double loop starts here
C Timeloop first
      DO J=1,TSTEP
C Atomloop second
        DO WI=0,(NATOM-1)
          WD   = INT(NDIR*RAN(NSEED))
          OLDX = X(WI)
          OLDY = Y(WI)
          NEWX = IAND((OLDX + DIRX(WD)),MX)
          NEWY = IAND((OLDY + DIRY(WD)),MY)
          IF (F(NEWX,NEWY).LT.2) THEN
```

```
C Geometry allows the step, so let's take it
            X(WI)  = NEWX
            Y(WI)  = NEWY
C Move particle hard core, head points first
            DO I=1,NDP(WD)
              ZX = IAND((OLDX + DXP(I,WD)),MX)
              ZY = IAND((OLDY + DYP(I,WD)),MY)
              F(ZX,ZY) = F(ZX,ZY) + 1
            ENDDO !I
C Then hard core tail points
            DO I=1,NDM(WD)
              ZX = IAND((OLDX + DXM(I,WD)),MX)
              ZY = IAND((OLDY + DYM(I,WD)),MY)
              F(ZX,ZY) = F(ZX,ZY) - 1
            ENDDO !I
          ELSE
C Collision with another atom, so let's count it
            PCOL = PCOL + 1
          ENDIF
        ENDDO !WI
      ENDDO !J
C-----------------------------------------------------------------------
C Calculate the minimum particle-particle distance for status report.
C Calculates only for open boundary condition!
C O(N^2) process.
      MIND2O = NX*NX + NY*NY
      DO I=0,(NATOM-2)
        OLDX = X(I)
        OLDY = Y(I)
        DO J=(I+1),(NATOM-1)
          ZX = OLDX-X(J)
          ZY = OLDY-Y(J)
          MIND2O = MIN(MIND2O,(ZX*ZX + ZY*ZY))
        ENDDO
      ENDDO
C-----------------------------------------------------------------------
C Output: Status file
      WRITE(LSTAT,*) '---------------------------'
      WRITE(LSTAT,*) 'Final configuration distances'
      IF (MIND2O.LT.D2MIN) THEN
        WRITE(LSTAT,*) '***************** WARNING !!! Particle clash.'
      ENDIF
      WRITE(LSTAT,9) '2*RAD   : ',2.0*RAD
      WRITE(LSTAT,9) 'MINDO   : ',SQRT(FLOAT(MIND2O))
      WRITE(LSTAT,8) 'MIND2O  : ',MIND2O
      WRITE(LSTAT,*) '---------------------------'
      WRITE(LSTAT,*) 'Final collisions'
```

```
      WRITE(LSTAT,*) 'PCOL      : ',PCOL
      WRITE(LSTAT,*) 'Coll rat%: ',
     #100.0*FLOAT(PCOL)/FLOAT(TSTEP)/FLOAT(NATOM)
      CLOSE(LSTAT)
C-----------------------------------------------------------------------
C Output: Final configuration if it is requested
      IF (WRITECONFFLAG) THEN
        WNAME = NAME(1:LENG)//'.xyfin'
        OPEN(UNIT=LDATA,FILE=WNAME,STATUS='UNKNOWN')
        DO I=0,(NATOM-1)
          WRITE(LDATA,'(I4,1X,I4)') X(I),Y(I)
        ENDDO
        CLOSE(LDATA)
      ENDIF
C-----------------------------------------------------------------------
      RETURN
      END
C-----------------------------------------------------------------------
      SUBROUTINE GEN2DCIR(R,NPNT,X,Y)
C-----------------------------------------------------------------------
C Routine to generate 2D circular step function table with radius R.
C Radius must be less than 50 lattice units.
C-----------------------------------------------------------------------
      IMPLICIT   NONE
C Input:
      REAL*4     R          !Real radius
C Output:
      INTEGER    NPNT       !Number of points in the table
      INTEGER    X(8000)    !Table of the X coordinates
      INTEGER    Y(8000)    !Table of the Y coordinates
C Local variables:
      REAL*4     R2,D
      INTEGER    I,J,I2,J2,C
C-----------------------------------------------------------------------
C Chekpoint: You never know what comes from the parameter file.
      IF (R.GE.50.0) THEN
        WRITE(6,*) 'Fatal Error!'
        WRITE(6,*) 'Radius is too big: ',R
        STOP
      ENDIF
C-----------------------------------------------------------------------
      R2    = R*R
      NPNT  = 0
      C     = INT(R)
C-----------------------------------------------------------------------
C Generate points on the lattice in a pattern of square and check the
C distance from the origin. If it is inside the radius then put it in
```

```
C the table.
C
C Points lying exactly on the boundary are excluded!
C
C The order of the indeces are good (in Fortran) for efficient linear
C memory access, no further sorting is necessary.
      DO J=-C,C
        J2 = J*J
        DO I=-C,C
          I2 = I*I
          D = FLOAT(I2 + J2)
          IF (D.LT.R2) THEN     !.LT. throws out boundary points
            NPNT    = NPNT + 1
            X(NPNT) = I
            Y(NPNT) = J
          ENDIF
        ENDDO !I
      ENDDO !J
      RETURN
      END
C-----------------------------------------------------------------------
      SUBROUTINE GEN2DDEL(NPNT,X,Y,NDIR,DIRX,DIRY,
     #           NDP,DXP,DYP,NDM,DXM,DYM)
C-----------------------------------------------------------------------
C Routine to generate 2D delta table of head and tail points for motion.
C (No explicit radius is present here, but it must be less than
C 50 units to fit in the table without outflow.)
C-----------------------------------------------------------------------
      IMPLICIT   NONE
C Input:
      INTEGER    NPNT            !Number of points
      INTEGER    X(8000)         !Table of the X coordinates
      INTEGER    Y(8000)         !Table of the Y coordinates
      INTEGER    NDIR            !Number of directions (1-8)
      INTEGER    DIRX(0:7)       !Direction table X
      INTEGER    DIRY(0:7)       !Direction table Y
C Output:
      INTEGER    NDP(0:7)        !Number of +1 delta elements (head)
      INTEGER    DXP(125,0:7)    !Delta = +1, X coord
      INTEGER    DYP(125,0:7)    !Delta = +1, Y coord
      INTEGER    NDM(0:7)        !Number of -1 delta elements (tail)
      INTEGER    DXM(125,0:7)    !Delta = -1, X coord
      INTEGER    DYM(125,0:7)    !Delta = -1, Y coord
C Local variables:
      INTEGER    C
      PARAMETER  (C=51)
      INTEGER    W(-C:C,-C:C)
```

```
      INTEGER    I,J,N,IX,IY
C-----------------------------------------------------------------------
C Let's go through all the possible directions.
      DO N=0,(NDIR-1)
        NDP(N) = 0
        NDM(N) = 0
        DO J=-C,C
          DO I=-C,C
            W(I,J) = 0
          ENDDO !J
        ENDDO !I
        DO I=1,NPNT
          IX = X(I)
          IY = Y(I)
          W(IX,IY) = W(IX,IY) + 1  !Points without shift
          IX = IX + DIRX(N)
          IY = IY + DIRY(N)
          W(IX,IY) = W(IX,IY) + 2  !Shifted points
        ENDDO !NPNT
C The index-order is good in Fortran.
        DO J=-C,C
          DO I=-C,C
            IF (W(I,J).EQ.1) THEN          !Tail points: Minus
              NDM(N) = NDM(N) + 1
              DXM(NDM(N),N) =  I
              DYM(NDM(N),N) =  J
            ELSEIF (W(I,J).EQ.2) THEN      !Head points: Plus
              NDP(N) = NDP(N) + 1
              DXP(NDP(N),N) =  I
              DYP(NDP(N),N) =  J
            ENDIF
          ENDDO !I
        ENDDO !J
      ENDDO !NDIR
      RETURN
      END
C-----------------------------------------------------------------------
      SUBROUTINE RDJOBNAMES (NUMBJOBS,NAMEJOBS,LENGJOBS)
C-----------------------------------------------------------------------
C Read jobnames and calculate lengths.
C Maximum number of jobs: 49, jobnamelengths: 30.
C-----------------------------------------------------------------------
      IMPLICIT     NONE
C Output:
      INTEGER      NUMBJOBS
      CHARACTER*30 NAMEJOBS(49)
      INTEGER      LENGJOBS(49)
```

```
C Local variables:
      INTEGER      LIN,I
      INTEGER      WLENG
      CHARACTER*30 WNAME
C-----------------------------------------------------------------------
      LIN = 10
      OPEN(UNIT=LIN,FILE='h2pia.job',STATUS='OLD')
      I = 1
 1    READ (LIN,'(A30)',ERR=99,END=99) NAMEJOBS(I)
      I = I+1
      GOTO 1
 99   CLOSE(LIN)
      NUMBJOBS = I-1
      DO I=1,NUMBJOBS
        WLENG = 30
        WNAME = NAMEJOBS(I)
        DO WHILE (WNAME(WLENG:WLENG).EQ.' ')
          WLENG = WLENG - 1
        ENDDO
        LENGJOBS(I) = WLENG
      ENDDO
      RETURN
      END
C-----------------------------------------------------------------------
      SUBROUTINE READPAR(FILNAM,LENG,NX,NY,NATOM,RAD,
     #TSTEP,GENSEEDFLAG,NSEED,
     #GENCONFFLAG,CONFFILENAME,LENGCFN,IRS,
     #CTRY,WRITECONFFLAG,NDIR,DIRX,DIRY)
C-----------------------------------------------------------------------
C Routine to read parameter file and do some data preprocessing.
C-----------------------------------------------------------------------
      IMPLICIT   NONE
C Input:
      CHARACTER  FILNAM*30
      INTEGER    LENG
C Output:
      INTEGER    NX,NY,NATOM
      REAL*4     RAD
      INTEGER    TSTEP
      LOGICAL    GENSEEDFLAG
      INTEGER    NSEED
      LOGICAL    GENCONFFLAG
      CHARACTER  CONFFILENAME*30
      INTEGER    LENGCFN
      INTEGER    IRS
      INTEGER    CTRY
      LOGICAL    WRITECONFFLAG
```

```
      INTEGER    NDIR              !Number of directions (1-8)
      INTEGER    DIRX(0:7)         !Direction table X component
      INTEGER    DIRY(0:7)         !Direction table Y component
C Local variables:
      INTEGER    I,LIO
C-----------------------------------------------------------------------
      LIO = 10
      OPEN(UNIT=LIO,FILE=FILNAM(1:LENG)//'.par',STATUS='OLD')
C Parameters are read
      READ(LIO,*) NX
      READ(LIO,*) NY
      READ(LIO,*) NATOM
      READ(LIO,*) RAD
      READ(LIO,*) TSTEP
      READ(LIO,*) GENSEEDFLAG
      READ(LIO,*) NSEED
      READ(LIO,*) GENCONFFLAG
      READ(LIO,'(A30)') CONFFILENAME
      READ(LIO,*) IRS
      READ(LIO,*) CTRY
      READ(LIO,*) WRITECONFFLAG
C Direction table for randomwalk is read here
      READ(LIO,*) NDIR
      DO I=0,(NDIR-1)
        READ(LIO,*) DIRX(I),DIRY(I)
      ENDDO !I
      CLOSE(LIO)
C Calculate the length of the config-file-name
      LENGCFN = 30
      DO WHILE (CONFFILENAME(LENGCFN:LENGCFN).EQ.' ')
        LENGCFN = LENGCFN - 1
      ENDDO
C We are done
      RETURN
      END
C-----------------------------------------------------------------------
      SUBROUTINE WRITEPAR(FILNAM,LENG,NX,NY,NATOM,RAD,
     #TSTEP,GENSEEDFLAG,NSEED,
     #GENCONFFLAG,CONFFILENAME,LENGCFN,IRS,
     #CTRY,WRITECONFFLAG,NDIR,DIRX,DIRY)
C-----------------------------------------------------------------------
C Routine to echo parameters to the log-file just in case.
C-----------------------------------------------------------------------
      IMPLICIT   NONE
C Input:
      CHARACTER  FILNAM*30
      INTEGER    LENG
```

```
C Output:
      INTEGER    NX,NY,NATOM
      REAL*4     RAD
      INTEGER    TSTEP
      LOGICAL    GENSEEDFLAG
      INTEGER    NSEED
      LOGICAL    GENCONFFLAG
      CHARACTER  CONFFILENAME*30
      INTEGER    LENGCFN
      INTEGER    IRS
      INTEGER    CTRY
      LOGICAL    WRITECONFFLAG
      INTEGER    NDIR             !Number of directions (1-8)
      INTEGER    DIRX(0:7)        !Direction table X component
      INTEGER    DIRY(0:7)        !Direction table Y component
C Local variables:
      INTEGER    I,LIO
C-----------------------------------------------------------------------
      LIO = 10
      OPEN(UNIT=LIO,FILE=FILNAM(1:LENG)//'.log',STATUS='UNKNOWN')
C Parameters are written
      WRITE(LIO,*) 'Program      : H2PIA'
      WRITE(LIO,*) 'FILNAM       : ',FILNAM
      WRITE(LIO,*) 'LENG         : ',LENG
      WRITE(LIO,*) 'NX           : ',NX
      WRITE(LIO,*) 'NY           : ',NY
      WRITE(LIO,*) 'NATOM        : ',NATOM
      WRITE(LIO,*) 'RAD          : ',RAD
      WRITE(LIO,*) 'TSTEP        : ',TSTEP
      WRITE(LIO,*) 'GENCONFFLAG  : ',GENCONFFLAG
      WRITE(LIO,*) 'CONFFILENAME : ',CONFFILENAME
      WRITE(LIO,*) 'LENGCFN      : ',LENGCFN
      WRITE(LIO,*) 'IRS          : ',IRS
      WRITE(LIO,*) 'CTRY         : ',CTRY
      WRITE(LIO,*) 'WRITECONFFLAG: ',WRITECONFFLAG
      WRITE(LIO,*) 'NDIR         : ',NDIR
C Direction table for randomwalk is written here
      DO I=0,(NDIR-1)
        WRITE(LIO,*) 'I: ',I,' DIR: ',DIRX(I),DIRY(I)
      ENDDO !I
C We are done.
      CLOSE(LIO)
      RETURN
      END
C-----------------------------------------------------------------------
      SUBROUTINE GENSEED(SEED)
C-----------------------------------------------------------------------
```

```
C Subroutine to generate seed for random number generators from
C the system clock.
C-----------------------------------------------------------------------
      IMPLICIT   NONE
C Output:
      INTEGER    SEED          !Seed
C Local:
      CHARACTER  TM*8          !Time string
      INTEGER    D1,D2,D3      !Work variables
C-----------------------------------------------------------------------
C Call time routine. (Not F77 standard, but it is very common routine.)
      CALL TIME(TM)
C Break string 'HH:MM:SS' into separated integer variables.
      READ(TM,'(I2,1X,I2,1X,I2)') D1,D2,D3
C Build integer seed in the form of: SSHHMMSS.
      NEGSEED = D3*1000000 + D1*10000 + D2*100 + D3
      RETURN
      END
C-----------------------------------------------------------------------
C23456789012345678901234567890123456789012345678901234567890123456789012
C-----------------------------------------------------------------------
```

Bibliography

[1] B.J. Alder, T.E. Wainwright, Studies in Molecular Dynamics. I. General Method. *J.Chem.Phys.*, **31**, p459-466 (1959).

[2] L. Verlet, Computer Experiments on Classical Fluids I. *Phys.Rev.*, **159**, p98-103 (1967).

[3] W.H. Beyer (editor), CRC Standard Mathematical Tables, 27th edition, [CRC Press, Boca Raton, Florida, 1989].

[4] M.P. Allen, D.J. Tildesley, Computer Simulation of Liquids, [Clarendon Press, Oxford, 1987].

[5] D.C. Rapaport, The Art of Molecular Dynamics Simulation, [Cambdidge University Press, 1995].

[6] D. Frenkel, B. Smit, Understanding Molecular Simulations, From Algorithms to Applications, [Academic Press, San Diego, CA, 1996].

[7] G. Ciccotti, D. Frenkel, I.R. McDonald, Simulations of Liquids and Solids, Molecular Dynamics and Monte Carlo Methods in Statistical Mechanics, [North-Holland, Amsterdam, 1987].

[8] H. Gould, J. Tobochnik, An Introduction to Computer Simulation Methods, Applications to Physical Systems, Part 1, [Addison-Wesley, Reading, Massachusetts, 1988].

[9] T. Campbell, R.K. Kalia, A. Nakano, F. Shimojo, K. Tsuruta, P. Vashista, S. Ogata, *it Phys.Rev.Lett.*, V82, N20, p4018, (1999).

[10] T. Campbell, R.K. Kalia, A. Nakano, P. Vashista, S. Ogata, S. Rodgers, *Phys.Rev.Lett.*, V82, N24, p4866, (1999).

[11] P. Vashista, Multimillion Atom Molecular Dynamics Simulations of Ceramic Materials and Interfaces on Parallel Computers, OC40 2, APS March Meeting, Atlanta, 1999.

[12] D. Hirshfeld, D.C. Rapaport, *Phys.Rev.Lett.*, V80, N24, p5337, (1998).

[13] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes, The Art of Scientific Computing, with CDROM ver. 2.06, (250 Mbytes of validated random bytes generated by a physical white noise process) [Cambridge University Press, 1994].

[14] N.G. Cooper (editor), Los Alamos Science, **N15**, 1987.

[15] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of State Calculations by Fast Computing Machines, *J.Chem.Phys.*, **21**, p1087-1092, (1953).

[16] S.S. Rao, The Finite Element Method in Engineering, 2nd edition, [Pergamon Press, New-York, NY, 1989]

[17] G.D. Doolen, Lattice-Gas Methods for PDE's, Theory, Applications and Hardware, Physica **D 47**, (1991).

[18] J. Hardy, Y. Pomeau, *J.Math.Phys.*, **13**, p1042, (1972), J. Hardy, Y. Pomeau, O. de Pazzis, *J.Math.Phys.*, **14**, p1746, (1973), J. Hardy, O. de Pazzis, Y. Pomeau, *Phys. Rev. A*, **13**, p1949, (1976).

[19] S. Wolfram, Cellular Automata and Complexity (collected papers), [Addison-Wesley, Reading, Massachusetts, 1994].

[20] U. Frish, B. Hasslacher, Y. Pomeau, Lattice-Gas Automata for the Navier-Stokes equation, *Phys.Rev.Letters*, **56**, N14, p1505, (1986).

[21] D.H. Rothman, S. Zaleski, Lattice-Gas models of phase separation: interfaces, phase transitions, and multiphase flow, *Rev.Mod.Phys*, **V66**, N4, p1417, (1994), D.H. Rothman, S. Zaleski, Lattice-Gas Cellular Automata, Simple models of complex hydrodynamics, [Cambridge University Press, 1997].

[22] B. Mayer, G. Kohler, S. Rasmussen, *Phys.Rev.E.*, V55, N4, p4489, (1997).

[23] A. Bunde, S. Havlin (editors), Fractals and Disordered Systems, (second revised and enlarged edition) [Springer, Berlin, 1996], Chapter 9: D. Stauffer, Cellular Automata, p361.

[24] J.F. Traub, A Continuous Model of Computation, *Physics Today*, p39, May 1999.

[25] T.F. Nagy, S.D. Mahanti, C. Tsallis, Correlation function studies on the Domany-Kinzel cellular automaton, *Physica A*, **250**, p345, (1998).

[26] J. von Neumann, The Computer and the Brain, [Yale University Press, New Haven, 1958], W. Aspray, A. Burks (editors), Papers of John von Neumann on Computing and Computer Theory, [The MIT Press, Cambridge, Massachusetts, 1987].

[27] D. Hillis, The Connection Machine, [The MIT Press, Cambridge, Massachusetts, 1992].

[28] J. von Neumann, Theory of Self-Reproducing Automata, [University of Illinois Press, Urbana, 1966].

[29] G.S. Almasi, A. Gottlieb, Highly Parallel Computing, 2nd edition, [The Benjamin/Cummings Publishing Company Inc., Redwood City, California, 1994]

[30] J.S. Evans, R.H. Eckhouse, Alpha RISC Architecture for Programmers, [Prentice Hall, New Jersey, 1999].

[31] R. Bhargava, OpenVMS, [McGraw-Hill, New-York, 1995].

[32] D.E. Knuth, The Art of Computer Programming, volume 3, Sorting and Searching, 2nd edition, [Addison-Wesley, Reading, Massachusetts, 1998].

[33] J.M. Perez-Jorda, Computer Physics Comm. **108**, p1-8, (1998).

[34] K. Dowd, C.R. Severance, High Performance Computing, 2nd edition, [O'Reilly and Associates, Sebastopol, California, 1998].

[35] K.R. Wadleigh, I.L. Crawford, Software Optimization for High Performance Computing, [Prentice Hall, New Jersey, 2000].

[36] D. Loshin, High Performance Computing Demistified, [AP Professional, Boston, 1994].

[37] A. Kruger, Efficient FORTRAN Programming, [John Wiley and Sons, New-York, 1990].

[38] R.W. Hockney, J.W. Eastwood, Computer Simulations Using Particles, [McGraw-Hill, New-York, 1981].

[39] D.E. Knuth, The Art of Computer Programming, volume 2, Seminumerical Algorithms, 2nd edition, [Addison-Wesley, Reading, Massachusetts, 1998].

[40] E. Arapaki, P. Argyrakis, I. Avramov, A. Milchev, *Phys.Rev.E.*, V56, N1, pR29, (1997).

[41] S. Caser, H.J. Hilhorst, *J.Phys.A.*, **28**, p3887-3900, (1995).

[42] J.S. Wang, *Physica A*, **254**, p179-184, (1998).

[43] J.W. Evans, *Rev.Mod.Phys.*, V65, N4, p1281, (1993).

[44] J.A. Barker, D. Henderson, What is "liquid"? Understanding the states of matter, *Rev.Mod.Phys.*, V48, N4, p587-671, (1976).

[45] M.A. Herkommer, Number Theory: A Programmer's Guide, [McGraw-Hill, New-York, 1999].

[46] Y. Uehara, T. Ree, F.H. Ree, *J.Chem.Phys.*, **70**, N4, p1876, (1979).

[47] I.K. Snook, D. Henderson, *J.Chem.Phys.*, V68, N5, p2134, (1978).

[48] E. Kierlik, M.L. Rosinberg, *Phys.Rev.A.*, V42, N6, p3382, (1990).

[49] D. Henderson (editor), Fundamentals of Inhomogeneous Fluids, [Marcel Dekker, Inc. New-York, 1992].

[50] T.F. Nagy, S.D. Mahanti, *to be published.*

[51] E. Akhmatskaya, B.D. Todd, P.J. Divis, D.J. Evans, K.E. Gubbins, L.A. Pozhar, (preprint, 1997).

[52] M. Merkel, H. Lowen, *Phys.Rev.E.*, V54, N6, p6623, (1996).

[53] B. Gotzelmann, S. Dietrich, cond-mat/9610203, (1996).

[54] P. Tarazona, *Phys.Rev.A.*, V31, N4, p2672, (1985).

[55] B. Gotzelmann, A. Haase, S. Dietrich, *Phys.Rev.E.*, V53, N4, p3456, (1996).

[56] J.A. White, A. Gonzalez, F.L. Roman, S. Valesco, *Phys.Rev.Lett.*, V84, N6, p1220, (2000).

[57] DFT Plus, Models Library, User's Guide, v1.00, (1996).

[58] M. Bender, R. Klein, A. Disch, A. Ebert, *IEEE Trans.Vis.Comp.Graph.*, V6, N1, p8, (2000).

[59] J. Karger, D.M Ruthven, Diffusion in Zeolites and other microporous solids, [A Wiley-Interscience Publication, John Wiley and Sons, New-York, 1992].

[60] T.J. Pinnavaia, M.F. Thorpe (editors), Access in Nanoporous Materials, [Plenum Press, New-York, 1995].

[61] L. Abrams, D.R. Corbin, *J.Incl.Phen.Mol.Rec.Chem.*, **21**, p1-46, (1995).

[62] T.F. Nagy, S.D. Mahanti, J.L. Dye, Computer modeling of pore space in zeolites, *Zeolites*, 19:57-64, (1997).

[63] J.M. Newsam, *J.Phys.Chem.* **93**, p7689-7694, (1989).

[64] M.E. Leonowicz, J.A. Lawton, S.L. Lawton, K.M. Rubin, *Science* **264**, p1910-1913, (1994).

[65] P.A. Wright, J.M. Thomas, A.K. Cheetham, A.K. Nowak, *Nature* V318, 19/26, p611, (1985).

[66] H. van Koningsveld, H. van Bekkum, J.C. Jansen, *Acta Crystallogr.* **B43**, p127-132, (1987).

[67] D.H. Olson, *J.Phys.Chem.* **74**, p2758-2764, (1970).

[68] J.L. Dye, M.J. Wagner, G.T. Overney, R.H. Huang, T.F. Nagy, D. Tománek, *J.Am.Chem.Soc.* **118**, p7329-7336, (1996).

[69] M.J. Wagner, J.L. Dye, *Solid State Chem.* **117**, p309-317, (1995).

[70] L. Echegoyen, A.E. Kaifer (editors), Physical Supramolecular Chemistry, J.L. Dye, p313-336, (1996).

[71] J.L. Dye, *Inorganic Chemistry*, Vol. 36, N. 18, p3816, (1997).

[72] R.H. Huang, M.J. Wagner, D.J. Gilbert, K.A. Reidy-Cedergren, D.L. Ward, M.K. Faber, J.L. Dye, *J.Am.Chem.Soc.*, bf 119, p3765-3772, (1997).

[73] J.L. Dye, *Macromol. Symp.* **134**, p29-39, (1998).

[74] J.E. Hendrickson, W.P. Pratt, Jr., R.C. Phillips, J.L. Dye, *J.Phys.Chem.B.*, Vol. 102, No. 20, p3917, (1998).

[75] J. Kim, A.S. Ichimura, R.H. Huang, M. Redko, R.C. Phillips, J.E. Jackson, J.L. Dye, *J.Am.Chem.Soc.*, **121**, p10666, (1999).

[76] M.J. Wagner, A.S. Ichimura, R.H. Huang, R.C. Phillips, James L. Dye, *J.Phys.Chem.B.*, Vol. 104, No. 5, p1078, (2000).

[77] S.S. Skiena, The Algorithm Design Manual, [Springer-Verlag, New-York, 1998].

[78] Application Visualization Systems (AVS) User's Manual v3.0, Startdent Computer Inc., (1991).

[79] Iris Explorer User's Guide v2.2, Silicon Graphics Inc., (1994).

[80] J.A. Sethian, *American Scientist*, V85, p254, (May-June, 1997).

[81] W.M. Meier, D.H. Olson, C. Baerlocher, Atlas of Zeolite Structure Types, *Zeolites*, **17**, (1996).

[82] G.B. Woods, J.S. Rowlinson, *J.Chem.Soc., Faraday Trans. 2*, **85**(6), p765, (1989).

[83] S. Yashonath, J.M. Thomas, A.K. Nowak, A.K. Cheetham, *Nature*, V331, p601, 18 February 1988.

[84] H. Jobic, J. Karger, M. Bee, *Phys.Rev.Lett.*, V82, N21, p4260, (1999).

[85] C. Saravanan, F. Jousse, S.M. Auerbach, *Phys.Rev.Lett.*, V80, N26, p5754, (1998).

[86] C. Saravanan, S.M. Auerbach, *J.Chem.Phys.*, **107**, (19), p8120, 15 November 1997, C. Saravanan, S.M. Auerbach, *J.Chem.Phys.*, **107**, (19), p8132, 15 November 1997.

[87] T.F. Nagy, S.D. Mahanti, *J.Chem.Phys.* **106**, (15), p6511, (1997).

[88] M.I. Khan, L.M. Meyer, R.C. Haushalter, A.L. Schweitzer, J. Zubeita, J.L. Dye, *Chem.Mater.*, Vol. 8, No. 1, p43, (1996).