

Czech Technical University in Prague
Faculty of Transportation Sciences

BACHELOR THESIS



Iliyas Boztayev

Simulation of two-dimensional turbulent flow using lattice-gas cellular automata

Department of Air Transport

Supervisor of the bachelor thesis: Mgr. Martin Scholtz, Ph.D.

Study programme: Technology and Technics
of Transport and Communications

Specialization: Aircraft Maintenance Technology

Prague 2015



K621..... Ústav letecké dopravy

ZADÁNÍ BAKALÁŘSKÉ PRÁCE (PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení studenta (včetně titulů):

Iliyas Boztayev

Kód studijního programu a studijní obor studenta:

B 3710 – TUL – Technologie údržby letadel

Název tématu (česky): **Simulace dvojrozměrného turbulentního toku pomocí celulárních automatů**

Název tématu (anglicky): Simulation of 2-dimensional Turbulent Flow Using Lattice-Gas Cellular Automata

Zásady pro vypracování

Při zpracování bakalářské práce se řiďte osnovou uvedenou v následujících bodech:

- Úvod
- Problematika celulárních automatů na obecné úrovni, implementace automatů v softwaru Maxima a C++, vykreslování výsledků pomocí softwaru GNUplot
- Detailnější studium „lattice-gas“ celulárních automatů, HPP a FHP modely, Naveirovy-Stokesovy rovnice jako limita těchto automatů
- Implementace a zopakování výsledků ze současné literatury
- Studium toku v okolí překážek, zejména vznik von Kármánových cest
- Řešení obtékání profilu libovolného tvaru v rámci FHP modelu
- Závěr

- Rozsah grafických prací: dle pokynů vedoucího bakalářské práce
- Rozsah průvodní zprávy: minimálně 35 stran textu (včetně obrázků, grafů a tabulek, které jsou součástí průvodní zprávy)
- Seznam odborné literatury: Brian Wylie, Application of Two-Dimensional Cellular Automaton Lattice-Gas Models to the Simulation of Hydrodynamics (Ph.D. thesis, 1990)
D. Wolf-Gladrow, Lattice-gas cellular automata and lattice Boltzmann models, Springer (2000)
U. Frish, B. Hasslacher, Y. Pomeau, Lattice-gas automata for the Navier-Stokes equation, Phys. Rev.

Vedoucí bakalářské práce: **Mgr. Martin Scholtz, Ph.D.**
Ing. Martin Novák, Ph.D

Datum zadání bakalářské práce: **24. října 2014**
(datum prvního zadání této práce, které musí být nejpozději 10 měsíců před datem prvního předpokládaného odevzdání této práce vyplývajícího ze standardní doby studia)


Datum odevzdání bakalářské práce: **24. srpna 2015**
a) datum prvního předpokládaného odevzdání práce vyplývající ze standardní doby studia a z doporučeného časového plánu studia
b) v případě odkladu odevzdání práce následující datum odevzdání práce vyplývající z doporučeného časového plánu studia


.....
doc. Ing. Daniel Hanus, CSc.
vedoucí
Ústavu letecké dopravy




.....
prof. Dr. Ing. Miroslav Svítek
děkan fakulty

Potvrzuji převzetí zadání bakalářské práce.


.....
Iliyas Boztayev
jméno a podpis studenta

V Praze dne 24. října 2014

I would like to thank all the people who helped me with my bachelor thesis. In particular, I would like to thank Mgr. Martin Scholtz, Ph.D., who taught me almost everything that I used for working on this thesis. Also I would like to thank him for patience and especially for delicious sandwiches with tea. Then I would like to thank my parents and my girlfriend for permanent moral support and helpful hints in the process of my study.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources and with the help of my supervisor.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical 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 paragraph 1 of the Copyright Act.

In Prague date

signature of the author

Název práce: Simulace dvojrozměrného turbulentního toku pomocí celulárních automatů

Autor: Iliyas Boztayev

Katedra: Ústav letecké dopravy

Vedoucí bakalářské práce: Mgr. Martin Scholtz, Ph.D., Ústav teoretické fyziky, MFF UK

Abstrakt: V práci se studuje simulace dvojrozměrného toku nestlačitelné, ale viskózní kapaliny kolem libovolné překážky, zejména leteckého profilu. V teoretické části podáváme přehled dvou typů hydrodynamických celulárních automatů (HPP a FHP modely) a úvod do jejich statistické analýzy. V části “Výsledky” popisujeme program vytvořený v jazyce C++ implementující FHP model a prezentujeme výsledky, jež byly s jeho pomocí získány. Práce je doplněna dodatky s teorií rotací a s obrázky dokumentujícími provedené simulace.

Klíčová slova: celulární automaty, profil křídla, turbulence, statistická fyzika

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

Author: Iliyas Boztayev

Department: Department of Air Transport

Supervisor: Mgr. Martin Scholtz, Ph.D., Institute of Theoretical Physics, CUNI

Abstract: In this thesis, we study the simulations of two-dimensional flow of the incompressible, but viscous liquid past the obstacle of an arbitrary shape, in particular, past the airfoil. In the theoretical part, we review two types of the lattice-gas cellular automata (HPP and FHP models) and introduction to their statistical analysis. In the part “Results” we describe the program created in the C++ language which implements the FHP model and we present the results obtained with the help of this program. The work is supplemented with appendices devoted to the theory of rotations and with figures and images accompanying the simulations which were performed.

Keywords: cellular automata, airfoil, turbulence, statistical physics

Contents

1	Introduction	7
2	Cellular Automata	11
2.1	Definition	11
2.2	A short history	11
2.3	One-dimensional cellular automata	12
2.3.1	The Laplace operator	13
2.3.2	Diffusion equation	13
2.4	Numerical solution of the diffusion equation	16
2.5	Rules for one dimensional CA	19
2.6	Two-dimensional cellular automata	19
2.7	Lattice gas cellular automata	20
3	Navier-Stokes equations	21
3.1	Tensor transformation	21
3.2	Derivation of the Navier-Stokes equation	25
4	HPP model	29
4.1	General propeties	29
5	FHP model	33
5.1	General propeties	33
6	Statistical physics of cellular automata	37
6.1	Statistical ensembles in classical mechanics	37
6.2	Phase space of cellular automata	39
6.3	Occupation numbers	40
6.4	Emergence of the Navier-Stokes equation	42
7	Results	45
7.1	Two-dimensional flow	45
7.2	Description of the airfoils	46
7.3	Implementation of FHP model	49
7.4	Illustrative examples	50
A	Demonstration notebook	63

B	Symmetries and rotation group	109
B.1	Symmetries in physics	109
B.2	Definition of the group	110
B.3	Linear groups	111
B.4	Orthogonal groups	115
B.5	Explicit form of $\mathbb{SO}(2)$ matrices	119
C	C++ codes	121
C.1	Diffusion	121
C.2	One dimensional CA	122
C.3	FHP model	124

Chapter 1

Introduction

The turbulent flow is one of the most complicated phenomena observed in the Nature. It ranges from the disordered and unpredictable flow of the rivers through Kármán vortices in the Earth's atmosphere to gargantuan clouds of gas near heavy stars in the Universe. From one point of view, turbulent motion is just a direct consequence of the Newton's laws of motion applied to a very complicated system – the fluid consisting of dozens of particles – and our inability to predict the turbulent flow rests merely in the huge complexity. While certainly true, there is something deeper in the unpredictability of turbulence. As the physics and mathematics of the 20th century first shown, the chaotic properties of the system may lead to the emergence of qualitatively new behavior and give rise to the new structures. Thus, considerable effort has been put into the understanding of turbulence and other chaotic systems.

Since the dawn of the physics, it was clear that although the basic principles can be understood perfectly on simple and idealized examples, equation describing real physical systems are usually too difficult to be solved. For this, numerical mathematics has developed many powerful methods for solving differential equations in an approximate, but in principle arbitrarily accurate way. With the appearance of modern computers, many tasks which seemed impenetrable in the past became feasible.

Unfortunately, the Navier-Stokes equation which is believed to govern the turbulence is complicated not only from the physical point of view, but also its mathematical properties are not well understood (compared to other equations of physics) and this equation resists precise numerical calculations quite successfully. This fact is less and less true today, because the machines and numerical techniques are getting more and more advanced, but still it is useful and fruitful to look for alternative approaches.

In this thesis we focus on one particular alternative approach to hydrodynamics and the Navier-Stokes equation. This approach is known as the “lattice-gas cellular automata” and is based on the more general concept of cellular automaton. Again, cellular automata are the efficient tool how to tackle several problems in various areas of mathematics, physics, biology or economy. On the more fundamental level, some people believe that the Nature herself is kind of cellular automaton and that ultimate physical laws will have similar structure.

Our goals in this these are much more modest than to solve this question. Our goals were to understand the basic concepts behind the cellular automata, get an idea (appropriate to bachelor thesis) about mathematical tools for their analysis and implement

appropriate cellular automaton *ab initio*, i.e. without using existing libraries.

As an application, we have chosen to study the turbulent flow past the realistic airfoils of arbitrary shape immersed in the fluid. In order to accomplish this goal, we initially applied our program to the simplest situations involving just a few particles, then we immersed simple obstacles in the fluid (wall, sphere) and checked that our program gives physically reasonable results. After gaining the confidence in our model, we applied it to the airfoils and studied, qualitatively, the emergence of turbulence in the vicinity of the airfoil.

The major advantage of the cellular automaton approach to hydrodynamics compared to standard numerical methods is high computational efficiency and easy parallelizability (spreading the calculation on high number of processors). This advantage is achieved through extremely simple *microdynamics*, i.e. very simple individual interactions between the particles of the fluid. This microdynamics is highly unphysical and in the chapter 5 (devoted to FHP model) we will introduce its simple rules. In addition, the velocity is not a continuous variable nor it is discretized in a usual way. Instead, the velocity is permitted to have only six different directions and its magnitude is fixed to constant (unit) value. Despite this oversimplification of the model, FHP model reproduces the solutions of Navier-Stokes equations on the macroscopic level and, later on, we discuss how this is possible, c.f. chapter 6.

As the program we developed stands, it is incapable of producing real, relevant technical or engineering data, as this would be task very much beyond the possible scope of the bachelor thesis. On the other hand, our program exhibits all defining features of the aforementioned FHP model and basic tools for the collection of data have been implemented. Thus, we believe that in the near future we will be able to extend and expand the program for the purpose of practical applications.

A secondary goal of the thesis was the educational one. In order to achieve the main goal, it was necessary to get acquainted with the C++ language itself and with supplementary tools and procedures, e.g. GNUplot, Mathematica, Maxima, operating system Linux. Last but not least, the thesis was written in the typesetting system \LaTeX which was unknown to the author of the thesis before. Along with these essential software tools, the author had to study new topic from mathematics, physics and statistical physics and non-conventional bit manipulation techniques used in the cellular automata programs. We believe that also this secondary goal has been achieved to appropriate extent.

The organization of the thesis is as follows. In the chapter 2 we introduce the notion of cellular automaton, supplemented with brief history, compare cellular automata to standard discretization of partial differential equations and show how the solution of simple heat equation can be found using standard numerical techniques (in C++). Then we classify all one-dimensional cellular automata according to Wolfram and briefly sketch some differences in two-dimensional case.

Chapter 3 deals with the Navier-Stokes equation, we present a simple derivation of this equation using the tensorial apparatus. In chapter 4 we present the simplest and historically the first lattice-gas cellular automaton, referred to as the HPP model. After discussing its drawback, we turn immediately to a more advanced FHP model in chapter 5.

In chapter 6 we explain the relation of cellular automata and, in particular, the FHP automaton to the solutions of the Navier-Stokes equation. Following closely publication

[29], we show only the main steps in the derivation of macroscopic limit of FHP model, trying to preserve the main idea.

All results are then collected in the chapter 7, where we present images generated by our program. Also we explain the basic properties of airfoils which we used as the obstacles for investigation of the turbulent flow past them. The final images represented in variety of ways for better understanding of the problem can be found in the appendices. In the appendix, also a brief introduction to the theory of rotation groups can be found. The rotational symmetry is a crucial issue behind the theoretical justification of validity of FHP model. For this reason, we have, together with the supervisor, written this text on rotational groups. However, for the lack of space in the thesis, the connection between rotation groups and cellular automata is not spelled in full details; hence, this text has been relegated to the appendix.

Chapter 2

Cellular Automata

2.1 Definition

Cellular automaton (henceforth CA) is mathematical model used in different branches of science and engineering. Generally, CA is a collection of elements which take on k possible values in discrete time and space according to the existing rules. Space can be represented by a regular lattice of sites and update rules in this case are defined by values of neighboring sites. This thesis is not purely mathematical but we believe that mathematical precision in certain cases is necessary and, thus, we introduce also the mathematical ideas behind the cellular automata and geometry. In our treatment of CA, we follow publications [29] with reference to [14].

In general case, the space can be represented by D -dimensional Euclidean lattice of sites $\mathcal{L} = \mathbb{Z}^D$. Finite set of possible states of each cell will be denoted by Q , $Q = \{q_1, q_2, \dots, q_k\}$, where q_k the one of the state. As noted above, update rules for the considered site are defined by values of neighboring sites. The coordinates of neighboring sites n form the neighborhood set $N \subset \mathbb{Z}^D$, $n \in N$. N includes also the coordinates of considered site.

A mapping $f : N \mapsto Q$ is called a *local configuration*. Put it in other way, we assign one possible state to the site from N , $f(n) = q_k$. The evolution of a cell is completely determined by its *local rule* $r : Q^N \mapsto Q$, where Q^N is the set of all mappings $f : N \mapsto Q$, i.e. the values set of neighboring sites and value of considered site at time t determine the value of considered site at time $t + 1$. In more detail, Q^N can be written as $Q^N = \{f : N \mapsto Q\} = \{\{n_1, q_k\}; \{n_2, q_k\}; \dots; \{n_r, q_k\}\}$, where r is, for example, neighborhood range (see below), q_k is value we assign by mapping f .

The global configuration of a CA at a certain time is called a *global configuration* \mathbf{g} . The global configuration \mathbf{g} at time t leads to a new global configuration \mathbf{g}' at time $t + 1$ whereby all cells enter a new state according to the local rule *synchronously*. Recall that time in CA is discrete.

2.2 A short history

First people who proposed the concept of cellular automata were John von Neumann and Stanislaw Ulam in the 1940's. John von Neumann proposed a self-reproducing cellular automaton [26] which at the same time realized a universal Turing machine [24]. In order

to accomplish these ends, Stanislaw Ulam advised von Neumann to use more abstract mathematical lattice vortex model, similar he used in his studies the growth of crystals. Thus, through the Neumann-Ulam team work was to create the first cellular automaton. This CA had a two-dimensional orthogonal lattice. They used Moore neighborhood (see below) with 29 states per cell.

The most popular version of two-dimensional CA is the *Game of life* introduced by John H. Conway in 1970s and popularized by Martin Gardner [9]. Each cell has two possible states: alive or dead. The update rules set includes:

1. cell stays alive if the cell has two or three living neighbours;
2. dead cell becomes alive if the cell has exactly 3 living neighbours;
3. alive cell becomes dead if the cell has fewer than 2 living neighbours;
4. alive cell becomes dead if the cell has more than 3 living neighbours.

In 1973, the so-called *HPP model* was proposed by Hardy, de Pazzis and Pomeau [11]. It was the first lattice gas cellular automata (LGCA) with the purpose to simulate the fluid flow and other physical problems. Unfortunately, this model does not lead to the Navier-Stokes equation in the macroscopic limit.

In 1986, it was discovered that a CA over a lattice with hexagonal symmetry leads to the Navier-Stokes equation in the macroscopic limit. This model was introduced by Frisch, Hasslacher, and Pomeau in the scientific paper [7] and was named FHP model, according to the initials of these three authors. The theoretical foundations of lattice gas automata were given soon after in [22] and [8].

2.3 One-dimensional cellular automata

The one-dimensional cellular automata consist of uniform sites or *cells* arranged in a row where each cell i has the finite number of states(values) k , at any time t . We designate the state of i -th cell at time t by symbol a_i^t . Each cell changes its state in time, in general, by the rule

$$a_i^t = F[a_{i-r}^{t-1}, \dots, a_i^{t-1}, \dots, a_{i+r}^{t-1}],$$

which is to say that the new cell state depends only on the state of the i -th cell and the r (range) neighbors to the left and right at the previous time level $t - 1$. [29, page 18]. F in this case, is called *update rule*. An alternative formulation of the update rule reads

$$a_i^t = f \left[\sum_{j=-r}^{j=r} \alpha_j a_{i+j}^{t-1} \right], \tag{2.1}$$

where the α_j are integer constants. [29, page 18]

The simplest of one-dimensional CA would be with two possible states per cell like the figure 2.1 shows. For example, very simple update rule for this CA can be given like

$$a_i^t = (a_{i-1}^{t-1} + a_{i+1}^{t-1}) \bmod 2 .$$

Mod 2 indicates that the remainder 0 or 1 after the division by 2 is taken. [21]

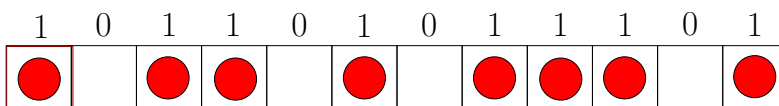


Figure 2.1: One-dimensional lattice

Lattice gas cellular automata is one of CA models which can be used for partial differential equations solving. Now we will show that LGCA can be used as discrete model of partial differential equations. As an example, consider the *diffusion equation*. We begin with the definition of the *Laplace operator*.

2.3.1 The Laplace operator

In many areas of physics and mathematics (and in particular in the fluid dynamics) we often encounter the so-called Laplace operator. The Laplace operator is a generalization of the second derivative for functions of many variables. In two dimensions, it is defined by

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}.$$

This notation means that, applied to any function $f(x, y)$ the Laplacian of f is

$$\Delta f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}.$$

Clearly, the Laplace operator in one dimension is simply

$$\Delta f = \frac{\partial^2 f}{\partial x^2}.$$

Thus, in one dimension with coordinate x , the Laplace operator is simply second partial derivative with respect to x . For this reason we occasionally write Δ instead of $\frac{\partial^2 f}{\partial x^2}$ even in one dimension.

2.3.2 Diffusion equation

Suppose we have a rod which is able to conduct the heat. Let the length of the rod be L and we introduce coordinate x along the rod. Since we assume that the rod can conduct the heat, we can associate the temperature with each point of the rod. This temperature will be described by function

$$u = u(x, t),$$

where t is time and x is the position on the rod.

In other words, each point of the rod with coordinate x has temperature $u(t, x)$ at time t . We will suppose that the rod has some initial temperature at time $t = 0$. The initial temperature is represented by a given function $u_0(x)$, so that

$$u_0(x) = u(x, 0)$$

We emphasize that function u_0 must be given and comprises the so-called *initial conditions* for the problem of heat conduction. The question is: suppose we have an initial distribution of the temperature at time $t = 0$ given by function u_0 . How will this distribution evolve in time? Can we find the temperature of the rod in arbitrary point at arbitrary later time $t > 0$? Well, we need an equation which describes this process. The equation is called the heat equation and it can be derived quite rigorously using basic physical assumptions. Here we state the results. The heat equation is

$$\frac{\partial u}{\partial t} = \alpha \Delta u,$$

where Δ is the Laplace operator in one dimension.

We know that the derivative of $f(x)$ is

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h},$$

but computer is not be able to work with the limits, hence we will work with equation

$$f'(x) = \frac{f(x+h) - f(x)}{h},$$

where h will acquire a very small value.

If we relax the limit, we will work with discrete values, so we can write

$$u(t, x) \rightarrow u_{i,j} = u(t_i, x_j),$$

where i and j are indices acquiring discrete values.

For simplicity, constant α in our heat equation will be set to 1, so we have

$$\frac{\partial u}{\partial t} = \Delta u, \tag{2.2}$$

or, equivalently

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}.$$

Now, let us find the discretized version of the diffusion equation (2.2). Derivative with respect to time reads

$$\frac{\partial u}{\partial t} = \frac{u_{i+1,j} - u_{i,j}}{h} = (u_t)_{i,j}.$$

Similarly, the derivative with respect to coordinate x is approximated by the difference equation

$$\frac{\partial u}{\partial x} = \frac{u_{i,j+1} - u_{i,j}}{k} = (u_x)_{i,j},$$

where k is the constant analogous to h . However, we need the second derivative with respect to coordinate x ,

$$\frac{\partial u^2}{\partial x^2} = \frac{(u_x)_{i,j+1} - (u_x)_{i,j}}{k},$$

Expanding this relation, we find

$$(u_{xx})_{i,j} = \frac{u_{i,j+2} - 2u_{i,j+1} + u_{i,j}}{k^2}. \quad (2.3)$$

Please note that, for evaluating the second derivative, we need values of our function at the points with the distance 2. In order to make expression (2.3) more symmetric with respect to the point at which the derivative is evaluated, we approximate the second derivative by

$$(u_{xx})_{i,j} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2}.$$

Next we rewrite the equation of heat using approximate expressions for the derivatives,

$$\frac{u_{i+1,j} - u_{i,j}}{h} = \frac{u_{i,j+2} - 2u_{i,j+1} + u_{i,j}}{k^2},$$

or, after obvious arrangements,

$$u_{i+1,j} = u_{i,j} + \frac{h}{k^2}(u_{i,j+2} - 2u_{i,j+1} + u_{i,j}).$$

It can be shown that the ratio of h/k^2 must be smaller than 0.5, otherwise the behavior of the solution is very difficult to predict, as we show in the next section.

Let us write the equation symmetrically with respect to the cell i ,

$$u_{i,j} = u_{i-1,j} + \frac{h}{k^2}(u_{i-1,j+1} - 2u_{i-1,j} + u_{i-1,j-1}), \quad (2.4)$$

which can be brought into the form

$$u_{i,j} = f\left[\sum_{k=-1}^{k=1} \alpha_j u_{j+k}^{i-1}\right].$$

This equation is of the same form as the equation (2.1) which defines the update rule. [29, page 21]

As noted above, only special types of cellular automata, for example LGCA provide discrete models for partial differential equations. The connection between the differential equations and lattice gas automata is not formal but deeply rooted in the ground of conservation laws. [29, page 21]

2.4 Numerical solution of the diffusion equation

In this section we briefly present numerical solution of the diffusion equation (2.2). The purpose is to illustrate the behavior of discretized differential equations so that we can appreciate differences between cellular automata and differential equations later on.

Equation (2.2) does not have a unique solution and in order to obtain one, we have to specify the boundary and the initial conditions. As we explained in section 2.3.2, the rod is assumed to have some initial distribution of the temperature at the beginning, this distribution being provided by function

$$u(0, x) = u_0(x). \quad (2.5)$$

Without the loss of generality we suppose that the length of the rod is $L = 2$ and its center is located at $x = 0$, so that the rod coincides with the interval

$$x \in [-1, 1]$$

on the real axis. Equation (2.5) then represents the initial conditions for the diffusion equation.

Mathematically, the need for specifying the initial conditions can be seen from the discretized version of the diffusion equation (2.4). The value of temperature at (discrete) time i at the position j is determined by the values at earlier time $i - 1$. Since we cannot continue infinitely to times $i - 2, i - 3, \dots$, there must be a time in which the temperature is given. For convenience, we choose $i = 0$ as the initial time.

Equation (2.4), however, shows yet another feature. If the left end of the rod corresponds to the position $j = 0$, then, according to (2.4), the temperature of this point at time i is determined by temperatures (at time $i - 1$) at points $j = 1, j = 0$ and $j = -1$. The point $j = -1$ does not lie in the rod and so the diffusion equation, in fact, does not tell us how the temperature of the end points will evolve in time. We have to supply also the *boundary* conditions, i.e. we have to prescribe the temperature of the end points at every time.

There are two types of boundary conditions we usually impose. Fixing the temperature on the end points corresponds to the choice of the so-called *Dirichlet boundary conditions*. Another possibility is to fix the flux of the heat. In this case the temperature can vary at the endpoints, but the amount of outgoing heat is prescribed. These conditions are called *von Neumann boundary conditions*. In our illustration we deal with the Dirichlet boundary conditions only.

Our simple program written in C-language works as follows. We consider grid (matrix) u_{ij} of dimension $M \times N$, the indices i and j acquire values

$$i = 0, 1, \dots, M - 1, \quad j = 0, 1, \dots, N - 1. \quad (2.6)$$

The matrix element u_{ij} is the temperature of the rod at time i at the position j . The actual position represented by index j is

$$x_j = -1 + j \frac{2}{N - 1}, \quad (2.7)$$

so that $x_0 = -1$ and $x_{N-1} = 1$ are the endpoints of the rod.

Let us turn to the initial and boundary conditions. First we define function

$$u_0 : [-1, 1] \mapsto \mathbb{R}^+, \quad (2.8)$$

which defines the initial distribution of the temperature. Then the values $u_0(-1)$ and $u_0(1)$ serve as fixed temperatures at the endpoints of the rod, i.e. we prescribe Dirichlet boundary conditions

$$u_{i,0} = u_0(-1), \quad u_{i,N-1} = u_0(1). \quad (2.9)$$

After specifying the initial and boundary conditions, we solve the equation on entire grid $u_{i,j}$ and corresponding C-code is listed in the appendix C.1.

To see how the diffusion equation works, let us start with the very sharp initial profile of the temperature distribution as shown in figure 2.2. This profile is given by function

$$u_0(x) = e^{-x^2/100}, \quad (2.10)$$

so that it represents (not normalized) Gaussian curve. In the middle of the rod ($x = 0$), the curve acquires its maximum value 1 and in the neighborhood it decreases rapidly, so that it effectively reaches zero at the endpoint. Using the Dirichlet boundary conditions, the endpoints are held at zero temperature during the calculation. Here, zero temperature does not mean literally that the endpoints have temperature of absolute zero, 0 kelvins. Notice that the diffusion equation (2.2) contains only the derivatives of u and so if some u is a solution, then any $u + \text{constant}$ is also a solution. Thus, from any solution u with boundary temperature t_0 we can form solution $u - t_0$ with zero boundary temperature and vice versa.

Physically we expect that since the rod is hot in the middle and cold on the boundaries, the heat will flow from the middle towards the ends of the rod. In other words, the temperature in the middle of the rod will be decreasing and the temperature of the nearby points will increase. But, since the endpoints have constant zero temperature, there will always be a flux of the heat through the endpoints, the heat will dissipate, and in the final state the temperature will be zero everywhere. Evolution in figure 2.2 shows that this is indeed the case. The initially sharp profile gets wider and wider and the maximum decreases until the temperature vanishes everywhere on the rod.

Another example is shown in figure 2.3 and we leave the physical interpretation of this process to the reader.

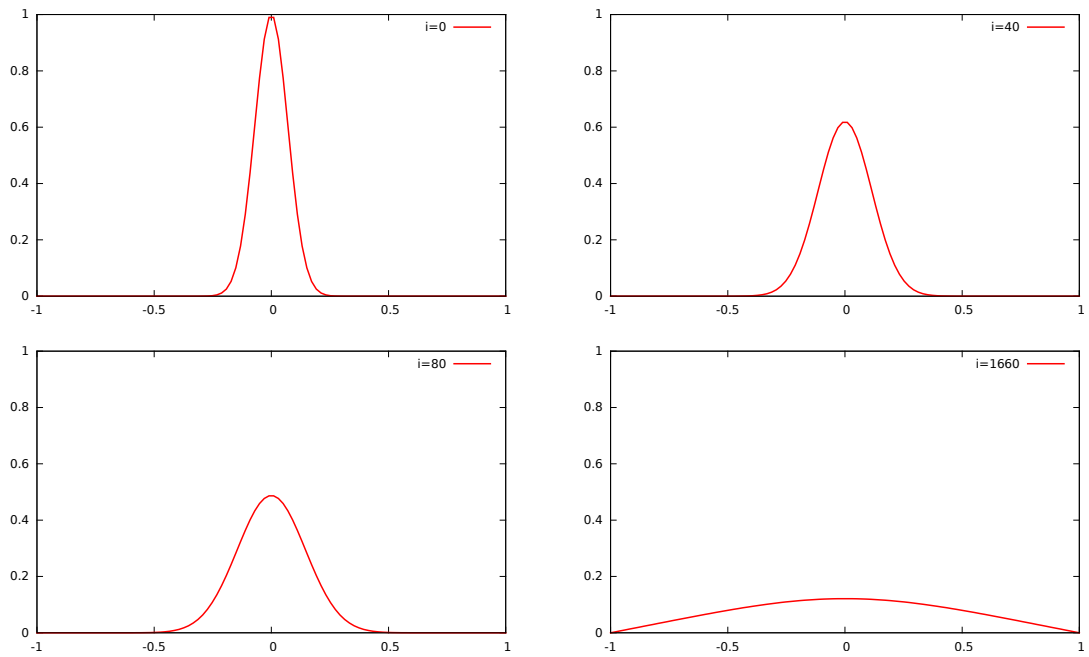


Figure 2.2: Solution of the diffusion equation for very sharp initial profile.

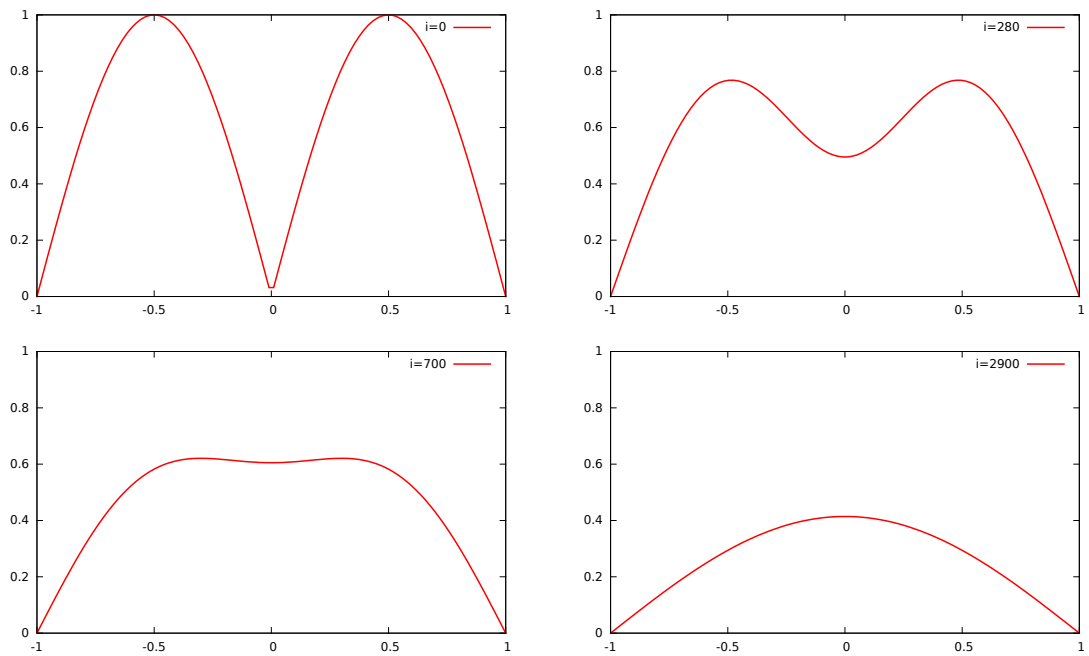


Figure 2.3: Solution of the diffusion equation for different initial profile.

2.5 Rules for one dimensional CA

Since the cellular automaton is a discrete and finite system, it has only finite number of possible states. In the case of one-dimensional cellular automaton with two states and such that each cell interacts only with its two neighbors, it is easy to calculate the total number of possible automata. By definition, the updated state of the cell depends on three binary numbers: 1 state of the cell itself and two states of neighboring cells. Three cells can be in $2^3 = 8$ possible different states. To each triple of cells we can assign two possible values of the updated state which gives us $2^8 = 256$ possible cellular automata. The rules for cellular automata can be encoded into single integer r , $0 \leq r \leq 255$, in a following way.

Let r be such an integer. Its binary representation has 8 digits, i.e.

$$r = r_7 r_6 r_5 r_4 r_3 r_2 r_1 r_0|_2, \quad \text{e.g. } 123 = 111101|_2. \quad (2.11)$$

Now, we have eight possible states of the triple of the cells, where the middle cell is the one we want to update and remaining two cells are its neighbors. Let's denote them as

$$0 = 000, \quad 1 = 001, \quad 2 = 010, \quad 3 = 011, \quad 4 = 100, \quad 5 = 101, \quad 6 = 110, \quad 7 = 111. \quad (2.12)$$

If we understand the number r as the rule for cellular automaton, we interpret its binary digit r_i as a new updated state of the middle cell represented by number i . or example, if $r = 123$, we have

$$\begin{array}{cccc} 000 \xrightarrow{r_0} 1, & 001 \xrightarrow{r_1} 0, & 010 \xrightarrow{r_2} 1, & 011 \xrightarrow{r_3} 1, \\ 100 \xrightarrow{r_4} 1, & 101 \xrightarrow{r_5} 1, & 110 \xrightarrow{r_6} 1, & 111 \xrightarrow{r_7} 1. \end{array}$$

In the scheme above, on the left hand side of each assignment we have the triple of cells and we want to update the state of the middle cell. The state of the triple is replaced by a number i from 0 to 7 according to (2.12). The digit r_i is then a new state of the cell in the middle. In this way, number r determines the cellular automaton completely.

All 256 automata are shown in figures C.1–C.8. We plotted these figures using the program in C++ and script in GNU-plot. The code and illustration are listed in the appendix C.2.

2.6 Two-dimensional cellular automata

The two-dimensional cellular automaton is more complicated model because we can use various kinds of cell's form and choose between other configurations of neighborhoods. Among many different choices of the cell neighborhoods, the two are the most common. In general we can define *Von Neumann neighborhood* of range r

$$N_{i,j} = \{(k, l) \in \mathcal{L} \mid |k - i| + |l - j| \leq r\}$$

and the *Moore neighborhood* of range r

$$N_{i,j} = \{(k, l) \in \mathcal{L} \mid |k - i| \leq r \wedge |l - j| \leq r\},$$

where i, j are coordinates of a cell for which we find a neighborhood, k, l are coordinate of neighboring cells. In the figure 2.4, we show the neighborhoods type for $r = 1$.

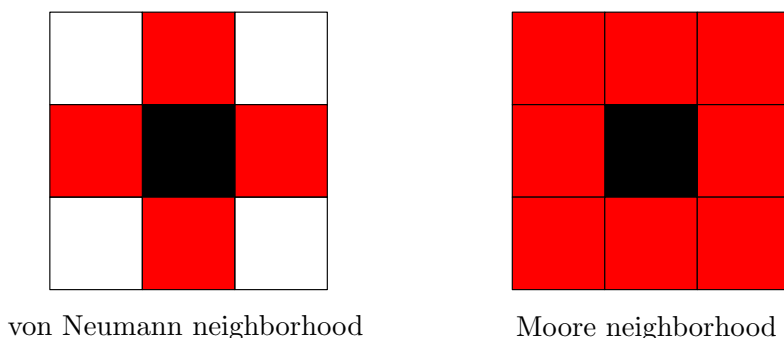


Figure 2.4: Kinds of neighborhoods

As has been noted above, the one example of 2D CA is a “Game of Life” introduced by Conway in 70s. This model of CA (Moore neighborhood) represents an elegant example of self-reproducing system with simple rules. This game was first published by Martin Gardner in the issue of *Scientific American* [9]. Currently, many of patterns which can be found in this model are used in other, non-mathematical disciplines, like sociology, biology, chemistry etc.

2.7 Lattice gas cellular automata

Knowing all of the above, we can ask what properties should have a cellular automaton in order to simulate real physical processes, in particular for the flow simulation. We define the basic patterns which cellular automaton must comply to [29]

1. The Navier-Stokes equation, for example, expresses the conservation of mass and momentum. [29, page 36] The cellular automata used for simulation should hold corresponding conserved quantities. [29, page 36]
2. Required model of CA must enable transport of information similar to a non-equilibrium physical phenomena.
3. The desired physical behavior of a lattice-gas cellular automata will show up in the macroscopic limit which can be derived from a theory of statistical mechanics on a lattice. The application of certain concepts of statistical mechanics requires that the microdynamics, i.e. the update rules, are reversible. [?, page 37]

Programmed cellular automaton that meets all of these criteria is a very complicated problem. In order to meet these requirements, LGCA upgrade process is divided into two stages, *collision* and *propagation*. Collision, by analogy with update rules of CA, sets a new cell value based on the values of neighboring cells, and propagation ensures distribution of system properties. Consider some CA model which, more or less, meets all of these criteria.

Chapter 3

Navier-Stokes equations

3.1 Tensor transformation

In virtue of symmetries and rotation group theory which we considered we can proceed to describe of the processes occurring in liquids and gases. In hydrodynamics to describe the motion of fluid used the system of differential equations called the Navier-Stokes equations named after Claude-Louis Navier and George Gabriel Stokes. Given system of equations for an incompressible fluid consists of the *equations of motion* and *continuity equations*. Existence of smooth solution of the Navier-Stokes equations is on the list of the so-called *millennium prize problems*. The key problem in solving the equation is its nonlinear nature.

First let's consider the isotropic situation where we represent the fluid by velocity field $\mathbf{V} = \mathbf{V}(t, \mathbf{r})$. For fluid of rest $\mathbf{V} = 0$. Isotropy property means uniformity in all direction, in simple terms we can not define the distinguished orientation in the liquid. Assume the plate with area $d\mathbf{S}$ in a isotropic liquid, where $d\mathbf{S} = \mathbf{n}dS$. In the picture 3.1 the plate is shown schematically, where the wells of the vessel has no influences isotropy. Consider the forces operating on the plate.

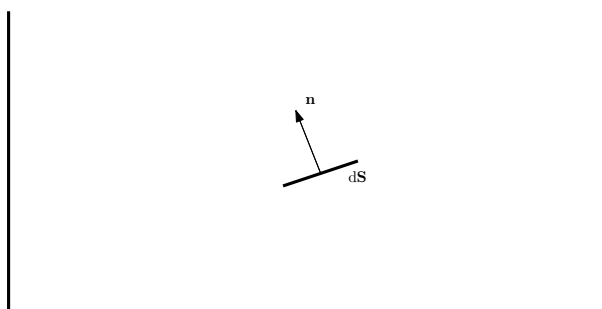


Figure 3.1: Distinguished direction

We can define the one distinguished direction \mathbf{n} in the isotropic liquid which is irrelevant with environment properties. Accordingly we can write

$$d\mathbf{F} \sim d\mathbf{S} \sim \mathbf{n}, \tag{3.1}$$

where the proportionality constant is pressure P with the opposite sign, because the pressure acting in opposite direction. We have

$$P = \frac{dF}{dS}, \tag{3.2}$$

$$d\mathbf{F} = -P d\mathbf{S}, \tag{3.3}$$

or using the isotropic tensor we can write by the components

$$dF_i = -P \delta_{ij} dS_j. \tag{3.4}$$

Let us consider anisotropic situation. In the picture 3.2 is schematically shown the plate in the anisotropic liquid. Anisotropy is represented by streamline in relation to which the velocities are tangential.

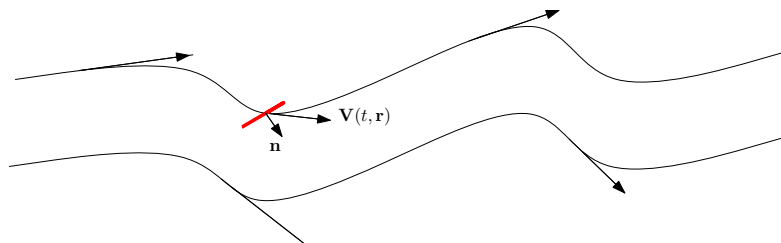


Figure 3.2: Anisotropic situation

In this case the pressure is dependent on the plate orientation, so we cannot express the pressure as in the isotropic situation. We must express the pressure otherwise. Let us consider the element of liquid in the form of a tetrahedron. 3.3

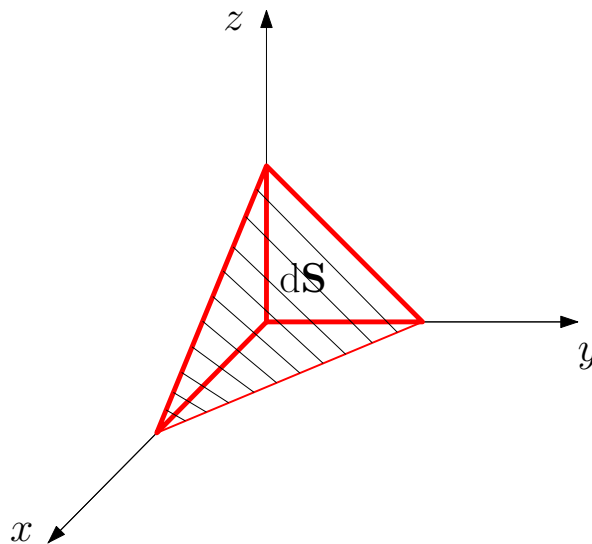


Figure 3.3: Element of liquid

For convenience in operation we select the element of liquid first and then select the coordinate system. In an anisotropic situation we can resolve the force F which acts on the front side into components in our coordinate system, see figure 3.4.

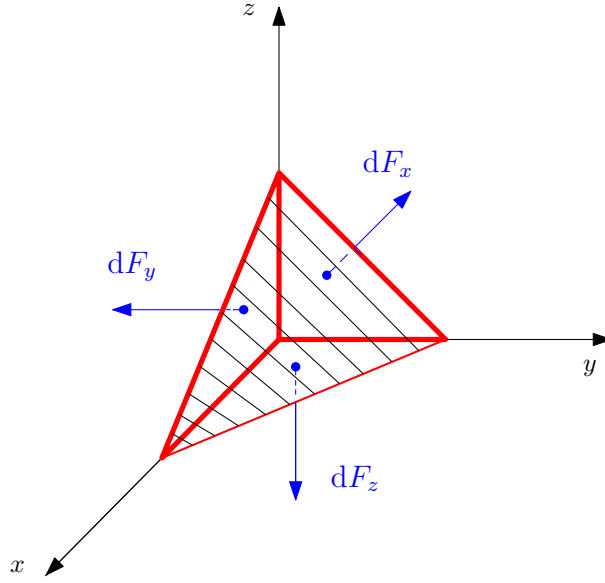


Figure 3.4: Decomposition of the force $d\mathbf{F}$

In this case the force F reads

$$dF = dF_x + dF_y + dF_z, \quad (3.5)$$

or using matrix form

$$F = \begin{pmatrix} F_x & 0 & 0 \\ 0 & F_y & 0 \\ 0 & 0 & F_z \end{pmatrix} \quad (3.6)$$

Note, in our selected coordinate system the matrix is diagonal. Also we have the relation $\mathbf{S} = \mathbf{n} dS$, where \mathbf{n} has form

$$\mathbf{n} = (n_x, n_y, n_z). \quad (3.7)$$

Using it we can represent the pressure by components

$$\sigma_x = \frac{dF_x}{dS_x}, \quad \sigma_y = \frac{dF_y}{dS_y}, \quad \sigma_z = \frac{dF_z}{dS_z}. \quad (3.8)$$

Consequently the force components reads

$$dF_x = \sigma_x dS_x, \quad dF_y = \sigma_y dS_y, \quad dF_z = \sigma_z dS_z, \quad (3.9)$$

or acquiring the matrix of stress tensor σ

$$\sigma = \begin{pmatrix} \sigma_x & 0 & 0 \\ 0 & \sigma_y & 0 \\ 0 & 0 & \sigma_z \end{pmatrix}, \quad (3.10)$$

we can write

$$dF_i = \sigma_{ij} dS_j. \quad (3.11)$$

So far, we have employed the coordinate system adapted to the selected element, i.e. we have chosen the axes so that the coordinate planes coincided with the faces of the volume element. However, the orientation of the volume element will vary from point to point so we have either to introduce curvilinear coordinates adapted to the volume element at each point, or, if we want to keep single Cartesian coordinate system, we have to find description of the element which is not aligned with the coordinate axes. Here we choose the latter possibility.

It is clear that, at each point (x, y, z) , we *can* introduce a local Cartesian coordinates (x', y', z') aligned to volume element at that point. Any two Cartesian systems at given point are related by a rotation. For a comprehensive review on rotations and their group properties, see appendix B. Hence, there exists a rotation which sends one Cartesian system into the other one,

$$R : (x', y', z') \mapsto (x, y, z) \quad (3.12)$$

Since we already know description of the stresses in aligned coordinates in terms of diagonal matrix σ_{ik} , and we know that the force is a vector and transforms in a definite way under rotations, we can write

$$dF'_i = R_{ij} dF_j = R_{ij} \sigma_{ij} dS_k = R_{ij} \sigma_{jk} R_{lk} dS'_l, \quad (3.13)$$

where dF'_i is force component in our coordinate system and dF_j is force component in element's coordinate system. Then we have

$$R_{ij} \sigma_{jk} R_{lk} dS'_l = \sigma'_{il} dS'_l. \quad (3.14)$$

Thus, we see that, under the rotation, the components of the matrix σ_{ik} transform as

$$\sigma'_{il} = R_{ij} \sigma_{jk} R_{lk}.$$

This relation actually says that σ_{ik} behaves as a *tensor* under rotations. Using the orthogonality of the rotation matrix R_{ik} , c.f. (B.36), we can invert the last relation to find

$$\sigma_{ik} = R_{ji} \sigma'_{jl} R_{lk}. \quad (3.15)$$

To conclude, we have shown that the matrix σ_{ik} is in fact the matrix of the components of the tensor. This tensor is called *stress tensor* and described the pressure exerted on the infinitesimal surface of arbitrary orientation.

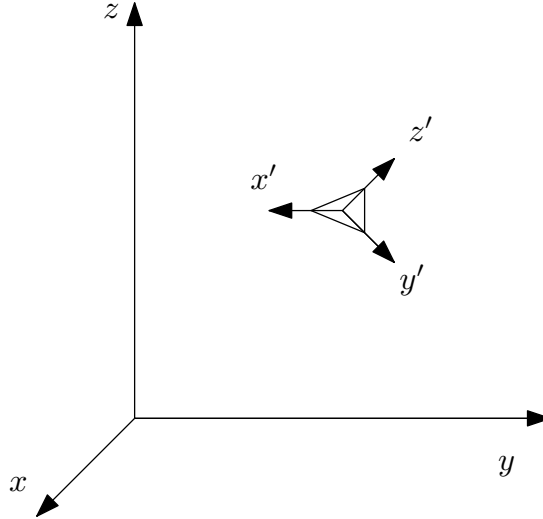


Figure 3.5: Two coordinate systems

3.2 Derivation of the Navier-Stokes equation

For the derivation of the Navier-Stokes equation we use the following theorem. *If law of conservation of angular momentum applies then $\sigma_{ij} = \sigma_{ji}$.* Let's consider Newton's second law

$$\mathbf{F} = m \mathbf{a}. \quad (3.16)$$

Divided both sides of the equation by V we have the equations of motion for the continuum,

$$\mathbf{f} = \rho \frac{d\mathbf{V}}{dt}, \quad (3.17)$$

where f is the force density defined by

$$\mathbf{f} \equiv \frac{d\mathbf{F}}{dt}.$$

We are concerned with the surface forces acting on the liquid. For the surface force we can find the relation

$$f_i = \frac{\partial \sigma_{ik}}{\partial X_k}, \quad (3.18)$$

Note, that the force (force density) does not depend just on the value at one point, but depends on point-to-point change of the stress tensor. Substituting the surface forces into the equations of motion (3.17) we find

$$f_i = \rho \frac{dV_i}{dt} = \frac{\partial \sigma_{ik}}{\partial X_k}. \quad (3.19)$$

Let us consider the isotropic liquid. In this case, the stress tensor should be isotropic,

$$\sigma_{ik} = A \times \delta_{ik}, \quad (3.20)$$

where A can be constant or function to be determined. We know that

$$dF_i = -PdS_i = -P\delta_{ij}dS_j \Rightarrow \sigma_{ik} = -P\delta_{ik}, \quad (3.21)$$

substitute the stress tensor in the motion equation

$$\rho \frac{d\mathbf{V}}{dt} = \frac{\partial \sigma_{ik}}{\partial X_k} = -\frac{\partial}{\partial X_k}(P\delta_{ik}) = -\frac{\partial P}{\partial X_i}, \quad (3.22)$$

or in a vector form

$$\rho \frac{d\mathbf{V}}{dt} = -\text{grad } P. \quad (3.23)$$

This equation is known as *Euler equation of ideal liquid*. It is one of the fundamental equations of hydrodynamics. However, because of the isotropy assumed in this equation, we have neglected the viscosity of the fluid. The viscosity is an essential feature of real fluid and, therefore, the Euler equation (6.34) describes unreasonably idealized situation. It can be shown, for example, that if the initial conditions for the Euler equation do not contain vortices, they will never appear during the time evolution of the fluid driven by the Euler equation. Thus, although the Euler equation can describe whirling fluid (if the initial conditions are chosen appropriately), it cannot describe the transition from the laminar flow to the turbulent one.

Another property of the fluid we have imposed above is the incompressibility. This is also a restriction because real fluid *are* compressible and this compressibility is apparent when the speed of the fluid compares to the speed of the sound. On the other hand, in the regime of the small speed, the compressibility of the fluids is negligible and since, in practice, we deal with fluids moving with small velocities, the assumption of the incompressibility is very accurate. In other other words, it is not the issue of compressibility which makes the Euler equation useless; it is the assumption of zero viscosity.

Much more realistic description of the flow is provided by the Navier-Stokes equation to be derived now. In the presence of viscosity, the fluid at motion cannot be regarded as isotropic environment anymore. (While both viscid and inviscid fluids at rest *are* isotropic) In order to take the viscosity into account, we have to modify the stress tensor (3.20). This can be done either on the phenomenological or microscopic level. In the latter approach, we would have to look at individual molecules constituting the fluid, calculate the interactions between them and impose some reasonable approximation in order to find effective friction between the layers. Here we sketch briefly the construction of the Navier-Stokes equation in a phenomenological way.

First, the stress tensor for the fluid at rest must reduce to the isotropic stress tensor (3.20) and, hence, it will be of the form

$$\sigma_{ik} = -P \delta_{ik} + \text{viscous terms.} \quad (3.24)$$

The origin of the viscosity lies in the effect of friction between the neighboring layers of the fluid moving with different velocities. If the two layers are moving with the same speed,

the friction is supposed to vanish. That means that the stress tensor cannot depend on the velocities themselves but rather on the rate of change of the velocity as passing from one layer to another one. In other words, it must depend on the derivatives of the velocity field, i.e. it must be composed of the tensor $\partial_i V_k$. However, the aforementioned general theorem states that the stress tensor must be symmetric, while the tensor $\partial_i V_k$ is not¹. In order to get symmetric tensor, we have to form the symmetric combination

$$\partial_i V_k + \partial_k V_i. \quad (3.25)$$

It is obvious that such object is indeed symmetric, for the interchange of the indices $i \leftrightarrow k$ yields

$$\partial_k V_i + \partial_i V_k = \partial_i V_k + \partial_k V_i. \quad (3.26)$$

Based on these considerations, we postulate the stress tensor in the form

$$\sigma_{ik} = -P \delta_{ik} + \xi (\partial_i V_k + \partial_k V_i), \quad (3.27)$$

where ξ is a constant. The value of constant ξ can be, in principle, computed from the microscopic model or, more easily, it can be measured in the experiment.

For the derivation of the Navier-Stokes equation in which viscosity of liquid is taken into consideration, we should consider the anisotropic case. In this case stress tensor have the form

$$\sigma_{ik} = P \delta_{ik} + \xi (\partial_i V_k + \partial_k V_i). \quad (3.28)$$

Inserting this tensor into the equations of motion (3.19) for the incompressible liquid, i.e. $\rho = \text{constant}$, which implies $\text{div} \mathbf{V} \equiv \partial_i V_i = 0$, we write the Navier-Stokes equation in the form

$$\rho \frac{dV_i}{dt} = \frac{\partial \sigma_{ik}}{\partial X_k} = -\frac{\partial}{\partial X_k} (P \delta_{ik} + \xi (\partial_i V_k + \partial_k V_i)) = -\frac{\partial P}{\partial X_i} + \xi \partial_k \partial_k V_i. \quad (3.29)$$

The operator $\partial_k \partial_k$ which appeared on the right hand side is, in fact, the Laplace operator

$$\partial_k \partial_k = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \equiv \Delta, \quad (3.30)$$

so that the Navier-Stokes equation can be rewritten in the index-free notation as

$$\rho \frac{d\mathbf{V}}{dt} = -\text{grad} P + \xi \Delta \mathbf{V}. \quad (3.31)$$

¹Symmetric tensor must satisfy $\sigma_{ik} = \sigma_{ki}$. This property is not exhibited by $\partial_i V_k$, i.e. $\partial_i V_k \neq \partial_k V_i$. For example,

$$\frac{\partial V_x}{\partial y} \neq \frac{\partial V_y}{\partial x}.$$

Since, by assumption, the fluid is incompressible, density ρ is a constant function and hence not affected by differential operators and we can multiply the Navier-Stokes equation with ρ^{-1} , so that we obtain

$$\frac{d\mathbf{V}}{dt} = -\text{grad } p + \nu \Delta \mathbf{V}, \quad (3.32)$$

where we have defined the *kinematic pressure* p and the *kinematic viscosity* ν by

$$p = \frac{P}{\rho}, \quad \nu = \frac{\xi}{\rho}. \quad (3.33)$$

Chapter 4

HPP model

4.1 General properties

HPP model was proposed in 1973 by Hardy, de Pazzis and Pomeau [29]. It is the first model of the lattice-gas group models of CA(LGCA) over a square lattice. Each site of the lattice can contain maximally four particles, where only the one particle is associated with the only one of four possible direction. This property is called *exclusion principle* and it is characteristic for all lattice-gas cellular automata. Consequently each site of lattice can be in one of sixteen different states. We will represent it by four bit word, where "1" means presence of the particle and "0" is absence. All particles in LGCA have the same mass m ($m = 1$) and are indistinguishable. [29]

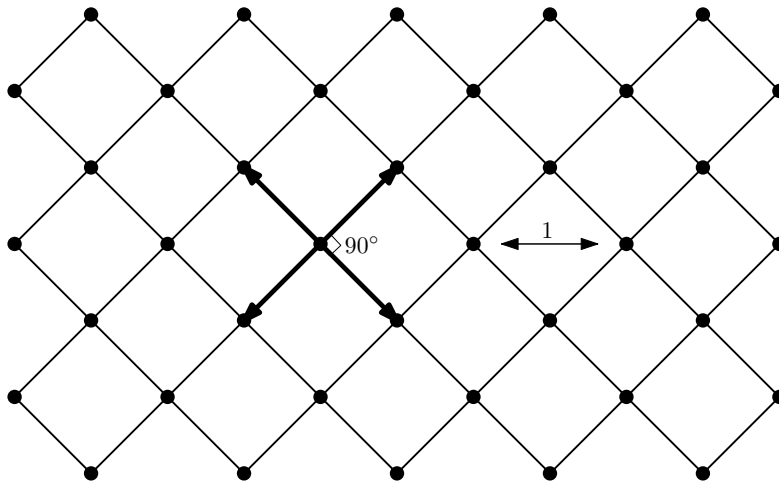


Figure 4.1: Lattice of nodes in HPP model.

It is convenient to split the update of LGCA into two steps: *collision* and *propagation*. The direction of particles *propagation* is represented by *lattice vectors* \mathbf{c}_i (i from 1 to 4). These vectors can be called the *lattice velocities* because velocities are given by the lattice vectors divided by the time step δt which is always set equal to 1 [29].

The collision should conserve mass and momentum while changing the occupation of the cells. For HPP, there is only one *collision* configuration when exactly two particles with opposite directions meet at the node. After collision, the directions of particles

momenta are rotated by 90° . The fact that only two particles participate in the collision leads to the conservation of difference in the number of particles with opposite momenta. In comparison with the mass and momentum conservation this invariant has no counterpart in the real world and is named the *spurious invariant*.

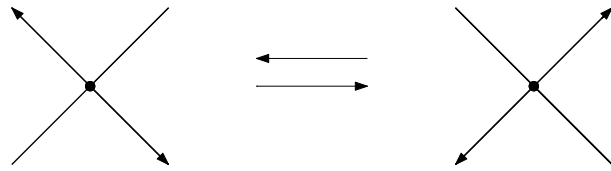


Figure 4.2: Allowed collisions in HPP model.

In the figure 4.3, the update process is shown. Part a) describes the initial state of the collision where we have two particles with the opposite momenta; b) is the collision step when the particles change their momenta; c) propagation process after the collision.

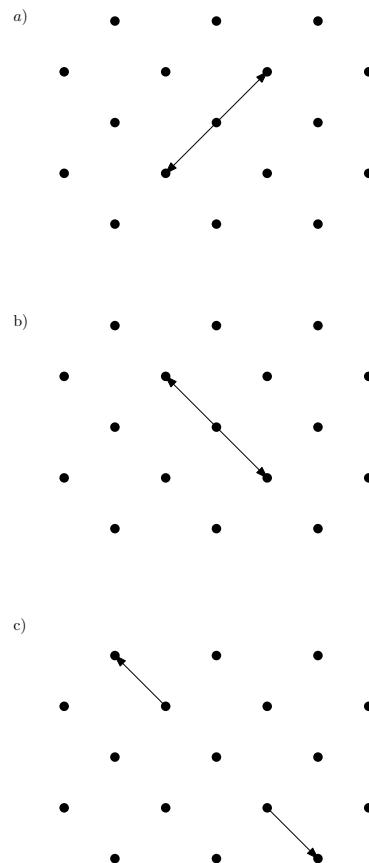


Figure 4.3: Update and propagation process of HPP model

Placing particles in 40×40 matrices in random order, in another words, to assign a number from 0 to 16 to each cell (that means absence of particles for 0 and four particles simultaneously for 16), we can observe the so-called “speckle”, as it is called, see figure 4.4. On the right-hand side of each matrix is shown the bar in which, according to color graduation, we can determine the number of particles in each cell. Unfortunately, reader

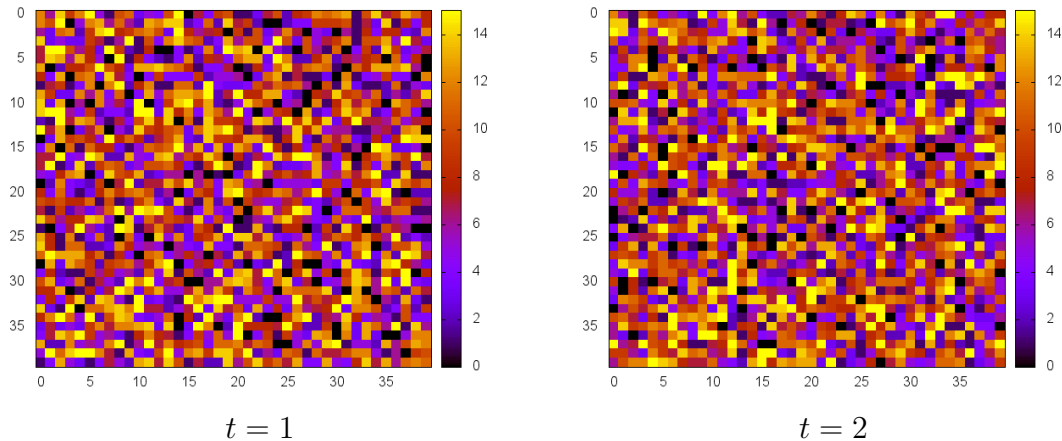


Figure 4.4: HPP noise and speckles

can not follow dynamic evolution of “speckle” in the book but if we create the computer animation, it clearly demonstrates a chaotic behavior of the system.

The *evolution* \mathcal{E} in time is deterministic and proceeds as an alternation of local collisions \mathcal{C} (only particles at the same node are involved) and propagation \mathcal{S} . [29]

$$\mathcal{E} = \mathcal{S} \circ \mathcal{C},$$

An important property of this particular LGCA is *reversible* updating. Indeed, for this model we can write

$$\mathcal{I} = \mathcal{C}^2,$$

where \mathcal{I} is *identity operator*. It means that twofold application of the collision operator leads back to the initial configuration. The FHP model introduced in chapter 5 does not have this property because of indeterministic character of the operator \mathcal{C} . Thus, mathematical description of both models is slightly different, as we discuss in the chapter 6.

This model does not lead to the Navier-Stokes equation in the macroscopic limit. The reason for this is insufficient degree of rotational symmetry of the lattice. Certain tensors composed of products of the lattice velocities so-called lattice tensors – are not isotropic over the grid [29]. In chapter 6 we discuss how the Navier-Stokes equation is recovered from the aforementioned FHP model which overcomes the problem with rotational symmetry.

Chapter 5

FHP model

5.1 General properties

As we remarked at the end of the previous chapter, the HPP lattice-gas model does not lead to the Navier-Stokes equation in the macroscopic limit, therefore this model is not useful for the flow simulation. The next step in the development of the lattice-gas cellular automata is the so-called *FHP model*, over a hexagonal lattice (see figure 5.1) with higher symmetry. [29, page 53] This model was introduced by Frisch, Hasslacher, and Pomeau in the scientific paper "Lattice gas cellular automata for the Navier-Stokes equation, 1986" and was named according to the initials of these three authors [7].

1. All particles have the same mass m ($m=1$) and are indistinguishable.
2. The direction of particles *propagation* is represented by *lattice vectors* (*lattice velocities*) \mathbf{c}_i (i from 1 to 6).
3. Exclusion principle applies, i.e. a given cell cannot be occupied by two or more particles with the same velocity.

At given cell, the velocity of a particle can have 6 different directions given by vectors

$$\mathbf{c}_i = \left(\cos \frac{\pi i}{3}, \sin \frac{\pi i}{3} \right), \quad i = 1, 2 \dots 6.$$

One of the major departures from the HPP model is the presence of non-deterministic rules for head-on collisions. For initial state $(i, i + 3)$ we have two different possible final states $(i + 1, i + 4)$ or $(i - 1, i + 2)$ and the choice of the final state must be a random process with equal probabilities for both states. If one chooses always one and the same final state the model becomes chiral and is no longer invariant with respect to spatial reflections (parity transformation). This is an undesired property because the hydrodynamic equations do not break parity symmetry [29, page 54]. In fact, it is a pseudo random choice. The two-particle collisions conserve not only mass and momentum but additionally conserve the difference of particles number in opposite directions. This invariant is not desired because the model will differ from ordinary hydrodynamics in the large-scale dynamics. The one way to avoid it is to add three-particle collisions where the initial state $(i, i + 3, i + 5)$ changes to final state $(i + 1, i + 4, i + 6)$. Important

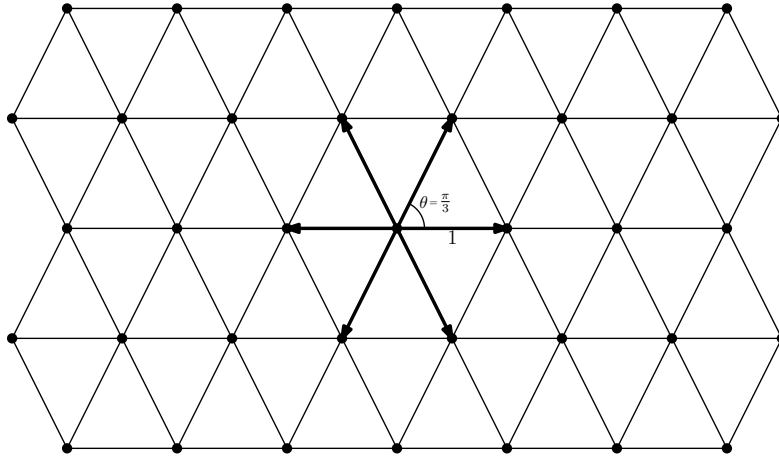


Figure 5.1: lattice

property of LGCA is that collisions are strictly local, i.e., only particles of the node are involved [29, page 55].

By adding new collision rules, we get different FHP models, named FHP-I, FHP-II, FHP-III in relation to the collision rules we use. The 2 or 3 particle collisions rules form the set of FHP-I model. (These models have the same form and differ only in their viscosity coefficient, which decreases with increasing number of collisions). [29]

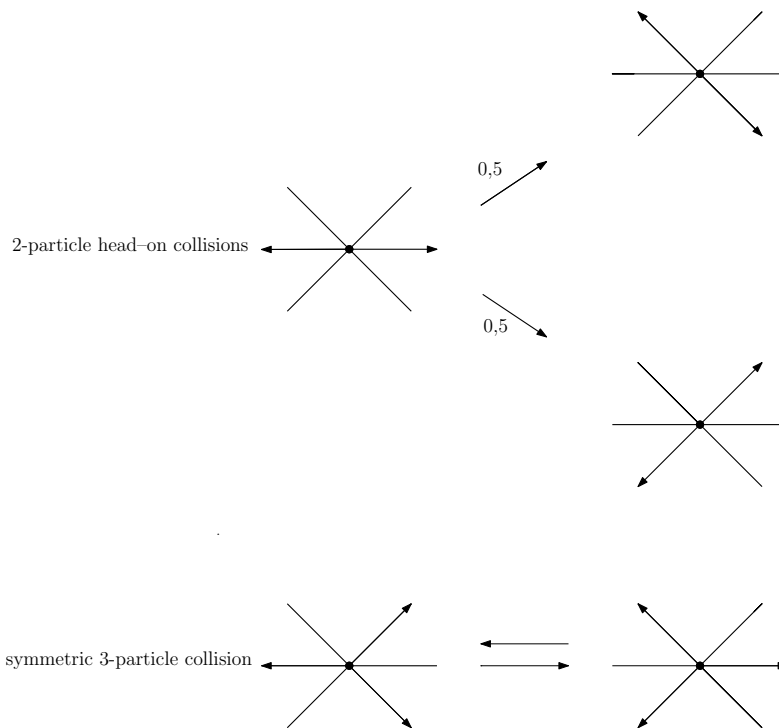


Figure 5.2: 2-particle and symmetric 3-particle collisions

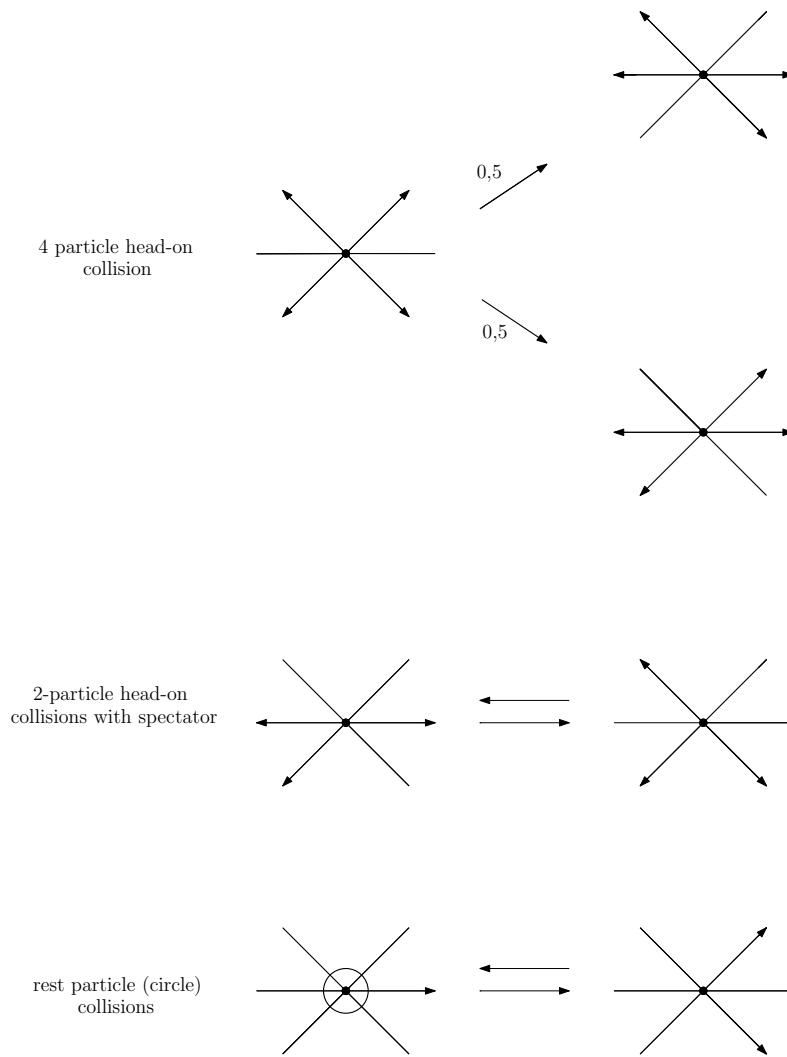


Figure 5.3: FHP collisions

The main goal of this thesis was to implement FHP model in appropriate programming language (we have chosen C++). This goal has been accomplished and we discuss our program in detail in chapter 7. There in, reader can find the description of the structure of the program and numerous figures and results that have been obtained by our program. The full C++ code can be found in the appendix C.3

Chapter 6

Statistical physics of cellular automata

In this chapter we show how the Navier-Stokes equation (3.31) can be obtained as a large scale limit of the FHP model introduced in chapter 5. In our exposition we follow [29] closely.

6.1 Statistical ensembles in classical mechanics

In classical mechanics we often encounter systems with huge numbers of degrees of freedom like gases or fluids. Although the evolution of such complicated system is driven by the Hamilton equations [10], it is impossible to solve these equations nor it is possible to specify the initial conditions with desired accuracy. In order to circumvent this fundamental obstacle in analyzing the dynamics of complicated systems, the methods of *statistical physics* have to be invoked. For an excellent review on statistical physics, see [17]. Although we do not present classical statistical physics here, we briefly sketch the main idea behind the concept of *statistical ensemble* as it plays an important rôle in the present context of cellular automata.

In the Hamiltonian formulation of classical mechanics, the state of each particle is described by six numbers: three coordinates of its position and three components of its momentum. System consisting of N particles is, therefore, described by $6N$ numbers. It is important that these numbers are treated as tantamount coordinates on an abstract space (manifold) called the *phase space*. In this formalism, the state of entire system is represented by a single point in the phase space and this point has $6N$ coordinates. As the configuration of the system changes in time (positions and momenta are evolving), points representing evolving state in the phase space form a curve called the *phase trajectory*.

Now, if the number N is big, e.g. 10^{23} for typical gas, it is impossible to calculate the phase trajectory. First, we cannot solve the equations of such enormous complexity, second, we cannot know the initial conditions for the system exactly. However, there are points in the phase space which are indistinguishable from the macroscopic point of view. For example, typical properties of the gas are the volume, pressure and temperature. Knowing these three quantities, it is irrelevant how exactly the particles are moving and what their momenta are. Thus, there are infinitely many points in the phase space which differ in the detailed motion of particles but share the same macroscopic characteristics like volume, pressure or temperature. We say that the *microstates* are different but the *macrostates* are the same.

Thus, with each macrostate there is associated the volume in the phase space. This *phase volume* contains points representing the same macrostate but many different microstates.

As the system evolves in time, it changes its macrostate, but we do not know in what microstate it is and hence we do not know the phase trajectory. Instead, we study how the initial phase volume evolves, so that, again, at each time we get different phase volume. The bigger the volume is, the bigger is our uncertainty in the knowledge of the state of the system. However, there is an important theorem by *Liouville* stating that, for conservative systems, the phase volume is constant in time. In fact, this theorem shows that the phase volume behaves like incompressible fluid which just means that it is governed by the same equation of continuity like the real fluids do.

Remark. In practice, the Liouville theorem is of little use because, typically, the phase volume is preserved, but it changes the shape into very complicated fractal structure. We have to approximate the fractal by the so-called *coarse-graining* of the phase space but the volume of this coarse-grained structure will be bigger than the original one. Hence, although in principle the information we have about the system remains constant (it is not decreasing), from the practical point of view the information is lost because the coarse-grained phase volume is growing. This is the root of famous second law of thermodynamics stating that the entropy of the system never decreases. Beautiful discussion on the information loss on a fundamental level and on the origin of entropy can be found in [18] and [23].

The phase volume, i.e. the region of the phase space which contains points representing given macrostate, is also called the *statistical ensemble* [17]. We can imagine it like having infinitely many copies of our system and each copy corresponds to different microstate. If we need to calculate some macroscopic quantity, e.g. pressure, we have to calculate the average, mean value. By mean value \bar{f} of quantity f we usually mean the time average,

$$\bar{f} = \lim_{T \rightarrow \infty} \frac{1}{T} \int_0^T f(t) dt. \quad (6.1)$$

In other words, one can measure the quantity f during sufficiently long time interval T and then calculate the average. In statistical physics, however, the average is calculated through the ensemble. That means, we have infinitely many copies of our system in the ensemble and each has slightly different pressure. We calculate the average of the pressure in the ensemble and the result will be actual mean value of the pressure in our real physical system. Each microstate in the ensemble has associated probability density ρ which is a function of phase space variables q ($3N$ -tuple of coordinates) and p ($3N$ -tuple of momenta). The ensemble average is then obtained by

$$\langle f \rangle = \frac{1}{Z} \int f(q, p) \rho(q, p) dq dp, \quad (6.2)$$

where Z is the normalization constant, usually called the *partition function*. It is the content of *ergodic hypothesis* that the time average is the same as the ensemble average,

$$\langle f \rangle = \bar{f}. \quad (6.3)$$

This hypothesis is a deep mathematical problem. It was proven in many situations which apply to physical needs, in other cases it is known that ergodicity is violated. For introduction to ergodic theory, see [27], for a review on the ergodicity in classical mechanics we recommend [4].

6.2 Phase space of cellular automata

After the explanation of what that statistical ensemble and phase space is in the physical context (where these notions first arose), we can now turn our attention to cellular automata.

Let L be the lattice of lattice-gas cellular automaton. In order to simulate infinite lattice, we impose periodic boundary conditions. Each node of the lattice consists of six cells to be labeled by index $i = 0, 1, \dots, 5$. By symbol $n_i(t, \mathbf{r})$ we denote the state of i -th cell in the node with the position vector \mathbf{r} at time t . Function n_i is boolean valued, i.e. it acquires value 1 if the cell is occupied and 0 otherwise. In other words, if the node with the position vector \mathbf{r} (at time t) contains particle moving with the velocity \mathbf{c}_i , then $n_i(t, \mathbf{r}) = 1$ (refer to chapter 5 for the definition of \mathbf{c}_i). In order to describe the transition from state at time t to state at time $t + 1$ we introduce the *collision function* Δ_i defined by

$$n_i(t + 1, \mathbf{r} + \mathbf{c}_i) = n_i(t, \mathbf{r}) + \Delta_i. \quad (6.4)$$

By the *state* of the lattice we mean the collection of values $n_i(t, \mathbf{r})$ for all nodes at given time. The set of all possible states will be denoted by Γ and referred to as the *phase space* of the automaton considered. The elements of Γ will be denoted by s, s', s'', \dots . If the lattice is in state $s \in \Gamma$, the state of individual node with the position vector \mathbf{r} will be denoted by $s(\mathbf{r})$. In other words, state s is determined by the states $s(\mathbf{r})$ for all possible values of \mathbf{r} .

Since the lattice will be typically very big, we have to employ the statistical approach. Hence, let the symbol

$$P(t, s), \quad s \in \Gamma, \quad (6.5)$$

denote the *probability* that lattice L is in the state s at the time t . The probability distribution at the initial time $t = 0$ is normalized by the condition

$$\sum_{s \in \Gamma} P(0, s) = 1. \quad (6.6)$$

This is usual condition expressing the fact that the lattice is in *some* state for sure.

We introduce the operators $\mathcal{S} : \Gamma \mapsto \Gamma$ and $\mathcal{C} : \Gamma \mapsto \Gamma$ on the phase space of the automaton like in the section 4.1. That is, \mathcal{C} represents the collisions and \mathcal{S} the propagation, so that

$$\mathcal{E} = \mathcal{S} \circ \mathcal{C} \quad (6.7)$$

is the evolution operator. Suppose $s \in \Gamma$ is an arbitrary state. Each particle in this state is moving with some velocity and after one time step each particle moves to neighboring

cell; this new state is denoted by $\mathcal{S}s$. Operator \mathcal{C} acts in such a way that if there is no collision configuration, the state $\mathcal{C}s = s$. Otherwise, in each cell where the collision happens is updated according to the rules of particular model. Notice that operator \mathcal{C} does not change the position of particles, merely the direction of velocities. Both operators \mathcal{C} and \mathcal{S} are invertible.

Now, if s' is the state at time t with the probability $P(t, s')$, this state evolves into new state $\mathcal{E}s'$ which, consequently, has the same probability:

$$P(t, s') = P(t+1, \mathcal{E}s') \quad \text{or} \quad P(t+1, \mathcal{S}\mathcal{C}s') = P(t, s'). \quad (6.8)$$

Let us write the state s' in the form $s' = \mathcal{C}^{-1}s$, where the invertibility of \mathcal{C} has been used. The last equation then reads

$$P(t+1, \mathcal{S}s) = P(t, \mathcal{C}^{-1}s). \quad (6.9)$$

Equation

$$P(t+1, \mathcal{E}s) = P(t, s) \quad (6.10)$$

is the expression of the fact that the probabilities are conserved during the evolution. It is a discrete version of the conservation of the phase volume explained in the section 6.1, whence we refer to (6.10) as the *Liouville theorem*.

In the FHP model, however, the collision process is not deterministic and operator \mathcal{C} is not invertible. In such a case, we have to consider all possible outcomes of the collision. If the lattice is in state s and $s(\mathbf{r})$ is the state of \mathbf{r} -th node, there is transition probability

$$A(s(\mathbf{r}) \rightarrow s'(\mathbf{r})) \quad (6.11)$$

that after the collision, the new state of the node will be $s'(\mathbf{r})$, where again $s' \in \Gamma$. Hence, the Liouville equation is replaced by the *Chapman-Kolmogorov equation* [16]

$$P(t+1, \mathcal{S}s) = \sum_{s' \in \Gamma} \prod_{\mathbf{r} \in L} A(s_{\mathbf{r}} \rightarrow s'_{\mathbf{r}}) P(t, s). \quad (6.12)$$

6.3 Occupation numbers

By *observable* we mean any quantity which can be calculated when the state of the lattice is known, i.e. it is arbitrary function f of the state of the lattice:

$$f = f(s), \quad s \in \Gamma. \quad (6.13)$$

Similarly to (6.2), we define the *ensemble average* of quantity f at time t by

$$\langle f(t) \rangle = \sum_{s \in \Gamma} P(t, s) f(s). \quad (6.14)$$

Since we have discrete system now, the probability density ρ in (6.2) is replaced by the probability distribution P and the integral is replaced by the sum; otherwise both expressions are the same.

An example of observable is the *occupation number* [29] at node \mathbf{r} , i.e. the number of particles present at the node with the position vector \mathbf{r} . We have already introduced the notation $n_i(t, \mathbf{r})$ for the number of particles at the node \mathbf{r} with the velocity \mathbf{c}_i ; this quantity can only take values 0 and 1. The *mean occupation number* is then

$$N_i(t, \mathbf{r}) = \langle n_i(t, \mathbf{r}) \rangle. \quad (6.15)$$

The total number at node \mathbf{r} is then

$$\rho(t, \mathbf{r}) = \sum_{i=0}^5 \langle n_i(t, \mathbf{r}) \rangle. \quad (6.16)$$

Recall that in the fluid dynamics we introduce the *mass density* [20] defined as the mass dm contained in the infinitesimal volume dV . Here, we have discrete system and we are assuming the unit mass for all particles. Thus, the mass density is simply equal to the number of particles; that is the reason why we will refer to the mean occupation number as the mass density and denote it by ρ .

Following this analogy further, we introduce the *current density* by

$$\mathbf{j}(t, \mathbf{r}) = \sum_i N_i \mathbf{c}_i, \quad (6.17)$$

which should be compared to definition $\mathbf{j} = \rho \mathbf{v}$ in hydrodynamics. The physical meaning of the current is that it represents the *momentum density*.

The collision function introduced in the equation (6.4) describes the change in the occupation number because of collisions happening in given node. Since collisions preserve the number of particles in the node, the mass density is constant during the collision and so is the total momentum:

$$\sum_i \Delta_i = 0, \quad \sum_i \Delta_i \mathbf{c}_i = 0. \quad (6.18)$$

Applying the sum to equation (6.4), we find

$$\sum_i n_i(t+1, \mathbf{r} + \mathbf{c}_i) = \sum_i n_i(t, \mathbf{r}). \quad (6.19)$$

Taking the average of the last equation, we get

$$\sum_i N_i(t+1, \mathbf{r} + \mathbf{c}_i) = \sum_i N_i(t, \mathbf{r}). \quad (6.20)$$

In the same way we can derive the relation

$$\sum_i N_i(t+1, \mathbf{r} + \mathbf{c}_i) \mathbf{c}_i = \sum_i N_i(t, \mathbf{r}) \mathbf{c}_i. \quad (6.21)$$

So far, we have introduced the notion of (mean) occupation numbers and derived several constraints imposed by the conservation of the mass and momentum. Now we are in position to formulate one of the most important result in the theory of lattice-gas cellular automata. The proof of the theorem can be found in [29] and [8]. In the

statement of the theorem, the *equilibrium* means that the average occupation numbers are time independent.

Theorem. Let the automaton be in the equilibrium state and let the mean occupation numbers N_i be the solutions to the Chapman-Kolmogorov equation (6.12). Then N_i satisfies the *Fermi-Dirac distribution*

$$N_i = \frac{1}{1 + e^{h + \mathbf{q} \cdot \mathbf{c}_i}}, \quad (6.22)$$

where $h \in \mathbb{R}$ and \mathbf{q} is a two-dimensional vector. Conversely, whenever N_i is given by the Fermi-Dirac distribution, it is a solution to Chapman-Kolmogorov equation.

The term ‘‘Fermi-Dirac distribution’’ comes from quantum physics [15]. The elementary particles can be divided into the *bosons* and *fermions*. The former have integer spin (0 or 1) and they are mediating non-gravitational interactions. The best known example of bosons is the photon, mediator of electromagnetic interaction, another examples are bosons W and Z (weak interaction) and gluons (strong interaction between quarks). These particles have the property that all of them can be in the same quantum state. This can happen at low temperatures when all bosons are trying to occupy the ground energy state – we talk about the *Einstein-Bose condensate* [19].

Fermions, on the other hand, are particles with half-integer spin (1/2, 3/2) and they constitute the ordinary matter: electrons, neutrons, protons and others. They cannot occupy all the same state as is dictated by the *Pauli exclusion principle*. Thanks to this principle, the electrons in the atoms are forced to occupy different energy levels which makes it possible to have different chemical elements. Surprisingly enough, the Pauli principle is deeply rooted in the geometry of the spacetime [28]. It is also interesting that although the electrons are fermions, under appropriate circumstances they can combine into the so-called *Cooper pairs*. The bound state of two electrons is a boson, however, and so the Cooper pairs can all occupy the ground state. This is in the hearth of the Bardeen-Schrieffer-Cooper theory of superconductivity [5].

The system of many fermions (satisfying the Pauli exclusion principle) in thermodynamical equilibrium is described by the Fermi-Dirac distribution [13]

$$\langle n_i \rangle = \frac{1}{1 + e^{(\varepsilon_i - \mu)/kT}}, \quad (6.23)$$

where n_i is the number of fermions occupying the state with energy ε_i , μ is the so-called *Fermi energy*, k is the Boltzmann constant and T is the thermodynamical temperature of the fermion ‘‘gas’’. We can see that the Fermi-Dirac distribution for fermions has the same form as the distribution for mean occupation number in the cellular automaton. This is not surprising, however, for the ‘‘exclusion principle’’ is built in the microdynamics of the cellular automaton: at given node, at most one particle can have velocity in given direction.

6.4 Emergence of the Navier-Stokes equation

In the rest of this chapter we sketch how the Navier-Stokes equation emerges from the cellular automaton driven by microdynamics described in chapter 5. We will omit many important details and calculations which can be found in [29], [7] [8].

Suppose that the initial distribution of ρ (mass density) and \mathbf{j} (momentum density) varies significantly on some large spatial scale. This scale will be denoted by ε^{-1} , so that the small parameter ε has the dimension of inverse length. The length is measured in the lattice units so that ε^{-1} is simultaneously the time scale. According to [29], there are three distinguished time scales on which three different phenomena take place:

1. scale ε^0 , relaxation towards the equilibrium state – this is very fast process;
2. scale ε^{-1} , creation of the sound waves and advection – slower, but still relatively fast process;
3. scale ε^{-2} , diffusion – significantly slower process.

Following [29], we introduce three time variables and two spatial variables by

$$t^*, \quad t_1 = \varepsilon t^*, \quad t_2 = \varepsilon^2 t^*, \quad \mathbf{r}^*, \quad \mathbf{r}_1 = \varepsilon \mathbf{r}^*. \quad (6.24)$$

Quantities decorated with the star are discrete, while remaining variables are treated as continuous, since ε is assumed to be small.

Now, the mean occupation numbers N_i can be expanded into the series of the form

$$N_i = N_i^0 + \varepsilon N_i^1 + \mathcal{O}(\varepsilon^2), \quad (6.25)$$

where N_i^0 is the equilibrium value of N_i . In order to describe the evolution of N_i , we expand $N_i(t+1, \mathbf{r} + \mathbf{c}_i)$ into the Taylor series (in both arguments) up to the second order derivatives:

$$\begin{aligned} N_i(t+1, \mathbf{r} + \mathbf{c}_i) &= N_i(t, \mathbf{r}) && (\mathcal{O}(0)\text{-term}) \\ &+ \frac{\partial N_i}{\partial t} + \frac{\partial N_i}{\partial x_{i\alpha}} c_{i\alpha} && (\mathcal{O}(1)\text{-terms}) \\ &+ \frac{1}{2} \frac{\partial^2 N_i}{\partial t^2} + \frac{1}{2} \frac{\partial^2 N_i}{\partial x_{i\alpha} \partial x_{i\beta}} c_{i\alpha} c_{i\beta} + \frac{\partial^2 N_i}{\partial t \partial x_{i\alpha}} c_{i\alpha} && (\mathcal{O}(2)\text{-terms}) \\ &+ \mathcal{O}(3). && (6.26) \end{aligned}$$

The fact that the relaxation to local equilibrium is very fast process happening on the time scale $\varepsilon^0 = 1$, i.e. in few iterations, means that in order to study the hydrodynamics in cellular automata, we can neglect processes of order 1. Then the partial derivatives can be expanded as

$$\partial_t = \varepsilon \partial_t^{(1)} + \varepsilon^2 \partial_t^{(2)}, \quad \partial_\alpha = \varepsilon \partial_\alpha^{(1)}, \quad (6.27)$$

where we use usual abbreviations for partial derivatives. Substituting all these expansions into conservation laws (6.20) and (6.21) and neglecting appropriate powers of ε , we find in the first order [29]

$$\partial_t^{(1)} \sum_i N_i^0 + \partial_\beta^{(1)} \sum_i c_{i\beta} N_i^0 = 0, \quad (6.28)$$

$$\partial_t^{(1)} \sum_i c_{i\alpha} N_i^0 + \partial_\beta^{(1)} \sum_i c_{i\alpha} c_{i\beta} N_i^0 = 0. \quad (6.29)$$

Recalling the definition of mass and momentum density, equations (6.16) and (6.17), we see that the first equation is in fact the continuity equation known from hydrodynamics

$$\frac{\partial^{(1)}\rho}{\partial t} + \nabla^{(1)} \cdot (\rho \mathbf{u}), \quad (6.30)$$

where the velocity is

$$u_\alpha = \sum_i c_{i\alpha}. \quad (6.31)$$

Equation (6.29) resembles the equation of motion for the continuum. It can be written in the form

$$\partial_t^{(1)}(\rho u_\alpha) + \partial_\beta^{(1)}(\rho u_\beta) P_{\alpha\beta}^{(0)} = 0, \quad (6.32)$$

where the equilibrium “stress-energy tensor” is

$$P_{\alpha\beta}^{(0)} = \sum_i c_{i\alpha} c_{i\beta} N_i^0. \quad (6.33)$$

Calculating the components of the stress-energy tensor and comparing it to usual stress-energy tensor for inviscid fluid [20], it can be shown [29] that equation (6.29) can be brought into the form

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla P, \quad (6.34)$$

where the pressure P is given by

$$P = \frac{\rho}{2\rho_0 g(\rho_0)} - \mathbf{u}^2, \quad g(\rho) = \frac{3 - \rho}{6 - \rho}. \quad (6.35)$$

Clearly, equation 6.34 is the *Euler equation* describing inviscid fluid. Even more cumbersome calculation of the $\mathcal{O}(\varepsilon^2)$ terms [8, 12] then reveals the *Navier-Stokes equation*, c.f. (3.31),

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla P + \nu \Delta \mathbf{u}, \quad (6.36)$$

where the kinematic viscosity ν is

$$\nu = \frac{\nu^u}{g(\rho_0)}, \quad (6.37)$$

and the coefficient ν^u must be calculated separately for each model.

This completes our brief discussion of how the Navier-Stokes emerges from the microdynamics of the cellular automaton on the large scale.

Chapter 7

Results

In the previous chapters we have introduced several theoretical ideas behind the simulation of the turbulent flow using the cellular automata. In particular, we have focused on the FHP model which is realistic enough to reproduce the Navier-Stokes equation, as we have indicated in chapter 6.

The main goal of this thesis was to implement FHP model in appropriate programming language and perform several simulations of the turbulent flow past the airfoils of arbitrary shape. This goal has been achieved and, in fact, we have presented some abilities of our program already in chapter 5 where we have used the results of our program in order to illustrate basic properties of FHP model.

In this chapter we describe our program in more detail, explain what kind of problems it is able to solve and we give a brief description of the airfoils we used for the simulations; the latter is subject of the next section.

7.1 Two-dimensional flow

In previous chapters, we did not discuss three-dimensional lattice-gas cellular automata since this subject is beyond the scope of the thesis. The review of problems and solutions associated with three-dimensional automata, see [6]. For this reason, we have implemented only two-dimensional FHP model.

On the other hand, real aeroplanes are, of course, three dimensional objects and so are the wings. Moreover, the cross sections of the wings are not all same: the shape of the cross section varies along the wing. Numerical solution of the Navier-Stokes equation for such realistic and complicated system is overwhelmingly complex, not solvable in the framework of bachelor thesis.

Nevertheless, for many reasons, it is quite common to study two-dimensional flow. First, some physical systems indeed behave like two-dimensional. This happens when the real three-dimensional flow is thick enough, so that the boundary effects can be neglected, and, in addition, the velocity field looks in the same way in all cuts of the flow. For example, the flow of the river in the river-basin can have this property: the flow changes along the basin but is almost independent of the height (or depth). If a foliation of the flow exists such that the two-dimensional layers of the flow are isometric, the flow is said to be two-dimensional and is governed by two-dimensional versions of the

hydrodynamic equations. In other words, although the restriction to two-dimensional flow may look as unphysical simplification, many real systems fall into this category.

Second obvious reason why the study of two-dimensional flow is of interest is that it is much easier to analyze it using the analytical methods. For stationary two-dimensional flow there exists well understood theory in which it is possible to obtain exact solutions of many (in principle, all) problems by virtue of the so-called complex potential and the vortex panel method. These methods have been reviewed in the thesis [25]. However, these methods apply only to *stationary* and *inviscid* flow which are assumptions much more restrictive than the dimension of the space. Hence, FHP model overcomes these drawbacks of usual theory of two-dimensional flow and hence is much more realistic.

We have mentioned that real aeroplanes and wings are three-dimensional. Restricting ourselves to two-dimensional models, we must choose a particular cross section of the real wing and use it as a two-dimensional profile of the wing. This two dimensional profile of the wing is called *airfoil*.

In the thesis [25], the problem of stationary flow past arbitrary airfoil has been addressed and the algorithm for the vortex panel method has been implemented. In this thesis we make a step towards the analysis of realistic turbulent flow past the arbitrary airfoil.

7.2 Description of the airfoils

Basic information on the history and construction of the airfoils can be found in numerous online resources, see, e.g. [1,2]. For the description of the terms used in relation to airfoils, see [25]. Airfoils gained traction in times of world wars. At the dawn of aviation, these airfoils were very primitive. Systematization and airfoil theory development permitted to create a new types of airfoil with novel aerodynamic characteristic. In this chapter, five different airfoils will be described. They were engineered in various historical periods and used for a different types of aircraft, from recreation high-wing monoplane to the trainer fighter. The figures used in this section are taken from [3] and complemented by the author of the thesis.

The first airfoil to be considered here is NACA-2418, shown in figure 7.1, which belongs to four-digit series airfoils. Nationally Advisory Committee For Aeronautics (NACA) was the first institution which began to systematize different types of airfoils, having developed a symbol system for them. This system permits to obtain basic information about airfoil from its name. The majority of parameters are represented as the percentage of the chord. Let us introduce this system on the example of NACA-2418 profile.

1. The first digit represents a maximum camber which is the maximum distance between chord and midline of an airfoil. In our case, maximum camber is equal to 2%,
2. The second digit represents a point on the chord of maximum camber from the leading edge in tens of percents of the chord. In our case, the maximum camber is placed over the distance 0.4 (40%) from the leading edge,
3. The third and the forth digits represent the maximal thickness which is the maximum distance between upper and lower surfaces. In our case, maximal thickness is equal to 18%

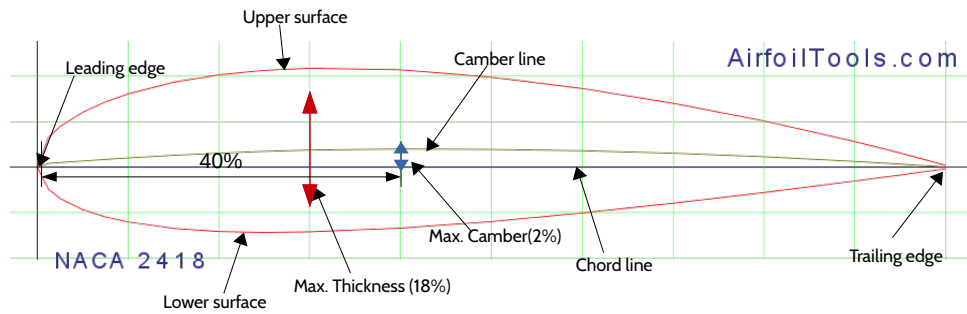


Figure 7.1: NACA-2418

The given airfoil has been chosen for review in this thesis because it is used as a root airfoil in the wing of aircrafts very important for Czech aviation, such as Zlin-526F, Zlin-726 and also for propeller of Sikorski S-61F helicopter.

Another four-digit airfoil which will be considered is NACA-2412, figure 7.2. Based on the explanation above, we can identify that the difference between airfoils rests in maximal thickness which is 12 % for this airfoil. This airfoil is of interest because it is used in Cessna-172 which is nowadays the most produced aircraft in the world. Also, this airfoil was used in very interesting aircraft – Beriev Be-103, the amphibious aircraft.

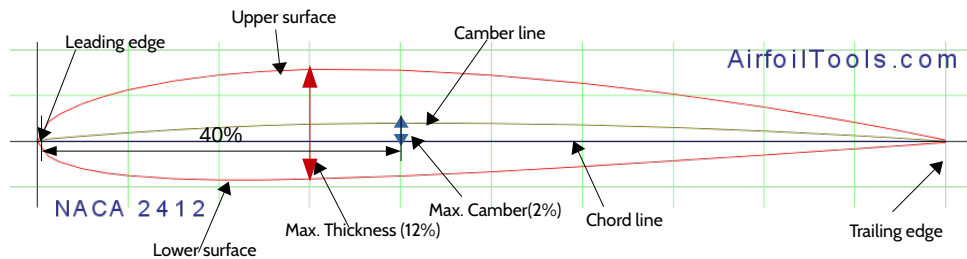


Figure 7.2: NACA-2412

Permanent increase of aircraft velocities claims significant decrease of the drag force. For that purpose, the so called *laminar-flow airfoils* were engineered. Laminar-flow is provided by the move aside of maximal thickness point from a leading edge. The first aircraft which employed the laminar airfoil was American second world war fighter P-51 Mustang. Laminar airfoils form 6 digit series in NACA classification. Let us consider the 6 digit series on the example of NACA 66,2-(1.8)15.5 which is used in P-51 F aircraft modification and shown in figure 7.3.

1. The first digit 6 represents the series of airfoil.
2. The second and third digits 6,2 (62%) represent the length of laminar flow as percentage of the chord.
3. The fourth and fifth digits 1.8 represent the middle of area with laminar flow and low drag force on the C_l/C_d diagram.
4. The sixth, seventh and eighth digits 15,5 represent the maximal thickness. (15.5%)

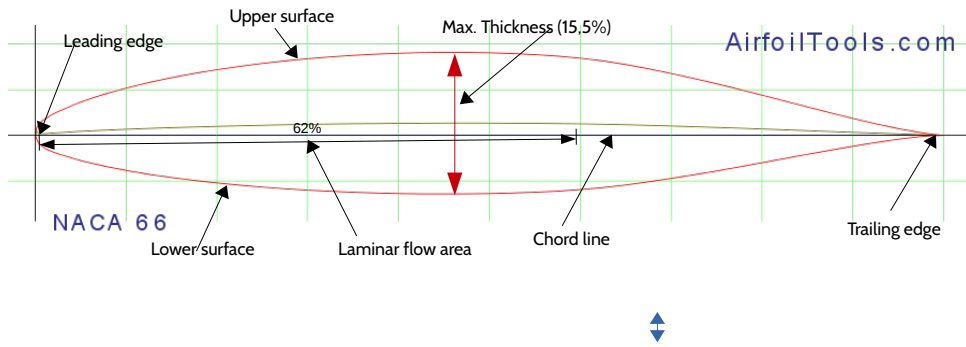


Figure 7.3: NACA 66,2-(1.8)15.5

Another representative of laminar flow airfoil which we will consider in this thesis is NACA 64-012, figure 7.4. This airfoil is used as a root airfoil in the Aero L-39 Albatros wing. This airfoil is symmetrical with maximal thickness equal to 12%.

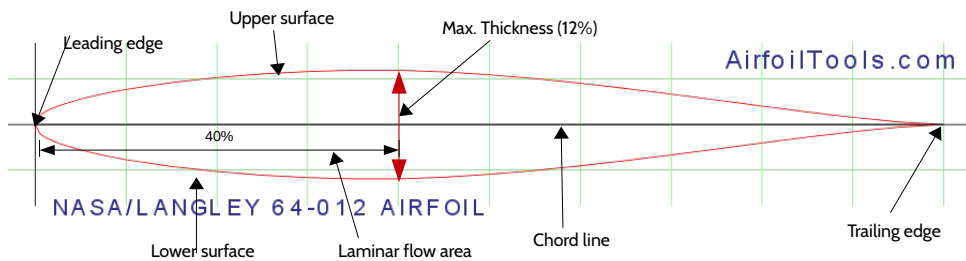


Figure 7.4: NASA 64-012

The last type of airfoil to be considered is Clark YH, figure 7.5. The Clark Y airfoil was designed in the beginning of twenties by American engineer Virginus E. Clark. The H letter in the name of airfoil means the turned up trailing edge. Because of its aerodynamic properties, this airfoil was used in large number of very successful Soviet aircrafts, e.g. Jakovlev Jak-18T, Jakovlev Jak-52, Ilyushin Il-2 and also was used in Let-11 C which was a Czechoslovak version of Jakovlev Jak-11. This airfoil is very popular in model aeroplane flying also because of the flatness of lower surface.

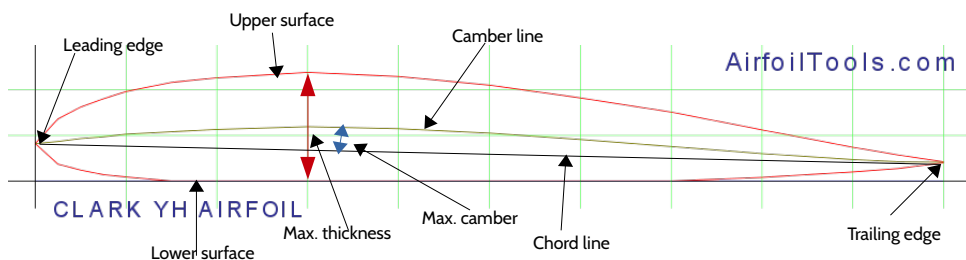


Figure 7.5: CLARK YH

7.3 Implementation of FHP model

In this section we describe in some detail the structure of the C++ program we have created in order to implement FHP model. For the definition of the model (type of the lattice, collision rules, etc.), refer to the chapter 5.

For the visualization of results we have used programs GNUplot and Mathematica. Thus, our program creates the data files which can be processed by them and, especially for GNUplot, it generates appropriate script converting the data files to graphics.

The program consists of two files. The first one is `main.cpp` in which all functions, definitions of variables and the main body of program are present, the second file, `init_conds.cpp` contains definitions of several different initial and boundary conditions.

Let us discuss the parameter which the program can be run with. It is possible to set the size of the grid and choose whether the grid points should be displayed in the resulting images. We found it useful to switch the grid points on in the cases where the grid is small enough, i.e. for demonstrational purposes. Such demonstrational figures are shown in section 7.4 below. For large grids (number of nodes of order 10^6), the density of the grid points is very big and the nodes become indistinguishable and hence not necessary.

Another parameter related to the grid is the density of the coarse-graining. The coarse-graining is necessary to calculate relevant macroscopic quantities, in particular the velocities. This procedure is analogous to introducing the “physically infinitesimally small volume element” in standard hydrodynamics. Thus, we divide the grid into a number of boxes of size fixed by the parameters of the program; we call them `dx` and `dy`.

For convenience in operation, there are some auxiliary parameters affecting the appearance of resulting images. In the process of coarse-graining, we associate a single vector of the velocity with each box and this vector is an average computed from individual velocities in each cell contained in the box. Thus, if the velocity of single particle is of order one, the averaged velocity is of the same order but it is associated with bigger area. The bigger the area is, the smaller is the magnitude of the averaged velocity. While this has no effect on the physical content of the computation, it is convenient to rescale the velocities in resulting image in order to get apparently smooth flow. The parameter `scale_factor` fulfills this task.

The user of the program can choose whether resulting images will be merged into single animation (`.gif` file), how many iterations of the calculation will be skipped between successive images, the total number of iterations and the type of the plot. We distinguish three types of plots:

1. vector plot, when the velocity is displayed as vector attached to the center of the coarse-grained boxes; by appropriate choice of the `scale_factor` and the dimension of the grid, the vectors apparently form the flow lines;
2. scalar density plot, in which the magnitude of the velocity is calculated and the box is displayed with color measuring the magnitude; a bar with the scale is shown next to such figure;
3. vector density plot, is similar to the previous one, but the color of the point now reflects also the direction of the velocity, not its magnitude only.

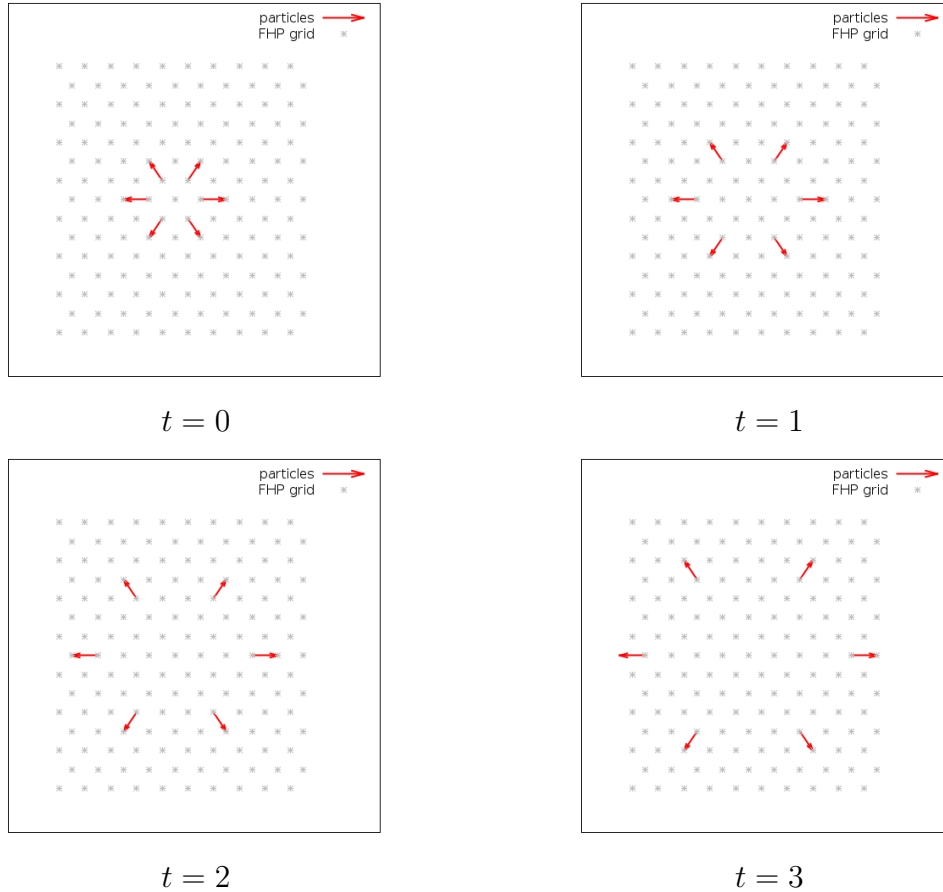


Figure 7.6: Collision-free motion of particles on the grid of dimension 10×15 .

Examples of all three types of the plots are given below.

Among the physical parameters, the most important one is `boundary_conditions` which determines, surprisingly, boundary conditions. Different types of boundary conditions and their physical meaning are described below, in the section 7.4.

We will not describe the algorithms and programming techniques employed but we will illustrate the capabilities of our program on numerous examples.

7.4 Illustrative examples

Let us start with the free motion when no collisions occur and particles are moving with constant velocities. We have chosen the initial state of containing six particles moving in all possible directions, as shown in figure 7.6.

The question, of course, is what happens on the boundary of the grid. In principle, there are three possibilities which make sense. The most trivial one is that particles reaching the boundary will leave the grid, they disappear from the system. In such a case, one needs permanent injection of the particles, otherwise the grid will become empty after short time. This type of boundary conditions is, in fact, quite physical if we wish to study the flow past obstacles in the wind tunnel or, if the grid is sufficiently large, the flow past the obstacles in unbounded space.

The second possible boundary conditions correspond to the fluid in a closed vessel with impenetrable walls, so that the particle is reflected on the boundary. Disadvantage of this approach rests in the fact that such closed system does not correspond to usual physical circumstances. Moreover, the artificial boundaries can have unphysical impact on the fluid dynamics, as they create the reversed flow of the particles. This flow can affect the particles inside the box which were not supposed to interact with the flow which has already passed the obstacle.

This drawback is suppressed by imposing periodic boundary conditions. In this case, the particle leaving the grid through, say, right boundary reappears on the left boundary with the same velocity. The purpose of such boundary conditions is to imitate the infinitely large grid by a finite one. For example, the flow past the cylinder with periodic boundary conditions corresponds to infinitely long tunnel with infinite number of cylinders.

Our program can simulate the “free” boundary conditions when particles leave the grid for ever and periodic boundary conditions. The “reflecting” boundary conditions are not implemented as we considered them unphysical but the structure of the program can be easily modified in order to accommodate such conditions. In what follows we always specify which boundary conditions have been employed. Examples of free and periodic boundary conditions are shown in figures 7.7 and 7.8.

Having clarified the free motion of particles and appropriate boundary conditions, we can turn our attention to the collisions. In figure 7.9 we show two pairs of particles moving in opposite directions. Recall that the main feature of FHP model is that it preserves the isotropy of the Navier-Stokes equation. Hence, although both pairs are moving in horizontal directions, after the collision each pair moves along different direction. For each collision, the resulting direction is chosen randomly with the probability $1/2$. (grid 10×10)

Let us now consider symmetric 3-particle collisions. In this case, the choice of the resulting state is given uniquely and hence no random choice is necessary. Thus, in figure 7.10 we can see two collision with the same initial velocities of the particles and both result in the same post-collision configuration.

The next type is 4-particles collision. In the figure 7.11 are shown two 4-particles collisions with the same initial velocities of the particles. Let us consider one of them. After the collision one of two pairs moves along its initial direction, the second pair acquires the horizontal direction. Similar to 2-particles collision, the resulting direction for one of two pairs is chosen randomly with the probability $1/2$.

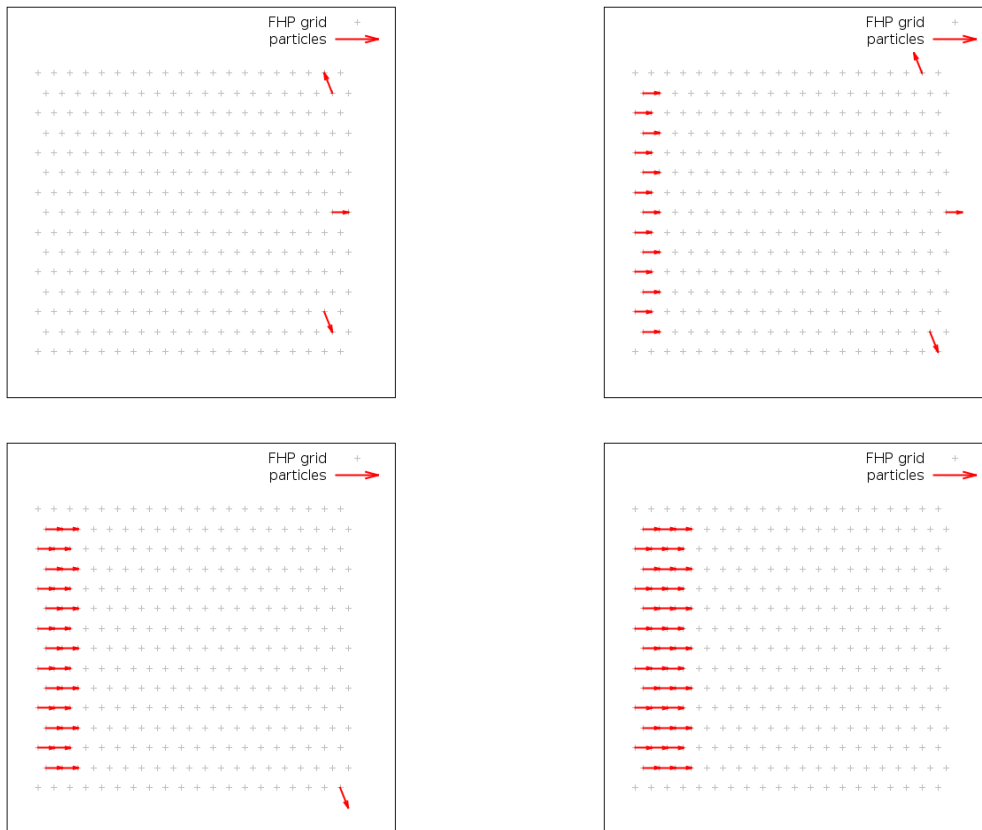


Figure 7.7: Motion of free particles (without collisions) with the ingoing flow and free boundary conditions. Three particles initially located close to the right wall are reaching the boundary and leaving the grid, while there is an incoming stream of the particle emerging from the left wall. (grid 20×15)

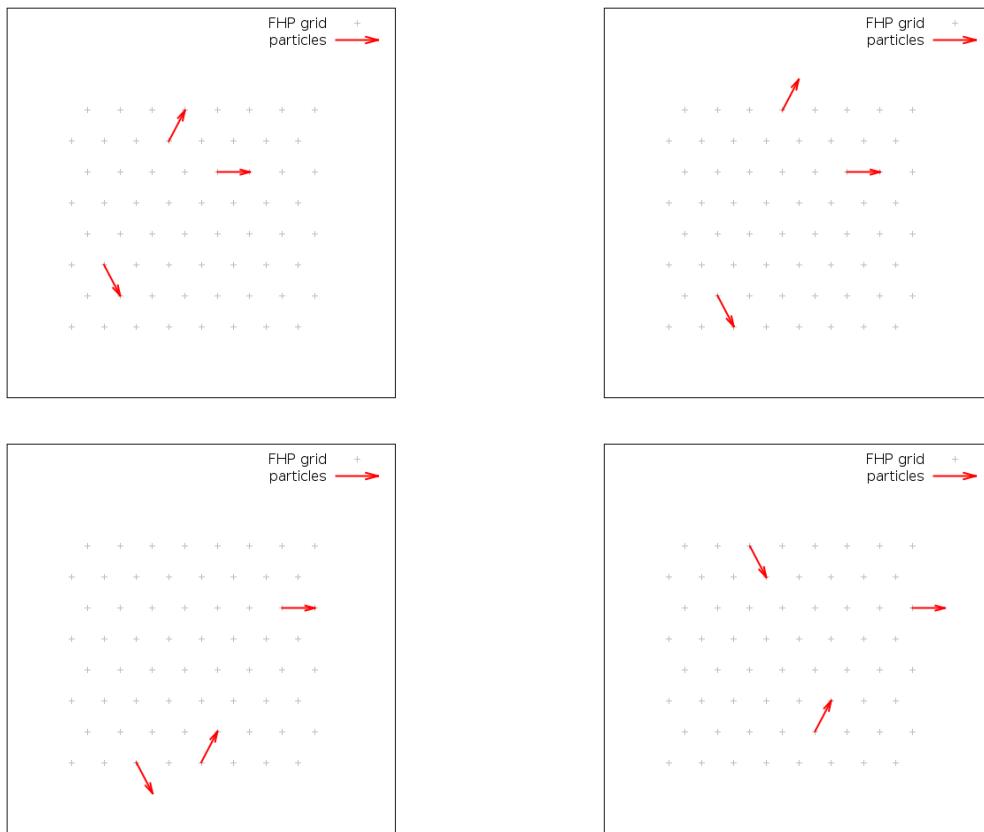


Figure 7.8: Motion of free particles (without collisions) with periodic boundary conditions. As can be seen, after reaching the boundary, particles reappear on the opposite wall. (grid 8×8)

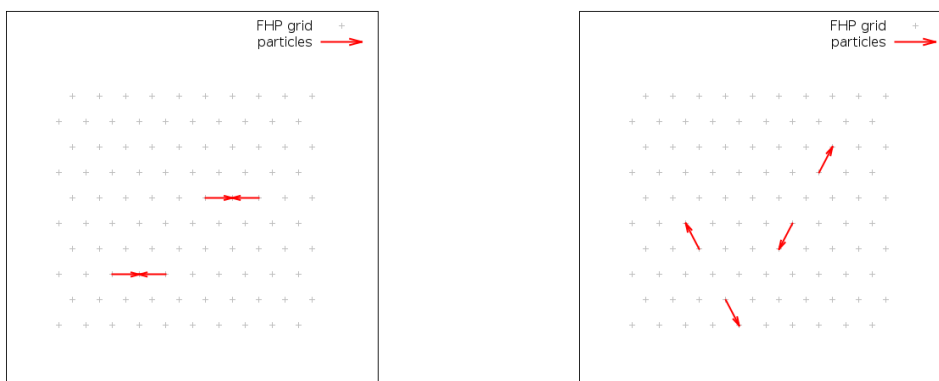


Figure 7.9: Two particle collisions. In order to preserve the isotropy, directions of particles after the collision are chosen randomly with the probability $1/2$.

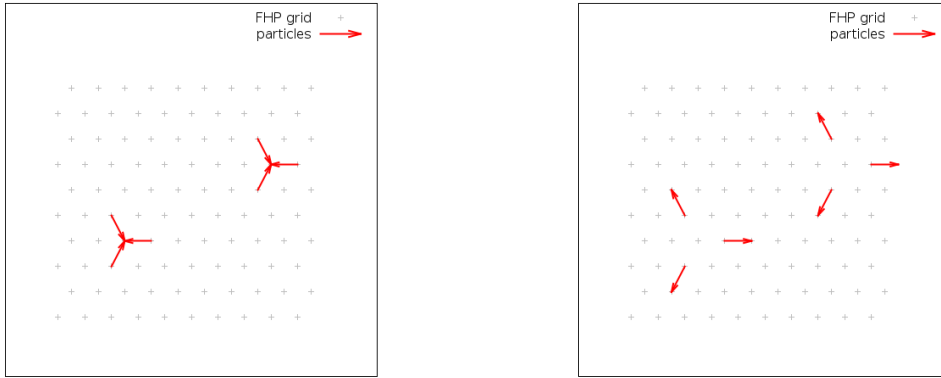


Figure 7.10: Symmetric three particle collision.

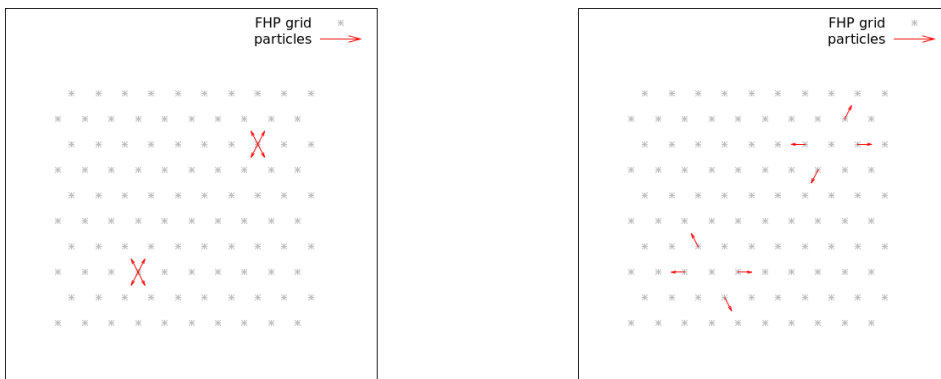


Figure 7.11: Four particle collision.

In the figure 7.12 is shown the next type of collision, called collision with spectator. This collision is notable because of the addition to 2-particle collision the one particle called “spectator”. In contrast to the pair of particles which leave their horizontal directions, “spectator” moves along its initial direction.

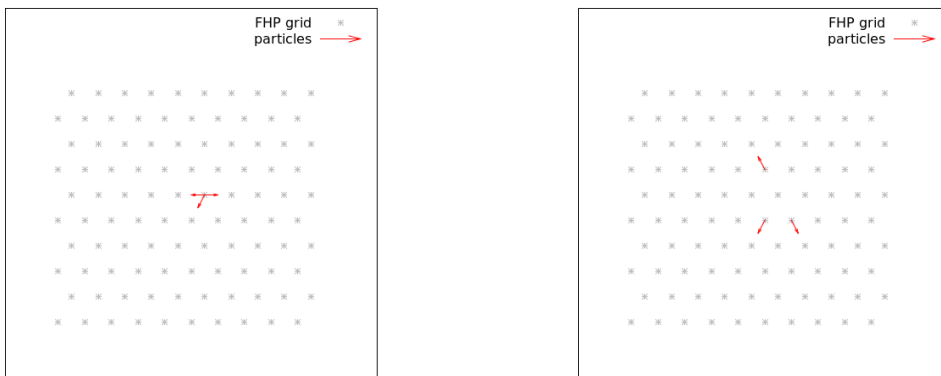


Figure 7.12: Collision with spectator.

The last type of collisions we explain is the rest particle collision. The red point in the

figure 7.13 is particle which does not change its position in time, lattice velocity vanishes; that is why it is called the “rest-particle”. After the collision with particle having \mathbf{c}_i they transform into two particles with \mathbf{c}_{i-1} and \mathbf{c}_{i+1} .

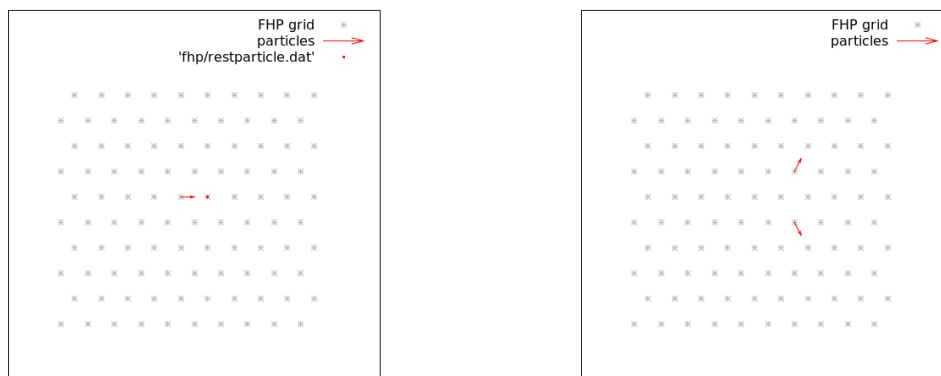


Figure 7.13: Rest-particle collision.

According to the number of collisions we use, we get a different kinds of FHP models, where 2 and 3-particles collisions form the minimal set of collisions for FHP model called FHP-I. [29, page 54] Adding other collision types to FHP model (called FHP-II, FHP-III) decreases the viscosity coefficient.

Now, having explained the boundary conditions and types of collisions that we use, let us consider the obstacles which we immerse into the flow in order to investigate of the turbulent flow that they produce. The most trivial obstacle is the wall shown in the figures 7.14, 7.4. Let us consider the behavior of the flow at several significant stages. Before the collision with obstacle, the flow is laminar and has the identical physical properties in each point. After the collision of the fluid with the wall the process of flowing past the wall starts. Proximately in front of the obstacle, the flow velocity approaches zero. In time, the periodic vortex structure behind the obstacle arises. Under the flow development, it gradually stabilizes.

Under certain condition of flow (certain Reynolds numbers for different obstacles) it can take on form of the repeating vortices called Kármán vortex street, named in honor of the engineer and fluid dynamicist Theodore von Kármán.

Next, more advanced example of the obstacle is a sphere (recall that we work in two-dimensional space), see figure 7.16. In the case of the sphere, the exhibits more smooth behavior, vortices arise behind the vertical axis of symmetry and, compared to the case of the wall, outstand with less intensity and chaotic nature. Under the flow development, it gradually stabilizes, in the same way as for a wall flow.

Type of obstacles of our primary interest is the airfoil. Flow past the airfoil is of special interest for investigating by dint of computer simulations, in particular by FHP model. Knowing flow phenomena we may hypothesize some properties of airfoil, speculate about its aerodynamic quality. Our program makes possible to enter the angle of incidence for airfoil which, clearly, has an influence on the airfoil flow. In particular, we can determine the angle of flap vortex, an approximate lifting strength in terms of velocities near upper and lower surfaces.

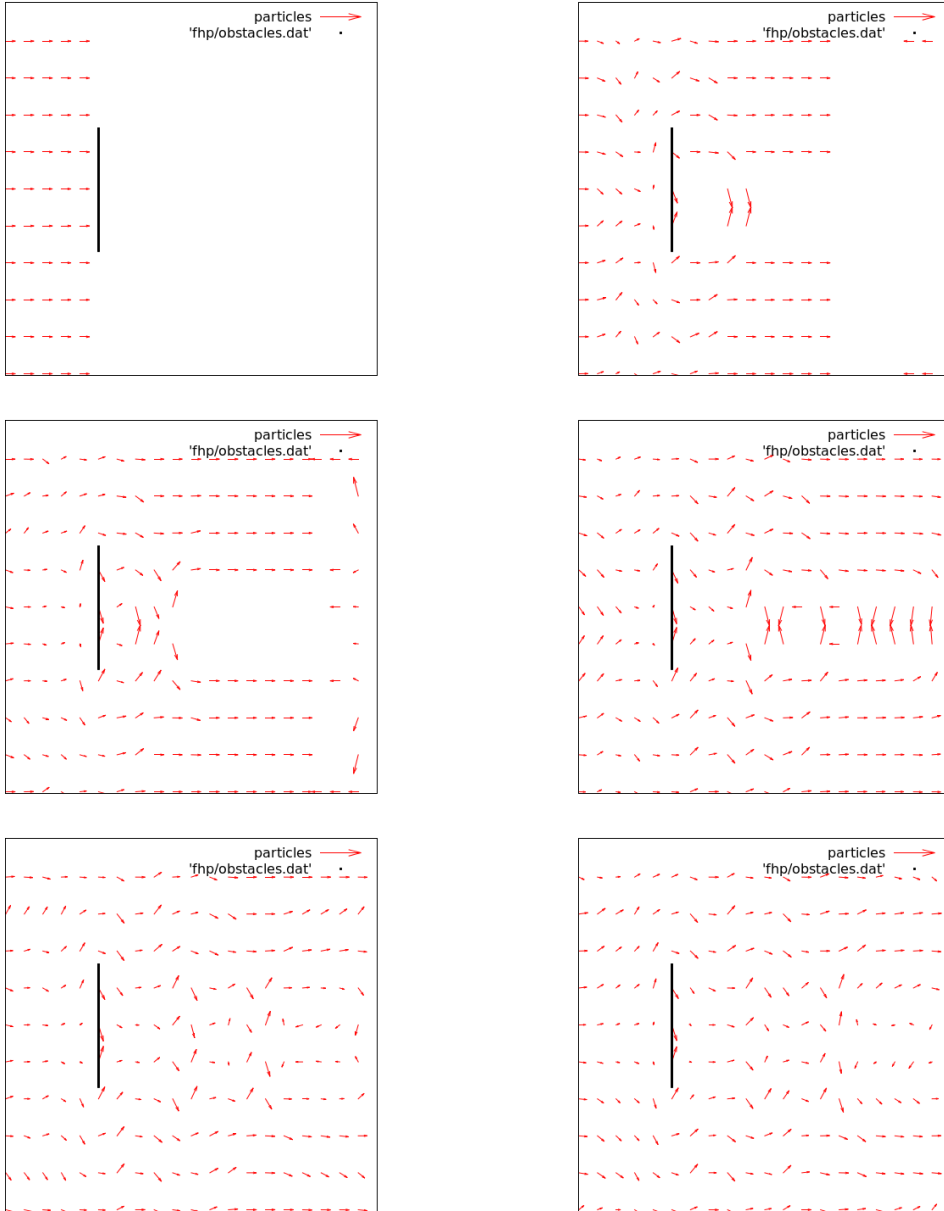


Figure 7.14: Flow past the wall, part I.

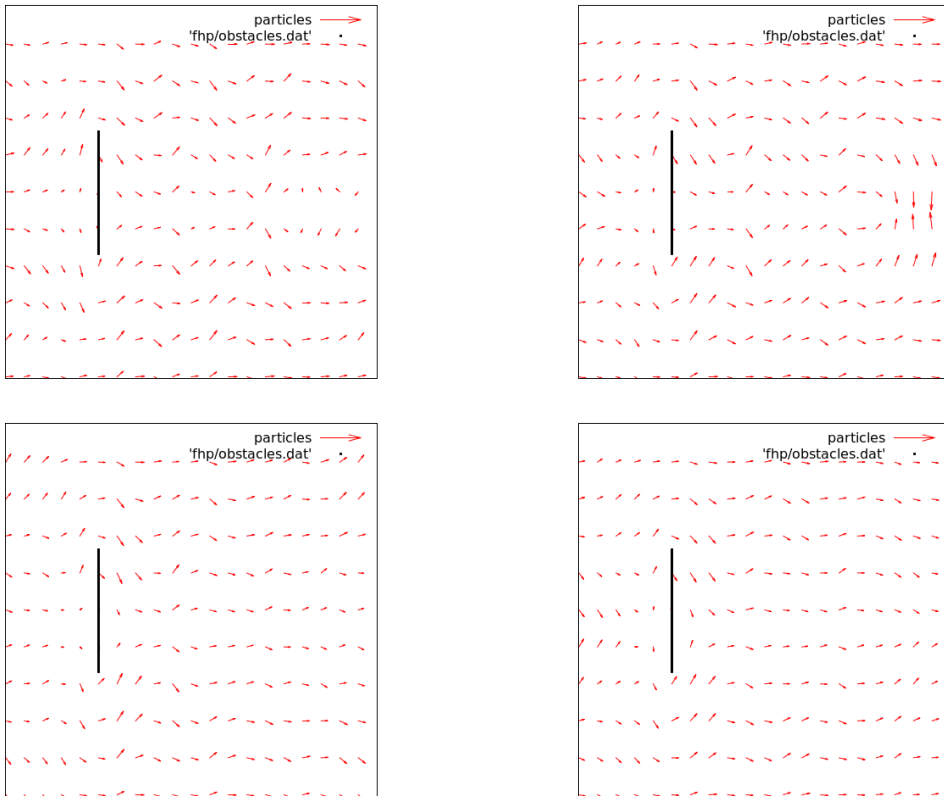
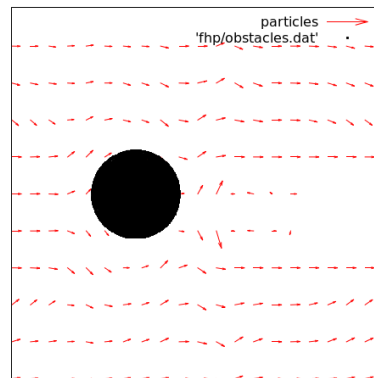
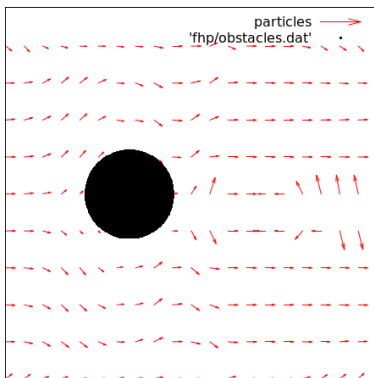
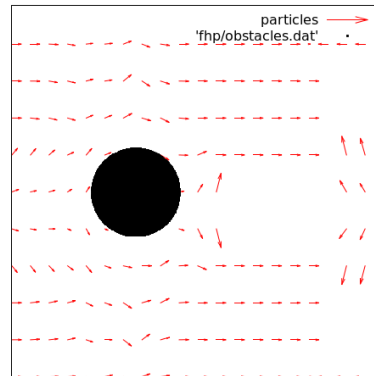
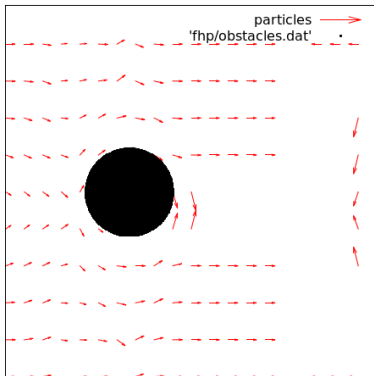
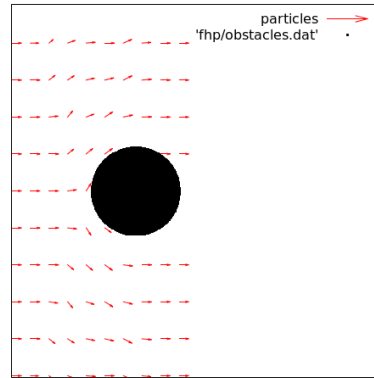
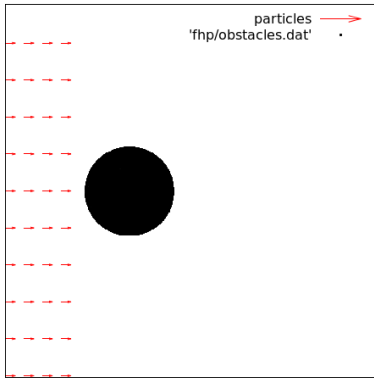
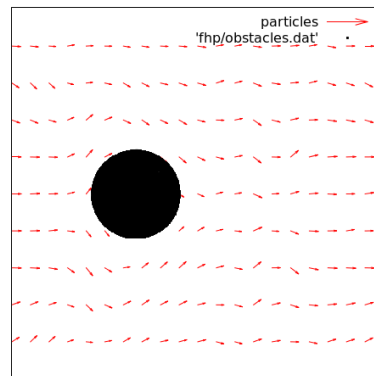
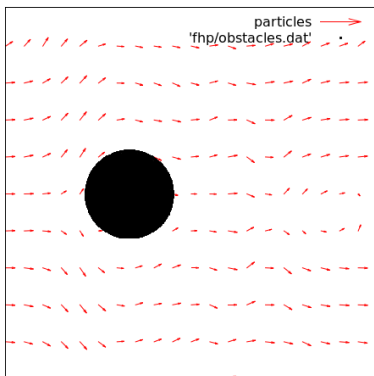
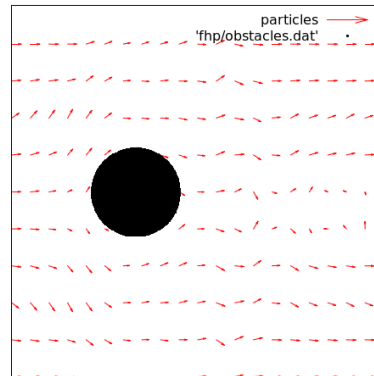
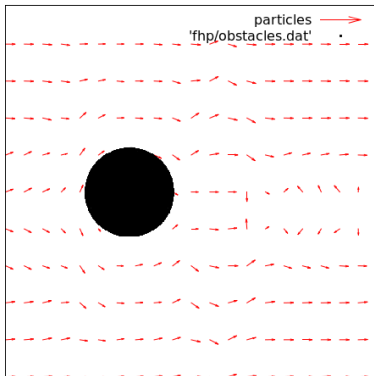
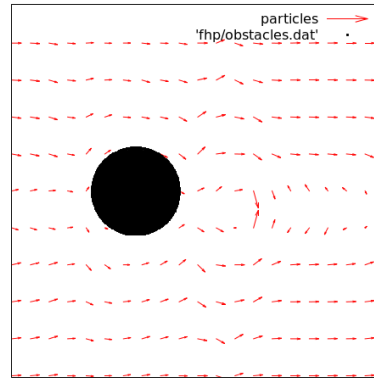
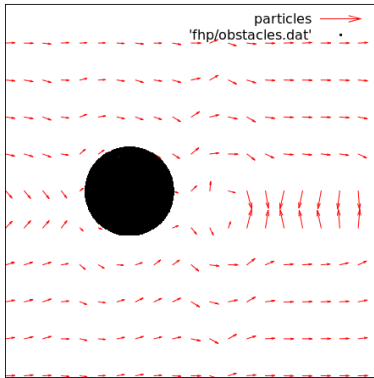


Figure 7.15: Flow past the wall, part II.





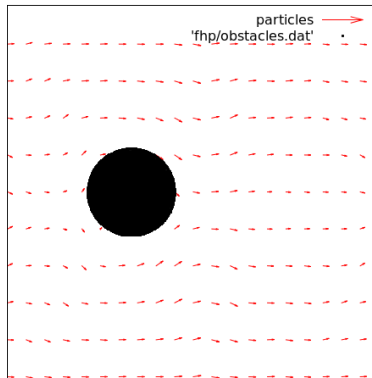
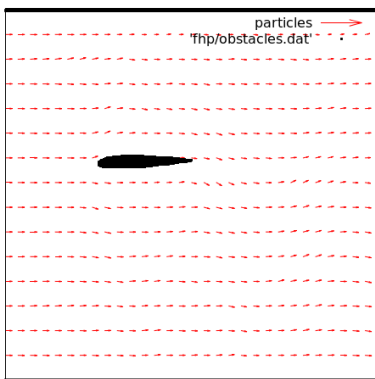
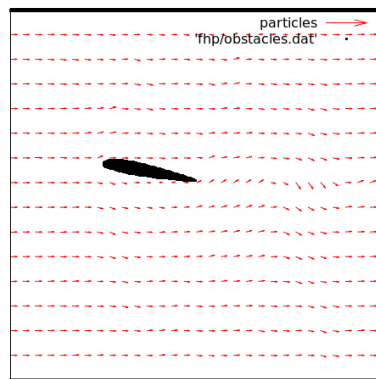


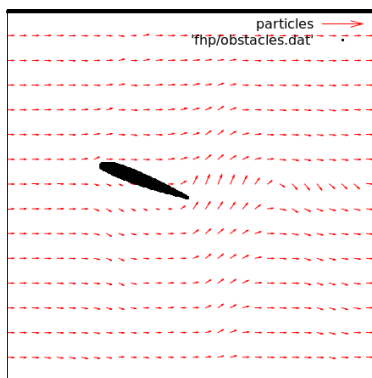
Figure 7.16: Sphere flow



$\alpha = -2.5^\circ$



$\alpha = 10^\circ$



$\alpha = 20^\circ$

Figure 7.17: Flow past the airfoil for different angles of attack (vector plot).

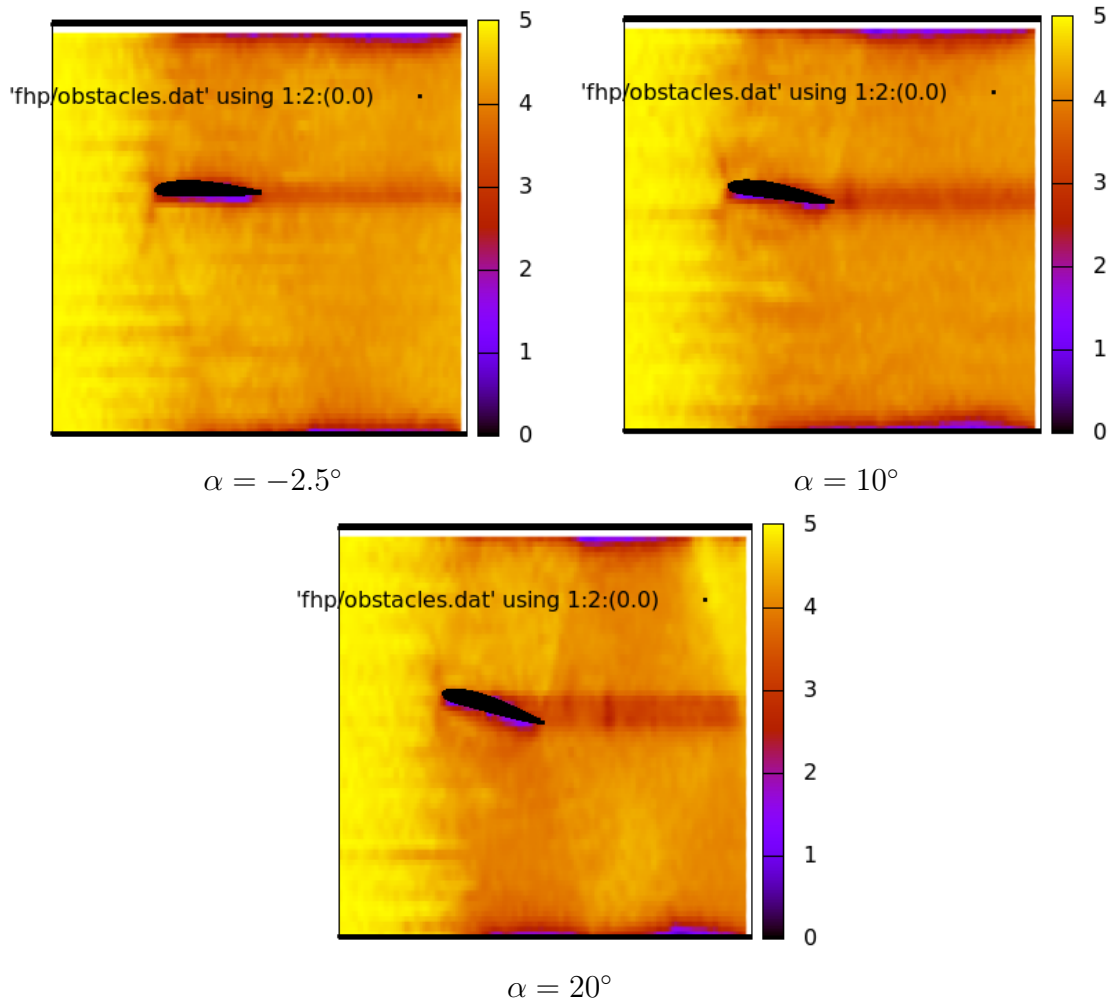


Figure 7.18: Flow past the airfoil for different angles of attack (scalar density plot).

In the figure 7.17 is shown the airfoil NACA-2418 flow for different angles of attack. It is evident that, under low angles of attack, the flow stays weakly turbulent. It must be stressed: flow velocity above upper surface of airfoil is higher than the flow velocity under lower surface. Vortex formation is observed on the certain profile depth, under the big angles of attack. Our program permits to visualize the flow not only in the vector mode, but also in the “color map plot” mode. We have employed two types of such plots. In the first one, the color of each point of the plot represents the magnitude of the velocity. In the second type, also the directions of the velocity are distinguished by different colors. For an illustration, see figures 7.18 and 7.19.

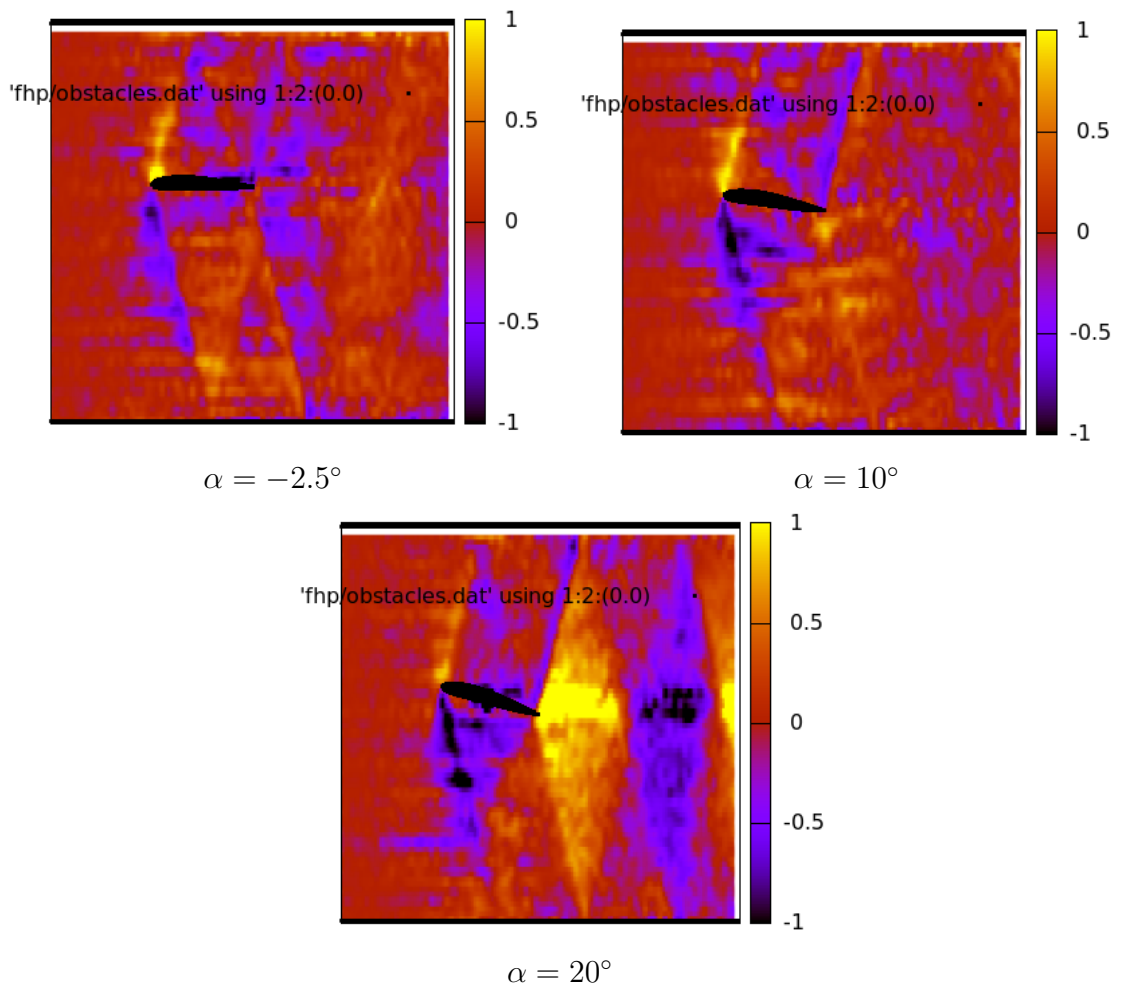


Figure 7.19: Flow past the airfoil for different angles of attack (vector density plot).

Appendix A

Demonstration notebook

In this demonstration notebook the final images for five different airfoils are presented. The calculations were made for three different angles of attack and the results are represented in three types of plot in order: vector plot, scalar density plot, vector density plot, which were described in previous chapter. The calculations for vector plot mode were made under the next conditions: $dx = 60$, $dy = 30$, `scale_factor = 60`, for scalar and vector density modes $dx = 15$, $dy = 15$. The order of matrix 1200x600. The boundary condition is chosen as `bc_free` for each calculation.

NACA66,2-(1.8)15.5

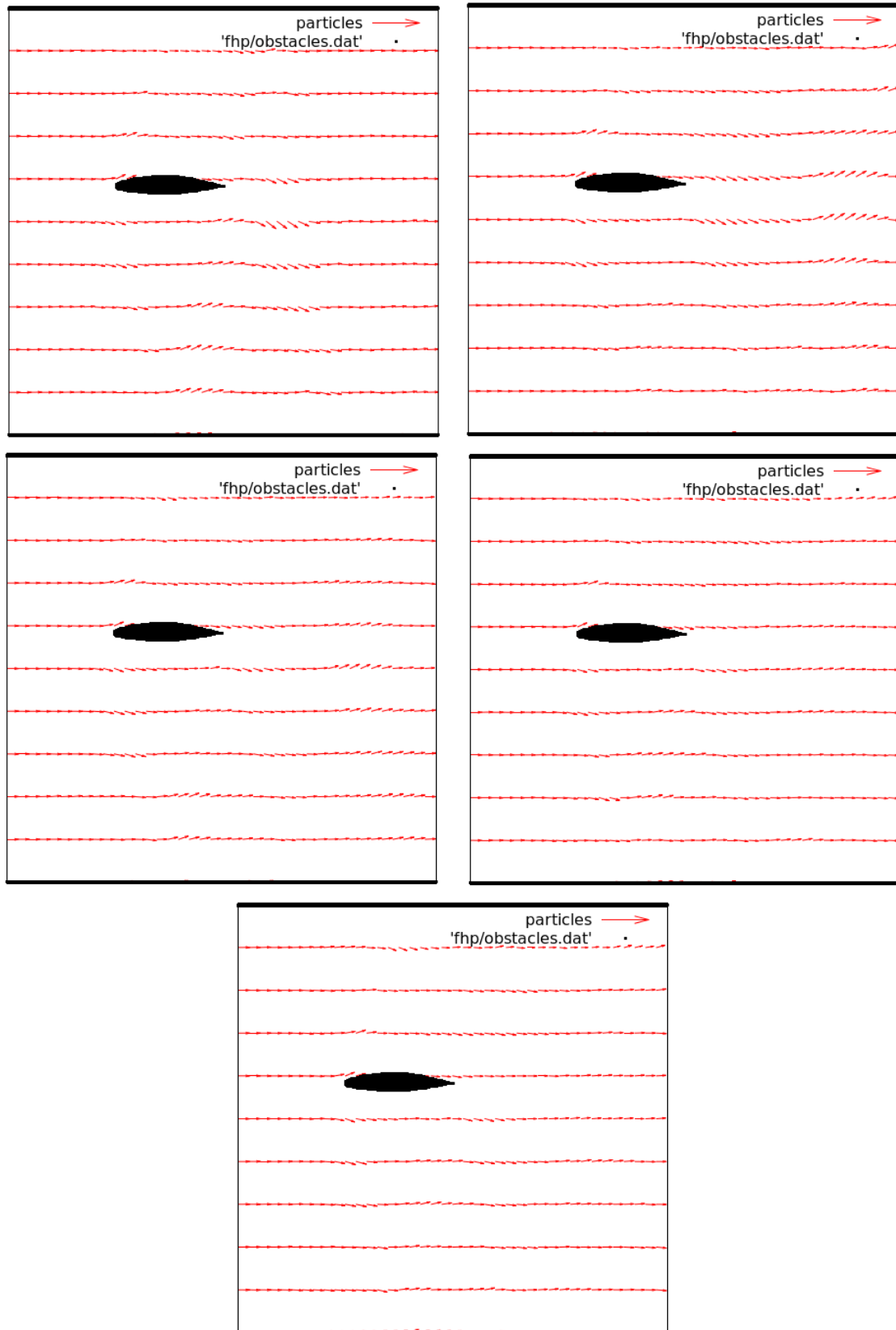


Figure A.1: Flow past the airfoil NACA66,2-(1.8)15.5 for $\alpha = 0$ (vector plot).

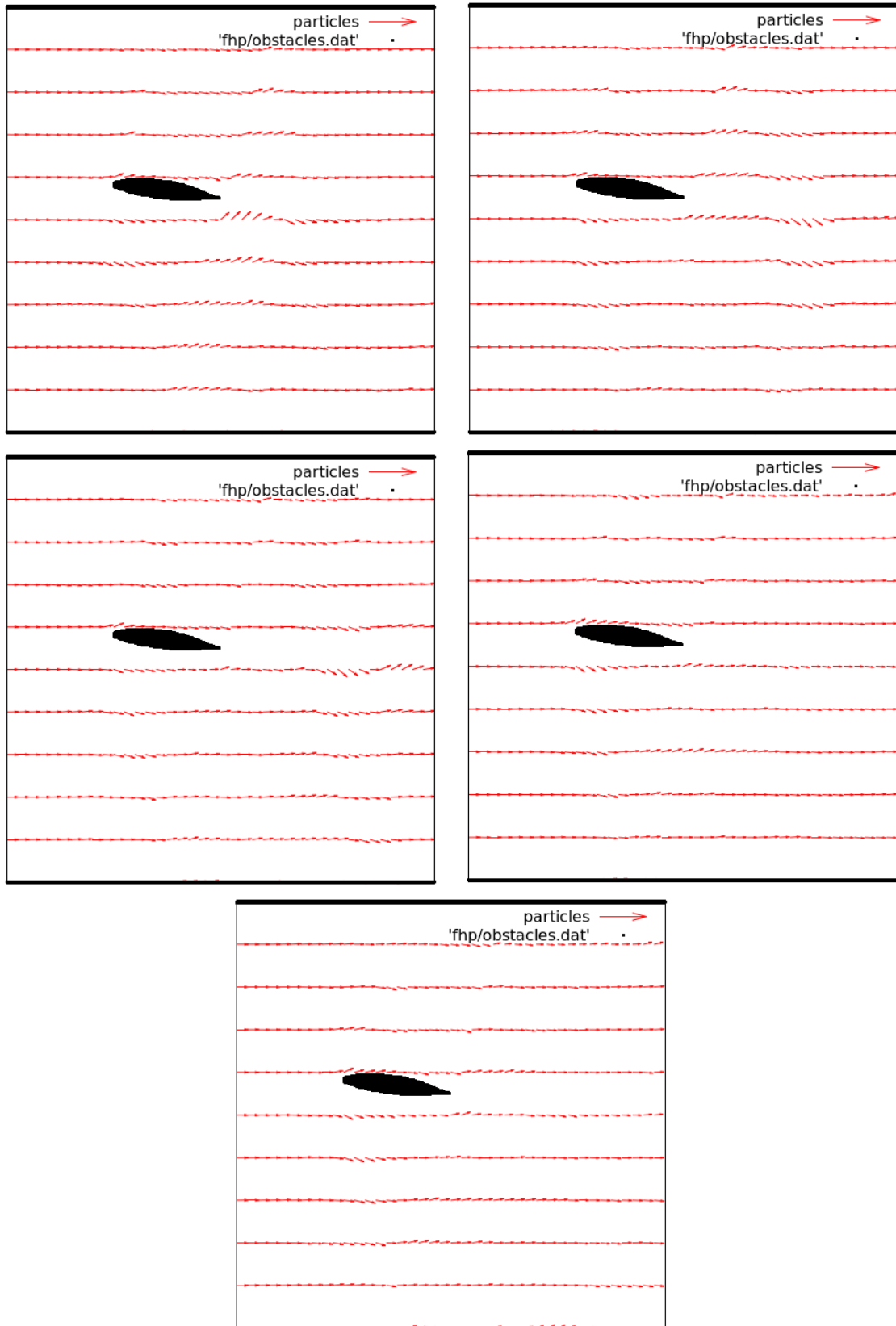


Figure A.2: Flow past the airfoil NACA66,2-(1.8)15.5 for $\alpha = 7.5$ (vector plot).

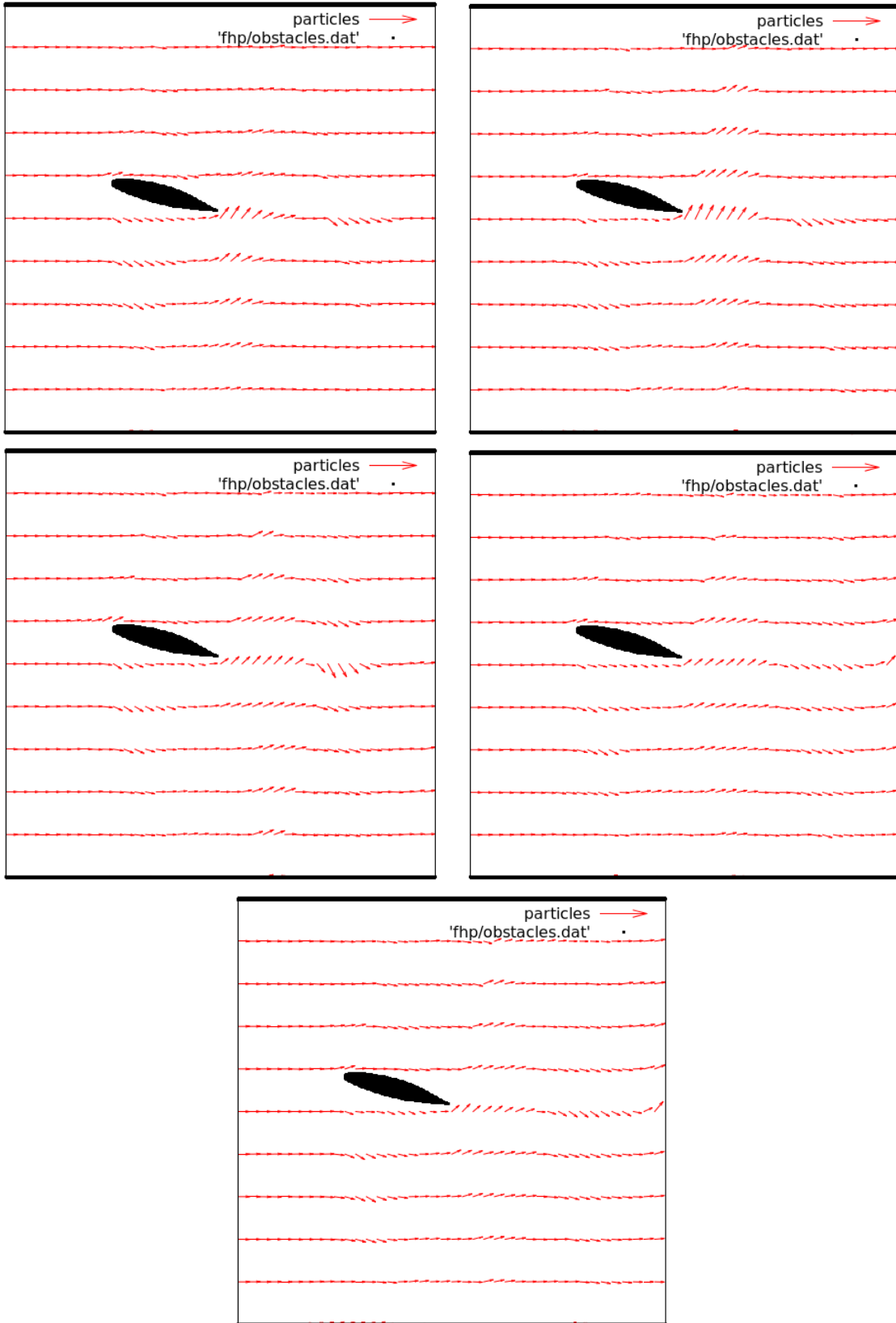


Figure A.3: Flow past the airfoil NACA66,2-(1.8)15.5 for $\alpha = 15$ (vector plot).

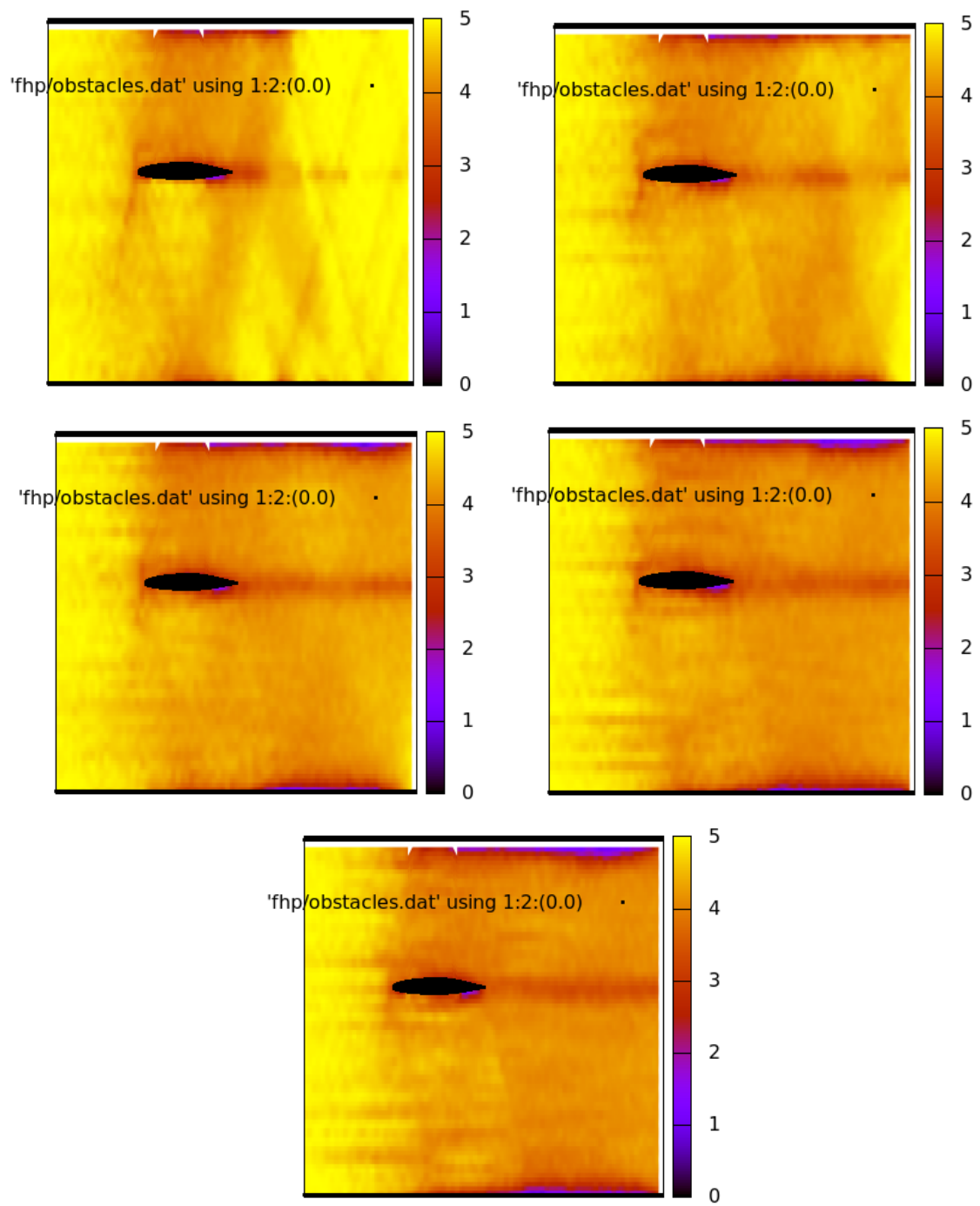


Figure A.4: Flow past the airfoil NACA66,2-(1.8)15.5 for $\alpha = 0$ (scalar density plot).

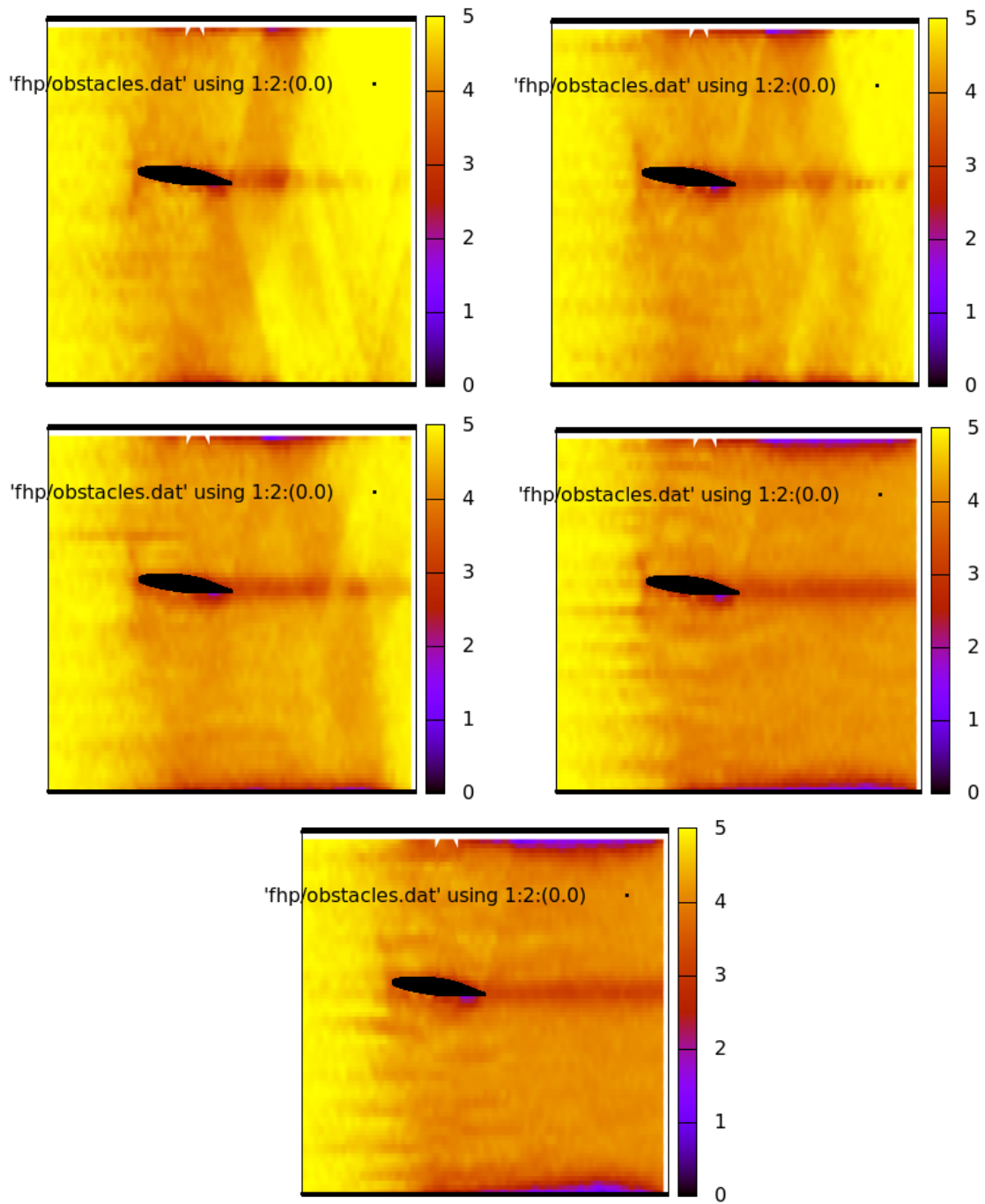


Figure A.5: Flow past the airfoil NACA66,2-(1.8)15.5 for $\alpha = 7.5$ (scalar density plot).

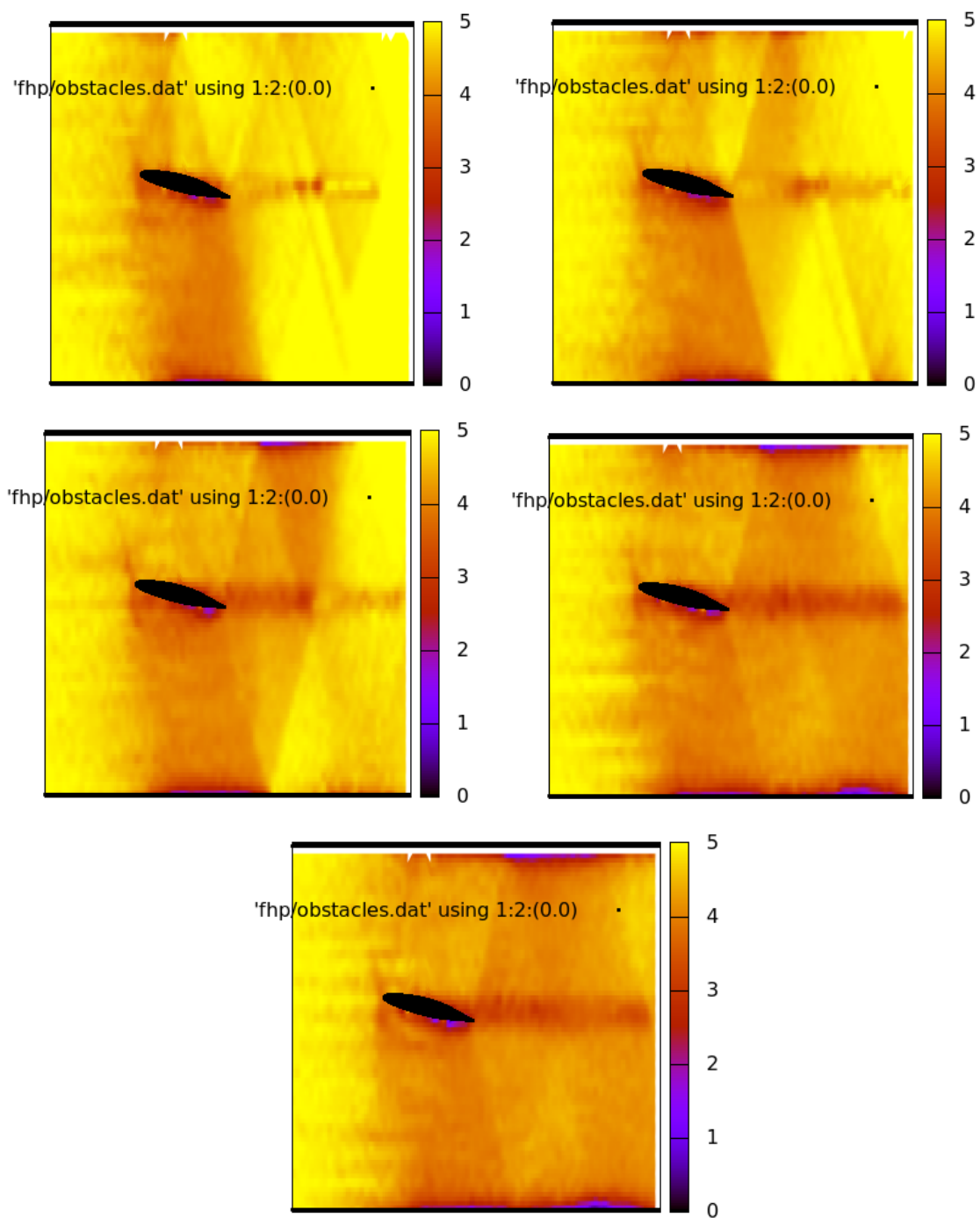


Figure A.6: Flow past the airfoil NACA66,2-(1.8)15.5 for $\alpha = 15$ (scalar density plot).

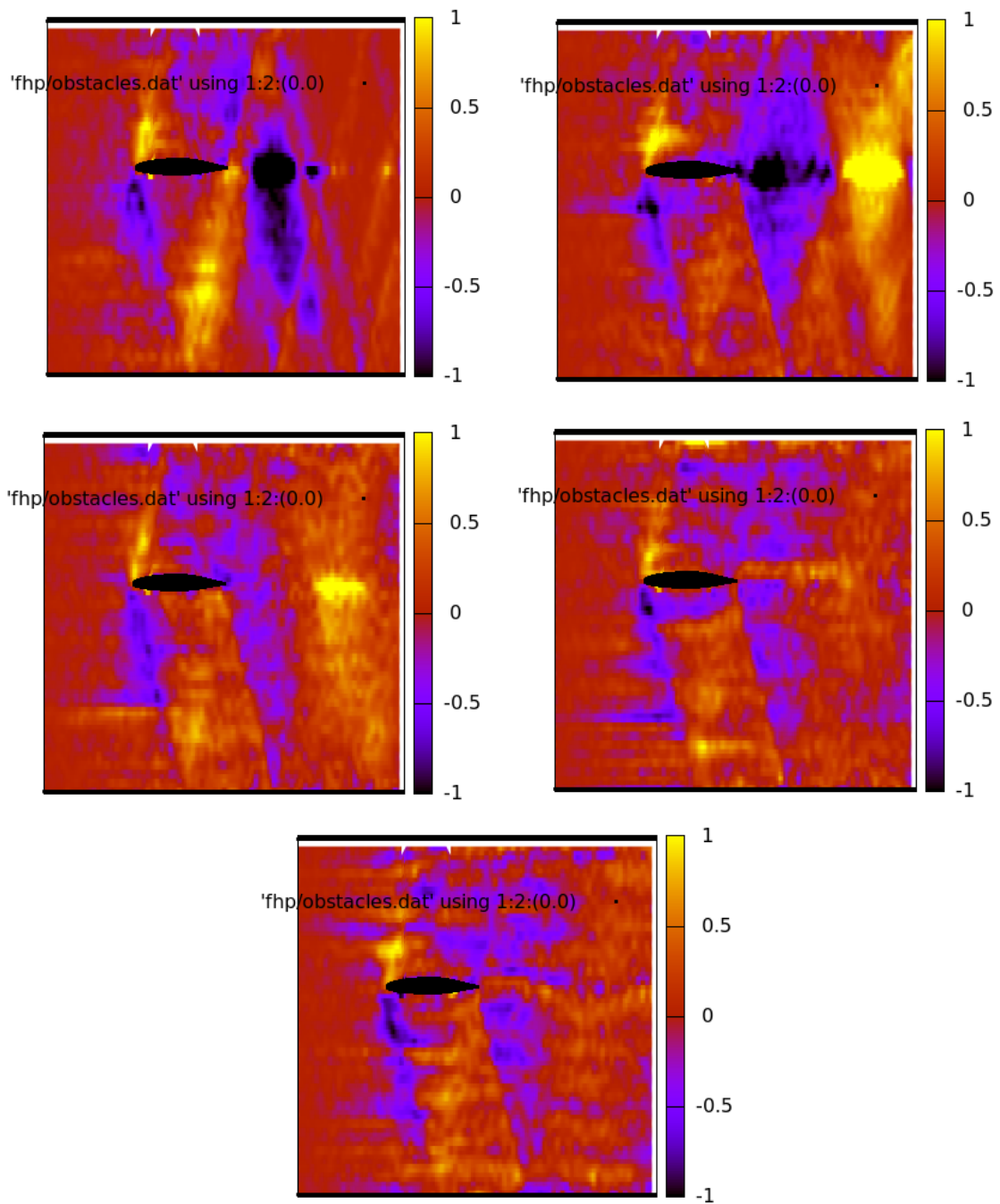


Figure A.7: Flow past the airfoil NACA66,2-(1.8)15.5 for $\alpha = 0$ (vector density plot).

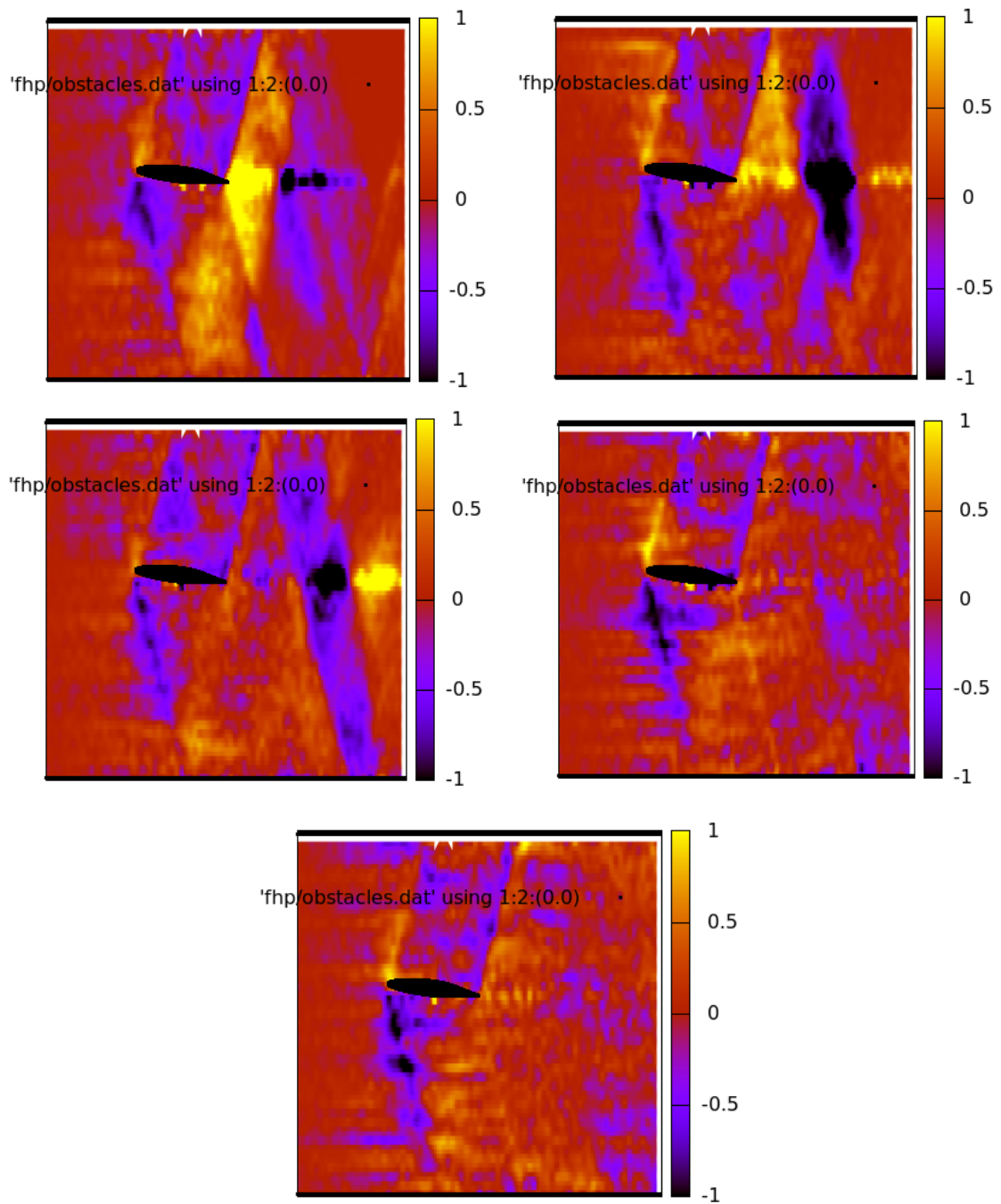


Figure A.8: Flow past the airfoil NACA66,2-(1.8)15.5 for $\alpha = 7.5$ (vector density plot).

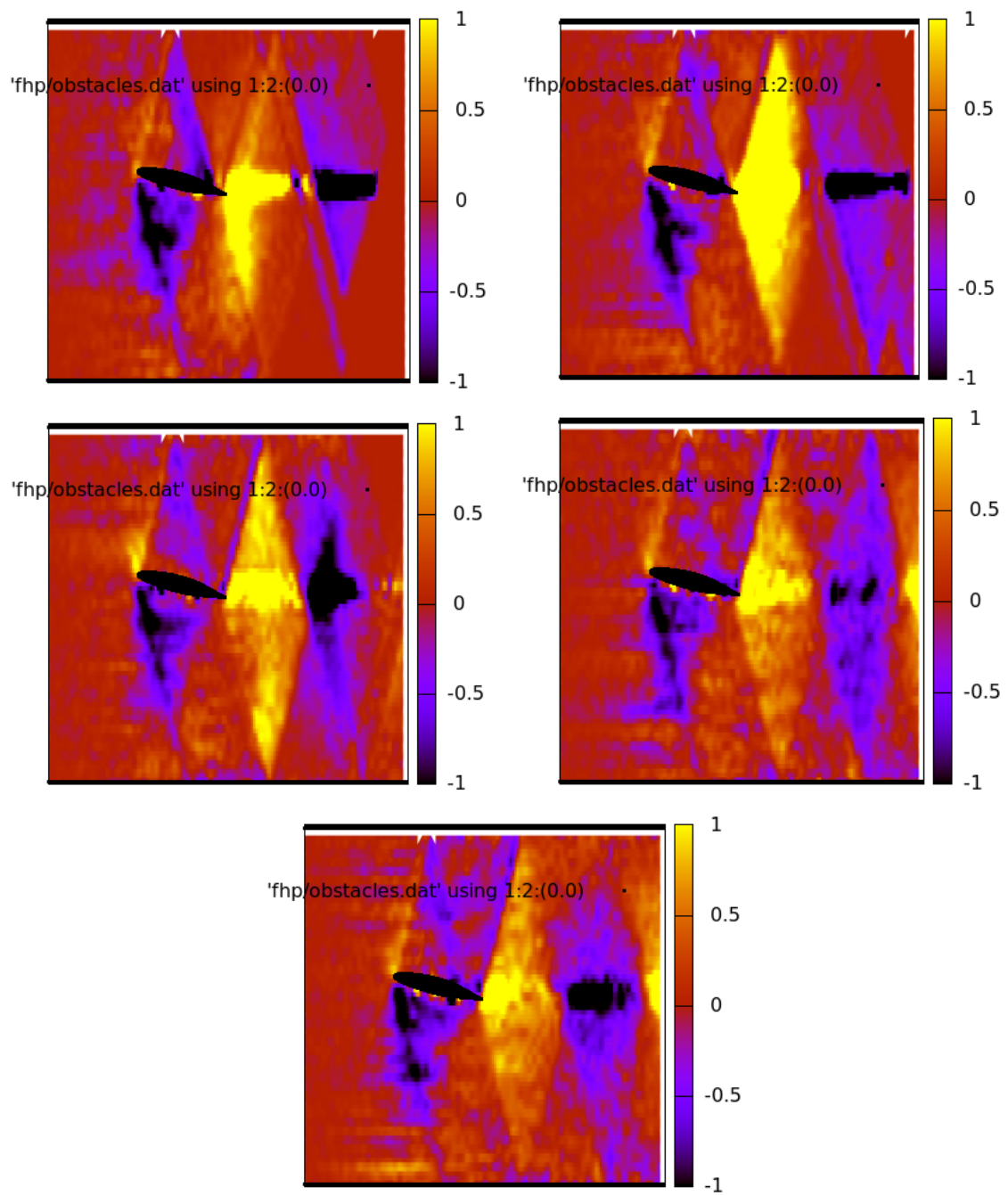


Figure A.9: Flow past the airfoil NACA66,2-(1.8)15.5 for $\alpha = 15$ (vector density plot).

Clark YH

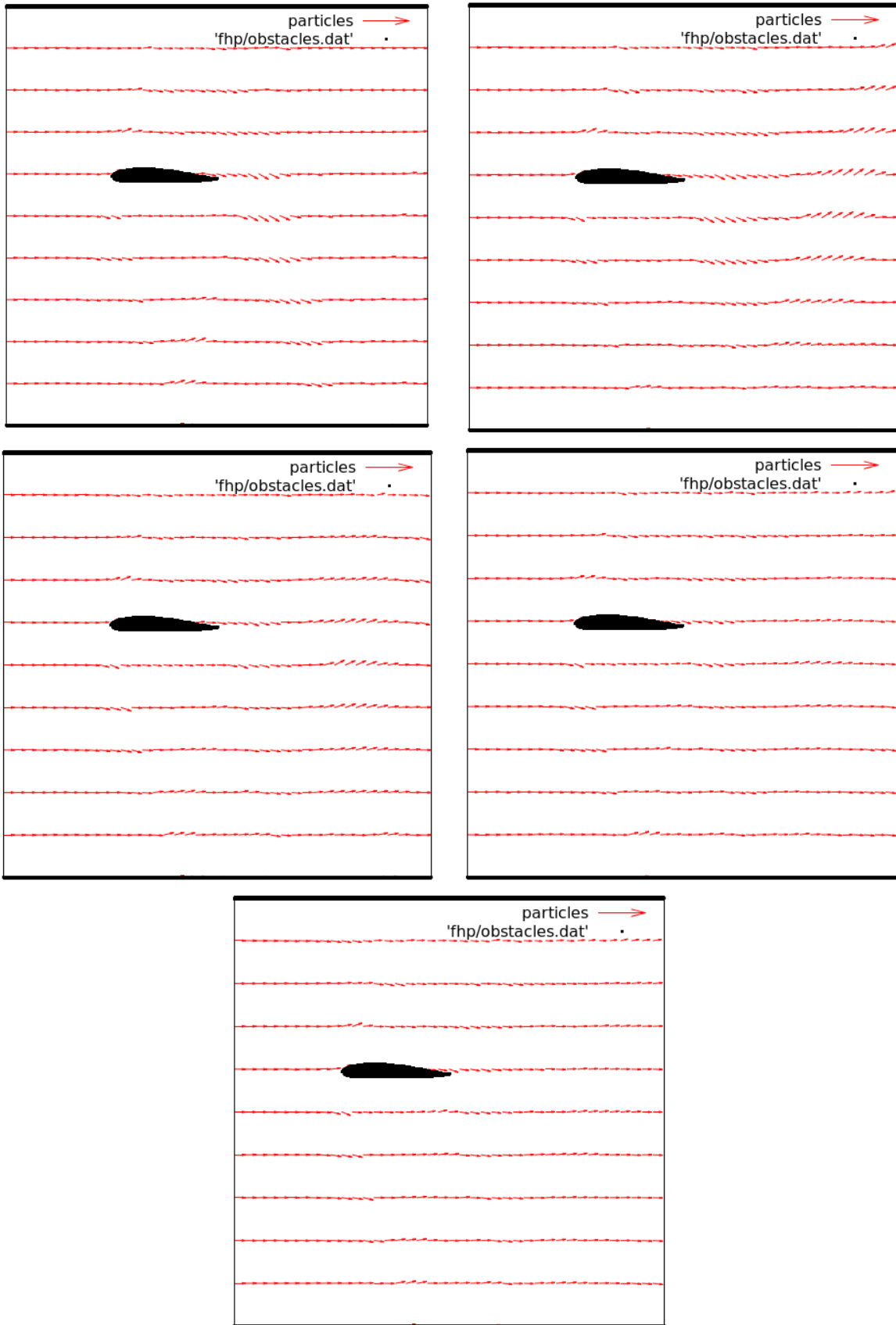


Figure A.10: Flow past the airfoil Clark YH for $\alpha = 0$ (vector plot).

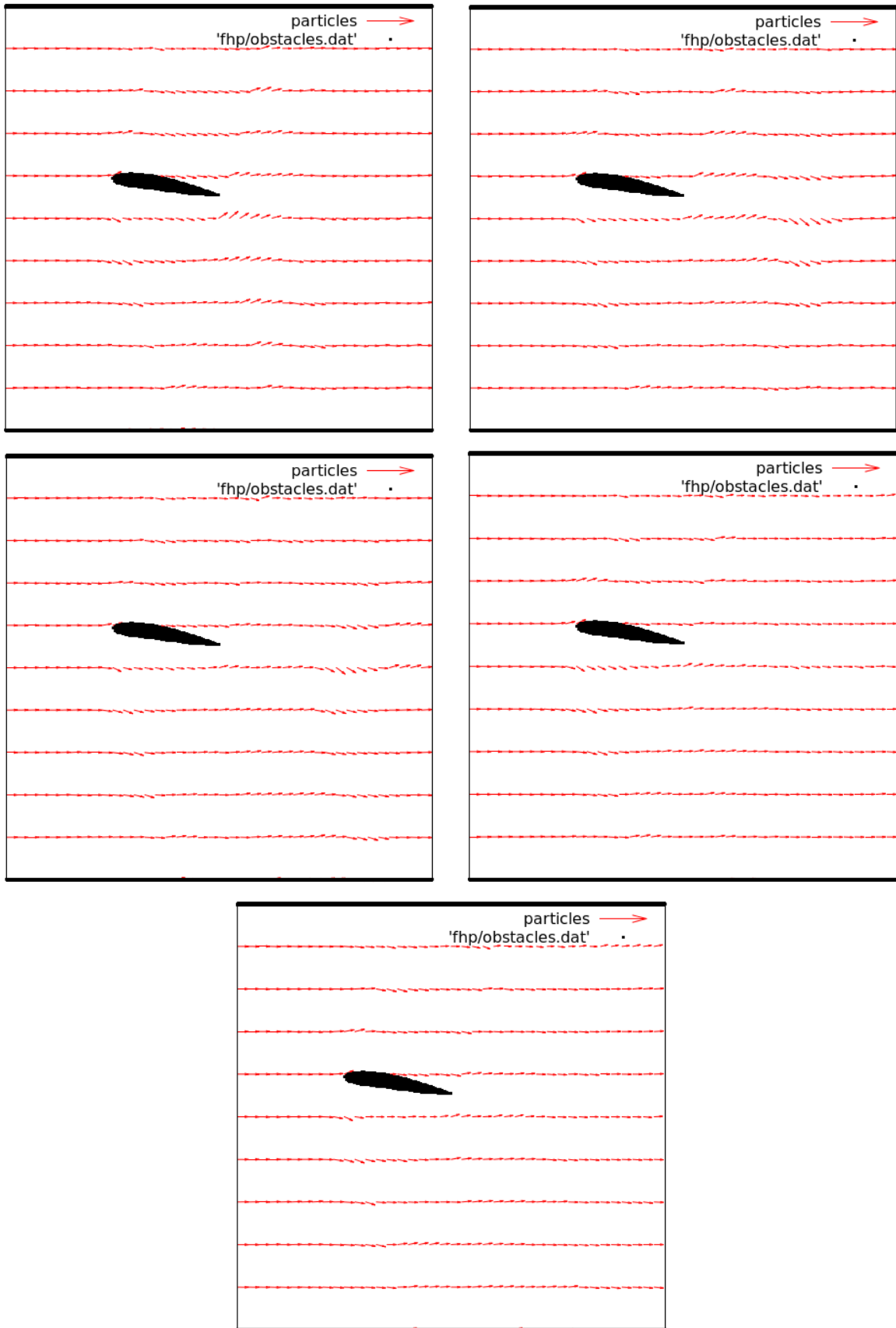


Figure A.11: Flow past the airfoil Clark YH for $\alpha = 7.5$ (vector plot).

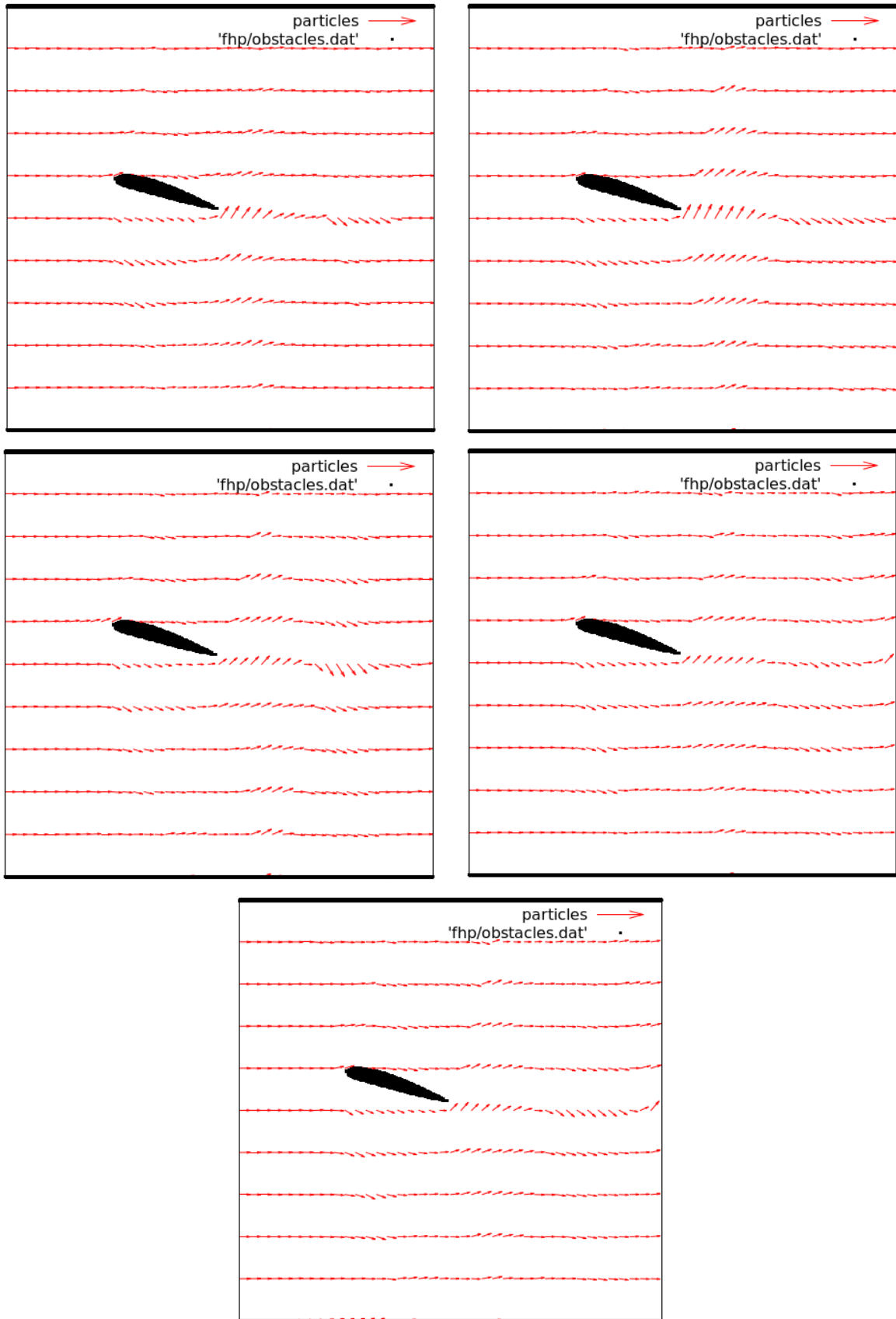


Figure A.12: Flow past the airfoil Clark YH for $\alpha = 15$ (vector plot).

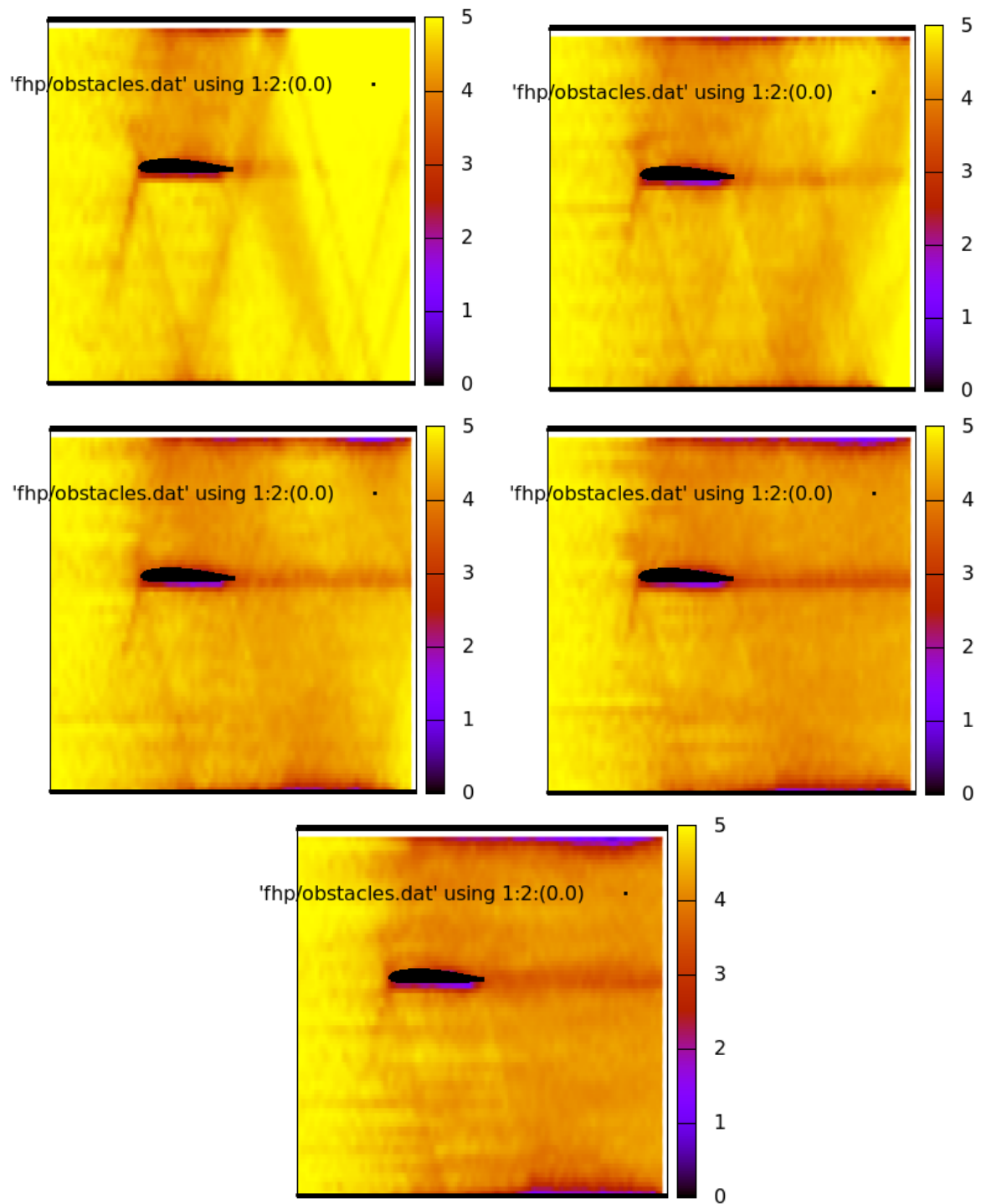


Figure A.13: Flow past the airfoil Clark YH for $\alpha = 0$ (scalar density plot).

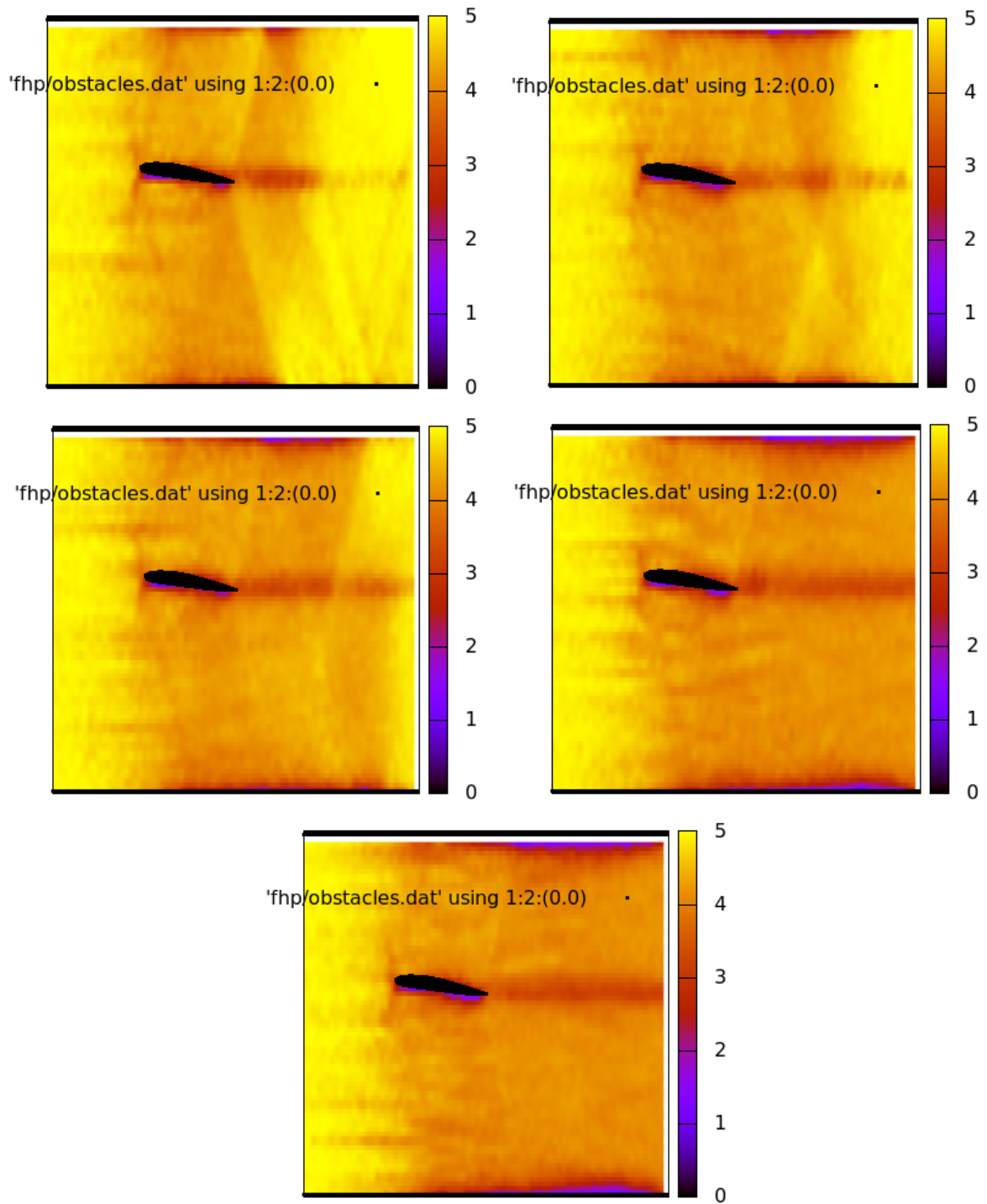


Figure A.14: Flow past the airfoil Clark YH for $\alpha = 7.5$ (scalar density plot).

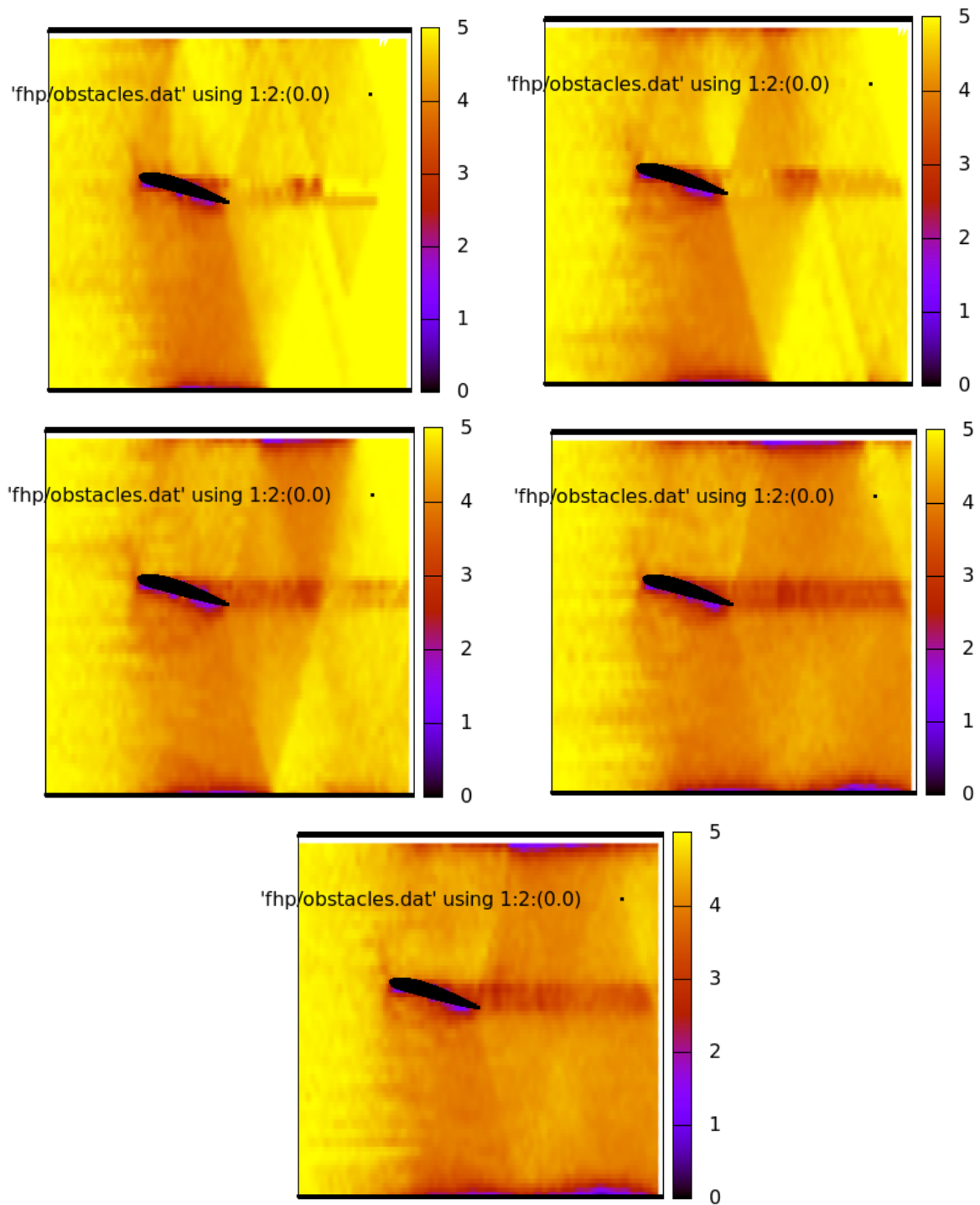


Figure A.15: Flow past the airfoil Clark YH for $\alpha = 15$ (scalar density plot).

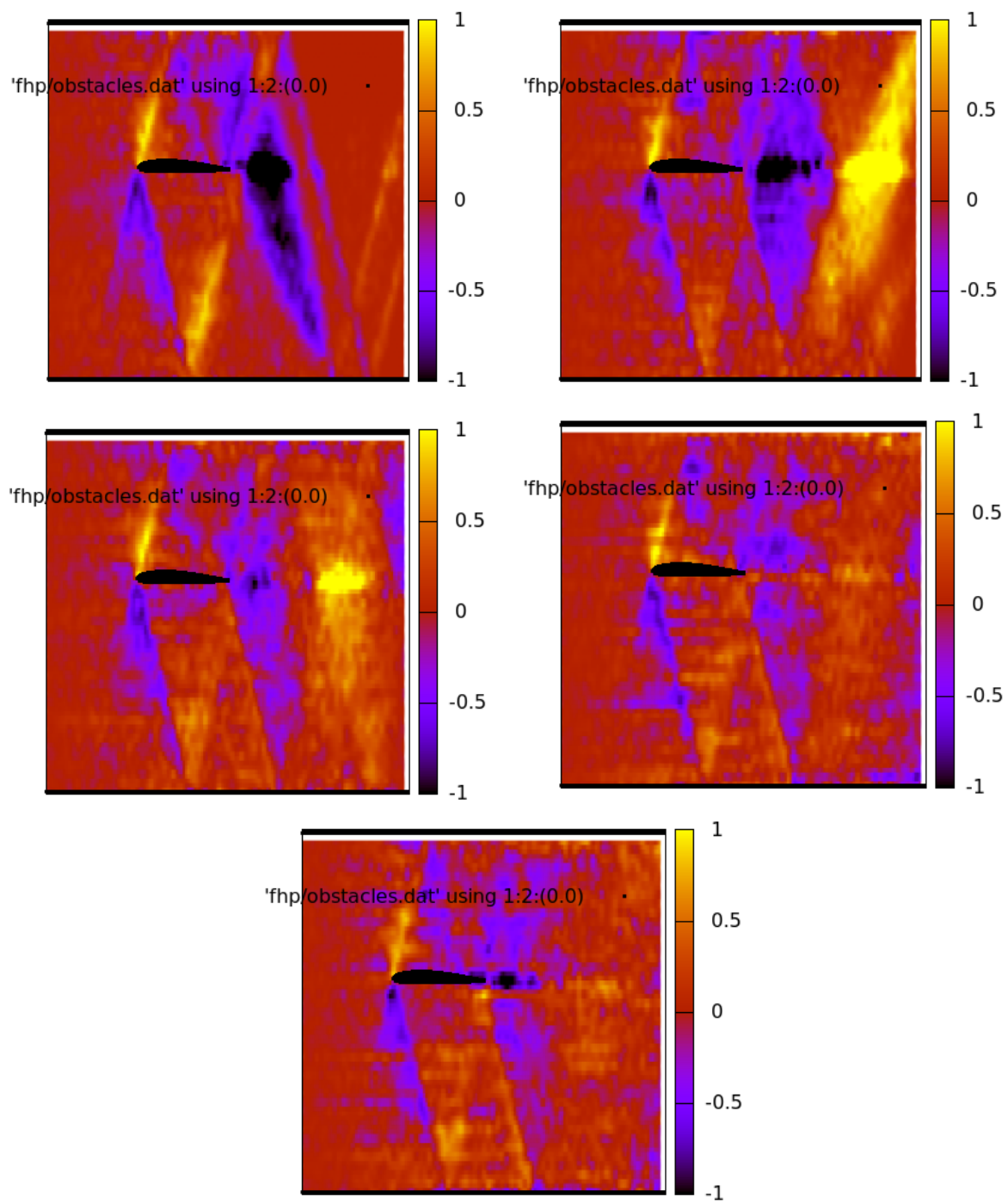


Figure A.16: Flow past the airfoil Clark YH for $\alpha = 0$ (vector density plot).

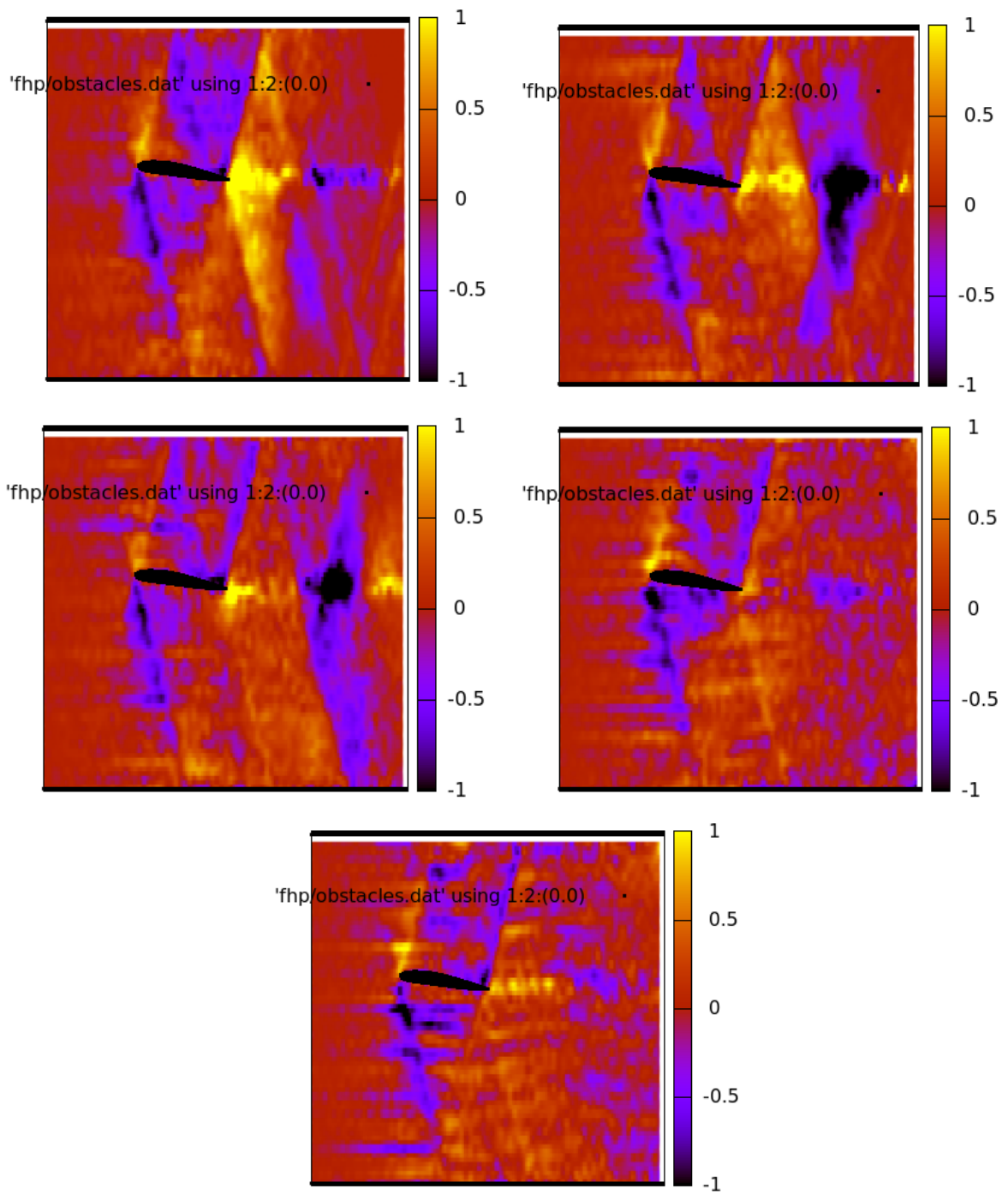


Figure A.17: Flow past the airfoil Clark YH for $\alpha = 7.5$ (vector density plot).

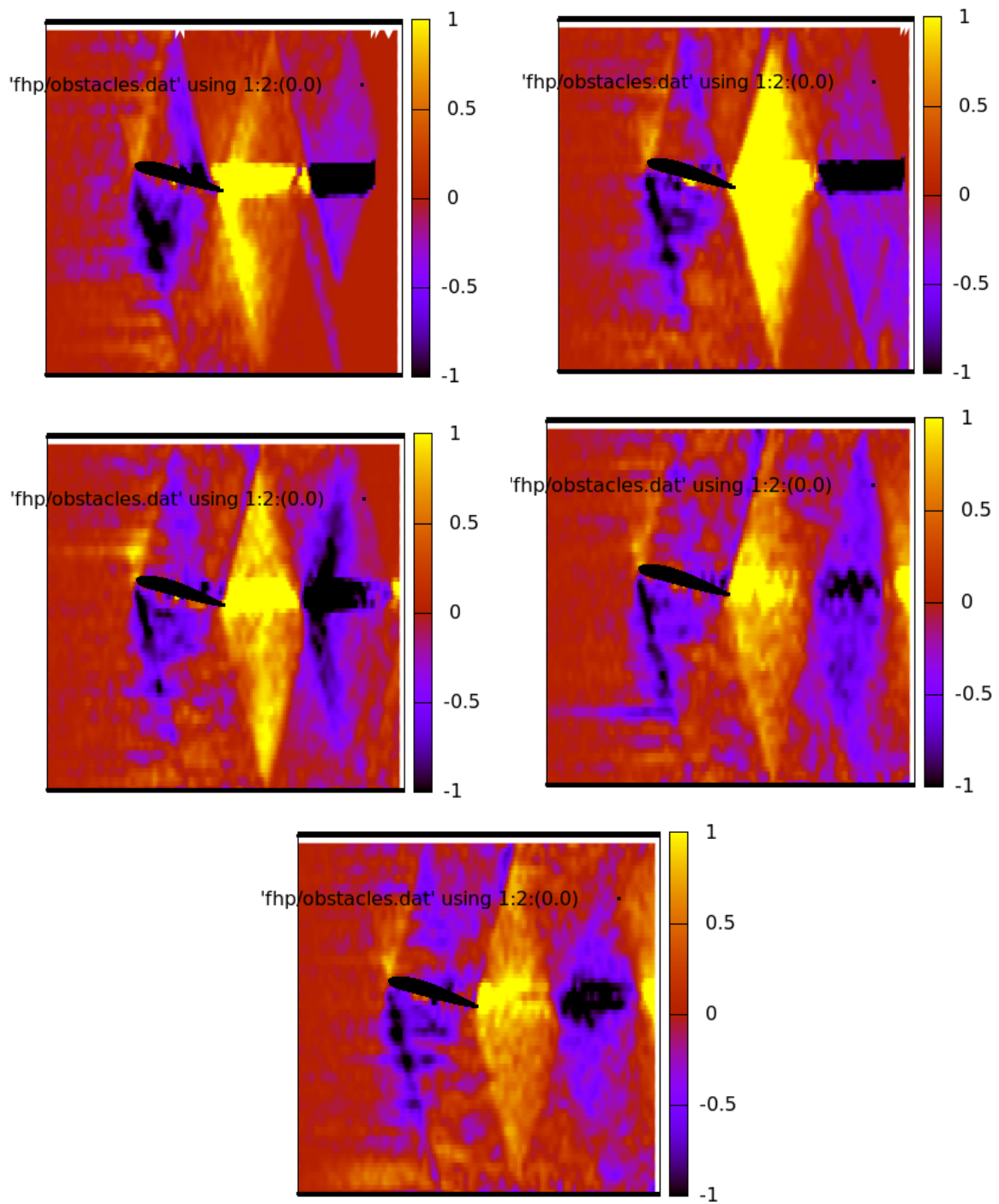


Figure A.18: Flow past the airfoil Clark YH for $\alpha = 15$ (vector density plot).

NACA2418

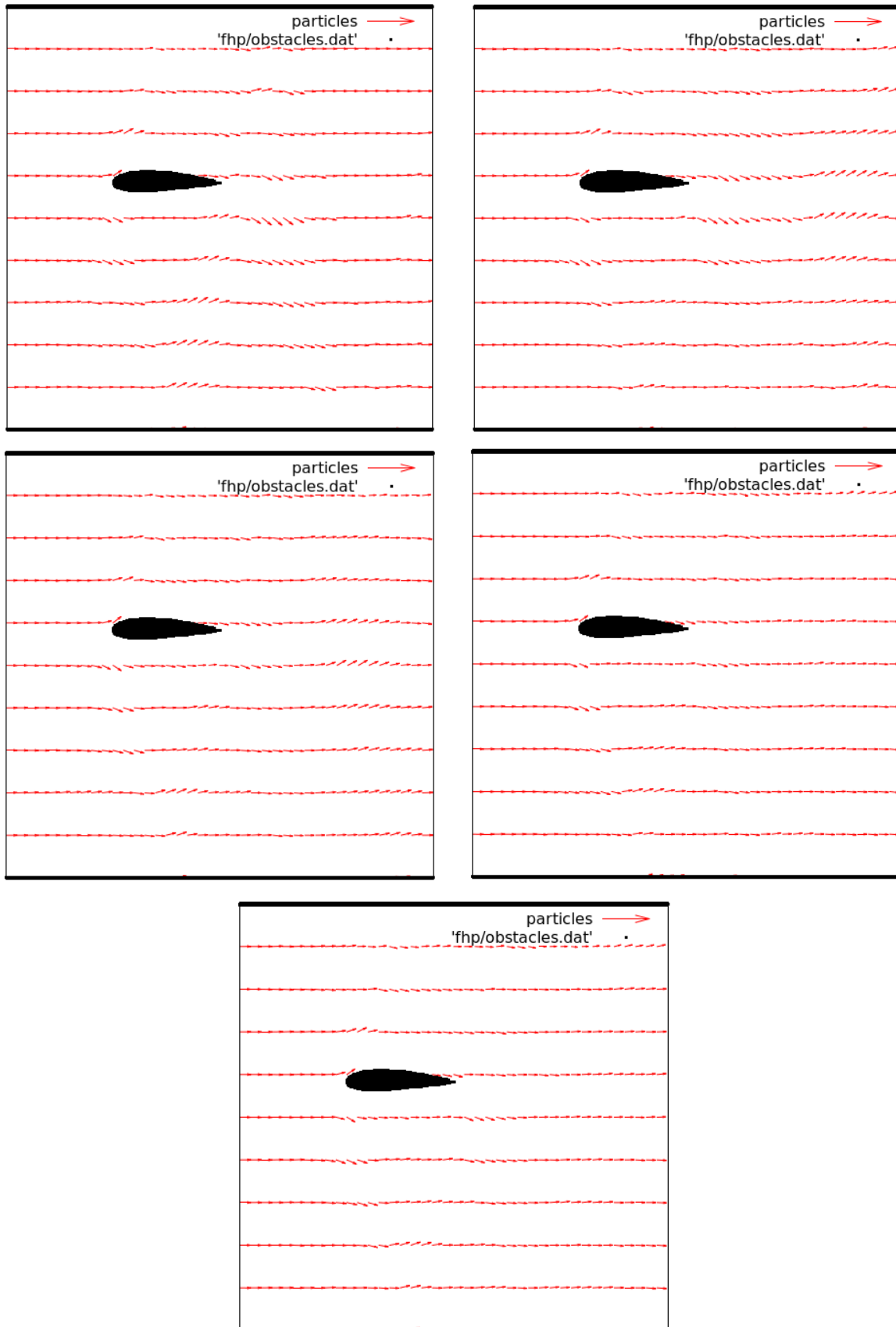


Figure A.19: Flow past the airfoil NACA2418 for $\alpha = 0$ (vector plot).

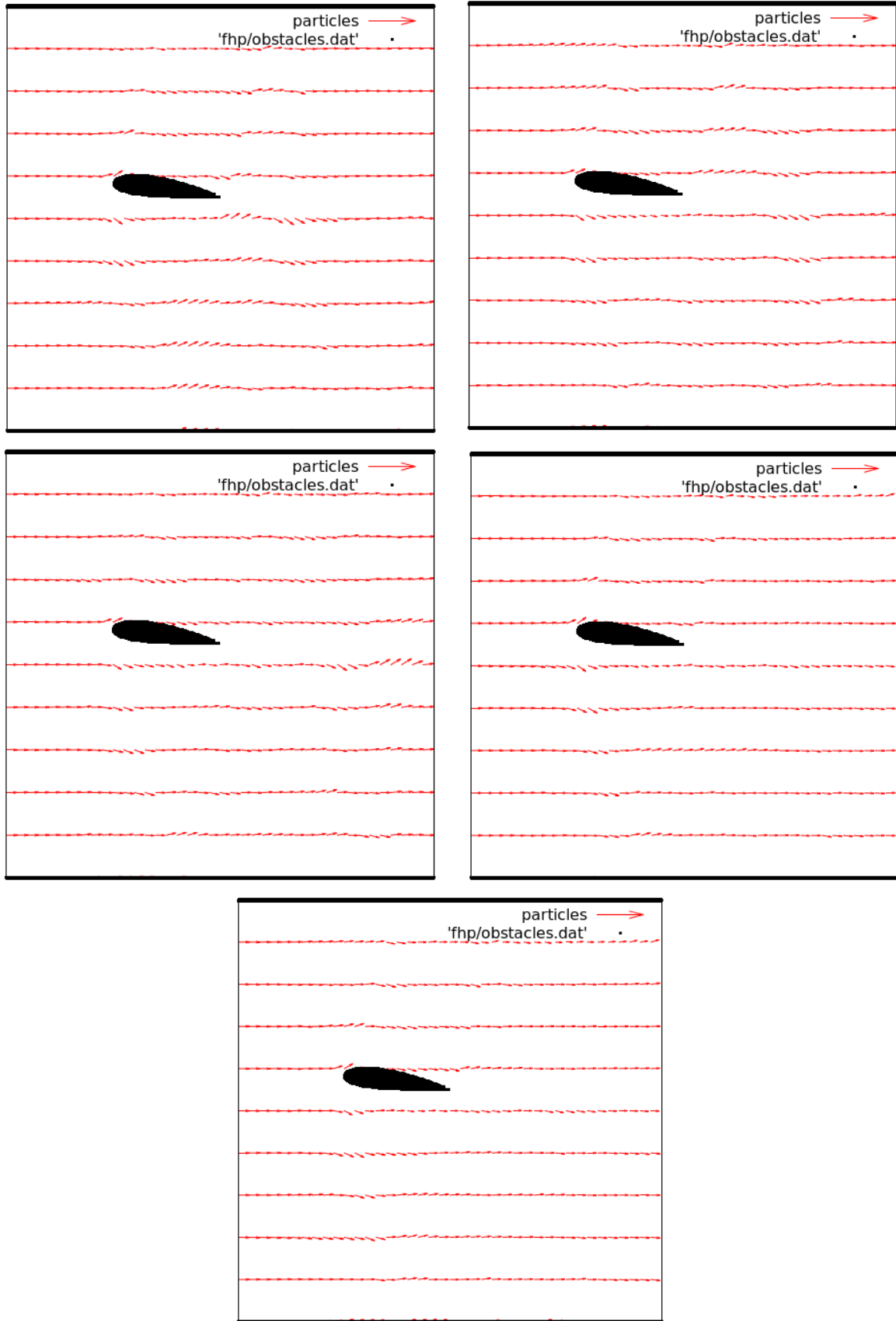


Figure A.20: Flow past the airfoil NACA2418 for $\alpha = 7.5$ (vector plot).

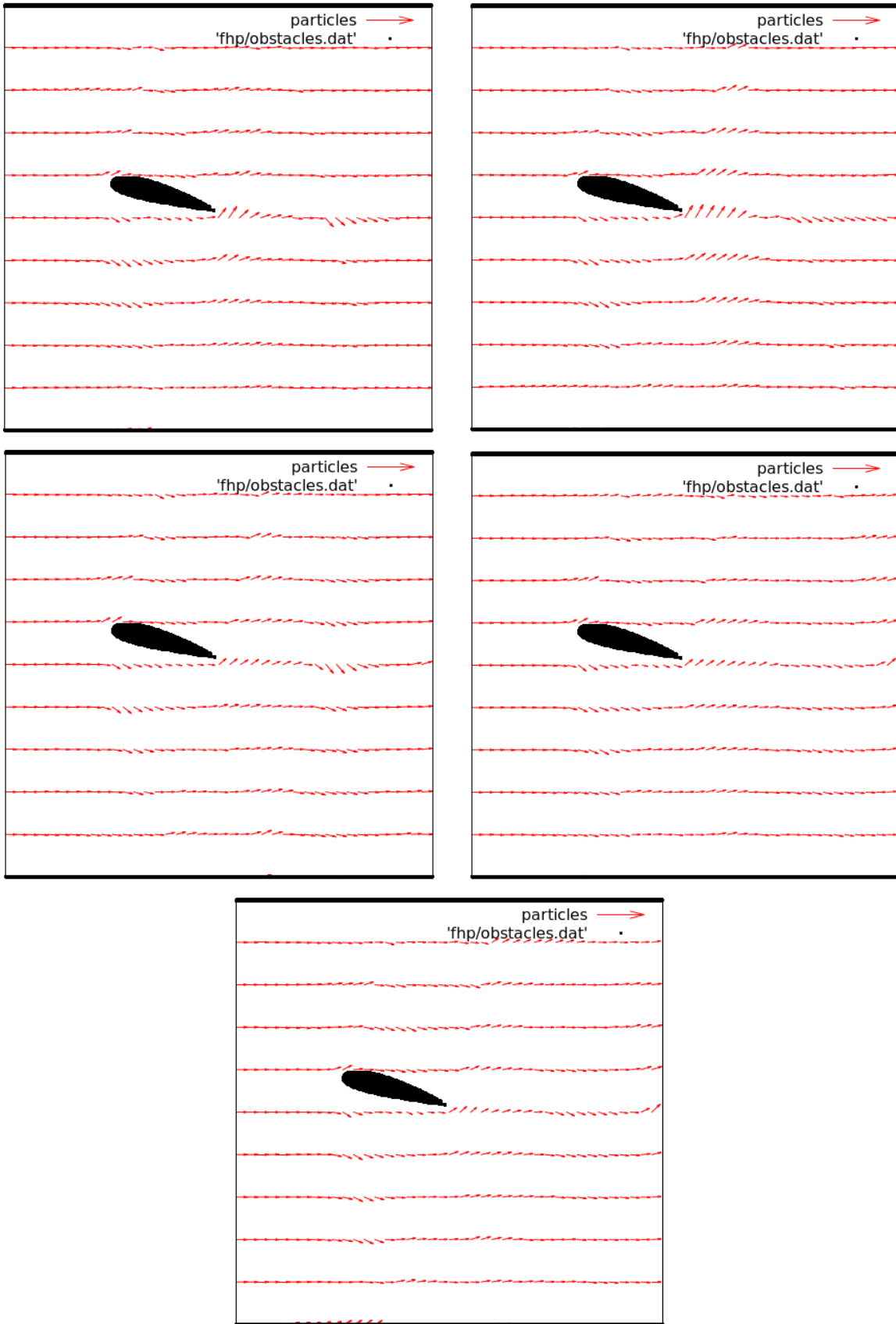


Figure A.21: Flow past the airfoil NACA2418 for $\alpha = 15$ (vector plot).

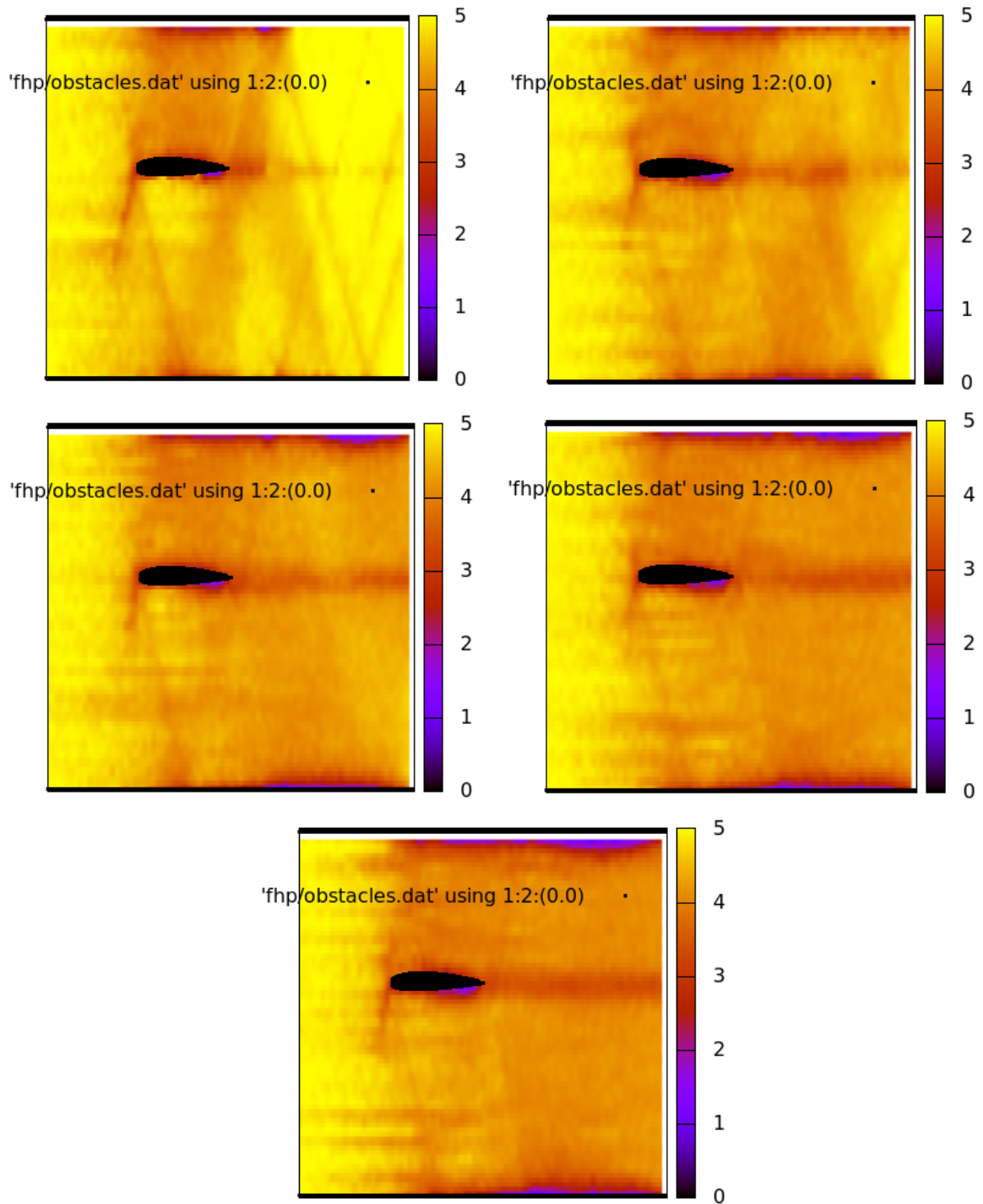


Figure A.22: Flow past the airfoil NACA2418 for $\alpha = 0$ (scalar density plot).

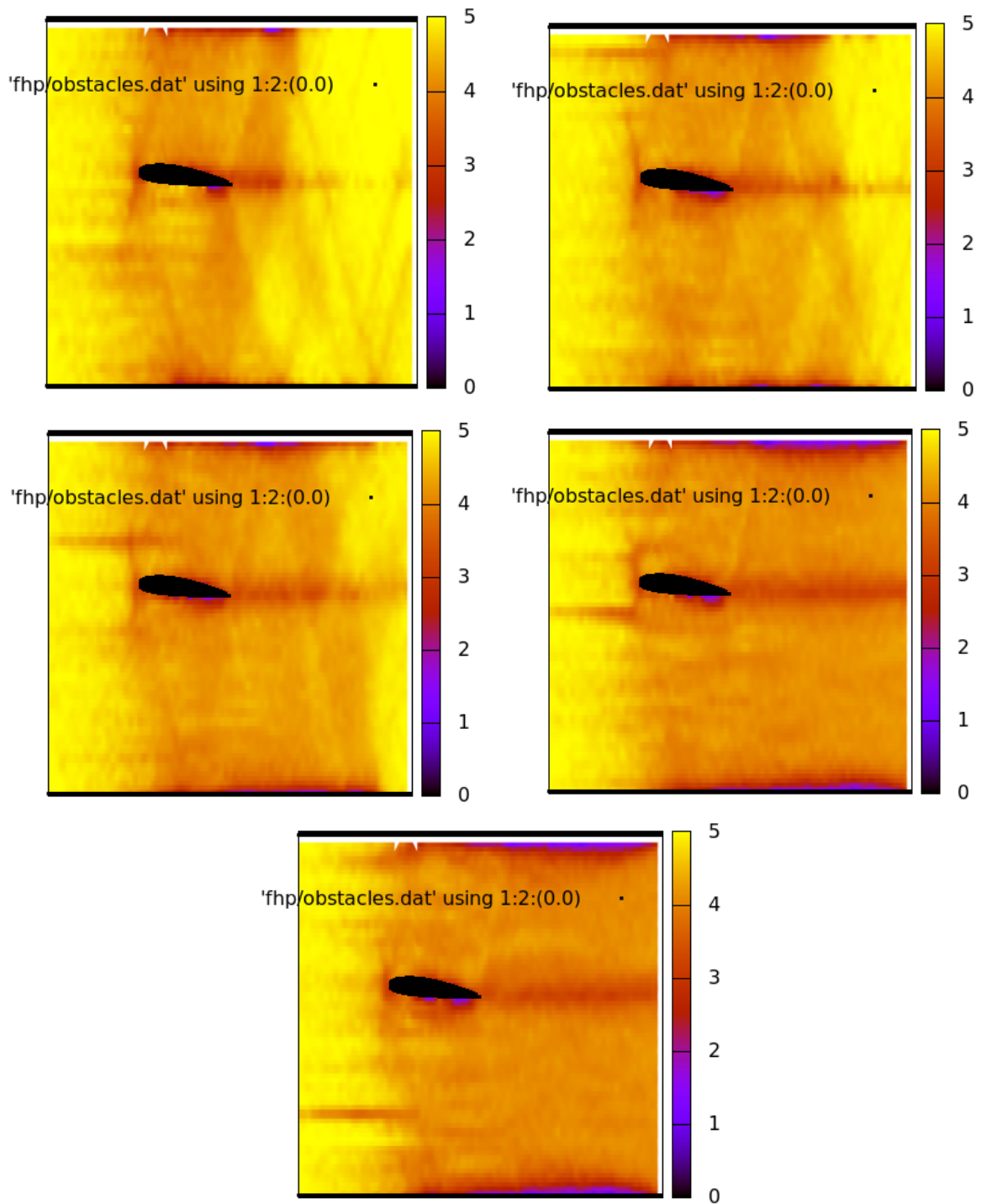


Figure A.23: Flow past the airfoil NACA2418 for $\alpha = 7.5$ (scalar density plot).

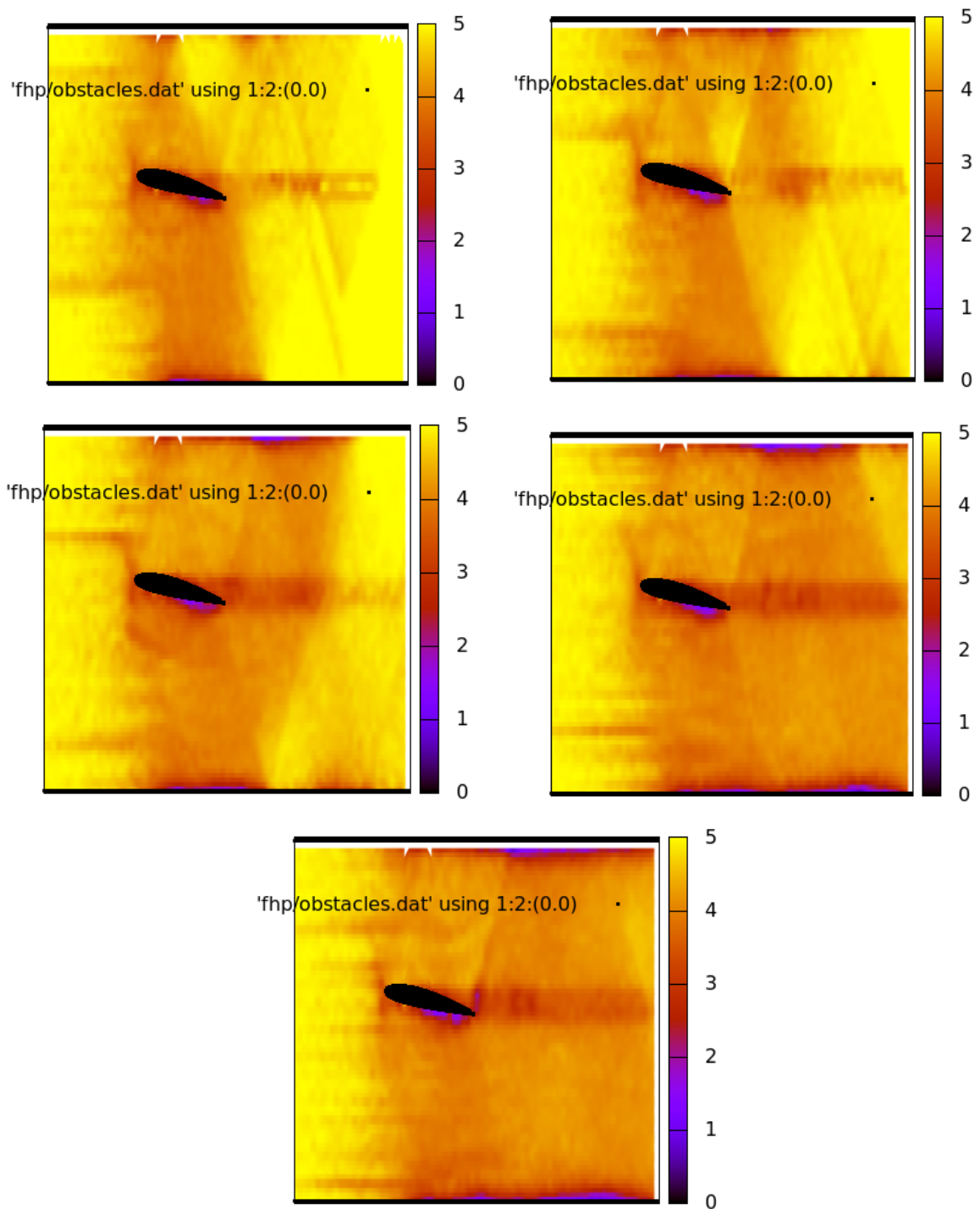


Figure A.24: Flow past the airfoil NACA2418 for $\alpha = 15$ (scalar density plot).

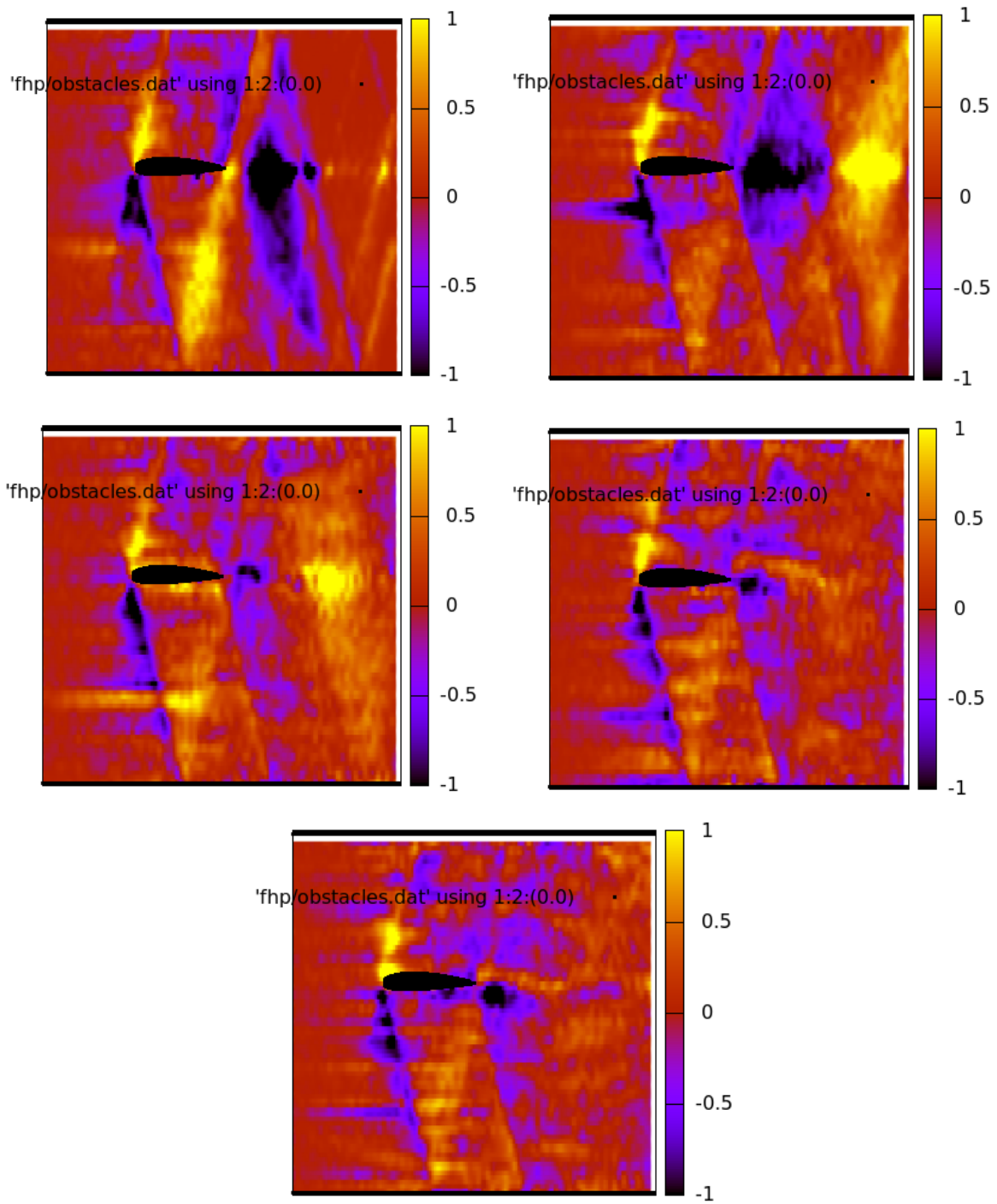


Figure A.25: Flow past the airfoil NACA2418 for $\alpha = 0$ (vector density plot).

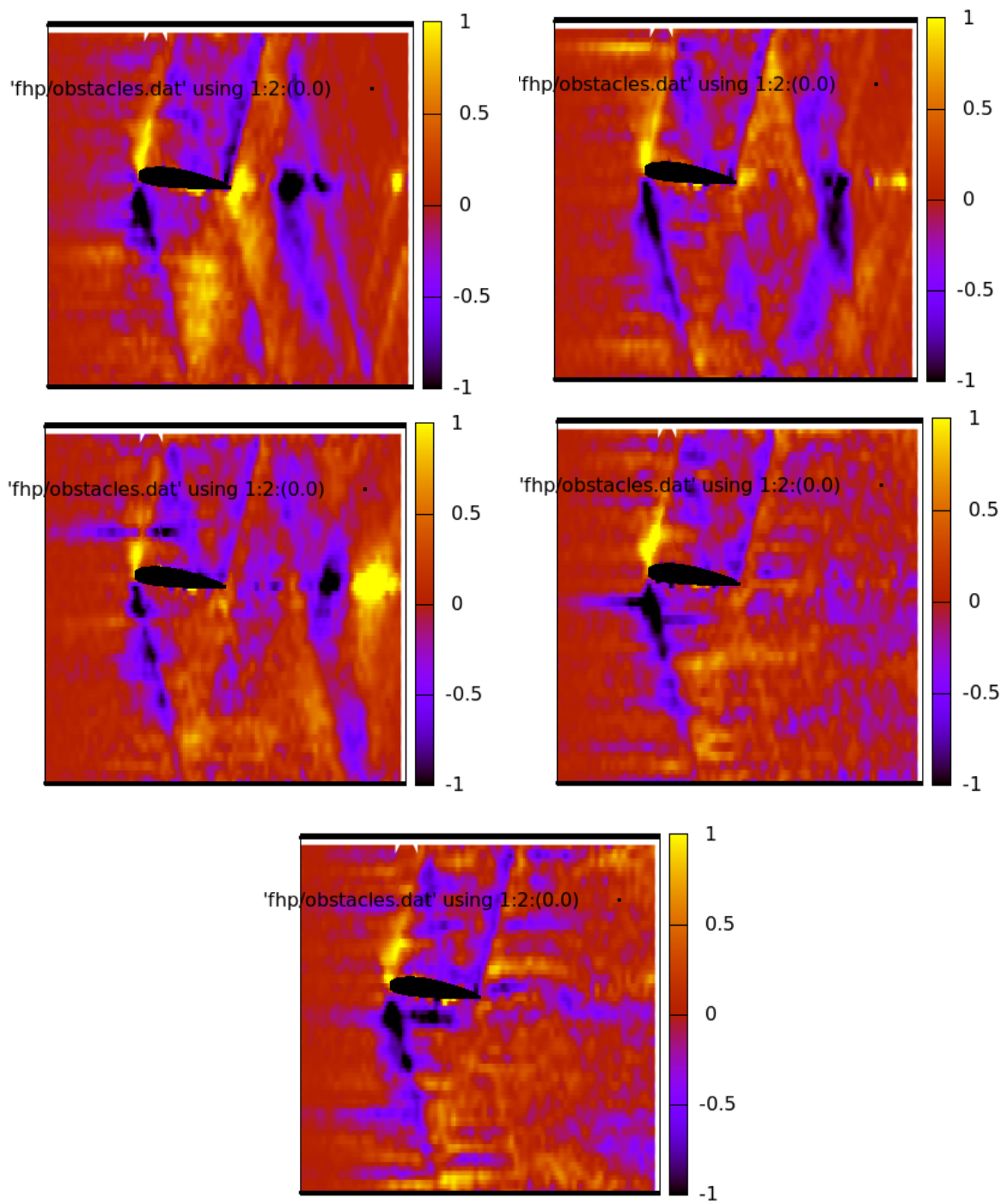


Figure A.26: Flow past the airfoil NACA2418 for $\alpha = 7.5$ (vector density plot).

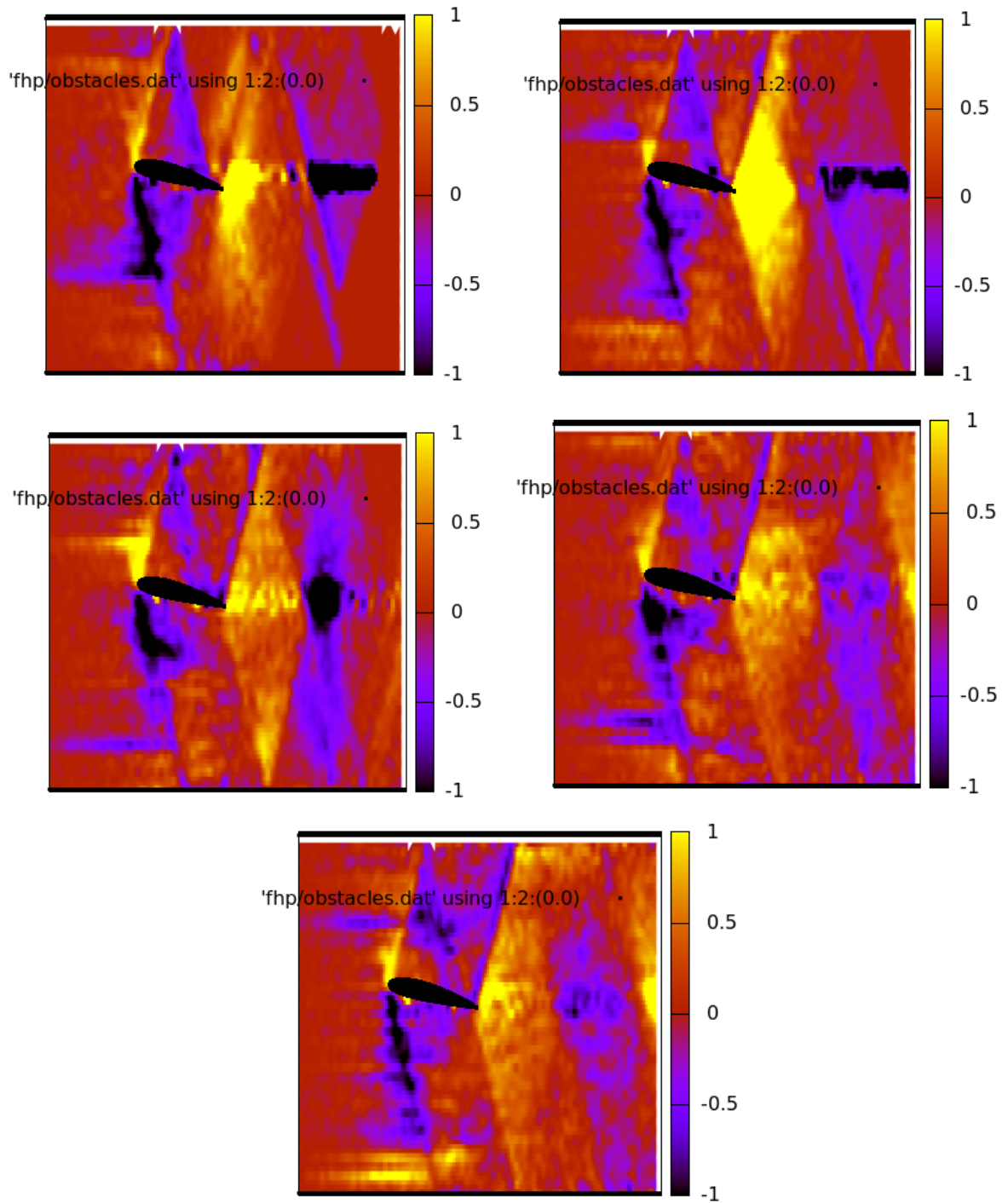


Figure A.27: Flow past the airfoil NACA2418 for $\alpha = 15$ (vector density plot).

NASA64-012A

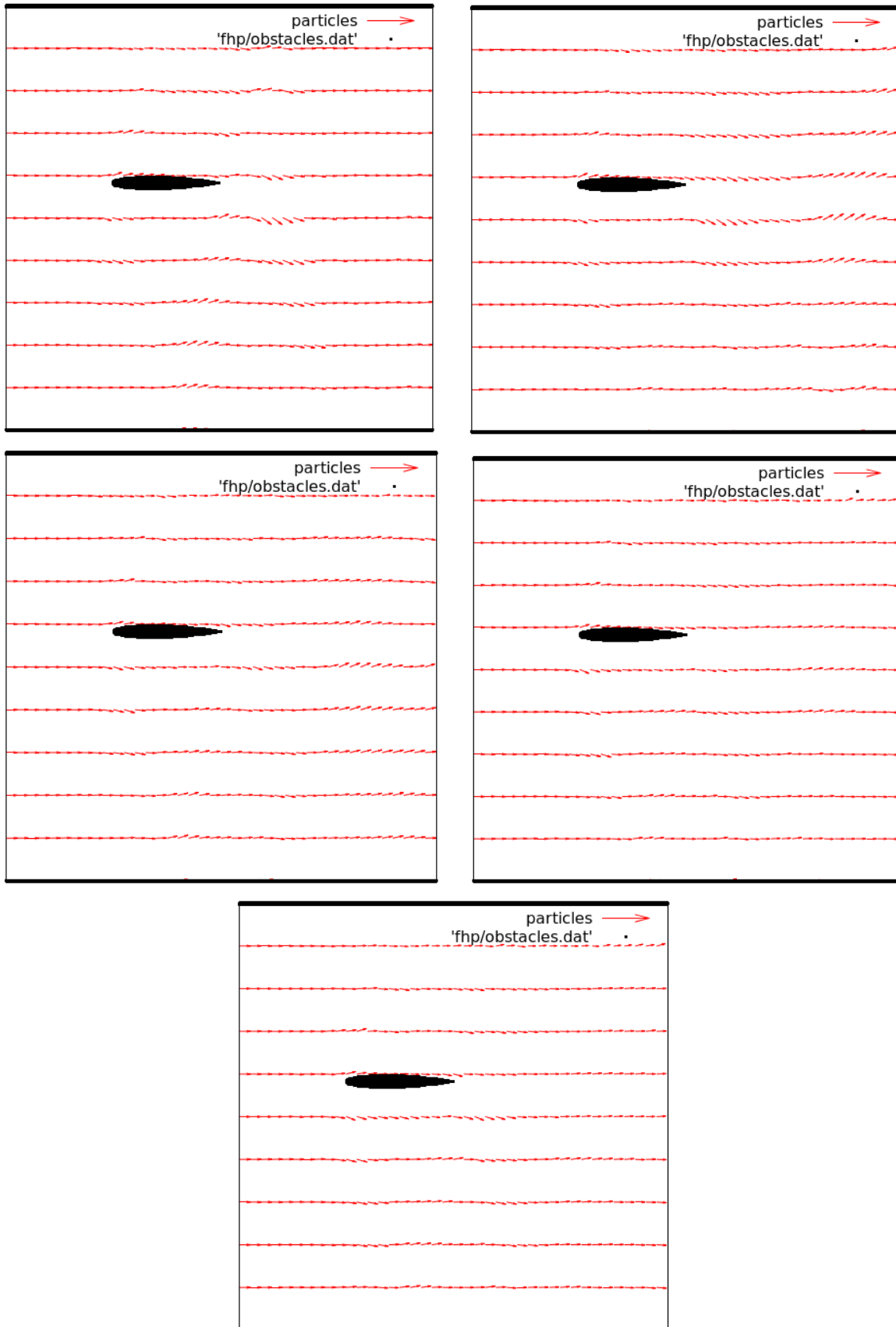


Figure A.28: Flow past the airfoil NACA64-012A for $\alpha = 0$ (vector plot).

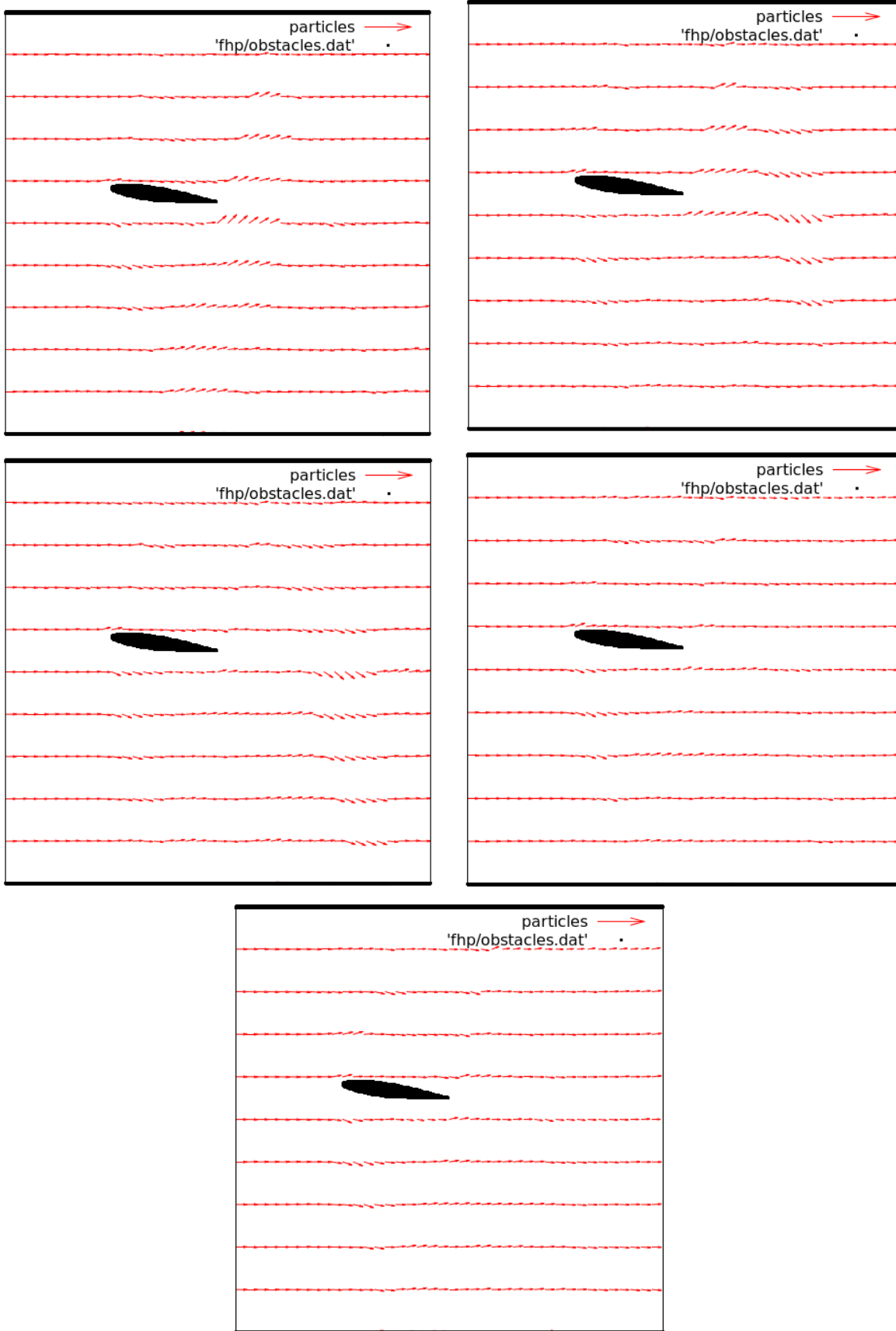


Figure A.29: Flow past the airfoil NACA64-012A for $\alpha = 7.5$ (vector plot).

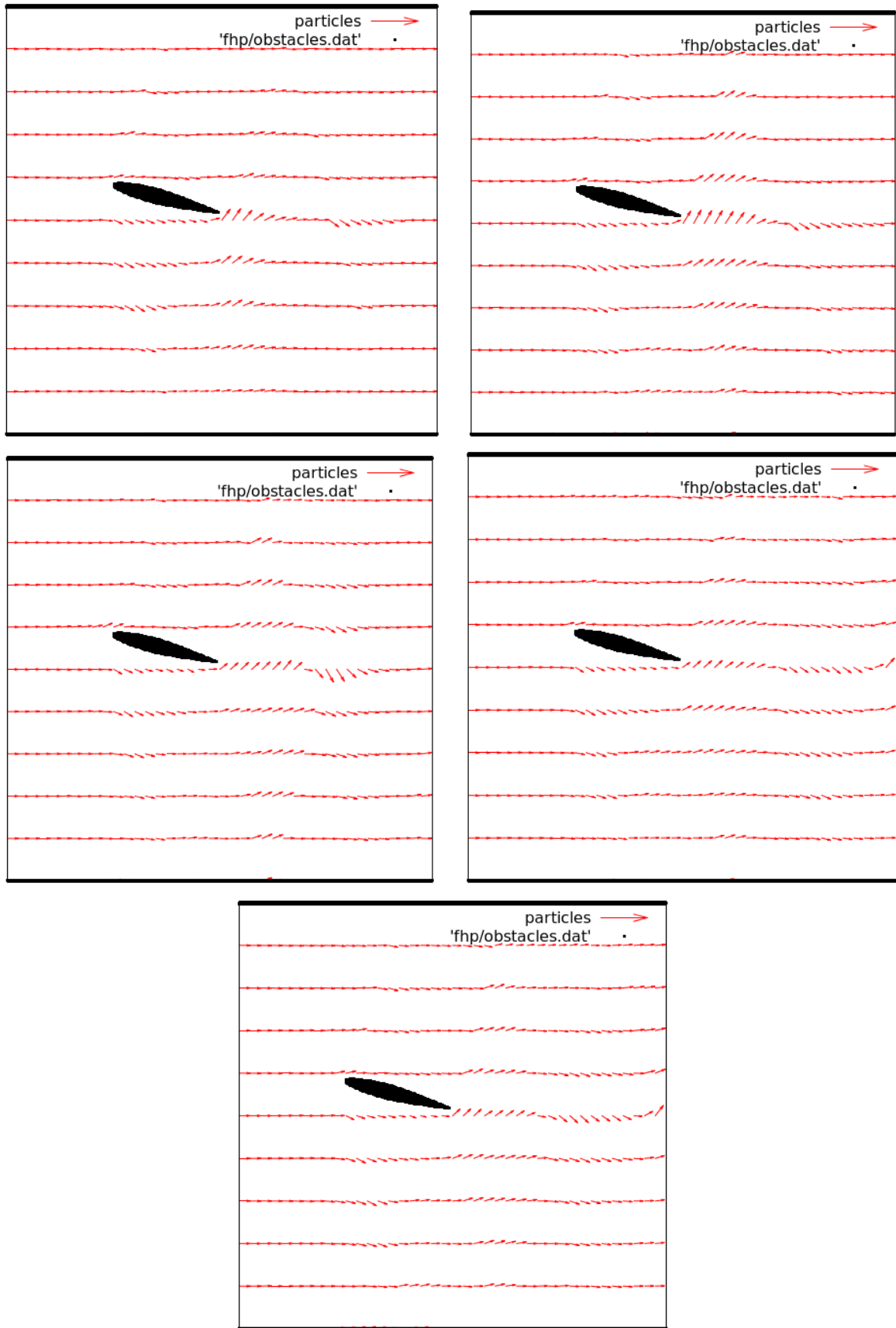


Figure A.30: Flow past the airfoil NACA64-012A for $\alpha = 15$ (vector plot).

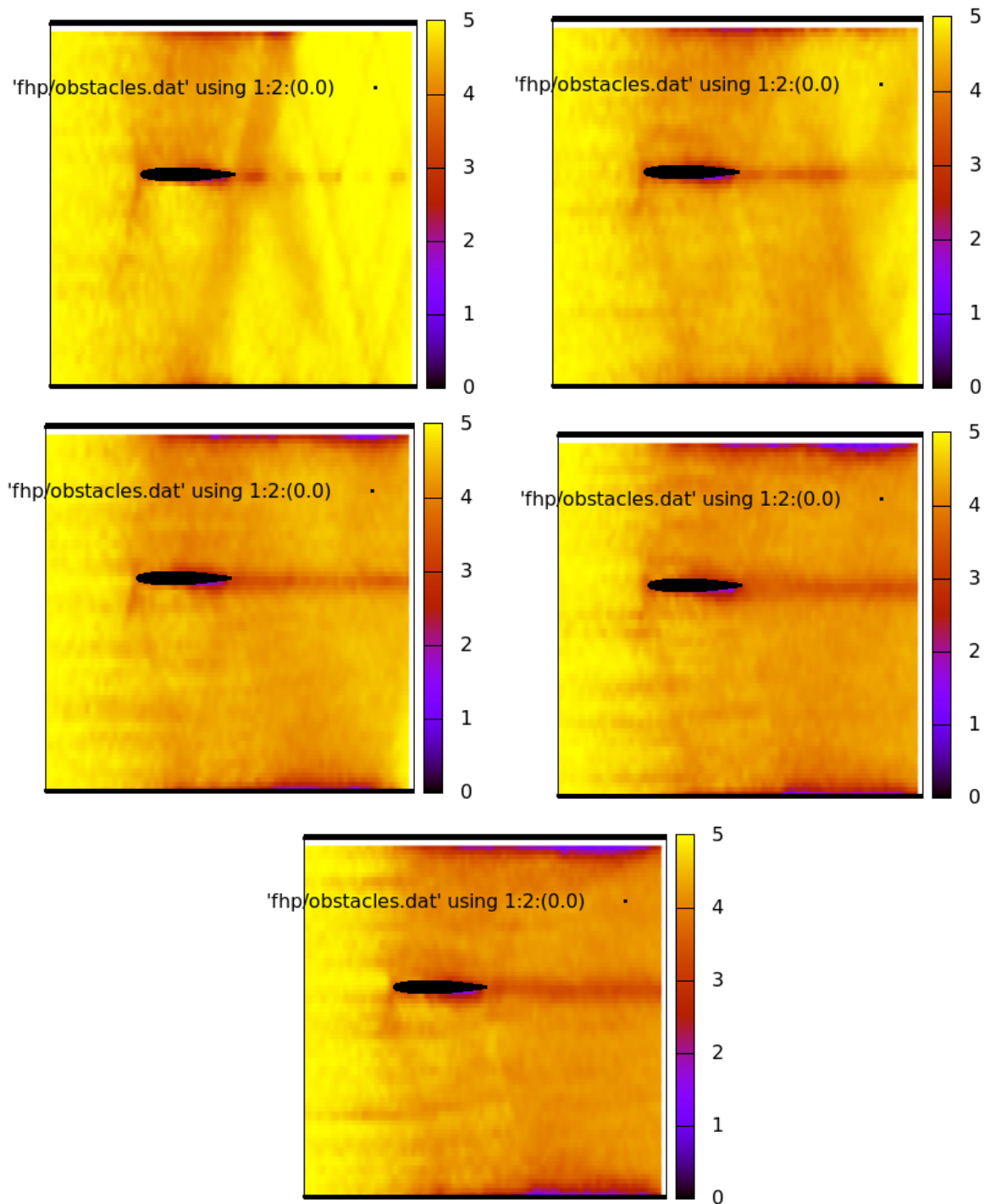


Figure A.31: Flow past the airfoil NACA64-012A for $\alpha = 0$ (scalar density plot).

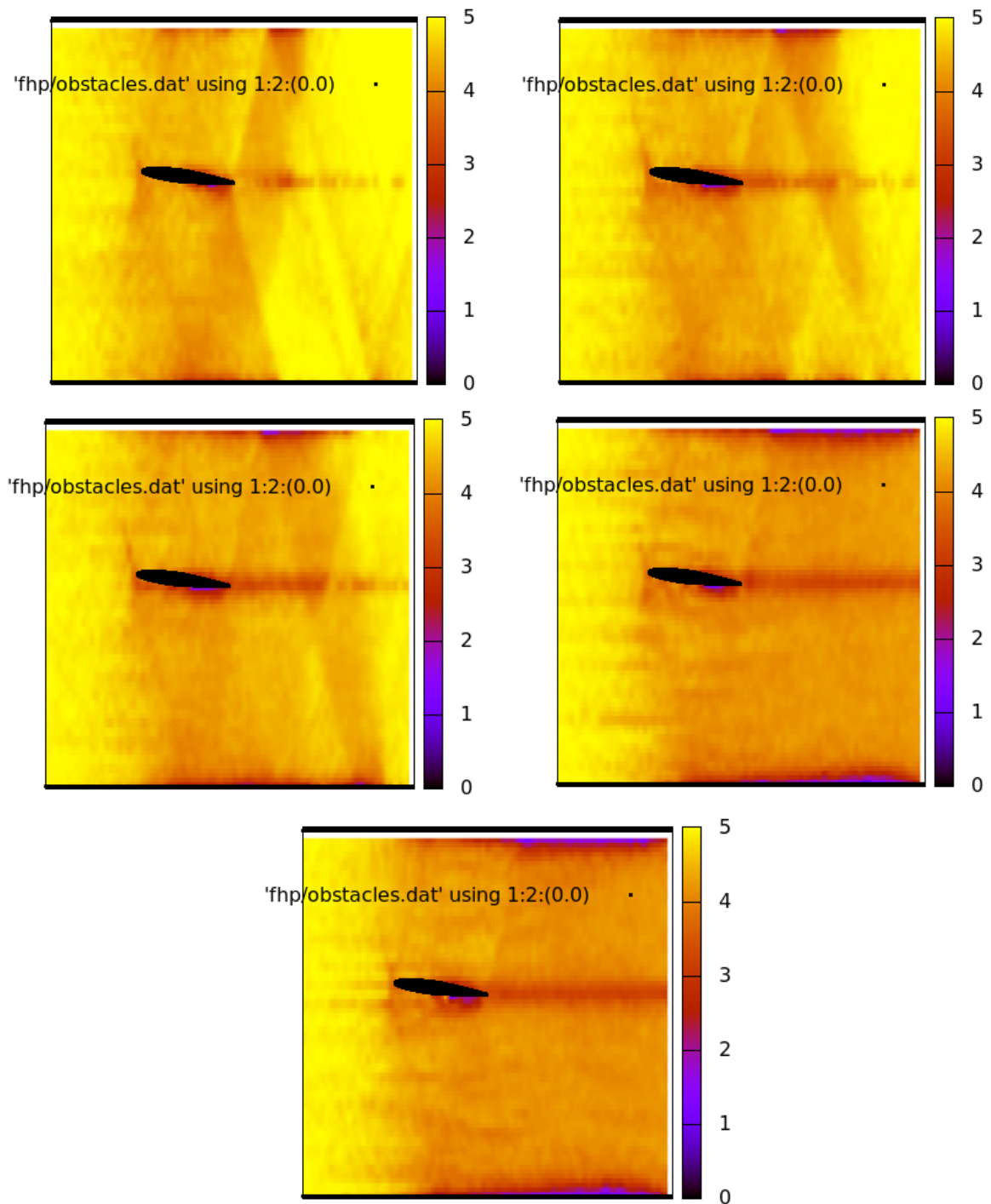


Figure A.32: Flow past the airfoil NACA64-012A for $\alpha = 7.5$ (scalar density plot).

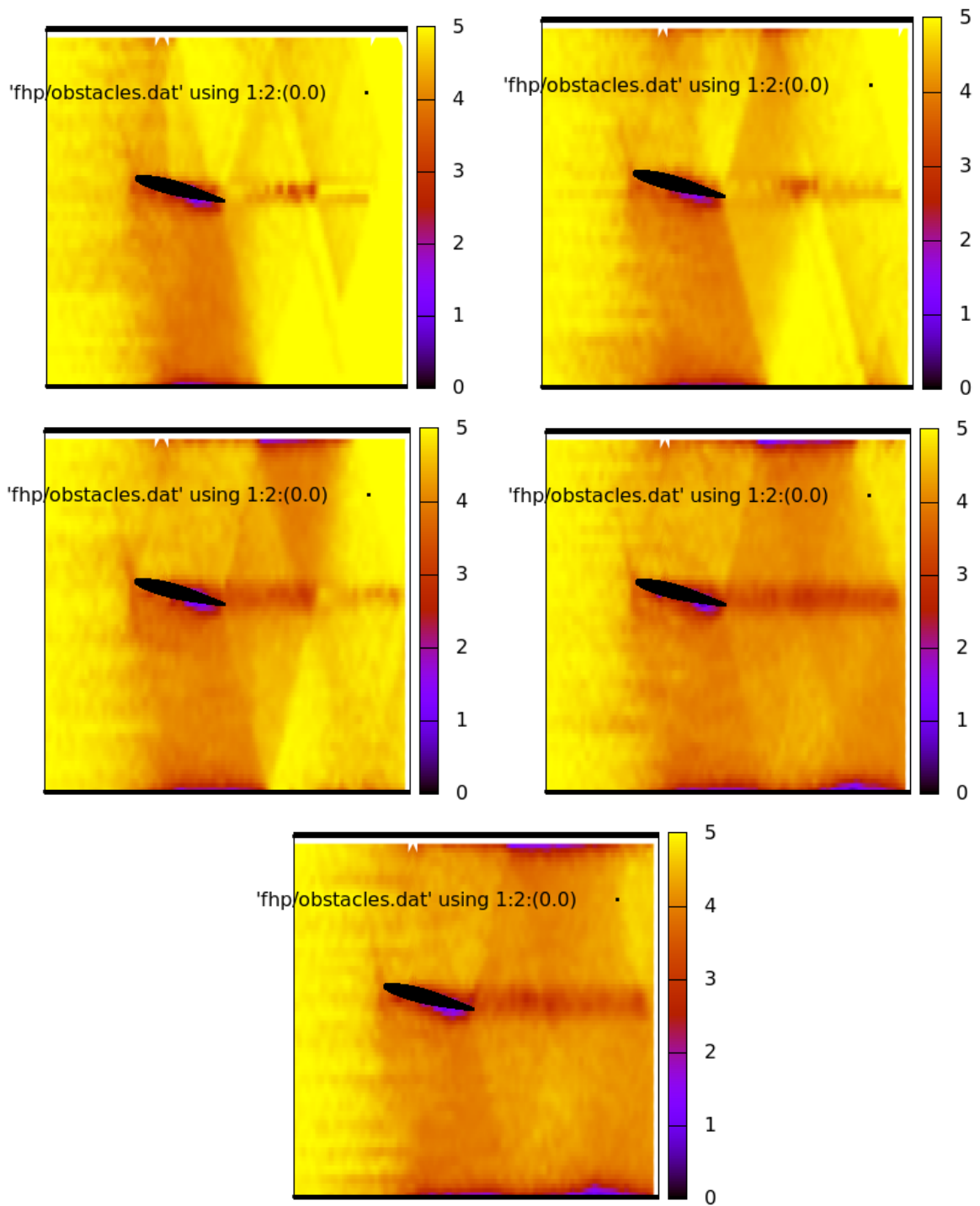


Figure A.33: Flow past the airfoil NACA64-012A for $\alpha = 15$ (scalar density plot).

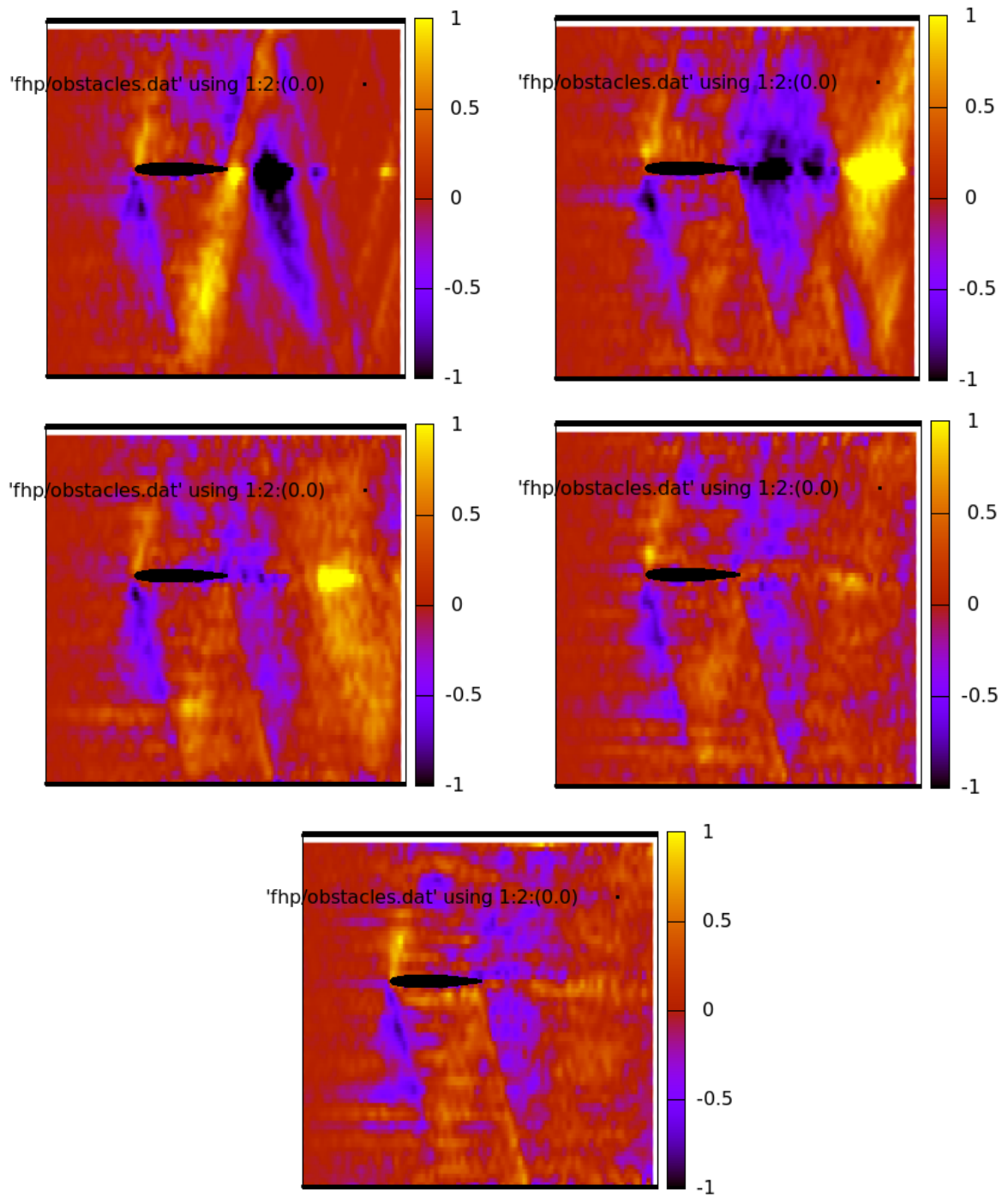


Figure A.34: Flow past the airfoil NACA64-012A for $\alpha = 0$ (vector density plot).

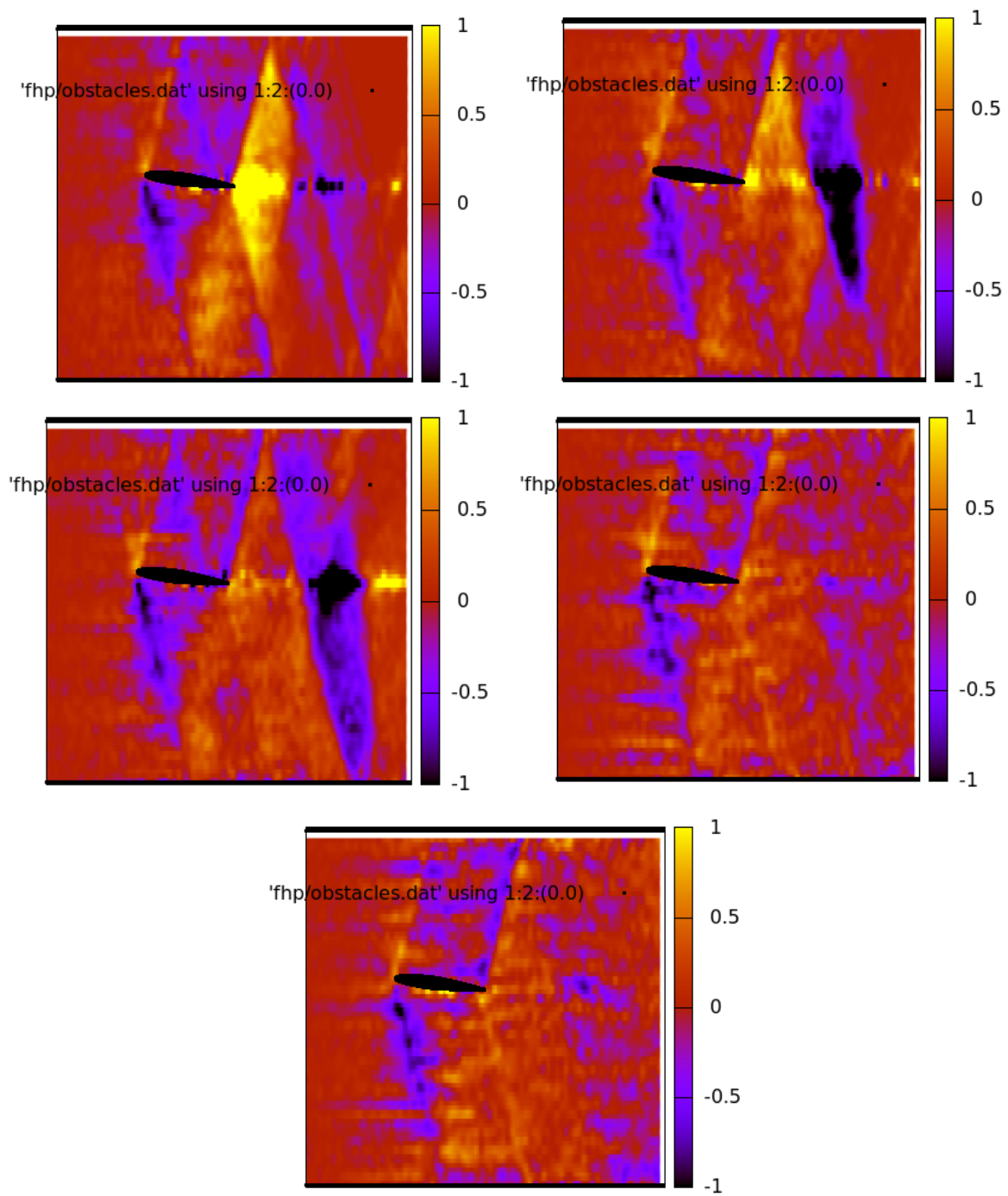


Figure A.35: Flow past the airfoil NACA64-012A for $\alpha = 7.5$ (vector density plot).

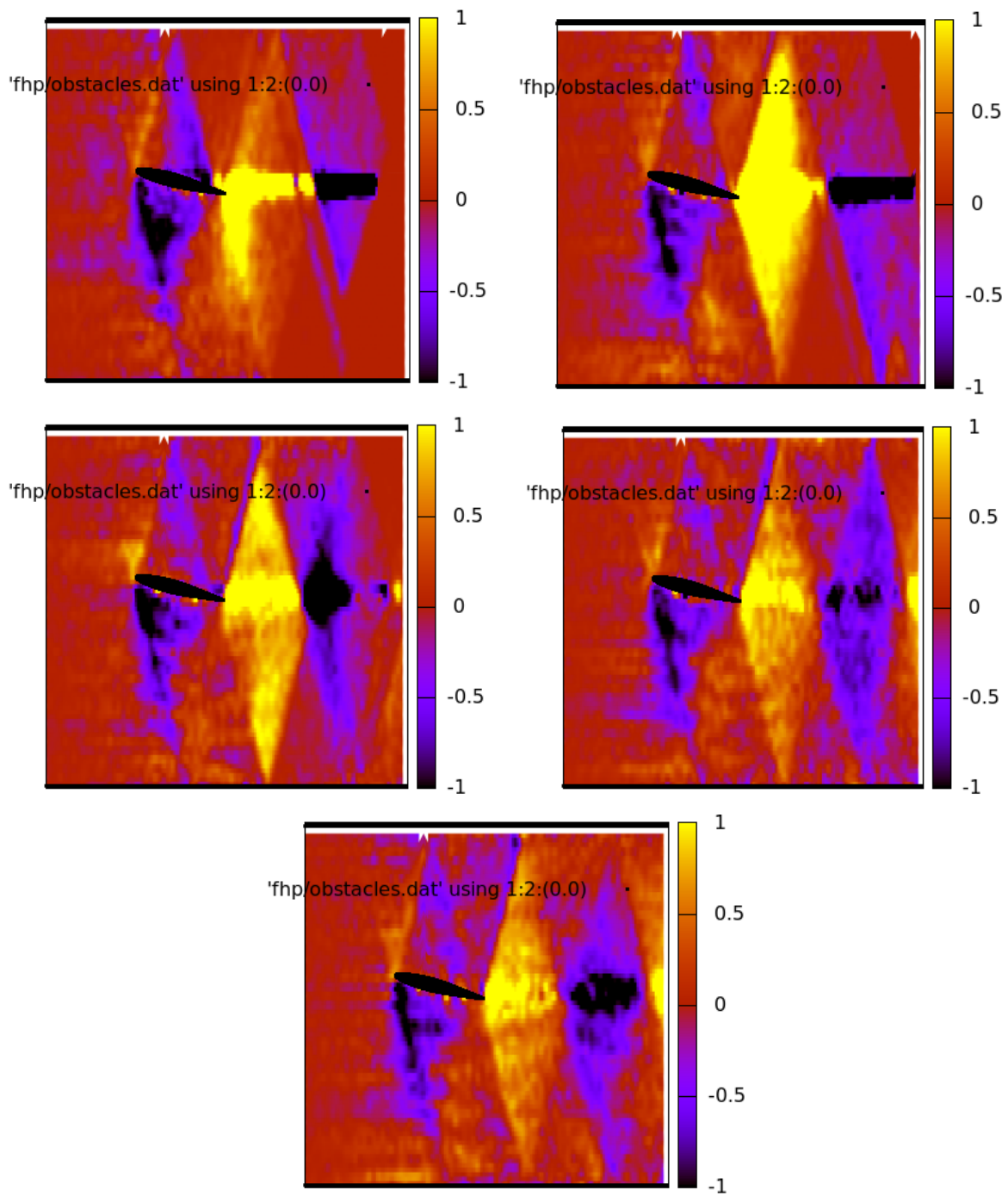


Figure A.36: Flow past the airfoil NACA64-012A for $\alpha = 15$ (vector density plot).

NACA2412

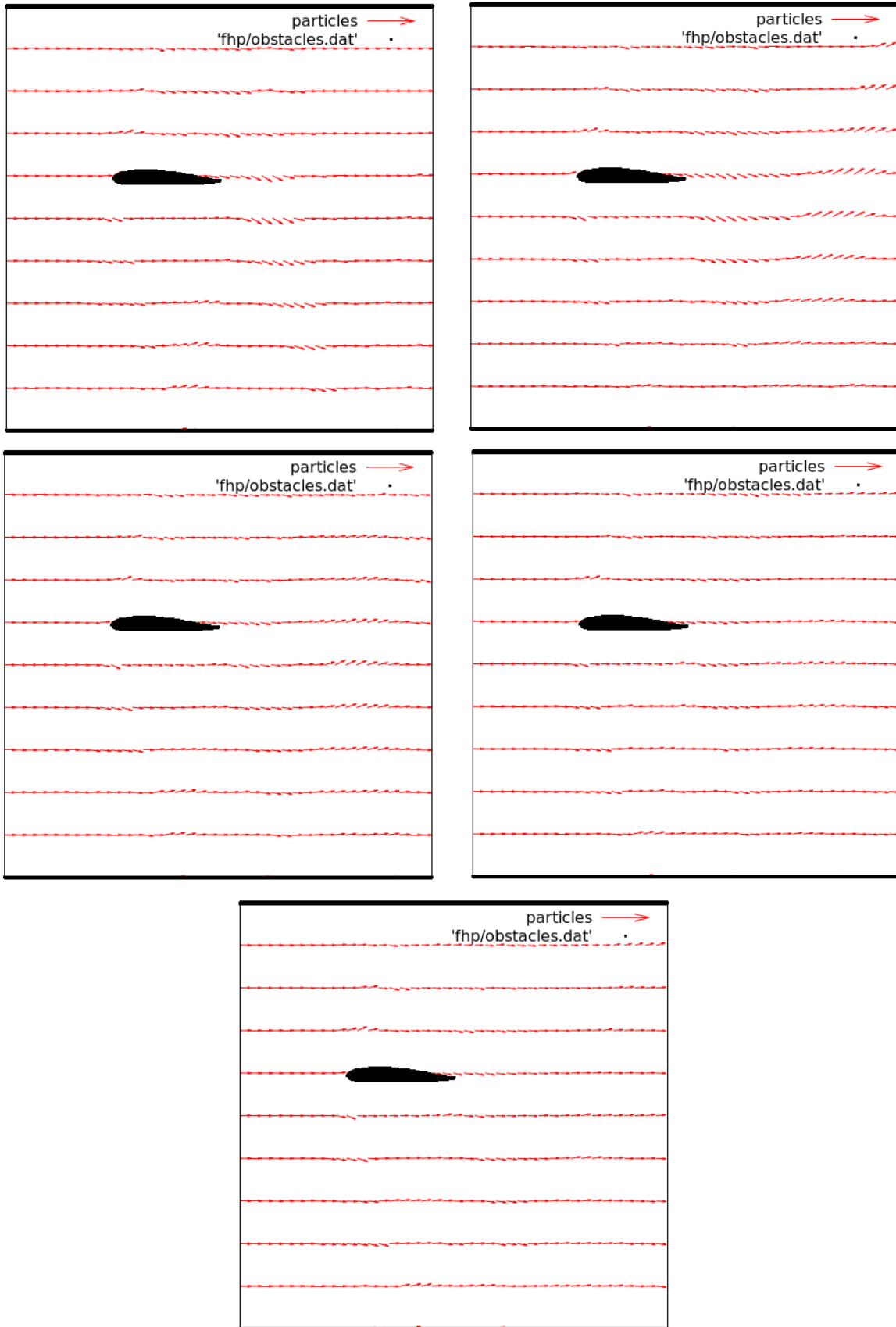


Figure A.37: Flow past the airfoil NACA2412 for $\alpha = 0$ (vector plot).

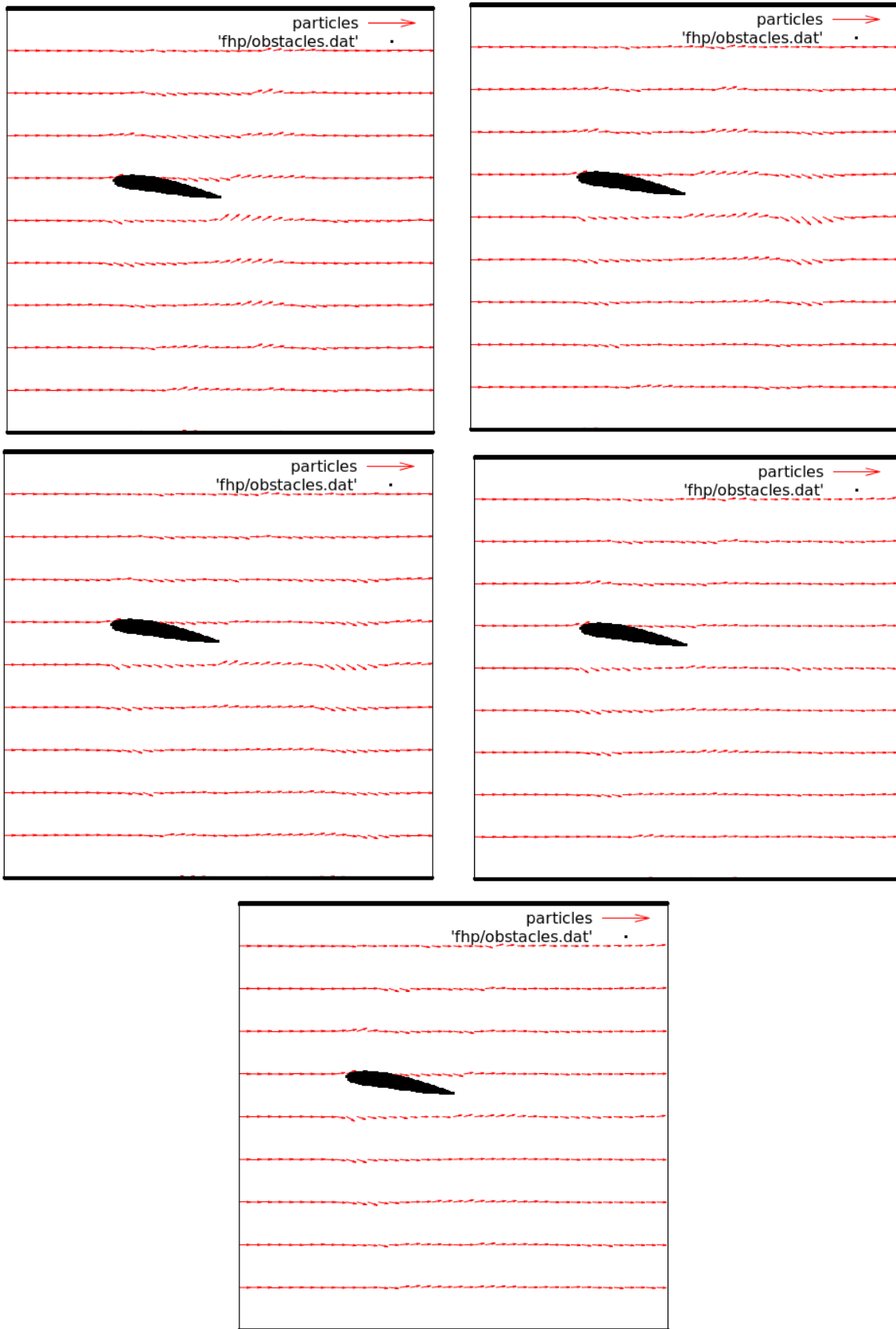


Figure A.38: Flow past the airfoil NACA2412 for $\alpha = 7.5$ (vector plot).

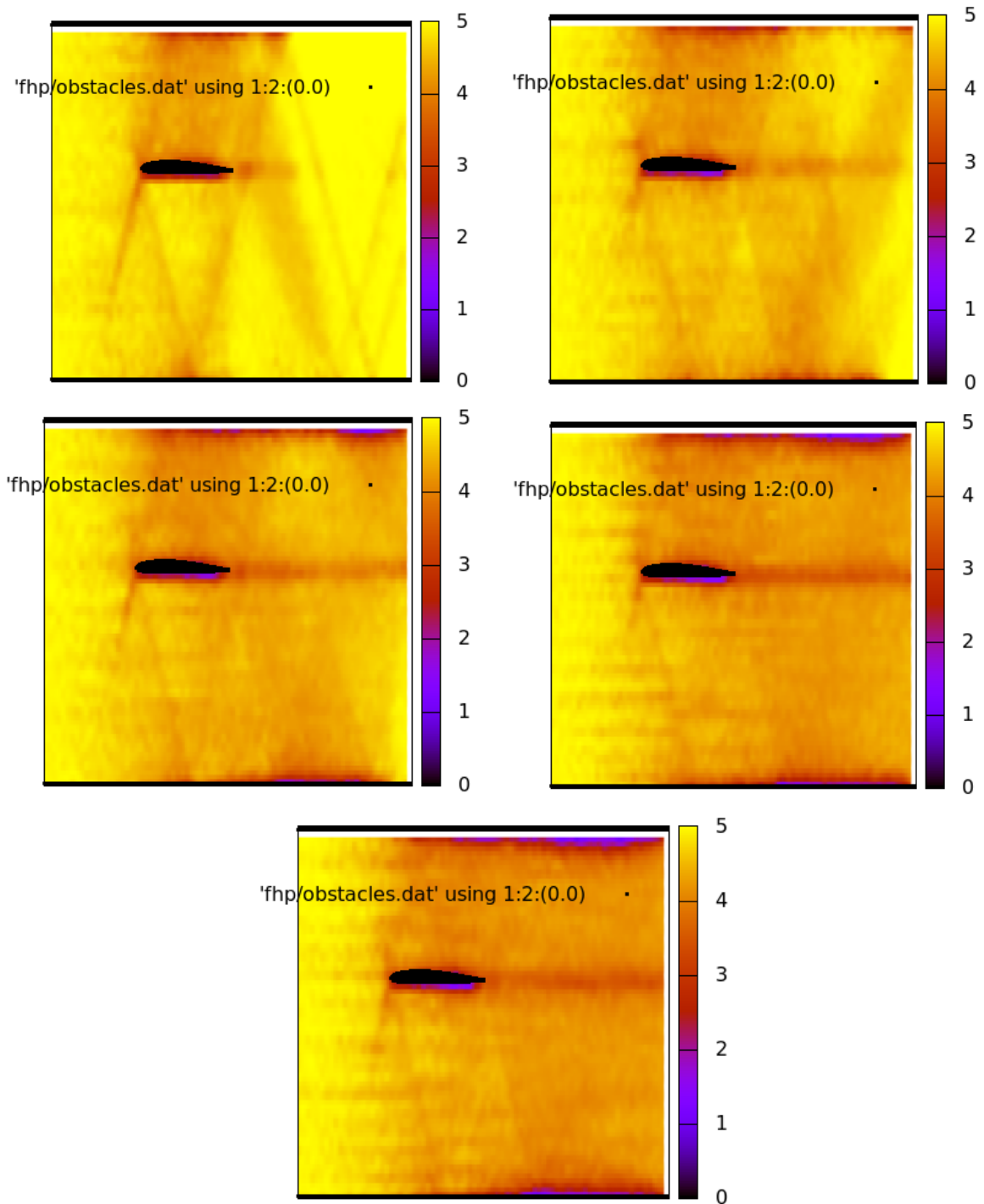


Figure A.40: Flow past the airfoil NACA2412 for $\alpha = 0$ (scalar density plot).

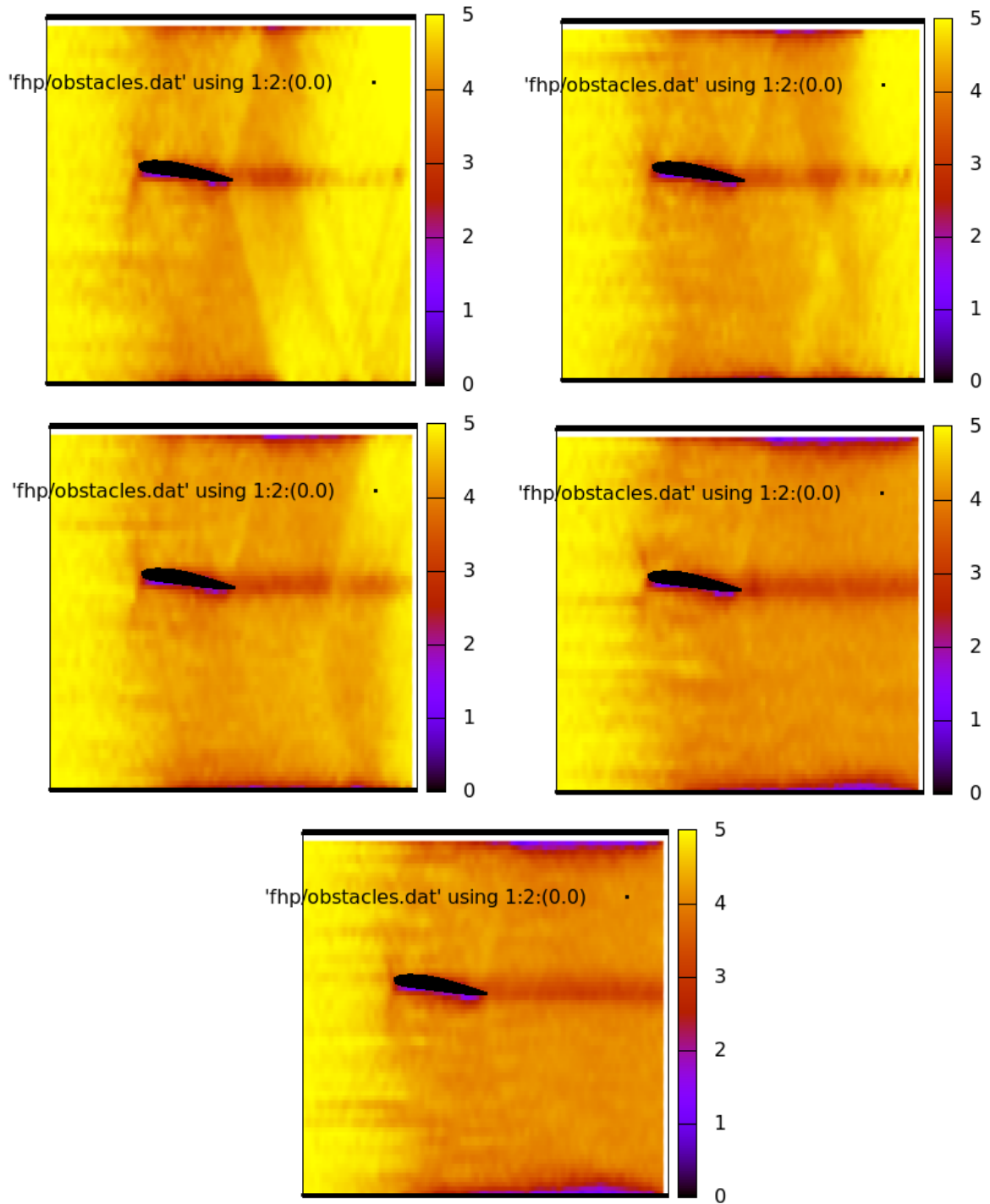


Figure A.41: Flow past the airfoil NACA2412 for $\alpha = 7.5$ (scalar density plot).

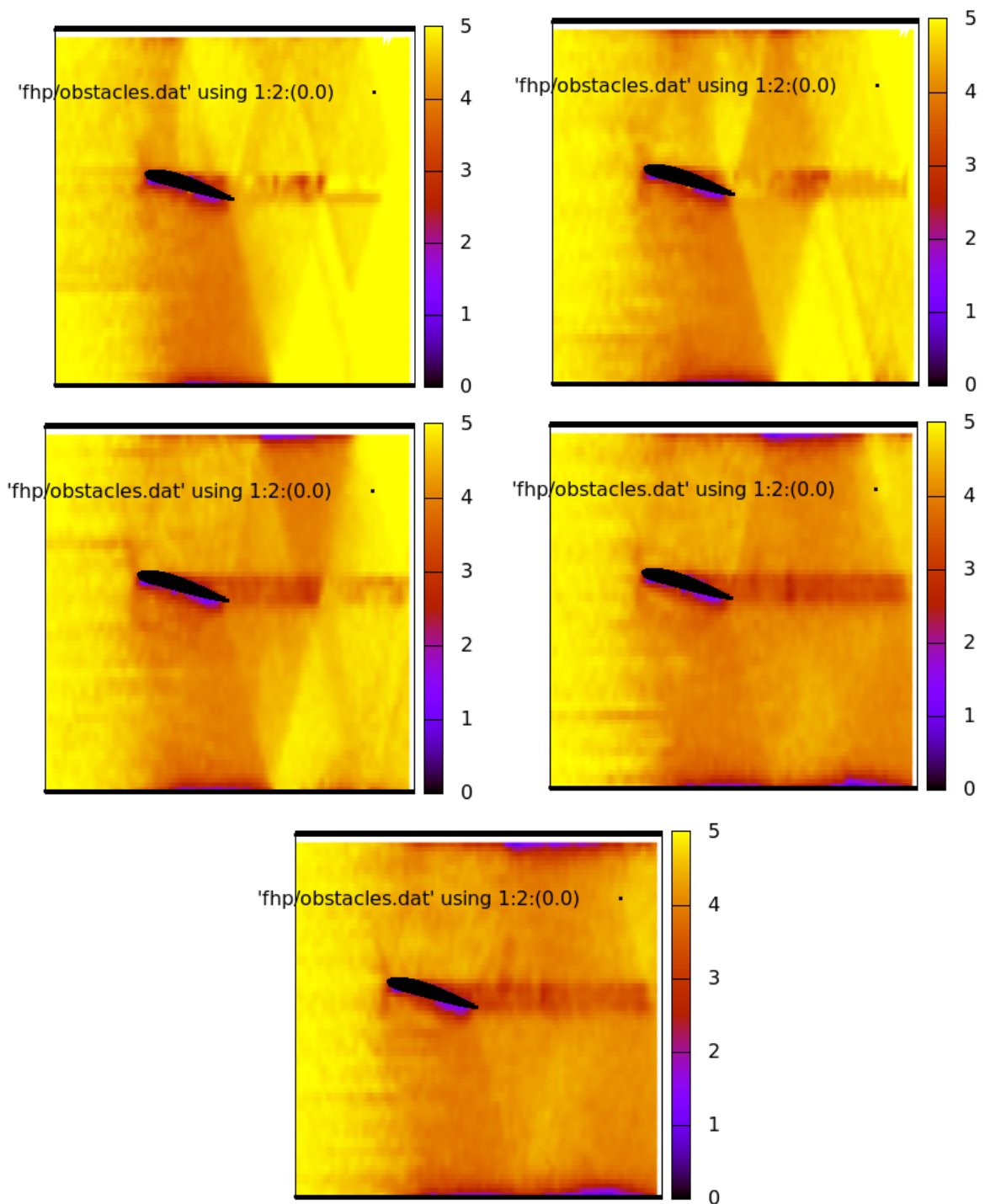


Figure A.42: Flow past the airfoil NACA2412 for $\alpha = 15$ (scalar density plot).

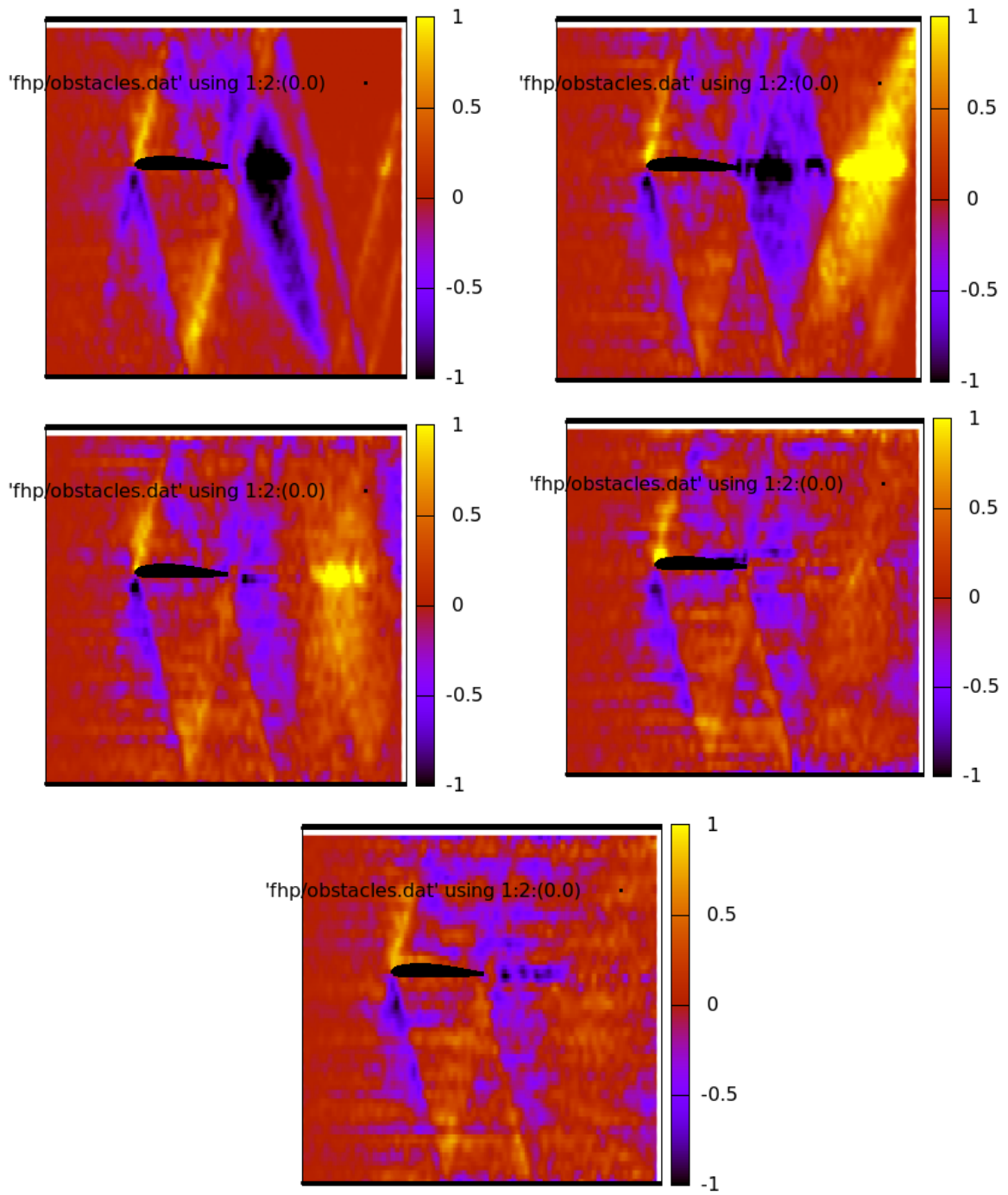


Figure A.43: Flow past the airfoil NACA2412 for $\alpha = 0$ (vector density plot).

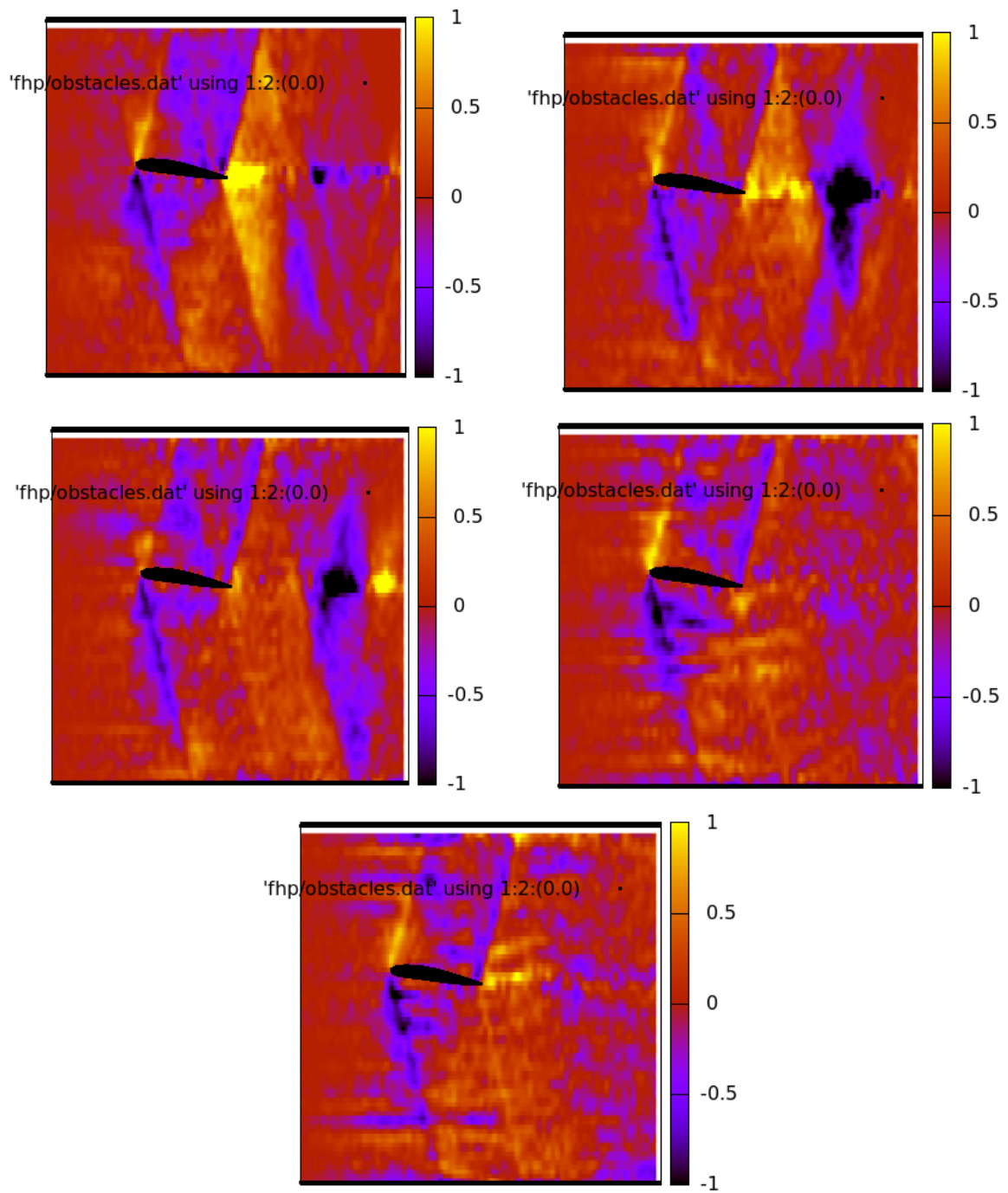


Figure A.44: Flow past the airfoil NACA2412 for $\alpha = 7.5$ (vector density plot).

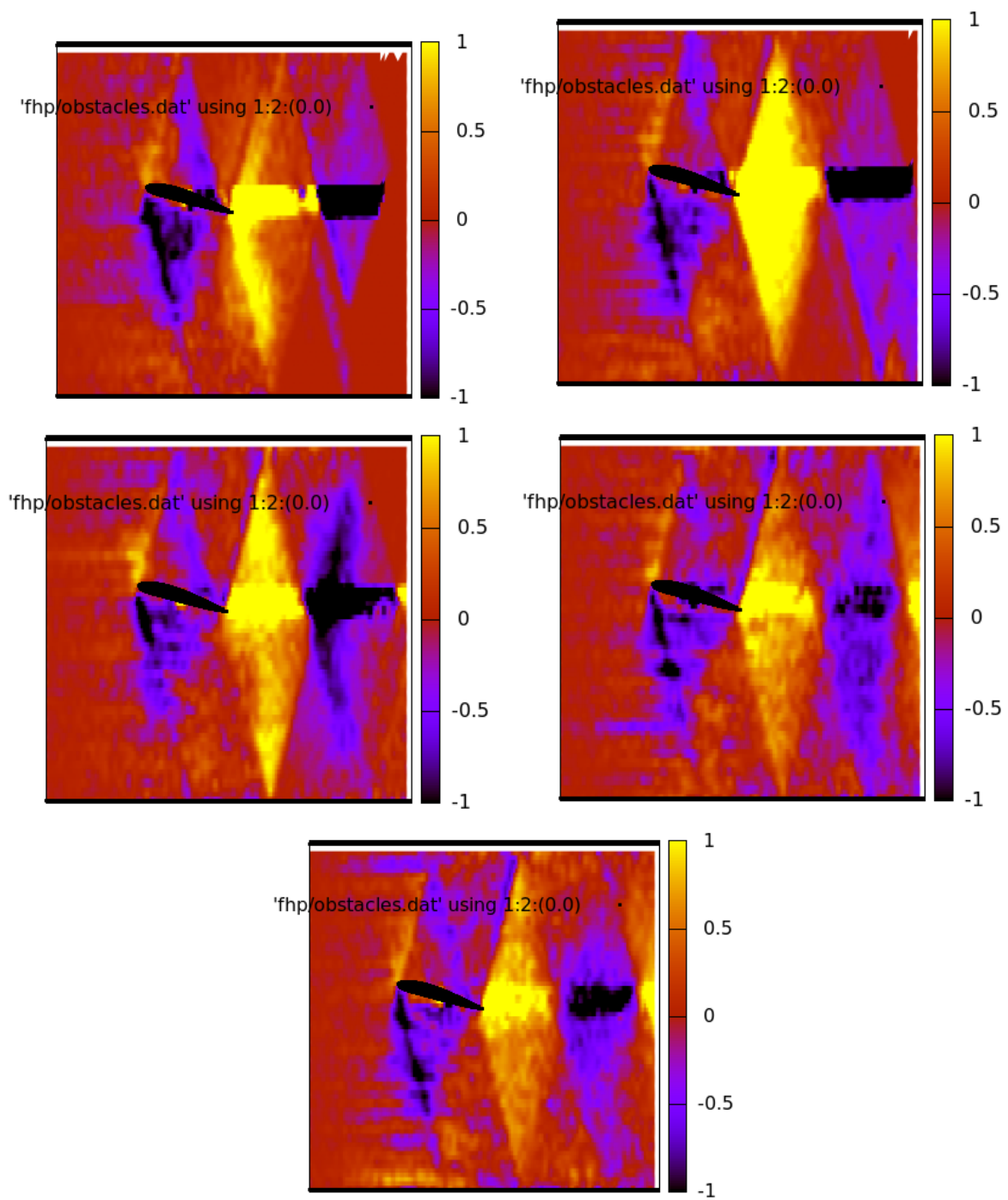


Figure A.45: Flow past the airfoil NACA2412 for $\alpha = 15$ (vector density plot).

Appendix B

Symmetries and rotation group

This chapter is written by Martin Scholtz and Iliyas Boztayev

B.1 Symmetries in physics

In physics, the notion of *symmetry* plays an important rôle. It turns out that whenever a physical system possesses some continuous symmetries, there exist associated *conserved quantities*; this is the content of the celebrated *Nöther theorems*. In particular, invariance of physical laws under translations in time implies the conservation of energy, invariance under translations in space gives us the momentum conservation and rotational invariance yields the conservation of angular momentum. The importance of symmetries becomes even more emphasized in relativistic field theories and in the modern gauge theories of elementary particles. For example, the gauge invariance of electromagnetic field implies the charge conservation. In this thesis we are, however, interested merely in classical Newtonian physics and we do not enter the discussion on relativistic theories.

In classical physics, we assume that the space is ordinary, flat Euclidean space which is homogeneous and isotropic. *Homogeneity* means that all points of the space are equivalent and the properties of the space do not vary from point to point. In other words, there is no preferred point in the Euclidean space. This must be respected by any reasonable physical law. Mathematically, any equation describing real physical process must be invariant under translation in space. According to the Nöther theorem, then, the total momentum of isolated physical system must be constant in time. Notice that the Nöther theorem is highly non-trivial result and in this thesis we make no attempt to explain its origin, as it is based on advanced variational calculus.

Another important property of Euclidean space, which is regarded as the “stage” for classical physics, is the *isotropy*. Isotropy means that all *directions* in the space are equivalent and there is no preferred direction. Again, this property must be respected by all physical equations and, hence, the equations of physics must be invariant under rotations. As a consequence, the angular momentum of isolated systems must be conserved.

Finally, time of classical physics is separated from the space, which is a contrast to special relativity theory, where time and space form a single object called space-time. Time flows uniformly from past infinity to future infinity; this can be expressed as the homogeneity of time, i.e. no time instant is preferred against any other instant. Thus,

physical laws must be invariant under translation in time, or, in other words, the physical laws do not change in time. This requirement then yields the energy conservation.

Before we start the discussion of mathematical details, let us make few comments on the relation of the symmetries to the cellular automata and the fluid dynamics. The Navier-Stokes equation, which is the central equation of the fluid dynamics, is invariant under translations and rotations as it should be. There are profound difficulties with solving this equation for any realistic situation and a huge amount of mathematical and physical literature is devoted to this problem. In addition, even the numerical solution of the Navier-Stokes equation is a difficult problem (which, however, is more-less solved in a number of situations).

The idea behind simulating the flow using the cellular automata is based on the notion of the symmetry as described above. Cellular automaton simulates the set of particles of the fluid. The point is that the interactions between these particles are chosen much simpler than real interactions between fluid particles. Thus, the *microdynamics* of cellular automata differs significantly from real microdynamics behind the Navier-Stokes equation. However, if the model is chosen wisely so that basic symmetries of the physics are preserved (at least to some accuracy), resulting *macrodynamics* is the same as for the Navier-Stokes equation. That means that if the interactions of the cellular automaton preserve the homogeneity and, more importantly, the isotropy, macrodynamics of the cellular automaton will be indistinguishable from the Navier-Stokes equation in a statistical sense.

The purpose of the theoretical part of this thesis is to make this statement more precise. Since the rotations play the prominent rôle in the following considerations, we discuss the notion of rotation in more detail.

B.2 Definition of the group

Mathematical structure known as *group* is an abstraction of what we intuitively understand as the “set of transformations”. More precisely, group (G, \cdot) is a set of elements G together with an operation of multiplication \bullet satisfying the following axioms:

(I) closure of the group under the multiplication,

$$\bullet : G \times G \mapsto G;$$

(II) existence of the identity element,

$$\exists e \in G : \forall g \in G : e \bullet g = g;$$

(III) existence of the inverse element,

$$\forall g \in G : \exists h \in G : h \bullet g = e;$$

element h satisfying this property is usually denoted $h = g^{-1}$;

(IV) multiplication is associative,

$$\forall f, g, h \in G : f \bullet (g \bullet h) = (f \bullet g) \bullet h.$$

In the definition we employed the symbol \bullet for the multiplication in order to emphasize that it is a more abstract notion than usual multiplication of numbers. For us, the group multiplication will be usually the matrix multiplication, see below. But sometimes it can be even addition, for example the integers form a group $(\mathbb{Z}, +)$, where the rôle of the multiplication of the mapping is played by ordinary addition of integers, i.e.

$$m \bullet n = m + n, \quad \text{where } m, n \in \mathbb{Z}.$$

We can easily verify that \mathbb{Z} indeed forms a group:

1. the sum of any two integers is an integer, so that \mathbb{Z} is closed under addition;
2. number $e = 0$ plays the rôle of the identity, for we have

$$m \bullet e = m + 0 = m \quad \text{for any } m \in \mathbb{Z};$$

3. to any $m \in \mathbb{Z}$, its inverse is $m^{-1} = -m$, for then

$$m \bullet m^{-1} = m + (-m) = m - m = 0.$$

Hence, $(\mathbb{Z}, +)$ is a simple example of the group.

In ordinary number algebra, we usually omit the symbol of multiplication and write ab instead of $a \cdot b$. The same convention is customary in the case of group theory. Once we specify what we mean by multiplication, we omit the symbol \bullet and write

$$g \bullet h \equiv gh.$$

We must not forget, however, that general group multiplication is not *commutative*, which means that, in general,

$$gh \neq hg.$$

The matrix multiplication is an example of non-commutative product. If the multiplication happens to be commutative, the group is said to be *Abelian* or *commutative*.

B.3 Linear groups

First of all, recall that multiplication of a vector by a matrix can be understood as the linear transformation of the vector. In general, neither the direction nor the length of the vector is preserved by a linear transformation. Let us have general 2×2 matrix A ,

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{B.1}$$

and arbitrary vector \mathbf{v} with the components

$$\mathbf{v} = \begin{pmatrix} x \\ y \end{pmatrix} \tag{B.2}$$

By matrix multiplication we can form another vector \mathbf{v}' ,

$$\mathbf{v}' = A\mathbf{v} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}. \quad (\text{B.3})$$

Written in components, we have

$$\begin{aligned} v'_1 &= ax + by, \\ v'_2 &= cx + dy, \end{aligned} \quad (\text{B.4})$$

or, more compactly,

$$v'_i = A_{ij} v_j. \quad (\text{B.5})$$

Here we employ the Einstein summation convention: whenever an index appears exactly twice in the expression, we automatically sum through this index. In particular, the last equation is equivalent to

$$v'_i = \sum_{j=1}^2 A_{ij} v_j. \quad (\text{B.6})$$

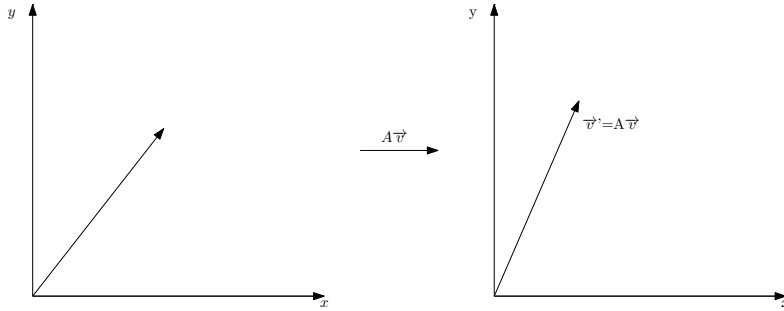


Figure B.1: Linear transformation of the vector \mathbf{v} by matrix A .

We can see that the multiplication of the vector by a matrix (B.3) or (B.6) can be regarded as a mapping which takes the original vector \mathbf{v} and produces another vector \mathbf{v}' , cf. figure B.1. In other words, matrix A transforms vectors of the plane into another vectors. This transformation is linear which translates into conditions

$$\begin{aligned} A(\mathbf{v} + \mathbf{w}) &= A\mathbf{v} + A\mathbf{w}, \\ A(\lambda \mathbf{v}) &= \lambda A(\mathbf{v}), \quad \text{where } \lambda \in \mathbb{R}. \end{aligned} \quad (\text{B.7})$$

We reiterate that such a general linear transformation changes both directions and lengths of vectors.

We might expect that since the square matrix represents a linear transformation, the set of all matrices forms a group. This is almost the case, but not entirely. To see the point, let us try to check the group axioms. First, the matrices are obviously closed under multiplication, because a product of two matrices. For 2×2 matrices

$$A = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}, \quad B = \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix}, \quad (\text{B.8})$$

their product is

$$AB = \begin{pmatrix} a_1 b_1 + a_2 b_3 & a_1 b_2 + a_2 b_4 \\ a_3 b_1 + a_4 b_3 & a_3 b_2 + a_4 b_4 \end{pmatrix}, \quad (\text{B.9})$$

which is another matrix. The first axiom of group is therefore satisfied. Notice that the matrix multiplication in the index form reads

$$(AB)_{ij} = \sum_{k=1}^2 A_{ik} B_{kj} \equiv A_{ik} B_{kj}, \quad (\text{B.10})$$

where we have invoked the Einstein summation convention in the last step.

Next, we define the *identity matrix* by

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \quad (\text{B.11})$$

It is an easy exercise to show that

$$AI = IA = A \quad (\text{B.12})$$

for any matrix A . Hence, the identity matrix plays the rôle of the identity element and the second group axiom is satisfied. The elements of the identity matrix are often denoted by the *Kronecker symbol* δ_{ij} which is defined by

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases} \quad (\text{B.13})$$

The properties of the identity matrix can be written in the index form as follows:

$$A_{ij} \delta_{jk} = \delta_{ij} A_{jk} = A_{ik}. \quad (\text{B.14})$$

Thus, the first two axioms of the group are satisfied by the matrix multiplication. The problematic axiom is the third one about the existence of the inverse element (inverse matrix). We know from the linear algebra that the determinant of matrix

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad (\text{B.15})$$

is defined as

$$\det A = ad - bc. \quad (\text{B.16})$$

Then we can define the *inverse matrix* A^{-1} by

$$A^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}. \quad (\text{B.17})$$

One can easily check that the product of A and A^{-1} is the identity matrix,

$$AA^{-1} = I. \quad (\text{B.18})$$

Apparently, also the third axiom of the group is satisfied, for we have found matrix A^{-1} for any matrix A such that their product is the identity element (identity matrix). However, prescription (B.17) works only if the denominator does not vanish, i.e. if

$$\det A \neq 0. \tag{B.19}$$

If the determinant of A is equal to zero, the inverse matrix does not exist. Consequently, the third group axiom is satisfied only for those matrices which have non-vanishing determinant. The set of *all* matrices does not constitute a group, only matrices with $\det A \neq 0$ do.

Recall that we interpret matrices as linear transformations of vectors. What is so special about matrices with vanishing determinant? They map linearly independent set of vectors into linearly dependent set. As a simple example, consider the matrix

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \tag{B.20}$$

with the determinant $\det A = 1 - 1 = 0$. Now, vectors

$$\mathbf{e}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \mathbf{e}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \tag{B.21}$$

are obviously linearly independent (one is not the multiple of the other one) and, in fact, arbitrary vector

$$\mathbf{v} = \begin{pmatrix} x \\ y \end{pmatrix} \tag{B.22}$$

can be written as a linear combination of these vectors,

$$\mathbf{v} = x \mathbf{e}_1 + y \mathbf{e}_2 = x \begin{pmatrix} 1 \\ 0 \end{pmatrix} + y \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}. \tag{B.23}$$

We say that vectors \mathbf{e}_1 and \mathbf{e}_2 form a *basis* for all vectors. What happens to these vectors under transformation with matrix A given by (B.20)? We have

$$\mathbf{e}'_1 = A \mathbf{e}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \mathbf{e}'_2 = A \mathbf{e}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}. \tag{B.24}$$

We can see that both vectors transform into the same vector $(1, 1)$. So, the vectors which were orthogonal originally are parallel after the transformation. Obviously, this transformation cannot be undone by any matrix A^{-1} because having \mathbf{e}'_1 , we cannot say what the original vector was: was it \mathbf{e}_1 or \mathbf{e}_2 ? Clearly, transformation by matrix (B.20) is not invertible.

Matrices with vanishing determinant are called *singular*, while those with non-vanishing determinant are called *regular*. To summarize, we have seen that the set of all matrices does not constitute a group because it contains also singular matrices for which the inverse does not exist. However, the set of all regular matrices already does form a group, because all regular matrices do have an inverse.

The group of all regular matrices is called *general linear group* and denoted by $\mathbb{GL}(n)$, where the number n specifies the rank of the matrices. For example, $\mathbb{GL}(3)$ is the set of all matrices of type 3×3 which have non-vanishing determinant. For several applications it is useful to study a *subgroup* of $\mathbb{GL}(n)$ which is made of matrices with the *unit determinant*. This subgroup is called *special linear group* and it is denoted by $\mathbb{SL}(n)$. Suppose that $A, B \in \mathbb{SL}(n)$, so that

$$\det A = \det B = 1. \tag{B.25}$$

The question is whether the product AB is again a special matrix. The answer is affirmative, however, for the general rule for determinants tells us

$$\det(AB) = \det A \det B = 1, \tag{B.26}$$

so the subgroup $\mathbb{SL}(n)$ is closed under multiplication and hence forms a group by itself. Matrices with $\det A = 1$ are called *unimodular*.

B.4 Orthogonal groups

In the previous section we have introduced linear matrix groups $\mathbb{GL}(n)$ and $\mathbb{SL}(n)$ comprised of matrices with non-vanishing determinant for $\mathbb{GL}(n)$ or unit determinant for $\mathbb{SL}(n)$. Now we focus on even more special matrices which do not represent general linear transformation, but particular type of transformations – rotations. Let us find out what additional properties must matrices representing rotations satisfy.

The crucial property of rotation is that it preserves the lengths of the vectors, while it changes the direction of the vector. The situation is illustrated in figure B.2. The length of the vector

$$\mathbf{v} = \begin{pmatrix} x \\ y \end{pmatrix} \tag{B.27}$$

is defined by the Pythagorean theorem,

$$\|\mathbf{v}\|^2 = x^2 + y^2. \tag{B.28}$$

We can write this relation in the matrix form, if we introduce transposed matrix

$$\mathbf{v}^T = (x \ y). \tag{B.29}$$

The length of the vector is then given by matrix multiplication

$$\mathbf{v}^T \mathbf{v} = (x \ y) \begin{pmatrix} x \\ y \end{pmatrix} = x^2 + y^2. \tag{B.30}$$

Now, let $R \in \mathbb{GL}(2)$ be a regular matrix of dimension 2. We want to express mathematically, that linear transformation with matrix R preserves the lengths of vectors. General linear transformation is, as we know, by

$$\mathbf{v}' = R \mathbf{v}. \tag{B.31}$$

The length of the new vector is $\mathbf{v}'^T \mathbf{v}$ and we require it is the same as the original length:

$$\mathbf{v}'^T \mathbf{v} = \mathbf{v}^T R^T R \mathbf{v}, \quad (\text{B.32})$$

where we have used usual rule for the transposition of product,

$$(A B)^T = B^T A^T. \quad (\text{B.33})$$

Since the relation (B.32) must hold for arbitrarily chosen vector \mathbf{v} , matrix R must satisfy

$$R^T R = I, \quad (\text{B.34})$$

for then we have

$$\mathbf{v}'^T R^T R \mathbf{v} = \mathbf{v}^T I \mathbf{v} = \mathbf{v}^T \mathbf{v}. \quad (\text{B.35})$$

In the index notation, matrix R must satisfy

$$R_{ij} R_{kj} = \delta_{ik}. \quad (\text{B.36})$$

Matrices R satisfying (B.34) or (B.36) are called *orthogonal matrices* and relations (B.34) and (B.36) are referred to as the *relations of orthogonality*. The set of orthogonal matrices of dimension n is denoted by $\mathbb{O}(n)$ and we claim that they form a group. As usually, we have to check that the axioms of the group are satisfied.

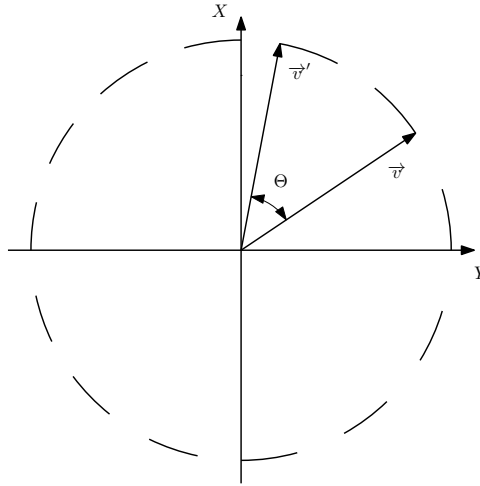


Figure B.2: Rotation by angle θ .

1. First, we have to verify that the product of orthogonal matrices is again an orthogonal matrix. Let P and Q be orthogonal matrices, i.e.

$$P^T P = I, \quad Q^T Q = I. \quad (\text{B.37})$$

Let $R = PQ$ be the product of the matrices. Then we have

$$R^T R = (PQ)^T (PQ) = Q^T \underbrace{P^T P}_I Q = Q^T Q = I, \quad (\text{B.38})$$

and hence R is orthogonal, so that $\mathbb{O}(n)$ is closed under matrix multiplication.

2. Next we have to check the existence of the identity element of $\mathbb{O}(n)$. We now that, in the group $\mathbb{GL}(n)$, the identity element is the identity matrix I and, clearly, there is no other reasonable candidate for the identity element in $\mathbb{O}(n)$ ¹. So, effectively, we have to check that the identity matrix I is orthogonal. This is trivial, for we have $I^T = I$ and therefore

$$I^T I = I I = I, \tag{B.39}$$

so that $I \in \mathbb{O}(n)$.

3. Finally, we have to check the existence of the inverse element; the question is, whether for any $R \in \mathbb{O}(n)$ there exists $R^{-1} \in \mathbb{O}(n)$. Let us have a look at the definition (B.34) of the orthogonal matrix. Since we have $\det AB = \det A \det B$ and $\det A^T = \det A$, applying the determinant to both side of relation (B.34) we find

$$\det(R^T R) = \det R^T \det R = |\det R|^2 = \det I = 1. \tag{B.40}$$

Thus,

$$|\det R|^2 = 1 \quad \text{and hence} \quad \det R = \pm 1. \tag{B.41}$$

In both cases, the determinant of any orthogonal matrix R must be non-zero (either 1 or -1), so that the inverse matrix R^{-1} exists. Multiplying (B.34) with the inverse R^{-1} we find

$$R^T = R^{-1} \tag{B.42}$$

and so the inverse matrix R^{-1} is just the transpose of R . The last question is, whether if R satisfies the orthogonality relation, so does R^T . This is very trivial, but for the sake of completeness we present the formal proof. Let us denote $Q = R^{-1} = R^T$. Is Q orthogonal? Yes:

$$Q^T Q = (R^T)^T R^T = R R^T = (R^T R)^T = I^T = I. \tag{B.43}$$

This completes the proof that $\mathbb{O}(n)$ is a group. ■

In the proof we have encountered an interesting auxiliary result that the determinant of any orthogonal matrix is either 1 or -1 ,

$$\det R = \pm 1. \tag{B.44}$$

As in the case of the special linear group $\mathbb{SL}(n)$, we call the matrices with determinant $\det R = 1$ *unimodular*. And, again, these matrices form a *subgroup* of $\mathbb{O}(n)$ which is called *special orthogonal group* and denoted $\mathbb{SO}(n)$. In order to proof that $\mathbb{SO}(n)$ is indeed a subgroup of $\mathbb{O}(n)$, it is sufficient to show that $\mathbb{SO}(n)$ is closed under matrix multiplication.

¹It can be easily checked that if G is a group and $H \subset G$ its subgroup, then their identity elements coincide. Thus, if $\mathbb{O}(n)$ is to be the subgroup of $\mathbb{GL}(n)$, then its identity element *must* be the identity matrix.

This is rather straightforward, for if R and Q are any special orthogonal matrices, then the determinant of their product is

$$\det(RQ) = \det R \det Q = 1 \cdot 1 = 1 \quad (\text{B.45})$$

and thus $RQ \in \mathbb{SO}(n)$. Simultaneously, we can see that the matrices with $\det R = -1$ *do not* constitute a subgroup of $\mathbb{O}(n)$. By the same argument, let R and Q be matrices with determinant -1 . Their product has determinant

$$\det(RQ) = \det R \det Q = (-1) \cdot (-1) = +1, \quad (\text{B.46})$$

and so the set of matrices with the determinant -1 *is not closed* under multiplication.

There is a geometric difference between orthogonal transformations with represented by matrix R with $\det R = 1$ and transformations having $\det R = -1$. Recall that, in order to define the rotation matrices, we have required that the transformation preserves lengths of vectors; this is a typical property of rotations. However, rotations are not the only transformations that preserve lengths! Consider, in the two dimensional case, the vector

$$\mathbf{v} = \begin{pmatrix} x \\ y \end{pmatrix}, \quad (\text{B.47})$$

and the matrix

$$R = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \quad (\text{B.48})$$

This is clearly the orthogonal matrix, for we have $R^T R = I$. Determinant of the matrix is

$$\det R = -1. \quad (\text{B.49})$$

Action of this matrix on the vector \mathbf{v} results in vector

$$\mathbf{v}' = R\mathbf{v} = \begin{pmatrix} x \\ -y \end{pmatrix} \quad (\text{B.50})$$

as the reader can easily check. From this we can infer that what matrix R actually does is the *reflection* about the x axis, as the figure B.3 illustrates. However, this is not a rotation, because different vectors are rotated by different angles, see again the figure B.3. Here, we have chosen one particular matrix to illustrate the point, but, in general, orthogonal group $\mathbb{O}(n)$ consists of matrices preserving lengths. Some of them are unimodular, $\det R = 1$, and they constitute the subgroup $\mathbb{SO}(n)$. These matrices represent *proper rotations*. Then, there are matrices with $\det R = -1$ and they represent rotations composed with the reflection about some axis. These matrices do not form a subgroup.

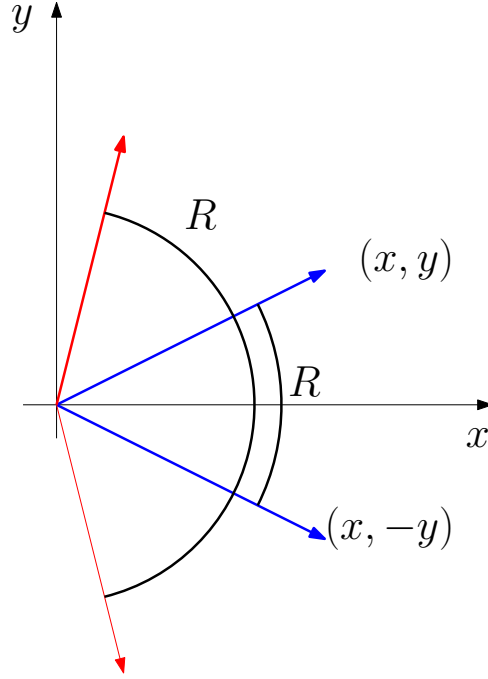


Figure B.3: Orthogonal matrices with $\det R = -1$ actually represent rotations connected with reflections. Here we show the effect of transformation with the matrix R given by (B.48). Notice that this matrix R does not represent the rotation of the plane as a whole, for different vectors are rotated by different angles.

B.5 Explicit form of $\mathbb{SO}(2)$ matrices

After the general discussion of rotation groups $\mathbb{O}(n)$ and $\mathbb{SO}(n)$, we construct explicit form of the $\mathbb{SO}(2)$ matrices, i.e. matrices representing rotations in two dimensional plane. We start with the general definition of orthogonal matrix, relation (B.34),

$$R^T R = I.$$

General matrix $R \in \mathbb{GL}(2)$ can be written in the form

$$R = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad R^T = \begin{pmatrix} a & c \\ b & d \end{pmatrix}. \quad (\text{B.51})$$

The orthogonality condition $R^T R = I$ then implies

$$\begin{pmatrix} a & c \\ b & d \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \quad (\text{B.52})$$

Written explicitly, the latter equation in components reads

$$a^2 + c^2 = 1, \quad ab + cd = 0, \quad b^2 + d^2 = 1. \quad (\text{B.53})$$

Notice that the matrix has 4 components, while we have written the three equations down only. The reason is that matrix $R^T R$ is always symmetric and, thus, has only three independent components. Nevertheless, the first of equations (B.53) is

$$a^2 + c^2 = 1. \quad (\text{B.54})$$

Since we are working in the real domain, both numbers a and c must be, in the absolute value, smaller than (or equal to) 1. We will satisfy this condition identically, if we put

$$a = \cos \phi, \quad c = \sin \phi, \quad (\text{B.55})$$

where ϕ is a real number. Similarly, the third equation, $b^2 + d^2 = 1$ will be satisfied identically if we set

$$b = \cos \theta, \quad d = \sin \theta, \quad (\text{B.56})$$

for some real number θ . However, the second equation, $ab + cd = 0$ tells us

$$\tan \theta = -\tan \phi \quad (\text{B.57})$$

and therefore $\theta = -\phi$ (careful reader can easily check that ignoring the solutions $\theta = -\pi + n\pi$ is without the loss of generality). To conclude, we have found

$$R = \begin{pmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{pmatrix}. \quad (\text{B.58})$$

This is the general form of matrix $R \in \mathbb{SO}(2)$. Obviously, parameter ϕ is to be identified with the angle of rotation and the choice of the signs has been made so that positive ϕ corresponds to counter-clockwise rotation.

Recall the trigonometric identities

$$\begin{aligned} \sin(\alpha + \beta) &= \sin \alpha \cos \beta + \sin \beta \cos \alpha, \\ \cos(\alpha + \beta) &= \cos \alpha \cos \beta - \sin \alpha \sin \beta. \end{aligned} \quad (\text{B.59})$$

With the help of these identities, we can derive the relation

$$R(\phi_1) R(\phi_2) = R(\phi_1 + \phi_2). \quad (\text{B.60})$$

This is equivalent to the statement that $\mathbb{SO}(2)$ matrices are closed under multiplication. Indeed, this result says that the composition of rotation by angle ϕ_1 with the rotation by angle ϕ_2 is equivalent to rotation by angle $\phi_1 + \phi_2$. Moreover, we have

$$R(\phi_1) R(\phi_2) = R(\phi_1 + \phi_2) = R(\phi_2 + \phi_1) = R(\phi_2) R(\phi_1), \quad (\text{B.61})$$

so that the group $\mathbb{SO}(2)$ is Abelian (commutative). Since the function $\cos \phi$ is even, while $\sin \phi$ is odd, we have

$$R(-\phi) = R(\phi)^T. \quad (\text{B.62})$$

By the orthogonality of R , we have

$$I = R^T R = R(-\phi)R(\phi), \quad (\text{B.63})$$

which translates into the statement, that the rotation through angle ϕ is inverse to rotation through angle $-\phi$.

Thus, the group of rotations in the plane is Abelian: the result of two rotations does not depend on the order in which we perform the rotations. The things get more complicated in three dimensions.

Appendix C

C++ codes

C.1 Diffusion

```
1  /*
2  under Linux, compile with
3
4  g++ -o diffusion diffusion.cpp -lboost_iostreams -lboost_system -lboost_filesystem
5
6  run with
7
8  ./diffusion
9
10 */
11
12
13 #include <iostream>
14 #include "gnuplot-iostream/gnuplot-iostream.h"
15 #include <boost/numeric/ublas/matrix.hpp>
16 #define pi 3.14159265358
17
18 using namespace std;
19
20
21 // initial temperature defined on x = -1..1
22 double u0(double x) {
23     // return 1 - x*x; // parabolic shape
24     return abs(sin( (x+1)*pi ));
25     //return exp(-100*x*x); // gaussian profile
26 }
27
28
29 int main(int argc, char** argv) {
30
31
32     // dimensions of the u
33     int M = 10000;
34     int N = 100;
35
36
37     double u[M][N]; //temperature of rod [time, position]
38
39     // clean the u
40     for (int i=0; i<M; i++)
41         for (int j=0; j<N; j++)
42             u[i][j] = 0;
43
44     // initial conditions at time 0
45     for (int j=0; j<N; j++)
```

```

46     u[0][j] = u0(-1. + 2.*j/(N-1));
47
48     // boundary conditions
49     for (int i=0; i<M; i++) {
50         u[i][0] = u0(-1);
51         u[i][N-1] = u0(1);
52     }
53
54     double dt = 1E-4; //time step
55     double dx = 2.0 / (N-1);
56
57     cout << dt/(dx*dx) << endl;
58
59     cout << "Calculating" << endl;
60
61     // solution
62     for (int i=1; i < M; i++) {
63         for (int j=1; j<N-1; j++) {
64             u[i][j] = u[i-1][j] + dt*(u[i-1][j+1] -2*u[i-1][j] + u[i-1][j-1])/(dx*dx);
65         }
66     }
67
68     // output
69
70     Gnuplot gp;
71
72     cout << "Creating images" << endl;
73
74     gp << "set term pdf" << endl;
75     gp << "set xrange [-1:1]" << endl;
76     gp << "set yrange [0:1]" << endl;
77
78
79     for (int i=0; i < M; i+=20) {
80         gp << "set output 'data/frame" << i << ".pdf'" << endl;
81         gp << "plot '-' with lines lw 4 title 'i=" << i << "'" << endl;
82         for (int j=0; j < N; j++)
83             gp << (-1.0 +2.0*j/(N-1)) << " " << u[i][j] << " " << endl;
84         gp << "e" << endl;
85         gp << "set output" << endl;
86     }
87
88     gp << "set term wxt" << endl;
89
90     cout << "done" << endl;
91 }

```

C.2 One dimensional CA

```

1
2 #include <iostream>
3 #include "gnuplot-iostream/gnuplot-iostream.h"
4 #include <boost/numeric/ublas/matrix.hpp>
5
6
7 using namespace std;
8
9
10
11 int update_cell(bool *rule, bool c1, bool c2, bool c3) {
12
13
14     //cout << 7-(c3 + 2*c2 + 4*c1);
15     return rule[7-(c3 + 2*c2 + 4*c1)];

```

```

16 }
17 }
18
19 bool *rule_to_array(int rule) {
20
21     bool *arr = new bool[8];
22
23     for (int i=7; i>= 0; i--) {
24         arr[i] = rule%2;
25         rule = rule / 2;
26     }
27
28     return arr;
29
30 }
31
32 void display_array(bool *rule) {
33     for (int i=0; i < 8; i++)
34         cout << rule[i];
35     cout << endl;
36 }
37
38
39 int main(int argc, char** argv) {
40
41     cout << "Initializing" << endl;
42     //int rule_number = 150;
43     for (int rule_number = 0; rule_number < 256; rule_number++) {
44         bool *rule;
45         rule = rule_to_array(rule_number);
46
47         Gnuplot gp;
48
49         int M = 100;
50         int N = 100;
51         bool **grid = new bool*[M];
52         for (int i=0; i<N; i++)
53             grid[i] = new bool[N];
54
55         for (int i=0; i<M; i++)
56             for (int j=0; j<N; j++)
57                 grid[i][j] = 0;
58
59         grid[0][N/3] = true;
60         grid[0][2*N/3] = true;
61         // grid[0][N/2] = true;
62
63         cout << "Calculating " << rule_number << "/255" << endl;
64
65         for (int i=1; i<M; i++) {
66
67             for (int j=1; j < N-1; j++) {
68                 grid[i][j] = update_cell(rule, grid[i-1][j-1], grid[i-1][j], grid[i-1][j+1]);
69             }
70             gp << endl;
71         }
72
73         cout << "Generating picture" << endl;
74
75         gp << "set term pngcairo" << endl;
76         gp << "set output 'data/rule" << rule_number << ".png" << endl;
77         gp << "set xrange [0:" << N << "]" << endl;
78         gp << "set yrange [0:" << M << "]" << endl;
79         gp << "unset colorbox" << endl;
80         gp << "unset xtics" << endl;
81         gp << "set palette defined (0 'white', 1 'red')" << endl;
82         gp << "unset ytics" << endl;
83         gp << "set size ratio 1" << endl;
84         gp << "plot '-' matrix with image \n";

```

```

85     for (int i=M-1; i>=0; i--) {
86         for (int j=0; j<N; j++) {
87             gp << grid[i][j] << " ";
88         }
89         gp << endl;
90     }
91     gp << 'e' << endl;
92     //gp << "set output" << endl;
93
94     cout << "Cleaning memory" << endl;
95
96     for (int j=0; j<N; j++)
97         delete[] grid[j];
98
99     delete[] grid;
100 }
101     cout << "done" << endl;
102 }

```

C.3 FHP model

```

1  /*
2
3  compile with
4
5  g++ -o main main.cpp -lboost_iostreams -lboost_system -lboost_filesystem
6
7  setting affinity
8  taskset -cp 0 30072
9
10 */
11
12 #include <iostream>
13 #include <fstream>
14 #include <sstream>
15 #include <string>
16 #include <utility>
17 #include <stdlib.h>
18 #include <math.h>
19 #include <time.h>
20 #include <boost/numeric/ublas/matrix.hpp>
21 #include <unistd.h>
22
23
24 using namespace std;
25
26
27 enum BoundaryConditions { bc_free, bc_periodic, bc_reflexive };
28
29 typedef pair<int, int> * PairList;
30
31 #define nmax 5000 // pocet kroku
32 #define dx 120 // sirka bunky, ve ktorej se urcuje prumerna rychlost
33 #define dy 60 // vyska
34 #define scale_factor 130 // delka sipky v obrazku
35 #define animation false // pokud true, vygeneruje se soubor s animaci, pokud false, ←
36 // vygeneruje se kazdy obrazek zvlast
37 #define skip_step 15
38 #define gridpoints false // zda se vykresli body mridky
39 #define density_plot true // mapa velikosti rychlosti
40 #define PI 3.14159265358
41
42 #define angle(i) (i*PI/3)
43 #define posX(i, j) ( ((j % 2)==0)?i:i+0.5 ) //

```

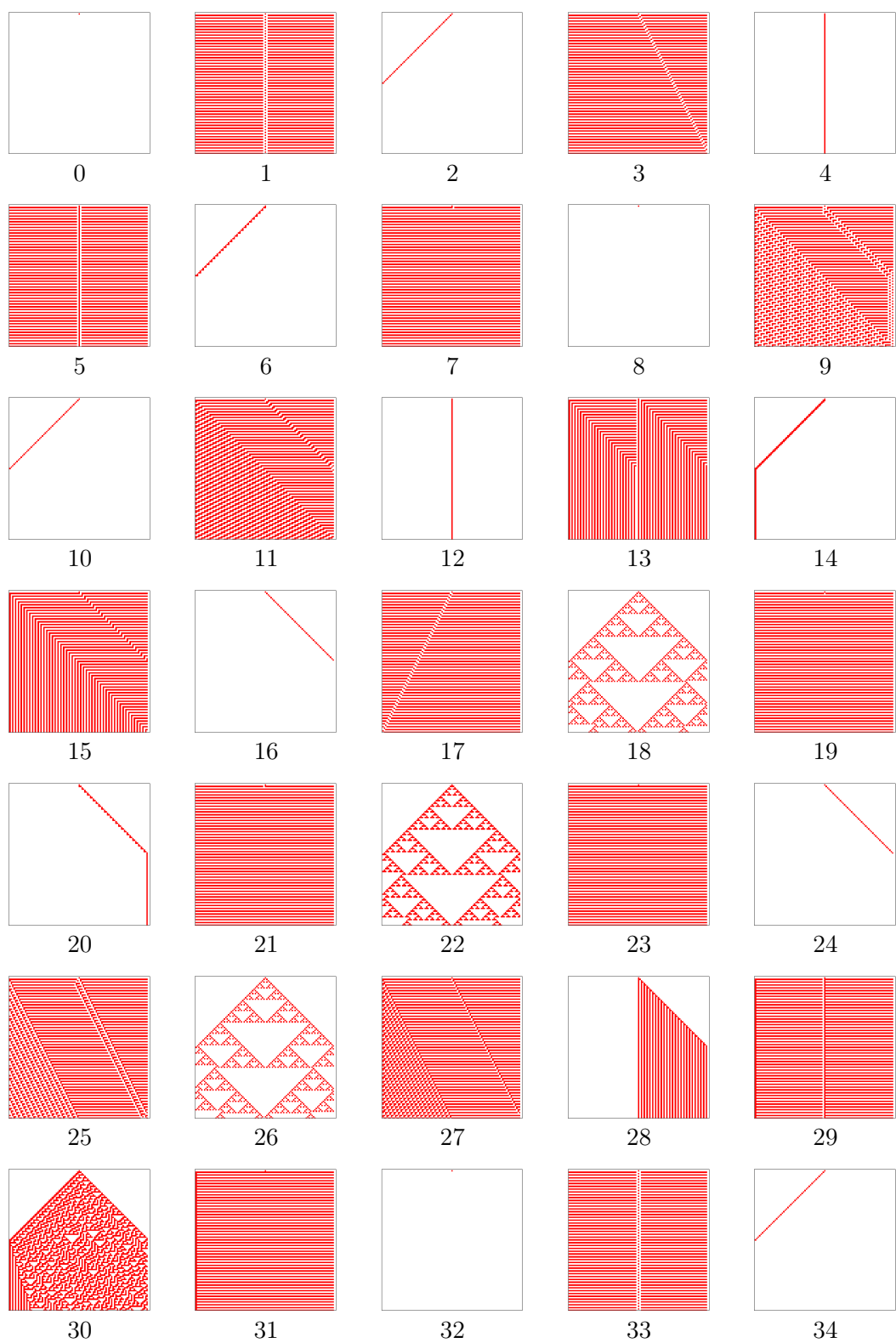



Figure C.1: One dimensional cellular automata 0–34

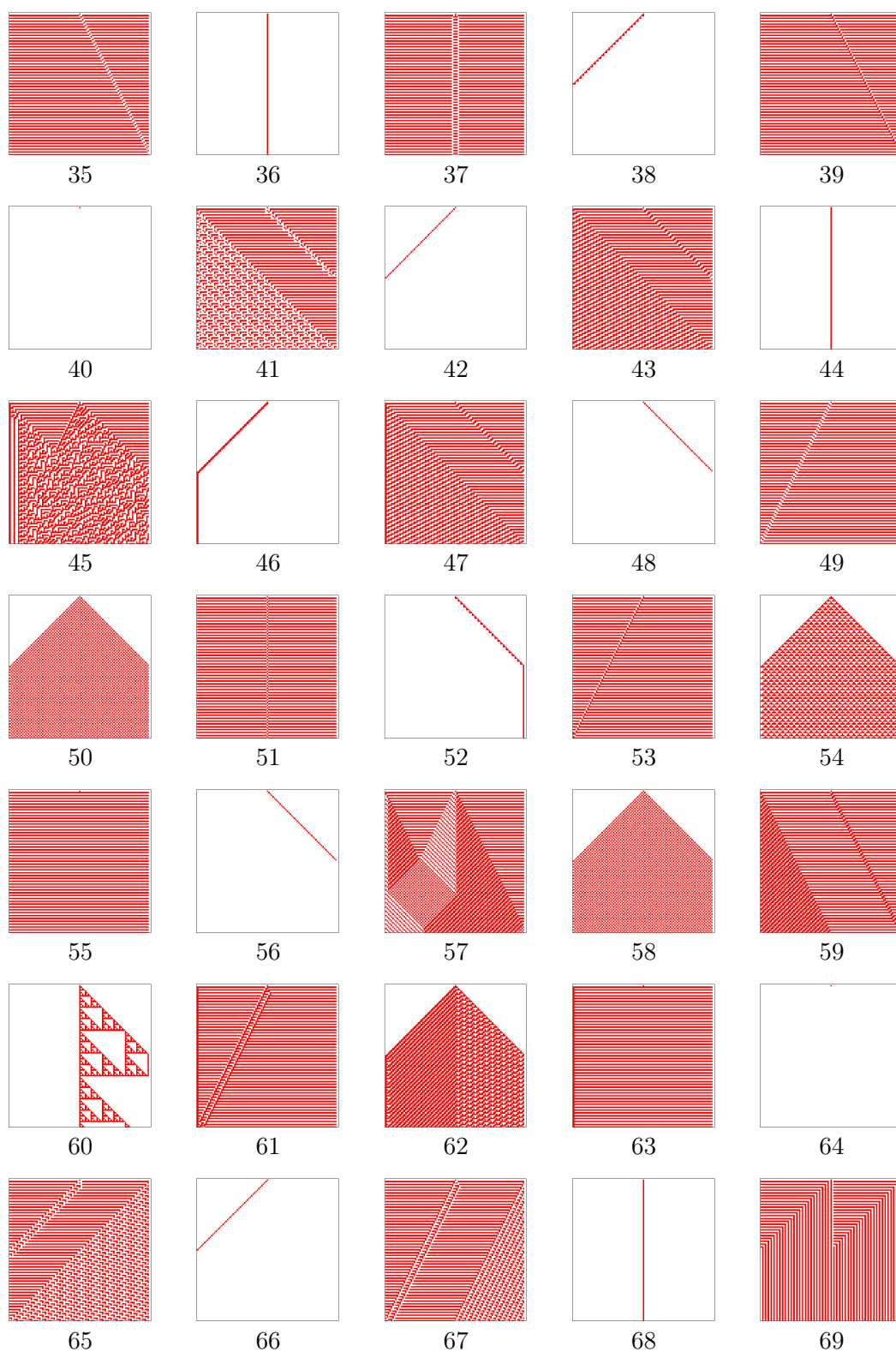


Figure C.2: One dimensional cellular automata 35–69

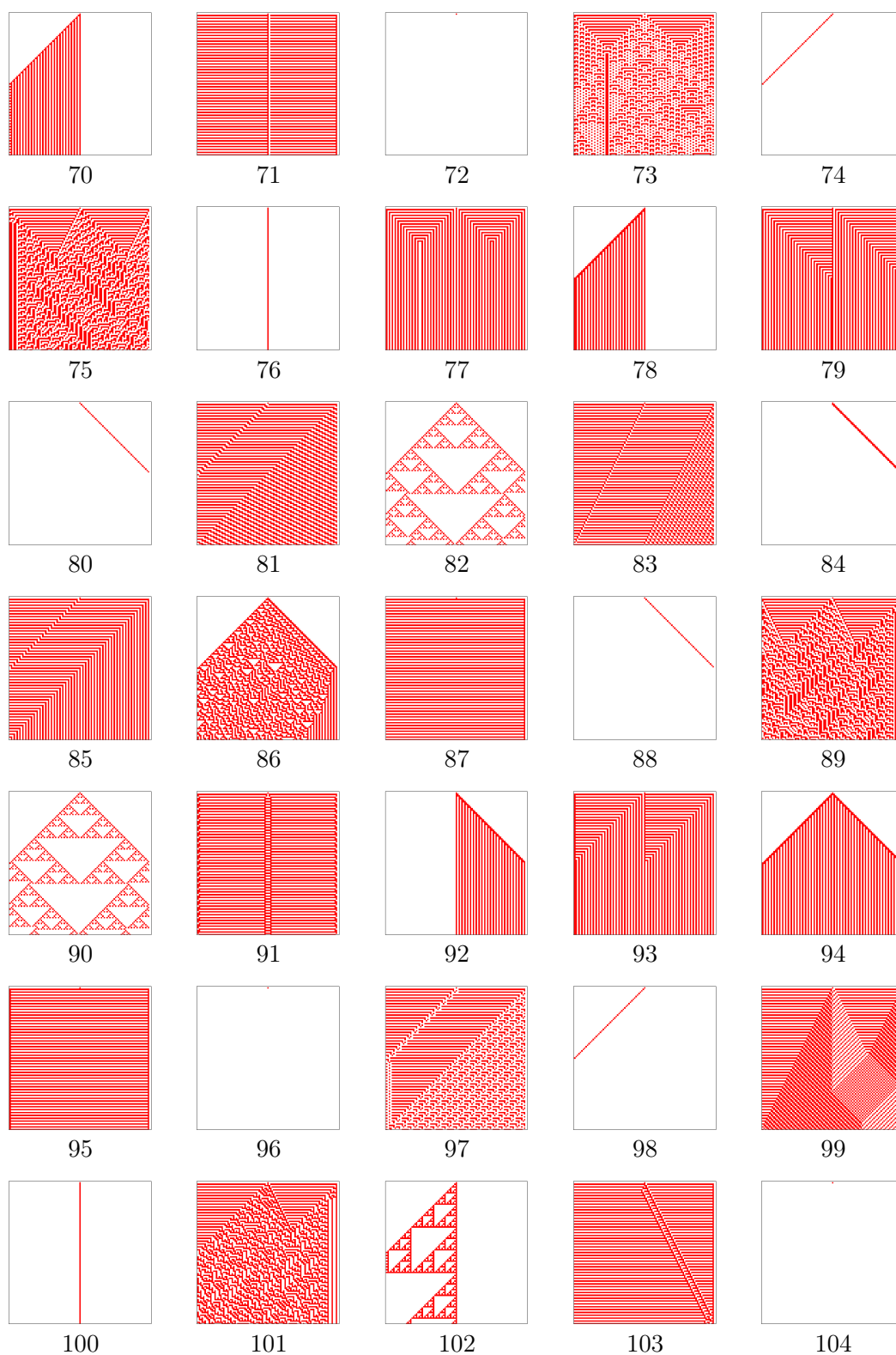


Figure C.3: One dimensional cellular automata 70–104

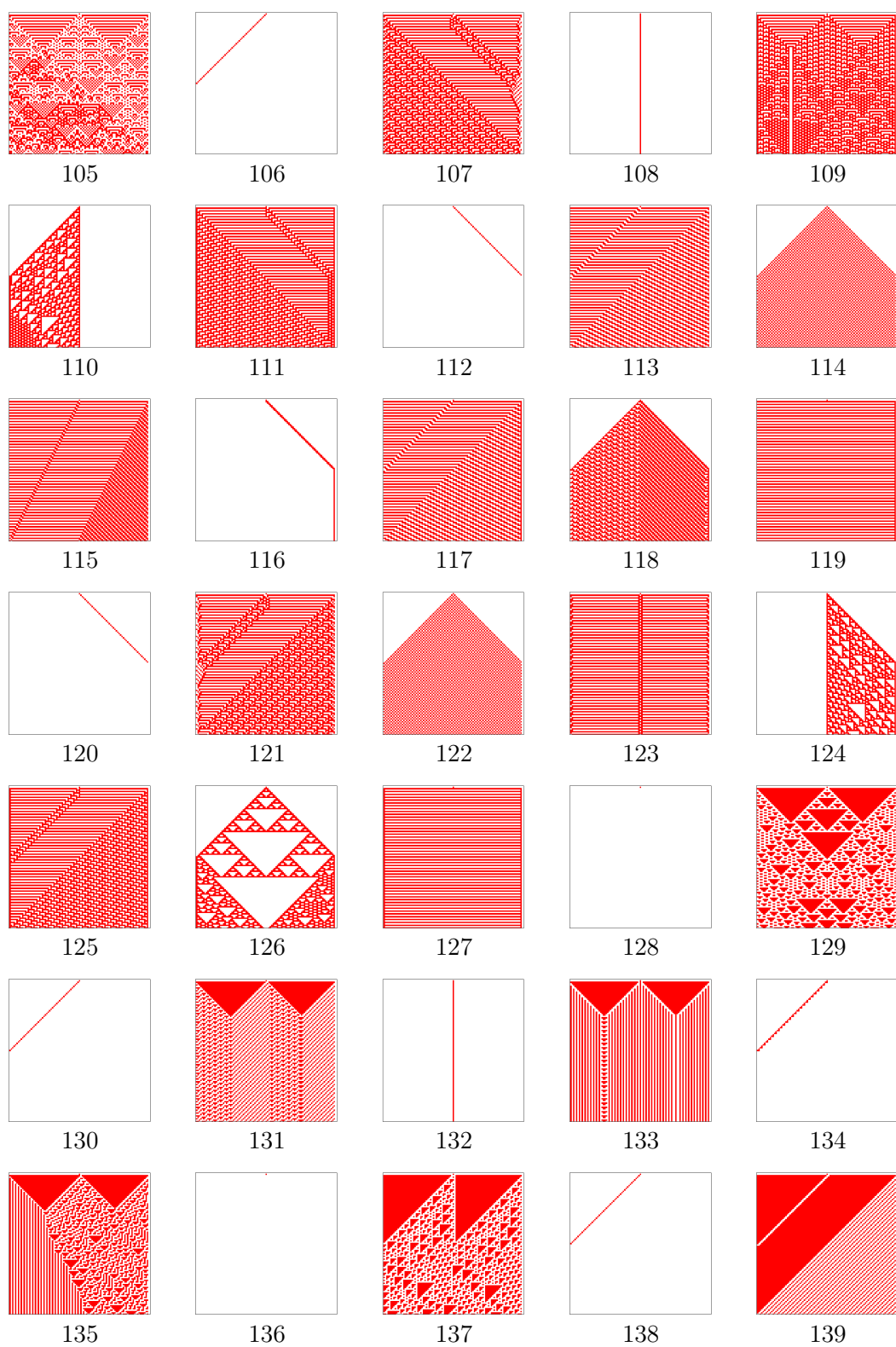


Figure C.4: One dimensional cellular automata 105–139

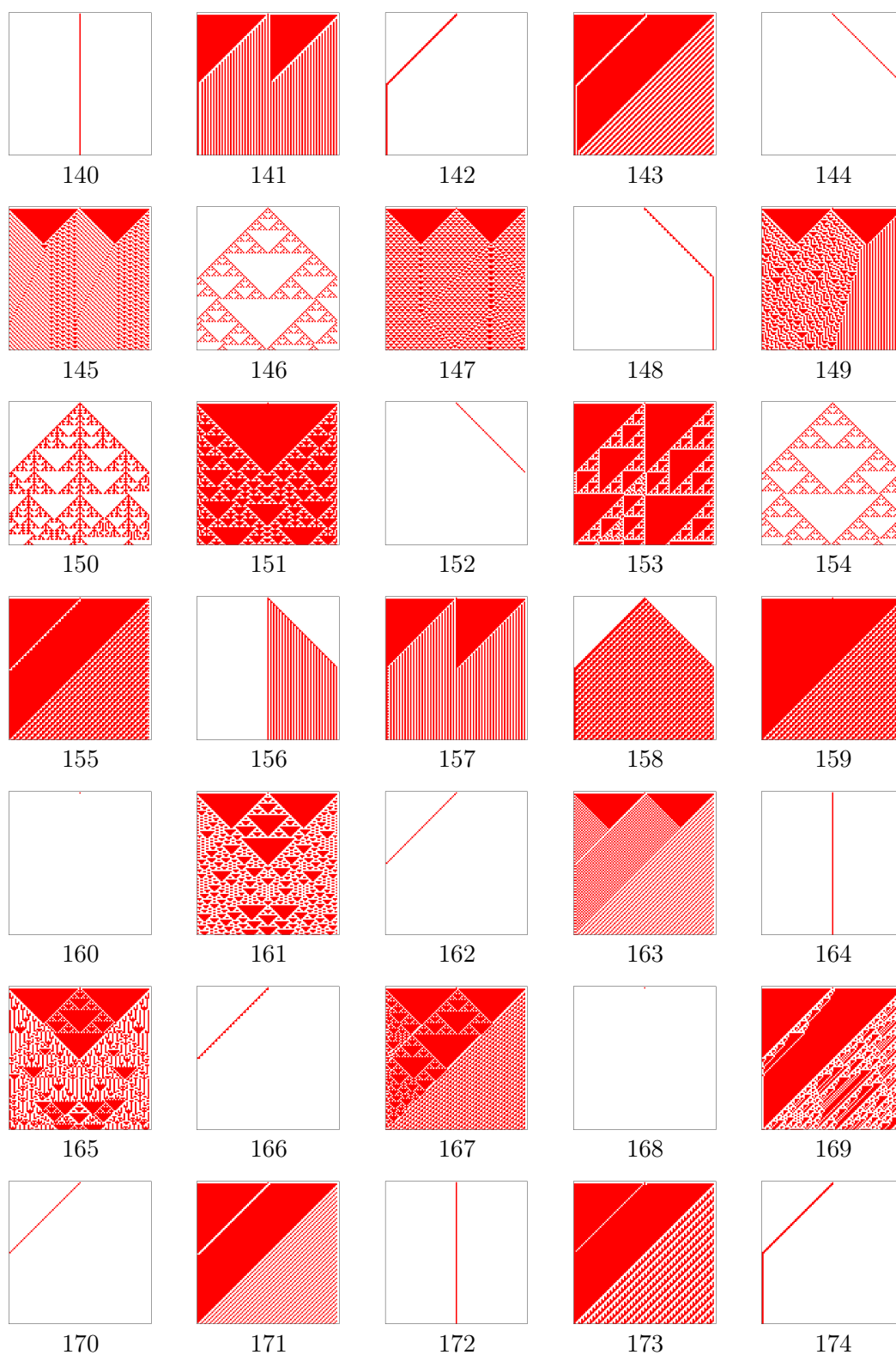


Figure C.5: One dimensional cellular automata 140–174

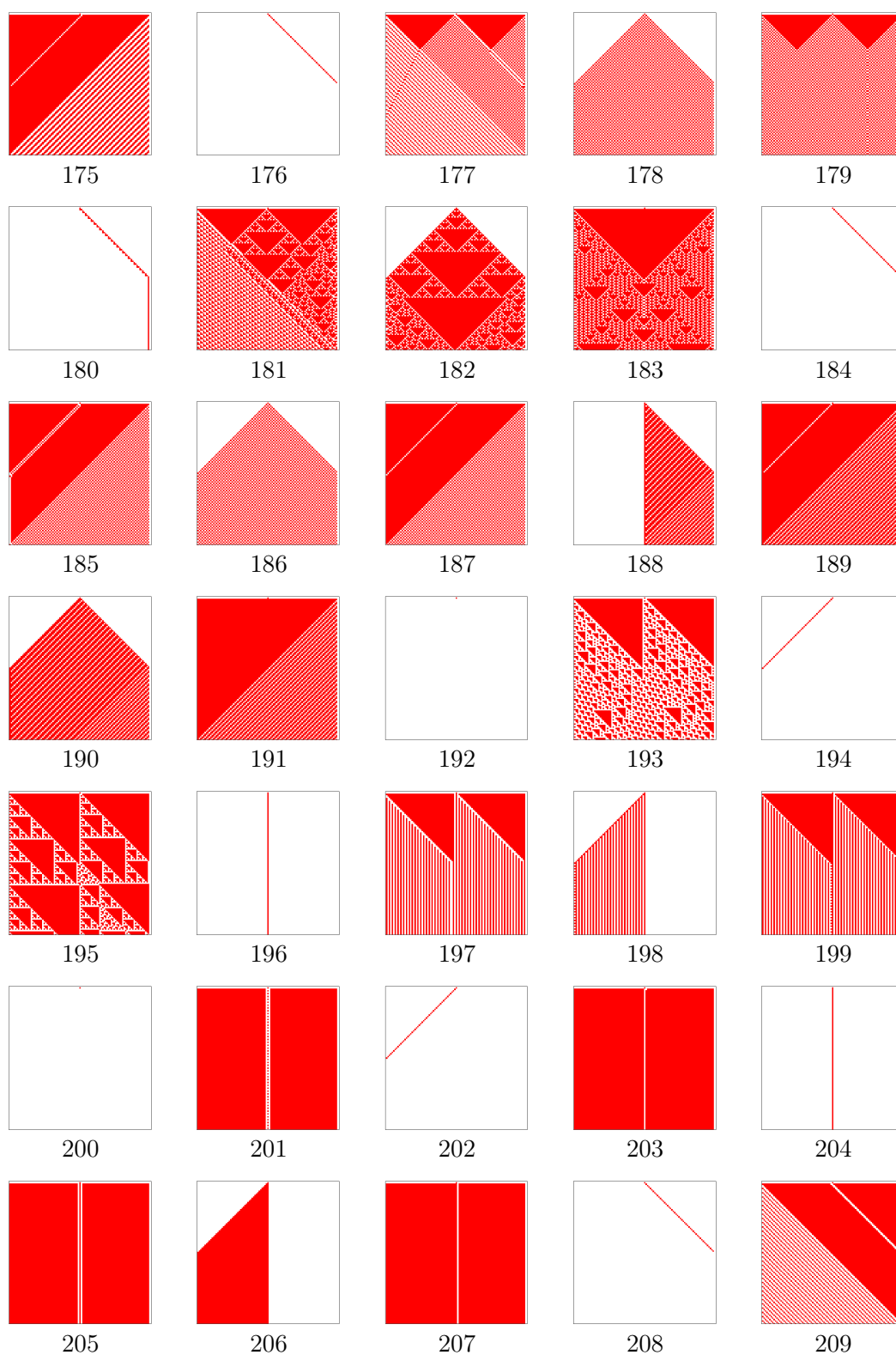


Figure C.6: One dimensional cellular automata 175–209

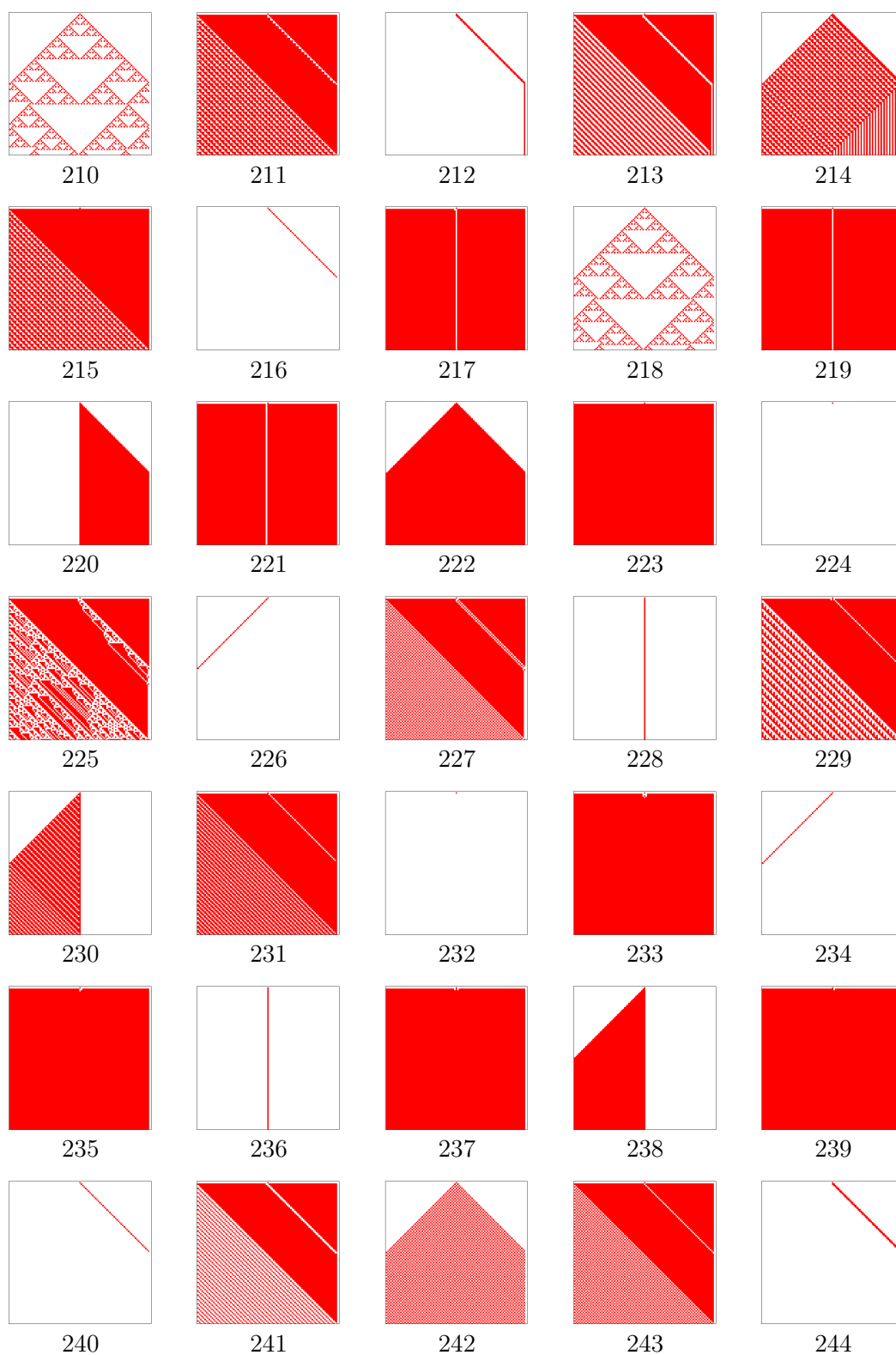


Figure C.7: One dimensional cellular automata 210-244

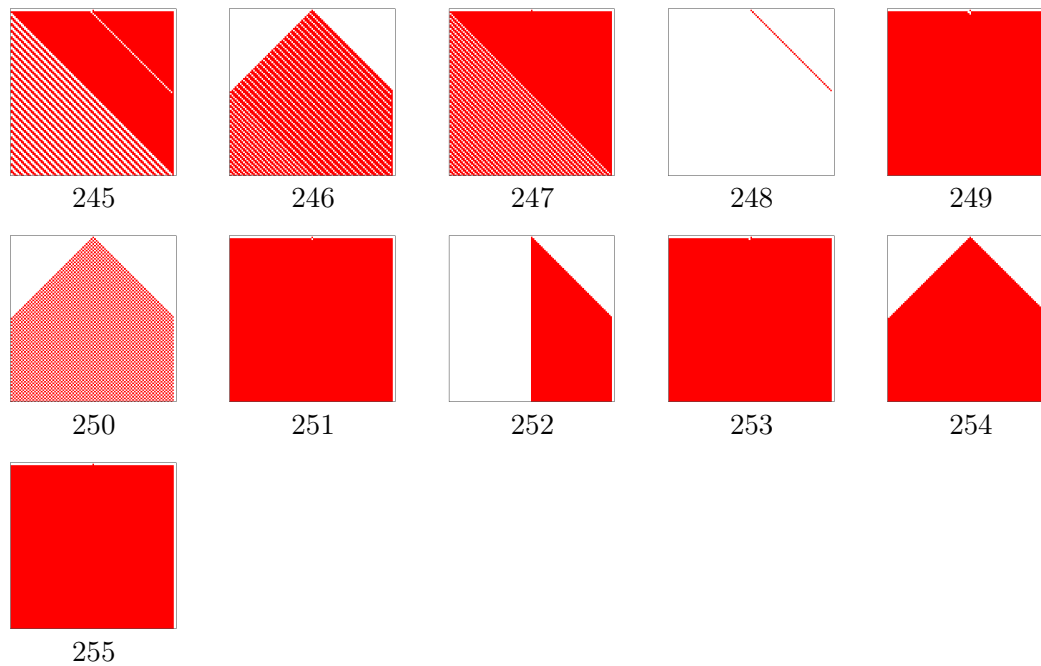


Figure C.8: One dimensional cellular automata 245-255

```

43 #define posY(i,j) (j*sqrt(3)/2)//
44 #define choice(n) (rand() % n)
45
46
47
48 bool *** M; // cells a,b,c,d,e,f and obstacle and rest particle
49 int N1;
50 int N2;
51 int flow = false;
52 bool scalar_plot = false; // pokud false, v obrazku typu densityplot je videt smer ←
    rychlosti
53 BoundaryConditions boundary_conditions;
54
55
56 int cntr1=0;
57 int cntr2=0;
58
59 ofstream gp;
60
61
62 PairList up = NULL;
63 PairList down = NULL;
64 int profile_length = 0;
65 int leading_edge = -1;
66 int trailing_edge = -1;
67
68
69
70 bool *** init_grid() {
71     bool *** m = new bool ** [N1];
72     for (int i=0; i < N1; i++) {
73         m[i] = new bool * [N2];
74         for (int j=0; j < N2; j++) {
75             m[i][j] = new bool [8];
76             for (int k=0; k < 8; k++)
77                 m[i][j][k] = false;
78         }
79     }

```



```

80     return m;
81 }
82
83 #include "init-conds.cpp"
84
85
86 void gnu_plot() {
87     ofstream out;
88
89     out.open("fhp/obstacles.dat");
90     for (int i=0; i<N1; i++)
91         for (int j=0; j<N2; j++) {
92             if (M[i][j][6])
93                 out << posX(i,j) << " " << posY(i,j) << endl;
94         }
95     out.close();
96
97     if (!gridpoints)
98         return;
99
100    out.open("fhp/gridpoints.dat");
101    for (int i=0; i<N1; i++)
102        for (int j=0; j<N2; j++)
103            out << posX(i,j) << " " << posY(i,j) << endl;
104    out.close();
105 }
106
107
108
109 void write_velocity (){
110
111     cntr2++;
112
113     // double scale_factor = (double)N1/(double)dx;
114
115     for (int i=0; i<=N1-dy; i += dy) {
116         for (int j=0; j<=N2-dx; j += dx) {
117
118
119             double vx = 0;
120             double vy = 0;
121             double x0 = posX(i,j); //posX(i,j) + posX(i,j+dx))/2;
122             double y0 = posY(i,j); //posY(i,j) + posY(i+dy,j))/2;
123             int particles = 0;
124             for (int k=i; k<i+dy; k++) {
125                 for (int l=j; l<j+dx; l++) {
126                     for (int m=0; m<6; m++) {
127                         if (M[k][l][m]) {
128                             vx += 0.5*cos(angle(m));
129                             vy += 0.5*sin(angle(m));
130                             particles++;
131                         }
132                     }
133                     if (M[k][l][7])
134                         particles++;
135                 }
136             }
137
138             if (particles!=0){
139                 vx = vx/particles;
140                 vy = vy/particles;
141                 double v = sqrt(vx*vx + vy*vy);
142                 double angle= atan(vy/vx);
143                 if (!scalar_plot)
144                     v = v * angle *10; else
145                     v = 10*v;
146                 if (!density_plot)
147                     gp << x0 << " " << y0 << " " << scale_factor*vx<< " " << ←
148                         scale_factor*vy<< endl;
149                     else {

```

```

149         gp << x0 << " " << y0 << " " << v << " ";
150 //         out<< x0 << " " << y0 << " " << v << " ";
151     }
152     gp << endl;
153 //     out << endl;
154 }
155
156
157 }
158
159 gp << endl;
160 //     out << endl;
161
162 }
163
164
165 ofstream out;
166 stringstream fileName;
167 fileName << "data/velocity";
168 fileName << cntr2 << ".dat";
169 string s = fileName.str();
170 out.open(s.c_str());
171
172 for (int k=0; k<profile_length - dx; k+=dx) {
173
174     double vx1 = 0;
175     double vy1 = 0;
176     double vx2 = 0;
177     double vy2 = 0;
178     int n1 = 0;
179     int n2 = 0;
180
181     for (int i=k; i<k+dx; i++) {
182         for (int j=up[i].second + dy; j < up[i].second + 2*dy; j++) {
183             for (int m=0; m<6; m++) {
184                 if (M[i][j][m]) {
185                     vx1 += 0.5*cos(angle(m));
186                     vy1 += 0.5*sin(angle(m));
187                     n1++;
188                 }
189                 if (M[i][j][7])
190                     n1++;
191             }
192         }
193
194         for (int j=down[i].second - dy; j > down[i].second - 2 *dy; j--) {
195             for (int m=0; m<6; m++) {
196                 if (M[i][j][m]) {
197                     vx2 += 0.5*cos(angle(m));
198                     vy2 += 0.5*sin(angle(m));
199                     n2++;
200                 }
201                 if (M[i][j][7])
202                     n2++;
203             }
204         }
205     }
206 }
207
208
209     if (n1!=0) {
210         vx1 = vx1 / n1;
211         vy1 = vy1 / n1;
212         int i0 = up[k + dx/2].first;
213         int j0 = up[k+dx/2].second + dy;
214         out << posX(i0,j0) << " " << posY(i0,j0) << " " << vx1*↵
                scale_factor << " " << scale_factor*vy1 << endl;
215
216     }
217

```

```

218     if (n2!=0) {
219         vx2 = vx2 / n2;
220         vy2 = vy2 / n2;
221         int i0 = down[k + dx/2].first;
222         int j0 = down[k+dx/2].second - dy;
223         out << posX(i0,j0) << " " << posY(i0,j0) << " " << scale_factor*←
                vx2 << " " << scale_factor*vy2 << endl;
224     }
225 }
226
227
228 }
229
230 out.close();
231
232 }
233
234
235
236 int nops(int i, int j) {
237     int n = 0;
238     for (int k=0; k<6; k++)
239         n += M[i][j][k];
240     return n;
241 }
242
243
244
245 int Mod(int a, int b) {
246
247     return ( (a>0)?(a%b):(b-((-a)%b)) );
248 }
249
250
251
252
253
254 void destroy_grid(bool *** m) {
255     for (int i=0; i<N1; i++) {
256         for (int j=0; j<N2; j++) {
257             delete [] m[i][j];
258         }
259         delete [] m[i];
260     }
261     delete [] m;
262 }
263
264
265 void update_status() {
266     // collisions
267
268     bool *** N;
269     N = init_grid();
270     int n;
271
272
273     for (int k=0; k < N1; k++)
274         for (int l=0; l < N2; l++)
275             N[k][l][6] = M[k][l][6];
276
277
278     for (int i=0; i < N1; i++)
279         for (int j=0; j < N2; j++) {
280
281
282             bool A = M[i][j][0];
283             bool B = M[i][j][1];
284             bool C = M[i][j][2];
285             bool D = M[i][j][3];
286             bool E = M[i][j][4];

```

```

287     bool F = M[i][j][5];
288     bool O = M[i][j][6]; //obstacle
289     bool R = M[i][j][7]; //rest particle
290
291     n = nops(i, j);
292
293     if (R && (n==1)) {
294         for (int k=0; k < 6; k++)
295             if (M[i][j][k]) {
296                 N[i][j][k] = false;
297                 N[i][j][7] = false;
298                 N[i][j][Mod(k+1,6)] = true;
299                 N[i][j][Mod(k-1,6)] = true;
300                 break;
301             }
302
303         continue;
304     }
305
306     if (n==2) {
307         for (int k=0; k < 6; k++) {
308             if (M[i][j][k] && M[i][j][Mod(k+2,6)]) {
309                 N[i][j][k] = false;
310                 N[i][j][Mod(k+2,6)] = false;
311                 N[i][j][7] = true;
312                 N[i][j][Mod(k+1,6)] = true;
313                 break;
314             }
315         }
316     }
317
318     if (n==2 && M[i][j][6]==true) {
319         for (int k=0; k < 6; k++) {
320             if (M[i][j][k] && M[i][j][Mod(k+2,6)]) {
321                 N[i][j][k] = false;
322                 N[i][j][Mod(k+2,6)] = false;
323                 N[i][j][7] = true;
324                 N[i][j][Mod(k+4,6)] = true;
325                 break;
326             }
327         }
328     }
329
330 }
331
332
333 if (O) {
334     N[i][j][0] = M[i][j][3];
335     N[i][j][1] = M[i][j][4];
336     N[i][j][2] = M[i][j][5];
337     N[i][j][3] = M[i][j][0];
338     N[i][j][4] = M[i][j][1];
339     N[i][j][5] = M[i][j][2];
340     N[i][j][6] = O;
341     N[i][j][7] = M[i][j][7];
342     continue;
343 }
344
345
346 bool xi=(bool)choice(2);
347 bool noxi = !xi;
348
349 bool triple = (A^B)&(B^C)&(C^D)&(D^E)&(E^F);
350
351 bool spectatorAD = (A&D)&(n==3);
352 bool spectatorBE = (B&E)&(n==3);
353 bool spectatorCF = (C&F)&(n==3);
354
355 bool db1 = A&D&(!(B|C|E|F));
356 bool db2 = B&E&(!(A|C|D|F));

```

```

357     bool db3 = C&F&!(A|B|D|E);
358
359     bool cha = (spectatorCF&(E|B)) | (spectatorBE&(F|C)) | spectatorAD | ↔
        triple | db1 | (db2&xi) | (db3&noxi);
360     bool chd = cha;
361     bool chb = (spectatorCF&(A|D)) | spectatorBE | (spectatorAD&(F|C)) | ↔
        spectatorBE | triple | db2 | (db1&noxi) | (db3&xi);
362     bool che = chb;
363     bool chc = spectatorCF | (spectatorBE&(A|D)) | (spectatorAD&(E|B)) | ↔
        spectatorCF | triple | db3 | (db1&xi) | (db2&noxi);
364     bool chf = chc;
365
366
367     N[i][j][0] = A^cha;
368     N[i][j][1] = B^chb;
369     N[i][j][2] = C^chc;
370     N[i][j][3] = D^chd;
371     N[i][j][4] = E^che;
372     N[i][j][5] = F^chf;
373     N[i][j][6] = 0;
374     N[i][j][7] = M[i][j][7];
375
376 }
377
378 destroy_grid(M);
379 M = init_grid();
380
381 for (int i=0; i<N1; i++) {
382     for (int j=0; j<N2; j++) {
383         M[i][j][6] = N[i][j][6];
384         M[i][j][7] = N[i][j][7];
385     }
386 }
387
388 // moving particles in interior grid
389 for (int i=1; i < N1-1; i++) {
390     for (int j=1; j < N2-1; j++) {
391         //for (int k=1; k<7; k++) {
392         M[i][j][0] = N[i-1][j][0];
393         M[i][j][1] = N[i-((j+1)%2)][j-1][1];
394         M[i][j][2] = N[i+(j%2)][j-1][2];
395         M[i][j][3] = N[i+1][j][3];
396         M[i][j][4] = N[i+(j%2)][j+1][4];
397         M[i][j][5] = N[i-((j+1)%2)][j+1][5];
398         M[i][j][6] = N[i][j][6];
399         M[i][j][7] = N[i][j][7];
400     }
401     // bottom boundary
402     M[i][0][0] = N[i-1][0][0];
403     M[i][0][3] = N[i+1][0][3];
404     M[i][0][4] = N[i][1][4];
405     M[i][0][5] = N[i-1][1][5];
406     // top boundary
407     int shift = (N2)%2;
408     M[i][N2-1][0] = N[i-1][N2-1][0];
409     M[i][N2-1][1] = N[i-shift][N2-2][1];
410     M[i][N2-1][2] = N[i+1-shift][N2-2][2];
411     M[i][N2-1][3] = N[i+1][N2-1][3];
412 }
413
414
415
416 // left-right boundaries
417 for (int j=1; j < N2-1; j++) {
418
419     if (j%2==1) {
420         M[0][j][1] = N[0][j-1][1];
421         M[0][j][5] = N[0][j+1][5];
422     } else {
423         M[N1-1][j][4] = N[N1-1][j+1][4];

```

```

424     M[N1-1][j][2] = N[N1-1][j-1][2];
425 }
426
427 M[0][j][2] = N[j%2][j-1][2];
428 M[0][j][3] = N[1][j][3];
429 M[0][j][4] = N[j%2][j+1][4];
430
431 M[N1-1][j][0] = N[N1-2][j][0];
432 M[N1-1][j][1] = N[N1-1-(1-j%2)][j-1][1];
433 M[N1-1][j][5] = N[N1-1-(1-j%2)][j+1][5];
434 }
435
436
437 // corners
438 M[0][0][3] = N[1][0][3];
439 M[0][0][4] = N[0][1][4];
440 M[N1-1][0][0] = N[N1-2][0][0];
441 M[N1-1][0][4] = N[N1-1][1][4];
442 M[N1-1][0][5] = N[N1-2][1][5];
443 if (N2%2==0) {
444     M[0][N2-1][2] = N[1][N2-2][2];
445     M[0][N2-1][3] = N[1][N2-1][3];
446
447     M[N1-1][N2-1][1] = N[N1-1][N2-2][1];
448     M[0][N2-1][1] = N[0][N2-2][1];
449 } else {
450
451     M[0][N2-1][2] = N[0][N2-2][2];
452     M[N1-1][N2-1][2] = N[N1-1][N2-2][2];
453
454     M[N1-1][N2-1][1] = N[N1-2][N2-2][1];
455     M[N1-1][N2-1][0] = N[N1-2][N2-1][0];
456 }
457 M[N1-1][N2-1][0] = N[N1-2][N2-1][0];
458 M[0][N2-1][3] = N[1][N2-1][3];
459
460
461 // different boundary conditions
462
463 switch (boundary_conditions) {
464
465     case bc_free: break;
466
467     case bc_reflexive: break;
468
469     case bc_periodic:
470         // top-bottom boundary
471         for (int i=1; i<N1-1; i++) {
472             M[i][0][1] = N[i-1][N2-1][1];
473             M[i][0][2] = N[i][N2-1][2];
474             M[i][N2-1][4] = N[i+1][0][4];
475             M[i][N2-1][5] = N[i][0][5];
476         }
477         // left-right boundary
478         for (int j=1; j<N2-1; j++) {
479
480             if (j%2==0) {
481                 M[0][j][1] = N[N1-1][j-1][1];
482                 M[0][j][5] = N[N1-1][j+1][5];
483             } else {
484                 M[N1-1][j][4] = N[0][j+1][4];
485                 M[N1-1][j][2] = N[0][j-1][2];
486             }
487             M[N1-1][j][3] = N[0][j][3];
488             M[0][j][0] = N[N1-1][j][0];
489         }
490         // corners
491         M[0][0][1] = N[N1-1][N2-1][1];
492         M[0][0][2] = N[0][N2-1][2];
493         M[0][0][5] = N[N1-1][1][5];

```

```

494
495         M[N1-1][0][1] = N[N1-2][N2-1][1];
496         M[N1-1][0][2] = N[N1-1][N2-1][2];
497         M[N1-1][0][3] = N[0][0][3];
498
499         M[0][N2-1][0] = N[N1-1][N2-1][0];
500         M[0][N2-1][4] = N[1][0][4];
501         M[0][N2-1][5] = N[0][0][5];
502
503         M[N1-1][N2-1][2] = N[0][N2-2][2];
504         M[N1-1][N2-1][3] = N[0][N2-1][3];
505         M[N1-1][N2-1][4] = N[0][0][4];
506         M[N1-1][N2-1][5] = N[N1-1][0][5];
507
508         break;
509     }
510 }
511
512 if (flow) {
513     for (int j=2; j < N2-2; j++)
514         M[0][j][0] = true;
515 }
516
517 destroy_grid(N);
518 }
519
520 }
521
522
523
524
525 int main()
526 {
527     srand (time(NULL));
528
529
530     cout << "Initialization" << endl;
531     cout << "Process ID " << getpid() << endl;
532     cout << "Building grid and obstacles" << endl;
533
534     //test_periodic_boundary();
535     //star();
536
537     boundary_conditions = bc_free;
538
539
540
541     airfoil(4800, 2400, -20);
542
543     //fill_fluid();
544     make_tunnel();
545     //fill_fluid();
546     flow = true;
547     boundary_conditions = bc_free;
548
549
550     cout << "Dimension of grid = " << N1 << "x" << N2 << endl;
551
552     gnu_plot();
553
554     gp.open("fhp/gnuplot.dat");
555
556     gp << "set xrange [-2:" << posX(N1, N2)+2 << "]" << endl;
557     gp << "set yrange [-2:" << posY(N1, N2) +2 << "]" << endl;
558     gp << "set pointsize 1" << endl;
559     gp << "set size ratio 1" << endl;
560
561
562
563

```

```

564 int counter = -1;
565 cout << (scalar_plot?"scalar plot":"vector plot") << endl;
566
567
568 cout << "Calculation in progress: " << endl;
569
570 for (int i = 0; i<nmax; i++) {
571
572
573
574
575 // update status of grid
576 counter ++;
577 if (counter%skip_step==0) {
578     counter = 1;
579
580     cout << "\033[J\033[F";
581     cout << "Calculation in progress: ";
582     cout << i << "/" << nmax << "(" << 100*i/nmax << "%)" << endl;
583
584
585 // write data to gnuplot
586 gp << "unset xtics" << endl << "unset ytics" << endl;
587 gp << "set term pngcairo" << endl;
588 gp << "set output 'fhp/ex5-velocity' << cntr2 << ".png'" << endl;
589
590 if (!density_plot) {
591     gp << "plot ";
592     if (gridpoints)
593         gp << "'fhp/gridpoints.dat' with points lc rgb 'gray' title 'FHP grid←
594             ', ";
595
596     gp << "'-' with vectors lw 1 lc rgb 'red' title 'particles', 'fhp/←
597         obstacles.dat' with points pointtype 5 lc rgb 'black' ps 0.3";
598     gp << endl;
599 } else {
600     gp << "set cbrange [-1:1]" << endl;
601     gp << "set pm3d map" << endl;
602     gp << "set pm3d interpolate 2,2" << endl;
603     gp << "splot '-' with pm3d, 'fhp/obstacles.dat' using 1:2:(0.0) with ←
604         points pointtype 5 lc rgb 'black' ps 0.3" << endl;
605
606 }
607 write_velocity();
608 gp << "e" << endl;
609 gp << "set output" << endl;
610 }
611 update_status();
612
613 }
614
615 cout << "\033[J\033[FCalculation finished " << endl;
616
617 cout << "Releasing resources" << endl;
618 destroy_grid(M);
619
620 if (up!=NULL)
621     delete [] up;
622 if (down!=NULL)
623     delete [] down;
624
625 cout << "Done!" << endl;
626
627 gp.close();
628
629 return 0;
630 }

```

```

1 /// different initial conditions

```



```

2
3 // creates particles in all directions
4
5 void star() {
6
7     N1 = 20;
8     N2 = 15;
9     M = init_grid();
10    int i = N1/2;
11    int j = N2/2;
12    M[i][j][0] = true;
13    M[i][j+1][1] = true;
14    M[i-1][j+1][2] = true;
15    M[i-2][j][3] = true;
16    M[i-1][j-1][4] = true;
17    M[i][j-1][5] = true;
18 }
19
20 void test_boundary() {
21
22     N1 = 20;
23     N2 = 15;
24     M = init_grid();
25
26     M[18][N2/2][0] = true;
27     M[18][13][2] = true;
28     M[18][2][5] = true;
29     flow = true;
30 }
31
32 void test_periodic_boundary() {
33
34     N1 = 8;
35     N2 = 8;
36     M = init_grid();
37     //M[N1/2][N2/2][1] = true;
38     //M[N1/2][N2-3][2] = true;
39     M[3][6][1] = true;
40     M[1][2][5] = true;
41     M[4][5][0] = true;
42 }
43
44
45 void test_2_collisions() {
46
47     N1 = 10;
48     N2 = 10;
49     M = init_grid();
50
51     M[2][2][0] = true;
52     M[4][2][3] = true;
53
54     M[5][5][0] = true;
55     M[7][5][3] = true;
56 }
57
58
59 void test_3_collisions() {
60
61     N1 = 10;
62     N2 = 10;
63     M = init_grid();
64
65     M[2][2][1] = true;
66     M[2][4][5] = true;
67     M[3][3][3] = true;
68
69     M[7][5][1] = true;
70     M[7][7][5] = true;
71     M[9][6][3] = true;

```

```

72 }
73 }
74
75 void wall2() {
76     for (int j=(150); j<=(250); j++){
77
78         M[j][j][6]=true;
79
80     }
81 }
82
83 }
84
85 }
86
87
88 // grid must be initialized to call this function
89 void make_tunnel() {
90
91     for (int i=0; i<N1; i++) {
92         M[i][0][6] = true;
93         M[i][1][6] = true;
94         M[i][N2-1][6] = true;
95         M[i][N2-2][6] = true;
96     }
97 }
98
99
100
101 void wall(int n1, int n2) {
102
103     N1 = n1;
104     N2 = n2;
105     M = init_grid();
106     flow = true;
107
108     for (int j=N2/3; j <2*N2/3; j++)
109         M[N1/2][j][6] = true;
110 }
111
112
113 void fill_fluid(int imin) {
114
115     for (int i=0; i<imin; i++)
116         for (int j=0; j<N2; j++)
117             if (!M[i][j][6])
118                 M[i][j][0]=true;
119 }
120
121
122
123 void fill_grid(int c) {
124
125     for (int i=0; i<N1; i++)
126         for (int j=0; j<N2; j++)
127             if (!M[i][j][6])
128                 M[i][j][c]=true;
129 }
130
131
132 void show_spectator() {
133     // N1 = 10 N2 = 10
134     M[80][30][0] = true;
135     M[5][5][5] = true;
136     M[5][5][4] = true;
137 }
138
139
140 void boundary() {
141

```

```

142     for (int i=0; i<N1; i++) {
143
144         M[i][1][6] = true;
145         M[i][N2-2][6] = true;
146         M[i][N2-1][6] = true;
147         M[i][0][6] = true;
148     }
149 }
150
151
152
153 // vsude,kde neni prekazka, nasadi nepohyblivou castici
154 void rest_fluid () {
155     for (int i=1; i<N1; i++)
156         for (int j=1; j<N2; j++)
157             if (!M[i][j][6])
158                 M[i][j][7] = true;
159 }
160
161
162
163 bool obstacle_on_right(int i, int j) {
164
165     for (int k=i+1; k< N1; k++)
166         if (M[k][j][6])
167             return true;
168
169     return false;
170 }
171
172 int find_next_obstacle(int i, int j) {
173
174     for (int k=i+1; k<N1; k++)
175         if (M[k][j][6])
176             return k;
177     return -1;
178 }
179
180 int find_next_empty_cell(int i, int j) {
181
182     for (int k=i+1; k<N1; k++)
183         if (!M[k][j][6])
184             return k;
185     return -1;
186 }
187
188
189 /// maps [0,1] x [0,1] onto [N1/3,2*N1/3] x [N2/3,2*N2/3]
190 void profile(int n1, int n2, double t0, double t1, double(*fx)(double t), double(*fy)(double t)) {
191
192     N1 = n1;
193     N2 = n2;
194     M = init_grid();
195
196     int obstacles[N2];
197     for (int j = 0; j<N2; j++)
198         obstacles[j] = 0;
199
200
201     for (double t=t0; t<=t1; t+= 1E-3) {
202         int i = (int)(N1/3. * (1 + fx(t) ) );
203         int j = (int)(N2/3. * (1 + fy(t) ) );
204         //cout << i << " " << j << endl;
205         if (!M[i][j][6]) {
206             M[i][j][6] = true;
207             obstacles[j] ++;
208         }
209     }
210 return;

```

```

211 }
212 }
213 }
214 }
215 }
216 }
217 /* cylinder, t = 0..2*PI */
218 double cyl_x(double t) {
219     return 0.5+0.5*cos(t);
220 }
221 }
222 }
223 }
224 double cyl_y(double t) {
225     return 0.5+0.5*sin(t);
226 }
227 }
228 }
229 void cylinder(int n1, int n2) {
230     profile(n1, n2, 0, 2*PI, &cyl_x, &cyl_y);
231 }
232 }
233 }
234 }
235 }
236 }
237 // parabola, t = -1..2
238 double par_x(double t) {
239     return (t<=1)?0.5*(1+t*t):1;
240 }
241 }
242 }
243 }
244 double par_y(double t) {
245     return (t<=1)?0.5*(1+t):(t-1);
246 }
247 }
248 void parabola(int n1, int n2) {
249     profile(n1, n2, -1, 2, &par_x, &par_y);
250 }
251 }
252 }
253 }
254 }
255 // double loop, t = -1..2
256 double loop_x(double t) {
257     return 0.5 + 0.5*cos(t);
258 }
259 }
260 }
261 }
262 double loop_y(double t) {
263     return 0.5 + 0.5*cos(2*t+0.5);
264 }
265 }
266 void double_loop(int n1, int n2) {
267     profile(n1, n2, 0, 8*PI, &loop_x, &loop_y);
268 }
269 }
270 }
271 void airfoil(int n1, int n2, double angle) {
272     ifstream in;
273     in.open("profiles/66,2-(1.8)15.5.dat");
274 }
275 }
276 N1 = n1; N2 = n2;
277 M = init_grid();
278 }
279 int obstacles[N2];
280 for (int j = 0; j<N2; j++)

```

```

281     obstacles[j] = 0;
282
283     int imin = N1;
284     int imax = 0;
285     int jmin = N2;
286     int jmax = 0;
287
288     while (!in.eof()) {
289         double x, y;
290         in >> x;
291         in >> y;
292
293
294         double alpha = angle* PI / 180.0;
295
296         double X = x * cos(alpha) - y*sin(alpha);
297         double Y = x * sin(alpha) + y*cos(alpha);
298
299         int i = (int)(N1/4. * (1. + X ) );
300         int j = (int)(N2/4. * (1. + Y ) ) + N2/3.;
301         if (i<imin)
302             imin = i;
303         if (i>imax)
304             imax = i;
305         if (j<jmin)
306             jmin = j;
307         if (j>jmax)
308             jmax = j;
309         if (!M[i][j][6])
310             obstacles[j]+=1;
311         M[i][j][6] = true;
312     }
313
314     in.close();
315
316     for (int j = 2; j<N2-2; j++) {
317         if (obstacles[j]>1) {
318             int i = 1;
319             bool inside = false;
320             while (i<N1) {
321                 i = find_next_obstacle(i,j);
322                 if (i<0)
323                     break;
324
325                 i = find_next_empty_cell(i,j);
326                 if (obstacle_on_right(i,j))
327                     inside = !inside;
328
329                 if (inside) {
330                     while ((i<N1-1)&&(!M[i][j][6])) {
331                         M[i][j][6] = true;
332                         i++;
333                     };
334                     inside = false;
335                 }
336             }
337         }
338     }
339 }
340
341 /// storing the profile
342
343 up = new pair<int, int>[imax-imin+1];
344 down = new pair<int, int>[imax-imin+1];
345
346 cout << "imin = " << imin << " imax = " << imax << endl;
347
348 for (int i=imin; i<imax; i++) {
349     int j = jmin;
350     while (!M[i][j][6])

```

```

351         j++;
352         down[i-imin] = make_pair(i,j);
353         while (M[i][j][6])
354             j++;
355         up[i-imin] = make_pair(i,j);
356     }
357
358     profile_length = imax-imin;
359     leading_edge = imin;
360     trailing_edge = imax;
361
362     /*
363     for (int i=0; i<profile_length; i++)
364         cout << "down (" << down[i].first << ", " << down[i].second << "),    up (" << ←
365             up[i].first << ", " << up[i].second << ")" << endl;
366     */
367     fill_fluid(imin);
368 }

```

Bibliography

- [1] <http://mail.tku.edu.tw/095980/airfoil%20design.pdf>.
- [2] <http://www.flyingmag.com/technicalities/technicalities-short-history-airfoils>.
- [3] <http://airfoiltools.com/>.
- [4] V.I. Arnold and A. Avez. *Ergodic Problems of Classical Mechanics*. The Mathematical physics monograph series. Benjamin, 1968.
- [5] John Bardeen, Leon N Cooper, and J Robert Schrieffer. Microscopic theory of superconductivity. *Physical Review*, 106(1):162, 1957.
- [6] G.D. Doolen. *Lattice Gas Methods: Theory, Applications, and Hardware*. A Bradford book. A Bradford Book, 1991.
- [7] B. Hasslacher Frisch, U. and Y. Pomeau. Lattice-gas automata for the navier-stokes equation. *Physical Review Letters*, 56(14):1505–1508, 1986.
- [8] D. d’Humieres B. Hasslacher P. Lallemand Y. Pomeau Frisch, U. and J.P. Rivet. Lattice gas hydrodynamics in two and three dimensions. *Complex Systems*, 1:649–707, 1987.
- [9] M Gardner. *Mathematical Games – The fantastic combinations of John Conway’s new solitaire game ‘life’*. Scientific American, 1970.
- [10] H Goldstein, C P Poole, and J L Safko. *Classical Mechanics (3rd Edition)*. Addison-Wesley, 3 edition, June 2001.
- [11] Y. Pomeau Hardy, J. and O. de Pazzis. Time evolution of a two-dimensional model system. i. invariant states and time correlation functions. *J. Math. Phys.*, 14:1746–1759, 1973.
- [12] Michel Hénon. Viscosity of a lattice gas. *Complex systems*, 1(4):762–790, 1987.
- [13] Charles Kittel and Ching-yao Fong. *Quantum theory of solids*, volume 33. Wiley New York, 1963.
- [14] M. Kutrib, R. Vollmar, and Th. Worsch. Introduction to the special issue on cellular automata. *Parallel Comput.*, 23(11):1567–1576, November 1997.
- [15] L. D. Landau and L. M. Lifshitz. *Quantum Mechanics Non-Relativistic Theory, Third Edition: Volume 3*. Butterworth-Heinemann, 3 edition, January 1981.

- [16] A. Papoulis and S.U. Pillai. *Probability, random variables, and stochastic processes*. McGraw-Hill electrical and electronic engineering series. McGraw-Hill, 2002.
- [17] R K Pathria and Paul D. Beale. *Statistical mechanics*. Academic Press, Boston, 3rd edition, 2011.
- [18] R. Penrose. *Cycles of Time: An Extraordinary New View of the Universe*. Knopf Doubleday Publishing Group, 2011.
- [19] C. Pethick and H. Smith. *Bose-Einstein condensation in dilute gases*. Cambridge University Press, 2002.
- [20] Constantine Pozrikidis. *Fluid dynamics: theory, computation, and numerical simulation*. Springer Science & Business Media, 2009.
- [21] Wolfram S. Cellular automata. *Los Alamos Science*, 9:2–21, 1983.
- [22] Wolfram S. Cellular automaton fluids: basic theory. *J. Stat. Phys.*, 45(3/4):471–526, 1986.
- [23] Leonard Susskind and James Lindesay. *An introduction to black holes, information and the string theory revolution*. OECD Publishing, 2005.
- [24] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.
- [25] Petr Veselý. Selected problems in computational fluid dynamics. Master’s thesis, Czech Technical University in Prague, Faculty of Transportation Sciences, 2013. <http://utf.mff.cuni.cz/~scholtz/index.php?section=students>.
- [26] John Von Neumann, Arthur W Burks, et al. Theory of self-reproducing automata. *IEEE Transactions on Neural Networks*, 5(1):3–14, 1966.
- [27] Peter Walters. An introduction to ergodic theory. Graduate Texts in Mathematics, Vol. 79. New York-Heidelberg-Berlin: Springer-Verlag. IX, 250 p. DM 69.50; \$ 32.40 (1982)., 1982.
- [28] Steven Weinberg. *The quantum theory of fields*. Cambridge university press, 1996.
- [29] D.A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models: An Introduction*. Number 1725 in Lattice Boltzmann modeling: an introduction for geoscientists and engineers. Springer, 2000.