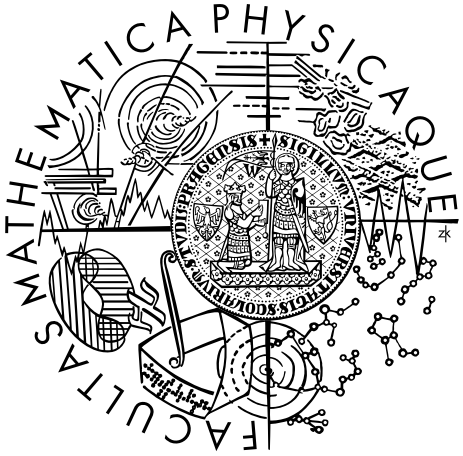
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Miroslav Tomasik

Simulation of two-dimensional flow past obstacles using lattice-gas cellular automata

Institute of Theoretical Physics

Supervisor of the master thesis: Martin Scholtz
Study programme: Physics
Study branch: Mathematical modelling

Prague 2017

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Simulation of two-dimensional flow past obstacles using lattice-gas cellular automata

Author: Miroslav Tomasik

Institute or Department: Institute of Theoretical Physics

Supervisor: Martin Scholtz, Institute of Theoretical Physics

Abstract: Cellular automata constitutes original computational methods, that found its application in various scientific disciplines. The special class of cellular automata, the lattice gas automata were succesfull in dealing with many challenges of hydrodynamic simulations, and they bootstraped one of the most perspective CFD methods, the Lattice Boltzmann models.

In the theoretical part, we follow the evolution of the lattice gas automata, explore the theory behind them, and from their microdynamics, we derive the hydrodynamic equations.

In the practical part, we implemented two most distinguished types of LGCA, the Pair-interaction automaton and FCHC. We applied them on the flow around obstacles of various shapes. The scientifically most relevant part concerns statistical properties of the turbulent flow simmulated by LGCA, but requires further research to conclude it.

Keywords: Cellular automata FCHC Pair-interaction Turbulence

Contents

How to use this diploma thesis

For the beginner in this field, we recommend to follow sequence of the chapters, since the theoretical part constitutes tutorial to lattice-gas cellular automata, but an experienced user can feel free to chose the topic he finds interesting.

In chapter one, we introduce the notion of cellular automata.

In chapter two, we continue with the special type cellular automata, the lattice-gas CA, that represents the original approach in CFD.

In chapter three, we present the better-known branch of LCCA, that started with FHP in two-dimensions and FCHC in three-dimensions.

In chapter four, we inspected microdynamics of FHP more theoretically and generally, so that the obtained results are valid for N-dimensional FHP-like LGCA, namely FCHC.

In chapter five, we derive macroscopic equations for FHP and FCHC.

In chapter six, we analyzed FCHC lattice and sketched FCHC collision algorithm as proposed by Henon.

In chapter seven, we introduced another successful branch of LGCA, the Pair-Interaction models. They are more comfortable to implement then FCHC in 3D, they are great model for ideal fluids, but they have some drawbacks for viscous fluids (viscosity is anisotropic).

Chapter eight concludes theoretical analysis of LGCA by showing what statistical properties does physical fluid exhibit, and that if we artificially impose its statistical properties on the LGCA, we obtain physically realistic model by every means, although this new generation of LGCA are beyond the scope of this thesis.

In chapter ten we introduce probabilistic methods that we intend to use in practical part, to inspect properties of fully-developed turbulence simulated by our models.

Practical part starts with the chapter ten, that summarizes what its content.

Chapter eleven contains general comments on the implementation.

Chapter twelve and thirteen discuss implementation of FCHC and standard algorithm of Pair-Interaction

Chapter fourteen motivates non-deterministic variant of Pair-interaction automata.

In chapter fifteen, we present results of the flow around obstacles simulated on our models.

Chapter sixteen is scientifically most challenging part, and although the obtained results are questionable from various positions, they constitute basis for the further research.

Part I

Theory

1. Cellular automata

Years before DNA and replication mechanism was discovered in living cells, John von Neumann was investigating self-replicating systems in theory, and layed basis for the "New kind of science" [3].

1.1 Game of Life

John Conway, by significant simplification of von Neumann ideas, introduced the Game of Life in 1970 that renewed general interest in cellular automata.

Depending on the initial conditions, evolution of this automaton can be chaotic, periodic or it can lead to the stable configurations.

The reason for this complexity hides in the fundamental property of Game of Life - it is the Turing-complete, so in principle any computer program can be simulated in the Game of Life.

For the purposes of this thesis, we have written a simple program implementing this game, since we can easily express the basic principles of cellular automata on it. (as it easily expresses basic principles of cellular automata.)

Let us have a rectangular grid, with black and white squares. White squares represent dead cells, black cells represent living cells. On Fig.1.1 we see such grid, that we chose for initial state of 'Life'.

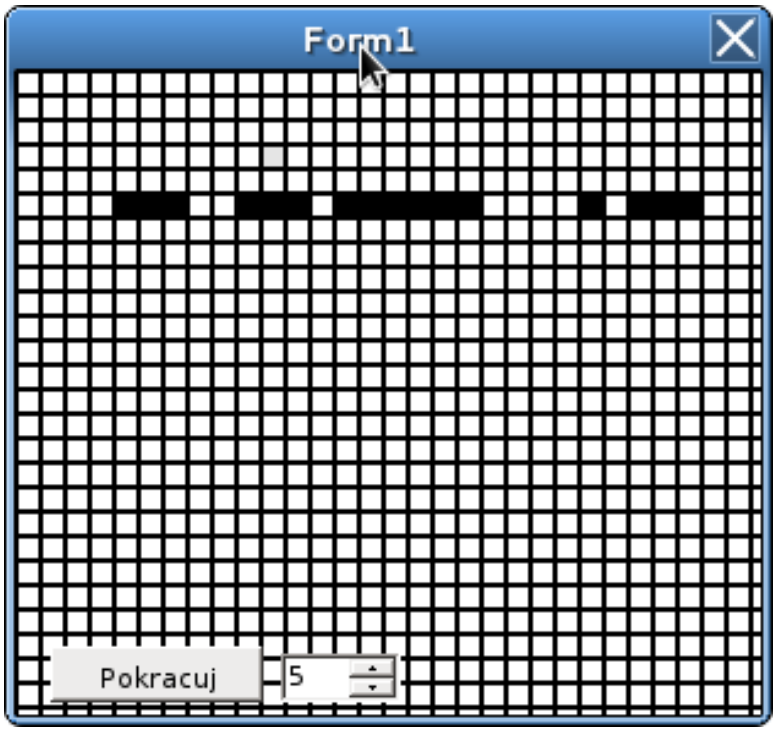


Figure 1.1: The initial state of 'Life' (at t=0)

Now we press 'Pokracuj' button and let the the Life evolve. In the discrete time steps, the grid is changing. We see that some cells are dying, but some cells are getting alive. What is the rule that kills the cell or leave it be?

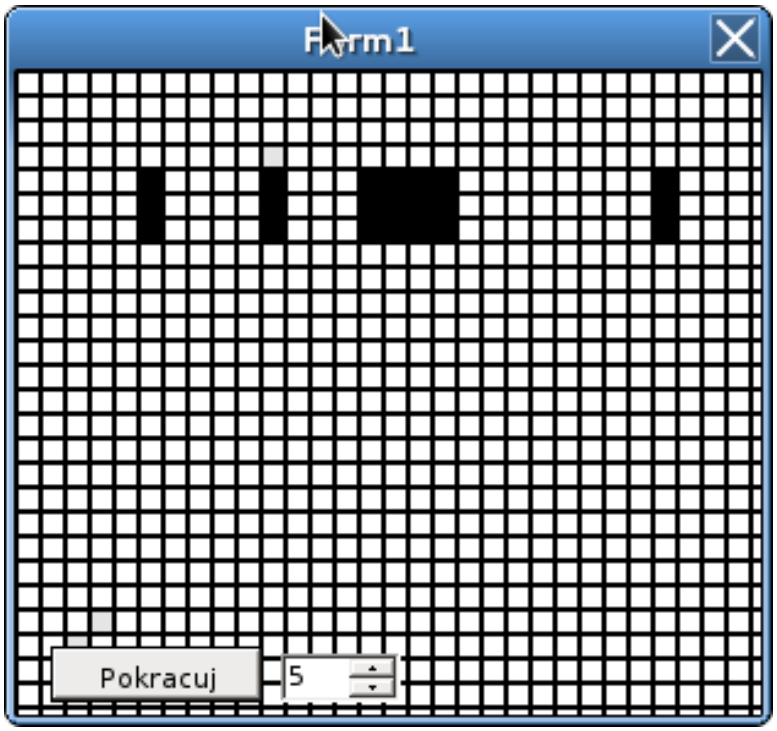


Figure 1.2: t=1

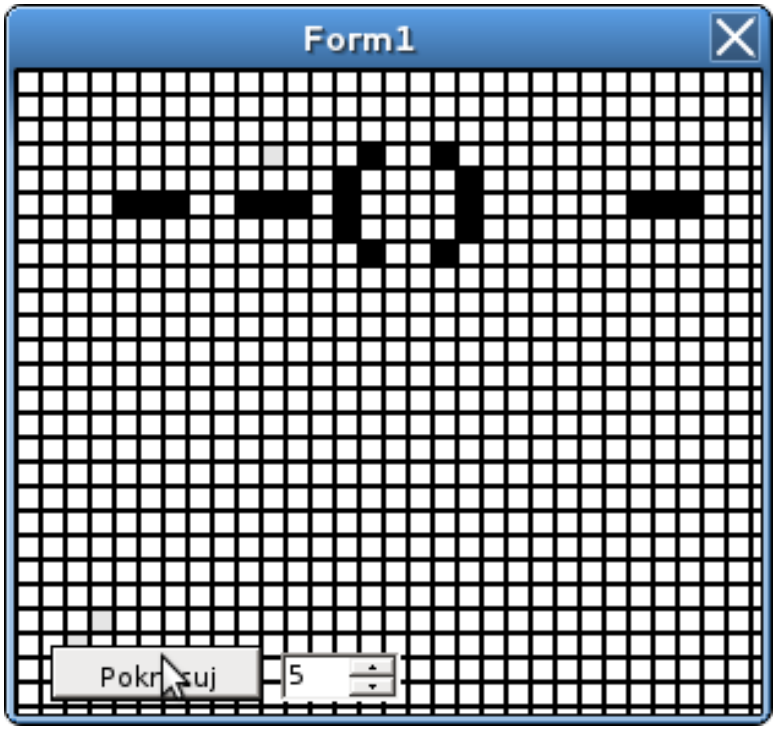


Figure 1.3: t=2

1. If the cell is alive, and 2 or 3 neighbouring cells are alive, the cell will stay alive in the next step. Otherwise it will die.

2. If the cell is dead, and **exactly** 3 neighboring cells are alive, the cell will get alive in the next step. Otherwise, it stays dead.

We see that the rule involves only the state of the certain cell and the states of its eight neighboring cells.

Let us proceed from this simple example to more general setting. In the next chapter, we will generalize main features of 'Life' and formally define the cellular automaton.

1.2 Cellular automaton in general

1. **Position of cells:**

   Instead of 2D rectangular grid of 'Life', cells might be arranged in arbitrary N dimensional regular grid, not necessary rectangular. (Regularity follows from definition of Kubrid. In general, cells might be positioned really wildly, e.g. on Penrose lattice, or arbritrary as proposed by Richard P. Feynman).

2. **Set of cell states Q:**

   In 'Life' cells can be dead or alive (set of states has cardinality 2). In general CA, set of states can be any finite set Q of cardinality K.

3. **Neighborhood:**

   In 'Life', state of the cell in the next step was determined by 8 neighbouring cells. We call these cells neighborhood of range r = 1 (in the distance of 1 cell). For general CA, we might consider neighborhood with arbitrary range. (Neighborhood with r = 2 in 'Life' would involve 9+16=25 cells).

4. **Update rule:**

   Update rule is an arbitrary bounded mapping U from Neighborhood to the set of states Q. Since the state of the Cell is determined only by the state of its Neighborhood, update rules in CA are local.

1.3 The most basic cellular automaton

In middle 1980s on the prestigious Princeton institute, Stephen Wolfram and his assistants were performing unusual computer experiments. They were simulating the evolution of the $1D$ cellular automata and they were analyzing the patterns they obtained [**?**] (to a despair of their senior colleagues, who did not understand this "new kind of science")[3].

The most basic, one dimension cellular automaton we can imagine, is the two state automaton with range $r = 1$.

One dimensional indicates that the cells are arranged in the row, see Figure 1.3

Figure 1.4: A time-step of one dimensional cellular automaton

Range r=1 means that in update rule, we consider only the closest neighbors of the cell (one cell to the left, and one cell to the right).

Example of an update rule is shown the Table 1.1.

a_t^{i-1}	a_t^i	a_t^{i+1}	a_{t+1}^i
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	1
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	0

Table 1.1: Rule 90

The three columns represent the state of a cell (a_t^i) and its left and right neighbor (a_t^{i-1} and a_t^{i+1} respectively). A living cell is denoted by 1, a dead cell is denoted by 0. The last column denotes the state of the middle cell (a_i) in the next step $t+1$. The sequence of *one*s and *zero*s in the last column is the binary *zapis* of a number. In this particular case, the number is 90, hence the table specifies the Rule 90. Since there is $2^8 = 256$ combinations for the last columns, there is 256 rules for this most basic cellular automaton.

Let us take a look at some pictures of these cellular automata (the implementation of this automaton is simple, and these pictures were plotted by our program).

These pictures represent the evolution of cellular automata with rules 90, 30, 45 and 73. The downer-most row is the initial configuration of CA, the 2nd row is configuration after 1st update etc.

First picture 1.5 is famous fractal known as Sierpinski carpet. Wolfram classified these 1D automata into four classes, based on the regularity of the pattern obtained.

Wolfram argues that variety of behavior seen even in these most simple CA

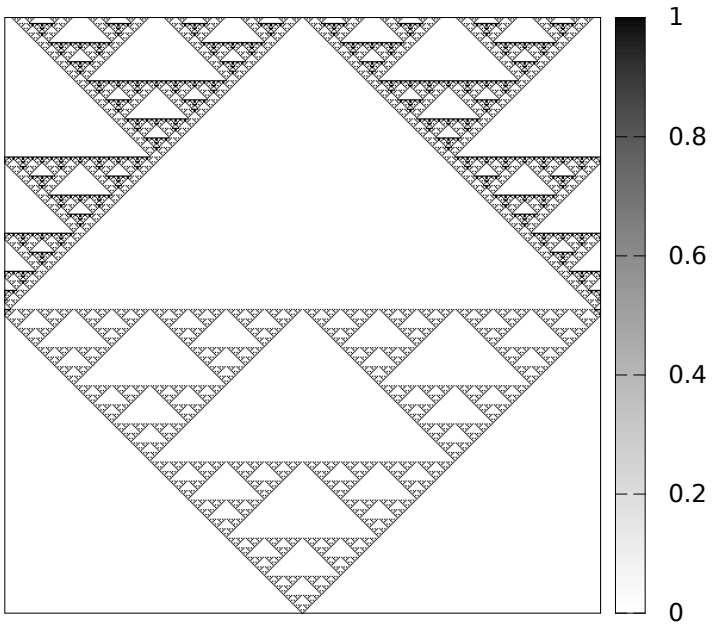


Figure 1.5: Rule 90

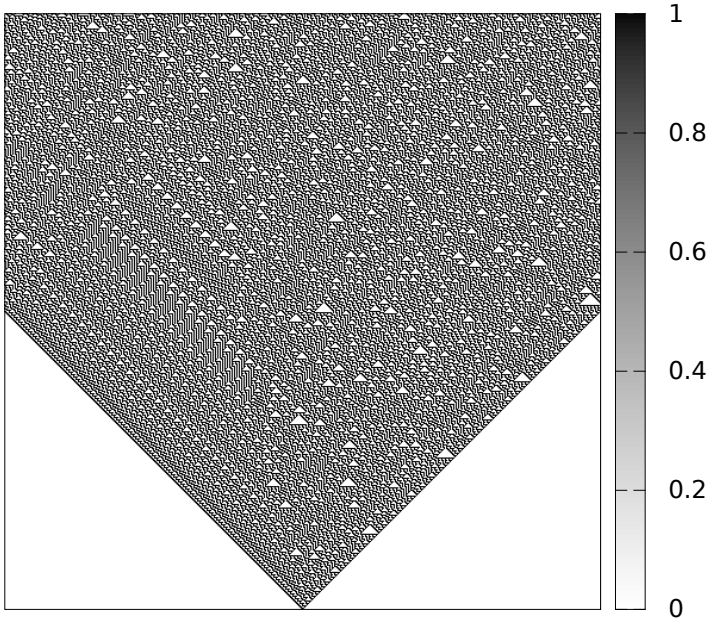


Figure 1.6: Rule 30

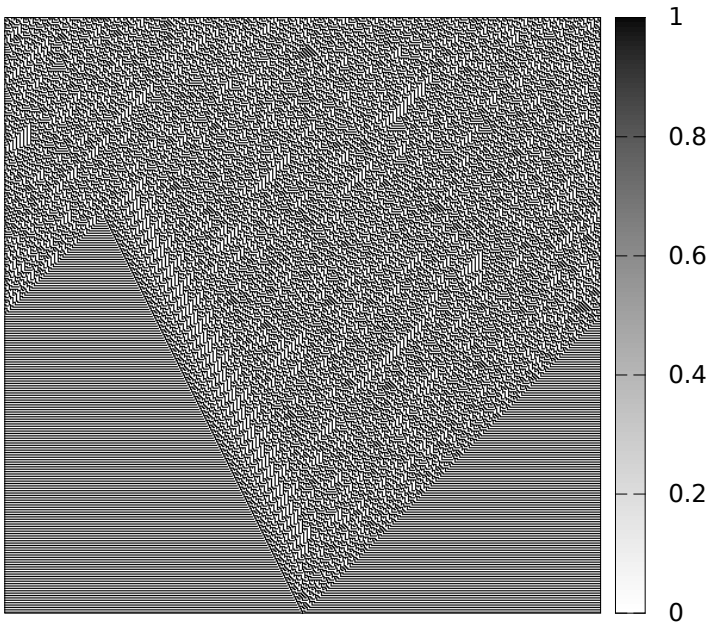


Figure 1.7: Rule 45

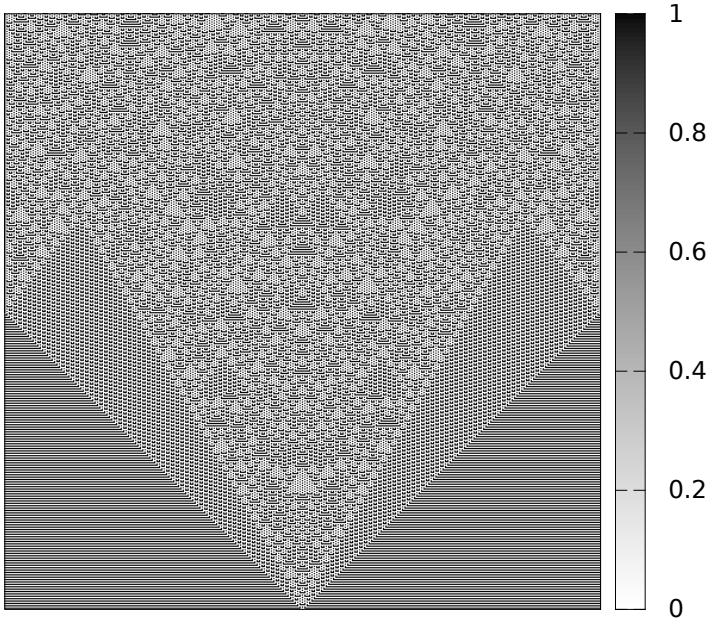


Figure 1.8: Rule 73

rises suspitions, and encourage hope, that in the infinitely large world of possible CAs, there is CA that model any complex phenomena you can imagine.

In his visionary book New kind of Science, Wolfram even proposes Cellular automaton that would constitute a unified theory of the fundamental physics. Although his ideas met with rejection among theorists (see [8]), many other notable physicists are attempting to construct such cellular automaton [9]).

Some of the basic construction principles for such automata are relevant also for our models, but mostly, we would diverge too far by further exploration. Focus of our work is much more modest that CAs describing universe. Our focus is on CAs that model flows of the fluids.

The connection of the cellular automata with flow of fluids is not obvious or formal. It is found on the very general level. What connects Navier-Stokes equations and CAs are their common symettries and conservation laws implied by these symetries. In these symmetries lies not only beauty of this method, but their strongest advantage over other well-established CFD methods.

2. Lattice gass cellular automata

History of Lattice Gass Cellular Automata began in 1973, when HPP model was proposed by Humphry, Pica and Picus.

Unfortunately, it could not do its job properly. For the reasons that we will explore in this chapter, its macroscopic limit did not converge to Navier-Stokes equations.

As we will see later, there came two different approaches how to make functioning LGCA and their common role model is this imperfect HPP.

Hence we will explain basic principles of LGCA on this model, and in subsequent chapters, we will "simply" upgrade it - either to FHP, Pair-interaction or their multi-dimensional variants.

2.1 From CA to LGCA

LGCA share many similarities with calular automata that we already saw, but we need to be aware of the differences.

In general, LGCA is composed of regular clusters of cells - so called nodes. They are arranged in N-dimensional regular lattice.

Lattice of HPP is the simple rectangular 2D grid. At every point of a grid, there is node sitting in, and this node is composed of 4 cells, see Figure 2.1.

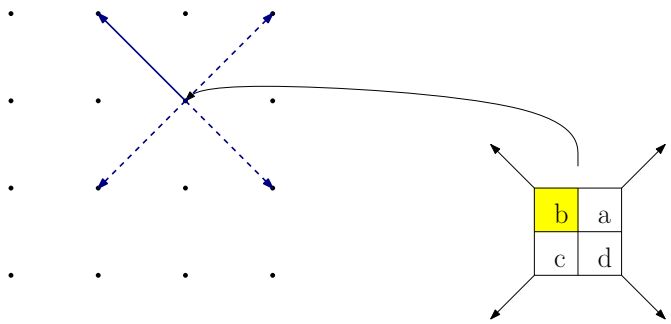


Figure 2.1: Rectangular grid

Each cell of the node can be in two states - empty (white square on the figure 2.1), or occupied by particle (yellow square on figure 2.1). We see that particle in the upper-left square head to the upper-left node.

2.2 Update rule

For any LGCA model, update is done in two subsequent steps, collision and propagation.

In HPP, there are only two collision configuration, and one collision rule that guide them, see Figure 2.3.

We see this configurations are symmetric. The first configuration is resolved to the other and vice versa.

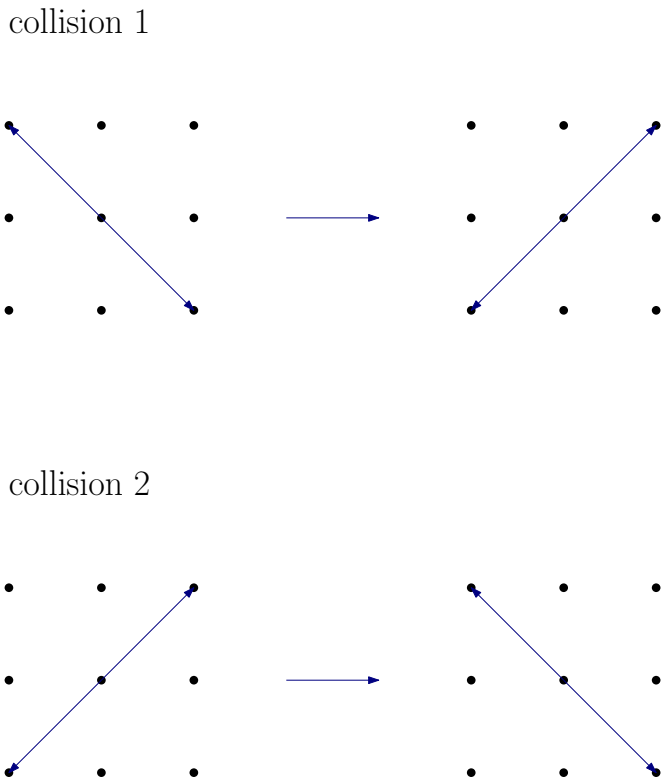


Figure 2.2: HPP colisions

It is easy to understand that there are no other collision configurations and collision rules. If any other state gets changed, it would break the conservation of momentum and would be physically unrealistic.

2.3 Propagation:

After the collision is resolved, propagation follows. During propagation, particle from the upper-left cell moves to the upper-left node, and will occupy they same cell (with the same velocity vector), so that momentum is also conserved by the propagation.

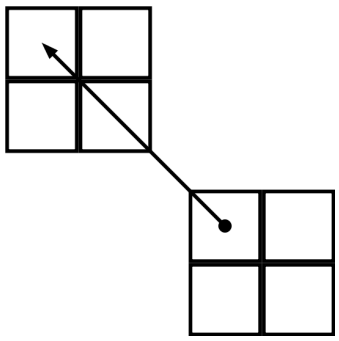


Figure 2.3: Propagation of particle from upper-left cell

Conservation laws:

We already made a note, that collision and propagation conserve momentum, and they obviously conserve mass, since particles are neither created, nor anihilated in these processes. Let us inspect these conservation laws in the more depth by considering symmetries of this model.

Suppose we are using periodic boundary condition. Then, the finite rectangular grid of HPP is actually a torus. It is easy to imagine that if we shift the grid by the discrete step (of length 1), we get the very same torus. So this rectangular grid of HPP is symmetric with respect to translation. As we know by Norther's theorems, translational symmetry implies conservation of momentum.

Also, the grid possesses some rough rotational symmetry - rotating the grid by 90 degrees, we get the same grid.

However, this rotation by 90 degrees is too crude, and so HPP do not preserve the angular momentum. This is the flaw of the model that cannot be overlooked - HPP do not preserve all physical quantities that Navier-Stokes equation would suggest.

Its another flaw goes in the opposite direction. HPP preserves quantities that are non-physical - so called spurious invariant.

Consider orientation of particles in Fig.2.3 before colission. One particle heads to north-east, the other south-west. After the collision, one particle heads to north-west, other particle to south-east. So the number of particles heading to the south, east, north and west do not change by collision (and neither by propagation). It is invariant as the time goes by.

To assert it more scientifistically, let us rozlozit the total momentum into the directions of lattice velocities

$$P = P_N + P_S + P_E + P_W. \tag{2.1}$$

This total momentum P is correctly conserved by HPP, but also quantities

$$P_{spur1} = P_N + P_E - P_S - P_W \tag{2.2}$$

and

$$P_{spur2} = P_N + P_W - P_S - P_E \tag{2.3}$$

are conserved, although these quantities have no physical counterparts.

To conclude this chapter and finish-off the HPP, it is physically implausible because

1. angular momentum is not conserved due to insufficient rotational symmetry,
2. other non-physical quantities, so called *spurious invariants* are conserved.

Although it is flawed model, it sparked interest of many bright minds and various successful LGCAs evolved from HPP. In the next chapter, we will introduce first successful branch of physically relevant LGCAs, the FHP model.

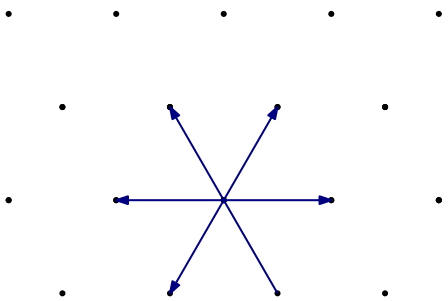
3. FHP

The first upgrade of HPP bears name after its inventors – Frisch, Humpfrey and Pomeu. They published it in 1986 together with its 3D brother FCHC, that we will analyze later, in this chapter we will stick to 2D model.

As any experienced gamer knows, 2D plane can be uniformly covered by hexagons.

If we chose hexagonal grid as the lattice for HPP-like automaton, we get the FHP.

On the Figure 3, we see part of this hexagonal lattice. The dots represent the nodes. On this figure, six lattice vectors points from one of the nodes into the neighbouring nodes.



Six lattice vectors per node imply that each node consists of six cells. Each of these cells can be occupied by one particle, that propagates along the lattice vectors to its neighbors.

3.1 Update rule

As in HPP (and any LGCA that we will study), update consists of two subsequent steps, collision and propagation. Propagation phase is straigt-forward - if the cell is occupied by one particle In the intersection of lines, we have nodes consisting of six cells.

So we have mesh with better (suffiecient!) rotational symmetry and nodes with richer and more interesting set of configurations.

And among these configurations is one that does not conserve number of particles in opposite directions, see Figure 3.1.

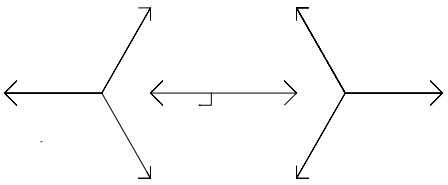


Figure 3.1: Symetric colision of 3 particles

This is the only symetric colison, however. For other colisions - see figures below, we have two states that are offering. If we are chosing randomly from these, we get rid of other spurious invariants. (maybe we should choose more careful

statements, as there exists spurious invariants for any type of FHP, but these invariants are under level of the noise, and they are not hapenning in collisions with obstacles, and regions around obstacles interest us most).

At least in the rough way, we showed how FHP works. In the next chapter, we are going to PROVE FHP really works - prove that FHP converges to Navier-Stokes equations.

4. Microdynamics of FHP

Although we will form In the previous chapters, we seen how FHP and HPP works in intuitive way. In this chapter, we will treat this automata more analytically, hence we need to get more formal.

We will try to make this chapter more general, so that the obtained statistical limit is applicable for all cellular automata from the same class as FHP - specifically its predecessor HPP and its 3D upgrade FCHC.

But since we already have concrete idea of how FHP works, it will be the good point to start.

4.1 FHP analytically

Now that we have good intuition about FHP model, we need to get more formal to treat LGCA analytically.

State of the node will be denoted by $n = (n_1, n_2, n_3, n_4, n_5, n_6)$, where $n_i = 0$ means that i^{th} cell in the node is empty, and $n_i = 1$ means that there is particle in the i^{th} cell.

State of the **specific node** (at the position **r** on the lattice) will be denoted by $\mathbf{n}(\mathbf{r})$, whereas state of **all the nodes** in the lattice will be denoted by $\mathbf{n(.)}$.

We already know that update of the whole lattice is local, hence we can treat it node by node.
We also know that update happens in discrete time steps, and it can be divided into two sub-steps - collision, and propagation.

4.2 Propagation

Propagation is straight-forward for all types of LGCA and can be captured by very simple equation

$$Sn_i(\mathbf{r}) = \mathbf{n_i}(\mathbf{r} + \mathbf{c_i}). \tag{4.1}$$

This equation means that state of the i^{th} cell in node $\mathbf{n(r)}$ propagates along the lattice vector c_i to the neighboring node $\mathbf{n}(\mathbf{r} + \mathbf{c_i})$, see figure below.

To put it in the nutshell, propagation is completely determined by the geometry of the lattice. Once we know what the lattice vectors c_i are, we know all about propagation (equation blabla).

Where the dark magic of LGCAs hides is the collision step.

4.3 Collision

The purpose of collision is to change the configuration of the node as much as possible (it means to change as many bits of the node as possible), while we preserve mass and momentum.
(if we change direction of the particle).

For HPP, we have only $2^4 = 16$ possible states in node, and only two collision configurations among them. Since these configurations are symmetric, we have only one collision rule.
For FHP, we have $2^6 = 64$ configurations, and whole bunch of collision configurations among them, but we can still draw the simple picture of these configurations:

As we see, head-on 2-particle collisions can result in two different states. If we wanted to preserve determinism, and we were systematically choosing only one of them, we would introduce additional, non-physical invariance - the model would become chiral. If we want to preserve the parity symmetry of the model, we need to assign equal probability to either of two final states.Hence, we are introducing non-determinism to the model.

We will express the probabilities of transition from state n to state n' by probability matrix:

$$A(n \to n') \geq 0 \tag{4.2}$$

As we have 64 possible states of the node, matrix A is of dimension 64×64. For example, the cell of matrix A that governs the head-on collisions looks like this:

$$A' = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix}$$

Since the collisions are symmetric, matrix A is symmetric as well.

Also, collisions are invariant to rotations or reflections of the node:

$$A(g(n) \to g(n')) = A(n \to n') \tag{4.3}$$

where $g \in G$, and G is the symmetry group of the node.

4.4 Collision operator

Interestingly, we can express the whole update step in one simple equation using collision operator Δ_i:

$$n_i(t+1, r+c_i) = n_i(t, r) + \Delta_i(t, r) \tag{4.4}$$

If this equation works, then, Δ_i must be:

1. $\Delta_i = 0$ if no collision is happening in $n(t, r)$. Then state of the cell $n_i(t, r)$ only propagates to $n_i(t+1, r+c_i)$.

2. $\Delta_i = 1$ if there is not particle in $n_i(t, r)$ yet, but gets there after collision.

3. $\Delta_i = -1$ if there is particle in the $n_i(t, r)$, but after collision, cell gets empty.

For example, Δ_i acquire very simple form for HPP:

$$\Delta_i = n_{i+1}n_{i+3}(1 - n_i)(1 - n_{i+2}) - n_i n_{i+2}(1 - n_{i+1})(1 - n_{i+3}) \tag{4.5}$$

As there are only 2 collision configuration in HPP, Δ_i has 2 terms. If either collision happens, corresponding term is 1.

4.5 Microscopic conservation laws

Using collision operator, conservation of mass and momentum is this simple:

$$\sum_i \Delta_i(t, r) = 0, \tag{4.6a}$$

$$\sum_i c_i \Delta_i(t, r) = 0 \tag{4.6b}$$

(Prove: unfold Δs) Conservation laws can be equivalently expressed in the form

$$\begin{aligned} \sum_i n_i(t + 1, r + c_i) &= \sum_i n_i(t, r), \\ \sum_i c_i n_i(t + 1, r + c_i) &= \sum_i c_i n_i(t, r). \end{aligned} \tag{4.7}$$

5. Statistical description of FHP

5.1 From microcosmos to macroworld

Although microdynamics of LGCA is physically unrealistic, laws of mass and momentum conservation are fulfilled in the nodes, as we showed by the end of last chapter.

By employing aparatus of statistical mechanics we will show that these microscopic conservation laws lead to physically realistic macroscopic description.

5.2 Liouville's theorem a.k.a. conservation of probabilities

Let us define the phase space Γ as the set of all possible states of the lattice $n(.)$. Imagine we want to initialize cellular automaton with some macroscopic velocity $\boldsymbol{v_0}$, macroscopic pressure p_0, and macroscopic density ρ_0. We can realize this macrostate by very many microstates of lattice $n(.)$. We assign initial probability to each of these microstates:

$$P(0, s(.)) \geq 0. \tag{5.1}$$

Of course, probabilities over whole lattice are normalized so $\sum_s(.)P(0, s(.)) = 1$.

In the statistical mechanics, Liouville's space state theorem postulates that the density of the phase space is constant. Microdynamics of our model implies equivalent theorem for LGCA.

$$P(t+1, \mathcal{E}s(.)) = P(t, s(.)) \tag{5.2}$$

It is obtained directly by applying the update formula

$$\mathcal{E}s(t, .) = s(t+1, .) \tag{5.3}$$

However, for indeterministic model such as FHP, the conservation of probability is governed by the more general formula

$$P(t+1, \mathcal{S}n'(.)) = \sum_{n(.)\in\Gamma} \prod_{n(.)} A(n(\boldsymbol{r} \to n'(\boldsymbol{r}))P(t, n(.)), \tag{5.4}$$

that constitutes the LGCA version of Chapman-Kolmogorov equation.

5.3 Mean occupation numbers

Motivated by the ensemble formalism of statistical physics, we define mean occupation numbers

$$N_i = \langle n_i \rangle = \sum_{s(.)\in\Gamma} n(s(.))P(t, s(.)).n \tag{5.5}$$

This formula naturally suggests definition of the mean mass density

$$\rho(t,r) = \sum_i N_i(t,r) \tag{5.6}$$

and the momentum density

$$j(t,r) = \sum_i c_i N_i \tag{5.7}$$

Due to conservation of probabilities, the conservation laws 4.7 implies conservation of the mean quantities

$$\sum_i N_i(t+1, r+c_i) = \sum_i N_i(t,r) \tag{5.8}$$

$$\sum_i c_i N_i(t+1, r+c_i) = \sum_i c_i N_i(t,r) \tag{5.9}$$

5.4 Equilibrium occupation number

After laborious definitions in previous sections, we are ready for one of the central theorems of FHP model, stating that the equilibrium occupation numbers are given by Fermi-Dirac distribution. Its implications will haunt us until the last chapter.

We state it without the lengthy proof, but we recommend [1] or [4] for the non-believers.

Since the formalism that we were using in previous section is general, independent of the dimension of FHP, we could state the last theorem for the FHP-like automaton in arbitrary dimension D with b lattice vectors $c_i \in R^D,\ i = 1...b$.

Theorem 1: The following statements are equivalent:

1. N_i^{eq}s are solutions of Chapman-Kolmogorov equation (reference)

2. N_i^{eq}s are solutions of set of b equations:

$$\Delta_i(N) = \sum_{nn'} (n_i' - n_i) A(n \to n') \prod_j N_j^{n_j} (1 - N_j)^{1-n_j} \tag{5.10}$$

3. N_i are given by Fermi-Dirac distribution where h is real number and **q** is D-dimensional vector.

To express N_i^{eq} explicitly as function of ρ and $\boldsymbol{u}$, we employ technique of Lagrange multipliers with natural constraints

$$\rho = \sum_i N_i = \sum_i \frac{1}{1 + exp(h + \boldsymbol{q}.\boldsymbol{c_i})} \tag{5.11}$$

$$\boldsymbol{u} = \sum_i \boldsymbol{c_i} \boldsymbol{N_i} = \sum_i \frac{\boldsymbol{c_i}}{1 + exp(h + \boldsymbol{q}.\boldsymbol{c_i})} \tag{5.12}$$

Explicit solutions are available only in few special cases. In general, we may use expansion for small Mach numbers ($\boldsymbol{u}/c_{\text{sound}}$). By expansion up to second order, equilibrium distribution for D-dimensional FHP-like automaton reads [1]

$$N_i^{eq}(\rho, \boldsymbol{u}) = \frac{\rho}{b} + \frac{D\rho}{c^2 b} \boldsymbol{c_i}.\boldsymbol{u} = \rho G(\rho) Q_{i\alpha\beta} u_\alpha u_\beta + O(u^3) \tag{5.13}$$

where

$$Q_{i\alpha\beta} = c_{i\alpha} c_{i\beta} - \frac{c^2}{D} \delta_{\alpha\beta} \tag{5.14}$$

and

$$G(\rho) = \frac{D^2}{2c^4 b} \frac{b - 2\rho}{b - \rho}. \tag{5.15}$$

5.5 Chapman-Enskog expansion

Because relaxation towards equilibrium values happens in few updates of the automaton, it is standard procedure to expand occupation numbers $N_i(t, r)$ around equilibrium occupation numbers $N_i^{eq}(\rho, \boldsymbol{u})$:

$$N_i(t, r) = N_i^0(t, r) + \epsilon N_i^1(t, r) + \mathcal{O}(\epsilon^2) \tag{5.16}$$

The equations of mass and momentum conservation 5.8 and 5.9 can be equivalently stated in the form

$$\sum_i N_i(t + 1, r + c_i) - N_i(t, r) = 0, \tag{5.17}$$

$$\sum_i c_i (N_i(t + 1, r + c_i) - N_i(t, r)) = 0, \tag{5.18}$$

that is more suitable for our purpose.

Expansion of $N_i(t + 1, r + c_i)$ around $N_i(t, r)$ leads to

$$\begin{aligned} N_i(t + 1, r + c_i) = N_i(t, r) + \partial_t N_i(t, r) + c_{i\alpha} \partial_\alpha N_i(t, r) \\ + \frac{1}{2} \partial_t \partial_t N_i(t, r) + \frac{1}{2} c_{i\alpha} c_{i\beta} \partial_\alpha \partial_\beta N_i(t, r) + c_{i\alpha} \partial_t \partial_\alpha N_i(t, r) + \mathcal{O}(\partial^3). \end{aligned} \tag{5.19}$$

The expression above is the mixture of various physical phenomena – diffusion, advection, propagation of sound waves or relaxation towards local equilibria. Each phenomena has its typical spatial and temporal scale, that corresponds to different powers of ϵ, see the Table 5.5.

[1] The expansion is required up to the second order, because the non-linear term in Navier-Stokes equations will emerge from the quadratic term

TEMPORAL SCALES		
Scale	**Rescaling of time**	**Phenomena**
1 step	t	Relaxation towards local equilibrium
100 steps	$t_1 = \frac{1}{100}t = \epsilon t$	advection, sound waves (perturbation of mass and densi
10 000 steps	$t_2 = \frac{1}{10000}t = \epsilon^2 t$	diffusion

SPATIAL SCALES		
Scale	**Rescaling of length**	**Phenomena**
1 lattice unit	$\mathbf{r}$	Relaxation towards local equilibrium
100 lattice units	$\boldsymbol{r_1} = \frac{1}{100}\boldsymbol{r} = \epsilon\boldsymbol{r}$	diffusion, advection, sound waves

To derive the hydrodynamical equation that we long for, we will exploit the multi-scale technique. It starts by grouping-up the terms of hydrodynamical temporal and spatial scales.

Using to rescaled the length and time, we deduce the rescaled differential operators

$$\begin{aligned} \partial_t &= \epsilon\partial_t^{(1)} + \epsilon^2\partial_t^{(2)} \\ \partial_\alpha &= \epsilon\partial_\alpha^{(1)} \end{aligned} \tag{5.20}$$

Now, we are ready to expand the conservation laws 5.17 and 5.18 by Chapman-Enskog. We insert expansion 5.19 of $N_i(t+1, r+c_i)$, then we insert expansion of $N_i(t, r)$ according to 5.16. Finally, we substitute differential operators according to 5.20.

We do not present the whole monstrous expansion, only the terms of order ϵ, that we are interested in.

From conservation of mass 5.17 we get

$$\partial_t^{(1)} \sum_i N_i^{(0)} + \partial_\beta \sum_i c_{i\beta} N_i^{(0)} = 0, \tag{5.21}$$

and from conservation of momentum 5.18 we get

$$\partial_t^{(1)} \sum_i c_{i\alpha} N_i^{(0)} + \partial_\beta \sum_i c_{i\alpha} c_{i\beta} N_i^{(0)} = 0. \tag{5.22}$$

Using definition of mass density, and the following definition of the *momentum flux tensor*

$$P_{\alpha\beta}^{(0)} := \sum_i c_{i\alpha} c_{i\beta} N_i^{(0)} = \sum_i c_{i\alpha} c_{i\beta} N_i^{eq}(\rho, \boldsymbol{u}). \tag{5.23}$$

we can write the conservation laws in the shorter form

$$\begin{aligned} \partial_t^{(1)}\rho + \nabla^{(1)}(\rho\boldsymbol{u}) &= 0, \\ \partial_t^{(1)}(\rho u_\alpha) + \nabla^{(1)} P_{\alpha\beta}^{(0)} &= 0 \end{aligned} \tag{5.24}$$

In the first equation, we recognize the famous continuity equation, but we have some work to do with the second equation.

Substituting 5.13 for N_i^{eq} into momentum flux tensor, its components read (after some algebraic simplification)

$$\begin{aligned} P_{xx}^{(0)} &= \frac{\rho}{2} g(\rho)(u_x^2 - u_y^2) + \frac{\rho}{2}, \\ P_{yy}^{(0)} &= \frac{\rho}{2} g(\rho)(u_x^2 - u_y^2) + \frac{\rho}{2}, \\ P_{xy}^{(0)} &= P_{yx}^{(0)} = \rho g(\rho) u_x u_y, \end{aligned} \tag{5.25}$$

where we defined $g(\rho) = \frac{3-\rho}{6-\rho}$.

Unfortunately, it does not match the momentum flux tensor of Navier-Stokes equation

$$\begin{aligned} P_{xx} &= \rho\, u_x^2 + p \\ P_{yy} &= \rho\, u_y^2 + p \\ P_{xy} &= P_{yx} = \rho u_x u_y \end{aligned} \tag{5.26}$$

Let us examine why are these tensors different.

For low values of $\boldsymbol{u}^2$, pressure p is given by isothermal relation [**?**] $p = \frac{\rho}{2} = \rho c_s^2$, where $c_s = \frac{1}{\sqrt{2}}$ is the speed of sound.

What about $g(\rho)$?

The disease of FHP, FCHC and all lattice-gas cellular automata to follow is, that $g(\rho)$ is never equal to 1, as it is in Navier-Stokes equations.

The reason lies in the broken *Galilei invariance* of LGCA,as they all have discrete rotational symmetry (by 60° in FHP and between 60° and 45° in FCHC).

In the final chapter on theory of LGCA, we will show the fundamental treatment of this flaw. Symptomatic treatment, such as rescaling the time

$$t \to \frac{t}{g(\rho)}, \tag{5.27}$$

does not solve all the associated problems (D'Humier et al. 1987).

However, using this rescaled time, and setting density to be constant ($\rho = \rho_0$ except for the pressure term 5.29), we get the familiar incompressible Euler equation

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u}\boldsymbol{\nabla})\boldsymbol{u} = -\boldsymbol{\nabla} P, \tag{5.28}$$

where we used

$$P = (\frac{\rho}{2\rho_0 g(\rho_0)} - \boldsymbol{u}^2). \tag{5.29}$$

.

This is as far as we can get in the first approximation. The derivation of the Navier-Stokes equations

$$\begin{aligned} \boldsymbol{\nabla}.\boldsymbol{u} &= 0, \\ \partial_t \boldsymbol{u} + (\boldsymbol{u}\boldsymbol{\nabla})\boldsymbol{u} &= -\boldsymbol{\nabla} P + \nu\, \nabla^2 \boldsymbol{u} \end{aligned} \tag{5.30}$$

would require terms of the order ϵ^2 from the Chapman-Enskog expansion to add in the equations 5.17 and 5.18, as done by Frisch [4].

6. FCHC

In two dimensions, we are lucky to find hexagon – a symmetric polygon that covers whole 2D plane and possesses sufficient rotational symmetry to make a good lattice for FHP.

In three dimensions, we have five symmetric candidates for LGCA but none of them works. They are so-called Plato solids. Out of them, dodecahedron or icosahedron might have sufficient rotational symmetry (interestingly, they have the same group of symmetries) but they do not cover whole 3D space uniformly. Only the cube does but LGCA build on a cubic grid would suffer from the same problem as HPP – insufficient rotational symmetry and insufficient degrees of freedom in the nodes.

Short analysis would show (see [1] for example) that, in 5 and more dimensions, we have only three symmetric solids – simplex, hypercube and its dual solid. None of them fits.

Fortunately, in four dimensions, we have three extra symmetric solids and one of them actually works.

6.1 Face-centered hypercube

Face-centered hypercube (we will use the shortcut FCHC) is the suitable solid for LGCA in four dimensions. It can be defined by its 24 vertices with the Cartesian coordinates

$$\begin{aligned}
&(\pm 1, \pm 1, 0, 0),\\
&(\pm 1, 0, \pm 1, 0),\\
&(\pm 1, 0, 0, \pm 1),\\
&(0, \pm 1, \pm 1, 0),\\
&(0, \pm 1, 0, \pm 1),\\
&(0, 0, \pm 1, \pm 1)
\end{aligned} \tag{6.1}$$

These 24 vertices correspond to 24 lattice vectors in four dimensions but if we project them into three dimensions (by "deleting" the fourth coordinate q_4) we get only 18 lattice vectors.

For $q_4 = 0$ we have 12 lattice vectors (dotted lines in Fig. 6.1), each vector corresponds to the single cell only. Hence, at most one particle propagates along each of these vectors.

For both $q_4 = -1$ and $q_4 = 1$ we get the same set of six lattice vectors. Each vector corresponds to the two cells, hence two particles can propagate along each vector.

It is possible and efficient to denote each node of the lattice by 3 Cartesian integer coordinates.

In the resulting lattice, each node is connected to 18 neighbors by the lattice vectors of Fig. 6.1, and 24 particles can propagate along them in the single step.

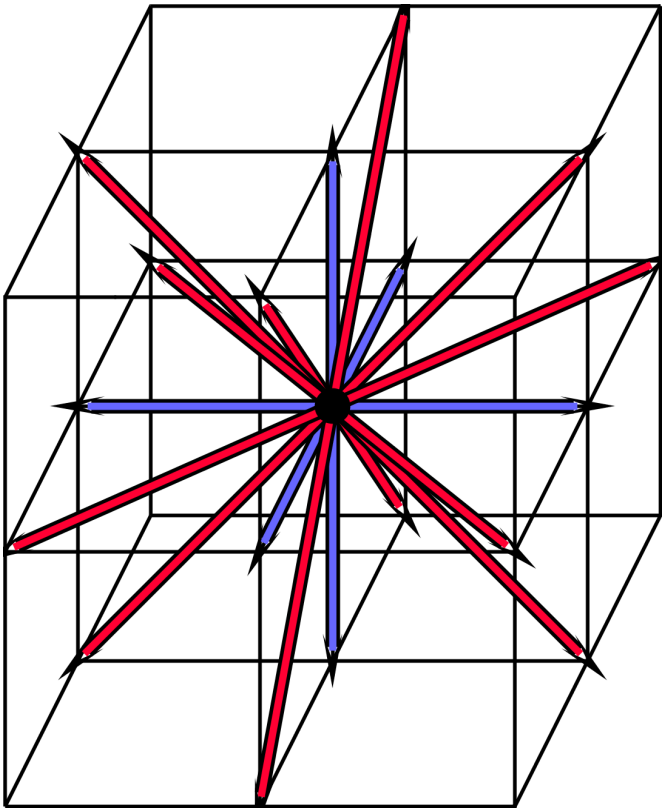


Figure 6.1: Projection of the lattice vectors into 3D. Red arrows are projections of vectors with $q_4 = 0$, blue arrows with $q_4 = \pm 1$.

6.2 Collision rules for FCHC

The FCHC lattice that we just described was proposed already by Frisch et. al. in 1986. However, they did not propose any recipe for a collision. It is easy to understand why it was not a simple task.

There are 24 cells in each node, hence the node can acquire $2^{24} = 16\ 777\ 216$ different states (that can be represented by 24 bits). It was easy to specify the collision rule for FHP in one simple table, but how do you specify rules for millions of states?

In the subsequent section we will show you how Henon answered to this challenge in his famous article [5].

6.2.1 Necessary conditions

Let us review what are the necessary conditions that collision rules must fulfill:

1. number of particles is preserved;
2. total momentum is preserved;
3. no other quantity is preserved;
4. exclusion principle – no two particles occupy one cell in the new state;

5. collision rules are invariant with respect to rotations and reflections of the node;

6. collisions satisfy the semi-detailed balance.

Henon proposed his own conditions on the collisions and showed that all conditions above are automatically fulfilled.

1. All collisions are isometries of FCHC – such transformations that preserve set of 24 lattice vectors above. It can be shown that this condition actually defines FHP collision set. For FCHC, this condition guarantees that conditions will be simple and leads us to the simple collision algorithm.

2. The choice of isometry is restricted by the momentum conservation only. There are 7009 possible values of momentum. But considering that collision rules are invariant under isometries of FCHC, and due to first restriction that collisions are isometries, we can consider 37 momenta only.

3. We want shear viscosity to be as low as possible, so that we can go to higher Reynolds numbers in our simulations. Suppressing the viscosity means is achieved by shortening the mean free path of the particles. In other words, we wish to change as many bits in the collision as possible. We will call such collisions "optimal isometries".

4. The isometry is randomly chosen among optimal isometries.

Having discussed the role of the isometries, let us define them properly.

6.3 Isometries of FCHC

By G we denote the group of isometries that preserve all 24 lattice vectors (or equivalently all vertexes of FCHC). Any isometry can be represented by 4×4 matrix

$$M = \begin{bmatrix} a_{11} & \dots & a_{14} \\ \dots & & \dots \\ a_{41} & \dots & a_{44} \end{bmatrix}. \tag{6.2}$$

Although the group G is of order 1152, it can be generated by five elements only, but it will be more comfortable to employ the generating set consisting of 12 elements to be described in the following section.

6.3.1 Generating set

Isometries $S_\alpha,\ \alpha = 1, 2, 3, 4$, are reflections over(?) plane $x_\alpha = 0$. For example, S_1 is represented by the matrix

$$S_1 = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.3}$$

that flips the sign of the first momentum coordinate q_1.

Another six isometries $P_{\alpha\beta}$ are reflections over plane $x_\alpha = x_\beta$. For example, P_{12} can be represented by matrix

$$P_{12} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{6.4}$$

that swaps the first and second coordinates of the momentum.

Another isometry Σ_1 is reflection over plane $x_1 + x_4 = x_2 + x_3$ represented by matrix

$$\Sigma_1 = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & -1 \\ 1 & 1 & -1 & 1 \\ 1 & -1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix}. \tag{6.5}$$

The last generating isometry Σ_2 is reflection over plane $x_1 = x_2 + x_3 + x_4$. In matrix representation it reads

$$\Sigma_2 = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \tag{6.6}$$

Any isometry of FCHC can be composed from these 12 isometries, so it can be uniquely expressed by

$$M = \begin{pmatrix} I \\ S_4 \end{pmatrix} \begin{pmatrix} I \\ S_3 \end{pmatrix} \begin{pmatrix} I \\ S_2 \end{pmatrix} \begin{pmatrix} I \\ S_1 \end{pmatrix} \begin{pmatrix} I \\ P_{34} \end{pmatrix} \begin{pmatrix} I \\ P_{23} \\ P_{24} \end{pmatrix} \begin{pmatrix} I \\ P_{12} \\ P_{13} \\ P_{14} \end{pmatrix} \begin{pmatrix} I \\ \Sigma_1 \\ \Sigma_2 \end{pmatrix} \tag{6.7}$$

where the brackets mean "one of the isometries inside the bracket".

6.3.2 Normalized momenta

Normalized momenta are such momenta that their components fulfill the following two conditions:

1. $q_1 \geq q_2 \geq q_3 \geq q_4 \geq 0$,
2. $q_0 = 0$ or $q_1 + g_4 = q_2 + q_3$.

Henon specified collision rules for normalized momenta only. There is 37 normalized momenta (out of 7009 total) so he significantly simplified the job.

If the state has normalized momentum (i.e. fulfilling 1), we just look into table and pick any isometry that is offered for that momentum. However, if the state doesn't have normalized momentum,

Class	w_{min}	Optimal isometries	
1	1/2	1	P_{12}
2	1/4	2	$P_{23}P_{12}$, $P_{23}P_{13}$
3	1/4	2	$S_4\Sigma_1$, $S_4\Sigma_2$
4	1/2	1	S_4
5	1/3	1	S_4P_{12}
6	1/4	4	$S_4\Sigma_1$, $S_4\Sigma_2$, $S_4P_{23}\Sigma_1$, $S_4P_{23}\Sigma_2$
7	1/3	1	S_4P_{23}
8	1/4	4	$P_{23}P_{12}$, $P_{23}P_{13}$, $S_4P_{23}P_{12}$, $S_4P_{23}P_{13}$
9	1/3	3	S_4S_3, S_3P_{34}, S_4P_{34}
10	1/6	6	$S_3P_{34}P_{12}$, $S_4P_{34}P_{12}$, $S_4S_3\Sigma_1$, $S_4S_3P_{34}P_{12}\Sigma_1$, $S_4S_3\Sigma_2$, $P_{34}P_{12}\Sigma_2$
11	1/6	6	$S_4S_2P_{23}$, $S_4S_3P_{23}$, $S_3S_2P_{24}$, $S_4S_3P_{24}$, $S_3S_2P_{34}$, $S_4S_2P_{34}$
12	0	12	$S_3S_1P_{34}P_{12}$, $S_4S_1P_{34}P_{12}$, $S_3S_2P_{34}P_{12}$, $S_4S_2P_{34}P_{12}$, $S_2S_1P_{24}P_{13}$, $S_4S_1P_{24}P_{13}$, $S_3S_2P_{24}P_{13}$, $S_4S_3P_{24}P_{13}$, $S_2S_1P_{23}P_{14}$, $S_3S_1P_{23}P_{14}$, $S_4S_2P_{23}P_{14}$, $S_4S_3P_{23}P_{14}$

Figure 6.2: Optimal isometries for 12 momenta classes

1. we normalize it by applying an appropriate isometry on the state. Let us denote this isometry by Γ. In fact, this isometry is equivalent to the change of the coordinate system.

2. Now that we have state with normalized momenta, search in the table and pick an optimal isometry.

3. We go back to the previous coordinate system by applying isometry Γ^{-1}.

6.3.3 Optimal isometries

However, in the step 2, we do not want to use all the isometries that conserve the momentum. Optimally, we would like to change as many bits as possible (that minimizes mean free path of particles, and that leads to desired low viscosity).

Although no further reduction then of 37 normalized momenta is possible, some of these momenta share same optimal isometries, that lead to minimal viscosity and preserve momentum.

Based on that, momenta can be divided into 12 classes, see Table. First column specifies momentum, second column specifies isometries that we can apply to this momentum.

Now it is possible, although not easy, to resolve collisions in FCHC.

7. Pair Interaction LGCA

Pair-interaction automata (PI) constitute another branch that successfully evolved from HPP model, somewhat in contrast to FHP. To put it in a nutshell, FHP changed the rectangular grid for hexagonal, and thus obtained better rotational symmetry, increased the degree of freedom of the nodes and added new collision states.

The pair-interaction model preserves the HPP rectangular lattice, and the degree of freedom in the node is incremented by the artificial definition of momentum. In FHP and HPP, momentum of the particle corresponds to its lattice velocity, or equivalently, to the cell that the particle occupies. In PI, the momentum is the independent of the lattice velocity, it is the inner degree of the particle, that might change in the collision. Although the differentiation in momentum and velocity might seem odd at the first glance, we will show that it leads to the correct microdynamics and offers us very efficient algorithm for implementation.

We will explore theory of Pair Interaction automaton in arbitrary dimension later on, but to get good understanding of PI algorithm, we start with its 2D version, so we can explain the theory in graphically.

We acknowledge that some of the pictures we are using are copy-pasted from the [1].

7.1 User-friendly guide to 2D PI

7.1.1 Pair-interaction

Geometry of the lattice is same as for HPP model. It consists of nodes arranged in rectangular grid:

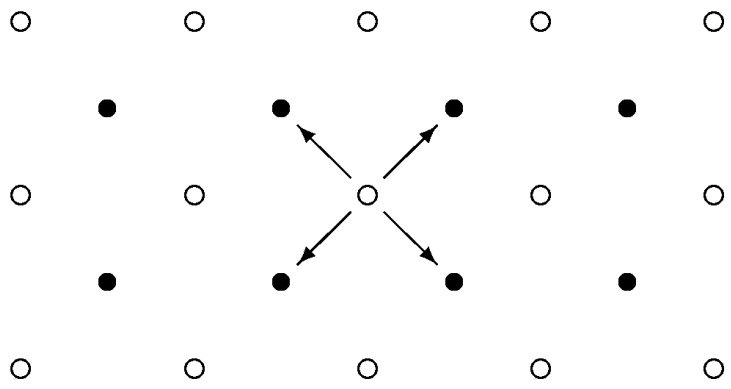


Figure 7.1: Lattice of 2D Pair Interaction automaton

Every node (the cirlce on the picture above) is connected to 4 diagonal neighbors.

If we look on the node in more detail, it looks like this:

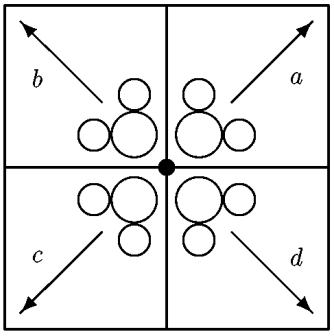


Figure 7.2: A node in detail

It consists of 4 cells, **a,b,c,d** on the picture. Each cell is represented by 3 bits: one mass bit (a big circle) and two momentum bits (small circles).

The node on the picture above is empty (all bits are set to zero).

Here we have another example of a node:

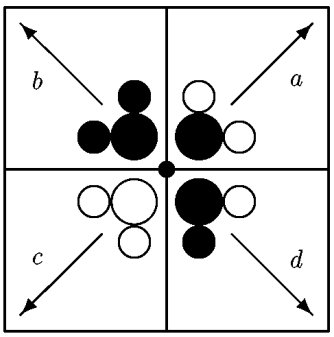


Figure 7.3: State of a node before collision

In this node, there are particles in the cells a,b,d (mass bit is set to 1 - the big circle is black).

Particle in the cell **a** is standing (both momentum bits are 0).

Particle in the cell **b** has momentum in direction [1,1] and particle in the cell **d** has momentum in direction [0,1].

7.1.2 Update rules

First two rules are true of every cellular automaton, the third rule is specific for lattice-gas cellular automata.

1. Lattice is changing in discrete time steps
2. Update rules are **local**. It means that state of the cell in the next step is determined by the current state of the cell itself and its four neighbors. Hence we can resolve update of the lattice node by node (suitable for parallel computing)
3. If this cellular automaton really simulates fluid dynamics and is physically realistic, update rule needs to conserve **mass, momentum and angular momentum**.

Update of the lattice is done in two steps, collision and propagation.

7.1.3 Collision

Collision changes configuration inside the single nodes. We require that momentum and mass is preserved in this step.

1) First, we make pairs of the cells in X direction:

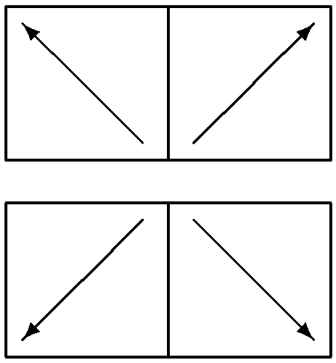


Figure 7.4: Pairs in X-direction

Then we change the configuration in these pairs (preserving total momentum and mass in the node). Therefore, node in 7.1.1 changes to this:

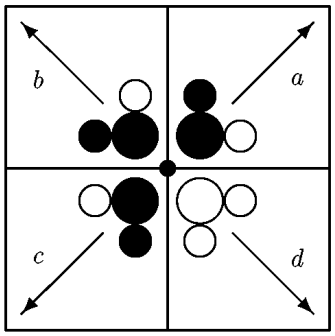


Figure 7.5: State of the node after pair-interaction in X direction

2) Second, we make pairs of the cells in Y direction:

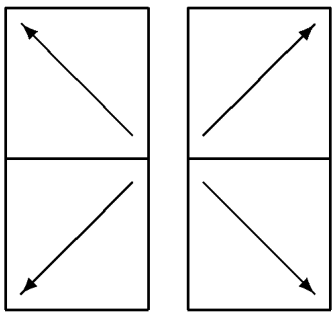


Figure 7.6: Pairs in Y-direction

Then we change the configuration in these pairs preserving mass and momentum in the node. Hence, the node in 7.1.3 changes into:

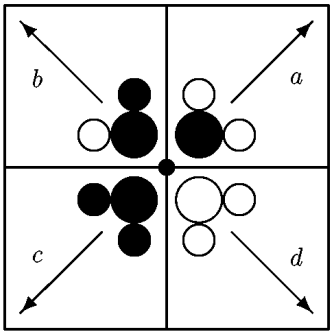


Figure 7.7: State of the node after pair-interaction in Y-direction

Actually, there is only 17 different configurations of the pairs. So instead of computing new configuration during every pair-interaction, we can just look in the table 15.2 below and change the configuration accordingly. We consider this to be the greatest practical advantage of PI comparing to other methods.

7.2 Propagation:

Particles propagates from a single node to the four neighboring nodes along the lattice vectors, see 7.1.1.

Particle from the cell **a** propagates to the node up-right, to the cell **a** as well.

Analogously, particle from cell **b** propagates to the node up-left, to the cell **b**.

Of course, particle carries all its bits (its momentum).

Generalization of this automaton to arbitrary dimension D is very straightforward. But we need to establish some mathematical formalism, as we haven't learned to draw D-dimensional pictures.

7.3 3D Pair-interaction cellular automaton

DIFFERENCES BETWEEN **2**D AND **N**D PAIR INTERACTION		
	2dim PI CA	**Ndim PI CA**
nodes	4 cells arranged into square	2^D cells arranged in hypercube
cells	1 mass bit, 2 momentum bits	1 mass bit, N momentum bits
neighbors	node has 4 neighbors	node has 2^D neighbors

In his famous paper, Nasilowski defined and used formalism that:

1. is more difficult then is necessary in our opinion (and in few cases he redefines it along the way)
2. therefore his arguments are more difficult then necessary

Instead, we will stick to the more modern formalism that we have been using. We consider it more brief and effective and reader should be already familiar with it.

In previous chapter, reader should get good understanding how 2D Pair-interaction works. believe that reader has already good idea how 2D Pair-interaction automaton works.

State of the node will be denoted by $\boldsymbol{n}(t, \boldsymbol{r})$, where $\boldsymbol{r}$ is its position on the lattice and t is the current time step. $\boldsymbol{n}(t, .)$ represents the whole lattice at time t, sometimes we can the time variable and use only $\boldsymbol{n}(.)$.

Each node consists of 2^D cells:

$$\boldsymbol{n} = (n_1, n_2, ..., n_{2^D}) \tag{7.1}$$

State of each node is given by $2^D(D+1)$ bits:

$$\boldsymbol{n} = \left\{ n_{i\alpha} \in \left\{0, 1\right\},\ i = 1..2^D,\ \alpha = 0, 1, ...D \right\} \tag{7.2}$$

where index i specifies cell of the node, and index α specifies the momentum bit of the cell. Actually, they are the Cartesian coordinates of the cell momenta.

Evolution of the lattice happens in discrete time steps - updates $\mathcal{U}$:

$$\mathcal{U}\boldsymbol{n}(t, .) = \boldsymbol{n}(t+1, .) \tag{7.3}$$

Update operator $\mathcal{U}$ can be further divided in two operations, collision $\mathcal{C}$ and propagation $\mathcal{P}$:

$$\mathcal{U} = \mathcal{P} \circ \mathcal{C} \tag{7.4}$$

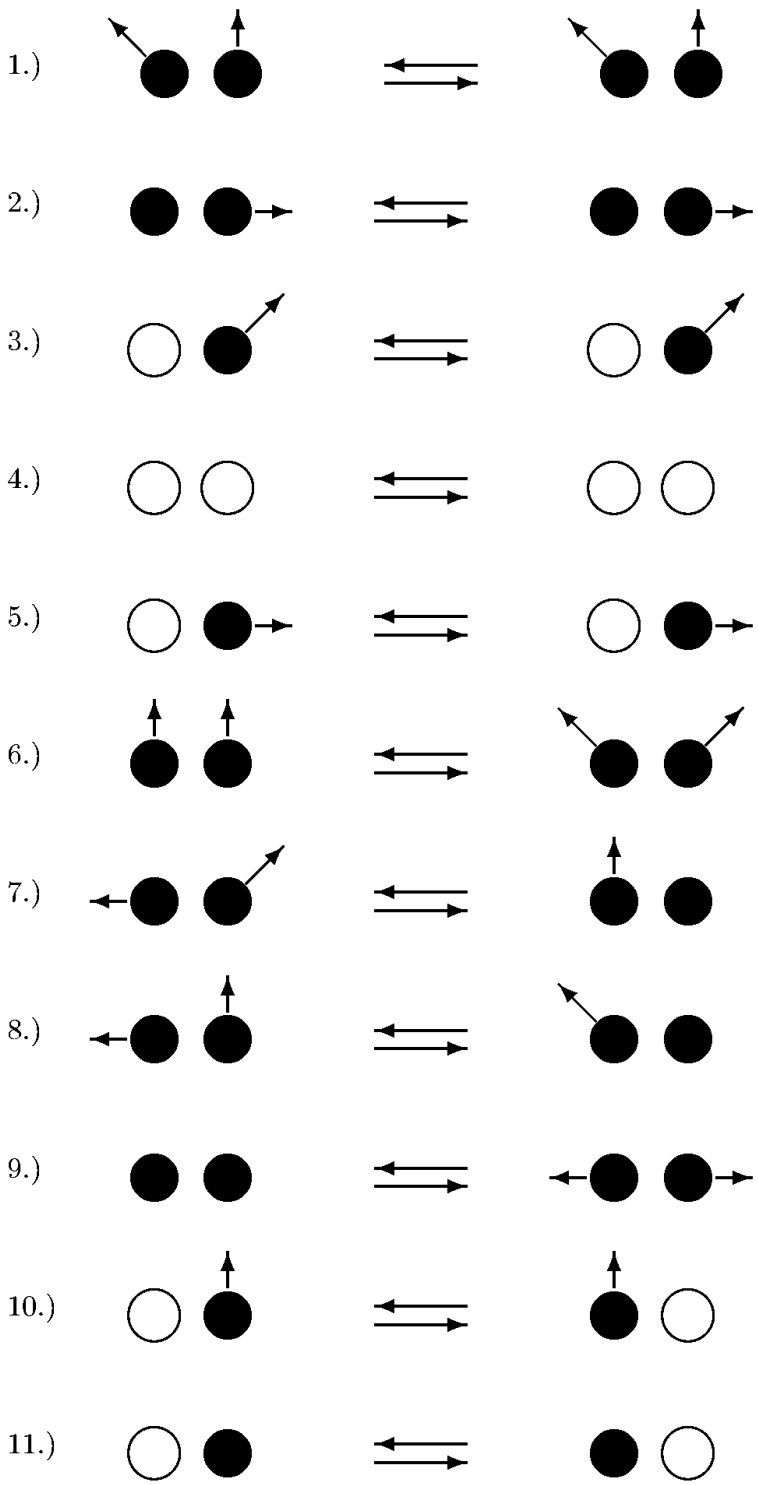


Figure 7.8: All admissible pair-interactions

7.4 Collision

The purpose of collision is to change the state of the node $\boldsymbol{n}$ into new state $\boldsymbol{n'}$ such that 2 conditions are fulfilled:

1. **Collision operator is one-to-one**:

   If $\mathcal{C} : \boldsymbol{n} \to \boldsymbol{n'}$, then $\mathcal{C} : \boldsymbol{n'} \to \boldsymbol{n}$.

   This reversibility condition allows us to apply Gibbs formalism later on.

2. **Collision preserves mass and momentum in the nodes**

   Mass and momentum are given by

$$m_0 = \sum_i n_{i0} \tag{7.5}$$

$$p_\alpha = \sum_i c_{i\alpha} n_{i\alpha}, \ \alpha = 1, 2, ...D \tag{7.6}$$

   Using these definitions, condition of mass and momentum conservation could be expressed by:

$$\begin{aligned} m_o = m'_o \quad &\Leftrightarrow \quad \sum_i n_{i0} = \sum_i n'_{i0}, \\ p_\alpha = p'_\alpha \quad &\Leftrightarrow \quad \sum_i c_{i\alpha} n_{i\alpha} = \sum_i c_{i\alpha} n'_{i\alpha}, \ \alpha = 1, 2, ...D \end{aligned} \tag{7.7}$$

   The easiest collision rule fulfilling these conditions would be identity

$$\mathcal{I} : \boldsymbol{n} \to \boldsymbol{n} \tag{7.8}$$

but this "non-collision" would lead to undesired, non-physical invariants. On the contrary, we want to change as many bits in the node as possible, or to say it physically, change directions to as many particles as conservation of momentum allows. Providing that, the mean free path of particles is reduced to minimum, and we can reach higher Reynold numbers in the simulations.

Such collision algorithm is surprisingly simple and was described in previous section graphically. Now we state collision rule more formally for arbitrary dimension D. The algorithm consists of D steps. In each step ($d = 1...D$), we create pairs of cells in d^{th} direction, or equivalently, we create pairs (n_i, n_j), such that $c_{id} = -c_{jd}$ and $c_{i\alpha} = c_{j\alpha}$ for $\alpha \neq d$. If $n_{id} = n_{jd}$ (it means that in direction d, momentum of the pair is zero), we swap the states of the cells n_i and n_j without changing the total momentum of the pairs. In the end of the collision, the total momentum of the node is conserved, because all the pair-interactions conserved the momentum.

7.5 Equilibrium statistics

We start by revoking the basic result of the microdynamics

$$\boldsymbol{n}(t+2,.) = \mathcal{U}^2\boldsymbol{n}(t,.) \tag{7.9}$$

In the odd and even time steps, the different parts of the hlattice are occupied, so it does not make sense to compare them. That is why we are comparing states of the lattice over the two time steps.

The microdynamical equation 7.9 implies conservation of probabilities

$$P(t,\boldsymbol{n}(.)) = P(t+2,\mathcal{U}^2\boldsymbol{n}(.)) \tag{7.10}$$

where $P(t,n(.))$ is the probability of occurrence of lattice in some state $n(.)$ from the statistical ensable of automata.

7.5.1 Gibbs distribution

Let's see if the Gibbs probability distribution fulfill the conservation law 7.10.

First we define the total momentum of the whole lattice $\mathcal{L}$:

$$m_\alpha(.) = \sum_{i,\boldsymbol{r}} c_{i\alpha} n_{i\alpha}(\boldsymbol{r}),\ \alpha = 0,1,..D \tag{7.11}$$

Now we can define Gibbs probability for our model:

$$P^E(n(.)) = \frac{W(n(.))}{Z} \tag{7.12}$$

where

$$W(n(.)) = exp(-\sum_{\alpha=0}^{D} \mu_\alpha m_\alpha(.)) \tag{7.13}$$

and Z is the partition function

$$Z = \sum_{n(.))} W(n(.)) \tag{7.14}$$

Conservation of local momenta 7.7 implies conservation of the total momentum

$$m_\alpha(.) = m'_\alpha(.),\ \alpha = 0,1,..D \tag{7.15}$$

that immediately implies conservation of Gibbs probability

$$P^E(t+2,\mathcal{E}^2 n(.)) = P^E(t,n(.)). \tag{7.16}$$

Additivity of the momentum cell by cell over the whole lattice (equation 7.11) implies that there is no statistical correlation between different cells:

$$P^E(n(.)) = \prod_{i,\boldsymbol{r}} P^E(n_i(\boldsymbol{r})) \tag{7.17}$$

Even further, if we consider, that

$$P(n_i(\boldsymbol{r})) = \frac{W(n_i(\boldsymbol{r}))}{Z} \tag{7.18}$$

where

$$W(n_i(\boldsymbol{r})) = exp(-\sum_{\alpha=0}^{D} \mu_\alpha c_{i\alpha} n_{i\alpha}(\boldsymbol{r})) = \prod_{\alpha=0}^{D} exp(-\mu_\alpha c_{i\alpha} n_{i\alpha}(\boldsymbol{r})) = \prod_{\alpha=0}^{D} W(n_{i\alpha}(\boldsymbol{r})) \tag{7.19}$$

we can see that bits of the cells $n_{i\alpha},\ \alpha = 1, 2, ...D$ are not statistically correlated.

For sure, they are correlated with n_{i0}, because $n_{i0} = 0 \Rightarrow n_{i\alpha} = 0\ for\ \forall\ \alpha = 1, 2, ...D$.

But for $\alpha = 1, 2, ...D$ we can write:

$$\sigma_{i\alpha} = prob(n_{i\alpha} = 1 | n_{i0} = 1) = \frac{W(n_{i\alpha})}{Z_{i\alpha}} \tag{7.20}$$

where

$$W(n_{i\alpha}) = exp(-\mu_\alpha c_{i\alpha} n_{i\alpha}) \tag{7.21}$$

and

$$Z_{i\alpha} = \sum_{n_{i\alpha}=0}^{1} W(n_{i\alpha}). \tag{7.22}$$

Hence

Little more laborious calculation would lead us to mean occupation numbers

$$\rho_i := < n_{i0} > = \frac{Z_i - 1}{Z_i} = \frac{e^{-\mu_0} \prod_{\alpha=1}^{D} (1 + e^{-\mu_\alpha c_{i\alpha}})}{1 + e^{-\mu_0} \prod_{\alpha=1}^{D} (1 + e^{-\mu_\alpha c_{i\alpha}})} = f\big(\mu_0 + \sum_{j=1}^{d} lnf(-\mu_j v_j)\big). \tag{7.23}$$

ρ_i can be interpreted as probability that cell n_i is occupied, but also as the equilibrium mass of the cell.

To define the density, we need to consider what is the volume of the node. Since lattice vectors $c_{i\alpha}$ span interval $[-1, 1]^D$, we can define volume V as

$$V = 2^D \tag{7.24}$$

Then the mass density reads

$$\rho = 2^{-D} \sum_i \rho_i \tag{7.25}$$

and the momentum density

$$q_a := 2^{-D} \sum_i c_{i\alpha} < n_{i\alpha} > = 2^{-D} \sum_i c_{i\alpha} \rho_i \sigma_{i\alpha}. \tag{7.26}$$

It is efficient to use only one quantity instead of mass and momentum density, and our formalism invites us to do so. We define the **hypermomentum** density

$$q_\alpha := 2^{-D} \sum_i c_{i\alpha} < n_{i\alpha} > \tag{7.27}$$

where q_0 is the mass density ρ, and $(q_1, q_2, ...q_D)$ is the momentum density.

Expanding formulas 7.23 and 7.20 in Taylor series and inserting to the last equation leads to rather lengthy calculation (Appendix B in [2]), that results in

$$\begin{aligned} \rho_i &= \rho + 2\frac{1-\rho}{2-\rho}\boldsymbol{v}.\boldsymbol{q} + 2\,\frac{(1-\rho)(1-2\rho}{(2-\rho)^2\rho}\left[(\boldsymbol{v}.\boldsymbol{q})^2 - q^2\right] + O(q^3), \\ \sigma_{ij} &= \frac{1}{2} + \frac{v_j q_j}{(2-\rho)\rho} + O(q^3) \end{aligned} \tag{7.28}$$

7.6 Hydrodynamic description

It is based on the assumption that the state of the cellular automaton is near its equilibrium. In that case, the whole state of the automaton can be specified by a few macroscopic order parameters q_α.

For example, expectation values $p_{i\alpha}$ are continuous differentiable functions of the parameter q

$$p_{i\alpha} = p_{i\alpha}(q, \nabla q, \nabla^2 q, ...) \tag{7.29}$$

and we can expand them in Chapman-Enskog series in terms of $p^{eq}_{i\alpha}$

$$p_{i\alpha} = p^{eq}_{i\alpha} + r_{i\alpha} + \mathcal{O}(\nabla^2) \tag{7.30}$$

with

$$r_{i\alpha} = \sum_{\beta j} R_{i\alpha\beta j} \nabla_j q_\beta. \tag{7.31}$$

We will use this expansion later on.

The resulting equations of this section will be the hydrodynamic equations for the pair-interaction automaton.

They are the set of evolution equations for the order parameters

$$\partial_t q_\alpha = \dot{q_\alpha}(q, \nabla q, \nabla^2 q, ...) \tag{7.32}$$

Due to conservation of the hypermomentum, the right hand side of 7.32 must be of the form

$$\dot{q_\alpha} = -\sum_k \nabla_k Q_{\alpha k}(q, \nabla q, \nabla^2 q). \tag{7.33}$$

Let us look back to the microdynamical propagation equation

$$n_{i\alpha}(t+1, \boldsymbol{r} + \boldsymbol{c_i}) = n'_{i\alpha}(t, \boldsymbol{r}), \tag{7.34}$$

that directly implies

$$p_{i\alpha}(t+1, \boldsymbol{r}+\boldsymbol{c_i}) = p'_{i\alpha}(t, \boldsymbol{r}). \tag{7.35}$$

If we expand $p_{i\alpha}(t+1, \boldsymbol{r}+\boldsymbol{c_i})$ in terms of $p_{i\alpha}(t, \boldsymbol{r})$, insert it to 7.35 and apply $2^{-D}\sum_i c_{i\alpha}$ on both sides, we get

$$2^{-D}\sum_i c_{i\alpha}\big[(\partial_t + c_{i\alpha}\nabla_\alpha) + \frac{1}{2}(\partial_t + c_{i\alpha}\nabla_\alpha)^2 + ...\big]p_{i\alpha} = 0 \tag{7.36}$$

The right-hand side disappeared because of the conservation of hypermomentum

$$2^{-D}\sum_i c_{i\alpha}p_{i\alpha} = 2^{-D}\sum_i c_{i\alpha}p'_{i\alpha}. \tag{7.37}$$

Analogously to FHP and FCHC model, we can define various temporal and spatial scales (see Table 5.5) and expand operator of time derivative and space derivative into modes

$$\begin{aligned} \partial_t &= \epsilon\partial_t^{(1)} + \epsilon^2\partial_t^{(2)} + ... \\ \partial_\alpha &= \epsilon\partial_\alpha^{(1)} \end{aligned} \tag{7.38}$$

Inserting 7.30 and 7.38 into 7.36 leads to

$$\begin{aligned} 0 = 2^{-D}\sum_i c_{i\alpha}(\epsilon\partial_t^{(1)} + c_i\nabla_i)p_{i\alpha}^{eq} \\ +2^{-D}\sum_i c_{i\alpha}\big[\epsilon^2\partial_t^{(2)}p_{i\alpha}^{eq} + \frac{1}{2}(\epsilon\partial_t^{(1)} + c_{i\alpha}\nabla_\alpha)^2 p_{i\alpha}^{eq} \\ +(\partial_t^{(1)} + c_{i\alpha}\nabla_\alpha)\sum_{j\beta} R_{\beta j\alpha i}\nabla_j q_\beta\big] + \mathcal{O}(\nabla^3) \end{aligned} \tag{7.39}$$

Each order of ϵ must vanish separately.
For the terms linear in ϵ we get

$$2^{-D}\sum_i c_{i\alpha}(\partial_t^{(1)} + c_{i\alpha}\nabla_\alpha)p_{i\alpha}^{eq} = 0 \tag{7.40}$$

or equivalently in terms of momentum density:

$$\partial_t^{(1)}q_\alpha + \sum_k \nabla_k Q^0_{\alpha k} = 0 \tag{7.41}$$

where we used definition of hypermomentum 7.27, and defined zeroth term of momentum flux tensor $Q^0_{\alpha k}$:

$$Q^0_{\alpha k} := 2^{-D}\sum_i c_{i\alpha}c_{ik}p_{i\alpha}^{eq} \tag{7.42}$$

We can split equation 7.6 into two - mass density conservation and momentum conservation:

$$\partial_t\rho + \boldsymbol{\nabla.g^0}(\rho, \boldsymbol{q}) = \mathcal{O}(\nabla^2) \tag{7.43}$$

$$\partial_t q_j + \sum_k \nabla_k Q^0_{jk}(\rho, \boldsymbol{q}) = \mathcal{O}(\nabla^2) \tag{7.44}$$

where we used $g_k := Q_{0k}$.

7.7 Hydrodynamic limiting cases

In this section, we will derive hydrodynamic limiting cases of our model, that corresponds to four physical approximations, namely

1. compressible Euler equation,
2. acoustic limit,
3. inviscid incompressible Euler equation,
4. incompressible Euler equation with viscosity.

Starting with equation 7.43, blabla dokonci
expand in $\epsilon \ll 1$, that is defined as the Knudsen number (ratio of mean free path and macroscopic scale).

1) Compressible Euler equation
For physical fluid, the equations read

$$\begin{aligned} \partial_t \rho + \boldsymbol{\nabla}.\boldsymbol{q} = 0, \\ \partial_t q_j + \sum_k \nabla_k \big[p(\rho)\, \delta_{jk} + \frac{1}{\rho}\, q_j\, q_k = 0 \end{aligned} \tag{7.45}$$

For our model, we need to pick up terms of order

$$\partial_t = O(\epsilon^2),\ \boldsymbol{\nabla} = O(\epsilon^2),\ \rho = \frac{1}{2} + O(\epsilon), \boldsymbol{q} = O(\epsilon) \tag{7.46}$$

and we get

$$\begin{aligned} \partial_t \rho + \boldsymbol{\nabla}.\big[(\frac{10}{9} - \frac{8}{9} big)\boldsymbol{q}\big] = 0, \\ \partial_t q_j + \sum_k \nabla_k \big[\frac{\rho}{2}\delta_{jk} + \frac{8}{9} q_j\, q_k\big] = 0 \end{aligned} \tag{7.47}$$

As in case of FHP, the absence of Galilei invariance in our model caused the unphysical dependence of the pressure p on ρ.

2) Acoustic limit
Picking up the terms of order

$$\partial_t = O(\epsilon),\ \boldsymbol{\nabla} = O(\epsilon),\ \rho = \rho_0 + O(\epsilon),\ \boldsymbol{q} = O(\epsilon) \tag{7.48}$$

leads to linear equations of sound

$$\begin{aligned} \partial_t \rho + 2\frac{1-\rho_0}{2-\rho_0} \boldsymbol{\nabla}.\boldsymbol{q} = 0, \\ \partial_t \boldsymbol{q} + \frac{1}{2}\nabla\rho = 0\,. \end{aligned} \tag{7.49}$$

3) Inviscid incompressible Euler equations

For the ideal physical fluid (ideal fluid means inviscid and incompressible), the equations read

$$\begin{aligned} \boldsymbol{\nabla}.\boldsymbol{u} &= 0, \\ (\partial_t + \boldsymbol{u}.\boldsymbol{\nabla})\boldsymbol{u} + \nabla\Phi &= 0, \end{aligned} \tag{7.50}$$

with $\Phi = p/\rho_0 = 2p$ being the kinematic pressure.

For our model, taking terms of order

$$\partial_t = O(\epsilon^3), \qquad \boldsymbol{\nabla} = O(\epsilon^2), \qquad \rho = \frac{1}{2} + O(\epsilon^2), \qquad \boldsymbol{q} = O(\epsilon) \tag{7.51}$$

from the expansion (cislo) leads us to

$$\begin{aligned} \boldsymbol{\nabla}.\boldsymbol{q} &= 0, \\ (\partial_t + \frac{8}{9}\boldsymbol{q}.\boldsymbol{\nabla})\boldsymbol{q} + \frac{1}{2}\boldsymbol{\nabla}\rho &= 0. \end{aligned} \tag{7.52}$$

At last, our automaton got the hydrodynamic equations right, as we can easily transform then into 7.50 by setting

$$\Phi = \frac{4}{9}\rho, \qquad \boldsymbol{u} = \frac{8}{9}\boldsymbol{q}. \tag{7.53}$$

The $\boldsymbol{u}$ is the hydrodynamic velocity and

Notice that the hydrodynamic velocity $\boldsymbol{u}$ is the momentum convection velocity, and can not be interpreted as the mean velocity of particles

$$\boldsymbol{w} = \frac{\langle \sum_i v_i n_{i0} \rangle}{\langle \sum_i n_{i0} \rangle} = ... = \frac{2(1-\rho)}{(2-\rho)\rho}\boldsymbol{q}. \tag{7.54}$$

The reason behind this difference lies in the broken Galilean invariance of our model. The ratio between the velocities is called the "g-factor" ($g(\rho)$, and for our special case, it is equal to

$$\boldsymbol{w} = \frac{4}{3}\boldsymbol{q} = \frac{2}{3}\boldsymbol{u}. \tag{7.55}$$

It already arised in the FHP and FCHC model (where $g(\rho) \sim 1/2$ for small ρ).

The Navier-Stokes equations

For most applications, the inviscid incompressible approximation is too much idealized, and we would like to find an approximation of Navier-Stokes equations

$$\begin{aligned} \boldsymbol{\nabla}.\boldsymbol{u} &= 0, \\ (\partial_t + \boldsymbol{u}.\boldsymbol{\nabla}\boldsymbol{u} + \nabla\Phi &= \nu\nabla^2\boldsymbol{u} \end{aligned} \tag{7.56}$$

with the friction term on the right hand side, containing the shear viscosity ν.

For our model, the analogous set of equations is obtained by picking up terms

$$\partial_t = O(\epsilon^2), \qquad \nabla = O(\epsilon), \qquad , \rho = \frac{1}{2} + O(\epsilon), \qquad \boldsymbol{q} = O(\epsilon). \tag{7.57}$$

that leads to

$$\begin{aligned} \boldsymbol{\nabla}.\boldsymbol{q} &= 0, \\ (\partial_t + \frac{8}{9}\boldsymbol{q}.\boldsymbol{\nabla})q_j + \frac{1}{2}\nabla_j\rho &= \sum_{klm} T_{jklm}\nabla_l\nabla_m q_k. \end{aligned} \tag{7.58}$$

Is this set of equations equivalent to Navier-Stokes 7.56? If we could show that T_{jklm} is isotropic, the equations would be equivalent by the transformation 7.53. However, this is not the case, our model do not posses symmetries that forces T_{jklm} to be isotropic. We need to go to the second approximation to get more information of the tensor T_{jklm}.

Lengthy calculation that can be found in [6] leads to

$$\begin{aligned} T_{00km} &= \frac{1}{9}\delta_{kl}, \\ T_{jklm} &= \frac{1}{6}(3\delta_{jl}\delta_{km} + \delta_{jm}\delta_{lk}(1+\gamma_{mk}) - 2\delta_{jklm}). \end{aligned} \tag{7.59}$$

To see if it is isotropic or not, let us consider simple example of two-dimensional flow in $x_1 - direction$

$$q_1 = q_1(t, x_2), \qquad q_2 = 0. \tag{7.60}$$

Then, the Navier-Stokes equations 7.56 reduces to

$$\partial_t q_1 = \nu(\nabla_2)^2 q_1, \tag{7.61}$$

where $\nu = T_{1122}$ is the shear viscosity coefficient.

Let us rotate the flow by the angle α. We get

$$\partial_t q_1' = \nu'(\nabla_2')^2 q_1' \tag{7.62}$$

where the prime quantities are obtained by transformation

$$\begin{aligned} \boldsymbol{x} &= R\,\boldsymbol{x'}, \\ \boldsymbol{\nabla} &= R\boldsymbol{\nabla}, \\ \boldsymbol{q} &= R\boldsymbol{q'} \\ T'_{j'k'l'm'} &= \sum_{jklm} T_{jklm}R_{jj'}R_{kk'}R_{ll'}R_{mm'} \end{aligned} \tag{7.63}$$

with the orthogonal matrix

$$R = \begin{pmatrix} cos(\alpha) & sin(\alpha) \\ -sin(\alpha) & cos(\alpha) \end{pmatrix}. \tag{7.64}$$

The viscosity transformed into rotated coordinate system reads

$$\begin{aligned} \nu' = T'_{1122} = T_{1122}cos^4(\alpha) + T_{2211}sin^4(\alpha) \\ +(T_{1111} + T_{2222} - T_{1212} - T_{2121} - T_{1221} - T_{2112})cos^2(\alpha)sin^2(\alpha). \end{aligned} \tag{7.65}$$

The viscosity tensor in original coordinate system reads

$$\begin{aligned} T_{1122} = T_{2211} &= \frac{1}{2}, \\ T_{1111} = T_{2222} &= \frac{1}{3}, \\ T_{1212} = T_{2121} &= 0, \\ T_{1221} &= \frac{1}{6}, \\ T_{2112} &= \frac{1}{2}. \end{aligned} \tag{7.66}$$

Inserting these components into 7.65yields

$$\nu' = \frac{1}{2}(cos^4(\alpha) + sin^4(\alpha)), \tag{7.67}$$

which clearly shows that the shear viscosity is angle-dependent in contrast with the physical fluid and proves anisotropy of the viscosity tensor 7.59.

8. From Boltzmann to Navier-Stokes

Because microdynamics of LGCA lead to equilibrium occupation numbers give by Fermi-Dirac distribution, there is no way to improve them substantionally u

In previous chapter we saw that microdynamics of LGCA lead to equilibrium occupation numbers give by Fermi-Dirac distribution. Treatment of the symptoms resulting from this flaw is imperfect.

In this chapter we will show what statistical properties would be desirable for better computational model. We will show that microdynamics of physical fluid lead to Boltzmann distribution, and starting there, we will derive Navier-Stokes equations in its most general form.

Indeed, the new generation of LGCA, the Lattice-Boltzmann models, were pushed one ladder of abstraction higher to meet the Boltzmann distribution of real fluids. But since these models do not have discrete microdynamics anymore, they are not CA in its true sense, and they are beyond the scope of this thesis. However, we would like to show some light at the tunnel and motivate the method, that is an efficient competitor to other CFD methods.

The name of the chapter is shamelessly stollen from [1]. In the original piece, it precedes chapter on Lattice-Boltzman model and is full of technicalities (we have seen Chapman - Enskog expansion already), we rather follow approach of [**?**], that offers better physical insight.

In Wolf-Gladrow, the derivation of Navier-Stokes is performed by the means of Chapman-Enskog expansion. We chose approach from [**?**], as it offers us better physical insight with less technicalities then [1], where the derivation of Navier-Stokes is performed by the means of Chapman-Enskog expansion.

8.1 Boltzmann equation

Let us consider an enslable of identical particles. Quantum effects, such as internal degrees of freedom in particles will be neglected. The state of the ensamble will be given by distribution function $f(\boldsymbol{x}, \boldsymbol{u}, t)$, that is defined by

$$dN = f(\boldsymbol{x}, \boldsymbol{u}, t)\boldsymbol{dx}\,\boldsymbol{du} \tag{8.1}$$

where dN is the number of particles in the infinitesimal phase element $[x, x + dx].[u, u + du]$. It must be emphasized that statistical description is appropriate only if dN is sufficiently large number.

Let us suppose that external force field $\boldsymbol{F} = (F_1, F_2, F_3)$ is affecting the system such that it is same for all particles in volume element $[x, x + dx]$ (it is reasonable assumption, since particles are infinitesimally close).

Then, at any time t the velocity of the particles in the element $[x, x + dx]$ will change from (let's say) $\boldsymbol{u'}$ to $\boldsymbol{u'} + \boldsymbol{F}dt$.

$$\partial_t f + \boldsymbol{v}.\nabla f + \frac{\boldsymbol{K}}{m}\partial_{\boldsymbol{v}} f = Q(f, f) \tag{8.2}$$

Thus, if we neglect the collisions between particles, we can say, that number of particles $f(\boldsymbol{x}, \boldsymbol{u}, t)$ is equal to $f(\boldsymbol{x} + \boldsymbol{u}dt, \boldsymbol{u} + \boldsymbol{F}dt, t + dt)$, or

$$f(\boldsymbol{x}, \boldsymbol{u}, t) - f(\boldsymbol{x} + \boldsymbol{u}dt, \boldsymbol{u} + \boldsymbol{F}dt, t + dt) = 0. \tag{8.3}$$

If we consider the collisions between particles, we need to write

$$f(\boldsymbol{x}, \boldsymbol{u}, t) - f(\boldsymbol{x} + \boldsymbol{u}dt, \boldsymbol{u} + \boldsymbol{F}dt, t + dt) = [\Delta f]_{coll}, \tag{8.4}$$

where right hand side stands for change of f in a time interval dt due to collisions.

Since the left hand side is the difference of function values in infinitesimal time dt, we can write it in differential form

$$\frac{\partial f}{\partial t} + u_i \frac{\partial f}{\partial x_i} + F_i \frac{\partial f}{\partial u_i} = \big[\frac{\partial f}{\partial t}\big]_{coll}. \tag{8.5}$$

This is the famous Boltzmann equation and we derived it in very intuitive way. We need to mention three subtle assumptions that Boltzmann used to derive it.

1. Only one-particle collisions are considered (so it would apply only to diluted gas)
2. *Molecular chaos hypothesis* - velocities of colliding particles are uncorrelated
3. External forces do not influence dynamics of collisions

8.2 Macroscopic quantities

By equation 8.1 we defined distribution function as the density probability, or density of particles in the phase space. Hence the number of particles over some region of phase space is obtained by

$$n(\boldsymbol{x}, t) = \int f(\boldsymbol{x}, \boldsymbol{u}, t) \mathrm{d}\boldsymbol{x} \mathrm{d}\boldsymbol{u}. \tag{8.6}$$

Note that by integrating only by $\mathrm{d}\boldsymbol{u}$, not by volume element $\mathrm{d}\boldsymbol{x}$, the quantities that we obtain are densities over unit volume.

Hence we define mass density

$$\rho(\boldsymbol{x}, t) = \int m\, f(\boldsymbol{x}, \boldsymbol{u}, t) \mathrm{d}\boldsymbol{u}, \tag{8.7}$$

and the mean velocity of the flow

$$\boldsymbol{v}(\boldsymbol{x}, t) = \int \boldsymbol{u}\, m\, f(\boldsymbol{x}, \boldsymbol{u}, t) \mathrm{d}\boldsymbol{u}. \tag{8.8}$$

For the low densities we can neglect many-particle collisions and consider only two-particle collisions that are elastic (preserving energy and momentum of the colliding pair). Then, the density function f is the famous Maxwell function

$$f^M(\boldsymbol{x}, \boldsymbol{v}, t) = n(\frac{m}{2\pi k_B T})^{3/2} e^{-\frac{m}{2k_B T}(\boldsymbol{v}-\boldsymbol{u})^2}. \tag{8.9}$$

8.3 Hydrodynamic equations

The right hand side of **??** is often referred to as collision integral and it is so ugly expression we do not to state it explicitly. However, its general properties can be analyzed anyway.

Since the collisions among particles should preserve the number of particles in the system, then

$$\int \left[\frac{\partial f}{\partial t}\right]_{coll} \mathrm{d}\boldsymbol{u} = 0. \tag{8.10}$$

The total momentum is also preserved by the collisions, so

$$\int \boldsymbol{u}\left[\frac{\partial f}{\partial t}\right]_{coll} \mathrm{d}\boldsymbol{u} = 0, \tag{8.11}$$

and finally, the total energy is conserved by the collisions, hence

$$\int u^2\left[\frac{\partial f}{\partial t}\right]_{coll} \mathrm{d}\boldsymbol{u} = 0. \tag{8.12}$$

And because total energy of the system is finite, it must be true that

$$\lim_{u\to\infty} u^2 f = 0. \tag{8.13}$$

Using equation 8.13 and performing integration by parts leads to

$$\int \frac{\partial f}{\partial u_i} \mathrm{d}\boldsymbol{u} = 0, \tag{8.14}$$

$$\int u_i \frac{\partial f}{\partial u_i} \mathrm{d}\boldsymbol{u} = -\delta_{ij}\frac{\rho}{m}, \tag{8.15}$$

$$\frac{1}{2}\int u^2 \frac{\partial f}{\partial u_i} \mathrm{d}\boldsymbol{u} = -v_i\frac{\rho}{m} \tag{8.16}$$

where we used definition of mean velocity and mass density (equations 8.11 and 8.13.

Now that we are equipped with equations above, let us multiply the Boltzmann equation by m and integrate over the whole domain. We obtain

$$m\int \frac{\partial f}{\partial t}\mathrm{d}\boldsymbol{u} + m\int u_i \frac{\partial f}{\partial x_i}\mathrm{d}\boldsymbol{u} + mF_i\int \frac{\partial f}{\partial u_i}\mathrm{d}\boldsymbol{u} = 0, \tag{8.17}$$

where right-hand side is zero due to equation 8.11.

We can delete the third term due to 8.14, and since $\boldsymbol{u}$ and $\boldsymbol{x}$ are considered to be independent variables in the phase space, we can swap the integration by u_i and derivation by x_i and t. Hence we get

$$\frac{\partial}{\partial t}\int m\, f \mathrm{d}\boldsymbol{u} + \frac{\partial}{\partial x_i}\int u_i\, m\, f\, \mathrm{d}\boldsymbol{u} = 0, \tag{8.18}$$

and applying definition of density **??** and mean velocity 8.8 we can write in the form

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_i}(\rho v_i) = 0. \tag{8.19}$$

This is the common form of continuity equation, or law of mass conservation.

If we multiply the Boltzmann equation by the mass m and integrate over the whole domain again, we get

$$\frac{\partial}{\partial t}(\rho\, v_i) + \frac{\partial}{\partial x_j} \int m\, u_i\, u_j\, f \,\mathrm{d}\boldsymbol{u} - \rho\, F_i = 0, \tag{8.20}$$

where we used equality 8.15.

We can break down the second term into

$$\int m\, u_i\, u_j\, f \,\mathrm{d}\boldsymbol{u} = \int m\, v_i\, v_j\, f \mathrm{d}\boldsymbol{u} + \int (v_i - u_i)(v_j - u_j)\, f \,\mathrm{d}\boldsymbol{u} = \rho\, v_i\, v_j + P_{ij} \tag{8.21}$$

where we define

$$P_{ij} = \int m(v_i - u_i)(v_j - u_j)\, f \,\mathrm{d}\boldsymbol{u}. \tag{8.22}$$

In case that pressure is isotropic, it is proportional to Kronecker delta

$$P_{ij} = P\, \delta_{ij} \tag{8.23}$$

with

$$P = \frac{1}{3} P_{ii}. \tag{8.24}$$

Hence equation 8.20 simplifies to

$$\frac{\partial}{\partial t}(\rho\, v_i) + \frac{\partial}{\partial x_j}(\rho v_i v_j) = -\frac{\partial P}{\partial x_i} + \rho\, F_i \tag{8.25}$$

or in the vector form

$$\frac{\partial}{\partial t}(\rho \boldsymbol{v}) + \nabla.\pi = \rho \boldsymbol{F} \tag{8.26}$$

where

$$\pi_{ij} = \rho\, v_i\, v_j + P\delta_{ij} \tag{8.27}$$

This is the second hydrodynamic equation, expressing the conservation of momentum.

9. Probabilistic tools in turbulence

So far, the most successful approach in turbulence research is by the means of statistical analysis.

And life-long research of Frisch et. al., who conceived LGCA.

As we saw the previous chapters, it wanted statistical physics to explain how LGCA works.

LGCA and Turbulence are such a nice couple.

Our thesis thrives to be a little . So far, the most successful approach in turbulence research is statistical analysis.

In this chapter, we introduce some basic and more advanced concepts of probability that will be handy in our research.

Following text requires prior knowledge of some probability and statistics. Basic concepts will be slightly reminded and will explain more interesting and advanced stuff in detail.

9.1 Random variable

Random variable is a map from the space of the initial conditions to real numbers:

$$v : \Omega \to \mathbb{R} \tag{9.1}$$

A good example would be the x-component of a velocity at the time t and the position $\boldsymbol{r}$ in the turbulent flow.

It is useful to define cumulative probability function

$$F(x) = \mathrm{Prob}\left\{v(\omega) < x\right\} \tag{9.2}$$

which describes the probability that random variable v takes a value smaller than x. Since probability is non-negative measure, $F(x)$ is non-decreasing function.

Hence, its derivative is a non-negative function,

$$p(x) = \frac{\mathrm{d}F(x)}{\mathrm{d}x} \geq 0. \tag{9.3}$$

And it happens to be the famous *probability density function*

Loosely speaking, $p(x)dx$ is the probability that random variable takes value between x and $x + dx$.

Of course, probability is normalized to one:

$$\int_{\mathbb{R}} p(x)\, dx = 1. \tag{9.4}$$

Using *probability density function*, we can express mean value of v

$$\langle v \rangle = \int_{\mathbb{R}} x\, p(x)\, dx, \tag{9.5}$$

variance of v

$$\sigma^2 = \langle v^2 \rangle = \int_{\mathbb{R}} x^2 \, p(x) \, dx, \tag{9.6}$$

or any other moment of v

$$\langle v^m \rangle = \int_{\mathbb{R}} x^m \, p(x) \, dx. \tag{9.7}$$

In the following text, we consider v to be centered. It means

$$\langle v \rangle = 0.$$

For non-centered variables w, all statements will be valid for $w' = w - \langle w \rangle$.

9.2 Characteristic function

Of special interest is the Fourier transform of *probability density function*

$$K(z) = \int_{\mathbb{R}} e^{izv} p(x) dx = \langle e^{izv} \rangle.$$

It is called *characteristic function* of variable v.

It is interesting for couple of reasons. For example, if we consider sum of two random variables $u+v$, corresponding $p.d.f.$ would be convolution of their respective $p.d.f.s$. But its characteristic function is product of respective characteristic functions. Hooray!

But for Guassian random variables, characteristic function leads us to various useful formulas.

9.3 Gaussian random variables

Gaussian random variable can be characterized by - the characteristic function. Because its characteristic function can be simplified to

$$K(z) = \langle e^{izv} \rangle = e^{-\frac{1}{2}\sigma^2 \, v^2}$$

9.4 Vector random variable

Let us go back to the example of random variable - the x-coordinate of velocity at specific time t and specific position $\boldsymbol{r}$ in turbulent flow v_x.

Not surprisingly, vector random variable would be velocity vector $\boldsymbol{v} = (v_x, v_y, v_z)$ in turbulent flow.

Some definitions from previous sections are easy to generalize to vector variables and we omit them, but sometimes we need to redefine them.

In general, moments of the vector random variables are tensors

$$\langle v_{i_1}, v_{i_2}, ..., v_{i_n} \rangle \tag{9.8}$$

The most important among them is the covariance

$$\Gamma_{ij} = \langle v_i v_j \rangle \tag{9.9}$$

Another definition worth of commentary is the definition of Gaussian random variables. Vector random variable $\boldsymbol{v}$ is Gaussian, if scalar product $\boldsymbol{v}.\boldsymbol{c}$ is Gaussian (in the sense of previous definition) for every $\boldsymbol{c} \in R^n$.

And if $\boldsymbol{v}$ is Gaussian, its characteristic function takes simple form

$$K(\boldsymbol{z}) = \langle e^{i\boldsymbol{z}.\boldsymbol{v}} \rangle = e^{-\frac{1}{2} z_i z_j \Gamma_{ij}} \tag{9.10}$$

We see that characteristic function $K(\boldsymbol{z})$ depends only on the covariance tensor Γ_{ij}.

Starting from that, we could show that any statistical moment of $\boldsymbol{v}$ can be expressed as function of covariance. For example

$$\langle v_i v_j v_k v_l \rangle = \Gamma_{ij}\Gamma_{kl} + \Gamma_{il}\Gamma_{jk} + \Gamma_{ik}\Gamma_{jl}. \tag{9.11}$$

We state it without proof, as it would require too much paper.

9.5 Random function

Random function can be defined as "the family of random variables" - sequence that is indexed by continuous time and spatial variables (vector random variable is sequence of random variables indexed by enumerable and finite set, e.g. $I = \{1, 2, ..N\}$.

Going back to our velocity example, random function would be the whole velocity field in turbulent flow $\boldsymbol{v}(t, \boldsymbol{r})$.

Definitions from the previous sections are easy to upgrade from random functions. Just bear in mind, that scalar product of random functions u and v means

$$\int_\Omega u\, v\, dx. \tag{9.12}$$

This definition of scalar product is correct under assumption that random functions are real and square-integrable (from L^2 space).

9.6 Stationary random variable

Loosely said, $v(t)$ is stationary if $v(t)$ and $v(t+h)$ have same statistical properties.

For example, for correlation function we have

$$\langle v(t_1 + h)v(t_2 + h) \rangle = \langle v(t_1)v(t_2) \rangle = \Gamma(t_1 - t_2). \tag{9.13}$$

Hence, variance $\langle v(t)^2 \rangle = \Gamma(0)$ does not depend on time at all.

But be careful, difference of stationary function is not necessarily stationary.

For example, if $v(t)$ is stationary, then $v(2t)$ is also stationary, because its variance

$$\langle v(2t)^2 \rangle = \Gamma(2t - 2t) = \Gamma(0) \tag{9.14}$$

does not depend on time. However, their difference

$$\langle (v(2t) - v(t))^2 \rangle = \langle v^2(t) \rangle + \langle v^2(2t) \rangle + 2\langle v(t)\, v(2t) \rangle = 2\Gamma(0) + \Gamma(2t - t) \tag{9.15}$$

does depend on time.

9.7 Spectrum of stationary random functions

To analyze stationary random function, we have very powerful tool at our reach - the expansion of a random variable into Fourier harmonics:

$$v(t, \omega) = \int_R e^{itf} \hat{v}(f, \omega) df \tag{9.16}$$

where f denotes frequency.

To filter-out temporal scales greater than F^{-1} from the function v, we define

$$v_F^<(t) = \int_{|f| \leq F} e^{itf} \hat{v}(f, \omega) df \tag{9.17}$$

Let us define mean kinetic energy of random function

$$E = \frac{1}{2} \langle v^2 \rangle \tag{9.18}$$

Energy of harmonics with frequencies lower than F are then

$$\mathcal{E}(F) = \frac{1}{2} \langle (v_F^<(t))^2 \rangle \tag{9.19}$$

and energy spectrum E(f) is than defined as

$$E(f) = \frac{d\mathcal{E}(f)}{df}. \tag{9.20}$$

Since $\mathcal{E}(f)$ is non-decreasing, we have $E(f) \geq 0$.

Now we can express mean kinetic energy as

$$E = \int_0^\infty E(f) \tag{9.21}$$

It could be also shown that energy spectrum $E(f)$ is the Fourier transform of correlation function

$$E(f) = \frac{1}{2\pi} \int_R e^{ifs} \Gamma(s)\, ds, \tag{9.22}$$

what is known as *Wiener-Khinchin formula* (that Einstein already used in 1914).

Direct consequence of this formula is an expression for the *second order structure function*

$$\langle (v(t') - v(t))^2 \rangle = 2 \int_R (1 - e^{if(t'-t)})\, E(f)\, df. \tag{9.23}$$

As Kolmogorov proved in 1940, a power-law holds for energy harmonics in turbulent case:

$$E(f) = C|f|^{-n}, \quad C > 0 \tag{9.24}$$

If we substitute this expression into the structure function 9.23, after tedious and hideous calculation we get

$$\begin{aligned} \langle (v(t') - v(t))^2 \rangle &= C\, A_n |t' - t|^{n-1}, \\ A_n &= \int_R (1 - e^{ix}) |x|^{-n} dx \end{aligned} \tag{9.25}$$

for $1 < n < 3$, else the structure function diverges. For n = 2, we recover Brownian motion (formulation from the Frisch).

Part II

Implementation and applications

10. Practical part

In the theoretical part of the thesis, we have introduced the notion of cellular automaton and basic models of lattice-gas cellular automata by which it is possible to mimic the flow of physical fluids governed by the Euler equations and, to certain extent, the Navier-Stokes equations. We have described the algorithms behind the microdynamics of these models and emphasized the importance of the symmetries of the lattice for reproducing correct macroscopic behavior.

In this part, we implement the aforementioned algorithms in the language C++ and visualize the results using the *GNU*plot software. The original results of this thesis are summarized in the points to follow.

1. We implemented Turing-complete 1D and 2D cellular automata to show their interesting properties and motivate theoretical part. These results were already included in the theoretical part and will not be discussed further.

2. We implement FCHC and PI – the two distinct 3D LGCA introduced in theoretical part.

3. We inspect 3D flow around obstacles of various shapes.

4. We study fully-developed turbulent flow and compare its statistical properties to the Kolmogorov–Obuchov K41 theory. [1].

   Unfortunately, this most original and scientifically most interesting part is not concluded, but it provides the basis for future research.

11. General comments on the implementation

According to Wolf-Gladrow ([1], page. 2017), the most effective language for the implementation of LGCA is C (with slightly better performance than Fortran).

We listened to this advice and started the implementation in C language, but we could not resist to use some "shortcuts" of C++, but we stayed true to the C-style of programming.

The programs have substantially grew over the time, and we got on the crossroad how to efficiently develop our two models further. The current option number one (in the process while writing these lines) is to use the pure C, and pack it as *extension module* to Python, mainly for our own comfort in future applications, but we would be glad if it finds its way to some fellow out there.

11.1 Parallelization

Languages C and C++ offer us three popular and well documented interfaces for parallelization:

1. OpenMP
2. MPI
3. Cuda

We rather obey the old Chinese saying '谁闻起来像腋下汗湿的龙，不应该使用Cuda', so Cuda is obviously out of the question.

For a short time, MPI was a hot candidate, we have even used it on cellular automaton that is not included in this thesis. But after consulting the more experienced colleague, we concluded that minuses would overweight the pluses.

Since we have opporunity to use clusters at Metacenter (the network of clusters from academic institutions all over the Czech republic) we expected we could employ huge amount of processors, if we use MPI and compute on various clusters parallely. But we learnt that in the practice we can hardly employ more then two hundreds of processors. Using OpenMP, we can run the computation on SMP machine with 64 processors, that is not significantly lower.

The drawbacks of MPI are obvious, it requires vast modification of the code, comparing to OpenMP, that needs only few more lines of code to add (and sometimes, even that can be tricky).

Another aspect is that parallelization with OpenMP can be included to Python extension module, that we are considering to develop.

Therefore, OpenMP seemed like an appropriate choice.

In the following chapter, we offer brief discussion of the most important parts of our source code.

12. Implementation of FCHC

In the chapter blabla, we inspected FCHC automaton theoretically, you should be familiar with it before we start.

According to [**?**], (find idea on lack of implementation). And field of practical applications is ruled by Lattice Boltzmann model nowadays, so we really do not find many applications.

As the more advanced "lattice Boltzmann model" evolved from LGCA, it attracted And nowadays, the "lattice Boltzmann model" has taken over the application.

Due to huge number of state of the node and huge number of colli As state of the node is represented by 24 bits. Therefore, integer with its 32 bits is enough to represent both node and obstacle (24 bits + 1 bit).

Not so long ago, table that would specify collision rules for 2^{24} states was problematic - it would require approximately $2^{24} \times 10 \times 4bytes = 640$MB memory to manage.

Henon proposed algorithm for collision that would by-pass this obstacle, compute the possible set of states in reasonable time and chose optimal state.

However, the algorithm is quiet expensive on computation time, especially comparing to pair-interaction LGCA.

Therefore, we have chosen the following approach for resolving collisions:

1. At the beginning of the computation, before the simulation of the flow starts, we create table of 2^{24} entries. Integer index of the entry is the state of the node, and using Henon algorithm, we compute the optimal isometries for the state.

2. Collisions during simulation are resolved by choosing the resulting state from the table, instead of computing the optimal isometries again and again. We are not generating the random numbers to chose the resulting state, as it is expensive on resources and was significantly slowing the algorithm in the testing phase. However, we believe that choosing random states is not necessary and our solution for choosing the optimal isometries leads to the uniform distribution due to large number of nodes (around 10^8).

Excerpt of the source code relevant to the creation of the table and collision follows.

12.1 Algorithm for the creation of the table

```
/* The node is represented as the integer, first 24 bits corresponds
   to the cells. If the value of the cell is 1, it is occupied by
   particle, otherwise it is 0. */

#define C1 1
#define C2 (C1<<1)
#define C3 (C1<<2)
```

```
#define C4 (C1<<3)
#define C5 (C1<<4)
#define C6 (C1<<5)
#define C7 (C1<<6)
#define C8 (C1<<7)
#define C9 (C1<<8)
#define C10 (C1<<9)
#define C11 (C1<<10)
#define C12 (C1<<11)
#define C13 (C1<<12)
#define C14 (C1<<13)
#define C15 (C1<<14)
#define C16 (C1<<15)
#define C17 (C1<<16)
#define C18 (C1<<17)
#define C19 (C1<<18)
#define C20 (C1<<19)
#define C21 (C1<<20)
#define C22 (C1<<21)
#define C23 (C1<<22)
#define C24 (C1<<23)

/* We assigned the 25th bit to the obstacle. If it is 1, this node is
   part of the obstacle and is not occupied by the particles (thus if
   this bit is 1, all other bits are 0) */
#define OBS (C1<<24)

/* We create the array of the cells, so we can efficiently iterate
   over the cells */
int C[24] = {
   C1, C2, C3, C4,
   C5, C6, C7, C8,
   C9, C10, C11, C12,
   C13, C14, C15, C16,
   C17, C18, C19, C20,
   C21, C22, C23, C24
};


/* Array of the cells, such that Reverse[i] lies on the diagonal to
   the C[i] from above */
int Reverse[24] = {
   C4, C3, C2, C1,
   C8, C7, C6, C5,
   C12,C11,C10,C9,
   C16,C15,C14,C13,
   C20,C19,C18,C17,
   C24,C23,C22,C21
};
```

```
/* Lattice velocities. The lattice velocity c[i] corresponds to the
    cell C[i] */
const int c[24][4] = {
   { 1,1,0,0 },
   { 1,-1,0,0 },
   { -1,1,0,0 },
   { -1,-1,0,0 },

   { 1,0,1,0 },
   { 1,0,-1,0 },
   { -1,0,1,0 },
   { -1,0,-1,0 },

   { 1,0,0,1 },
   { 1,0,0,-1 },
   { -1,0,0,1 },
   { -1,0,0,-1 },

   { 0,1,1,0 },
   { 0,1,-1,0 },
   { 0,-1,1,0 },
   { 0,-1,-1,0 },

   { 0,1,0,1 },
   { 0,1,0,-1 },
   { 0,-1,0,1 },
   { 0,-1,0,-1 },

   { 0,0,1,1 },
   { 0,0,1,-1 },
   { 0,0,-1,1 },
   { 0,0,-1,-1 }
};

/* This function swap the i-th and j-th bit of the node n */
int switchBits(int n, int i, int j)
{
   int a, b;
   a = n & C[i];
   b = n & C[j];
   if (a > 0 && b == 0)
   {
      n ^= C[i];
      n |= C[j];
   }
   else if (b > 0 && a == 0)
   {
      n |= C[i];
      n ^= C[j];
   }
   return n;
```

```
}



/* Implementation of the isometries \Sigma_1 and \Sigma_2 on the node
   n*/
/* As all isometries that will follow, it is achieved by swapping the
   appropriate bits in the node */
/* It also transforms the momentum vector q */
/* The parameter j is dummy parameter, we need it so that all
   isometries have the same set of parameters */
int sigma(int n, int* q, int i, int j)
{
   int a = q[0];
   int b = q[1];
   int c = q[2];
   int d = q[3];

   if (i == 1)
   {
      q[0] = ( a + b + c - d ) / 2;
      q[1] = ( a + b - c + d ) / 2;
      q[2] = ( a - b + c + d ) / 2;
      q[3] = (-a + b + c + d ) / 2;

      n = switchBits(n, 1, 21);
      n = switchBits(n, 2, 22);
      n = switchBits(n, 5, 17);
      n = switchBits(n, 6, 18);
      n = switchBits(n, 8, 12);
      n = switchBits(n, 11, 15);
   }
   else
   {
      q[0] = ( a + b + c + d ) / 2;
      q[1] = ( a + b - c - d ) / 2;
      q[2] = ( a - b + c - d ) / 2;
      q[3] = ( a - b - c + d ) / 2;

      n = switchBits(n, 1, 20);
      n = switchBits(n, 2, 23);
      n = switchBits(n, 5, 16);
      n = switchBits(n, 6, 19);
      n = switchBits(n, 9, 12);
      n = switchBits(n, 10, 15);
   }
   return n;
}

/* Isometry S_i is the reflection over plane x_i */
int S(int n, int* q, int i, int j)
```

```
{
   switch (i)
   {
   case 1:
      q[0] = -q[0];
      n = switchBits(n, 0, 2);
      n = switchBits(n, 1, 3);
      n = switchBits(n, 4, 6);
      n = switchBits(n, 5, 7);
      n = switchBits(n, 8, 10);
      n = switchBits(n, 9, 11);
      break;
   case 2:
      q[1] = -q[1];
      n = switchBits(n, 0, 1);
      n = switchBits(n, 2, 3);
      n = switchBits(n, 12, 14);
      n = switchBits(n, 13, 15);
      n = switchBits(n, 16, 18);
      n = switchBits(n, 17, 19);
      break;
   case 3:
      q[2] = -q[2];
      n = switchBits(n, 4, 5);
      n = switchBits(n, 6, 7);
      n = switchBits(n, 12, 13);
      n = switchBits(n, 14, 15);
      n = switchBits(n, 20, 22);
      n = switchBits(n, 21, 23);
      break;
   case 4:
      q[3] = -q[3];
      n = switchBits(n, 8, 9);
      n = switchBits(n, 10, 11);
      n = switchBits(n, 16, 17);
      n = switchBits(n, 18, 19);
      n = switchBits(n, 20, 21);
      n = switchBits(n, 22, 23);
      break;
   default:
      break;
   }
   return n;
}

/* Isometry P_ij, reflecton over the plain x_i = x_j */
int P(int n, int*q, int i, int j)
{
   int a = i - 1;
   int b = j - 1;
   int qa = q[a];
```

```
q[a] = q[b];
q[b] = qa;

if (i == 1)
{
   if (j == 2)
   {
      n = switchBits(n, 1, 2);
      n = switchBits(n, 4, 12);
      n = switchBits(n, 5, 13);
      n = switchBits(n, 6, 14);
      n = switchBits(n, 7, 15);
      n = switchBits(n, 8, 16);
      n = switchBits(n, 9, 17);
      n = switchBits(n, 10, 18);
      n = switchBits(n, 11, 19);
   }
   else if (j == 3)
   {
      n = switchBits(n, 0, 12);
      n = switchBits(n, 1, 14);
      n = switchBits(n, 2, 13);
      n = switchBits(n, 3, 15);
      n = switchBits(n, 5, 6);
      n = switchBits(n, 8, 20);
      n = switchBits(n, 9, 21);
      n = switchBits(n, 10, 22);
      n = switchBits(n, 11, 23);
   }
   else if (j == 4)
   {
      n = switchBits(n, 0, 16);
      n = switchBits(n, 1, 18);
      n = switchBits(n, 2, 17);
      n = switchBits(n, 3, 19);
      n = switchBits(n, 4, 20);
      n = switchBits(n, 5, 22);
      n = switchBits(n, 6, 21);
      n = switchBits(n, 7, 23);
      n = switchBits(n, 9, 10);
   }
}
else if (i == 2)
{
   if (j == 3)
   {
      n = switchBits(n, 0, 4);
      n = switchBits(n, 1, 5);
      n = switchBits(n, 2, 6);
      n = switchBits(n, 3, 7);
      n = switchBits(n, 13, 14);
```

```
            n = switchBits(n, 16, 20);
            n = switchBits(n, 17, 21);
            n = switchBits(n, 18, 22);
            n = switchBits(n, 19, 23);
         }
         else if (j == 4)
         {
            n = switchBits(n, 0, 8);
            n = switchBits(n, 1, 9);
            n = switchBits(n, 2, 10);
            n = switchBits(n, 3, 11);
            n = switchBits(n, 12, 20);
            n = switchBits(n, 13, 22);
            n = switchBits(n, 14, 21);
            n = switchBits(n, 15, 23);
            n = switchBits(n, 17, 18);
         }
      }
      else if (i == 3 && j == 4)
      {
         n = switchBits(n, 4, 8);
         n = switchBits(n, 5, 9);
         n = switchBits(n, 6, 10);
         n = switchBits(n, 7, 11);
         n = switchBits(n, 12, 16);
         n = switchBits(n, 13, 17);
         n = switchBits(n, 14, 18);
         n = switchBits(n, 15, 19);
         n = switchBits(n, 21, 22);
      }
      return n;
   }

   /* Computes the total momentum of the node */
   int* momenta(int node)
   {
      int*q = new int[4]{ 0,0,0,0 };

      for (int i = 0; i < 24; ++i)
      {
         /* if the cell C[i] is occupied by particle, count its momentum */
         if (C[i] & node)
         {
            for (int j = 0; j < 4; ++j)
            {
               q[j] += c[i][j];
            }
         }
      }
      return q;
   }
```

```
/* We need to remember all the isometries that normalize the momentum,
   so we can transform it back to its original momentum */
/* We define this array so that we can reffer to the isometry by the
   single index */
int(*iso[])(int n, int*q, int i, int j) = { S,P,sigma };


/* This function does all isometries that normalized the node, but in
   the reverse order, so that original momentum is achieved */
void goBack(int**steps, int* nodes, int*q, int step, int length)
{
   for (int i = 1; i < length; i++)
   {
      for (int j = step - 1; j >= 0; --j)
      {
         nodes[i] = iso[steps[j][0]](nodes[i], q, steps[j][1],
            steps[j][2]);
      }
   }
}

/* Creates the entry for the state 'n' that we will insert into the
   table of collisions */
/* It is implementation of the Henon algorithm, that computes the set
   of optimal isometries */
int* newNode(int n, int**steps)
{
   int*q = momenta(n);

   int step = 0;

   // invert negative q[i]
   for (int i = 0; i < 4; i++)
   {
      if (q[i] < 0)
      {
         //S changes sign of q[i];
         n = S(n, q, i+1, 0);
         //we record each step to the array "steps" so we can go back
         steps[step][0] = 0;
         steps[step][1] = i+1;
         steps[step][2] = 0;
         ++step;
      }
   }
   // sort q[i] from high to low
   int highIndex;
   int highValue;
   for (int i = 0; i < 4; ++i)
   {
```

```
      highIndex = i;
      highValue = q[i];
      for (int j = i+1; j < 4; ++j)
      {
         if (q[j] > highValue)
         {
            highValue = q[j];
            highIndex = j;
         }
      }
      if (highIndex > i)
      {
         n = P(n, q, i + 1, highIndex + 1);
         steps[step][0] = 1;
         steps[step][1] = i + 1;
         steps[step][2] = highIndex + 1;
         ++step;
      }
   }
   // to fulfill second condition:
   if (q[3] > 0)
   {
      if (q[0] + q[3] == q[1] + q[2])
      {
         n = sigma(n, q, 2, 0);
         steps[step][0] = 2;
         steps[step][1] = 2;
         steps[step][2] = 0;
         ++step;
      }
      else if (q[0] + q[3] > q[1] + q[2])
      {
         n = sigma(n, q, 1, 0);
         steps[step][0] = 2;
         steps[step][1] = 1;
         steps[step][2] = 0;
         ++step;
      }
   }
   if (q[3] < 0)
   {
      n = S(n, q, 4, 0);
      //we record each step to the steps field, since we will go back
      steps[step][0] = 0;
      steps[step][1] = 4;
      steps[step][2] = 0;
      ++step;
   }


   int* nodes;
```

```
//class 12
if (q[0] == 0)
{
   int length = 13;
   nodes = new int[length];
   nodes[0] = length - 1;
   //S3 S1 P34 P12, S4 S1 P34 P12, S3 S2 P34 P12, S4 S2 P34 P12,
   //S2 S1 P24 P13, S4 S1 P24 P13, S3 S2 P24 P13, S4 S3 P24 P13,
   //S2 S1 P23 P14, S3 S1 P23 P14, S4 S2 P23 P14, S4 S3 P23 P14

   nodes[1] = P(n, q, 1, 2);
   nodes[1] = P(nodes[1], q, 3, 4);
   nodes[1] = S(nodes[1], q, 1, 0);
   nodes[1] = S(nodes[1], q, 3, 0);

   nodes[2] = P(n, q, 1, 2);
   nodes[2] = P(nodes[2], q, 3, 4);
   nodes[2] = S(nodes[2], q, 1, 0);
   nodes[2] = S(nodes[2], q, 4, 0);

   nodes[3] = P(n, q, 1, 2);
   nodes[3] = P(nodes[3], q, 3, 4);
   nodes[3] = S(nodes[3], q, 2, 0);
   nodes[3] = S(nodes[3], q, 3, 0);

   nodes[4] = P(n, q, 1, 2);
   nodes[4] = P(nodes[4], q, 3, 4);
   nodes[4] = S(nodes[4], q, 2, 0);
   nodes[4] = S(nodes[4], q, 4, 0);

   nodes[5] = P(n, q, 1, 3);
   nodes[5] = P(nodes[5], q, 2, 4);
   nodes[5] = S(nodes[5], q, 1, 0);
   nodes[5] = S(nodes[5], q, 2, 0);

   nodes[6] = P(n, q, 1, 3);
   nodes[6] = P(nodes[6], q, 2, 4);
   nodes[6] = S(nodes[6], q, 1, 0);
   nodes[6] = S(nodes[6], q, 4, 0);

   nodes[7] = P(n, q, 1, 3);
   nodes[7] = P(nodes[7], q, 2, 4);
   nodes[7] = S(nodes[7], q, 2, 0);
   nodes[7] = S(nodes[7], q, 3, 0);

   nodes[8] = P(n, q, 1, 3);
   nodes[8] = P(nodes[8], q, 2, 4);
   nodes[8] = S(nodes[8], q, 3, 0);
   nodes[8] = S(nodes[8], q, 4, 0);

   nodes[9] = P(n, q, 1, 4);
```

```
    nodes[9] = P(nodes[9], q, 2, 3);
    nodes[9] = S(nodes[9], q, 1, 0);
    nodes[9] = S(nodes[9], q, 2, 0);

    nodes[10] = P(n, q, 1, 4);
    nodes[10] = P(nodes[10], q, 2, 3);
    nodes[10] = S(nodes[10], q, 1, 0);
    nodes[10] = S(nodes[10], q, 3, 0);

    nodes[11] = P(n, q, 1, 4);
    nodes[11] = P(nodes[11], q, 2, 3);
    nodes[11] = S(nodes[11], q, 2, 0);
    nodes[11] = S(nodes[11], q, 4, 0);

    nodes[12] = P(n, q, 1, 4);
    nodes[12] = P(nodes[12], q, 2, 3);
    nodes[12] = S(nodes[12], q, 3, 0);
    nodes[12] = S(nodes[12], q, 4, 0);

    goBack(steps, nodes, q, step, length);

}
//class 11
else if (q[1] == 0)
{
    int length = 7;
    nodes = new int[length];
    nodes[0] = length - 1;
    //S4 S2 P23,  S4 S3 P23,  S3 S2 P24,
    //S4 S3 P24,  S3 S2 P34,  S4 S2 P34

    nodes[1] = P(n, q, 2, 3);
    nodes[1] = S(nodes[1], q, 2, 0);
    nodes[1] = S(nodes[1], q, 4, 0);

    nodes[2] = P(n, q, 2, 3);
    nodes[2] = S(nodes[2], q, 3, 0);
    nodes[2] = S(nodes[2], q, 4, 0);

    nodes[3] = P(n, q, 2, 4);
    nodes[3] = S(nodes[3], q, 2, 0);
    nodes[3] = S(nodes[3], q, 3, 0);

    nodes[4] = P(n, q, 2, 4);
    nodes[4] = S(nodes[4], q, 3, 0);
    nodes[4] = S(nodes[4], q, 4, 0);

    nodes[5] = P(n, q, 3, 4);
    nodes[5] = S(nodes[5], q, 2, 0);
    nodes[5] = S(nodes[5], q, 3, 0);
```

```
    nodes[6] = P(n, q, 3, 4);
    nodes[6] = S(nodes[6], q, 2, 0);
    nodes[6] = S(nodes[6], q, 4, 0);


    goBack(steps, nodes, q, step, length);

}
//class 10
else if (q[2] == 0 && q[0]==q[1])
{
    int length = 7;
    nodes = new int[length];
    nodes[0] = length - 1;
    //S3,P34,P12, S4,P34,P12, S4,S3,sigma1, S4,S3,P34,P12,sigma1
    //S4,S3,sigma2, P34,P12,sigma2
    nodes[1] = P(n, q, 1, 2);
    nodes[1] = P(nodes[1], q, 3, 4);
    nodes[1] = S(nodes[1], q, 3, 0);

    nodes[2] = P(n, q, 1, 2);
    nodes[2] = P(nodes[2], q, 3, 4);
    nodes[2] = S(nodes[2], q, 4, 0);

    nodes[3] = sigma(n, q, 1, 0);
    nodes[3] = S(nodes[3], q, 3, 0);
    nodes[3] = S(nodes[3], q, 4, 0);

    nodes[4] = sigma(n, q, 1, 0);
    nodes[4] = P(nodes[4], q, 1, 2);
    nodes[4] = P(nodes[4], q, 3, 4);
    nodes[4] = S(nodes[4], q, 3, 0);
    nodes[4] = S(nodes[4], q, 4, 0);

    nodes[5] = sigma(n, q, 2, 0);
    nodes[5] = S(nodes[5], q, 3, 0);
    nodes[5] = S(nodes[5], q, 4, 0);

    nodes[6] = sigma(n, q, 2, 0);
    nodes[6] = P(nodes[6], q, 1, 2);
    nodes[6] = P(nodes[6], q, 3, 4);

    goBack(steps, nodes, q, step, length);
}
//class 9
else if (q[2] == 0 && q[0] > q[1])
{
    int length = 4;
    nodes = new int[length];
    nodes[0] = length - 1;
```

```
   nodes[1] = S(n, q, 3, 0);
   nodes[1] = S(nodes[1], q, 4, 0);

   nodes[2] = P(n, q, 3, 4);
   nodes[2] = S(nodes[2], q, 3, 0);

   nodes[3] = P(n, q, 3, 4);
   nodes[3] = S(nodes[3], q, 4, 0);

   goBack(steps, nodes, q, step, length);
}
//class 8
else if (q[0]==q[1] && q[1] == q[2] && q[2] > q[3] && q[3]==0)
{
   int length = 5;
   nodes = new int[length];
   nodes[0] = length - 1;
   //1
   nodes[1] = P(n, q, 1, 2);
   nodes[1] = P(nodes[1], q, 2, 3);
   //2
   nodes[2] = P(n, q, 1, 3);
   nodes[2] = P(nodes[2], q, 2, 3);
   //3
   nodes[3] = P(n, q, 1, 2);
   nodes[3] = P(nodes[3], q, 2, 3);
   nodes[3] = S(nodes[3], q, 4, 0);

   //4
   nodes[4] = P(n, q, 1, 3);
   nodes[4] = P(nodes[4], q, 2, 3);
   nodes[4] = S(nodes[4], q, 4, 0);

   goBack(steps, nodes, q, step, length);
}
//class 6 or 7
else if (q[0]>q[1] && q[1]==q[2] && q[2] > q[3] && q[3] == 0)
{
   //class 6
   if (q[0] == 2*q[1])
   {
      int length = 5;
      nodes = new int[length];

      nodes[0] = length - 1;

      nodes[1] = sigma(n, q, 1, 0);
      nodes[1] = S(nodes[1], q, 4, 0);

      nodes[2] = sigma(n, q, 2, 0);
      nodes[2] = S(nodes[2], q, 4, 0);
```

```
            nodes[3] = sigma(n, q, 1, 0);
            nodes[3] = P(nodes[3], q, 2, 3);
            nodes[3] = S(nodes[3], q, 4, 0);

            nodes[4] = sigma(n, q, 2, 0);
            nodes[4] = P(nodes[4], q, 2, 3);
            nodes[4] = S(nodes[4], q, 4, 0);

            goBack(steps, nodes, q, step, length);

        }
        //class 7
        else
        {
            int length = 2;
            nodes = new int[length];

            nodes[0] = length - 1;

            nodes[1] = P(n, q, 2, 3);
            nodes[1] = S(nodes[1], q, 4, 0);

            goBack(steps, nodes, q, step, length);
        }

    }
    //class 5
    else if (q[0]==q[1] && q[1]>q[2] && q[2] > q[3] && q[4] == 0)
    {
        int length = 2;
        nodes = new int[length];

        nodes[0] = length - 1;
        //1
        nodes[1] = P(n, q, 1, 2);
        nodes[1] = S(nodes[1], q, 4, 0);

        goBack(steps, nodes, q, step, length);
    }
    //class 3,4
    else if (q[0]>q[1] && q[1]>q[2] && q[2] > q[3] && q[3] == 0)
    {
        //class 3
        if (q[0]==q[1]+q[2])
        {
            int length = 3;
            nodes = new int[length];

            nodes[0] = length - 1;
```

```
      nodes[1] = sigma(n, q, 1, 0);
      nodes[1] = S(nodes[1], q, 4, 0);

      nodes[2] = sigma(n, q, 2, 0);
      nodes[2] = S(nodes[2], q, 4, 0);

      goBack(steps, nodes, q, step, length);

   }
   //class 4
   else
   {
      int length = 2;
      nodes = new int[length];

      nodes[0] = length - 1;

      nodes[1] = S(n, q, 4, 0);

      goBack(steps, nodes, q, step, length);
   }
}
//class 2
else if (q[0]==q[1] && q[1]==q[2] && q[2]>q[3] && q[3]>0)
{
   int length = 3;
   nodes = new int[length];

   nodes[0] = length - 1;

   nodes[1] = P(n, q, 1, 2);
   nodes[1] = P(nodes[1], q, 2, 3);

   nodes[2] = P(n, q, 1, 3);
   nodes[2] = P(nodes[2], q, 2, 3);

   goBack(steps, nodes, q, step, length);
}
//class 1
else if (q[0]==q[1] && q[1]>q[2] && q[2] > q[3] && q[3] > 0)
{
   int length = 2;
   nodes = new int[length];

   nodes[0] = length - 1;

   nodes[1] = P(n, q, 1, 2);

   goBack(steps, nodes, q, step, length);
}
else
```

```
      nodes = nullptr;
   return nodes;
}

/*  This is the high-level function that creates the table of
    isometries */
/* For all 2^24 possible states of the node, it calls the function
    newNode and save the set of new states */
void fillTable(int** table)
{
   int n;

   int ** steps = new int*[20];
   for(int i = 0; i<20; ++i)
      steps[i] = new int[3];

   for ( n = 0; n < OBS; ++n)
      table[n] = newNode(n, steps);
}
```

12.2 Algorithm for collision

Once we have the table specifying optimal isometrical states ready (by the function *fillTable* above), the algorithm for collision is to implement. The only non-trivial task is to achieve uniform distribution of chosen isometries without generation of the random numbers, as it is expensive on CPU time.

```
/* For all nodes 'n' on the grid, this function looks in the table and
    choose from optimal final states */

void Collision(int***grid, int**table, int t, int X, int Y, int Z)
{
   int x,y,z;

   int n;
   int k;
#pragma omp parallel for private (x,y,z,n,k)
   for (x = 0; x < X; ++x)
      for (y = 0; y < Y; ++y)
         for (z = 0; z < Z; ++z)
         {
            //We find that node with position [x][y][y] on the grid is
                in the state n
            n = grid[x][y][z];
            // If there is obstacle in the node, we skip it.
            if (n & OBS)
               break;
            /* k is initialized by the external parameter t,
                and k is moduled by number of optimal isometries for the
                   state n */
```

```
            k = (t % table[n][0]) + 1;

            // we assign the k-th optimal state to the node
            grid[x][y][z] = table[n][k];

            //every step, we increment the external parameter, so we
               hopefully achieve the uniform distribution of k (there
               is huge number of nodes on the grid)
            ++t;
         }
}
```

12.3 Propagation in FCHC

Although the collisions are resolved in the four dimensional space (and the particles are the four dimensional objects), for the purpose of propagation, we can simply 'forget' its four-dimensional nature.

The nodes and particles are sitting on the three dimensional grid and propagate along the 3D projection of lattice vectors 6.1.

```
/* The propagation is happening in all the nodes at once, but in the
   computer simulation, it is happening sequentially, so we need to
   use to grids (propagation happening on the one grid would overwrite
   some nodes by new particles before the old particles propagated
   away). */
/* Hence, we are using the two grids, 'int***even' and 'int***odd'.
   'Even' grid is occupied at even times, 'Odd' grid is occupied at
   odd times. For example, in odd time, first parameter of Propagation
   (int***from) will be 'Odd' and second parameter (int***to) will be
   'Even' */

void Propagation(int***from, int***to, int**table, int X, int Y, int Z)
{
   int x,y,z;
   int i;
   int n;
   int new_x, new_y, new_z;

#pragma omp parallel for private (n, new_x, new_y, new_z, x, y, z, i)
   for ( x = 0; x < X; ++x)
      for ( y = 0; y < Y; ++y)
         for ( z = 0; z < Z; ++z)
         {
            // The current state of the node at [x][y][z]
            n = from[x][y][z];
            // We check every cell
            for ( i = 0; i < 24; ++i)
            {
               // If the cell C[i] is occupied by particle, we propagate
                  it to the corresponding node.
```

```
            if (n & C[i])
            {
               new_x = PeriodicBC(x + c[i][0], X);
               new_y = PeriodicBC(y + c[i][1], Y);
               new_z = PeriodicBC(z + c[i][2], Z);
               // If there is obstacle in the node, particle's
                  velocity is reversed.
               // e.g. particle with momentum [1,0,-1,0] gains
                  momentum [-1,0,1,0] by the reflection from the
                  obstacle
               if (to[new_x][new_y][new_z] & OBS)
                  to[new_x][new_y][new_z] |= Reverse[i];
               else
                  to[new_x][new_y][new_z] |= C[i];
            }
         }
         from[x][y][z] = 0;
      }
}
```

13. Implementation of the Pair-Interaction LGCA in 3D

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%." Donald Knuth in "Structured programming with GoTo statement".

For sure, the implementation of the collision algorithm is in the critical 3%, as it is repeated more then 10^8 times per one update of the lattice.

With some knowledge of C compiler, we could do better, but for our own applications, this should be enough.

13.1 Implementation of the collision algorithm

```
// The node consists of 8 cells represented by single bits on the
    first, second, ... and eighth position.
#define A 1
#define B 2
#define C 4
#define D 8
#define E 16
#define F 32
#define G 64
#define H 128

/* To iterate over the cells, we arrange it in the char */
unsigned char cell[8] = {A, B, C, D, E, F, G, H};

/* The definition of the type Node. It consists of 8 mass bits (char
    m) and 3x8 momentum bits (char p[3]). It the node is part of the
    obstacle, node.o == 1. */
typedef struct
{
   unsigned char m;
   unsigned char p[3];
   int o;
} Node;
// Sure. char would be enough for and obstacle, but does not 64 bit
    processor prefer to manage 64 bit structures (m + p[3] + o = 64
    bits)?

// 3 directions (X, Y, Z), 4 pair-interactions in each direction, 2
    cells in each pair
unsigned char Pair[3][4][2] =
```

```
{
   {
      {A,E}, {B,F}, {C,G}, {D,H}
   },
   {
      {A,C}, {B,D}, {E,G}, {F,H}
   },
   {
      {A,B}, {C,D}, {E,F}, {G,H}
   }
};

// Finally we have have everything prepared for the collision
    algorithm.
Node collision(Node node)
{
   //mass
   unsigned char m = node.m;

   //momentum
    unsigned char * p = node.p;

   //d,u ... index for downer and upper momentum
   int d,u;

   // l, r ... left/right cell in the pair
   // ml, mr are non-zero, if there is a particle in the left/right
       cell of the pair (mass-left, mass-right)
   // lu, ld, ru, rd ... momenta of the particles (left-upper,
       left-downer, right-upper, right-downer)
   unsigned char l, r, ml, mr, lu, ld, ru, rd;

   for (int i = 0; i < 3; ++i)
   {
      d = (i+1) % 3;
      u = (i+2) % 3;
      for (int j = 0; j < 4; ++j)
      {
         l  = Pair[i][j][0];
         r  = Pair[i][j][1];

         ml = m&l;
         mr = m&r;
         ld = p[d]&l;
         lu = p[u]&l;
         rd = p[d]&r;
         ru = p[u]&r;

         /* PAIR INTERACTIONS */
         // case 6a
         if (ml && mr)
```

```
{
   if (lu && ru && !ld && !rd)
   {
      p[d] |= l;
      p[d] |= r;
   }
   //case 6b
   else if (lu && ru && ld && rd)
   {
      p[d] ^= l;
      p[d] ^= r;
   }
   //case 7a
   else if (ld && !lu && ru && rd)
   {
      p[d] ^= l;
      p[u] |= l;
      p[d] ^= r;
      p[u] ^= r;
   }
   else if (ld && lu && !ru && rd)
   {
      p[d] ^= l;
      p[d] ^= r;
      p[u] ^= l;
      p[u] |= r;
   }
   //case 7b
   else if (!ld && lu && !ru && !rd)
   {
      p[d] |= l;
      p[d] |= r;
      p[u] ^= l;
      p[u] |= r;
   }
   else if (!ld && !lu && ru && !rd)
   {
      p[d] |= l;
      p[d] |= r;
      p[u] |= l;
      p[u] ^= r;
   }
   //case 8a
   else if (ld && !lu && ru && !rd)
   {
      p[u] |= l;
      p[u] ^= r;
   }
   else if (!ld && lu && !ru && rd)
   {
      p[u] |= r;
```

```
            p[u] ^= l;
        }
        //case 8b
        else if (ld && lu && !ru && !rd)
        {
            p[u] |= l;
            p[u] ^= r;
        }
        else if (!ld && !lu && ru && rd)
        {
            p[u] |= r;
            p[u] ^= l;
        }
        //case 9a
        else if (!ld && !lu && !ru && !rd)
        {
            p[d] |= l;
            p[d] |= r;
        }
        //case 9b
        else if (ld && rd && !lu && !ru)
        {
            p[d] ^= l;
            p[d] ^= r;
        }
    }
    //case 10a
    else if (!ml && mr && ru && !rd)
    {
        m |= l;
        m ^= r;
        p[u] ^= r;
        p[u] |= l;
        if (p[i]&r)
        {
            p[i] ^= r;
            p[i] |= l;
        }
    }
    //case 10b
    else if (ml && !mr && lu && !ld)
    {
        m ^= l;
        m |= r;
        p[u] ^= l;
        p[u] |= r;
        if (p[i]&l)
        {
            p[i] ^= l;
            p[i] |= r;
        }
```

```
            }
            //case 11a
            else if (!ml && mr && !ru && !rd)
            {
               m |= 1;
               m ^= r;
               if (p[i]&r)
               {
                  p[i] ^= r;
                  p[i] |= 1;
               }
            }
            //case 11b
            else if (ml && !mr && !lu && !ld)
            {
               m |= r;
               m ^= 1;
               if (p[i]&1)
               {
                  p[i] ^= 1;
                  p[i] |= r;
               }
            }
         }
      }
      node.m = m;
      for(int i = 0; i < 3; ++i)
         node.p[i] = p[i];
      return node;
   }

   // This function resolves the collisions for all nodes on the lattice.
   // This is the function we call from main().
   void Collision(Node***array, int X, int Y, int Z, int start)
   {
      int x,y,z;

   #pragma omp parallel for private (x,y,z)
      for (x = start; x < X; x+=2)
      {
         for(y = start; y < Y; y+=2)
         {
            for(z = start; z < Z; z+=2)
            {
               if(array[x][y][z].o)
                  break;
               array[x][y][z] = collision(array[x][y][z]);
            }
         }
      }
   }
```

13.2 Implementation of the Propagation

Before we comment on the implementation, we want to emphasize one of the strong sides of PI automaton.

In FCHC, the lattice consisted of all the nodes with the discrete Cartesian coordinates (up to some upper boundary denoted by X, Y, Z). All of these nodes were interconnected and formed single lattice.

But the same lattice for Pair-Interaction automaton would contain 4 independent sub-lattices.

Let us consider node that has all Cartesian indices odd. It is connected to 8 nodes that have even Cartesian indices (because lattice vectors for PI are exclusively ± 1). These nodes create one of the four sub-lattices.

Another thing is, that we can avoid spurious Zannetti invariants if, for example, nodes with even indices are occupied at even time steps, and odd nodes at odd times. (for more on Zanneti invariants, see page blabla in [1]).

This gives us additional speed-up in computation, as we do not need two distinct lattices for FCHC (or FHP).

```
/* Group of cells that propagates in X-direction, Y-direction and
   Z-direction respectively. */
#define dirX (B+D+F+H)
#define dirY (E+F+G+H)
#define dirZ (A+B+E+F)
/* For example, because A & dirX == 0, particle in the cell A
   propagates in -X direction,
   but A & dirZ == 1, so it propagates in (positive) Z direction. */

// Implements periodic boundary condition.
// If index n flows under 0 or over upper boundary N, we assign it
   value from the other side of boundary.
int PeriodicBC(int n, int N)
{
   if (n<0) return N-1;
   else if (n<N) return n;
   // else n==N
   else return 0;
}

/* This function propagates particles among the nodes. */
/* If start == 0, we are at the even time step. Then it propagates
   from nodes with even indices (e.g. at position [0,2,124] on the
   lattice) to the nodes with odd indices (e.g. to the [1,1,123] etc).
   */
/* If start == 1, we are at the odd time step. Then it propagates from
   odd indices to the even indices. */

void Propagation(Node***array, int X, int Y, int Z, int start)
{
   Node node;
   unsigned char m;
   unsigned char*p;
```

```
    int x, y, z;
   int xN, yN, zN;
   int c, i;

#pragma omp parallel for private (m, p, x, y, z, xN, yN, zN, c, i)
   // Depending on start, we go through odd or even nodes
    for(x = start; x < X; x+=2)
   {
      for(y = start; y < Y; y+=2)
      {
         for(z = start; z < Z; z+=2)
         {
            // mass of the cells in the node (tells us which cells are
                occupied)
            m = array[x][y][z].m;
            // momentum of the particles in the node
            p = array[x][y][z].p;

            // we look at each cell in node
            for (c = 0; c < 8; ++c)
            {
               //if there is particle in the cell, we want to propagate
                   it to corresponding node
               if(m & cell[c])
               {
                  // if particle from cell[c] propagates in positive
                      direction of X
                  if (cell[c] & dirX)
                     xN = PeriodicBC(x+1,X);
                  // else particle propagates in negative direction of x
                  else
                     xN = PeriodicBC(x-1,X);
                  if (cell[c] & dirY)
                     yN = PeriodicBC(y+1,Y);
                  else
                     yN = PeriodicBC(y-1,Y);
                  if (cell[c] & dirZ)
                     zN = PeriodicBC(z+1,Z);
                  else
                     zN = PeriodicBC(z-1,Z);

                  // if there is obstacle in the new node,
                  // it stays in the old node, but we reflect it to
                      diagonal cell
                  if( array[xN][yN][zN].o )
                  {
                     array[xN][yN][zN].m |= cell[7-c];
                     for (i = 0; i < 3; ++i)
                     {
                        if(p[i] & cell[c])
                           array[xN][yN][zN].p[i] |= cell[7-c];
```

```
                    }
                    /*particles disappear on 0 and Z-1 */
                    // this holds in this particular computation

                    if(zN == 0 || zN == Z-1)
                    {
                       array[xN][yN][zN].m = 0;
                       for (int i = 0; i < 3; ++i)
                       {
                          array[xN][yN][zN].p[i] = 0;
                       }
                    }
                 }
                 else //if there is no obstacle in the new node, we
                     propagate particle to the new node
                 {
                    array[xN][yN][zN].m |= cell[c];
                    for(i = 0; i < 3; ++i)
                    {
                       if(p[i] & cell[c])
                          array[xN][yN][zN].p[i] |= cell[c];
                    }
                    /*particles disappear on the 0 and X-1 */

                    if (zN == 0 || zN == Z-1)
                    {
                       array[xN][yN][zN].m = 0;
                       for (i = 0; i < 3; ++i)
                          array[xN][yN][zN].p[i] = 0;
                    }
                 }
              }
           }
           //in the old array, we set the node to 0
           array[x][y][z].m = 0;
           for(i = 0; i < 3; ++i)
           {
              array[x][y][z].p[i] = 0;
           }
        }
     }
  }
}
```

---

h

14. Non-deterministic PI

Pair-Interaction that Nasilowski proposed is deterministic, and it might be considered to be its advantage. The collision (usually) leads to the maximal change of the state, that minimizes the viscosity (as we previously discussed). Moreover, it offers theoretical ground for using Gibbs distribution in derivation of macroscopic equations.

But let us explore following examples. First, consider node with one standing particle.

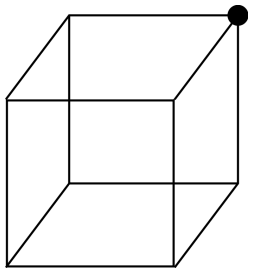


Figure 14.1: Node before collision

By deterministic pair-interactions in X,Y and Z direction, it is always resolved into the state

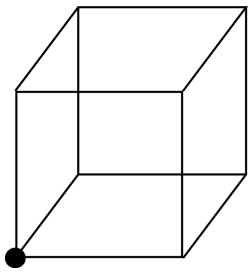


Figure 14.2: Node after deterministic collision

But it is no better then

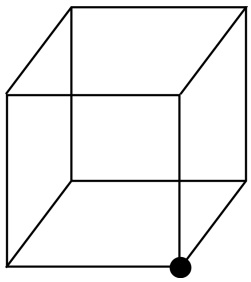


Figure 14.3: Another acceptable state after collision

or any other state with one standing particle.

Even better example is the node with two particles with momenta in X direction.

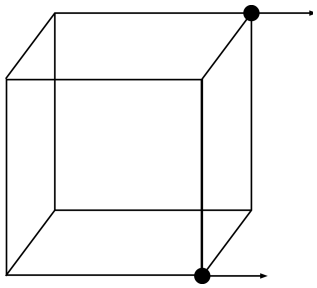


Figure 14.4: Node before collision, and after deterministic collision

The deterministic collision do not change its state (pair interaction in Y direction, followed by pair interaction in Z direction leads to the same state for this particular example).

However, collision that would be non-deterministic (let's say that pair interaction happens with probability 1/2) can lead to the following state

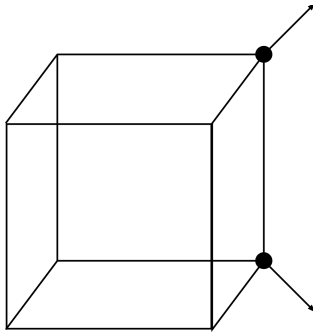


Figure 14.5: Node after non-deterministic collision

where both particles gained additional momenta, one in Z, and other in -Z direction. We could continue with the examples showing that deterministic PI do not realize all available states, even though some of them are more desirable.

Also, by chosing one of the many available states by non-deterministic PI, we believe that statistical avareging might work faster and lead to better results when statistical properties of the flow are inspected.

On the other hand, non-deterministic model, has one obvious drawback. Generation of random numbers is costy process and significantly slows down the collisions (although it might be cheated, we prefer to try honest non-deterministic model, then we can optimize).

The non-deterministic automaton was achieved by simple modification of the collision algorithm.

```
void Collision(Node***array, int X, int Y, int Z, int start)
{
  /* Rundom numbers are generated by Mersen-Twister pseudo-random
     generator */
  random_device rd;
```

```
   mt19937 rng(rd());

   /* As there are 12 pair interactions per one collision, we want to
      generate 12 random bits with uniform distribution */
   int upper = pow(2, 12);
   uniform_int_distribution<int> uni(0,upper - 1);

   int r;
   int x,y,z;

#pragma omp parallel for private (x,y,z)
   for (x = start; x < X; x+=2)
   {
      for(y = start; y < Y; y+=2)
      {
         for(z = start; z < Z; z+=2)
         {
               r =  = uni(rng);
               array[x][y][z] = collision(array[x][y][z], r);
         }
      }
   }
}

/*  Comparing to deterministic collision function, it has second
    parameter ran */
/* first 12 bi
Node collision(Node node, int ran)
{
   //mass
   unsigned char m = node.m;

   //momentum
    unsigned char *p = node.p;

   //d,u ... index for downer and upper momentum
   int d,u;

   // l, r ... left/right cell in the pair
   // ml, mr ... particle in left/right cell is present (mass-left,
      mass-right)
   // lu, ld, ru, rd ... momenta of particles (left-upper, left-downer,
      right-upper, right-downer)
   unsigned char l, r, ml, mr, lu, ld, ru, rd;

   // For first pair-interaction, first bit is set to one, other bits
      are zero
   int count = 1;
   for (int i = 0; i < 3; ++i)
   {
      d = (i+1) % 3;
```

```
u = (i+2) % 3;
for (int j = 0; j < 4; ++j)
{
   /* If the ran has corresponding bit equal to 1, the
      pair-interaction go on, else it is skipped */
   if (!( ran &count))
   {
      count <<= 1;
      break;
   }
   count <<= 1;
   ....
   ....
   /* The rest of the function is not modified */
```

14.1 Exploding cube

To demonstrate the difference of deterministic and non-deterministic automaton in the crystal form, we simulated the "explosion of the cube".

The simulations were performed on the lattice $240 \times 240 \times 240$ nodes for deterministic PI, and for the slower non-deterministic PI, size of the domain was $120 \times 120 \times 120$. Periodic boundary conditions were used.

Into the cube with the side of the length 3 times smaller then the lattice (hence the volume of the cube was 1/27 of the lattice), we put particles into all available cells, with maximal momenta in all directions. Therefore, the total momentum of the particles is 0.

The evolution of the system for both versions of PI is to be seen on the following pictures.

The is only short excerpt of the evolution of deterministic PI, but the patterns you will se were repeating in the infinite loop. We performed simulation up to 9000 steps and the pattern did not change.

On the other hand, the pattern in the non-deterministic PI broke after first explosion already - see Figure 14.1.

Although the inertia of the bulk flow is present in the non-deterministic PI after many periods, it exhibits the realistic "spilling" of the particles, in contrast to the perfect evolution of deterministic automaton.

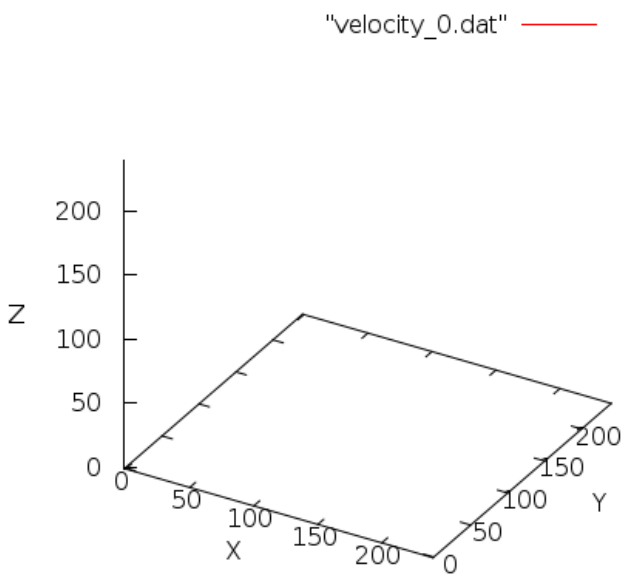
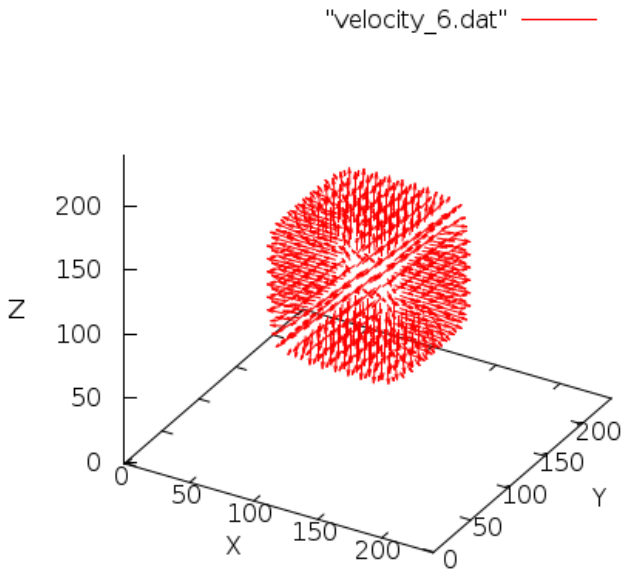


Figure 14.6: Deterministic PI - time 0



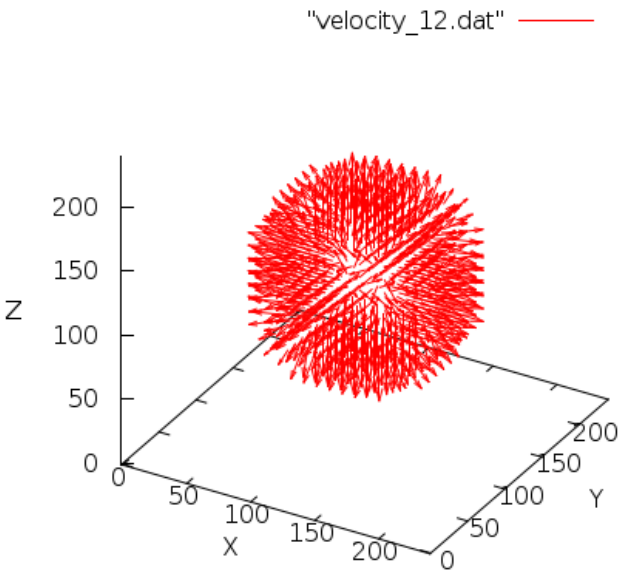


Figure 14.7: Deterministic PI - time 12

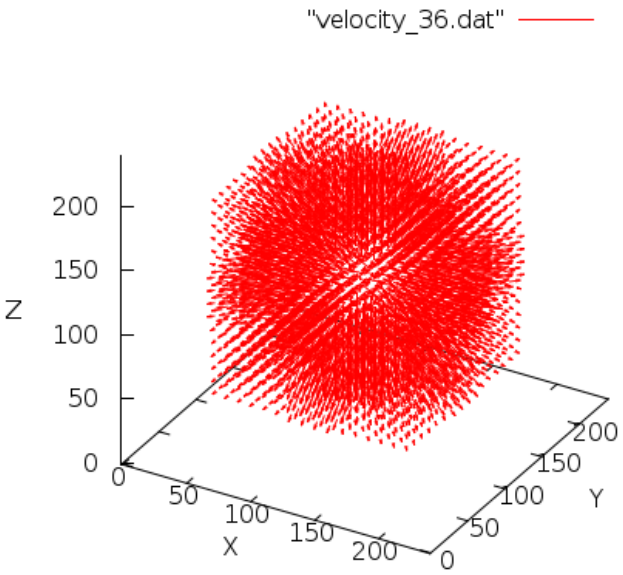


Figure 14.8: Deterministic PI - time 36

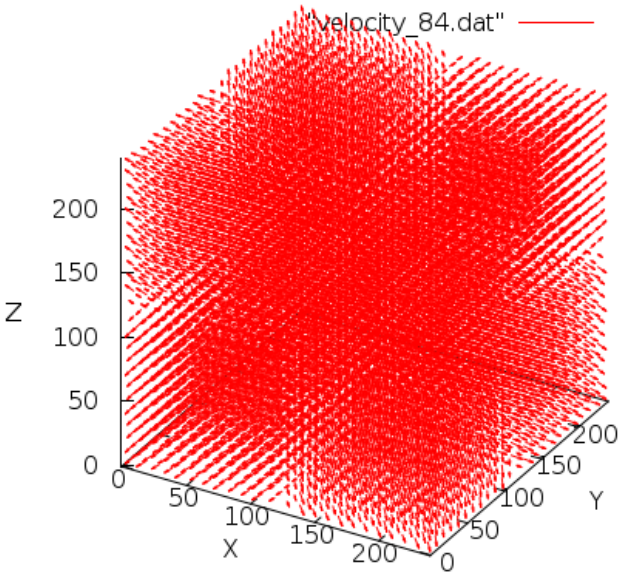


Figure 14.9: Deterministic PI - time 84

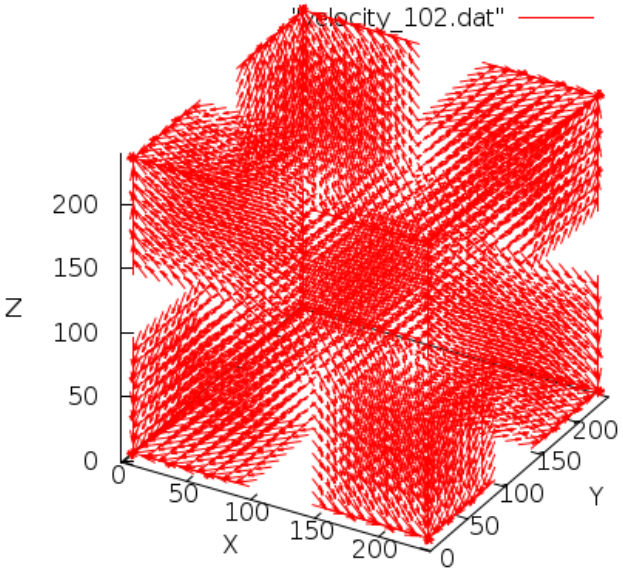


Figure 14.10: Deterministic PI - time 102

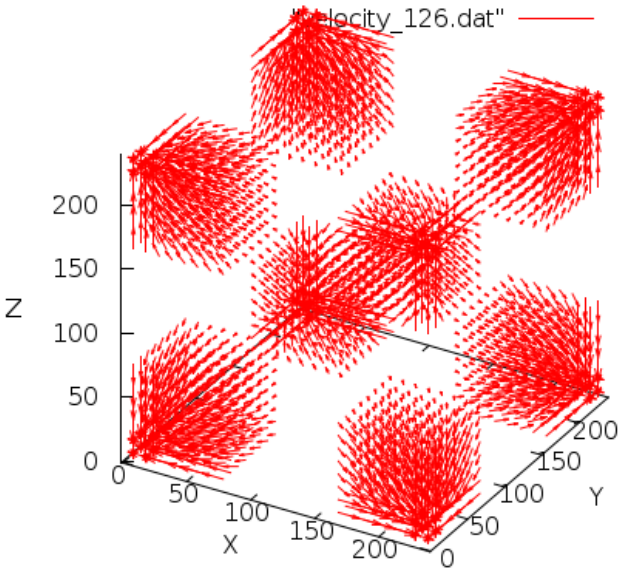


Figure 14.11: Deterministic PI - time 126

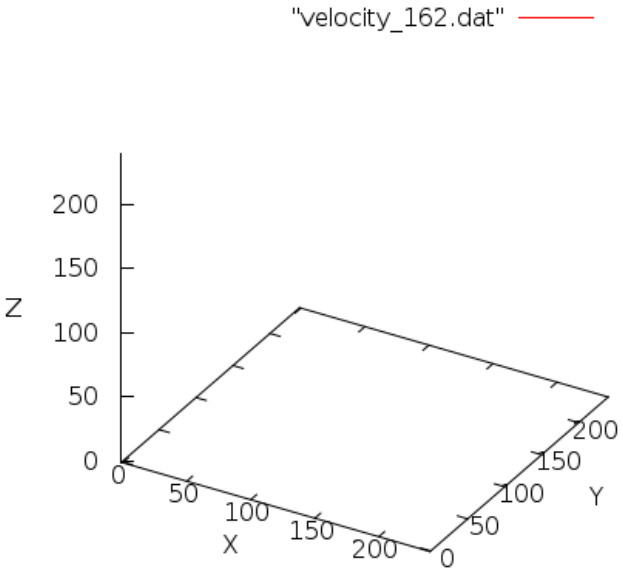


Figure 14.12: Deterministic PI - time 162

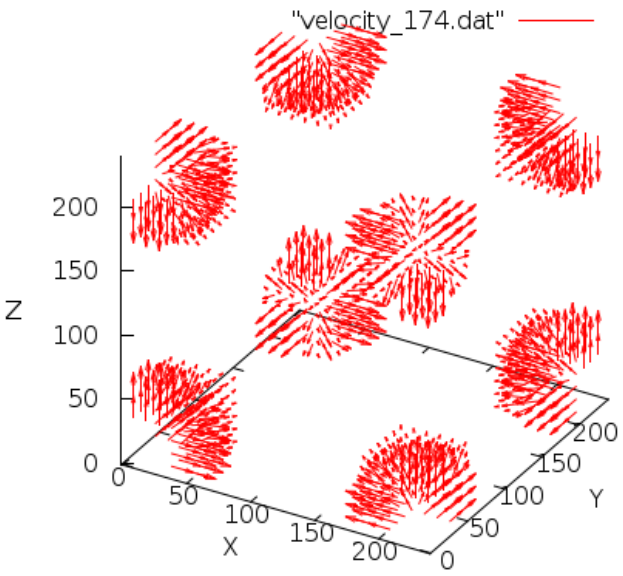


Figure 14.13: Deterministic PI - time 174

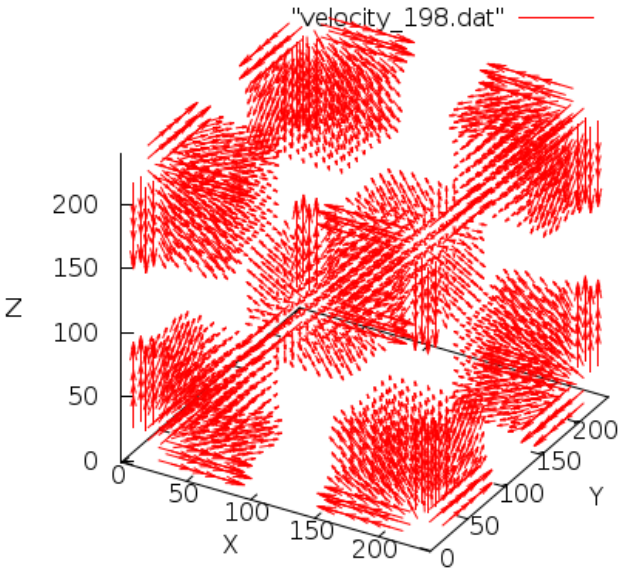


Figure 14.14: Deterministic PI - time 198

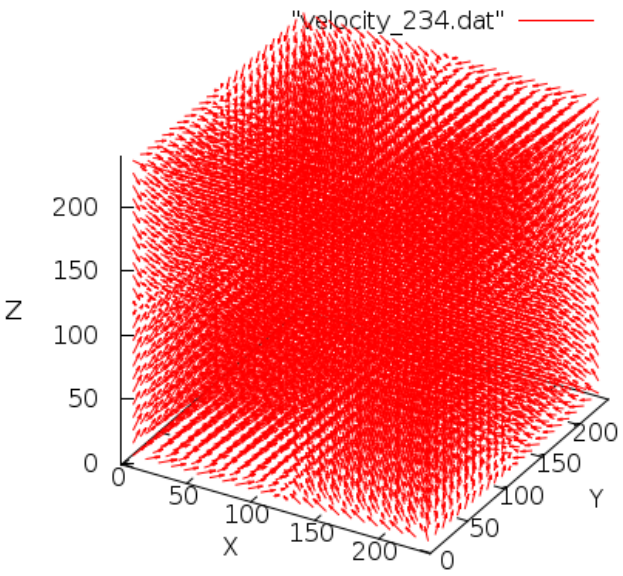


Figure 14.15: Deterministic PI - time 234

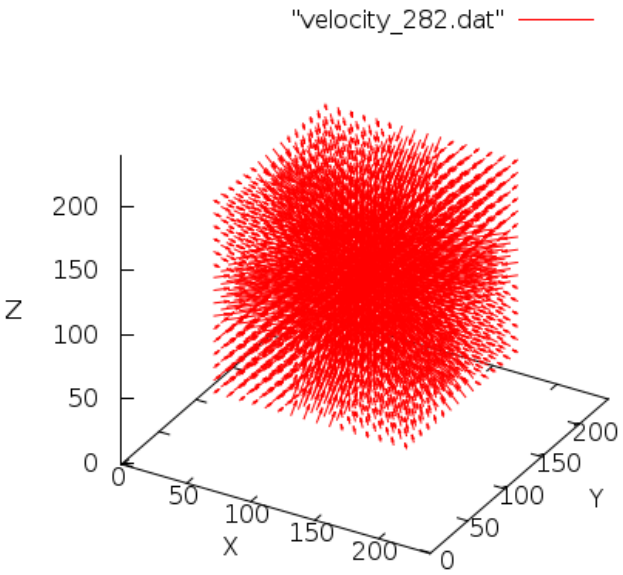


Figure 14.16: Deterministic PI - time 282

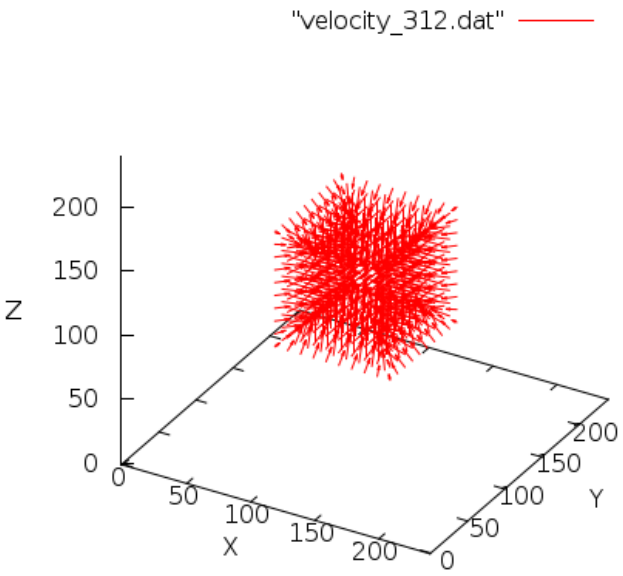


Figure 14.17: Deterministic PI - time 312

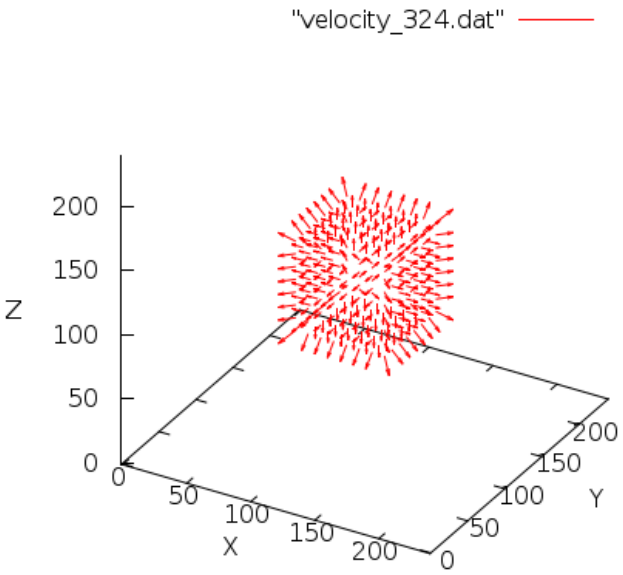


Figure 14.18: Deterministic PI - time 324

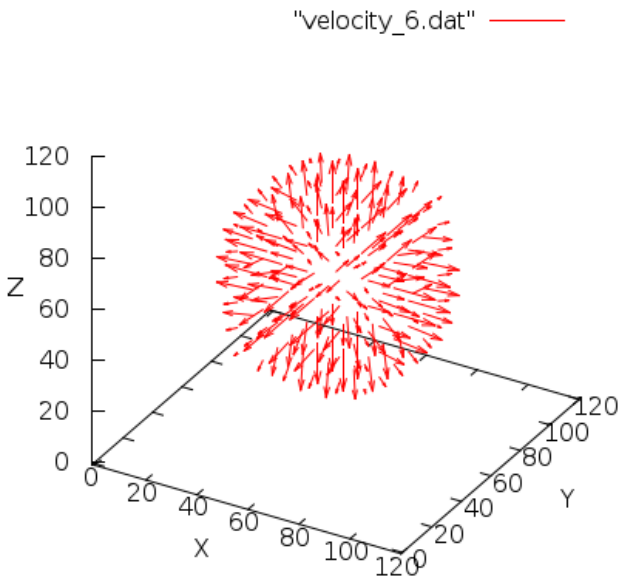


Figure 14.19: Non-deterministic PI - time 6

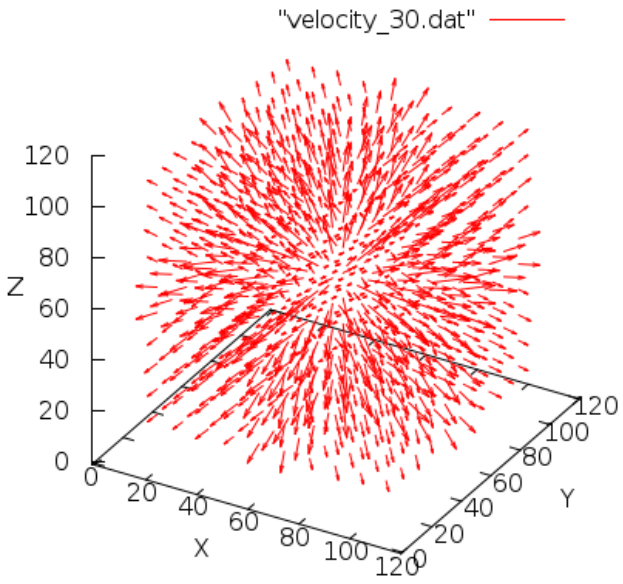


Figure 14.20: Non-deterministic PI - time 30

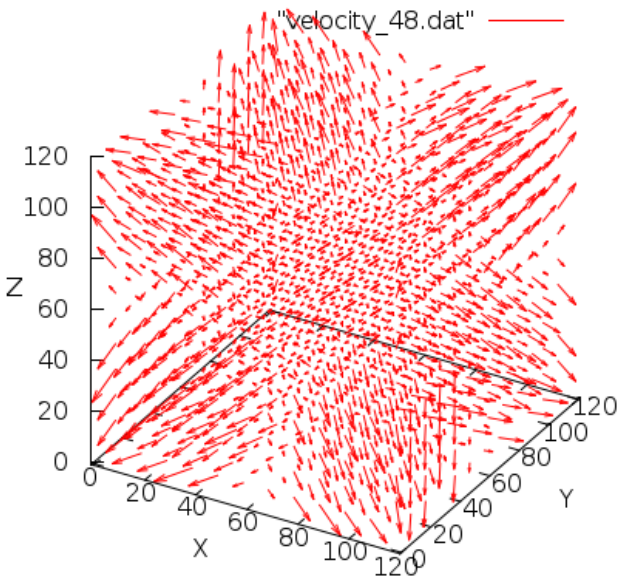


Figure 14.21: Non-deterministic PI - time 48

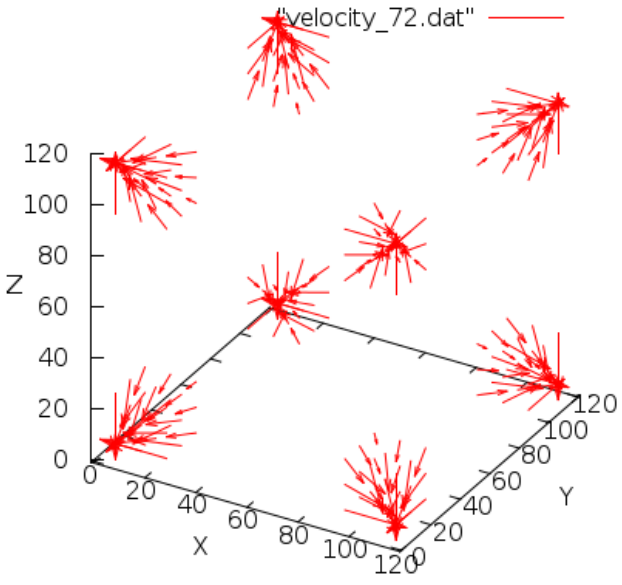


Figure 14.22: Non-deterministic PI - time 72

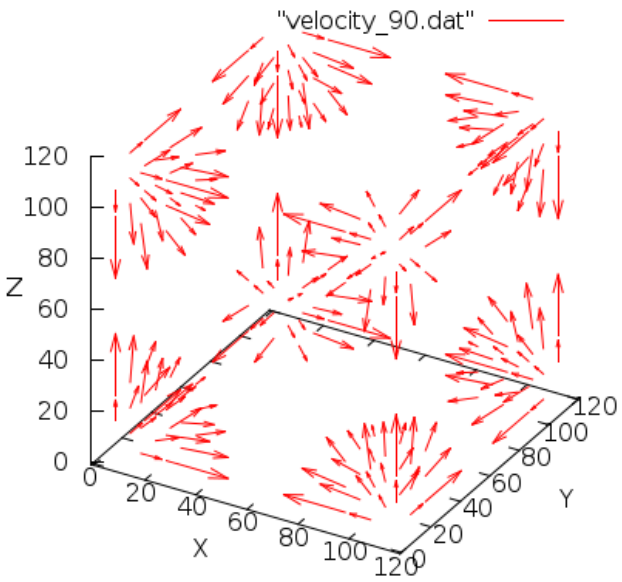


Figure 14.23: Non-deterministic PI - time 90

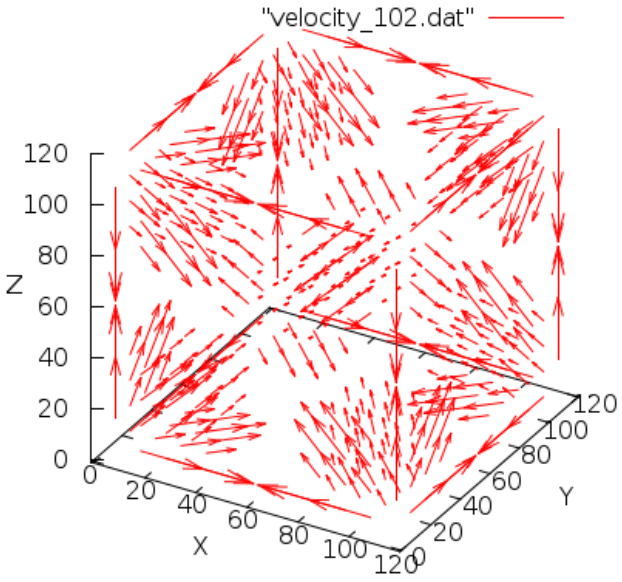


Figure 14.24: Non-deterministic PI - time 102

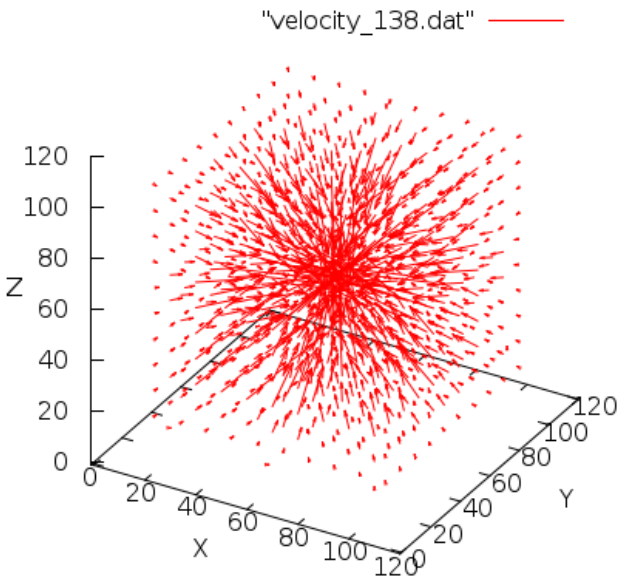


Figure 14.25: Non-deterministic PI - time 138

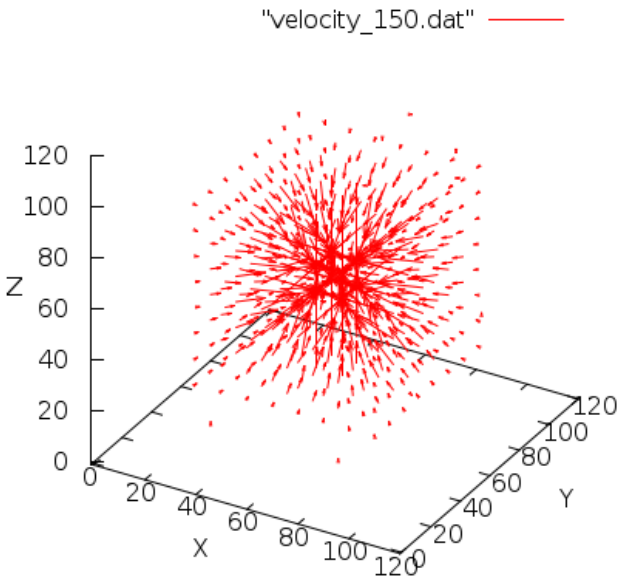


Figure 14.26: Non-deterministic PI - time 150

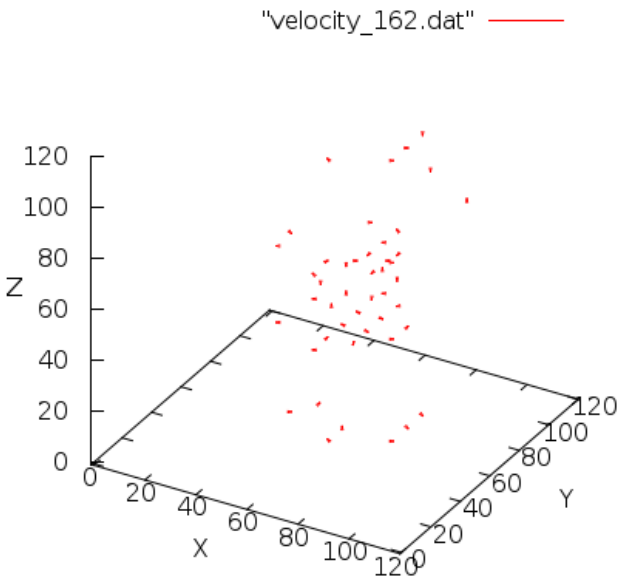


Figure 14.27: Non-deterministic PI - time 162

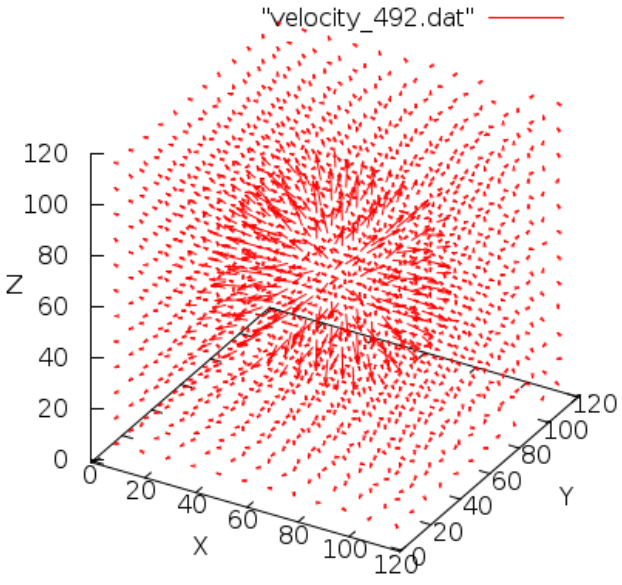


Figure 14.28: Non-deterministic PI - time 492

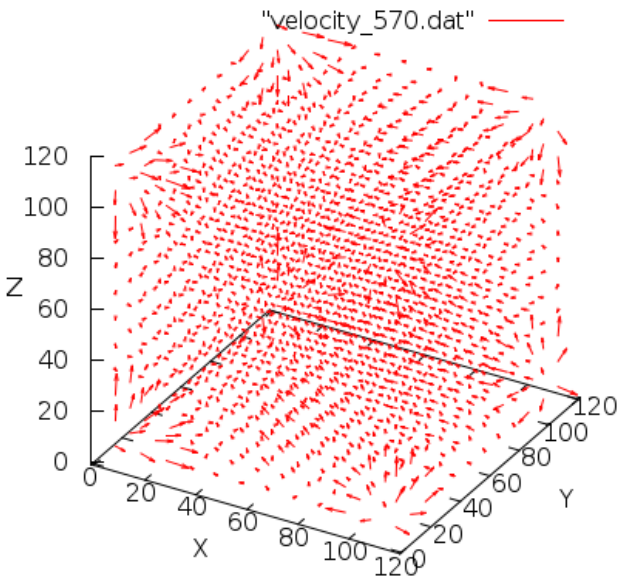


Figure 14.29: Non-deterministic PI - time 570

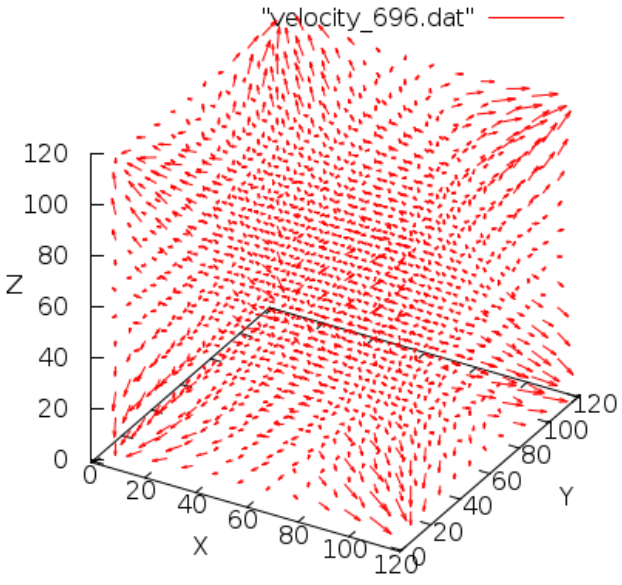


Figure 14.30: Non-deterministic PI - time 696

15. Study of the flow around the obstacles

In previous chapter, we explained microdynamics of FCHC and PI, that is non-physical. However, if we avarage velocities of the particles over sufficiently large region, we obtain physical velocity field.

The function that implements computation of velocity from LGCA lattice is simple and similar for FCHC and PI, so we comment only on FCHC implementation.

```
void compute_velocity(int***grid, double****v, int X, int Y, int Z,
    int side, int I, int J, int K)
{
   // We compute mean velocity over the cube with the side of length
      'side'.
   // N is the number nodes in this cube.
   double N = side*side*side;

   int i, j, k, l, m, n;

   int x, y, z;

#pragma omp parallel for private (i,j,k,l,m,n,x,y,z)

   for (i = 0; i < I; ++i)
   {
      for (j = 0; j < J; ++j)
      {
         for (k = 0; k < K; ++k)
         {
            for (x = i*side; x < (i + 1)*side; ++x)
            {
               for (y = j*side; y < (j + 1)*side; ++y)
               {
                  for (z = k*side; z < (k + 1)*side; ++z)
                  {
                     n = grid[x][y][z];
                     for (m = 0; m < 24; ++m)
                     {
                        if (n & C[m])
                        {
                           for (l = 0; l < 3; ++l)
                           {
                              v[i][j][k][l] += c[m][l];
                           }
                        }
                     }
                  }
```

```
            }
         }
         for (l = 0; l < 3; l++)
            v[i][j][k][l] /= N;
      }
   }
}
```

In the following section, we present the simulation of the flow around the spherical obstacle and round plate.

Particles with velocity in positive Z direction are created on a plain $z = 1$, and at the plain $z = Z - 1$ they propagate freely out of the tunnel.

The flow is actually happening in the tunnel with the square intersection. This tunnel is made of same 'material' as the obstacle, meaning that same no-slip condition holds at the tunnel and obstacle.

That no-slip condition holds at the obstacles can be seen from following excerpt of Propagation function (from FCHC, but same principle is applied in PI).

```
/* Particle at the cell C[i] propagates along the lattice vector c[i]
   to a new node. */
if (n & C[i])
{
   new_x = PeriodicBC(x + c[i][0], X);
   new_y = PeriodicBC(y + c[i][1], Y);
   new_z = PeriodicBC(z + c[i][2], Z);
   if (to[new_x][new_y][new_z] & OBS)
   /* However, if there is obstacle in that node, it gets to the cell
      Reverse[i], that is cell diagonal to the C[i]. Let's say it had
      velocity v1 = [1,0,-1,0], by reflection it gained velocity v2 =
      [-1,0,1,0]. In the next step, particle propagates back to the
      node where it came from. */
   /* Hence, the velocity at the obstacle is v1 + v2 = 0, so we really
      fulfilled no-slip condition. */
      to[new_x][new_y][new_z] |= Reverse[i];
   else
      to[new_x][new_y][new_z] |= C[i];
}
```

15.1 Flow around the sphere

We emphasize that the vector field is the physical velocity field. Each velocity vector is computed over $N = 80^3/8 = 64000$ nodes for PI and $N = 80^3 = 521000$ for FCHC (by the function $compute_velocity$ above).

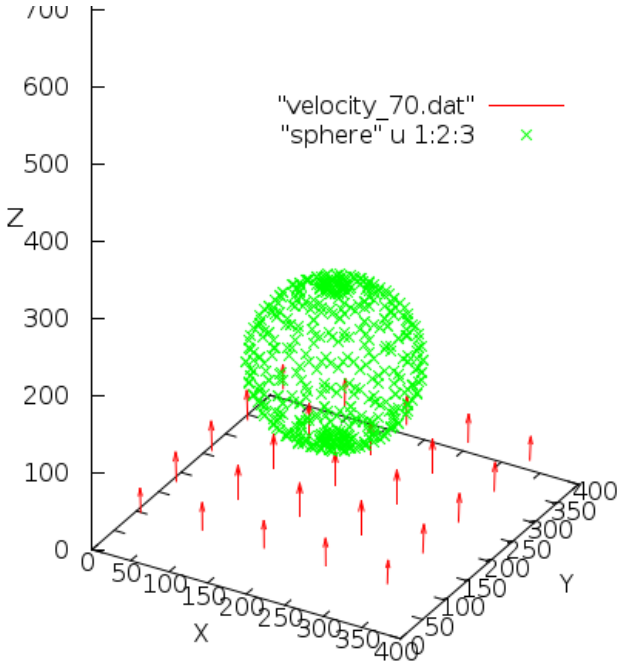


Figure 15.1: PI - flow around sphere, time 70

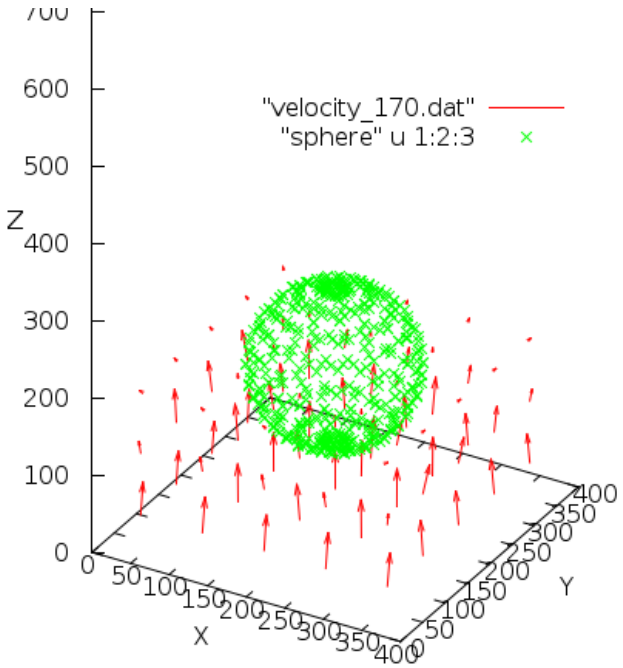


Figure 15.2: PI - flow around sphere, time 170

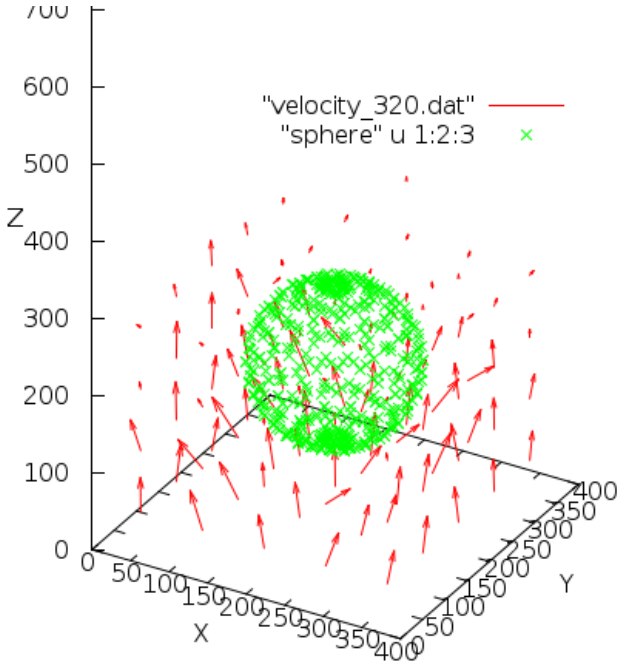


Figure 15.3: PI - flow around sphere, time 320

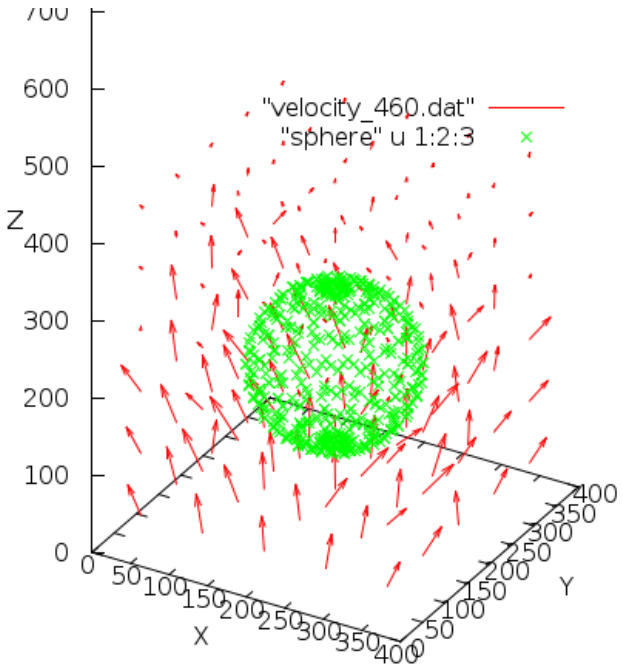


Figure 15.4: PI - flow around sphere, time 460

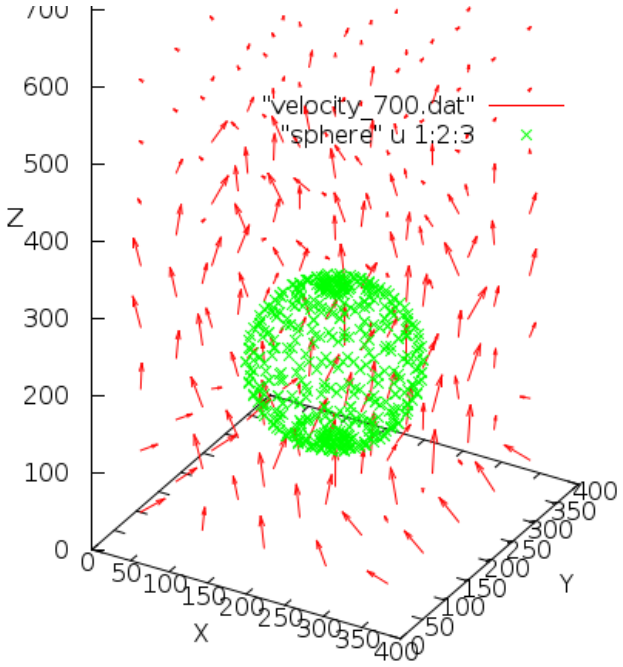


Figure 15.5: PI - flow around sphere, time 700

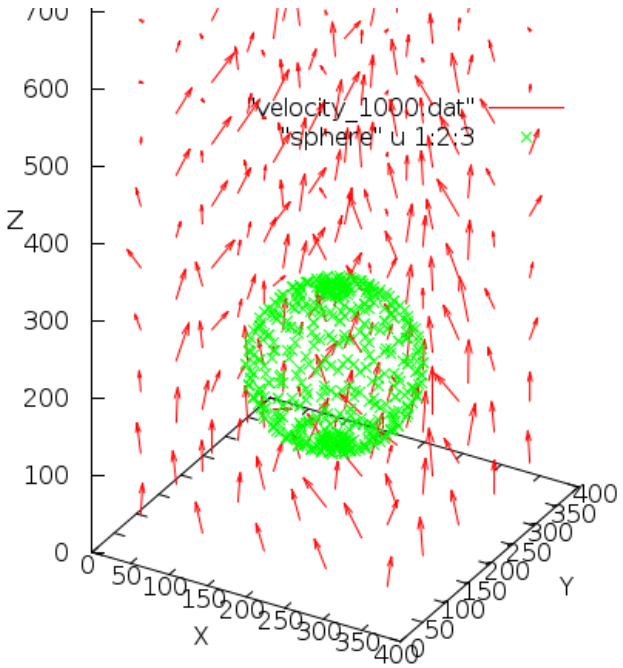


Figure 15.6: PI - flow around sphere, time 1000

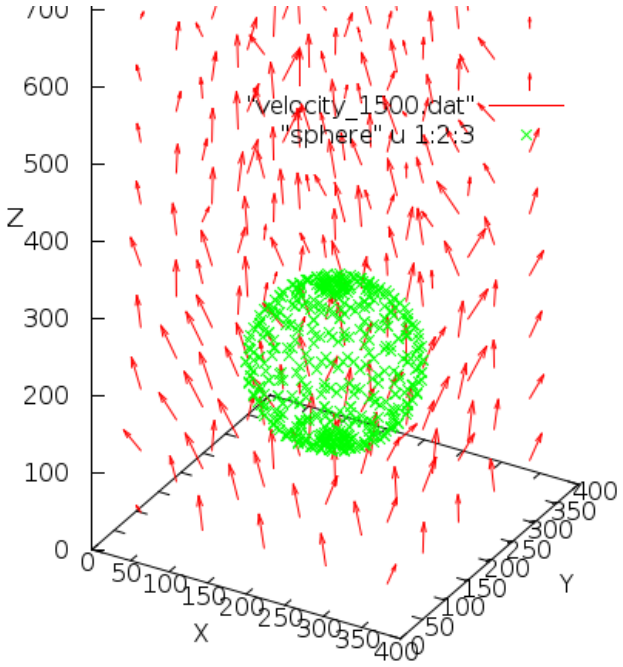


Figure 15.7: PI - flow around sphere, time 1500

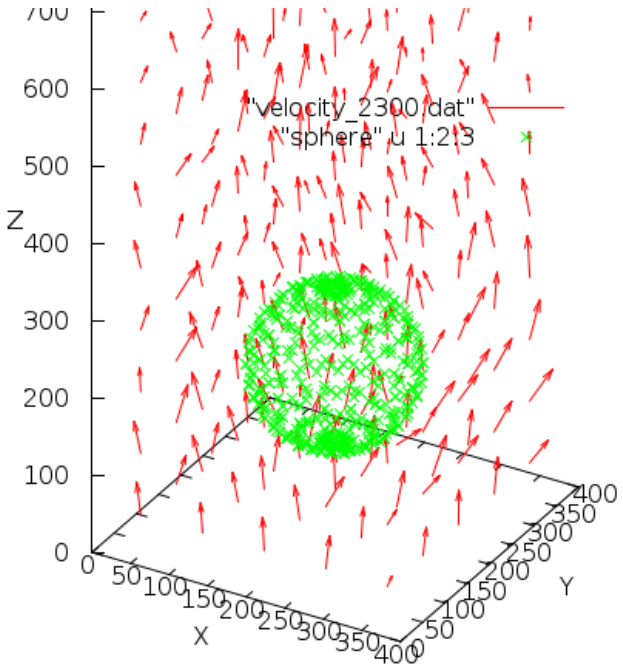


Figure 15.8: PI - flow around sphere, time 2300

15.2 Flow around the disk

Flow around the disk was performed on the Pair-interaction model only, the macroscopic velocity field was obtained by averaging over $N = 40^3/8 = 8000$ nodes.

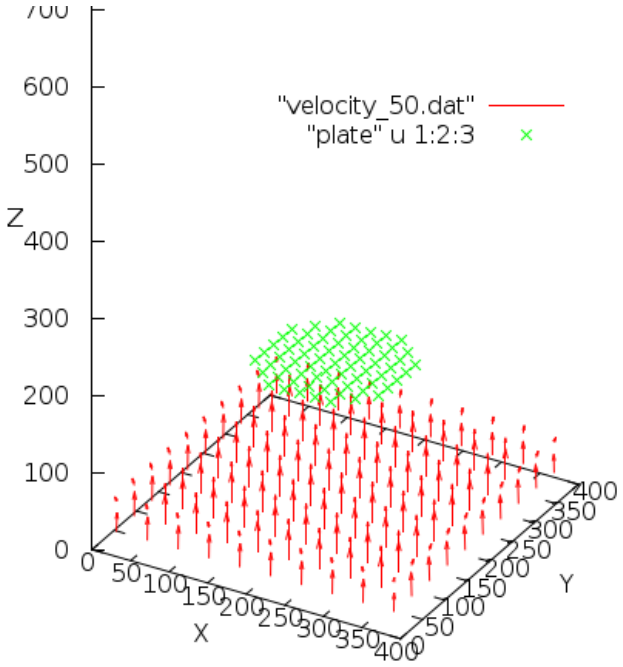


Figure 15.9: PI - flow around plate, time 50

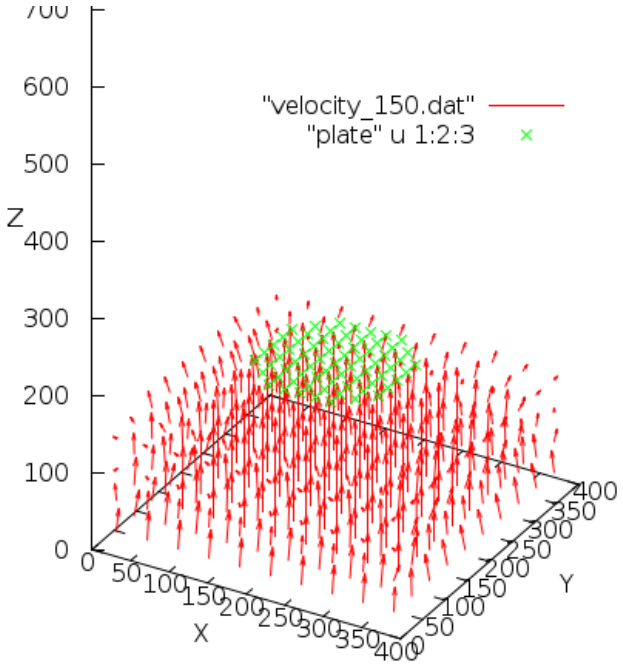


Figure 15.10: PI - flow around plate, time 150

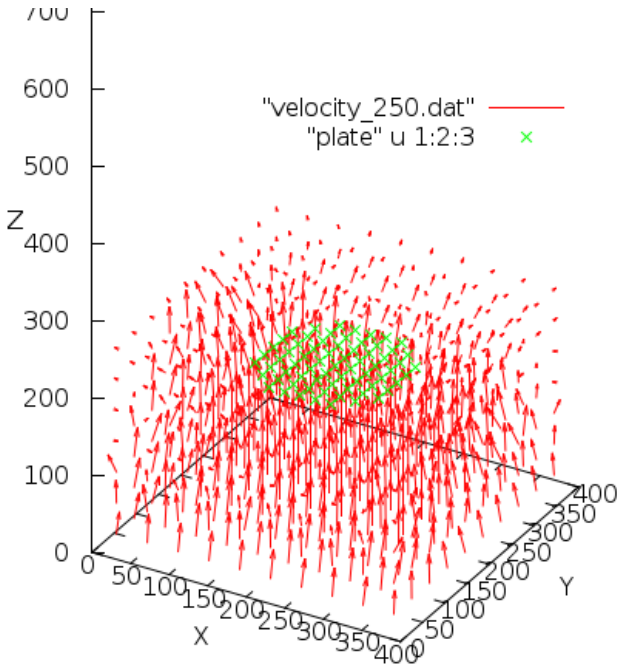


Figure 15.11: PI - flow around plate, time 250

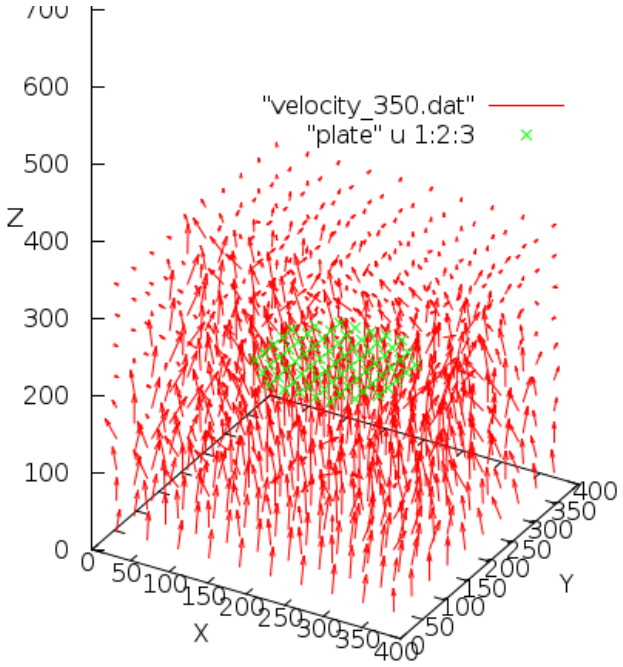


Figure 15.12: PI - flow around plate, time 350

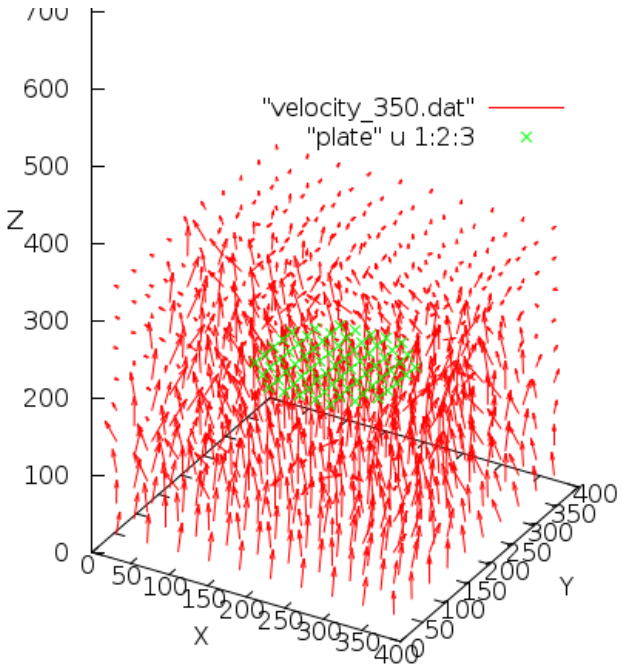


Figure 15.13: PI - flow around plate, time 350

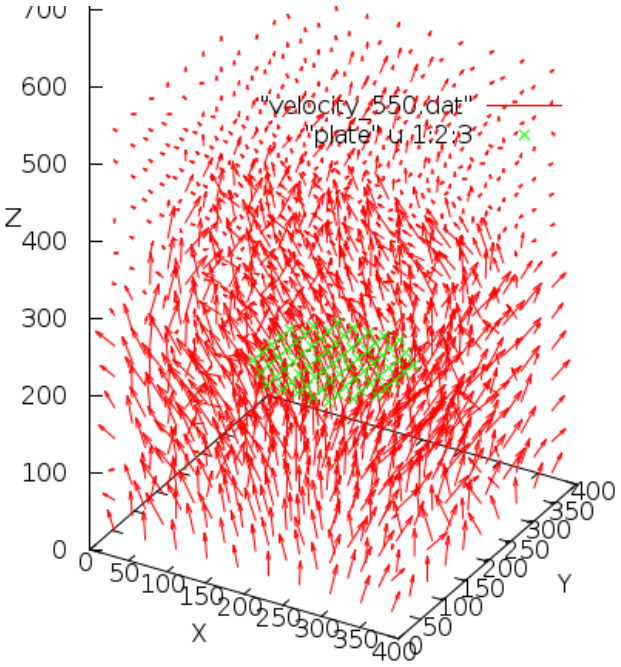


Figure 15.14: PI - flow around plate, time 550

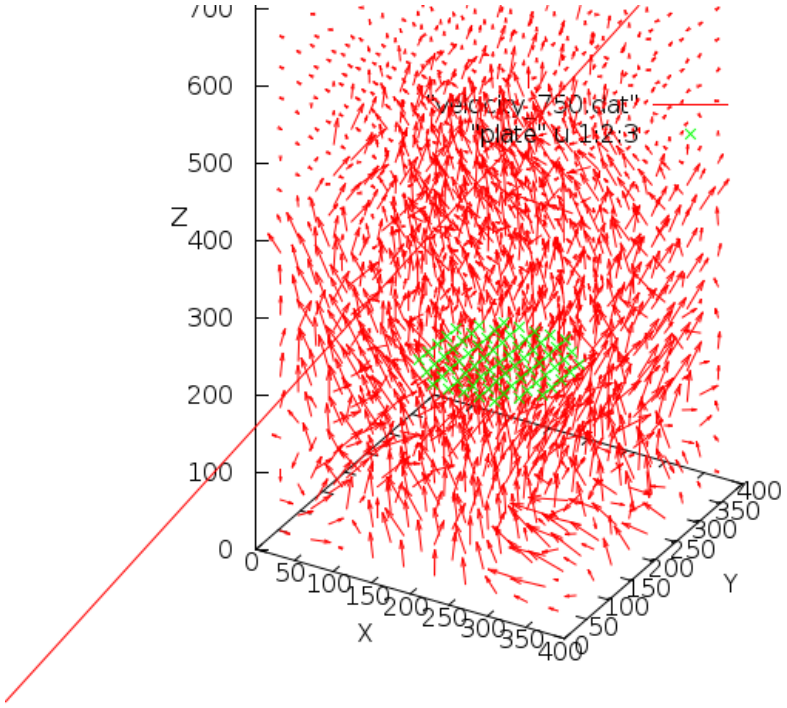


Figure 15.15: PI - flow around plate, time 750

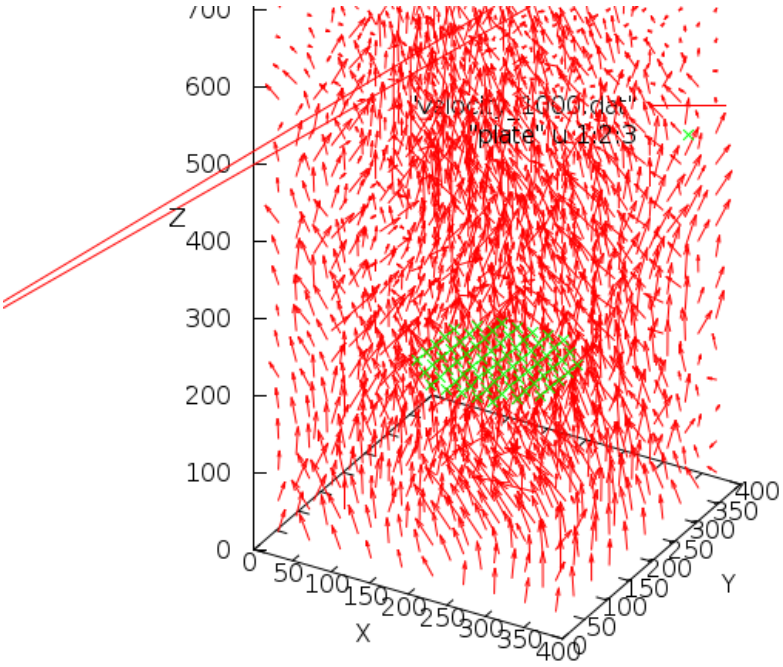


Figure 15.16: PI - flow around plate, time 1000

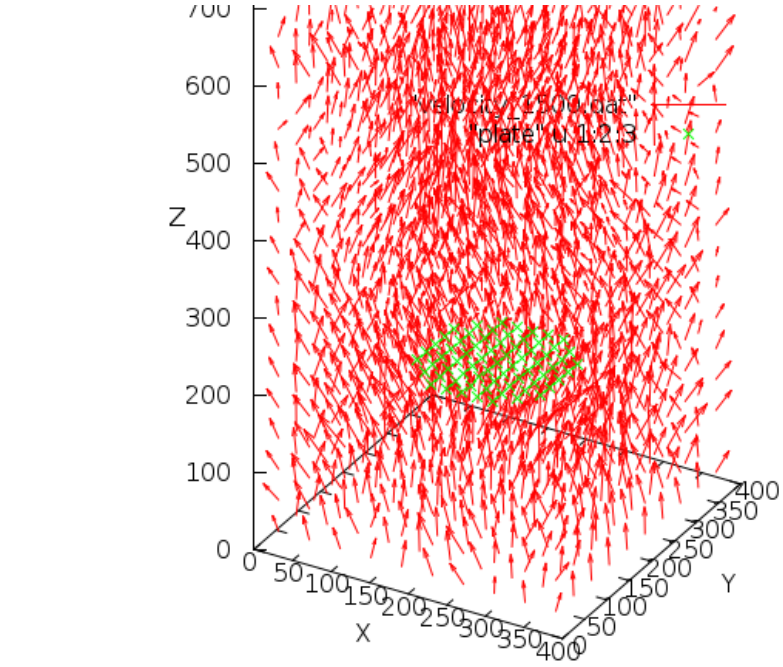


Figure 15.17: PI - flow around plate, time 1500

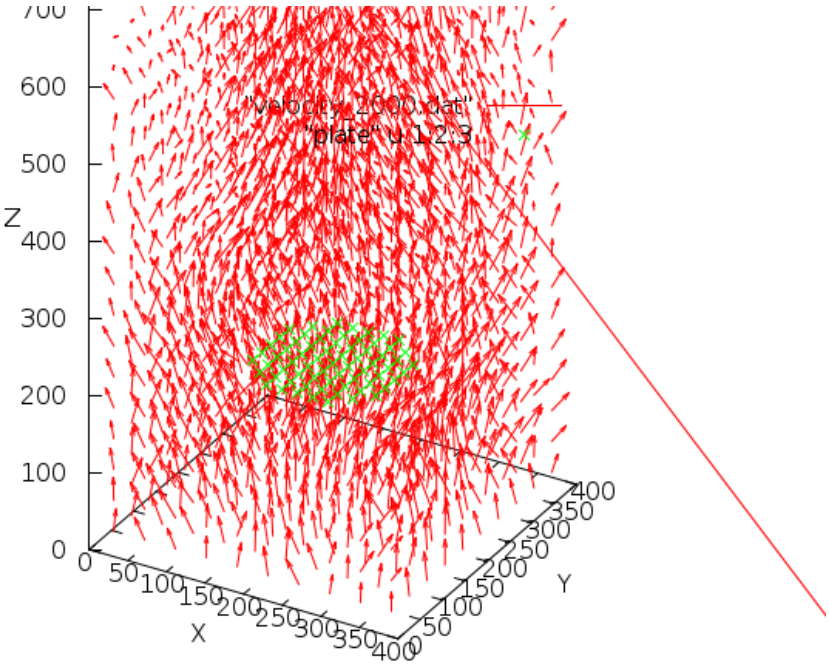


Figure 15.18: PI - flow around plate, time 2000

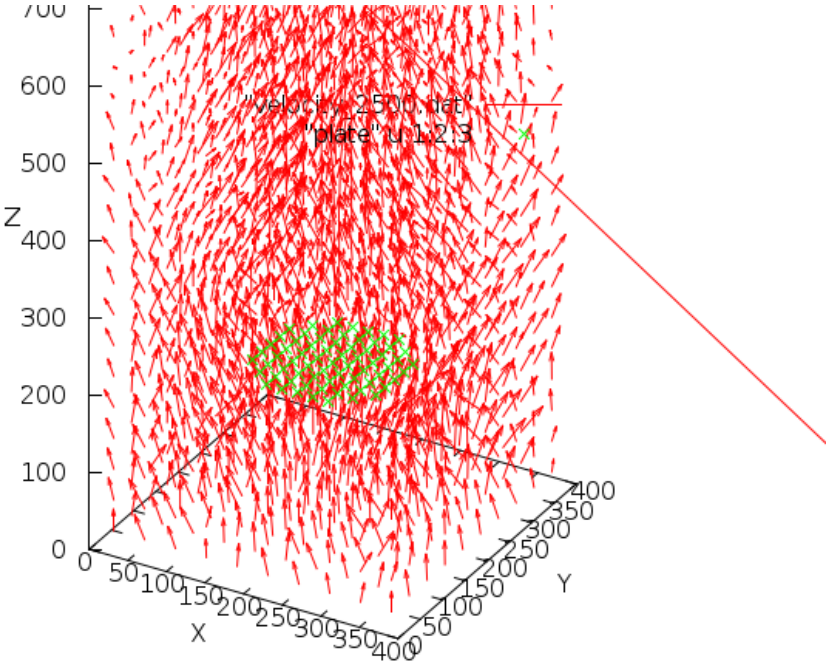


Figure 15.19: PI - flow around plate, time 2500

16. Fully developed turbulence simulated on LGCA

We are intending to simulate the fully developed turbulent flow using FCHC and PI-3D model and see how close we can get to the theoretical prediction of K41 theory and experimental results.

16.1 Inward flow on the sphere

To develop the turbulent flow that would be isotropic, we chose following setting for simulation:

1. We defined sphere with diameter 800 nodes.
2. Every time step, a new layer of the particles was created with the momenta directed into the middle of the sphere.
3. If the particle strayed out the sphere, it was forgotten.

Since the particles have discrete momenta, most of them cannot be directed from the sphere to the middle, but we can impose any direction on the mean flow (up to the precision allowed by coarse graining).

To see what directions of particle momenta are allowed, consider the cube inscribed in the sphere as on the figure 16.1.

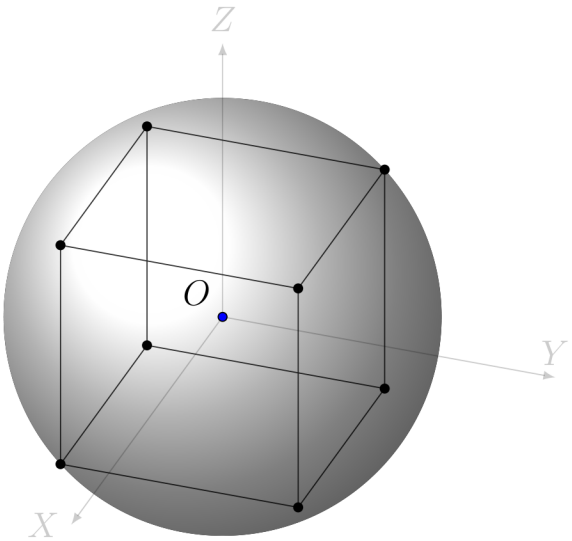


Figure 16.1: State of a node before collision

From a single node, we can direct particle in 26 basic direction, corresponding to the faces, edges and vertices of the cube.

Six directions correspond to the vectors, that are normal of the faces of the cube

$$[1,0,0],\ [-1,0,0],\ [0,1,0],\ [0,-1,0],\ [0,0,1],\ [0,0,-1]. \tag{16.1}$$

Eight directions corresponds to the "diagonal" vectors

$$[1,1,1],\ [-1,1,1],...,[-1,-1,-1] \tag{16.2}$$

and the other twelve vectors read

$$[1,1,0],\ [1,0,1],\ [-1,1,0],...,[0,-1,-1]. \tag{16.3}$$

Therefore, we divide the cube on the 26 surface areas. On each of these areas, particles have momentum in the direction corresponding to one of the vectors above, that points to the middle of the sphere.

Following figures indicate that setting described above results in the mean flow that is symmetric and directs to the middle of the sphere.

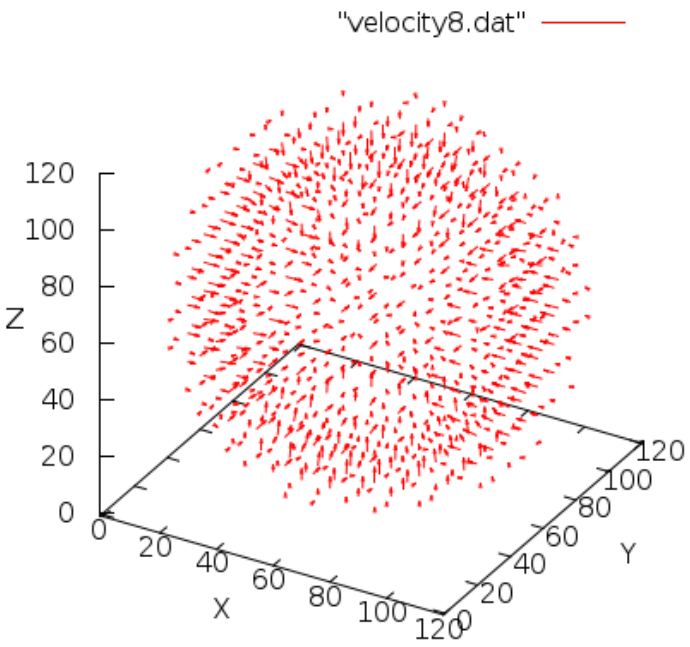


Figure 16.2: Velocity field at time 8 - FCHC inward flow

16.1.1 FCHC implementation of inward flow from the sphere

Functions that are imposing inward flow on the sphere are similar for PI and FCHC, so we explain the algorithm only on one of them.

```
// Cells in CELL_DIR[0] sum-up to momentum [6,0,0,0]
// Cells in CELL_DIR[1] sum-up to momentum [0,6,0,0]
// ...
// Cells in CELL_DIR[3] sum-up to momentum [-6,0,0,0]
// ...
```

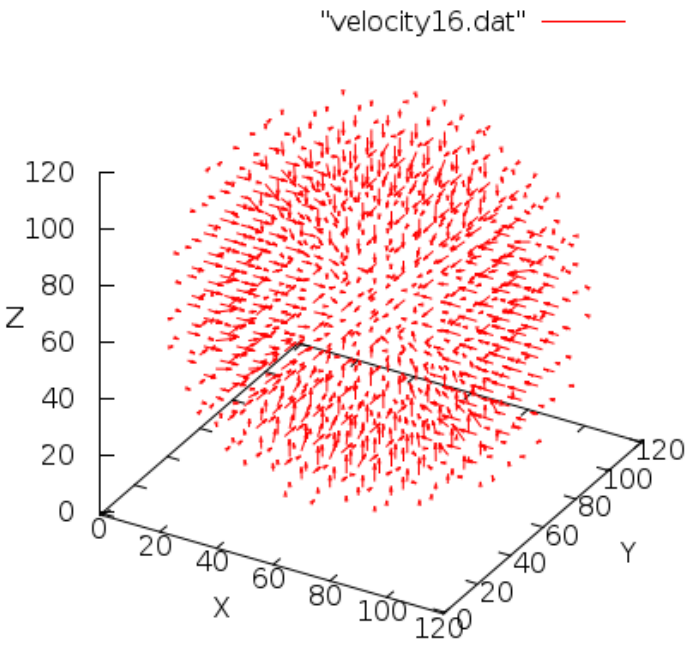


Figure 16.3: Velocity field at time 16 - FCHC inward flow

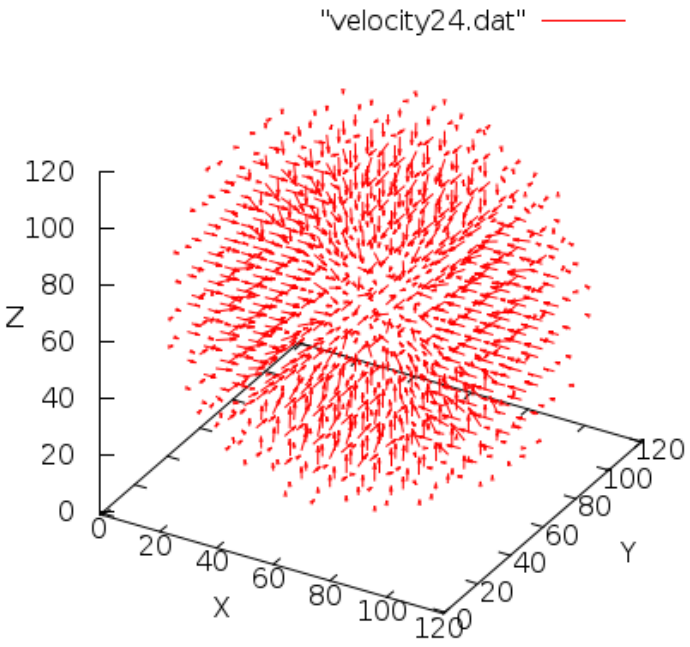


Figure 16.4: Velocity field at time 24 - FCHC inward flow

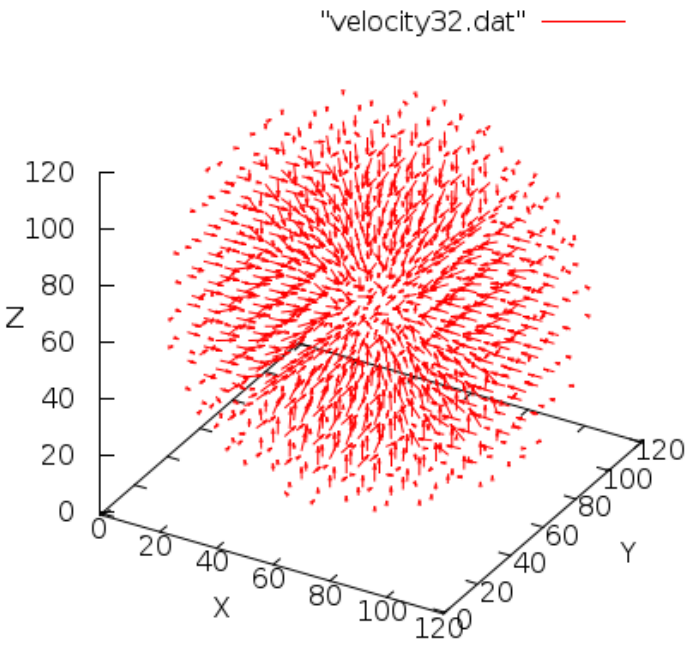


Figure 16.5: Velocity field at time 32 - FCHC inward flow

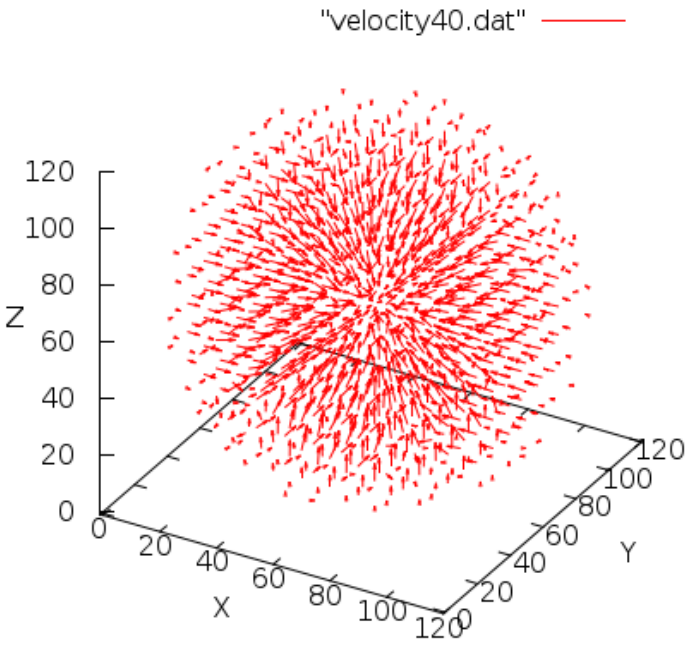


Figure 16.6: Velocity field at time 40 - FCHC inward flow

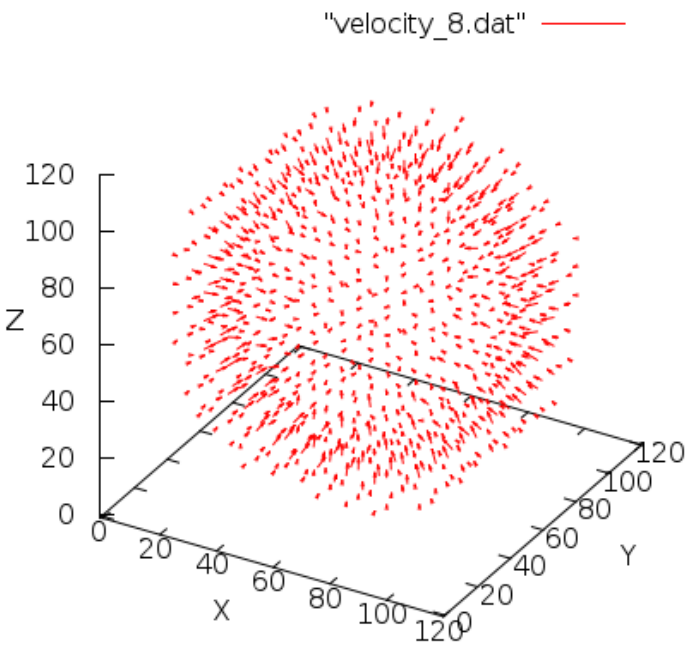


Figure 16.7: Velocity field at time 8 - PI inward flow

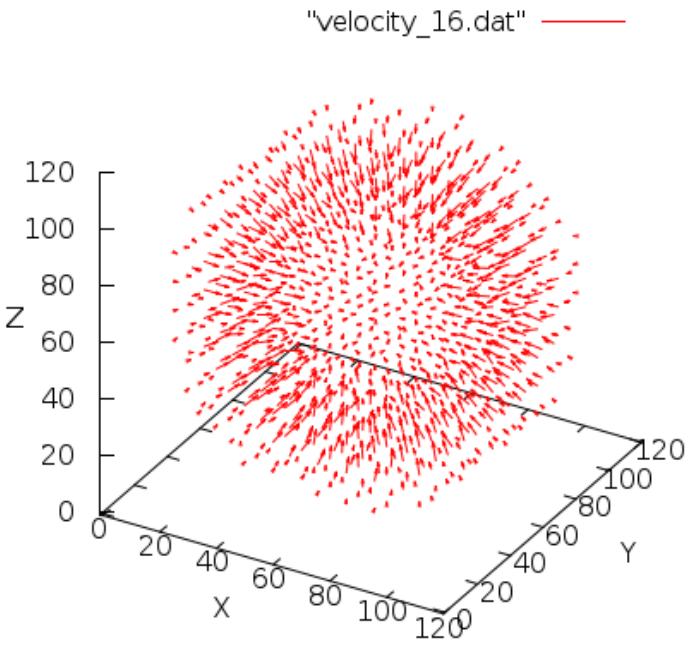


Figure 16.8: Velocity field at time 16 - PI inward flow

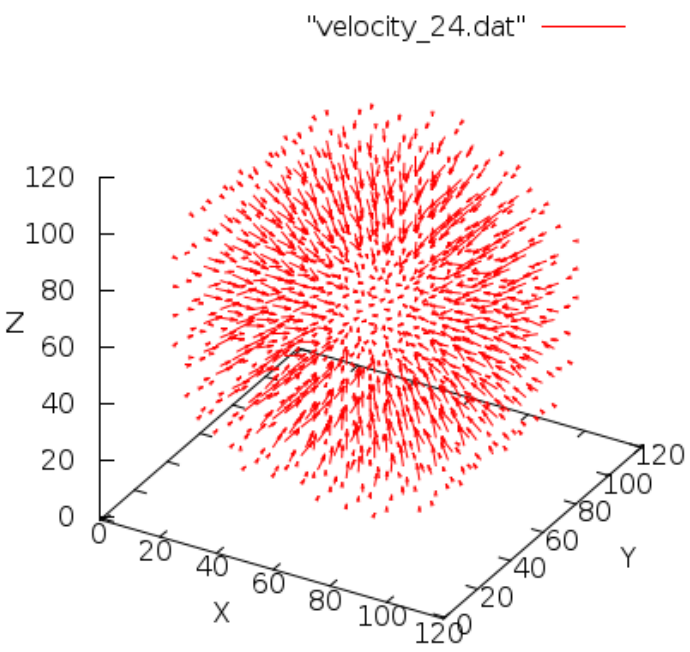


Figure 16.9: Velocity field at time 24 - PI inward flow

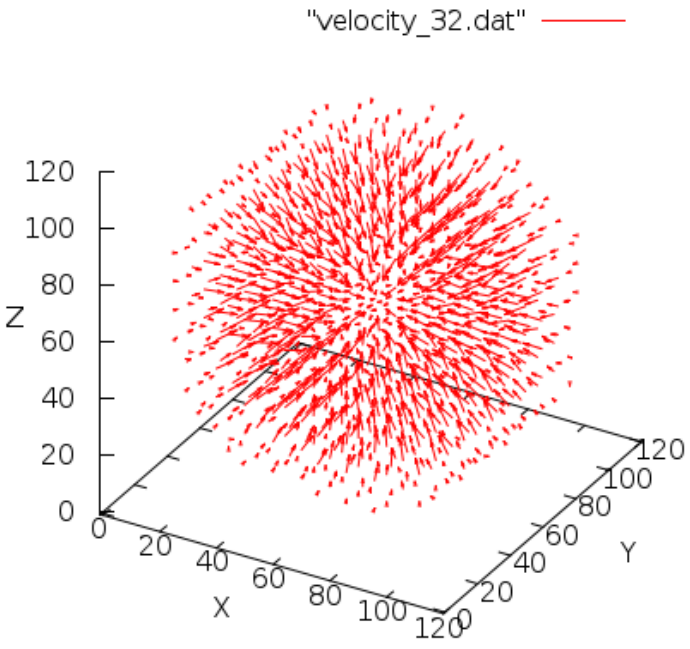


Figure 16.10: Velocity field at time 32 - PI inward flow

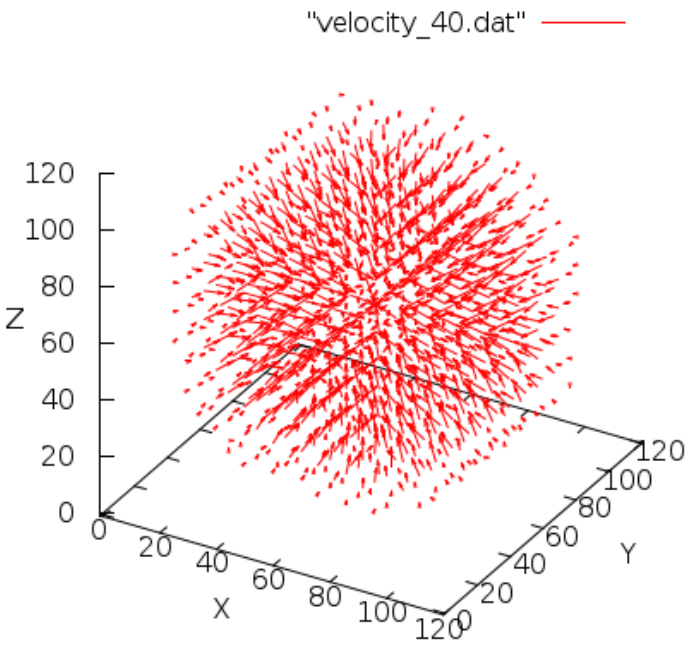


Figure 16.11: Velocity field at time 40 - PI inward flow

```
// Cells in CELL_DIR[5] sum-up to momentum [0,0,-6,0]
int CELL_DIR[6][6] = {
   { C1, C2, C5, C6, C9, C10 },
   { C1, C3, C13, C14, C17, C18 },
   { C5, C7, C13, C15, C21, C22 },
   { C3, C4, C7, C8, C11, C12 },
   { C2, C4, C15, C16, C19, C20 },
   { C6, C8, C14, C16, C23, C24 }
};

/*
 To achieve momentum in XY direction, we should not use all cells from
    CELL_DIR[0] and CELL_DIR[1].
 For example, particle in C3 has positive momentum in Y direction, but
    negative momentum in X direction, so we do not include it.
 The case is even stronger for 3 non-zero components of required
    momentum.
 Hence we implemented function flow_dir that computes which cells
    should be occupied.
 Arguments a, b and c specify components of momentum:
 0: positive X component (CELL_DIR[0])
 1: positive Y component (CELL_DIR[1])
 2: positive Z component ...

 3: negative X component ...
```

```
 4: negative Y component ...
 5: negative Z component (CELL_DIR[5])
 */
int flow_dir(int a, int b, int c = -1)
{
   int dir;
   int not_dir;
   int i, j, k;

   for (i = 0; i < 6; i++)
   {
      // Into dir, we add all cells that have positive momentum in the
         direction 'a' and 'b'
      // We want to occupy these cells by particles.
      dir |= CELL_DIR[a][i];
      dir |= CELL_DIR[b][i];

      // Into not_dir, we add all cells with negative momenta in
         direction in 'a' and 'b'.
      // These cells must not be occupied by particles.
      not_dir |= CELL_DIR[(a + 3) % 6][i];
      not_dir |= CELL_DIR[(b + 3) % 6][i];

      // Same procedure for another component of momentum, if it is
         specified.
      if (c > 0)
      {
         dir |= CELL_DIR[c][i];
         not_dir |= CELL_DIR[(c + 3) % 6][i];
      }
   }
   // To conclude:
   // dir are cells that we want to occupy,
   // not_dir are cells that must not be occupied,
   // so ~not_dir are cells that can be occupied (negation of not_dir
      flips its bits).

   //We return cells that we want to occupy and can be occupied.
   return dir & (~not_dir);
}


/* Set_initial_sphere is called just before Collision and Propagation
    at every time step. */
void set_initial_sphere(int***a, int X, int Y, int Z, int R)
{
   // The particles will be created on the layer between R and R-2.
   int R2out = R*R;
   int R2in = (R - 2)*(R - 2);

   // We will identify segments of the sphere by its distance from the
```

```
      middle.
   int up = R*cos(PI / 4);
   int down = R*sin(PI / 4);

   // These will identify position of node on lattice
   int x, y, z;

   // These are x-R, y-R and z-R, so that [0,0,0] is in the middle of
      the sphere.
   int x1, y1, z1;

   // These are squared x1, y1, z1.
   int x2, y2, z2;

   int i;
   int c;
#pragma omp parallel for private (x, y, z, x1, y1, z1, x2, y2, z2, i,
    c)
   for (x = 0; x < X; ++x)
   {
      x1 = x - R;
      x2 = x1*x1;
      for (y = 0; y < Y; ++y)
      {
         y1 = y - R;
         y2 = y1*y1;
         for (z = 0; z < Z; ++z)
         {
            z1 = z - R;
            z2 = z1*z1;

            // If the node is on the layer between R and R-2, we occupy
               it by particles.
            if (x2 + y2 + z2 > R2in && x2 + y2 + z2 < R2out)
            {
               c = 0;
               // If the node is on the upper-most segment of sphere,
                  particles have momentum in negative Z direction only
               if (z1 > up)
               {
                  for (i = 0; i < 6; ++i)
                  {
                     c |= CELL_DIR[5][i];
                  }
               }
               // If the node is on the downer-most segment, particles
                  have momentum in positive Z direction.
               else if (z1 < -up)
               {
                  for (i = 0; i < 6; i++)
                  {
```

```
      c |= CELL_DIR[2][i];
   }
}
// Similarly for segments on axis Y and Z.
else if (y1 > up)
{
   for (i = 0; i < 6; i++)
   {
      c |= CELL_DIR[4][i];
   }
}
else if (y1 < -up)
{
   for (i = 0; i < 6; i++)
   {
      c |= CELL_DIR[1][i];
   }
}
else if (x1 > up)
{
   for (i = 0; i < 6; i++)
   {
      c |= CELL_DIR[3][i];
   }
}
else if (x1 < -up)
{
   for (i = 0; i < 6; i++)
   {
      c |= CELL_DIR[0][i];
   }
}
// If x1 is close to 0, then particles should sustain
    X-component of position.
// Hence X-component of momentum of particles should be
    zero.
else if (x1 < down && x1 > -down)
{
   // If y1 is positive, we send particles in negative Y
       direction, and vice versa.
   // Same for z1.
   c = flow_dir(y1 > 0 ? 4 : 1, z1 > 0 ? 5 : 2);
}
else if (y1 < down && y1 > -down)
{
   c = flow_dir(x1 > 0 ? 3 : 0, z1 > 0 ? 5 : 2);
}
else if (z1 < down && z1 > -down)
{
   c = flow_dir(x1 > 0 ? 3 : 0, y1 > 0 ? 4 : 1);
}
```

```
            // Else the node is on the segment corresponding to one
                of 8 corners of the inscribed cube.
            else
            {
               c = flow_dir(x1 > 0 ? 3 : 0, y1 > 0 ? 4 : 1, z1 > 0 ?
                  5 : 2);
            }
            a[x][y][z] = c;
         }
      }
   }
}
```

16.2 Statistical properties of the flow

So far, the best understanding of the phenomena arising in turbulent flow is by them means of statistical analysis.

In previous chapter, we described setting of the simulation that should lead to the fully-developed turbulent flow.

In this chapter, we will inspect statistical properties of this flow to see whether isotropy and homogenity was recovered in our model, as predicted by K41 theory and confirmed by experimental data.

The weak point of our approach might be implicit assumption of the ergodicity of the models. All the statistical quantities that we computed and visualized were obtained by averaging over 10 000 to 15 000 time-steps, or equivalently $1000-1500$ units of time.

16.2.1 First statistical moment - the mean velocity field

Assuming ergodicity, we define the first statistical moment of the velocity field $v(t, \boldsymbol{r})$ as

$$\langle v(\boldsymbol{r}) \rangle = \sum_{t=T_1}^{T_2} v(t, \boldsymbol{r}), \tag{16.4}$$

In both our models, we used similar implementation.

```
/* Every-time that velocity is updated, its value is counted */
void compute_mean(double****v, double****mean, int I, int J, int K,
    int div)
{
   // i,j,k denotes position vector, l denotes component of the vector
   int i, j, k, l;

#pragma omp parallel for private (i, j, k, l)
   for ( i = 0; i < I; i++)
      for ( j = 0; j < J; j++)
         for ( k = 0; k < K; k++)
```

```
            for ( l = 0; l < 3; l++)
               mean[i][j][k][l] += v[i][j][k][l];

}

/*  So far we have sum of velocities over specific time interval, only
    now we compute the mean */
void finalize_mean(double****mean, int I, int J, int K, int steps)
{
   int i, j, k, l;

#pragma omp parallel for private (i, j, k, l)
   for ( i = 0; i < I; i++)
      for ( j = 0; j < J; j++)
         for ( k = 0; k < K; k++)
            for ( l = 0; l < 3; l++)
               mean[i][j][k][l] /= steps;
}
```

16.2.2 Second statistical moment – the covariance tensor

As discussed in chapter on probabilistic methods, all higher statistical moments can be obtained from the second moment (for the quantities that have normal distribution). That makes it one of the most important quantities that characterize velocity field.

Intuitively speaking, this tensor measure how strong are correlated components of the velocities.

For the centered functions, the covariance tensor reads

$$\Gamma^c_{ij} = \langle v_{ij} \rangle, \tag{16.5}$$

but for the functions with non-zero mean value (that will be our case), formula has to be modified

$$\Gamma_{ij} = \langle v_{ij} \rangle - \langle v_i \rangle \langle v_j \rangle \,. \tag{16.6}$$

Implementing this formula is very straight-forward.

```
/* The covariant tensor double*****g was initialized by zero values. */
/* Each update of velocity field v is followed by function
    'covariance_tensor', */
void covariance_tensor(double****v,double*****g, int I, int J, int K)
{
   int i,j,k,d,e;
#pragma omp parallel for private (i,j,k,d,e)
   for(i=0; i<I; ++i)
      for(j=0; j<J; ++j)
         for(k=0; k<K; ++k)
            for(d=0; d<3; ++d)
               for(e=0; e<3; ++e)
```

```
                g[i][j][k][d][e] += (v[i][j][k][d]*v[i][j][k][e]);
}

/* After specified number of 'steps', finalize_covariance_tensor is
    called */
void finalize_covariance_tensor(double****mean, double*****g, int I,
    int J, int K, int steps)
{
   // Variables i,j,k stands for position vector in the velocity field,
   // d,e are indexes of the tensor 'g'
   int i,j,k,d,e;
#pragma omp parallel for private (i,j,k,d,e)
   for(i=0; i<I; ++i)
      for(j=0; j<J; ++j)
         for(k=0; k<K; ++k)
            for(d=0; d<3; ++d)
               for(e=0; e<3; ++e)
               // Avaraging by number of 'steps' and subtracting mean
                   velocity product, we obtained the covariant tensor at
                   r = (i,j,k)
                  g[i][j][k][d][e] = (g[i][j][k][d][e] / steps ) -
                     mean[i][j][k][d]*mean[i][j][k][e];
}
```

In order to interpret the data obtained in the simulation, we have to compare the correlation tensor to one predicted by Kolmogorov's K41 theory of fully-developed turbulence.

In the isotropic situation, there is no preferred direction and, thus, the only second-order tensor at our disposal is the Kronecker delta δ_{ij}. Hence, the correlation tensor must be of the form

$$\Gamma_{ij} = K\,\delta_{ij}, \tag{16.7}$$

where K is, in general, function of spatial coordinates, having dimension of velocity-squared. However, the requirement of global isotropy implies global homogeneity, so that K is in fact constant. Physically speaking, this form of correlation tensor means that different Cartesian components of the velocity field at given point are uncorrelated, i.e. independent.

In the anisotropic situation, on the other hand, there is a preferred direction which can be represented by a unit vector $\boldsymbol{n}$. In that case, the most general second-order tensor is given by

$$\Gamma_{ij} = K_1\,\delta_{ij} + K_2\,n_i\,n_j, \tag{16.8}$$

where K_1 and K_2 are functions in general. In other words, there are just two independent components of the correlation tensor.

In the most general situation, when the anisotropy is specified by three linearly independent vectors, we get six independent components, which simply corresponds to a general second-order symmetric tensor.

Any such tensor defines a quadratic form and a family of surfaces parametrized by constant C via equation

$$\Gamma_{ij}\,x_i\,x_j = C. \tag{16.9}$$

In the isotropic case these surfaces are concentric spheres. In the anisotropic case, we will get ellipsoids, because correlation tensor is positive-definite[1]. We can conclude that the deviation of the surfaces from spherical shape measures the failure of the system to be isotropic.

The Kolmogorov theory then makes assumptions known as Kolmogorov hypotheses, asserting that the correlation functions inside the inertial interval are independent of the external scale (energy injection) and of the dissipation scale.

These assumptions, together with isotropy and dimensional analysis, then restrict possible functional dependence of correlators on the parameters of the system, namely the dissipation rate and viscosity (see, e.g. [7] and references therein). In this thesis, we do not go into details of Kolmogorov's theory, instead we focus on the question to what extent the assumption of isotropy is satisfied in the cellular automaton model.

16.2.3 Structure functions

Another important quantity characterizing the velocity field are the structure functions.

For the centered velocity field, formula for the second-order structure functions reads

$$S_2^c = \langle\, [v_r(x_M) - v_r(x_R)]^2 \,\rangle, \tag{16.10}$$

where v_r is the projection of the velocity vector onto radial vector connecting the points x_M (middle of the sphere) and x_R (variable, running over the sphere with radius R).

Because velocity field that we study has non-zero mean value, we need to implement modified formula

$$S_2 = \langle\, [v_r(x_M) - v_r(x_R)]^2 \,\rangle - [\, u_r(x_M) - v_r(x_R)\,]^2. \tag{16.11}$$

where $\boldsymbol{u}$ is the mean velocity.

In this way, we examine the dependence of the structure function S_2 on the angles θ and ϕ. In the fully developed turbulent flow, S_2 should be constant function of θ and ϕ. For our imperfect models, we can expect certain dependence.

```
/* After every update of velocity field, struct_on_sphere sums
    [v_r(x_M) - v_r(x_R)] so at the end of computaton, we can compute
    the structure function by formul \ref{str2} */
/* The angular distance between various points x_R on the sphere is PI
    / N */
void struct_on_sphere(double****v, double**Cor, int I, int J, int K,
    int N=CIRC)
{
   /* x and y iterates over the sphere */
   int x, y;
```

[1]This can be easily seen by the following argument. In the frame of main axes of Γ_{ij}, this tensor is diagonal and its diagonal terms are $\Gamma_{ii} = \langle v_i^2 \rangle \geq 0$. Since quadratic forms are inertial with respect to rotations, this property is preserved in any frame.

```
   /* Cartesian coordinates of x_R */
   int i, j, k;

   /* Radius of of the sphere that i,j,k parametrize */
   int R = I/4;

   double sin_theta, sin_phi, cos_theta, cos_phi;

   /* Initialization of angles */
   double theta = 0;
   double phi = 0;

   /* The angles theta and phi will grow by PI / N */
   double step = PI / N;

   /* coordinates of v_M are fixed, it is the middle of the sphere */
   double* v_s = v[I/2][J/2][K/2];

   double* v_r = new double[3]();

   double v_x, v_y, v_z;


//#pragma omp parallel for private (i,j,k, x,y, sin_theta,sin_phi,
    cos_theta,cos_phi, theta, phi, v_r, v_x, v_y, v_z)
   for (x = 0; x < 2*N; ++x)
   {
      phi += step;
      sin_phi = sin(phi);
      cos_phi = cos(phi);

      for (y = 0; y < N; ++y)
      {
         theta += step;
         sin_theta = sin(theta);
         cos_theta = cos(theta);

         /* We transform theta and phi into Cartesian coordinates i, j,
            k
         i = R*cos_phi*cos_theta;
         j = R*sin_phi*cos_theta;
         k = R*sin_theta;

         // We transform i, j, k into coordinate frame of the lattice
         v_r = v[i + I/2][j + J/2][k + K/2];

         /* Projection onto the radial vector
         v_x = (v_s[0] - v_r[0])*cos_phi*cos_theta;
         v_y = (v_s[1] - v_r[1])*sin_phi*cos_theta;
         v_z = (v_s[2] - v_r[2])*sin_theta;
```

```
            /* Add it to the previous sum */
            /* It will be processed by another function at the end of
               computation to obtain mean value */
            struc[x][y] += pow(v_x + v_y + v_z , 2);
         }
      }
   }

/* This function implements the formula \ref{str2} */
/* Implementation is similar to previous function, but it uses mean
   velocities and differs in final line */
void finalize_struct(double**str, double****mean, int I, int J, int K,
    int d, int N=CIRC)
{
   int x,y;
   int i, j, k;

   int R = I/4;
   int I2 = I/2;
   int J2 = J/2;
   int K2 = K/2;

   double sin_theta, sin_phi, cos_theta, cos_phi;

   double theta = 0;
   double phi = 0;
   double step = PI / N;

   double* v_s = mean[I2][J2][K2];
   double* v_r;

   //projections
   double v_x, v_y, v_z;


//#pragma omp parallel for private
    (i,j,k,x,y,sin_theta,sin_phi,cos_theta,cos_phi,theta,phi,v_r,v_x,v_y,v_z)
   for (x = 0; x < 2*N; ++x)
   {
      sin_phi = sin(phi);
      cos_phi = cos(phi);
      phi += step;

      for (y = 0; y < N; ++y)
      {
         sin_theta = sin(theta);
         cos_theta = cos(theta);
         theta += step;

         // coordinates of v_r
         i = R*cos_phi*cos_theta;
```

```
        j = R*sin_phi*cos_theta;
        k = R*sin_theta;


        v_r = mean[i + I2][j + J2][k + K2];

        v_x = (v_s[0] - v_r[0])*cos_phi*cos_theta;
        v_y = (v_s[1] - v_r[1])*sin_phi*cos_theta;
        v_z = (v_s[2] - v_r[2])*sin_theta;

        str[x][y] = ( str[x][y] / d ) - pow(v_x + v_y + v_z , 2);
      }
   }
}
```

16.3 Graphical representation of the obtained results

The figures 16.3 and 16.3 show that the correlation tensor is represented by highly ellipsoidal surfaces and hence, the isotropy seems to be significantly violated. This can have several reasons. First, an obvious possibility is a mistake in the code, but after careful inspection and previous applications with reasonable outcomes, we believe this is not the case.

Another and the most plausible source of anisotropy is the limited size of the grid. Based on the theoretical analysis, we expect that (much) larger grids will produce more isotropic configurations. Anisotropy can also be attributed to the initial and boundary conditions on the sphere, which are isotropic only to extent allowed by the geometry of the lattice and the coarse-graining of the velocity field. This question is left for the future investigation.

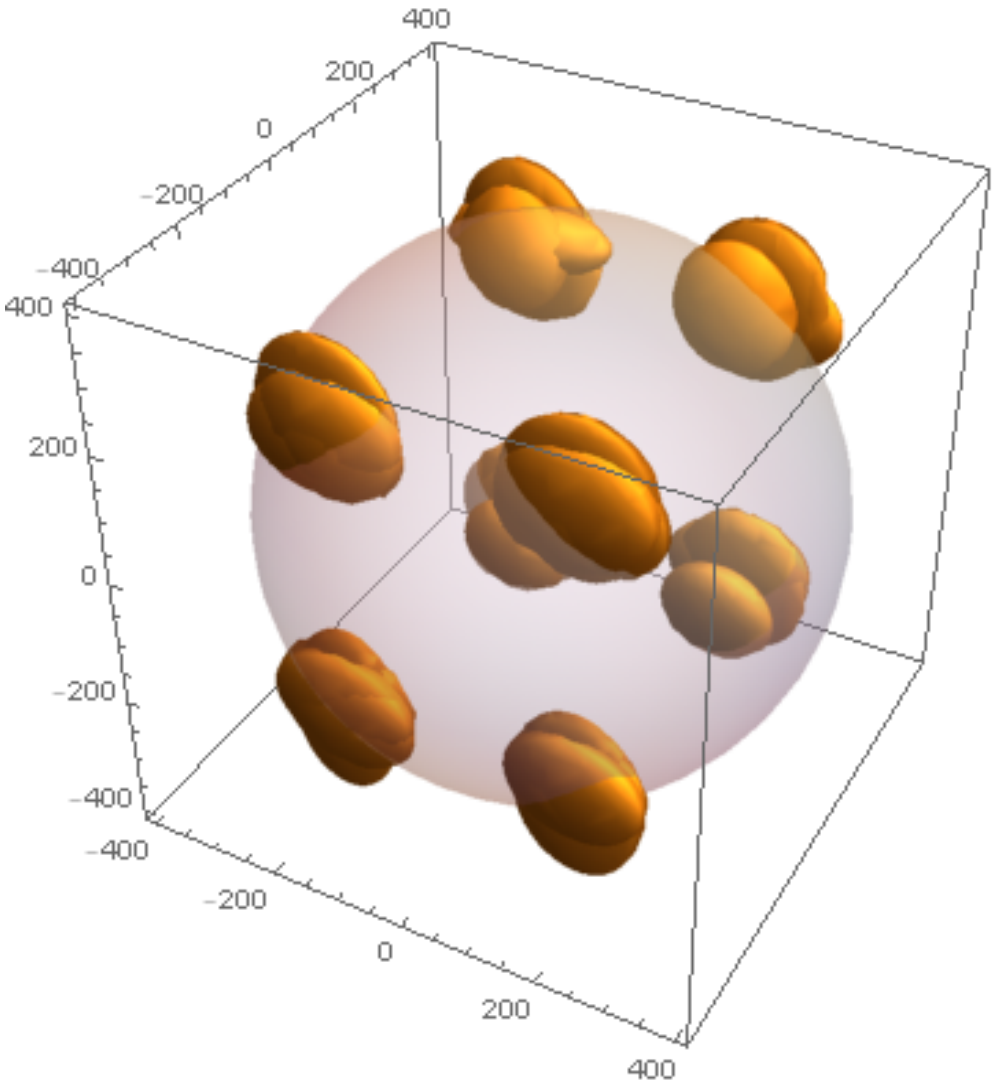


Figure 16.12: Covariance tensor field for PI with deterministic (standard) algorithm - only the tensors with highest norm are presented

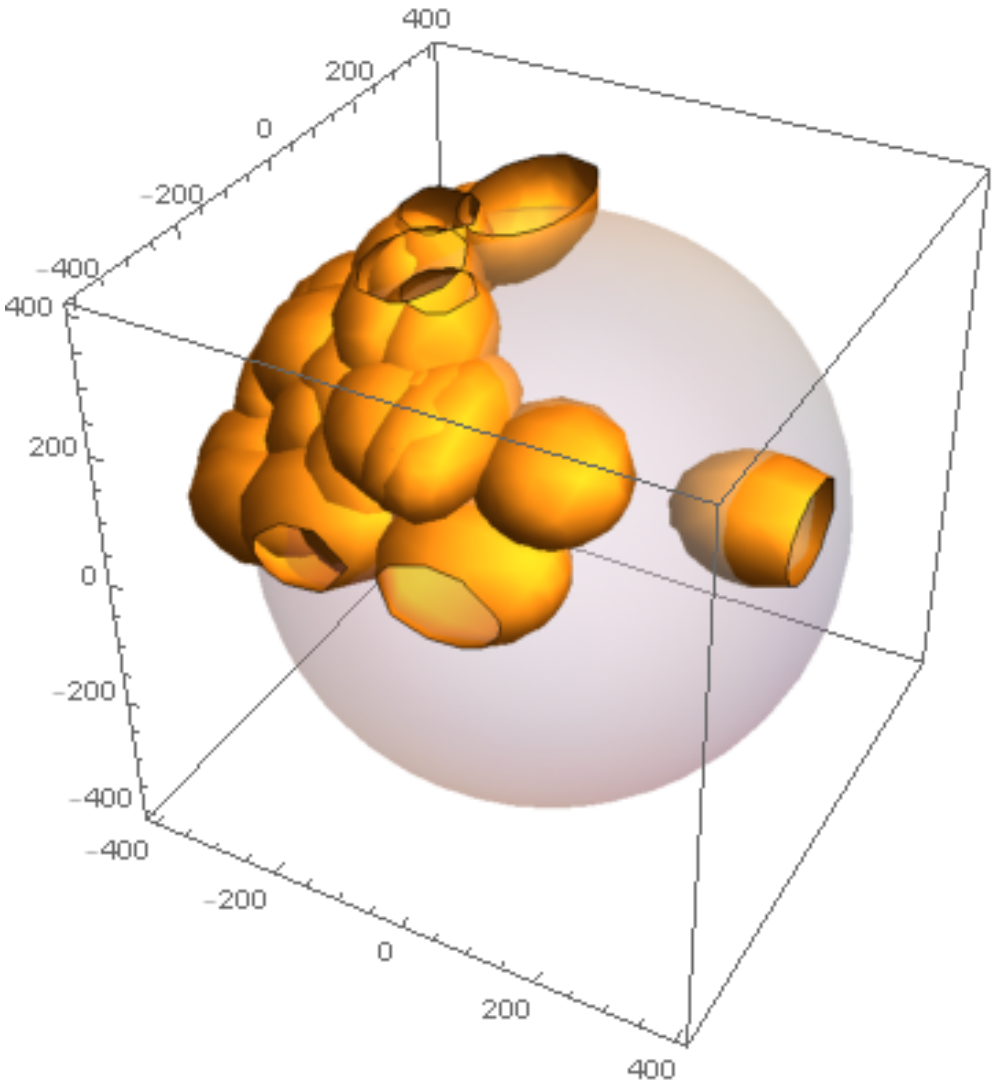


Figure 16.13: Covariance tensor field for PI with non-deterministic algorithm - future simulations are required to confirm and explain the discrepancy

Bibliography

[1] Wolf-Gladrow, Dieter A.: *Lattice-Gas Cellular Automata and Lattice Boltzmann Models – An Introduction* . New York: Springer, 2005.

[2] Henon, Michel: *Isometric collision rules for the four dimensional FCHC Lattice Gas*. Complex Systems 1, 1987

[3] Wolfram, Stephen: *New kind of science* . Wolfram Media, Inc., 2002

[4] Frisch, Uriel: *Lattice Gas Hydrodynamics in Two and Three Dimensions*. Complex Systems 1, 1987

[5] Bodenheimer, Michel, Laughlin, Gregory P., Rozyczka, Michal, Yorke, Harold W.: *Numerical methods in Astrophysics - An Introduction*. Taylor & Francis Group, LLC, 2007

[6] Nasilowski, Ralf : *A Cellular-Automaton Fluid Model with Simple Rules in Arbitrarily Many Dimensions*. Journal of Statistical Physics, Vol. 65, Nos. 1/2, 1991

[7] Scholtz, Martin: *Vplyv anizotropie na stabilitu skalovacich rezimov v modeli advekcie pasivnej vektorovej primesi*. Kosice, 2007

[8] Aaronson, Scott : *BOOK REVIEW on A New Kind of Science*. Quantum Information and Computation, Vol. 1, No. 0, 2001

[9] 't Hooft, Gerald : *The Cellular Automaton Interpretation of Quantum Mechanics*. arXiv:1405.1548v3 [quant-ph] 21 Dec 2015

List of Figures

List of Tables

List of Abbreviations

Attachments