Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-MSC-2011-03

Masterarbeit

im Studiengang "Angewandte Informatik"

Simulation of weak turbulent Rayleigh-Bénard convection on a GPU

Jens Zudrop

am Max-Planck-Institut für
Dynamik und Selbstorganisation

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

11. April 2011

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel.	+49 (5 51) 39-1 44 14
Fax	+49 (5 51) 39-1 44 15
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 11. April 2011

Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-MSC-2011-03

Master Thesis

"Applied Computer Science"

Simulation of weak turbulent Rayleigh-Bénard convection on a GPU

Jens Zudrop

Supervised by:
Prof. Dr. E. Bodenschatz
Max Planck Institute for Dynamic and Self-Organization

Second reviewer:
Prof. Dr. G. Lube
Georg-August-University of Göttingen, Faculty of Mathematics
Institute for Numerical and Applied Mathematics

Abstract

In this master thesis spiral defect chaos in Rayleigh-Bénard convection of large aspect ratio cells is studied by means of numerical simulations on graphics cards. We use a decomposition of the velocity field into toroidal-poloidal fields for periodic domains to avoid solving explicitly the incompressibility condition and eliminate the pressure term from the Boussinesq equations. Applying a Galerkin method to expansions of the flow into Fourier series (horizontal plane) and Chandrasekhar polynomials (vertical direction) we are able to employ the so called pseudo-spectral simulation technique to solve the equations numerically. The basics of this simulation techniques (applying fast Fourier transformations) are presented as well as details. To simulate flow in bounded domains with no-slip boundary conditions in arbitrary direction we employ a penalization method which allows us to keep the simulation code unchanged (and therefore efficient) by introducing an additional penalization term. This technique allows us to simulate convection patterns in geometries by far more complex than those corresponding to periodic boundary conditions. The time-stepping is realized by an implicit-explicit splitting schema.
To achieve high performance this pseudo-spectral method (including penalization) is implemented on graphics cards. Therefore we present the basics of CUDA programming and some performance guidelines with respect to the implemented memory management and usage. Furthermore we have implemented a distributed pseudo-spectral code to simulate even larger systems, which might exceed the limits of one GPU's memory, on multiple GPUs. All implementations have been verified by checking all operations by comparison to results of a symbolic computing toolbox. These pseudo-spectral simulations have been used to simulate spiral defect chaos in large aspect ratio convection cells. Firstly we study the onset of spiral defect chaos as a function of aspect ratio and Rayleigh number. Secondly we consider some basic statistical properties and compare them to two-dimensional turbulent fluid flow. Finally in this master thesis we have shown that extensive fluid dynamic calculations of significant size can be done using relatively small resources taking advantage of GPU.

Contents

List of Figures

Introduction

A fluid layer between two plates, heated from below and cooled from above, is one of the best studied systems in fluid dynamics and the archetype of a pattern forming system. It is a strongly nonlinear, self-organizing system. It was first studied experimentally by the french physicist Henry Bénard in 1900 (see [1]). A first theoretical treatment of the problem is due to Rayleigh (see [2]). In honor to Bénard and Rayleigh this rather simple experimental setup is called Rayleigh-Bénard Convection.

But it is still of great interest to today's scientists and new effects, patterns, states are observed every year. This is caused by its self-interacting and strong nonlinear structure. Since the system is relatively easy accessible experimentally, numerically and theoretically it is a perfect playground for mathematicians, theoretical and experimental physicists. In particular, one can directly compare numerical simulations and experiments. It is also considered as a reference problem for the study of transition to chaos and turbulence in fluid dynamics. In our case a numerical approach will be presented which considers statistical properties of a specific state of Rayleigh-Bénard Convection.

Motivation and goals

The goal of this master thesis is to investigate a specific spatio-temporal chaotic state of Rayleigh-Bénard Convection called spiral defect chaos (SDC). Phenomenologically this state can be described in the following way: Convection rolls form large spirals which are rotating and advecting themselves through the fluid layer. They coarsen over time and might decay. We study spiral defect chaos with respect to the following questions: Does the horizontal velocity field of the fluid have any features comparable to two dimensional turbulence, i.e. is there anything like an inverse energy cascade, scaling of structure functions, scaling of energy spectrum?

For our study we use numerical simulations in which the equations are solved by means of a so-called pseudo-spectral method which offers high accuracy and high performance on simple domains. We can extend our simulations to flows with more complex boundaries without losses in performance by taking advantage of penalization methods.

As computational fluid dynamics needs large computing resources and times, high parallelism is required. A modern approach to high performance computing are graphics cards (GPU). Over the last twenty years they developed from small coprocessors to large computing units which offer a massive computing performance. Large graphics cards manufactures provide a way to use this computing performance through a simple programming interface. Treating the right problem with this approach may speed up simulation runtimes by a large factor compared to original single CPU or message passing interface (MPI) implementations. Therefore an implementation on a graphics card will be presented. This technology allows large parallelism on a single desktop computer. Furthermore we present a way to implement a simulation code which uses multiple graphics cards to run our simulation code on a distributed system. This is realized by taking advantage of a cooperation of message passing interface and the programming interface of our graphics cards.

Structure of the thesis

To achieve the goals of this thesis a wide range of topics has to be covered such as fluid dynamics, numerical analysis, practical implementation on GPUs. For better readability these topics are presented in separate chapters.

Chapter 1 considers some theoretical aspects. It presents the basic equations and some basic properties of them. Furthermore the so-called Boussinesq approximation will be introduced which simplifies our equations to incompressible ones. Of course this is only a valid approximation in a specific range. It introduces also the linear stability analysis which gives some insight why a fluid state is an unstable one. The last section of this chapter introduces a class of model equations which are widely-used as a model for pattern forming systems, in particular for Rayleigh-Bénard convection.

Chapter 2 is a chapter about the numerical schema. It introduces some numerical analysis like Galerkin methods and describes the idea of pseudo-spectral simulations. An expansion into orthogonal polynomials (Chandrasekhar polynomials) and Fourier series is presented. Using fast Fourier transformations (FFT) reduces the complexity which requires some insight into FFT algorithms. As we are considering non-stationary flows time-stepping methods are also of importance. A number of methods is presented in this section of the chapter. As comparisons to experimental results and simulations within "real" domains are of importance a so-called penalization method is presented which keeps our simulation code almost unchanged by inserting a new nonlinear operator into the original system of equations.

The following Chapter is separated into two larger parts. At first Section 3.1 discusses some features of GPU programming. It consists of a short introduction to graphics cards hardware architecture and an overview of best practices for programming. Knowledge about hardware is absolutely necessary to understand how to organize memory and code to achieve high computing performance. Section 3.2 considers some advanced implementation details and demonstrates the process of verification. It gives a detailed insight into data management and techniques which have been used to achieve the maximum performance. Furthermore an implementation for multiple graphics cards through message passing interface is presented.

In Chapter 4 we discuss the results obtained by our numerical simulations. We study statistical properties of SDC state and furthermore review code performance and accuracy. The last chapter gives a short summary and an outlook about future work and open problems.

Chapter 1

Theory of Rayleigh-Bénard Convection

This chapter describes the basic ideas of hydromagnetic instability and pattern formation in case of Rayleigh-Bénard convection. This rather simple experimental setup was first considered by Henry Bénard. It consists of a more or less thin layer of fluid which is heated from below and therefore the temperature gradient is adverse. This adverse temperature gradient leads to an expansion of the fluid at the bottom (which means that the density will be lower compared to the top). One might clearly guess that this top-heavy arrangement is becoming unstable.

On the other hand fluid's viscosity will dissipate energy as soon as the fluid is moving. Of course this physical process will inhibit the fluid's tendency to start flowing around. One might guess again that a hydrodynamic instability can only occur when the temperature gradient exceeds a certain limit. It is exactly this intuitive presumption which can be figured out by an experiment. Of course it is not that simple and there are more things than just the temperature gradient which are of importance to hydromagnetic instabilities. It will be shown later that the so called Rayleigh- and

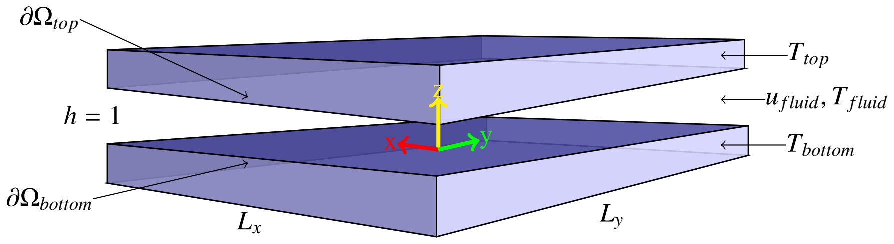

Figure 1.1: Geometry of the fluid layer heated from below. The vertical dimensions are L_x and L_y and the horizontal height is $h = 1$ length units (without loss of generality). The bottom platte is hotter than the one on top of the fluid layer. Both plates are assumed to be infinitely good conducting plates. The coordinate system of the layer is placed at $\frac{1}{2}h$.

Prandtl-Number,

$$\begin{aligned} Ra &= \frac{\alpha g \Delta T d^3}{\nu\kappa} \\ Pr &= \frac{\nu}{\kappa} \end{aligned}$$

which are both dimensionless characteristic numbers, control the instability behaviour. The coefficients have the following meaning:

α:	volume expansion coefficient
g:	gravitational acceleration
ΔT:	temperature difference, i.e.: $\Delta T = T_{bottom} - T_{top}$
d:	volume height
ν:	kinematic viscosity
κ:	thermal diffusivity

For the simulations we use a coordinate indicated as in Figure 1.1 and we will consider only newtonian fluids, that is the viscosity η is the proportional factor between the shear rate $\dot{\gamma}$ and the shear stress τ. With the fluid's density ρ we get the following definition of ν:

$$\begin{aligned} \eta &= \frac{\tau}{\dot{\gamma}} \\ \nu &= \frac{\eta}{\rho} \end{aligned}$$

The equations of movement for fluids are well known and in case of a Newtonian fluid they are called Navier-Stokes equations. Together with the heat equation (describing the evolution of temperature) this is the mathematical framework for the simulations described in Chapter 2. The next section gives a more detailed outline of the governing equations.

1.1 Governing equations

The partial differential equation system describing the motion of a fluid is well known. Reynold's transport theorem states that for $f : \Omega \times (0, \infty) \to \mathbb{R}$ (smooth enough), $v(x, t)$ a vector field and for every volume $V \in \Omega$ the following relation holds:

$$\frac{d}{dt} \int_{V(t)} f(x,t)dx = \int_{V(t)} \frac{\partial f}{\partial t}(x,t) + \nabla \cdot (fv)(x,t)dx$$

.

Taking advantage of mass conservation and using Reynold's theorem with $f(x, t) = \rho(x, t)$ and $v(x, t) = u(x, t)$ (where u denotes the velocity field) we get the so-called continuity equation:

$$\frac{\partial \rho}{\partial t} + div(\rho u) = 0 \qquad \text{in } \Omega \times (0, \infty) \tag{1.1}$$

Conservation of linear momentum gives the following three differential equations

$$\frac{\partial}{\partial t}(\rho u) + div(\rho u \otimes u) = \rho F + div\underline{T} \qquad \text{in } \Omega \times (0, \infty) \tag{1.2}$$

where $\underline{T}$ is the stress tensor and $F = (0, 0, -g)^T$ denotes an external force (in this case gravity). As $\underline{T}$ is a symmetric tensor of order 2 and is linear with respect to the velocity gradient, $\underline{T}$ can be decomposed as follows (where I denotes the unity tensor and η and λ are the fluid's viscosities)

$$\underline{T} = 2\eta\underline{D}(u) + \lambda div(u) - pI$$

where p is the normal pressure and $\underline{D}(u)$ is the so-called deformation tensor:

$$\underline{D}(u) = \frac{1}{2}(\nabla u + \nabla u^T)$$

The last conservation law applied is the conservation of energy e. Taking advantage of Reynold's theorem with $f = e$ leads to (ϵ is internal energy):

$$\frac{\partial e}{\partial t} + div(eu) = \rho F \cdot u + div(\underline{T} \cdot u) + div(\sigma) \qquad \text{in } \Omega \times (0, \infty)$$
$$e = \rho\epsilon + \frac{1}{2}\rho\|u\|^2$$

where σ is a vector field of the form:

$$\sigma = \zeta\nabla\epsilon$$

Assuming that the internal energy ϵ is proportional to the temperature T and applying some algebra to the equation of energy conservation will lead to an evolution equation for the temperature:

$$\rho\left(\frac{\partial(cT)}{\partial t} + u \cdot \nabla(cT)\right) = \eta\underline{D}(u) : \underline{D}(u) + \lambda(\nabla \cdot u)^2 - p\nabla \cdot u + \zeta\Delta(cT)$$

To close the partial differential equation system a fourth equation of state is needed, describing the relation between pressure, density and energy. We assume it to be of the following general form:

$$p = p(\rho, e)$$

To get a unique solution there are additional boundary conditions which have to be satisfied. As we consider non-stationary problems we have to consider an initial condition in Ω for $t = t_0$ and some boundary conditions on $\partial\Omega$ for all $t > t_0$. The most popular ones are Dirichlet, Neumann or Navier-Conditions (see [21], page 9). A more detailed insight will be given in Section 1.1.2.

These equations are the basic hydrodynamic equations. Of course they are complicated and hard to analyse because of their strong nonlinear structure. In many cases they can be simplified by taking advantage of some physical simplifications. One of them is the so-called incompressibility assumption which is the starting point for the so-called Boussinesq approximation which will be described in the following section.

1.1.1 Boussinesq Approximation

In many fluid mechanic problems fluid's velocity magnitude is much smaller than it's sound speed, i.e. the Mach number

$$M = \frac{V}{a} << 1$$

where a is the sound speed and V is the maximum fluid speed. In this case one can assume that the density is a constant. The continuity equation (1.1) reduces to:

$$\nabla \cdot u = 0 \tag{1.3}$$

Of course one has to clarify if this assumption will be correct for hydrodynamic problems. If we consider equation (1.2) a constant density will lead to a constant external force for each fluid lump. So modelling hydrodynamic problems by an incompressible fluid will remove any buoyancy. This is the starting point of Boussinesq's approximation which loosely speaking neglects compressibility except for the buoyancy term. Therefore we expand the density $\rho = \rho(T)$ in a Taylor series around a specific temperature T_0:

$$\begin{aligned} \rho &= \rho_0\Big(1 - \alpha(T - T_0)\Big) + O\Big(\theta^2\Big) \\ \theta &= T - T_0 \end{aligned}$$

Neglecting higher order terms in θ, the momentum equation (1.2) now becomes

$$\frac{\partial}{\partial t}(u) + div(u \otimes u) = e_z\Big(1 - \alpha(T - T_0)\Big) - \frac{\nabla p}{\rho_0} + \nu \Delta u$$

where e_z denotes the unity vector in z direction. So the complete partial differential equation system in case of a Boussinesq approximation in $\Omega \times (0, \infty)$ (neglecting friction as a heat source, i.e. $\nu \underline{D}(u) : \underline{D}(u) = 0$) is given by:

$$\frac{\partial}{\partial t}u + div(u \otimes u) - \nu \Delta u + \frac{\nabla p}{\rho_0} = -g e_z\Big(1 - \alpha(T - T_0)\Big) \tag{1.4}$$

$$\frac{\partial T}{\partial t} + u \cdot \nabla T - \kappa \Delta T = 0 \tag{1.5}$$

$$\nabla \cdot u = 0 \tag{1.6}$$

$$u = u_0 \qquad \text{in } \Omega \times \{0\} \tag{1.7}$$

$$T = T_0 \qquad \text{in } \Omega \times \{0\} \tag{1.8}$$

1.1.2 Boundary conditions

In case of no-slip boundaries (i.e.: rigid walls) we additionally get the following boundary conditions on $\partial\Omega \times (0, \infty)$:

$$\begin{aligned} & u_i = 0 \quad \forall i \in \{1; 2; 3\} \\ \overbrace{\Rightarrow}^{\text{equation (1.3)}} \quad & \frac{\partial}{\partial z} u_3 = 0 \end{aligned}$$

In case of no-surface-tension boundaries (i.e.: free surfaces) we get on $\partial\Omega \times (0, \infty)$:

$$\begin{aligned} \frac{\partial u_1}{\partial z} + \frac{\partial u_3}{\partial x} &= 0 \\ \frac{\partial u_2}{\partial z} + \frac{\partial u_3}{\partial y} &= 0 \\ u_3 &= 0 \\ \Rightarrow \frac{\partial u_1}{\partial z} = \frac{\partial u_2}{\partial z} &= 0 \\ \overbrace{\Rightarrow}^{\text{equation (1.3)}} \frac{\partial^2 u_3}{\partial z^2} &= 0 \end{aligned}$$

The boundary conditions for the heat equation are the following ones:

$$\begin{aligned} T &= T_{bottom} \qquad \text{on } \partial\Omega_{bottom} \times [0; \infty) \\ T &= T_{top} \qquad \text{on } \partial\Omega_{top} \times [0; \infty) \end{aligned}$$

One can easily verify that the following functions are stationary solutions of the Boussinesq approximation (1.4) in case of the simple semi-periodic cube given in Figure 1.2:

$$\tilde{u}(x, t) = 0; \tag{1.9}$$

$$\tilde{T}(x, t) = T_{bottom} - \frac{(T_{bottom} - T_{top})z}{d} \tag{1.10}$$

$$\tilde{p}(x, t) = z + \alpha \frac{(T_{bottom} - T_{top})z^2}{2d} \tag{1.11}$$

One can introduce perturbation functions u',T',p' around this state:

$$\begin{aligned} u &= \tilde{u} + u' \\ T &= \tilde{T} + T' \\ p &= \tilde{p} + p' \end{aligned}$$

Nondimensionalizing equations (1.4) and inserting the perturbation ansatz will give the following system of equations:

$$\frac{1}{Pr}\left(\frac{\partial}{\partial t}(u') + div(u' \otimes u')\right) - \Delta u' + \nabla p' = Rae_z T' \tag{1.12}$$

$$\frac{\partial T'}{\partial t} - u_3' + u' \cdot \nabla T' - \Delta T' = 0 \tag{1.13}$$

$$\nabla \cdot u' = 0 \tag{1.14}$$

These pertubation equations have the following boundary and initial conditions (in case of no-slip boundaries):

$$u' = 0 \qquad \text{on } \partial\Omega \times (0; \infty) \tag{1.15}$$

$$T' = 0 \qquad \text{on } \partial\Omega \times (0; \infty) \tag{1.16}$$

$$u' = u_0 \qquad \text{in } \Omega \times \{0\} \tag{1.17}$$

$$T' = T_0 - \tilde{T} \qquad \text{in } \Omega \times \{0\} \tag{1.18}$$

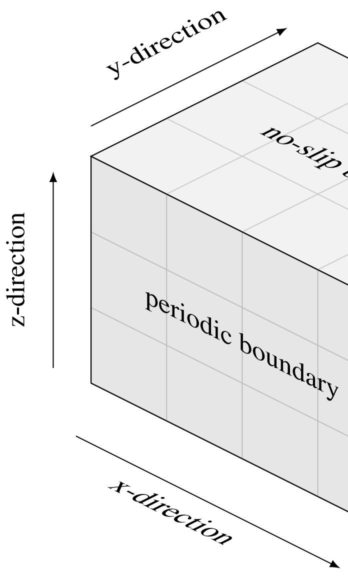

Figure 1.2: The simulation domain is a simple semi-periodic cube with different boundary conditions, i.e. no-slip condition in vertical direction and periodic boundary conditions in horizontal direction. Applying expansions into Fourier series and Chandrasekhar polynomials leads to a regular discretization grid.

1.2 Toroidal-Poloidal decomposition

For our numerical simulations we will consider a simple cube with periodic boundaries in the horizontal direction and rigid boundaries in the vertical direction. Let L_x and L_y be the length of the cube in x and y direction. A sketch of the simulation domain Ω is illustrated in Figure 1.2.

So the boundary conditions (1.15) and (1.16) will change to:

$$\begin{aligned} u'(x,y,0,t) &= u'(x,y,d,t) = 0 \\ T'(x,y,0,t) &= T'(x,y,d,t) = 0 \\ u'(0,y,z,t) &= u'(L_x,y,z,t) \\ u'(x,0,z,t) &= u'(x,L_y,z,t) \\ T'(0,y,z,t) &= T'(L_x,y,z,t) \\ T'(x,0,z,t) &= T'(x,L_y,z,t) \end{aligned}$$

Many numerical simulation schemata (e.g. Finite Volume Method) for incompressible, non-stationary Navier-Stokes equations suffer from problems to satisfy incompressibility conditions due to insufficient initial conditions for pressure and velocity or rounding errors (and therefore in many cases a correction step is needed). In case of a simple periodic cube, [15] and [16] have shown that there exists a decomposition of the velocity field u, such that

$$u = \delta f + \epsilon g + H$$

where $f : \mathbb{R}^3 \longrightarrow \mathbb{R}$ and $g : \mathbb{R}^3 \longrightarrow \mathbb{R}$ are mean free scalar fields and $H(z) : \mathbb{R} \longrightarrow (F(z), G(z), 0)^T$ with:

$$\begin{aligned} \delta &= (\partial_x\partial_z, \partial_y\partial_z, -\Delta_2)^T \\ \epsilon &= (\partial_y, -\partial_x, 0)^T \\ \Delta_2 &= \partial_x^2 + \partial_y^2 \end{aligned}$$

δf is the poloidal and ϵg the toroidal part. One can easily verify that the incompressibility condition $\nabla \cdot u = 0$ is implicitly correct by this decomposition. Furthermore the mean flow part (represented by F) is now separated from the rest of the velocity field. Since

$$\begin{aligned} \delta \cdot \delta f &= \Delta\Delta_2 f \\ \epsilon \cdot \epsilon g &= \Delta_2 g \\ \delta \cdot \epsilon g = \epsilon \cdot \delta f &= 0 \\ < \delta f >^{xy} = < \epsilon g >^{xy} &= 0 \end{aligned}$$

we get a new set of equations by applying $\delta, \epsilon, < \cdot >^{xy}$ to (1.12), where $< \cdot >^{xy}$ denotes the average over x-y-plane:

$$\frac{1}{Pr}\left(\frac{\partial}{\partial t}\Delta\Delta_2 f + \delta \cdot div(u \otimes u)\right) = -Ra\Delta_2 T' + \Delta^2\Delta_2 f \tag{1.19}$$

$$\frac{1}{Pr}\left(\frac{\partial}{\partial t}\Delta_2 g + \epsilon \cdot div(u \otimes u)\right) = \Delta\Delta_2 g \tag{1.20}$$

$$\frac{\partial T'}{\partial t} + \Delta_2 f + (u \cdot \nabla)T' = \Delta T' \tag{1.21}$$

$$\frac{1}{Pr}\left(\frac{\partial F}{\partial t} + \frac{\partial}{\partial z} < u_1 u_3 >^{xy}\right) = \frac{\partial^2}{\partial z^2}F \tag{1.22}$$

$$\frac{1}{Pr}\left(\frac{\partial G}{\partial t} + \frac{\partial}{\partial z} < u_2 u_3 >^{xy}\right) = \frac{\partial^2}{\partial z^2}G \tag{1.23}$$

By this decomposition the pressure field is also eliminated which simplifies our numerical schema. As shown in [15] this set of equations is equivalent to the incompressible Boussinesq equations. The boundary conditions are the following ones on $\partial\Omega_{bottom}$ and $\partial\Omega_{top}$:

$$f = \partial_z f = g = T' = F = G = 0 \tag{1.24}$$

1.3 Linear stability analysis and some bifurcation theory

So far the equations of motion and its boundary conditions are clear. In the introduction we have given several heuristic arguments suggesting that the simple solution of the Boussinesq problem, presented in (1.9) to (1.11), becomes unstable. We have also seen that the only control parameters are Rayleigh and Prandtl number. Thus, for a detailed description of instabilities we need to (i) specify the type of instability and (ii) determine the specific values of Prandtl and Rayleigh number which separate stability and instability.

Some answers can be given within the context of bifurcation theory which is not restricted to Rayleigh-Bénard Convection but can also give answers to other instability problems. First of all we consider a simple ordinary differential equation of the form

$$\frac{d}{dt}x = f(x)$$

with $x \in \mathbb{R}$ and $t \in \mathbb{R}$. For different initial/boundary conditions

$$x(t = t_0) = x_0$$

we will get another trajectory $x(t)$. If one assumes $x(t)$ to be the trajectory of a particle within a force field (somehow represented by f) one can ask if there is point in space $x_{equilibrium}$ such that for some different initial conditions a, b, c the trajectory of their particles tend to $x_{equilibrium}$ in the limit of infinite time:

$$x_{a,b,c}(t) \longrightarrow x_{equilibrium} \quad , t \longrightarrow \infty$$

But what will happen if we introduce some kind of a small perturbation δ to $x_a(t)$? Does $x_a + \delta \longrightarrow x_{equilibrium}$ still hold? To give an answer to this question we write an arbitrary trajectory $x(t)$ around the equilibrium state as:

$$x(t) = x_{equilibrium} + \eta(t)$$

The next step is to expand f around the equilibrium state x_0:

$$f(x) = f(x_0) + Df|_{x=x_{equilibrium}}\eta + O(\eta^2)$$

Assuming that higher order terms are not important since we consider an infinitesimal perturbation we get the following relation taking advantage of the fact that $x_{equilibrium}$ is an equilibrium state:

$$\frac{d}{dt}\eta = Df|_{x=x_{equilibrium}}\eta$$

This is a linear ordinary differential equation of order 1 and one might decompose $Df|_{x=x_0}$ into its eigenvalues, i.e. the differential equation can written in the following form

$$\frac{d}{dt}\eta = \lambda\eta$$

, if you assume without loss of generality that η is an eigenvector. Obviously the solution of this ordinary differential equation is of the form

$$\eta(t) = Ce^{\lambda t} \tag{1.25}$$

where C is a constant adapted to fulfill the boundary conditions. Obviously the following relation holds (where $\mathcal{R}(\lambda)$ denotes the real part of λ):

$$\eta(t) \longrightarrow 0 \text{ for } t \longrightarrow \infty \Leftrightarrow \mathcal{R}(\lambda) < 0$$

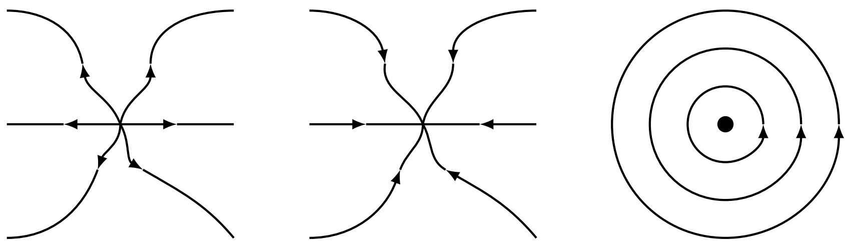

Figure 1.3: Sketch of three simple stability / instability types ([7]) in phase space. The first one is an unstable mode, i.e. $\mathcal{R}(\lambda) > 0$ for the linearization and its eigenvalues λ. Furthermore in this case $\mathcal{I}(\lambda) = 0$ ($\mathcal{I}(\lambda)$ denoting the imaginary part of λ) since there is no spiral around the linearization point. The second case shows a stable mode, i.e.: $\mathcal{R}(\lambda) < 0$ and $\mathcal{I}(\lambda) = 0$ for the linearization. The eigenvalues of the linearization around the third mode is pure imaginary ($\mathcal{R}(\lambda) = 0$ and $\mathcal{I}(\lambda) \neq 0$) and therefore its dynamic is periodic.

So the question of a stable equilibrium point $x_{equilibrium}$ reduces to the question if all eigenvalues of the Jacobian have negative real part in $x_{equilibrium}$.

In case of Rayleigh-Bénard Convection we have two different control parameters. Therefore the eigenvalues of the Jacobian of the linearized equation system will depend on these parameters and the question now turns into the following (assuming that we study the same fluid which fixes our Prandtl number): Does $\mathcal{R}(\lambda(R))$ (the real part of eigenvalues as a function of the Rayleigh number) have a zero crossing as a function of Ra?

To analyse the stability region of Rayleigh-Bénard Convection in detail we write equations (1.19) to (1.23) in the following way:

$$V = (\theta, f, g, F, G)^T$$
$$B\frac{\partial}{\partial t}V = N(V|V) + L(V)$$

L and B represent linear differential operators and N a bilinear one. So the linearized problem has the following form (, taking advantage of the fact that the linearized part of $N(V|V)$ is zero):

$$B\frac{\partial}{\partial t}V = L(V) \tag{1.26}$$

As the simulation domain is a periodic cube (in the horizontal directions) V is expandable as a Fourier series. To analyse the stability of the analytic solution of the Boussinesq problem given in equation (1.9) the perturbation V is written in the form (compare to equation (1.25)):

$$V(x, y, z, t) = V(q_x, q_y, z, t)e^{i(q_x,q_y)\cdot(x,y)^T}e^{\lambda t}$$

Inserting this ansatz into equation (1.26) leads to the following generalized eigenvalue problem

$$L \cdot V(q_x, q_y, z, t) = \lambda(q_x, q_y)B \cdot V(q_x, q_y, z, t)$$

where L and B are the matrix representations of the linear differential operators. The vector $V(q_x, q_y, z, t)$ now keeps the Fourier coefficients of $V(x, y, z, t)$ with respect to the

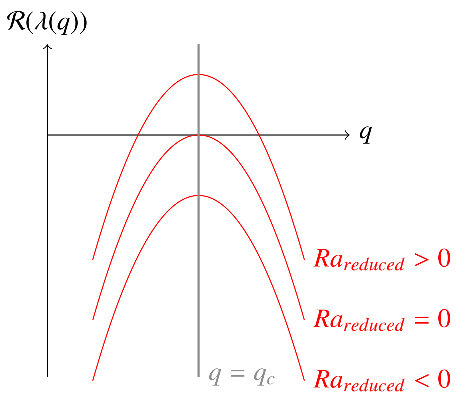

Figure 1.4: Growth rate of different modes as a function of the reduced Rayleigh number (see [4]) in case of the first stationary bifurcation. Notice that q_c is the first mode which is becoming unstable, because the real part of its eigenvalue has the first zero crossing for $Ra_{reduced} = 0$.

horizontal plane. As we are looking for the Rayleigh number at which the bifurcation from steady state takes place, $\lambda = 0$ will lead to:

$$L \cdot V(q_x, q_y, z, t) = 0$$

Solving this equation system with respect to z will lead to a series of exponential functions (a more detailed overview is given in [3] and [9]). This procedure leads to a critical Rayleigh number of $Ra_c \approx 1707.8$ (see [9], page 52). For higher values the steady state solution is an unstable one. The growth rate of a mode as a function of the reduced Rayleigh number

$$Ra_{reduced} = \frac{Ra - Ra_c}{Ra_c}$$

is plotted in Figure 1.4.

1.4 Further instabilities

A linear stability analysis shows: If the Rayleigh number is increased (and the Prandtl number is fixed by selecting a certain fluid), a bifurcation takes place at $Ra = Ra_c$ and the state given in (1.9) to (1.11) becomes unstable for $Ra > Ra_c$. The formerly (for $Ra < Ra_c$) unstable mode consisting of parallel rolls can now itself become stable under certain conditions. To see this one has to linearize the system of partial differential equations around this mode consisting of straight rolls. Using the same argumentation of the last section leads to a differential equation with periodic coefficients (because the state around which the equations are linearized is itself periodic). By Bloch's theorem (see [4]) we know that the solution is of the following form:

$$\eta = e^{\lambda(Q,q)t} e^{iQ\cdot x} \eta(z, t)$$

The mode of whose instability is investigated is represented by q and Q is the wavevector of the perturbation. As mentioned before it is again not clear whether the unstable mode Q might become itself unstable. If it is unstable, too, a roll pattern made of this mode Q will never be observed in an experiment or a numerical simulation. The stability region of the parallel rolls mode is often represented in a graph called Busse-Balloon. Figure 1.5 shows a cross section of the stability region at $Pr = 1.03$ with respect to perturbations of different wavelengths and for different reduced Rayleigh numbers.

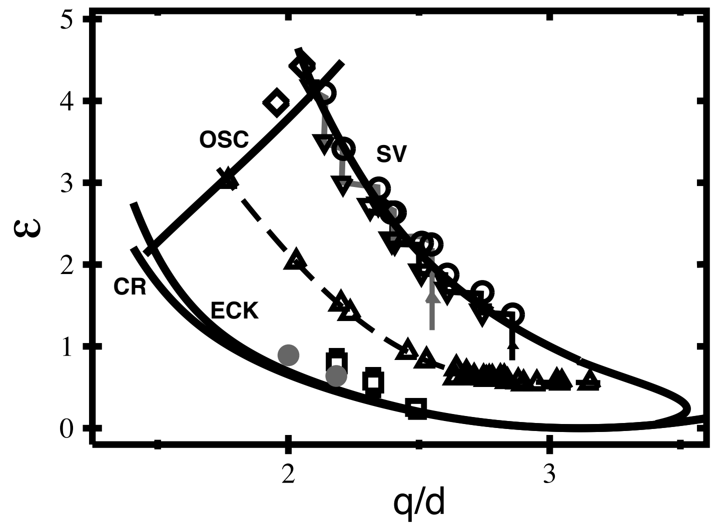

Figure 1.5: Stability regions of straight rolls for $Pr = 1.03$ (from [6]), ϵ denotes the reduced Rayleigh number, q the mode number and d the length of the cube. The theoretical results are represented by lines, while the experimental results are denoted by the geometrical shapes. The Figure shows stability boundaries for SV (skew-varicose), ECK (Eckhaus), OSC (oscillatory), CR (cross-roll) instabilities. The experimental results are obtained by Plapp (1997): Open circles, before SV-instability; upside-down triangles, after SV-instability; diamonds, onset of oscillatory instability; square, localized cross-roll instability; solid circles, numerically verified boundary of cross-roll instability; triangles, spiral defect chaos.

There are a number of different instabilities. Some instabilities for stripe states are the following ones (a more detailed overview can be found in [4], section 4.2):

Zig-Zag instability: stationary instability of stripe states with respect to long-wavelength, transverse modes (i.e.: small Q, traverse with respect to q)

Eckhaus instability: stationary instability of stripe states with respect to long-wavelength, longitudinal modes (i.e.: small Q, longitudinal with respect to q)

Cross-Roll instability: stationary instability of stripe states with respect to finite-wavelength, traverse modes (i.e.: finite Q, traverse with respect to q)

Skew-Varicose instability: stationary instability of stripe states with respect to long-wavelength, skew modes (i.e.: small Q, skew with respect to q)

Furthermore in large aspect ratio systems amplitude and phase might vary over space. It should be noticed, that this is not an instability of the stripe pattern but a modulation. In many cases so-called dislocations develop. Dislocations of (stripe) patterns are phase jumps of phase Φ of the complex amplitude. So dislocations can be found by a contour integral around the dislocation (where $\oint$ denotes a contour integral in a small enough area around the dislocation point):

$$\frac{1}{2\pi}\oint \nabla\Phi \cdot ds = \pm 1$$

A more detailed description of interactions of dislocations and its dynamics can be found in [4].

1.5 Spiral Defect Chaos (SDC)

As mentioned in the previous section parallel rolls states, which are the simplest patterns that are linearly stable at moderate Rayleigh numbers, can become unstable if one increases the Rayleigh or Prandtl number even farther away from equilibrium. Driven far enough from the onset some systems show states which are disordered in space and chaotic in time, i.e. display an irregular behaviour in time, although the system itself is deterministic (see [4]). One of this states in Rayleigh-Bénard Convection is the so-called spiral defect chaos (SDC). It occurs in large aspect ratio systems and shows convection rolls which advect through the fluid layer over time and form spirals of different sizes. These spirals rotate some times before they become unstable and build other local patterns.

Of course the question arises how we can understand the dynamics of these patterns and why can we see them? One qualitative way to explain why chaotic states arise is the following one: As we drive the system farther away from onset the previously stable patterns like rolls become unstable. In some cases they become unstable to states with a reduced symmetry. One example are roll patterns which become unstable to zig-zags as the control parameter is varied over some range. The former stripe pattern is unstable to these zig-zags and their amplitude starts to grow. After some time their amplitude is saturated by the nonlinearity. Nevertheless their amplitude can grow up to a value of the former stripe state's amplitude. So these instabilities might form new dislocations and patterns which themselves might become unstable and form new ones, too. Figure 1.6 shows the temperature field of a numerical simulation of a spiral defect chaotic state in a fluid layer with aspect ratio 50. It is worth mentioning that spiral defect chaotic states can develop even within the control parameter region in which rolls are stable. In many cases the question whether a roll state or a spiral defect chaotic state developes depends on the choice of initial conditions. A simple way to illustrate the behaviour of pattern forming systems are model equations. The most commonly used is the Swift-Hohenberg equation which can be extended to show spiral defect chaotic states, too.

1.6 Model equations of Swift-Hohenberg type

Equations of Swift-Hohenbeg type are model equations, which show simple pattern formation processes. They were first introduced by Swift and Hohenberg in 1977 (see [8]). Some simple patterns (stipe states and hexagons) created by numerical simulations in periodic domains are shown in Figure 1.7. Furthermore it can model many of the spatio-temporal pattern formation processes of Rayleigh-Bénard Convection by

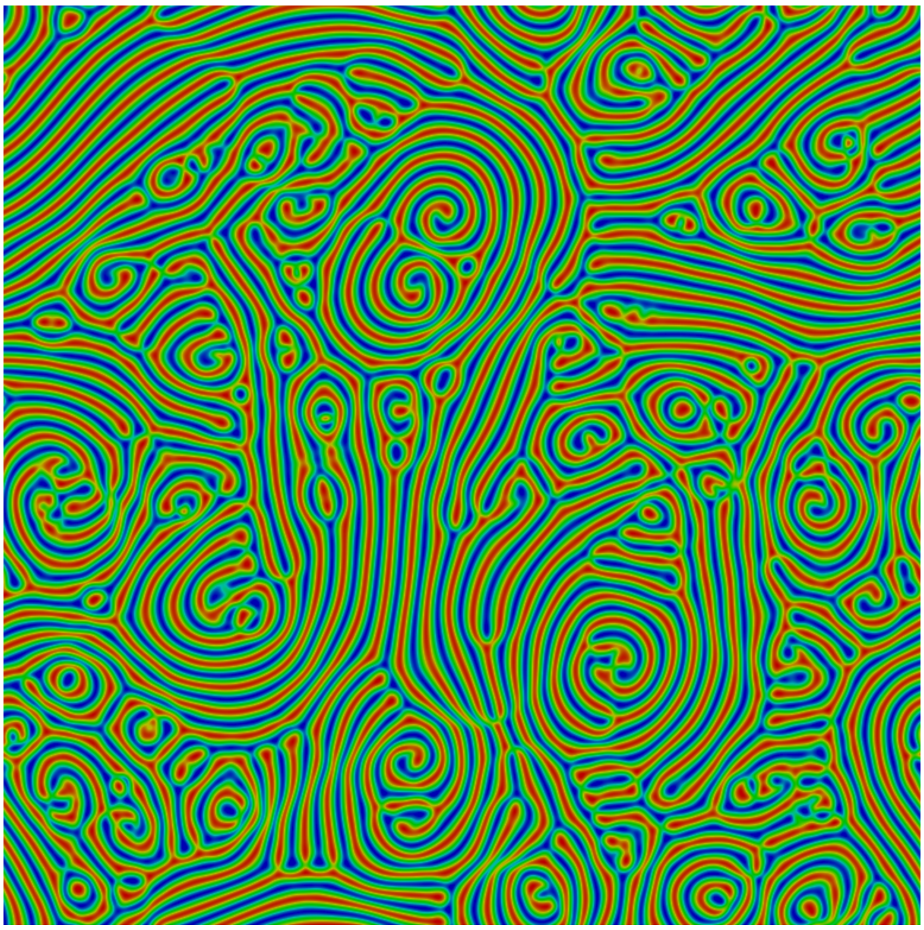

Figure 1.6: Numerical simulation of spiral defect chaotic state (Boussinesq equations) within a periodic domain for $\epsilon = 0.72$, $Pr = 0.96$. The aspect ratio is 50, in both horizontal directions 1024 Fourier coefficients were used for the simulation. The color represents the temperature of the fluid. Red represents warm fluid and blue cold one.

means of a simplified model equation. It is not derivable from the macroscopic Boussinesq or Navier-Stokes equations. Actually Swift and Hohenberg themselves introduced these equations in a different context. To describe more than simple patterns one has to extend them by additional terms.

The simplest form of this equation type is the following one:

$$\frac{\partial}{\partial t}u(x,y,t) = \epsilon u - (\Delta + 1)^2 u - u^3 \tag{1.27}$$

In this case $u(x,y,t)$ is a model function of the fluid's vertical velocity and is therefore not a function of the vertical coordinate z. In this case ϵ is the bifurcation parameter and controls the instability. One can easily show by the following ansatz for the perturbation

$$\eta(x,y,t) = Ae^{\lambda t}e^{i(q_x x + q_y y)}$$

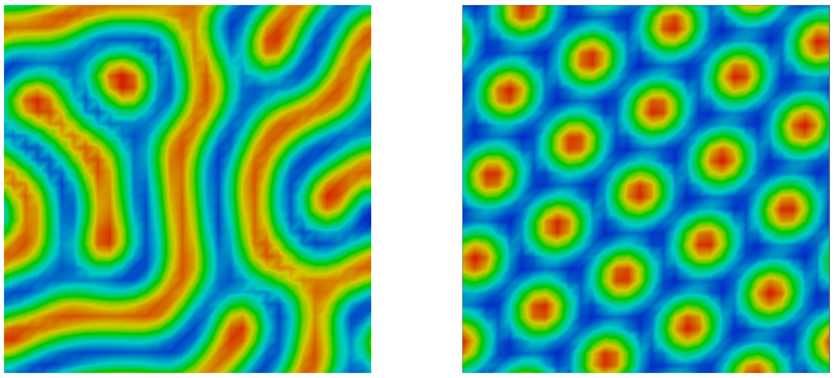

Figure 1.7: Two stationary patterns obtained by numerical simulations of the simplest Swift-Hohenberg equation in a periodic domain without mean flow. Since the Swift-Hohenberg equation has a potential function which never decreases in time, it can only create stationary patterns.

that the first mode ($||q||^2 = 1$) is becoming unstable for $\epsilon > 0$. This can be obtained by linearizing equation (1.27) around $u = 0$ and rewriting the linear differential operator in Fourier space, which is:

$$L(u) = -(1 - ||q||^2)^2 u$$

So this equation has the same initial bifurcation properties like the Boussinesq equations. Of course this simple equation can not describe the complete behaviour of a fluid layer heated from below in detail. One way to see this is to look for monotonic functionals. In case of periodic boundary conditions one may consider the following functional:

$$F(t) = \int\int -\frac{1}{2}\epsilon u^2 + \frac{1}{4}u^4 + \frac{1}{2}\Big((\Delta + 1)u\Big)^2 dxdy \tag{1.28}$$

Integration by parts and taking advantage of periodic boundary conditions will give:

$$\frac{dF}{dt} = -\int\int \left(\frac{\partial}{\partial t}u\right)^2 dxdy < 0$$

Therefore the function $F(t)$ can only decrease within a periodic domain over time and one reaches a steady state if the functional reaches its minimum. So equation (1.27) cannot describe spatio-temporal chaotic states. But it can describe simple states like stable convection rolls or hexagons.

To describe more than these basic patterns the Swift-Hohenberg model equations have to be extended by additional terms. They have to represent the so-called mean flow part advecting patterns through the fluid layer. One option is the following one:

$$\begin{aligned} \frac{\partial}{\partial t}u + V \cdot \nabla u &= \epsilon u - (\Delta + 1)^2 u - u^3 + \delta u^2 \\ V &= (\partial_y \zeta, -\partial_x \zeta) \\ \Delta \zeta &= c \cdot \hat{z} \cdot \nabla u \times \nabla(\Delta u) \end{aligned}$$

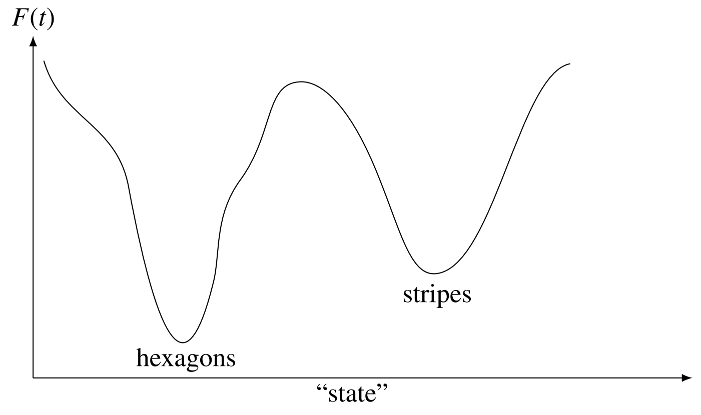

Figure 1.8: Shape of a functional given in equation (1.28) (see [4], page 182). There is some kind of a potential-barrier between stripes and hexagons. Therefore a potential system like the ordinary Swift-Hohenberg equation cannot describe non-potential, spatio-temporal chaotic states, which are not transient.

In this case $\hat{z}$ is a unit vector along the z-axes, ζ represents the stream function, V is the mean flow and c is a constant representing the strength of the mean flow. It is worth mentioning that the given equations are invariant with respect to the symmetry $u \rightarrow -u$ and rotations, as one would expect of the Boussinesq equations, too. The added mean flow term destroys the potential nature of the original Swift-Hohenberg equation (1.27). Therefore it can describe more complicated patterns and even spatio-temporal chaotic states. They also describe a pattern state which is visually comparable to spiral defect chaos. Figure 1.9 shows a spatio-temporal chaotic state similar to spiral defect chaos obtained by a numerical simulation of Swift-Hohenberg equations (including mean flow). The discrete equations for two-dimensional simulations of Swift-Hohenberg type equations in case of a Galerkin method for a periodic domain can be found in Appendix B.

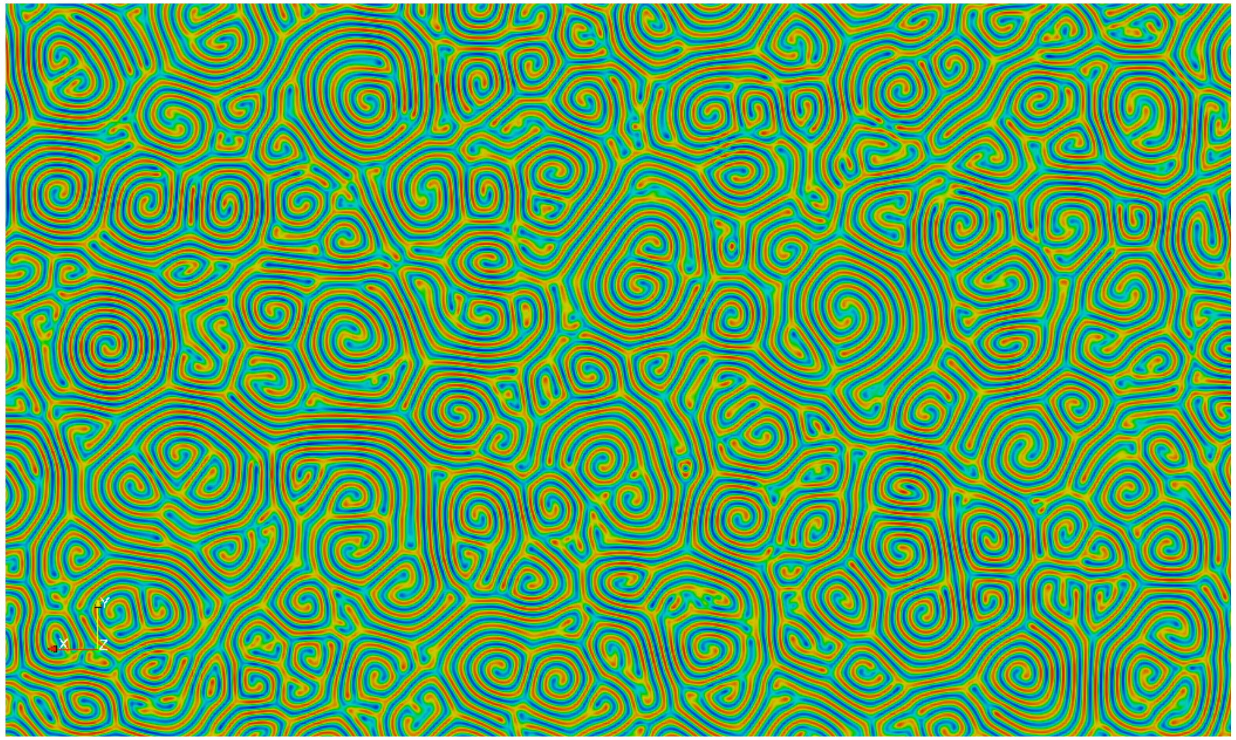

Figure 1.9: Numerical simulation of a spatio temporal chaotic state in case of Swift-Hohenberg equation with mean flow. The state obtained is similar to spiral defect chaos. This figure shows a section of a very large simulation (running also on graphics cards) with periodic boundaries. The complete set of discrete equations can be found in the appendix.

Chapter 2

Numerical methods for Rayleigh-Bénard Convection

Simulating fluid's motion has many faces. First of all one has to choose a method appropriate to the simulated physical problem. Common numerical methods like finite difference (FDM), finite volume (FVM), finite element methods (FEM) are applicable to complex geometries and are not easy to implement, especially on unstructured grids. Of course they differ from the mathematical perspective, as FVM and FEM will give weak solutions while FDM will not.

To take advantage of our simple periodic domain in the horizontal directions applying spectral methods is an appropriate choice. To describe the method in one sentence one might say, that spectral methods are Galerkin methods which use a representation of high order (in many cases high order (complex-) polynomials) on a structured grid. By choosing a high order scheme you will achieve (in many cases) a very high convergence rate (which is in theory higher than any polynomial in $1/N$ where N denotes the approximation oder). Compared to other methods like FEM this will lead to an equation system which is not sparse. So the question arises whether spectral methods are efficient ones? The answer is yes, due to the fact that there exist fast transformations for many ansatz functions on simple/periodic domains which reduce the complexity of the simulation scheme. The following section gives an mathematical motivation of our numerical simulation schema by introducing weak formulations. Readers who are not interested in this mathematical background can skip this section and continue reading with section 2.1.2 which describes the numerical method (pseudo-spectral simulations) in detail (i.e. the discrete equations and how to calculate the nonlinear operator efficiently).

2.1 Theory and Galerkin Method

Galerkin Methods are heavily used in numerical analysis and theory of partial differential equations. It is a variational method and the basic idea for many techniques to solve partial differential equations numerically (e.g. finite volume or finite element methods). To understand them we have to introduce the so-called weak formulation which extends the differentiation respectively the definition of a solution in a weaker sense.

2.1.1 Weak formulation

In many cases it is well known that partial differential equations do not have a strong solution. Especially in the cases of continuous but non-differentiable boundary conditions obviously strong solutions cannot exist. So the idea of weak solutions is to extend differentiation to a so-called weak form such that it includes the strong differentiation and specifies a well defined solution theory. This means that we are looking for a Hilbert space which is defined in a right way to define differentiation in a weak sense. Integration by parts is well known from basic analysis. In case of a function $u \in C^1(\Omega)$ and $\phi \in C_0^\infty(\Omega)$ (where $C_0^\infty(\Omega) = \{f \in C^\infty | \quad f|_{\partial\Omega} = 0\}$) this means:

$$\int_\Omega \frac{\partial u}{\partial x_i} \phi dx = - \int_\Omega u \frac{\partial \phi}{\partial x_i} dx$$

So it turns out to use this property to generalize the strong derivative to a weaker form:

Definition 1. *Let $u \in L^1_{loc}\Omega$ and $g \in L^1_{loc}(\Omega)$ such that they satisfy*

$$\int_\Omega g\phi dx = - \int_\Omega u \frac{\partial \phi}{\partial x_i} dx$$

for all $\phi \in C_0^\infty(\Omega)$ then g is the weak derivative of u with respect to x_i. Using the scalar product of $L^2(\Omega)$ one may write in a short form:

$$(g, \phi)_{L^2} = -(u, \partial_{x_i}\phi)_{L^2}$$

By this definition we introduce the following family of function spaces, which are called Sobolev spaces (f^α denoting the weak derivative of f of order α)

$$W^{k,p}(\Omega) = \{f \in L^p(\Omega) | f^\alpha \in L^p(\Omega) \forall \alpha \in \mathbb{N}^d : ||\alpha|| < k\}$$

where $||\alpha|| = \alpha_1 + \cdots + \alpha_d$ and $f : \Omega \subset \mathbb{R}^d \to \mathbb{R}$. Together with the following definition $W^{k,p}$ is becoming a normed function space:

$$||u||_{W^{k,p}} = \begin{cases} \left(\sum_{|\alpha| \leq k} ||u||^p_{L^p} \right)^p & \text{if } 1 \leq p < \infty \\ \sum_{|\alpha| \leq k} ess \sup_\Omega |u^\alpha| & \text{if } p = \infty \end{cases}$$

To shorten notations we introduce the following important function space:

$$H^k(\Omega) = W^{k,2}(\Omega)$$

It is a well known fact, that (see [11], page 249):

For $1 \leq p \leq \infty$ $W^{k,p}$ is a Banach space and $H^k(\Omega)$ is a Hilbert space with the following scalar product.

$$< u, v >_{H^k(\Omega)} = \sum_{|\alpha| \leq k} < D^\alpha u, D^\alpha v >_{L^2(\Omega)}$$

To fulfill boundary conditions we have to introduce another function space in the following way

$$W_0^{k,p}(\Omega) = \overline{C_0^\infty(\Omega)}^{||\cdot||_{W^{k,p}}}$$

and of course accordingly:

$$H_0^k\Omega = W_0^{k,2}(\Omega)$$

Of course the definition of $W^{k,p}(\Omega)$ is the right one to speak about boundary values. A priori functions $f \in L^p(\Omega)$ are only defined almost everywhere in $\Omega \subset \mathbb{R}^n$. As $\partial\Omega$ is only a $n-1$ dimensional manifold with Lebesgue measure zero, speaking about boundary values is not defined. Luckily the so-called trace operator T can be defined such that (see [11], page 258)

$$T : W^{k,p}(\Omega) \to L^p(\partial\Omega)$$

and for all $u \in W^{k,p}(\Omega)$:

$$||T(u)||_{L^p(\partial\Omega)} \leq C||u||_{W^{k,p}(\Omega)}$$

Furthermore it can be shown:

$$u \in W_0^{k,p}(\Omega) \Leftrightarrow T(u) = 0 \text{ on } \partial\Omega$$

To study time-dependent partial differential equations an analogous technique with respect to time derivatives has to be introduced. This can be done by introducing the so-called Bochner integral which is an integral generalized to Banach space functions.

Definition 2. *The function space $L^p(0;T;X)$, where X is a Banach space, consists of all measurable function $u : [0;T] \to X$ where*

$$||u||_{L^p(0;T;X)} = \begin{cases} \left(\int_0^T ||u(t)||_X^p dt\right)^{\frac{1}{p}} & \text{if } 1 \leq p < \infty \\ ess\sup_{0\leq t\leq T} ||u(t)||_X & \text{if } p = \infty \end{cases}$$

.

In case of Boussinesq equations (1.12) one can introduce the following weak formulation for the momentum equation after taking advantage of integration by parts and periodic boundary conditions

$$\begin{aligned} -\frac{1}{Pr}\int_0^T < u, \frac{\partial v}{\partial t} >_{L^2} dt + \int_0^T div(u\otimes u)vdt + \int_0^T \nabla u : \nabla v dt - \int_0^T pdiv(v)dt \\ = Ra\int_0^T e_z T v dt \\ 0 = \int_0^T q\cdot div(u)dt \end{aligned}$$

where the equations have to be fulfilled for $p \in L_0^2(\Omega)$ and for all $q \in L_0^2(\Omega)$.

2.1.2 Galerkin Method

One way to treat partial differential equations numerically are Galerkin methods. The idea is to find a set of functions which approximates our solution and test function space with high accuracy (in many cases one can find a dense subset). So each (test-) function can be represented as a series of these functions. To obtain a numerical schema which can be implemented, we truncate this series in an appropriate way and obtain a discrete set of equations. So for a given differential equation

$$\begin{aligned} L(u)(x) &= f(x) \text{ in } \Omega \\ u(x) &= u_0(x) \text{ in } \partial\Omega \end{aligned}$$

one expands the solution $u(x) \in X$ into a finite series:

$$u(x) = \sum_{i=0}^{N} u_i \phi_i(x)$$

In many cases the following relation holds

$$X = \overline{\{\phi_0, \phi_1, \phi_2, \cdots\}}^{\|\cdot\|_X} \tag{2.1}$$

, which means that $\{\phi_0, \phi_1, \phi_2, \cdots\}$ is a dense subset of X. Introducing the residual

$$r(x) = L(u)(x) - f(x)$$

the goal is to minimize $r(x)$. Taking advantage of the fact that X is a Hilbert space, the minimal residual method tries to minimize the residual by introducing a set of weighting functions

$$W = \{w_0, w_1, w_2, \cdots\}$$

and the condition

$$\begin{aligned} < w_1(x), r(x) >_X &= 0 \\ &\vdots \\ < w_N(x), r(x) >_X &= 0 \end{aligned}$$

The so-called Galerkin method is obtained by projecting onto ϕ_i, which means:

$$\phi_i(x) = w_i(x) \quad \forall i \in \{1, \cdots, N\}$$

This gives an equation system or a system of ordinary differential equations to determine the coefficients u_i.

Expansion into Fourier series

In some cases relation (2.1) is extended by the following condition:

$$\begin{aligned} < \phi_i, \phi_j >_x &= \delta_{ij} \\ \|\phi_i\|_X &= 1 \end{aligned}$$

In such a case a function $f \in X$ is represented by a so-called generalized Fourier series in the following way:

$$f(x) = \sum_{i}^{\infty} < f, \phi_i >_X \phi_i(x)$$

In case of periodic boundary conditions a complete orthonormal system is given by the following set with the given definition of a scalar product:

$$C^0_L = \{f : [0; L] \to \mathbb{R} | f(0) = f(L), f \in C^0\} = \overline{\{e^{\frac{2\pi ikx}{L}} | k \in \mathbb{N}\}}$$
$$< f, g >_{C^0_L} = \frac{1}{L} \int_0^L f(x)\overline{g}(x)dx$$

So on periodic domains it is obviously a nice way to expand a solution into a Fourier series of the form

$$f = \sum_{k=-\infty}^{\infty} \hat{f}_k e^{\frac{2\pi ikx}{L}}$$

where $\hat{f}_k$ are so-called Fourier coefficients. One can show that if a function f is in C^p_L the truncated Fourier series up to order N

$$\tilde{f} = \sum_{k=-N}^{N} \hat{f}_k e^{\frac{2\pi ikx}{L}}$$

converges to f asymptotically like (see [20]):

$$||f - \tilde{f}||_\infty = O(N^{1-p})$$

So for a smooth function this means that a truncated Fourier series converges faster than any polynomial in $\frac{1}{N}$. Additionally there are some simple calculation rules regarding to differentiation:

$$\frac{\partial f}{\partial x} = \sum_{k=-\infty}^{\infty} \frac{2\pi ik}{L} \hat{f}_k e^{\frac{2\pi ikx}{L}}$$
$$\frac{\partial^2 f}{\partial x^2} = \sum_{k=-\infty}^{\infty} \frac{-(2\pi k)^2}{L^2} \hat{f}_k e^{\frac{ikx}{L}}$$

Chandrasekhar Polynomials

As presented in the previous section it is reasonable to expand periodic functions into Fourier series. This is the case for the horizontal directions. Of course the Boussinesq equations (1.19) to (1.23) depend on the z-coordinate, too. So the question arises how to expand the vertical dependence. Of course we have to keep in mind, that the different functions in equations (1.19) to (1.23) have to satisfy different boundary conditions (see (1.24)). In case of f one has to look for an orthogonal set of functions $\psi_i(z)$ satisfying:

$$\psi_i(-\frac{1}{2}) = \psi_i(\frac{1}{2}) = 0 \tag{2.2}$$
$$\frac{\partial}{\partial z}\psi_i(-\frac{1}{2}) = \frac{\partial}{\partial z}\psi_i(-\frac{1}{2}) = 0 \tag{2.3}$$
$$< \psi_i, \psi_j >= \delta_{ij} \tag{2.4}$$

The solution of this equation system is well known. The set of orthonormal functions is given by Chandrasekhar polynomials:

$$\psi_k(z) = \begin{cases} \frac{cosh(\lambda_k z)}{cosh(\lambda_k \frac{1}{2})} - \frac{cosh(i\lambda_k z)}{cosh(i\lambda_k \frac{1}{2})} & \text{if } k \text{ is odd} \\ \frac{sinh(\lambda_k z)}{sinh(\lambda_k \frac{1}{2})} - \frac{sinh(i\lambda_k z)}{sinh(i\lambda_k \frac{1}{2})} & \text{if } k \text{ is even} \end{cases}$$

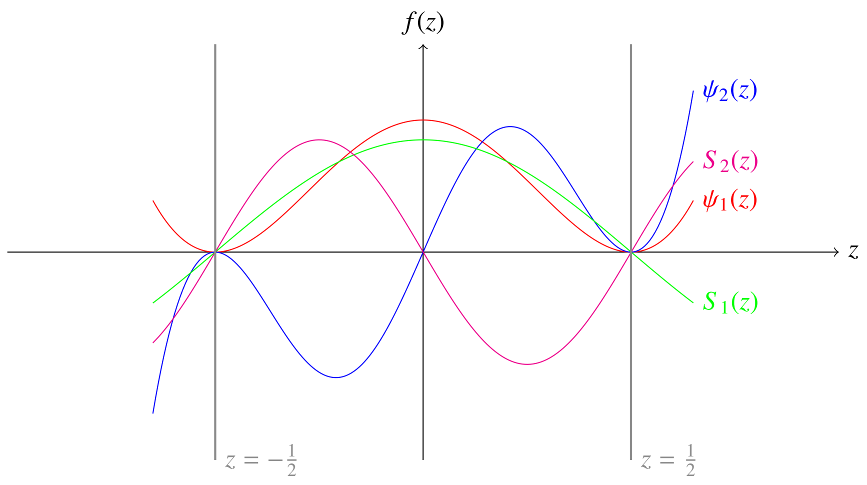

Figure 2.1: Plot of the vertical ansatz functions for T', f, g, F, G. ψ_1 and ψ_2 are used for f to fulfill the boundary conditions (for each ansatz function). This choice of ansatz functions leads to a system of ordinary differential equations without any additional algebraic condition.

It is easy to verify that these functions satisfy the requirements and additionally the following simple relation holds true:

$$\frac{\partial^4}{\partial z^4}\psi_k(z) = \lambda_k^4 \psi_k(z)$$

Using a bit more functional analysis one can even show that these polynomials constitute a dense subset in an appropriate subspace of functions satisfying (2.2) to (2.4).

2.1.3 Penalization methods

The possibility of direct comparison of theoretical, numerical and theoretical results is one of the key features of Rayleigh-Bénard Convection. Therefore, it is legitimate to ask whether simulations in a periodic domain are realistic enough to be compared to experimental situations. The main reason why numerical simulations in periodic domains are so extensively used is the efficiency they provide for calculating the nonlinear part by means of fast Fourier transformations. Of course, fast transformation methods also exist for a number of sets of orthogonal functions on other simple domains such as cylinders or spheres, such as Tschebychev polynomials on spheres. This approach is well known and has been in use for a long time (e.g. see [12]). The main disadvantage is the necessity to change the set of ansatz functions every time the geometry of the problem is changed. More complex geometries are not accessible at all, as neither the set of "right" ansatz functions nor the corresponding fast transformation method is known.

To circumvent these problems we apply a so-called penalization method, which is a method that allows you to simulate more complex geometries without the need to change large parts of your simulation code. As it is still a spectral code with Fourier series expansions efficiency is guaranteed. The idea is to introduce new terms into the Navier-Stokes and heat equation which forces the simulation result to have the correct

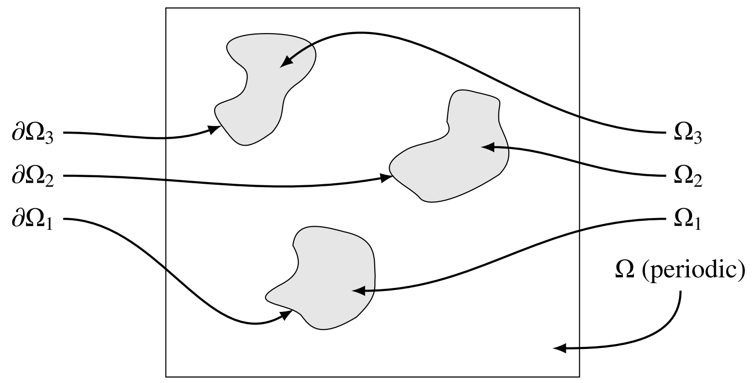

Figure 2.2: Sketch of the simulation domain Ω seen from above. To realize no-slip boundary conditions on $\partial\Omega = \partial\Omega_1 \cup \partial\Omega_2 \cup \partial\Omega_3$ we introduce volume penalizations on $\overline{\Omega_1 \cup \Omega_2 \cup \Omega_3}$ as a function of the penalization parameter η which forces the simulation result to fulfill the boundary conditions as $\eta \to 0$. So no-slip boundary conditions do not affect the choice of ansatz functions and the numerical simulation schema as Ω is still periodic.

boundary conditions. Of course some analysis has to clarify that the simulation result is more or less correct.

If one wants to simulate the Rayleigh-Bénard problem within a bounded open domain $\Omega \subset [0; L_x] \times [0; L_y] \times [-0.5; 0.5]$ (with rigid boundaries on top and bottom) which is non periodic in the horizontal direction one has to replace the periodic boundary conditions by no-slip conditions on the boundary $\partial\Omega$. First we define the complement of Ω within the periodic box:

$$\Omega^c = \overline{[0; L_x] \times [0; L_y] \times [-0.5; 0.5] \setminus \Omega} \tag{2.5}$$

One of the most frequently used penalization method is the L^2-penalization in which we replace momentum equations (1.12) by:

$$\frac{1}{Pr}\left(\frac{\partial}{\partial t}(u') + div(u' \otimes u')\right) - \Delta u' + \nabla p' = Rae_z T' - \frac{1}{\eta}\mathbf{1}_{\Omega^c} u' \tag{2.6}$$

$$\frac{\partial T'}{\partial t} - u'_3 + u' \cdot \nabla T' - \Delta T' = -\frac{1}{\eta}\mathbf{1}_{\Omega^c} T' \tag{2.7}$$

$$\nabla \cdot u' = 0 \tag{2.8}$$

where $\mathbf{1}_{\Omega^c}$ represents the following characteristic function

$$\mathbf{1}_{\Omega^c} = \begin{cases} 1 & \text{if } x \notin \Omega \\ 0 & \text{if } x \in \Omega \end{cases}$$

and $\eta \in \mathbb{R}^+$ and we want to satisfy the following boundary conditions by this penalization:

$$\begin{aligned} u' &= 0 \quad \forall (x,t) \in \partial\Omega \times [0; T] \\ T' &= 0 \quad \forall (x,t) \in \partial\Omega \times [0; T] \end{aligned}$$

Convergence results

The following results show some basic convergence results as $\eta \to 0$ and are presented in [13]. In case of a two-dimensional flow Angot et al. even have shown some rigorous convergence results and error estimators.

First we want to consider again the following penalized formulation of the incompressible Navier-Stokes equation:

$$\frac{1}{Pr}\left(\frac{\partial}{\partial t}(u'_\eta) + div(u'_\eta \otimes u'_\eta)\right) - \Delta u'_\eta + \nabla p'_\eta = Rae_z T' - \frac{1}{\eta}\mathbf{1}_{\Omega^c} u'_\eta$$
$$\nabla \cdot u'_\eta = 0$$

By expanding $u'_\eta = u + \eta\tilde{u}$ and $p'_\eta = p + \eta\tilde{p}$ and some algebraic manipulations (for details see [13], page 500) one can show, that (where Ω^c denotes again the penalization volume):

$$\mathbf{1}_{\Omega^c} u = 0 \tag{2.9}$$
$$\tilde{u}|_{\Omega^c} + \nabla p'|_{\Omega^c} = 0 \tag{2.10}$$
$$\Delta p'|_{\Omega^c} = 0 \tag{2.11}$$
$$\nabla \cdot \tilde{u}|_{\Omega^c} = 0 \tag{2.12}$$

By equation (2.10) one can verify that $\tilde{u}$ satisfies a Darcy-type law in the penalization volume Ω^c. The pressure satisfies a Neumann condition (see equation (2.11)). Therefore the fluid flow inside the penalization medium follows the flow through a porous medium.

Furthermore Angot et al. ([13]) have shown in case of a two-dimensional flow that the sequence $(u'_\eta)_\eta$, as η tends to zero, converges to a limit u' with

$$u'|_{\Omega^c} = 0$$

and the limit u' is the unique solution of the (unpenalized) Navier-Stokes equations in Ω. Additionally there exists a weak limit in the following way

$$\frac{1}{\eta}\mathbf{1}_{\Omega^c} u'_\eta \rightharpoonup \breve{u} \text{ in } W$$

where $W = \{\phi \in L^2(0;T;H^1_0(\Omega)); \partial_t\phi \in L^2(0;T;H^1_0(\Omega)); \phi(T) = 0\}$ (see [13]). This means that u'_η is more or less unique in Ω^c and not arbitrary in the limit $\eta \to 0$.

This simulation technique gives access to complex geometries (without any z-dependence) and in our cases $\eta = 10^{-2}$ is sufficient, to achieve convergence towards no-slip boundaries. Figure 2.4 shows some simulation results within a bounded cube and no-slip boundary conditions on all boundaries of the cube. For this simulation the last three points of the grid (in x and y direction) where penalized.

2.2 The Pseudo-Spectral Method

Employing Galerkin Methods for simulations requires to extract a finite number of equations that can be implemented in software. Therefore this section describes the basic discrete equations obtained by the Boussinesq approximation and the poloidal-toroidal decomposition. Furthermore we consider the penalization methods and show

how to get the discrete penalizations. The basic idea is to take a finite dimensional subspace of our solution space. This means representing the solutions of the system of partial differential equations by truncated Fourier series. We will see in Section 2.2.1 that changing between physical and Fourier space allows us to calculate the nonlinear operators efficiently. Since we change between physical and Fourier space and we are working with truncated Fourier series we have to make sure that we get rid of modes with a wavenumber higher than the highest wavenumber in our truncated Fourier series. This effect is called dealiasing and is described in Section 2.2.2.

2.2.1 Discrete equations

To obtain the set of discrete equations we expand all functions as a tensor product representation, i.e.:

$$T'(x,y,z,t) = \sum_{k_3=1}^{N} \sum_{k_2=-N_1}^{N_1} \sum_{k_1=-N_2}^{N_2} \hat{T}'_{k_1,k_2,k_3}(t) e^{\frac{2\pi i k_1 x}{Lx}} e^{\frac{2\pi i k_2 y}{Ly}} sin(k_3\pi(z+0.5)) \tag{2.13}$$

$$f(x,y,z,t) = \sum_{k_3=1}^{N} \sum_{k_2=-N_1}^{N_1} \sum_{k_1=-N_2}^{N_2} \hat{f}_{k_1,k_2,k_3}(t) e^{\frac{2\pi i k_1 x}{Lx}} e^{\frac{2\pi i k_2 y}{Ly}} \psi_{k_3}(z) \tag{2.14}$$

$$g(x,y,z,t) = \sum_{k_3=1}^{N} \sum_{k_2=-N_1}^{N_1} \sum_{k_1=-N_2}^{N_2} \hat{g}_{k_1,k_2,k_3}(t) e^{\frac{2\pi i k_1 x}{Lx}} e^{\frac{2\pi i k_2 y}{Ly}} sin(k_3\pi(z+0.5)) \tag{2.15}$$

$$F(x,y,z,t) = \sum_{k_3=1}^{N} \hat{F}_{k_3}(t) sin(k_3\pi(z+0.5)) \tag{2.16}$$

$$G(x,y,z,t) = \sum_{k_3=1}^{N} \hat{G}_{k_3}(t) sin(k_3\pi(z+0.5)) \tag{2.17}$$

The choice of vertical ansatz functions leads to an implicit satisfaction of our boundary conditions. To get the full set of equations one has to insert these series expansions into equations (1.19) to (1.23).

To abbreviate notations within the nonlinear operators we introduce some temporary fields $u_i^j(x,y,k_3,t)$ and $T_1(k_3)$ to describe the velocity field in the following way (with $S_n(z) = sin(n\pi(z+0.5))$):

$$\begin{aligned}
u_1 &= \sum_{k_3=1}^{N} \partial_z \psi_{k_3}(z) u_1^1(x,y,k_3,t) + S_{k_3}(z) u_1^2(x,y,k_3,t) \\
u_2 &= \sum_{k_3=1}^{N} \partial_z \psi_{k_3}(z) u_2^1(x,y,k_3,t) + S_{k_3}(z) u_2^2(x,y,k_3,t) \\
u_3 &= \sum_{k_3=1}^{N} \psi_{k_3}(z) u_3^1(x,y,k_3,t) \\
T' &= \sum_{k_3=1}^{N} S_{k_3}(z) T_1(x,y,k_3,t)
\end{aligned}$$

By definition of poloidal-toroidal decomposition these temporary variables are given by:

$$
\begin{aligned}
u_1^1(x,y,k_3,t) &= \sum_{k_2=-N_1}^{N_1} \sum_{k_1=-N_2}^{N_2} \frac{2\pi i k_1}{L_x} \hat{f}_{k_1,k_2,k_3}(t) e^{\frac{2\pi i k_1 x}{L_x}} e^{\frac{2\pi i k_2 y}{L_y}} \\
u_2^1(x,y,k_3,t) &= \sum_{k_2=-N_1}^{N_1} \sum_{k_1=-N_2}^{N_2} \frac{2\pi i k_2}{L_y} \hat{f}_{k_1,k_2,k_3}(t) e^{\frac{2\pi i k_1 x}{L_x}} e^{\frac{2\pi i k_2 y}{L_y}} \\
u_3^1(x,y,k_3,t) &= \sum_{k_2=-N_1}^{N_1} \sum_{k_1=-N_2}^{N_2} \left(\left(\frac{2\pi i k_1}{L_x}\right)^2 + \left(\frac{2\pi i k_2}{L_y}\right)^2 \right) \hat{f}_{k_1,k_2,k_3}(t) e^{\frac{2\pi i k_1 x}{L_x}} e^{\frac{2\pi i k_2 y}{L_y}} \\
u_1^2(x,y,k_3,t) &= \sum_{k_2=-N_1}^{N_1} \sum_{k_1=-N_2}^{N_2} \frac{2\pi i k_2}{L_y} \hat{g}_{k_1,k_2,k_3}(t) e^{\frac{2\pi i k_1 x}{L_x}} e^{\frac{2\pi i k_2 y}{L_y}} + \delta_{k_1,k_2} \hat{F}_{k_3} \\
u_2^2(x,y,k_3,t) &= \sum_{k_2=-N_1}^{N_1} \sum_{k_1=-N_2}^{N_2} -\frac{2\pi i k_1}{L_x} \hat{g}_{k_1,k_2,k_3}(t) e^{\frac{2\pi i k_1 x}{L_x}} e^{\frac{2\pi i k_2 y}{L_y}} + \delta_{k_1,k_2} \hat{G}_{k_3}
\end{aligned}
$$

By this variables the nonlinear part of the momentum equation becomes:

$$
(u \cdot \nabla) u = div(u \otimes u) = \begin{pmatrix} \partial_x(u_1 u_1) + \partial_y(u_2 u_1) + \partial_z(u_3 u_1) \\ \partial_x(u_1 u_2) + \partial_y(u_2 u_2) + \partial_z(u_3 u_2) \\ \partial_x(u_1 u_3) + \partial_y(u_2 u_3) + \partial_z(u_3 u_3) \end{pmatrix}
$$

To calculate the projections onto the horizontal and vertical functions one has to calculate ($\partial_z \psi_i = \psi_i'$):

- Projections of the linear operators
- Projections of the nonlinear operators

First we consider the projections of the linear operators. The linear operators are given by:

$$
\begin{aligned}
\frac{\partial}{\partial t} T' &= \Delta T' - \Delta_2 f \\
\frac{1}{Pr} \frac{\partial}{\partial t} \Delta \Delta_2 f &= \Delta^2 \Delta_2 f - Ra \Delta_2 T' \\
\frac{1}{Pr} \frac{\partial}{\partial t} \Delta_2 g &= \Delta \Delta_2 g \\
\frac{1}{Pr} \frac{\partial}{\partial t} F &= \frac{\partial^2}{\partial z^2} F \\
\frac{1}{Pr} \frac{\partial}{\partial t} G &= \frac{\partial^2}{\partial z^2} G
\end{aligned}
$$

Taking advantage of expansions (2.13) to (2.17) the complete set of discrete equations given in Appendix A is obtained.

Convolutions sums and FFT

As presented in the previous section in many cases one has to evaluate an integral of the following type:

$$
\int_0^{L_x} \int_0^{L_y} \Bigg(\sum_{\substack{-N_1 \le l_1 \le N_1 \\ -N_2 \le l_2 \le N_2}} \hat{f}_{l_1,l_2} e^{\frac{2\pi i l_1}{L_x} x} e^{\frac{2\pi i l_2}{L_y} y} \Bigg) \Bigg(\sum_{\substack{-N_1 \le m_1 \le N_1 \\ -N_2 \le m_2 \le N_2}} \hat{g}_{m_1,m_2} e^{\frac{2\pi i m_1}{L_x} x} e^{\frac{2\pi i m_2}{L_y} y} \Bigg) \overline{e^{\frac{2\pi i k_1}{L_x} x} e^{\frac{2\pi i k_2}{L_y} y}} dxdy
$$

By some simple algebra one can easily show, that this integral equals the following double sum:

$$\sum_{\substack{-N_1\le l_1\le N_1\\ -N_1\le m_1\le N_1\\ l_1+m_1=k_1}}\ \sum_{\substack{-N_2\le l_2\le N_2\\ -N_2\le m_2\le N_2\\ l_2+m_2=k_2}} \hat{f}_{l_1,l_2}\hat{g}_{m_1,m_2} = \sum_{\substack{-N_1\le l_1\le N_1\\ -N_1\le k_1-l_1\le N_1}}\ \sum_{\substack{-N_2\le l_2\le N_2\\ -N_2\le k_2-l_2\le N_2}} \hat{f}_{l_1,l_2}\hat{g}_{k_1-l_1,k_2-l_2}$$

This is a two dimensional convolution sum which is of complexity $O(N_1^2N_2^2)$ (in case of a naive implementation). It is well known that these convolution sums can be evaluated efficiently by making use of fast transformation methods (i.e. fast Fourier transformations). In the following we denote the convolution sum given above by

$$\left(\hat{f} * \hat{g}\right)(k_1,k_2) = \sum_{\substack{-N_1\le l_1\le N_1\\ -N_1\le k_1-l_1\le N_1}}\ \sum_{\substack{-N_2\le l_2\le N_2\\ -N_2\le k_2-l_2\le N_2}} \hat{f}_{l_1,l_2}\hat{g}_{k_1-l_1,k_2-l_2}$$

Denoting by $\mathcal{F}$ the fast Fourier transformation and by $\mathcal{F}^{-1}$ the inverse fast Fourier transformation it is well known that the convolution sum can be calculated as follows:

$$\hat{f} * \hat{g} = \mathcal{F}\left(\mathcal{F}^{-1}(\hat{f}) \cdot \mathcal{F}^{-1}(\hat{g})\right)$$

In case of a two dimensional fast Fourier transformation each transformation is of complexity $O(N_1N_2logN_1N_2)$. In section 2.2.1 we have noticed that the whole numerical schema is of quadratic complexity $O(N_3^2)$ with respect to N_3. The reason for this complexity is that there exists no fast transformation method for Chandrasekhar polynomials. Since N_3 is small, typically $N_3 = 2$, this quadratic complexity is not of great importance. So the whole numerical schema is of complexity $O(N_3^2N_1N_2logN_1N_2)$.

Discrete penalization

To obtain the discrete equations for the penalized equations we apply again the same technique to decompose the solution u' into a poloidal, toroidal and mean flow part and get the following system of equations which has to be solved numerically:

$$\frac{1}{Pr}\left(\frac{\partial}{\partial t}\Delta\Delta_2 f + \delta \cdot div(u \otimes u)\right) = -Ra\Delta_2 T' + \Delta^2\Delta_2 f - \frac{1}{\eta}\delta \cdot \mathbf{1}_{\Omega^c}u' \tag{2.18}$$

$$\frac{1}{Pr}\left(\frac{\partial}{\partial t}\Delta_2 g + \epsilon \cdot div(u \otimes u)\right) = \Delta\Delta_2 g - \frac{1}{\eta}\epsilon \cdot \mathbf{1}_{\Omega^c}u' \tag{2.19}$$

$$\frac{\partial T'}{\partial t} + \Delta_2 f + (u \cdot \nabla)T' = \Delta T' - \frac{1}{\eta}\mathbf{1}_{\Omega^c}T' \tag{2.20}$$

$$\frac{1}{Pr}\left(\frac{\partial F}{\partial t} + \frac{\partial}{\partial z} < u_1u_3 >^{xy}\right) = \frac{\partial^2}{\partial z^2}F - \frac{1}{\eta} < \mathbf{1}_{\Omega^c}u' >^{xy} \tag{2.21}$$

$$\frac{1}{Pr}\left(\frac{\partial G}{\partial t} + \frac{\partial}{\partial z} < u_1u_3 >^{xy}\right) = \frac{\partial^2}{\partial z^2}G - \frac{1}{\eta} < \mathbf{1}_{\Omega^c}u' >^{xy} \tag{2.22}$$

Projecting these equations onto the horizontal and vertical ansatz functions gives again a set of discrete equations which are used to solve the system of partial differential equations numerically. To shorten notations the next formulas show the projections of the penalizations terms. So for the evolution equation for f we get:

$$
\begin{aligned}
&\frac{1}{L_xL_y}\int_{-\frac{1}{2}}^{\frac{1}{2}}\int_0^{L_y}\int_0^{L_x} \delta\cdot\frac{1}{\eta}\mathbf{1}_{\Omega^c}u'\overline{e^{\frac{2\pi i k_1 x}{L_x}}e^{\frac{2\pi i k_2 y}{L_y}}\psi_n(z)} = \\
&\quad \frac{1}{\eta}\sum_{j=1}^{N_3}\Bigg(\frac{2\pi i k_1}{L_x}\Big(< \partial_z^2\psi_j, \psi_n > \mathcal{F}_{x,y}\{\mathbf{1}_{\Omega^c}u_1^1(j)\}_{k_1,k_2} \\
&\quad + < \partial_z S_j, \psi_n > \mathcal{F}_{x,y}\{\mathbf{1}_{\Omega^c}u_1^2(j)\}_{k_1,k_2}\Big) \\
&\quad + \frac{2\pi i k_2}{L_y}\Big(< \partial_z^2\psi_j, \psi_n > \mathcal{F}_{x,y}\{\mathbf{1}_{\Omega^c}u_2^1(j)\}_{k_1,k_2} \\
&\quad + < \partial_z S_j, \psi_n > \mathcal{F}_{x,y}\{\mathbf{1}_{\Omega^c}u_2^2(j)\}_{k_1,k_2}\Big) \\
&\quad + (2\pi)^2|\frac{k}{L}|^2\Big(< \psi_j, \psi_n > \mathbf{1}_{\Omega^c}u_3^1(j)\}_{k_1,k_2}\Big)\Bigg)
\end{aligned}
$$

The penalization term for g is given by:

$$
\begin{aligned}
&\frac{1}{L_xL_y}\int_{-\frac{1}{2}}^{\frac{1}{2}}\int_0^{L_y}\int_0^{L_x} \epsilon\cdot\frac{1}{\eta}\mathbf{1}_{\Omega^c}u'\overline{e^{\frac{2\pi i k_1 x}{L_x}}e^{\frac{2\pi i k_2 y}{L_y}}\psi_n(z)} = \\
&\quad \frac{1}{\eta}\sum_{j=1}^{N_3}\Bigg(\frac{2\pi i k_2}{L_y}\Big(< \partial_z^2\psi_j, \psi_n > \mathcal{F}_{x,y}\{\mathbf{1}_{\Omega^c}u_1^1(j)\}_{k_1,k_2} \\
&\quad + < \partial_z S_j, \psi_n > \mathcal{F}_{x,y}\{\mathbf{1}_{\Omega^c}u_1^2(j)\}_{k_1,k_2}\Big) \\
&\quad - \frac{2\pi i k_1}{L_x}\Big(< \partial_z^2\psi_j, \psi_n > \mathcal{F}_{x,y}\{\mathbf{1}_{\Omega^c}u_2^1(j)\}_{k_1,k_2} \\
&\quad + < \partial_z S_j, \psi_n > \mathcal{F}_{x,y}\{\mathbf{1}_{\Omega^c}u_2^2(j)\}_{k_1,k_2}\Big)\Bigg)
\end{aligned}
$$

The penalization terms for F and G are given by:

$$
\begin{aligned}
&\int_{-\frac{1}{2}}^{\frac{1}{2}}\frac{1}{\eta} < \mathbf{1}_{\Omega^c}u_1 >^{xy}\overline{S_n(z)} \\
&\quad = \frac{1}{\eta}\sum_{j=1}^{N_3} < \partial_z\psi_j, S_n > \mathcal{F}_{x,y}\{\mathbf{1}_{\Omega^c}u_1^1(j)\}_{0,0} + < S_j, S_n > \mathcal{F}_{x,y}\{\mathbf{1}_{\Omega^c}u_1^2(j)\}_{0,0} \\
&\int_{-\frac{1}{2}}^{\frac{1}{2}}\frac{1}{\eta} < \mathbf{1}_{\Omega^c}u_2 >^{xy}\overline{S_n(z)} \\
&\quad = \frac{1}{\eta}\sum_{j=1}^{N_3} < \partial_z\psi_j, S_n > \mathcal{F}_{x,y}\{\mathbf{1}_{\Omega^c}u_2^1(j)\}_{0,0} + < S_j, S_n > \mathcal{F}_{x,y}\{\mathbf{1}_{\Omega^c}u_2^2(j)\}_{0,0}
\end{aligned}
$$

Figure 2.4 and 2.3 show some simulation results for penalized simulations. The first one simulates convection in a rectangular cell with no-slip boundary conditions in all directions. The result is no longer periodic with respect to the rectangular non-periodic domain. The second simulation shows convection around a grid of cylinders. The outer boundaries of the cell are still periodic. In both cases $\eta = 10^{-2}$ is sufficient.

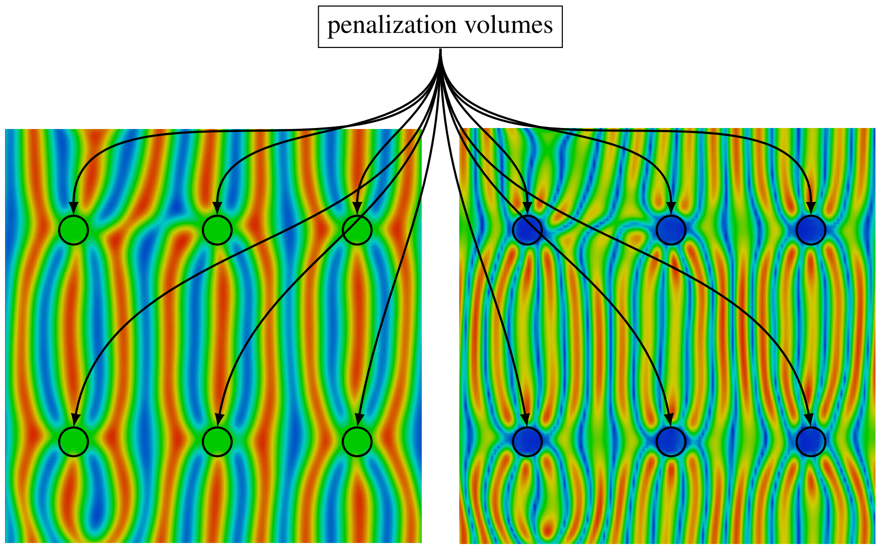

Figure 2.3: Simulation result in a more complex domain. This example shows the convection around a 2×3 grid of cylinders. In this case the boundaries of the surrounding rectangle are periodic ones. The aspect ratio is $\Gamma = 10$ and $Ra = 2900$, $Pr = 1$. 128×128 modes were used in the horizontal plane. The left graph shows the temperature field at $z = 0$ (red indicates hot fluid, blue cold fluid) and the right graph shows the magnitude of the velocity field at $z = 0$ (red indicates higher magnitudes and blue indicates zero velocity). The fluid's velocity and temperature are forced to be zero (in this case the maximum magnitude of T' and $|u|^2$ is smaller than 10^{-5}) by penalization with $\eta = 10^{-2}$.

2.2.2 Dealiasing

In Section 2.2.1 we have presented one of the discrete nonlinear operators (i.e. convolutions). Due to the fact that one has to consider truncated Fourier series in case of a numerical simulation schema, the results of the nonlinear operator suffer from an error called "aliasing". This error source is related to truncated Fourier series and an undersampling phenomenon. To illustrate this effect in one space dimension let us consider two periodic signals u, v represented by a Fourier series (denoting by u_j the value of u at $x_j = \frac{jL_x}{N_1}$ where N_1 denotes the number of Fourier coefficients and $j \in \{0, \cdots, N_1 - 1\}$):

$$u_j = \sum_{k_1=-N_1}^{N_1} \hat{u}_{k_1} e^{\frac{2\pi i k_1}{L_x} x_j}$$

$$v_j = \sum_{k_1=-N_1}^{N_1} \hat{v}_{k_1} e^{\frac{2\pi i k_1}{L_x} x_j}$$

As defined in Section 2.2.1 the result $\hat{s}_{k_1}$ of the nonlinear operator can be calculated as follows:

$$\hat{s}_{k_1} = \frac{1}{N_1} \sum_{j=0}^{N_1-1} u_j v_j e^{-\frac{2\pi i k_1}{L_x} x_j}$$

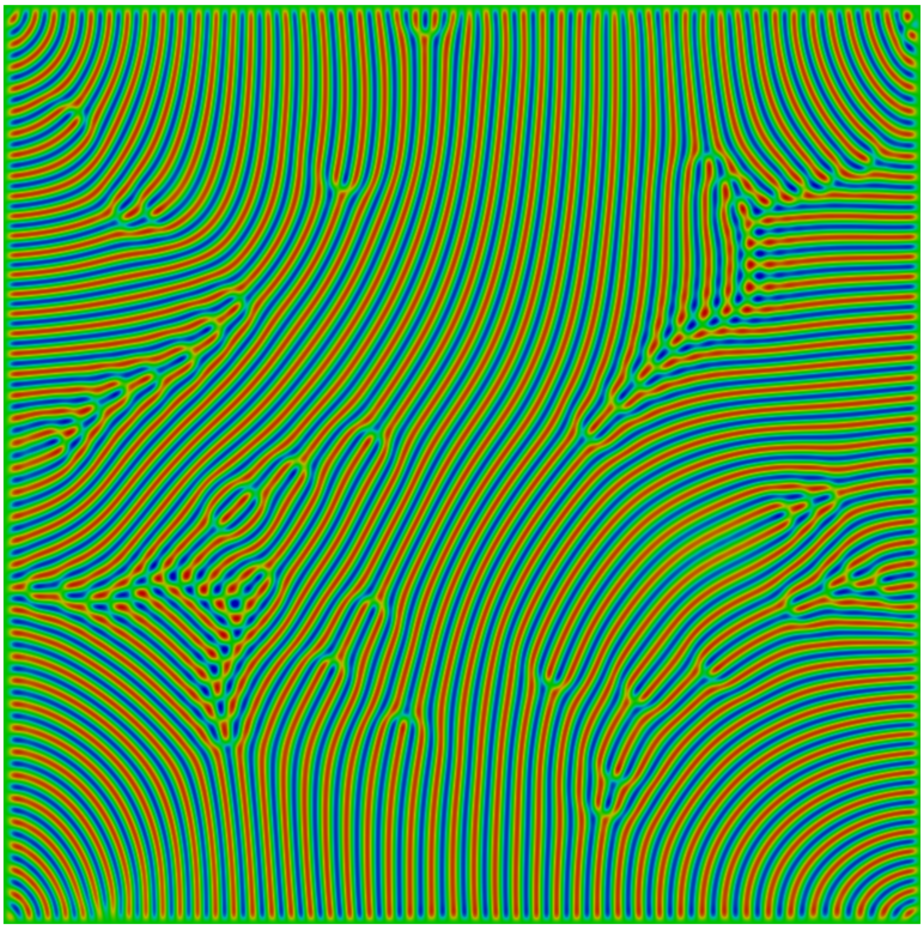

Figure 2.4: Numerical simulation of Rayleigh-Bénard Convection for $Ra = 2100$, $Pr = 0.96$ within a bounded domain. To keep the simulation schema nearly unchanged a penalization technique described in section 2.1.3 with $\eta = 0.01$ was used. Notice that the simulation is executed within the complete domain (including the green boundaries of the rectangle) while the simulation result is just the inner part of the rectangle. The penalization terms force the temperature field to tend to zero in the vanishing limit of η outside of the inner rectangle.

Inserting the expansions u_j and v_j as Fourier series in this definition leads to:

$$\hat{s}_{k_1} = \sum_{l_1+m_1=k_1} \hat{u}_{l_1}\hat{v}_{m_1} + \underbrace{\sum_{l_1+m_1=k_1\pm N_1} \hat{u}_{l_1}\hat{v}_{m_1}}_{\text{aliasing term}} \tag{2.23}$$

The last sum of the right side is the so-called aliasing term. Since this term is created by truncation of Fourier series (i.e. $N_1 < \infty$) this is an error term representing a higher/lower wavenumber of $k_1 \pm N_1$ as one of wavenumber k_1.

Of course we want to avoid aliasing errors in case of numerical simulations, since we do not have a rigorous error bound for them. The most commonly used techniques are padding and truncation techniques. The idea is to pad the given vector of Fourier coefficients with zeroes in such a way that the aliasing term vanishes. The padding technique therefore adds to the vector given Fourier coefficients zeros of the same length. It is easy to verify that the aliasing term

$$\sum_{l_1+m_1=k_1\pm 2\cdot N_1} \hat{u}_{l_1}\hat{v}_{m_1}$$

is zero for all $k_1 \in \{-N_1, \cdots, N_1\}$. Of course this technique has the disadvantage of doubling the length of each transformation. Another technique to avoid aliasing errors is a truncation technique. First of all we define a new field of Fourier coefficients $\breve{u}$ and $\breve{v}$ in the following way (with $M_1 \geq \frac{3}{2}N_1$):

$$\breve{u}_{k_1} = \begin{cases} \hat{u}_{k_1} & \text{if } -\frac{M_1}{2} \leq k_1 \leq \frac{M_1}{2} \\ 0 & \text{sonst} \end{cases}$$

In the same way we define $\breve{v}_{k_1}$. If we now build $\breve{s}_{k_1}$ analogue to (2.23) the aliasing term vanishes and we get:

$$\hat{s}_{k_1} = \breve{s}_{k_1} \text{ if } k_1 \leq |N_1/2|$$

A formal proof of this technique can be found in [12]. Of course this technique has the advantage of a smaller number of Fourier coefficients compared to the first technique, but has the disadvantage of an uncomfortable transformation size with respect to many fast Fourier transformation implementations. Since this is a major drawback in many cases it is the preferred way to choose a data field of Fourier coefficients of an arbitrary size fitting to the FFT algorithm and setting the outer one-third Fourier coefficients to zero before each nonlinear operator evaluation. This algorithm is called two-third rule, also introduced and described in [12]. Of course this procedure limits the number of usable Fourier coefficients for a given number of Fourier coefficients up to two-third but has the advantage of being conform with respect to many fast Fourier transformation libraries.

2.3 Time-stepping methods

Since the governing, analytic equations (1.19) to (1.23) are time dependent ones and our semi-discrete equations of the numerical simulation schema are still time dependent (, as we have discretized them at first with respect to the space coordinates x, y, z), a time-stepping method is needed. Of course there are different aspects which have to be considered when choosing a time-stepping method.

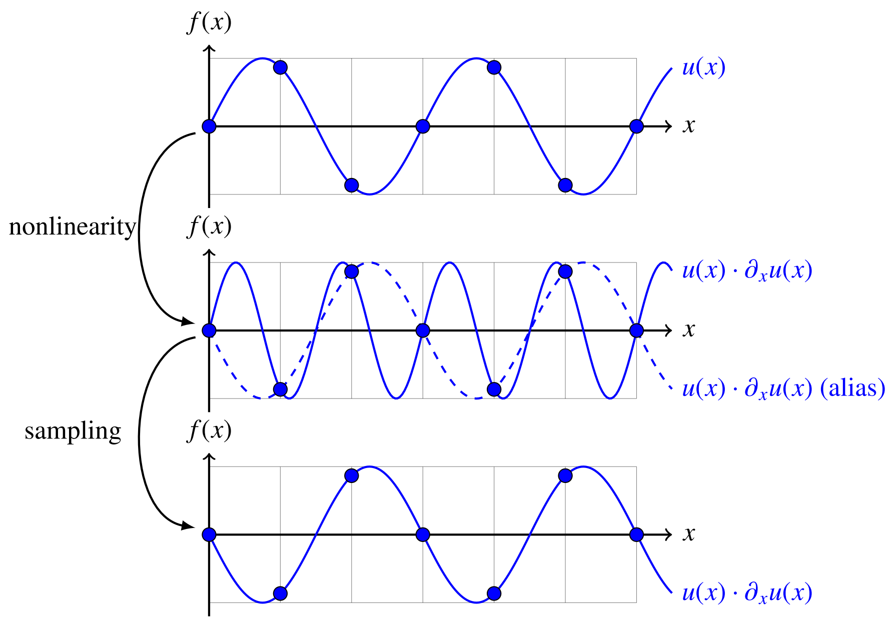
$f(x)$
$u(x)$
x
nonlinearity
$f(x)$
$u(x) \cdot \partial_x u(x)$
x
$u(x) \cdot \partial_x u(x)$ (alias)
sampling
$f(x)$
x
$u(x) \cdot \partial_x u(x)$

Figure 2.5: Visualization of aliasing errors, caused by the second order nonlinearity of the Navier-Stokes equations and truncated Fourier series (see [14]). The thin grid represents the sampling points. One can easily verify that the nonlinear operator doubles the frequency of u. If the result has a frequency which is too high the discretization will interpret the signal as an alias (the Nyquist-Shannon theorem is not satisfied). Padding or truncation techniques have to be used to avoid errors of this type.

First of all one has to be aware of the fact that the linear operator is a stiff one (at least for large N_1 and N_2, which will be the case for our simulation goals). This can be seen easily by considering the block-diagonals of the matrix L for T' and f (in case of g the stiffness is obvious, since it is a diagonal matrix):

$$\begin{pmatrix} L_{T'} & L_{T',f} \\ L_{f,T'} & L_f \end{pmatrix} \cdot \begin{pmatrix} \hat{T}'_{k_1,k_2,1} \\ \vdots \\ \hat{T}'_{k_1,k_2,N_3} \\ \hat{f}_{k_1,k_2,1} \\ \vdots \\ \hat{f}_{k_1,k_2,N_3} \end{pmatrix}$$

Since $L_{T'}$ and L_f are again diagonal matrices in case of two vertical ansatz functions, and their smallest and largest eigenvalues are proportional to $(N_1^2 + N_2^2)$ and $(N_1^2 + N_2^2)^2$, this is indeed a stiff operator.

Therefore a time-stepping method which is appropriate for stiff ordinary differential equations is required in this case. In many cases one might choose a fully implicit method, which has the disadvantage to invert the complete operator

$$B^{-1}\big(L + V\big)$$

in each iteration. Of course calculating the inverse of $B^{-1}L$ can be done once in parallel before starting the simulation. As the matrix $B^{-1}L$ has to be inverted for each pair (k_1, k_2) independently, this is not too problematic. Nevertheless calculating the inverse of the complete operator $B^{-1}(L + V)$ is too complex since the nonlinear part is even more complex and couples mode pairs (k_1, k_2) with other mode pairs (k_1', k_2'). So a time-stepping method is needed which offers a large stability region, but avoids the necessity to calculate the inverse of the complete or nonlinear operator. There are two choices by which this needs can be satisfied:

- choosing a explicit method, which has a sufficient large stability region
- choosing a method which treats linear and nonlinear part differently

The first one leads to a time-stepping method which is called Exponential Time Differencing and the second one to a splitting method. Both methods can achieve different orders and are described in the following sections.

2.3.1 Exponential Time Differencing

Exponential Time Differencing (ETD) is an explicit time-stepping method (introduced for example in [17]). Lets rewrite our systems of ordinary differential equations again in the following abstract way:

$$\partial_t V = B^{-1}L \cdot V + B^{-1}N(V)$$

To derive exponential time differencing (ETD) method we use an integrating factor $e^{-B^{-1}Lt}$ (where $B^{-1}L$ is our matrix representation of the linear differential operator). Integration over a time-step of length h from $t_{n+1} = t_n + h$ will give:

$$\begin{aligned} [V(t_{n+1})e^{-B^{-1}Lh} - V(t_n)]e^{-B^{-1}Lt_n} &= \int_{t_n}^{t_{n+1}} e^{-B^{-1}Lt}N(V(t))dt \\ \Leftrightarrow V(t_{n+1}) = V(t_n)e^{B^{-1}Lh} + e^{B^{-1}Lh} &\int_{t_n}^{t_{n+1}} e^{-B^{-1}Lt}N(V(t))dt \end{aligned}$$

So the different versions of Exponential Time Differencing are derived by different approximations of the integral. In case of fourth order Runge-Kutta method we get the following

$$\begin{aligned} a_n &= V_n e^{B^{-1}Lh/2} + L^{-1}B[e^{B^{-1}Lh/2} - I]N(V_n) \\ b_n &= V_n e^{B^{-1}Lh/2} + L^{-1}B[e^{B^{-1}Lh/2} - I]N(a_n) \\ c_n &= a_n e^{B^{-1}Lh/2} + L^{-1}B[e^{B^{-1}Lh/2} - I][2N(b_n) - N(V_n)] \\ V_{n+1} &= e^{B^{-1}Lh}V_n + [\alpha N(V_n) + 2\beta[N(a_n) + N(b_n)] + \gamma N(c_n)] \end{aligned}$$

where:

$$\begin{aligned} \alpha &= h^{-2}(B^{-1}L)^{-3}[-4 - B^{-1}Lh + e^{B^{-1}Lh}[4 - 3B^{-1}Lh + (B^{-1}Lh)^2]] \\ \beta &= h^{-2}(B^{-1}L)^{-3}[2 + B^{-1}Lh + e^{B^{-1}Lh}(-2 + B^{-1}Lh)] \\ \gamma &= h^{-2}(B^{-1}L)^{-3}[-4 - 3B^{-1}Lh - (B^{-1}Lh)^2 + e^{B^{-1}Lh}(4 - B^{-1}Lh)] \end{aligned}$$

Those terms α,β and γ are higher order variants of $\frac{e^h - 1}{h}$. So they suffer from cancellation errors for small h. Du and Zhu in [18] suggest an approximation of this term by a complex contour integral. Since α, β and γ are fixed for fixed h we chose to calculate them once by high precision arithmetics (using MPFR, see [24]) and use them as constants during simulation.

As you can see from the previous equations a matrix exponential of the form $e^{B^{-1}Lx}$ with $x \in \mathbb{R}^n$ has to be calculated. Since this is a more complex task for matrices which are not diagonal, we chose to use this method only for simulations of Swift-Hohenberg equations (in this case the linear part is diagonalized by choosing Fourier series expansions in both space directions). In case of the Boussinesq equations this procedure is much too complex (with respect to the algorithm and memory usage) and so we chose to take advantage of other time-stepping techniques, which are called splitting methods. These methods will be described in the following section.

2.3.2 Splitting Methods

Splitting methods are time-stepping methods which treat the linear and the nonlinear part in a different way. We consider again the following abstract form of our system of ordinary differential equations:

$$\partial_t V = B^{-1}L \cdot V + B^{-1}N(V)$$

Since we have a stiff linear differential operator L and a nonlinear differential operator N we use an implicit method for the linear part and an explicit method for the nonlinear part. So the algorithm looks like this:

We use $V(t_n)$ as an input for our time stepping method and apply an implicit schema to it. So we get an intermediate field, for example by a simple implicit Euler method (where h denotes the step size):

$$\tilde{V}(t_n) = V(t_{n-1}) + hB^{-1}L \cdot \tilde{V}(t_n)$$

This intermediate field $\tilde{V}(t_n)$ is then used as input for the explicit time-stepping method of the nonlinear part. For example a second order Adams-Bashforth method can be used:

$$V(t_{n+1}) = \tilde{V}(t_n) + h\Big(\frac{3}{2}B^{-1}L\tilde{V}(t_n) - \frac{1}{2}B^{-1}LV(t_{n-1})\Big)$$

In case of Boussinesq equations we use a simple implicit Euler method for the implicit part and a second order Adams-Bashforth method for the explicit part. We apply the nonlinear time-stepping method after the linear one, as the nonlinear part forces to fulfill the boundary conditions (if a penalization method is applied).

In case of this implicit-explicit splitting method the step size can be nearly arbitrary large (typically one can choose up to $h = 0.05t_v$, where $t_v = d^2/\kappa$ denotes the vertical diffusion time). Of course one has to choose the step size with respect to accuracy, i.e. to satisfy different numerical properties, like the well known CFL condition, which states that a fluid particle should not move farther than the minimal distance between two grid points.

Chapter 3

Implementation of Pseudo-Spectral Methods on a GPU

This Chapter describes the implementation of pseudo-spectral methods on graphics cards. We use NVIDIA's graphics cards programming interface CUDA to show how a pseudo-spectral method is implemented efficiently on a GPU. The first Section 3.1 describes some general details of a graphics card's processing unit which have to be known if you want to implement algorithms on a GPU. Section 3.2 uses this knowledge to show our spectral simulations as an implementation example.

3.1 GPU architecture

CUDA (computing uniform device architecture) is NVIDIA's approach towards high performance computing. It allows the developer to use the massive parallel architecture of a GPU for his own purpose. This is possible by using some additional programming language bindings (e.g. C, C++, FORTRAN) and a specific CUDA compiler (namely NVCC) which extends the features of the general compiler (like GCC on Linux systems). To achieve a nice speed-up of your simulation code compared to original CPU implementations a deeper knowledge of a GPU's hardware architecture is an absolute requirement. This enables the programmer to write efficient codes for GPUs. Therefore the next section introduces the basic features of a GPU hardware and explains how to use them in a way to get their highest performance. There are a number of applications which already have shown the performance of graphics cards for scientific simulations. Some of them are:

- molecular dynamics (see [27])
- simulations of Lattice-Boltzmann equations (see [25])
- finite difference methods in electrodynamics (see [26])

3.1.1 Execution unit architecture

A GPU is a hierarchical, massive parallel execution unit. This means that the large execution unit (which is the GPU) is subdivided into smaller execution units which are called Multi-Cores. They are subdivided again into smaller units which are called CUDA-Cores (see [22]). All in all the execution unit structure can be summarized as follows:

GPU: largest logical execution unit, consists of many Multi-Cores

Multi-Core: execution unit, consists of many CUDA-Cores

CUDA-Core: smallest logical execution unit

Of course the CUDA-Cores itself is a small execution unit, which has some of the basic features of a processor like an arithmetic logical unit (ALU), several registers, etc.. They will not be presented here, because they are not accessible in many cases and their utilization is controlled by the CUDA driver API (a more detailed description of this topic is given in [22]).

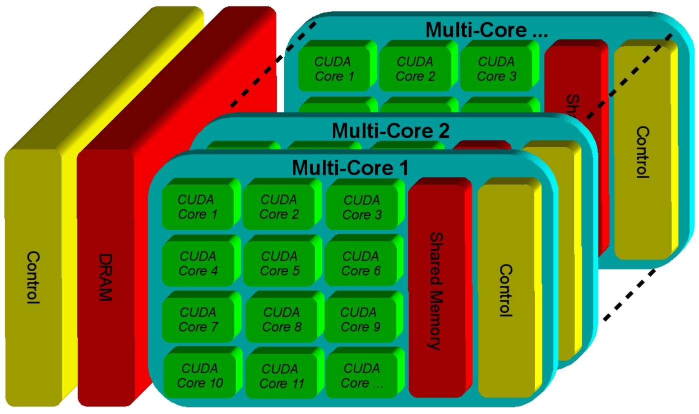

Figure 3.1: Layout of a CUDA GPU. Each GPU consists of multiple Multi-Cores. All Multi-Cores have access to the global memory of the GPU. A Multi-Core has many CUDA-Cores and all of them have access to the shared memory. The CUDA-Cores themselves have several registers and some local memory.

3.1.2 Memory architecture

The memory architecture of a CUDA GPU is closely related to the structure of its execution units. Therefore each hierarchical layer has its own memory layer which is only accessible by a single unit of this layer and all of its subunits. Therefore a graphics card has three memories which are the following ones:

global memory: largest memory, which is accessible by the whole GPU (including all Multi-Cores and CUDA-Cores)

shared memory: smaller memory, which is accessible by a single Multi-Core (including all CUDA-Cores)

local memory: smallest memory, which is accessible by a single CUDA-Core

Of course there are a number of registers which are accessible by a CUDA-Core, but they are of less importance to a software developer, as their utilization is generally managed by the CUDA driver API. Figure 3.1 summarizes the hardware architecture of a single GPU showing its execution units, together with their memory layers.

The reason to use different memory layers is a rather simple one. As the different layers are spatially more or less close to their execution units, the latency and so their bandwidth are different ones. Therefore the developer is obliged to make use of local and shared memory as much as possible. Of course this requires a well organized data structure to ensure that the data is accessible from the different execution units at any time if there is a need to read or update them.

3.1.3 CUDA C/C++

One way to make use of a GPU's computing power is to use an extended version of C/C++. So the developer can use normal C/C++ code and can introduce some specific CUDA code which handles memory transfer and CUDA kernels to achieve the required computations.

One should mention that different graphics processors support different compute capabilities. Different capabilities support themselves different features like double precision floating point operations (compute capability ≥ 1.3), support of object oriented design, etc.. All of these features are described in [22]. Since code with higher compute capabilities is not executable on graphic cards which support only a lower one, we chosen to use configuration files which specify the level of support. Therefore the developed code is executable on all CUDA enabled devices which support CUDA 1.0 and higher.

In general all functions/methods are separated into three classes:

host functions: Functions which are executed by the host (i.e. the CPU). They are callable only from other host functions and not from any device function. They are defined by the "__host__" macro. The nvcc compiler separates this code part from the device code and uses the system's C-compiler to compile the code into object files.

global functions: Functions which are executed by CUDA devices (i.e. the GPU) and are callable only from host functions. These functions are not callable from other global or device functions. Global functions are defined by the "__global__" macro. The nvcc compiler separates global and device functions from host code and compiles them for the specific CUDA target platform.

device functions: Functions which are executed by the CUDA device and are callable from all other global or device functions, but not from other host function. These functions are defined by the "__device__" macro.

To take advantage of the large amount of CUDA cores we use a large amount of threads, each of them running on a different CUDA core. Which CUDA core executes a specific thread is determined at runtime. Therefore the programmer should never assume any order of thread execution. Threads are logically ordered in blocks and blocks themselves are ordered into grids. This hierarchy is clearly related to the execution unit architecture and to the memory architecture, i.e. a block (consisting of many threads) is running on a single Multi-Core and therefore has access to shared memory of this

Multi-Core unit. Of course each thread within this block has its own local memory and some registers which are only accessible by this thread. The whole grid is running on a single GPU and therefore all threads within this grid have access to the global memory of this CUDA device. One can imagine a block as a one, two or three dimensional array of threads, and a grid as a one, two or three dimensional array of blocks (the dimension of the grid-/block-array can be specified by the programmer).

To specify how many threads per block and how many blocks per grid are used a global function call has to be configured. This means that each time a global function is called from a host function two vectors (each consisting of up to three elements) of dim3-type have to be used to specify dimensions of grids and blocks. Listing 3.1 shows a small example how this configuration is realized within the source code.

```
/*a function which is executed on the CPU (host)
* and exclusively callable from the CPU (host)
*/
__host__ void a_host_function() {

  //...some previous code executed on the CPU

  //...create dimensions of grid
  dim3 grid_dim = create_grid_dimensions(data_size);
  //...create dimensions of block
  dim3 block_dim = create_block_dimensions(data_size);

  //...call the global function
  //...notice that data is a pointer to device memory
  a_global_function<<<grid_dim, block_dim>>>(data)

  //...other host code

}
```

Listing 3.1: A global function call from a host function has to be configured to specify dimensions of blocks and grids and the number of threads.

CUDA C/C++ is a single-instruction-multiple-data model (SIMD model), i.e. each CUDA-Core executes the same code but all of them have different data. Of course a mechanism is needed to provide the possibility to handle different tasks on different CUDA-Cores. Therefore the programmer has to include an algorithm to distinguish between the different CUDA-Cores to specify a certain task for a CUDA-Core. CUDA provides a simple mechanism to realize this, which works in the following way: Within each global and device function the following built-in identifiers are accessible:

threadIdx: Identifier for a thread within a block (i.e. within the array of threads). ThreadIdx has three properties threadIdx.x, threadIdx.y, threadIdx.z to identify the position of a thread within a block.

blockIdx: Identifier for block within a grid (i.e. within the array of blocks). BlockIdx has three properties blockIdx.x, blockIdx.y, blockIdx.z to identify the position of a block within a grid.

blockDim: Identifier representing the size of the current block. The three sizes are represented by blockDim.x, blockDim.y, blockDim.z.

gridDim: Identifier representing the size of the current grid. The three sizes are represented by gridDim.x, gridDim.y, gridDim.z.

Since the numerical simulation of Rayleigh-Bénard Convection requires to handle a large number of Fourier coefficients (in *x* and *y* direction) a simple algorithm is used to figure out which thread handles a specific Fourier coefficient. Listing 3.2 describes this algorithm which is used for the numerical simulation and is an easy way to determine a thread to handle a specific Fourier coefficient.

```
__device__ int get_global_index() {
  return threadIdx.x + threadIdx.y*blockDim.x + threadIdx
      .z*blockDim.x*blockDim.y + blockIdx.x*blockDim.x*
      blockDim.y*blockDim.z + blockIdx.y*blockDim.x*
      blockDim.y*blockDim.z*gridDim.x + blockIdx.z*
      blockDim.x*blockDim.y*blockDim.z*gridDim.x*gridDim.y
      ;
}
```

Listing 3.2: Device function used to figure out which thread handles a specific Fourier coefficient. This simple function returns a unique integer number for each thread between zero and the total number of threads. As there are as many threads as matrix entries it is easy to determine which thread is responsible for a specific Fourier coefficient.

It is worth mentioning that each host thread is able to handle only a single GPU. Conversely a single graphic processor is accessible by up to four host threads, but data from one host thread which is located on the GPU is not accessible to any of the other threads accessing the same GPU. Therefore a simulation using multiple GPUs has to use the host memory to exchange data between different GPUs even if the GPUs are located in the same machine. If the GPUs are distributed across different machines a fast interconnect has to be used to reduce latency, because of data transfers. One standard way to implement multi GPU support is to use OpenMP (in case of multiple graphic cards within the same machine), Message Passing Interface (MPI, in case of graphic cards distributed across different machines) or a combination of both (hybrid parallelization).

3.1.4 Streams and parallel execution

Graphics cards are massive parallel processors. As a large amount of threads is executed on them, concurrency is not guaranteed between different threads. Since synchronization (i.e. waiting for other threads to reach a certain position in the code) slows program execution down, graphics cards are in favor to load the next kernel even before the previous kernel has finished completely and some CUDA-Cores become idle. Of course there might be some threads of the previous kernel which have not finished their execution at this timepoint. This feature provides maximal parallelism. In some cases this feature is not wanted, as the result of a previous kernel launch is an input of the later one. Therefore program execution should wait as long as the previous kernel is executed and all threads have finished their execution. This blocking feature can be realized by calling the cudaThreadSynchronize function. This procedure forces the graphics card to finish all kernels before returning from cudaThreadSynchronize function. The major disadvantage is that any parallelism between different kernels is completely destroyed, but in some cases this feature is needed, because a kernel call needs the complete output of a previous kernel.

Therefore the CUDA-API provides so-called streams as a possibility to increase parallelism, but to avoid race conditions. To take advantage of streams you add several kernels to a stream and start the stream. This procedure assures that kernels are executed in the right order without any race conditions (i.e. all kernels in a stream are executed serially). Streams (except for the default stream) are always executed asynchronously, i.e. the kernel call will return immediately and so multiple streams can be executed at the same time. Streams are identified by the cudaStream_t structure. Listing 3.3 gives a small example how to implement stream creation and execution of kernels within its context. Streams are used in many cases to calculate the linear operator since the different linear parts for f, g, θ, F, G can be computed in parallel.

```
//...previous host code
...

//...create streams
cudaStream_t stream[number_of_streams];
for (int i = 0; i < number_of_streams; ++i)
  cudaStreamCreate(&stream[i]);

//...execute kernels and add them to different streams
for (int i = 0; i < number_of_streams; ++i)
  a_global_function<<<grid_dim, block_dim,stream[i]>>>(
      data);

//...synchronize all streams, since streams are executed
    asynchronously
cudaThreadSynchronize();

//...other host code
...
```

Listing 3.3: CUDA streams are used to increase parallelism and to avoid hardcoded synchronization. This increases the performance and assures right execution order of different threads.

In some cases (often caused by making use of shared memory) a smaller synchronization level is needed, because cudaThreadSynchronize synchronizes all threads of the complete GPU. Furthermore this synchronization technique is accessible exclusively from host code. Therefore the CUDA-API provides a more subtle synchronization process which is also accessible from code which executed on the CUDA device. Calling the __syncthreads function within a global or device function forces all threads within the same block to synchronize to this point. This function is very important to calculate scalar-products or reduce-summations for example (which is necessary to calculate structure functions and energy spectra).

Another important feature which is available since CUDA 1.1 and higher are atomic functions. Suppose we want the threads of a block to access and modify the same variable (for example every thread increases the variable by one). Since all threads are executed asynchronously it may happen that one thread (say thread 2) reads out the value to modify the variable just after another thread (say thread 1) has read the same value, but before writing the new result back. Now thread 1 writes his result back to the memory and thread 2 will do the same. This leads to incorrect data, since the update of thread 1 is lost. Atomic functions remedy this problem, since they are executed without any interruption. In the described example the function atomicAdd is the right function to use. The complete set of atomic functions and a way to program other atomic functions are described in [22].

3.2 Efficient implementation

This section describes the implementation of spectral methods on graphics cards in detail. We apply theory of Chapter 2 and knowledge of Section 3.1 to present a specific implementation. Although it is a specific implementation for Rayleigh-Bénard Convection it is an example of how to parallelize other numerical methods on GPUs. One of the basic questions is the data management as figured out in the previous chapter. Therefore the following section describes the data flow and the data management in detail and presents their level of efficiency. Furthermore the following sections describe some other aspects of software engineering. As some basic object oriented aspects of C++ (as far as possible in combination with CUDA) were used to implement the simulation method, these will be presented as well. Another point is the CUDA fast Fourier transformation library CuFFT which was used as an FFT library within this project.

3.2.1 Data flow and structure

As described in Section 3.1 data management is one of the important topics in programming graphics cards. So this is the first point which has be considered. It could not be overemphasized that using a nice data structure which takes advantage of the hierarchical structure of GPU memory is a key to a well performing simulation.

The workspace of the simulation are the Fourier coefficients $\hat{f}(k_1,k_2,i), \hat{g}(k_1,k_2,i), \hat{T}'(k_1,k_2,i), F(i), G(i)$ of functions f, g, T', F, G given by equations (1.19) to (1.23). The discrete expansions (2.13) to (2.17) lead to a discrete set of equations and projections of the linear and nonlinear parts, presented in the previous chapter, require computation of N_3^2 two dimensional convolutions. So the idea is to arrange the Fourier coefficients as a three-dimensional array for each function. Each layer of an array holds the Fourier coefficients for the horizontal plane. Using for example the temperature coefficients, Figure 3.2 shows how data is ordered within global memory of the GPU. This arrangement allows simple pointer arithmetic and batched fast Fourier transformations, too. By making use of real FFTs the memory amount is roughly reduced by a factor of one-half compared to a complex to complex transformation. Of course the Fourier coefficients of f, g, F, G are ordered in a similar way and take advantage of these features, too.

So these are the data fields which are used for simulation operations. Of course the linear operator is easy to compute, as it is only a simple matrix-matrix product in Fourier space. Equation system (A.12) gives the mathematical equation system (in case of f and T') and it is obvious, that there is only a coupling of the Fourier coefficients for different vertical ansatz function number (indicated by the third index in equation

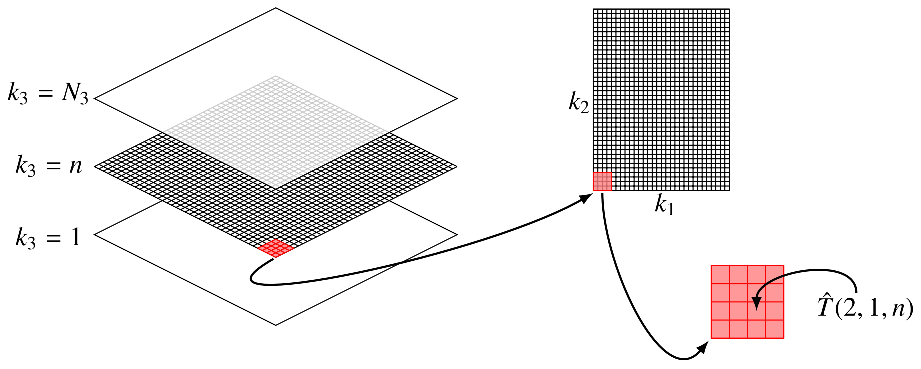

Figure 3.2: Data structure of temperature field within the global GPU memory. Other data arrays are structured in a similar way. This arrangement allows easy pointer arithmetic to iterate through the data. Moreover batched fast Fourier transformations are possible.

(A.12)). The discrete equations for $\hat{g}, F, G$ are simple pointwise operations in Fourier space. Therefore it is a natural approach to use one thread per Fourier coefficient to calculate its linear part. A more complicated approach has to be used to calculate the nonlinear part. As described in Section 2.2.1 it is the goal to compute nonlinearities in physical space by a simple pointwise multiplication and to use fast Fourier transformations to switch between physical and Fourier space. As we are only investigating moderate Rayleigh and Prandtl numbers (i.e. considering only weak turbulent convection) the number of vertical ansatz functions is typically small and in our case $N_3 = 2$ is sufficient. Therefore we use fast transformation methods of complexity $O(N_1 N_2 log(N_1 N_2))$ in the horizontal plane and use simple iterative method of complexity $O(N_3^2)$ for transformations in the vertical plane. This is still efficient, as fast transformation methods are slower than naive implementations for a small number of coefficients N_3, because fast transformation methods require an additional constant time for calculations. The idea is therefore to leave all data (i.e. the Fourier coefficients) inside the global memory and iterate over all vertical ansatz functions in a double loop executed on the CPU, as described e.g. in equation (A.15).

The algorithm to calculate the nonlinear operator of f, g, T', F, G therefore has the following structure: First $u_1^1(i), u_2^1(i), u_3^1(i), u_1^2(i), u_2^2(i), \partial_x T_1(i), \partial_y T_1(i), T_1(i)$ are calculated for all $i \in \{1, \cdots, N_3\}$ using fast Fourier transformations. In a loop we build all convolution sums in physical space, i.e. build $u_1^1(i) \cdot u_1^1(j), u_1^1(i) \cdot u_2^1(j), \cdots$ and apply inverse fast Fourier transformations to them. Using the values of the scalar products, which appear in the nonlinear discrete equations, the nonlinear operator can be calculated by a pointwise addition / multiplication inside the surrounding loop. The whole algorithm is summarized in Figure 3.3. Green nodes indicate that this step of the algorithm is calculated on the GPU, while red ones are executed on the CPU. The data stays for the whole algorithm (including linear/nonlinear operator and time-stepping) within global GPU memory. To increase parallelism we have used stream execution wherever it is possible. Additionally the execution performance is increased by making use of shared memory. Of course this performance optimizations have only little effect, as the most time consuming operation is the (inverse) fast Fourier transformation. In addition this influence decreases as the system size in the horizontal plane increases.

Calculating the penalization term is (logically) included in the nonlinear term, although it is a linear operator. The reason to implement it in this way is a relative simple one. As the penalization requires to compute convolution sums in Fourier space / pointwise products in physical space of the type

$$\mathbf{1}_{\Omega^c} u_1^1(i), \mathbf{1}_{\Omega^c} u_2^1(i), \cdots$$

one is able to reuse outputs of Fourier transformations of $u_1^1(i)$ to reduce the number of transformations. As we assume the penalization mask to be a function of the horizontal coordinates, i.e. $\mathbf{1}_{\Omega^c} = \mathbf{1}_{\Omega^c}(x, y)$, the memory for the penalizations terms has to have the dimension $N_1 \times N_2$. The algorithmic part of the penalization is described in figure 3.3 by the left most column.

3.2.2 Object oriented design

The simulation software is programmed in CUDA-C/C++. To allow a maximal reusability and take advantage of object oriented programming techniques, C++ features were used as much as possible. As the different versions of CUDA support different levels of object orientation not all features could be used. A detailed description of the supported features is given in [22].

Data fields in Fourier and physical space are represented by matrix classes. Both data fields can be realized as matrices in host and in device memory. The data fields are therefore handled by the following four classes: matrix_host, matrix_host_physical, matrix_device, matrix_device_physical. All of them inherit from matrix class. Matrix classes which keep their data in host memory are used only for initialization and input and output to disk. All classes have methods to execute simple algebraic manipulations like scaling, addition and multiplication of matrices. Furthermore the Fourier representations have methods to calculate first derivative, laplacian, and many more.

Operators are split into two classes: Linear and nonlinear ones. The linear ones inherit from class linear_operator and nonlinear operators from class nonlinear_operator. All operator classes provide the calculate_operator interface, which applies the operator to the given input. Furthermore linear operators provide a method to get their matrix representation. A more detailed overview about the most important classes and their inheritance tree can be seen in Figure 3.4.

3.2.3 CuFFT

Since an efficient implementation requires a well performing fast Fourier transformation, we chose to use the CUDA FFT library CuFFT, which is delivered together with the CUDA software development kit (SDK). This library offers one, two, three and arbitrary dimensional transformations. Since CuFFT 3.0 this library uses shared memory which speeds transformations up by a large factor. Compared to CPUs its peak and average performance during transformations is significantly higher (see [23]), if a specific transformation size is reached. Since all transformations and other computations for the numerical simulations are executed by the GPU, memory transfers to the CPU can be ignored completely.

As many other CPU versions of fast Fourier transformations (like FFTW library) CuFFT is working with transformation plans. So the procedure to calculate a Fourier transformed of a given data field is the following one: First create a CuFFT transformation plan, which specifies the size of the transformation, the type and whether there

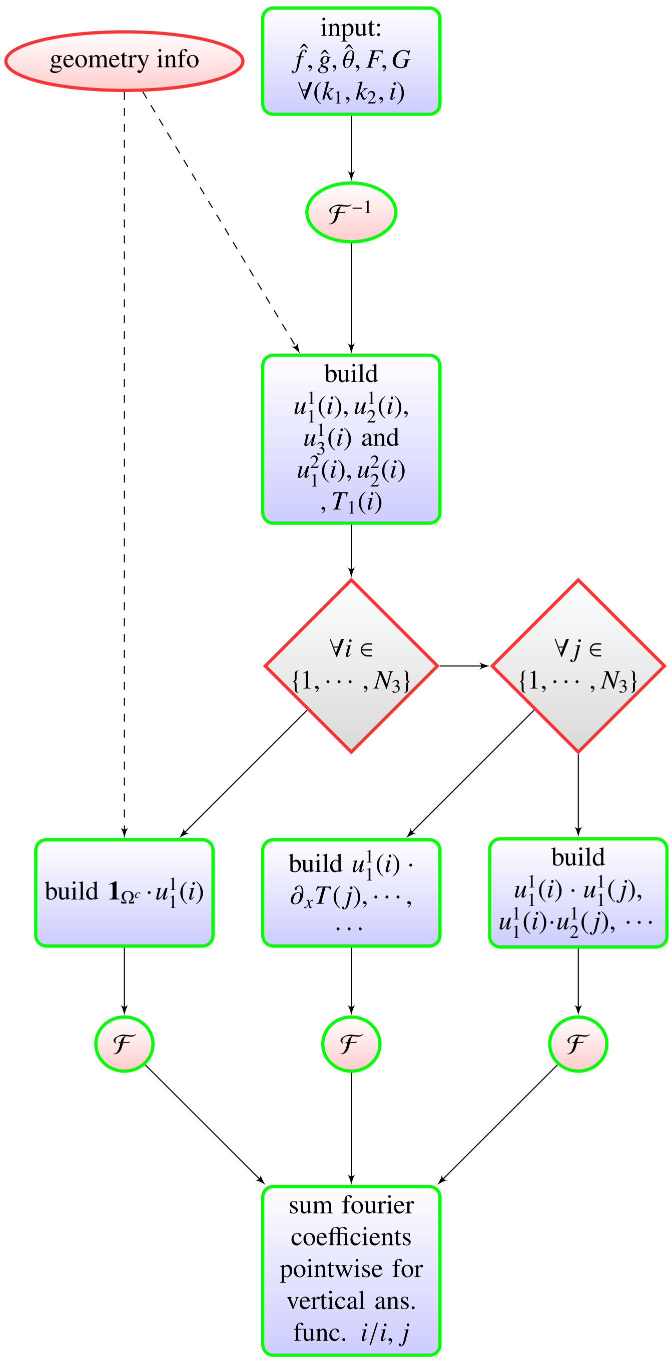
geometry info
input: $\hat{f}, \hat{g}, \hat{\theta}, F, G$ $\forall(k_1, k_2, i)$
$\mathcal{F}^{-1}$
build $u_1^1(i), u_2^1(i)$, $u_3^1(i)$ and $u_1^2(i), u_2^2(i)$ $, T_1(i)$
$\forall i \in \{1, \cdots, N_3\}$
$\forall j \in \{1, \cdots, N_3\}$
build $\mathbf{1}_{\Omega^c} \cdot u_1^1(i)$
build $u_1^1(i) \cdot \partial_x T(j), \cdots, \cdots$
build $u_1^1(i) \cdot u_1^1(j)$, $u_1^1(i) \cdot u_2^1(j), \cdots$
$\mathcal{F}$
$\mathcal{F}$
$\mathcal{F}$
sum fourier coefficients pointwise for vertical ans. func. $i/i, j$

Figure 3.3: Flowchart representing the evaluation of the nonlinear operator, including penalizations. States with a green lining are completely calculated within the GPU environment. Red ones are controlled by the host. i.e. the CPU.

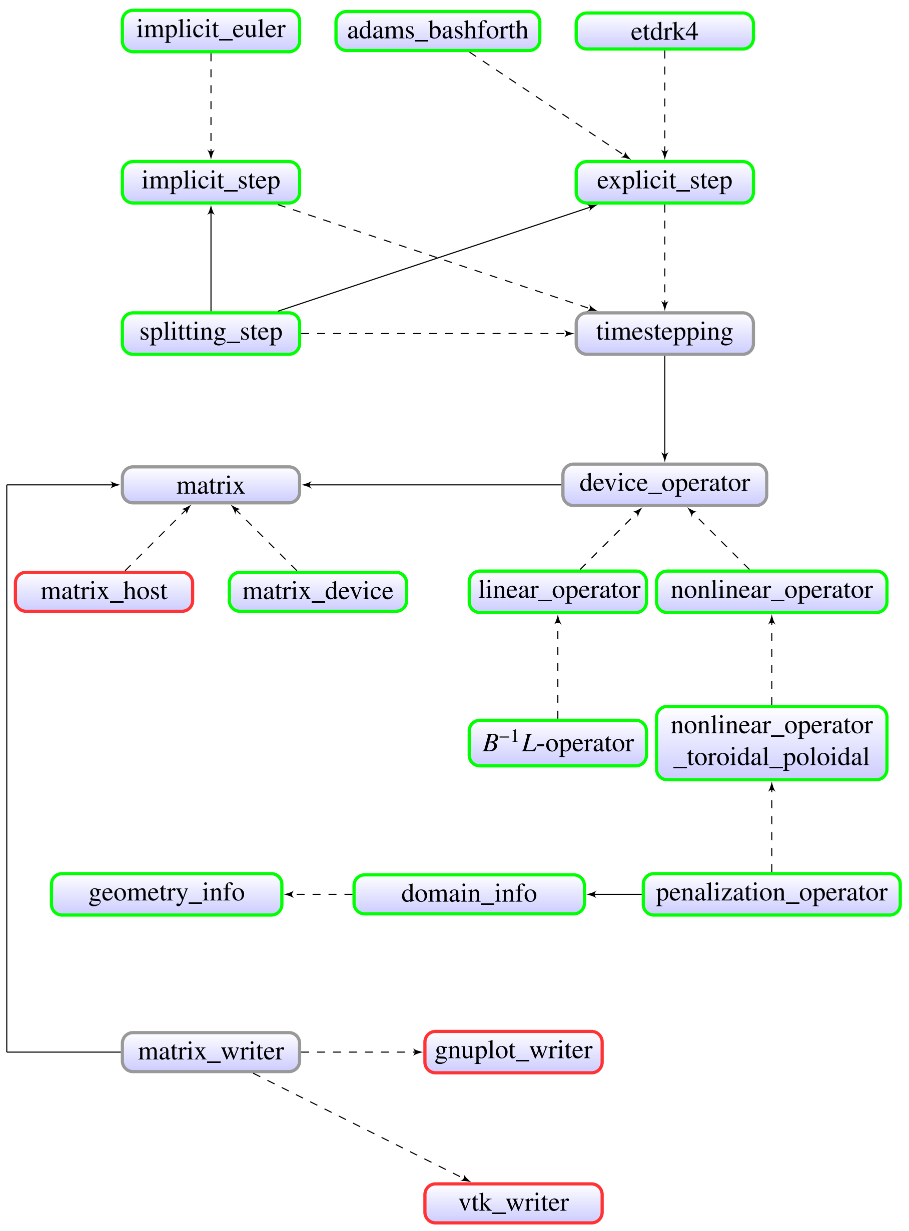

Figure 3.4: Class diagram of the simulation package. Classes with a green lining keep their data within the global memory of the GPU, while red linings indicate data storage within the RAM of the CPU. The dashed lines represent inheritance, while solid lines represent directed associations. Notice that except for input and output all operations are executed on the CUDA device. As moving data from host memory to device memory is a very slow operation, this bottleneck does not affect performance of the code except for input and output operations. The diagram shows only the most important classes.

will be a batched transformation (which speeds up repeated transformations of the same datasize). In a second step make use of the created transformation plan and apply transformation algorithms to the given data field. In-place and out-of-place algorithms are possible. In a third step destroy created CuFFT plans and make use of Fourier coefficients. Listing 3.4 gives you a short example how fast Fourier transformations with CuFFT can be implemented.

```
/*
* a function which needs the fourier transformed of f
*/
__host__ void a_host_function(matrix_device* f) {

 //create cufft plan
 int num_x = 2*(f->get_dimension().at(0) -1);
 int num_y = f->get_dimension().at(1);
 cufftPlan2d(&c2r_plan ,num_y,num_x,CUFFT_C2R);

 //execute the plan
 cufftExecC2R(c2r_plan, f->get_data() , f_physical);

 //destroy cufft plan
 cufftDestroy(c2r_plan);
}
```

Listing 3.4: Example of usage of CUDA's fast Fourier transformation library (CuFFT). The transformation process is separated into three steps: Create then execute and destroy the CuFFT plan. The transformation size is always a logical transformation size (in case of complex to real and real to complex transformations the number of samples in physical space).

It is worth mentioning that all CuFFT transformations (unlike most other fast Fourier transformation libraries) are not normalized. Therefore we work with correct Fourier coefficients in Fourier space and apply a normalization factor in physical space. The transformation factor in case of a two dimensional transformation is (where N_1^{real} denotes the number of columns of the Fourier coefficient matrix and N_2^{real} its rows):

$$\frac{1}{2(N_1^{real}-1)N_2^{real}}$$

This means that applying a backward transformation after a forward transformation to some data field will not deliver the same data field. CuFFT also supports streamwise execution to increase parallelization and can be executed in single or double precision.

CuFFT's maximum performance can be achieved, when two requirements are full filled (see [23]):

- data fits into CUDA's shared memory algorithm
- the logical data size is a power of a single factor

The maximum performance is achieved if data size is a power of two. In this case shared memory use is optimized for smaller sub-transformations, which increases memory bandwidth compared to global memory use. In cases of arbitrary transformation size a mixed radix FFT algorithm is applied which typically has lower performance and less accuracy (see [23]). Later CuFFT versions (> 3.0) implement special algorithms for real to complex transformations which reduce the number of computations and memory usage by a half. Until now CuFFT supports only a single GPU.

3.2.4 Verification

Of course the questions arises how to check whether any implementation is working correctly. Since there are nearly no analytic solutions known (at least within the interesting Rayleigh number range) it is not easy to check whether an implementation is correct or not. Therefore we chose to make use of computer algebra systems (in this case Mathematica) and apply the same operators B, L, N to some simple test cases and project the results on our vertical and horizontal ansatz functions. For example one can define T', f, g, F, G in an arbitrary way as a tensor product of Fourier series and Chandrasekhar functions. The same initial conditions are implemented in our fully numerical simulation. For example the Fourier coefficients of the nonlinear operator of f can be checked by calculating the following projections:

$$\frac{1}{L_x L_y}\int_0^{L_x}\int_0^{L_y} \delta\cdot((u\cdot\nabla)u)e^{\frac{2\pi i k_1 x}{L_x}}e^{\frac{2\pi i k_2 y}{L_y}}\psi_n(z)dxdydz$$

The results obtained by symbolic computations with Mathematica were compared to numerical results obtained by a simulation step. Numerical and symbolic results differ only by a value which is in the range of numerical accuracy of single or double precision. This procedure was realized for the complete set of equations, including linear operator L and nonlinear operator N and the inverse of the linear operator B as well.

This verification has shown, that all discrete equations and their implementations are correct. Furthermore it should be mentioned that rotational and translational invariance of the analytic equations is inherited to the discrete equations. Rotational invariance can be checked numerically by setting all Fourier coefficients to a specific value. In this case all operators have to deliver again the same value for all modes with the same wave number.

3.2.5 Distributed implementation

To be able to simulate even larger systems we chose to implement a distributed version of the spectral simulation code which uses multiple graphics cards. Of course this enables to you use multiple graphics cards per computer as well, as graphics cards in different computers connected by a network. The data transfer is realized by the so-called message passing interface (MPI).

Currently there are some limitations with respect to multi-GPU programming. First of all there is no support to copy data from one GPU to another one without using host memory, even if they are located in the same computer. A second problem is that each host thread is dedicated to exactly one GPU at a specific time point and vice versa GPU memory which is available to a host thread is not available to another one. These limitations restrict asynchronous data transfer between two different GPUs, since it is not possible to implement this by making use of multiple threads. Nevertheless a distributed implementation is still worth it as the system size increases. One of the most important questions is again the data management, of course. To clarify data flow, consider one time-step of $\hat{f}$: To compute $\hat{f}(k_1, k_2, n, t_{n+1})$ we have to calculate the linear and nonlinear operator. The first one requires knowledge of $\hat{f}(k_1,k_2,1,t_n)$, $\hat{f}(k_1,k_2,2,t_n)$ and $\hat{T}(k_1,k_2,1,t_n)$, $\hat{T}(k_1,k_2,2,t_n)$. The linear operator is therefore dependent on four other data entries, which have the same mode number with respect to the horizontal plane. The nonlinear operator requires to know more, since the fast Fourier transformation is a global operation. Therefore data organization should be adjusted in such a way, that each GPU keeps the data required to compute the linear operator and to reduce data transfer for Fourier transformations.

```
/*a function which is executed on the CPU (host)
* calculates a 2D-FFT of given input data
*/
__host__ void a_host_function(CUDA_FLOAT_REAL* data, int
   my_mpi_id, int number_of_mpi_processes) {

 //... calculate r2c 1D-FFT row-by-row by GPU
 ...
 //... copy the data to host
 ...

 //... distribute the data
 for(int i = 0; i < num_neighbours; i++){
   //... send process i
   MPI_Isend(mpiSendBufferRowOrdered + i*offset, offset
       *2, MPI_CUDA_FFT_DATATYPE, i, TAG, MPI_COMM_WORLD,
       send_request);
   //... recieve proces
   MPI_Irecv(mpiReceiveBufferColumnOrdered+i*offset ,
       offset*2, MPI_CUDA_FFT_DATATYPE, i, TAG,
       MPI_COMM_WORLD, receive_request));
 }

 //... copy back to GPU
 ...
 //... caculate c2c 1D-FFT column-by-column by GPU
 ...
}
```

Listing 3.5: A simple example of message passing interface to distribute data to different MPI processes. To increase performance and avoid deadlocks asynchronous data transfer is needed.

One easily verifies that a two dimensional Fourier transformation can be implemented by first transforming each row and then take this intermediate result and apply a column-by-column transformation to it. So each GPU handles a specific number of complete columns of the coefficient matrix in Fourier space and a specific number of complete rows of the coefficient matrix in physical space. By this realization it is guaranteed that at least one one-dimensional transformation can be executed without any data transfer to other GPUs. Figure 3.5 shows the procedure used to transform real data in physical space to complex data in Fourier space and Listing 3.5 shows how a CUDA-MPI implementation of a two-dimensional Fourier transformation looks like.

To verify correctness of our implementation we chose again to verify it by some simple test cases and compare results to the single GPU implementation (this is possible, since this has already been verified). The results obtained by the distributed implementation are the same as the ones obtained by the single GPU implementation regarding numerical accuracy of the system.

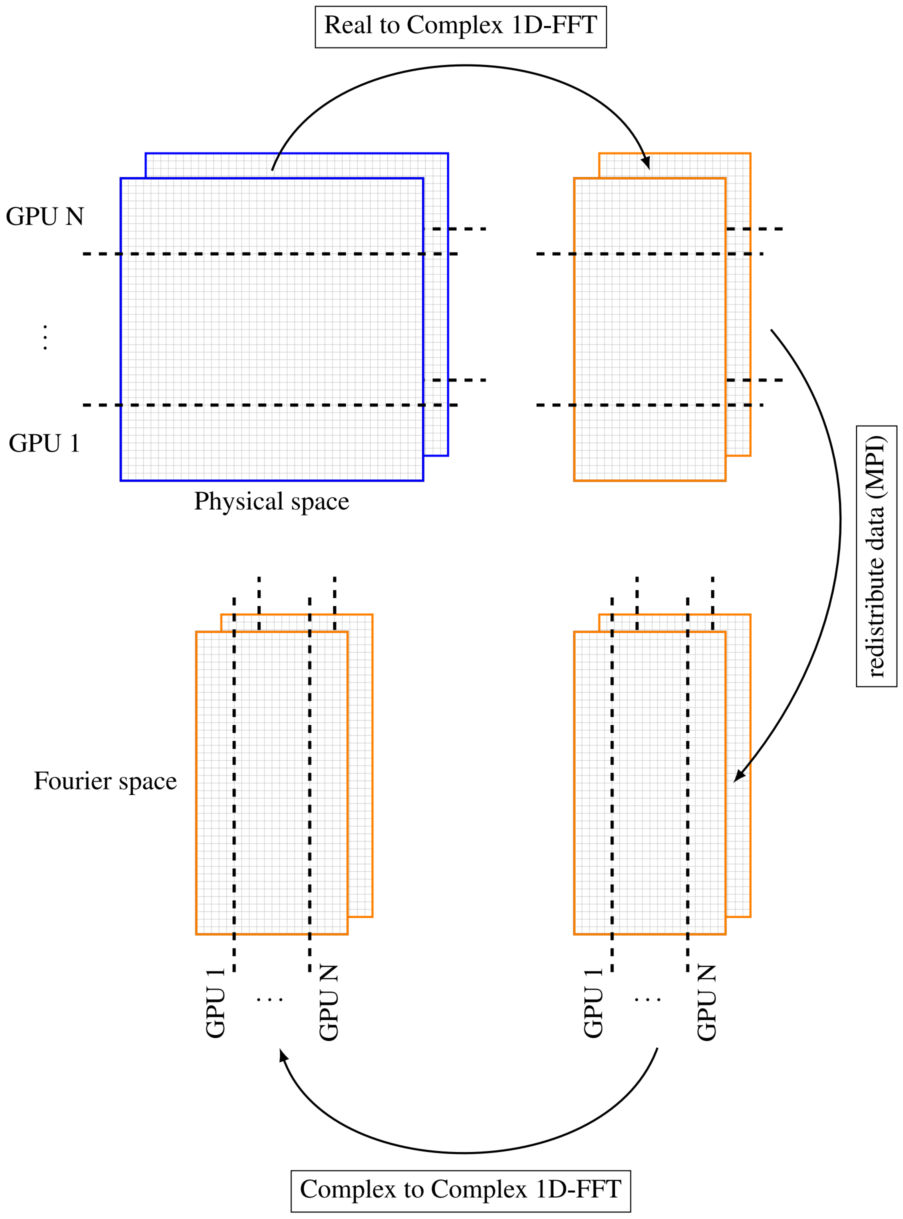

Figure 3.5: Memory management in a distributed system implementation. Notice that in real space (real data, double or single precision, indicated by the blue color) each GPU holds a complete row and in Fourier space each GPU holds a complete column (complex data, double or single precision, indicated by the orange color). The dimension of the Fourier coefficient matrix is roughly halved, but the Fourier coefficients are complex.

Chapter 4

Simulations

After verifying (as far as possible) that all simulation results are correct we have used numerical simulations to study some properties of spiral defect chaos.

Due to the fact, that memory of consumer graphics cards is limited, the system size is in general limited by the amount of global memory which is available. In our case we have used a GTX 470 (a card of NVIDIA's Fermi architecture) with 1280 mega bytes of memory. This allows us to simulate systems with $1024 \times 1024 \times 2$ degree of freedoms. In the following sections some physical results, as well as some performance considerations, are presented.

4.1 Onset of SDC

First of all we have studied onset of spiral defect chaos in detail. Since spatio-temporal chaotic states in Rayleigh-Bénard Convection are still poorly understood, it is still unclear which parameter controls onset of spiral defect chaos in a convection layer. Figure 4.3 shows some experimental results about onset of spiral defect chaos. One notices immediately that spiral defect chaos occurs for small reduced Rayleigh numbers in large aspect ratio systems, while for small aspect ratios a high reduced Rayleigh number is necessary. For $\Gamma \geq 70$ it is already found for $Ra_{reduced} \geq 0.2$ and for $\Gamma \leq 50$, $Ra_{reduced} \geq 0.6$ is needed. The transition takes place in a relatively sharp area around $\Gamma = 60$. It seems that these experimental results do not depend on the exact geometry of the fluid layer (with respect to both horizontal directions).

Therefore we decided to check the onset border of spiral defect chaos as a function of aspect ratio Γ for $Pr = 1$ at first. Based on experimental results (see Figure 4.3) we chose seven aspect ratios and for each aspect ratio again three Rayleigh numbers; one above, one below and one on the onset border of spiral defect chaos (based on Figure 4.3). To simulate a bounded cube with no-slip boundary conditions in all directions, we use the penalization method presented in section 2.1.3 with $\eta = 10^{-2}$.

Of course the question arises how to check if spiral defect chaos develops at a given parameter triple (Γ, Ra, P). The idea is to force the system to build spirals if it is possible for the given parameters by an appropriate choice of initial conditions. Therefore we use a special initial condition, straight convection rolls with two dislocations. This forces the convection pattern to create transverse rolls, which might create spirals and spiral defect chaos. An initial condition with two dislocations can be created (in case of a periodic domain) by selecting a specific mode $k = (k_1, k_2)$ and modulating its phase

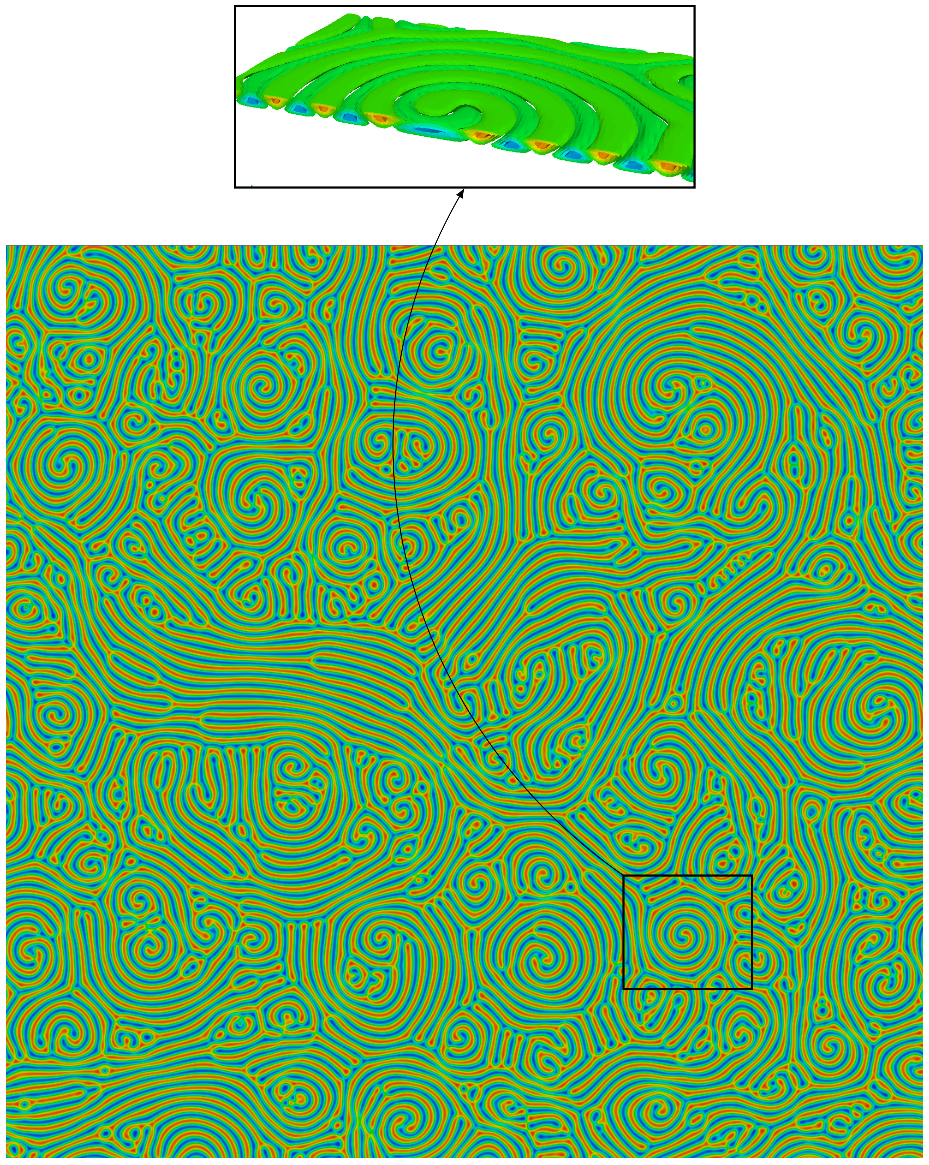

Figure 4.1: Simulation results in a convection cell with aspect ratio $\Gamma = 100$ at $t = 2000 t_\nu$. The Prandtl number is one and the Rayleigh number is 2900. The simulation used 1024×1024 modes in the horizontal plane. For a GTX 470 (current single GPU testing system) this is the maximum grid size. Simulations on GPUs with more global memory expand these limits and distributed simulations can even simulate large systems. The detail on top shows the structure of the convection rolls (isosurfaces of temperature perturbation with respect to the stationary state, blue indicates cold and red hot fluid).

(as seen in section 1.4 dislocations are singularities of phase's gradient),

$$exp\Big(i\big(k_1 \cdot x + k_2 \cdot y \pm \Phi((x,y)^T - (x^1,y^1)^T) \pm \Phi((x,y)^T - (x^2,y^2)^T)\big)\Big) \tag{4.1}$$

where (x^1, y^1) and (x^2, y^2) represent the places of both dislocations and Φ denotes argument of the complex plane to introduce a phase jump if $((x,y)^T - (x^i, y^i)) \to 0$.

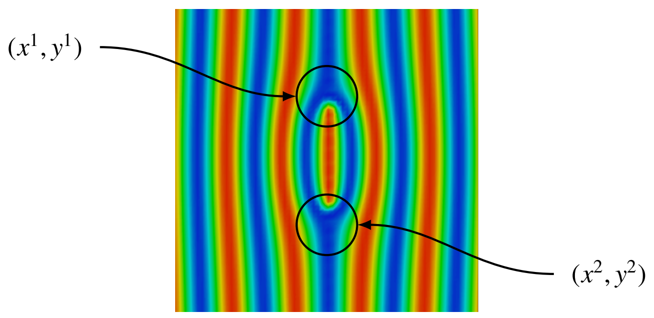

Figure 4.2: Initial condition to check if spiral defect chaos develops. Inserting two dislocations within the temperature field forces the convection pattern to develop transverse rolls between both dislocation points. If spiral defect chaos is not a transient state for the given parameters, the spiral will not vanish as time increases.

Figure 4.2 shows a small example of the initial conditions in a periodic box of aspect ratio $\Gamma = 10$. Of course the wavelength of the pattern is adapted to the given aspect ratio. The whole list of simulation parameters used to investigate onset of spiral defect chaos at $Pr = 1$ in a square cell is shown in Table 4.1.

Figure 4.12 shows some simulation results in a square cell with no-slip boundary conditions in all directions. It shows, that for $Ra = 2595$ and $Ra = 2664$ it is a transient state (if existent), but for $Ra = 2698$ spirals develop and advect through the cell. The simulation results are plotted at $t = 5000t_\nu$. Of course it is not completely clarified if spiral defect chaos is a transient state for larger Rayleigh numbers, too.

The complete set of simulation results with respect to onset of spiral defect chaos is presented in Figure 4.4. In our simulations spiral defect chaos developed only for aspect ratios larger than 20. The line is only a visual hint and the results are obtained for square cells. One easily notices again the tendency of lower Rayleigh numbers in high aspect ratio cells and the sharp bend of the onset boundary around aspect ratio 60.

4.2 Statistical properties of SDC

Almost all theories of turbulence are statistical ones. As we want to compare the spiral defect chaos state to two dimensional turbulent flow, one has to investigate comparable statistical properties of our convection problem. One of the workhorses of turbulence theory is the structure function of order p. Is is defined in the following way

$$S_p(r) =< |f(x+r) - f(x)|^p >$$

where f denotes a scalar field and $< \cdot >$ the so called ensemble average. In case of a simple periodic cube one can easily show that convection patterns and convection

aspect ratio (Γ)	Ra	Pr	Δt	modes	cell type	η
20	2596	1	0.005	256×256	square cell	0.01
	2681	1	0.005	256×256	square cell	0.01
	2732	1	0.005	256×256	square cell	0.01
40	2595	1	0.005	512×512	square cell	0.01
	2664	1	0.005	512×512	square cell	0.01
	2698	1	0.005	512×512	square cell	0.01
50	2562	1	0.005	512×512	square cell	0.01
	2647	1	0.005	512×512	square cell	0.01
	2698	1	0.005	512×512	square cell	0.01
60	2305	1	0.002	1024×1024	square cell	0.01
	2391	1	0.002	1024×1024	square cell	0.01
	2476	1	0.002	1024×1024	square cell	0.01
70	2066	1	0.002	1024×1024	square cell	0.01
	2135	1	0.002	1024×1024	square cell	0.01
	2220	1	0.002	1024×1024	square cell	0.01
80	2049	1	0.002	1024×1024	square cell	0.01
	2135	1	0.002	1024×1024	square cell	0.01
	2186	1	0.002	1024×1024	square cell	0.01
100	2015	1	0.002	1024×1024	square cell	0.01
	2100	1	0.002	1024×1024	square cell	0.01
	2186	1	0.002	1024×1024	square cell	0.01

Table 4.1: List of all simulation parameters used to investigate onset of spiral defect chaos as a function of aspect ratio Γ and Rayleigh number. The geometry is a square box with no-slip boundary conditions in all directions. η represents the penalization parameter.

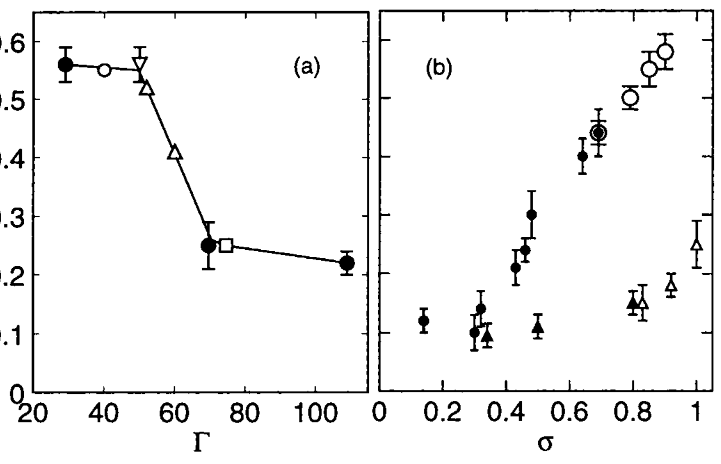

Figure 4.3: Reduced Rayleigh number at which SDC sets in as a function of Prandtl number (σ) and aspect ratio (Γ) ([6], page 751). The left diagram shows experimental results for $\sigma = 1$ and the right one for $\Gamma = 30$ (triangles) and $\Gamma = 70$ (circles) and open circles represent pure gas, while solid ones represent gas mixtures. The geometric shapes in the left diagram have to be interpreted in the following way: solid circles, $\Gamma = 29, 70, 109$ (Liu, Bajaj, Ahlers, unpublished); open circle, $\Gamma = 40$ (Ecke, Hu, 1997); triangle, $\Gamma = 50, 60$ (Hu, 1995); square, $\Gamma = 75$ (Morris, 1996); upside-down triangle, $\Gamma = 50$ square cell (Cakmur, 1997). The onset of SDC seems to be a function of the aspect ratio and does not depend on the geometry of the fluid layer ([6]).

properties are homogenous, since the governing equations are invariant with respect to rotations and translations. Therefore ensemble averages will equal volume averages and the structure function of order p can be rewritten in the following form:

$$S_p(r) = \frac{1}{V} \int_V |f(x+r) - f(x)|^p$$

where V is the observed volume.

4.2.1 Structure functions

In case of a two dimensional phenomenon the volume V has to be replaced by the interrogation area A. This is the case for spiral defect chaos, as the vertical dependence is not important and the chosen numerical approach uses only a small number of vertical ansatz functions. Therefore this numerical approach technique is not sufficient to study vertical statistical properties.

So we are studying two dimensional statistical properties of SDC. The interesting fields are therefore u_1, u_2, u_3 and the temperature field T'. To compute them numerically we take advantage of the Parseval-Plancherel-Theorem. The structure function of order p of the temperature field is given by:

$$S_p^{T'}(r) = \frac{1}{L_x L_y} \int_0^{L_x} \int_0^{L_y} |T'(x + r_x, y + r_y, z = 0) - T'(x, y, z = 0)|^p dxdy$$

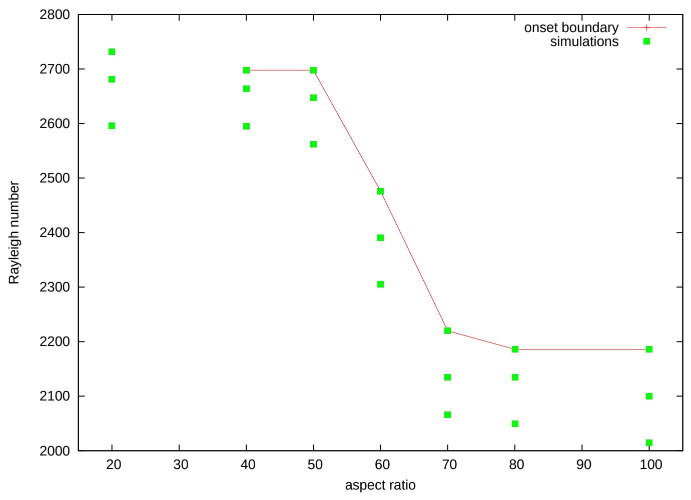

Figure 4.4: Onset of spiral defect chaos as a function of aspect ratio and Rayleigh number. The simulation parameters are described in Table 4.1. We have used initial condition 4.1 to check if spiral defect chaos develops. For $\Gamma = 20$ no spiral defect chaos can be obtained for the given Rayleigh numbers.

Taking advantage of the Fourier expansion of T' we write:

$$
\begin{aligned}
& T'(x + r_x, y + r_y, z) - T'(x, y, z) && (4.2)\\
&= \sum_{k_1=-N_1}^{N_1} \sum_{k_2=-N_2}^{N_2} \left(\sum_{k_3=1}^{N_3} \hat{T}'(k_1, k_2, k_3)\left(e^{\frac{2\pi i k_1 r_x}{L_x}} e^{\frac{2\pi i k_2 r_y}{L_y}} - 1\right) S_{k_3}(z) \right) e^{\frac{2\pi i k_1 x}{L_x}} e^{\frac{2\pi i k_2 y}{L_y}} && (4.3)\\
&= \sum_{k_1=-N_1}^{N_1} \sum_{k_2=-N_2}^{N_2} \hat{T'}^{z}_{r}(k_1, k_2) e^{\frac{2\pi i k_1 x}{L_x}} e^{\frac{2\pi i k_2 y}{L_y}} && (4.4)
\end{aligned}
$$

So we can rewrite this difference as a Fourier series. To build the integrand one has to transform the given series into physical space and apply $|\cdot|^p$ to this field. This field is again a periodic one with respect to the cube length in x and y direction. So we rewrite this new field by a Fourier series expansion:

$$
|T'(x + r_x, y + r_y, z) - T'(x, y, z)|^{\frac{p}{2}} = \sum_{k_1=-N_1}^{N_1} \sum_{k_2=-N_2}^{N_2} \hat{T'}^{z}_{r,\frac{p}{2}}(k_1, k_2) e^{\frac{2\pi i k_1 x}{L_x}} e^{\frac{2\pi i k_2 y}{L_y}}
$$

Taking advantage of the Parseval-Plancherel-Theorem and the upper Fourier series expansion the integral of the structure function can be rewritten as follows:

$$
\begin{aligned}
\frac{1}{L_x L_y}\int_0^{L_x}\int_0^{L_y} & |T'(x+r_x,y+r_y,z)-T'(x,y,z)|^{\frac{p}{2}}\overline{|T'(x+r_x,y+r_y,z)-T'(x,y,z)|^{\frac{p}{2}}} \\
&= \sum_{k_1=-N_1}^{N_1}\sum_{k_2=-N_2}^{N_2} \hat{T'}^{z}_{r,\frac{p}{2}}(k_1,k_2)\overline{\hat{T'}^{z}_{r,\frac{p}{2}}(k_1,k_2)} \\
&= \sum_{k_1=-N_1}^{N_1}\sum_{k_2=-N_2}^{N_2} |\hat{T'}^{z}_{r,\frac{p}{2}}(k_1,k_2)|^2
\end{aligned}
$$

In the same way one calculates the difference for first component of the velocity field:

$$
\begin{aligned}
& u_1(x+r_x,y+r_y,z)-u_1(x,y,z) \\
&= \sum_{k_1=-N_1}^{N_1}\sum_{k_2=-N_2}^{N_2}\Bigg(\sum_{k_3=1}^{N_3}\frac{2\pi i k_1}{L_x}\hat{f}(k_1,k_2,k_3)\big(e^{\frac{2\pi i k_1 r_x}{L_x}}e^{\frac{2\pi i k_2 r_y}{L_y}}-1\big)\partial_z\psi_{k_3}(z) \\
&+\frac{2\pi i k_2}{L_y}\hat{g}(k_1,k_2,k_3)\big(e^{\frac{2\pi i k_1 r_x}{L_x}}e^{\frac{2\pi i k_2 r_y}{L_y}}-1\big)S_{k_3}(z)\Bigg)e^{\frac{2\pi i k_1 x}{L_x}}e^{\frac{2\pi i k_2 y}{L_y}}
\end{aligned}
$$

Therefore one may write:

$$
|u_1(x+r_x,y+r_y,z)-u_1(x,y,z)|^{\frac{p}{2}} = \sum_{k_1=-N_1}^{N_1}\sum_{k_2=-N_2}^{N_2}\hat{u}^{z}_{1,r,\frac{p}{2}}(k_1,k_2)e^{\frac{2\pi i k_1 x}{L_x}}e^{\frac{2\pi i k_2 y}{L_y}}
$$

And by the same arguments:

$$
\begin{aligned}
\frac{1}{L_x L_y}\int_0^{L_x}\int_0^{L_y} & |u_1(x+r_x,y+r_y,z)-u_1(x,y,z)|^{\frac{p}{2}}\overline{|u_1(x+r_x,y+r_y,z)-u_1(x,y,z)|^{\frac{p}{2}}} \\
&= \sum_{k_1=-N_1}^{N_1}\sum_{k_2=-N_2}^{N_2}\hat{u}^{z}_{1,r,\frac{p}{2}}(k_1,k_2)\overline{\hat{u}^{z}_{1,r,\frac{p}{2}}(k_1,k_2)} \\
&= \sum_{k_1=-N_1}^{N_1}\sum_{k_2=-N_2}^{N_2}|\hat{u}^{z}_{1,r,\frac{p}{2}}(k_1,k_2)|^2
\end{aligned}
$$

Figure 4.5 shows structure functions of the velocity field at $z = 0$ (as a function of x, y) obtained by a numerical simulation of spiral defect chaotic state in a periodic domain of length 200 (i.e. $\Gamma = 100$). We have used $1024 \times 1024 \times 2$ modes. The structure function is shown for $10 \leq \frac{r \cdot N}{L} \leq 512$, since there is obviously a correlation for small $\frac{r \cdot N}{L}$ as the pattern is formed of convection rolls. Figure 4.6 shows the structure function of the temperature field at $z = 0$ for the same simulation parameters.

4.2.2 Energy spectrum

Another very important statistical quantity is the energy distribution with respect to the different scales or modes k. In two-dimensional, homogenous, isotropic turbulence there are different ranges of the energy spectrum in which you will find a power-law behaviour. Compared to three-dimensional turbulence the unique feature of two-dimensional turbulence is its inverse energy cascade, i.e. smaller scales transfer energy

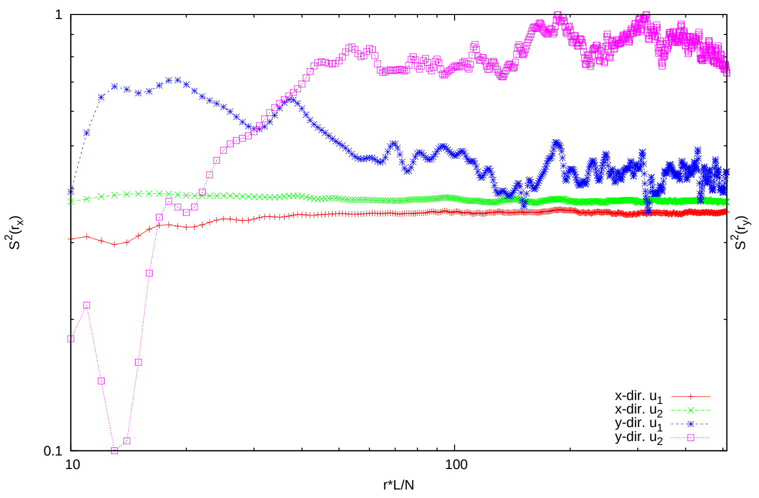

Figure 4.5: Second order structure function of velocity field. The data is obtained by a numerical simulation with $N = N_1 = N_2 = 1024, \Gamma = 100, L = L_x = L_y = 200, Pr = 1, Ra = 2900, t = 5000$ in a periodic domain for random initial conditions. The data is plotted for $r * L/N = 10, \cdots, 512$.

to larger ones. To compare spiral defect chaos to two-dimensional turbulence we study the energy spectrum of the velocity field at $z = 0$ (where $D_0(k)$ denotes the disk with radius k around 0) :

$$E(k) = \int_{\partial D_0(k)} |\widehat{u_{i,z=0}(k_1, k_2)}|^2 dk_1 dk_2 \tag{4.5}$$

In our periodic cube this means:

$$E(k) = \sum_{k^2 \le k_1^2 + k_2^2 < (k+1)^2} |\widehat{u_{i,z=0}(k_1, k_2)}|^2 \tag{4.6}$$

Figure 4.7 shows the energy spectrum of the horizontal velocity components at $z = 0$. The simulation parameters are still $N = N_1 = N_2 = 1024$, $Pr = 1, Ra = 2900$, $\Gamma = 100$ and the simulation domain is a periodic cube. In addition to the unstable modes at $k \approx 100$, modes with low mode number become active. These modes advect the spirals through the fluid layer and represent a mean flow with low wavenumber. Additionally Figure 4.8 shows that part of the amplitude of f and g that play a part in u_1 and u_2. A curve fit within $10 \le k \le 90$, $120 \le k \le 200$ and $250 \le k \le 300$ of type $c_1 k^{c_2}$ gave:

- for $10 \le k \le 90$: $c_1 = 2.2814 \cdot 10^{-3}, c_2 = -0.32284$
- for $120 \le k \le 200$: $c_1 = 64.6631, c_2 = -14.9294$
- for $250 \le k \le 300$: $c_1 = 118.3618, c_2 = -24.6229$

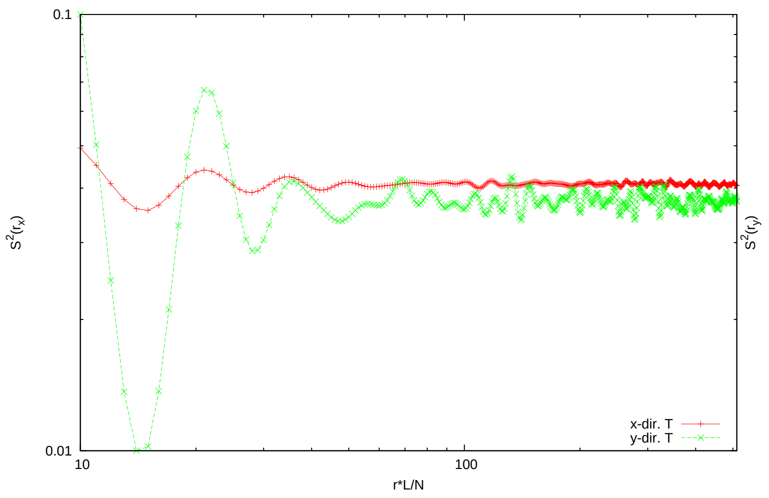

Figure 4.6: Second order structure function of temperature field. The data is obtained by a numerical simulation with $N = N_1 = N_2 = 1024, \Gamma = 100, L = L_x = L_y = 200, Pr = 1, Ra = 2900, t = 5000$ in a periodic domain for random initial conditions. The data is plotted for $r * L/N = 10, \cdots, 512$.

Furthermore we have considered the enstrophy at $z = const$:

$$\omega = \nabla \times (u_1, u_2)^T$$

By some simple algebraic manipulation one can show:

$$\omega = -\Delta_2 g$$

So we get an evolution equation for g and by taking advantage of divergence theorem and periodic boundary conditions we can show:

$$\partial_t \int_0^{L_x} \int_0^{L_y} \omega(x, y) dx dy = 0$$

Therefore we calculated the amplitude spectrum of the enstrophy, too. The results are shown in Figure 4.9. Two different regions with potential power-law behaviour can be seen. A curve fit of type $c_1 \cdot k^{c_2}$ for $1 \leq k \leq 30$ and $120 \leq k \leq 180$ gave the following coefficients:

- for $1 \leq k \leq 30$: $c_1 = 1.62253 \cdot 10^{-6}, c_2 = 2.00178$
- for $120 \leq k \leq 180$: $c_1 = 171365350 \cdot 10^6, c_2 = -8.35548$

For relatively low mode numbers $k \leq 30$ the fit has shown a k^2 scaling behaviour for the enstrophy amplitude spectrum. This suggests a k^0 scaling for the energy spectrum (shown in Figure 4.7, curve fit gives $c_2 = -0.32284$ within this range).

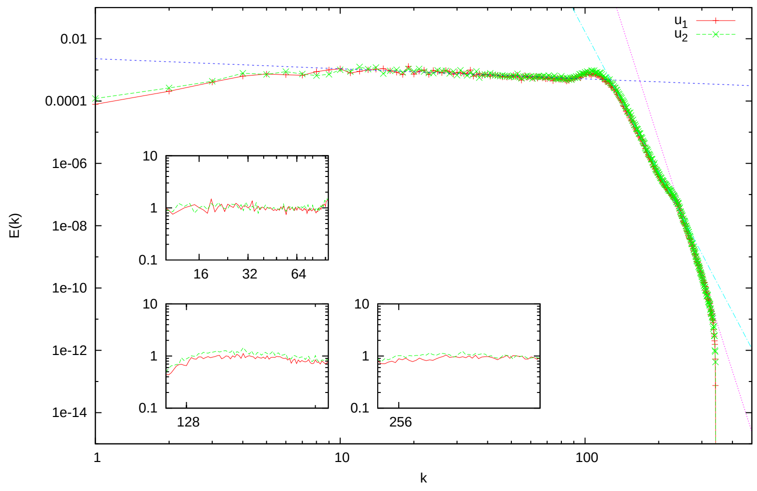

Figure 4.7: Energy spectrum of horizontal velocity component u_1, u_2 at $z = 0$ in log-log coordinates. The data is obtained by a numerical simulation with $N = N_1 = N_2 = 1024, \Gamma = 100, L = L_x = L_y = 200, Pr = 1, Ra = 2900, t = 5000$ in a periodic domain for random initial conditions. The subplots show the numerical data divided by the fitted data (starting from left top: power-law fit for $10 \le k \le 90$, $120 \le k \le 200$, $250 \le k \le 300$). The fitted curves are of $c_1 k^{c_2}$ type and the fitted coefficients are: $c_1 = 2.2814 \cdot 10^{-3}, c_2 = -0.32284$ for $k \in [10; 100]$; $c_1 = 64.6631, c_2 = -14.9294$ for $k \in [120; 200]$; $c_1 = 118.3618, c_2 = -24.6229$ for $k \in [250; 300]$

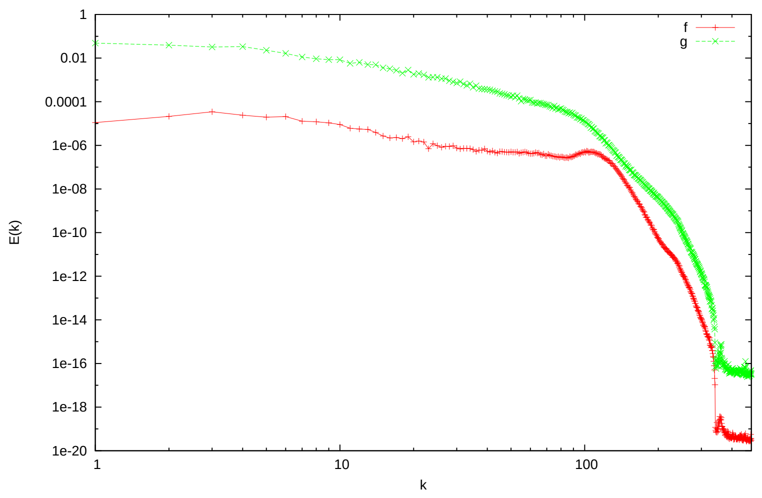

Figure 4.8: Parts of the amplitude spectrum of f and g that play a part in u_1, u_2 at $z = 0$ in log-log coordinates. The data is obtained by a numerical simulation with $N = N_1 = N_2 = 1024, \Gamma = 100, L = L_x = L_y = 200, Pr = 1, Ra = 2900, t = 5000$ in a periodic domain for random initial conditions.

4.3 Code performance and scalability

Currently the code offers the possibility to simulate systems with 1024×1024×2 modes (test system has 1.2 GB of global memory). So one can achieve very high aspect ratios. By making use of implicit explicit time-stepping methods step sizes up to $h = 0.05t_\nu$ are possible (independent of the number of modes, assuming that the CFL is fulfilled). A single time-step at maximum systemsize takes less than one-third of a second. This simulation speed points out performance of our code. Of course this highly related to data management within global memory space of the GPU. So increasing size of global memory will give access to large systems. Nevertheless you will reach a point where the number of degree of freedoms becomes to large for global memory. In such a situation one has to use our distributed implementation to interchange data between different GPUs.

In our testing system with two GPUs in a single machine (each of them having 1 GB global memory space) we are able to double system size in each direction, i.e. $2048 \times 2048 \times 2$ modes. In this situation each GPU handles roughly $1024 \times 2048 \times 2$ degree of freedoms and fast Fourier transformation is the only global operation which requires MPI activities. Of course this reduces performance and the performance benefit of GPUs compared to CPUs is reduced, as additional data management is required.

To review code performance we decided to execute several simulation tests. In order to check weak and strong scaling we used a GPU cluster consisting of 10 computing nodes, each of them having access to a S1070 Tesla GPU. As already mentioned there are two different scaling tests:

weak scaling: Considers runtime of a simulation with a fixed number of degree of

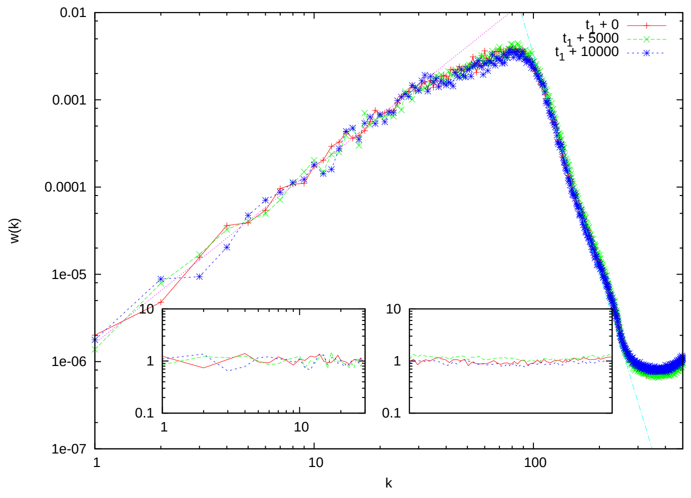

Figure 4.9: Amplitude spectrum of enstrophy $\omega = \nabla \times (u_1 u_2)^T$ at $z = 0$ in log-log coordinates. The data is obtained by a numerical simulation with $N = N_1 = N_2 = 1024, \Gamma = 100, L = L_x = L_y = 200, Pr = 1, Ra = 2900, t = 5000$ in a periodic domain for random initial conditions. The dashed lines represent a linear fit in log-log coordinates. The left curve is fitted for $1 \leq k \leq 30$ and the right one for $120 \leq k \leq 180$. The subplots show the numerical data divided by the fitted data. The fitted curves are of $c_1 k^{c_2}$ type and the fitted coefficients are: $c_1 = 1.62253 \cdot 10^{-6}, c_2 = 2.00178$ for $k \in [1; 30]$; $c_1 = 171365350 \cdot 10^6, c_2 = -8.35548$ for $k \in [120; 180]$

freedoms per GPU as the number of graphics cards is varied.

strong scaling: Considers runtime of a simulation with fixed system size as the number of graphics cards is varied.

In case of weak scaling we have used $1024 \times (N_{GPU} \cdot 1024) \times 2$ degree of freedoms for each computing node (where N_{GPU} denotes the number of GPUs) and used 2, 4, 6, 8 computing nodes. Figure 4.10 shows the test results. The results show a nice scaling behaviour, but this can be related to relatively small system size in our test cases (compared to the single GPU capabilities). Since copying from device to host memory and vice versa is an expensive operation, its relative cost with respect to a whole computing step decreases.

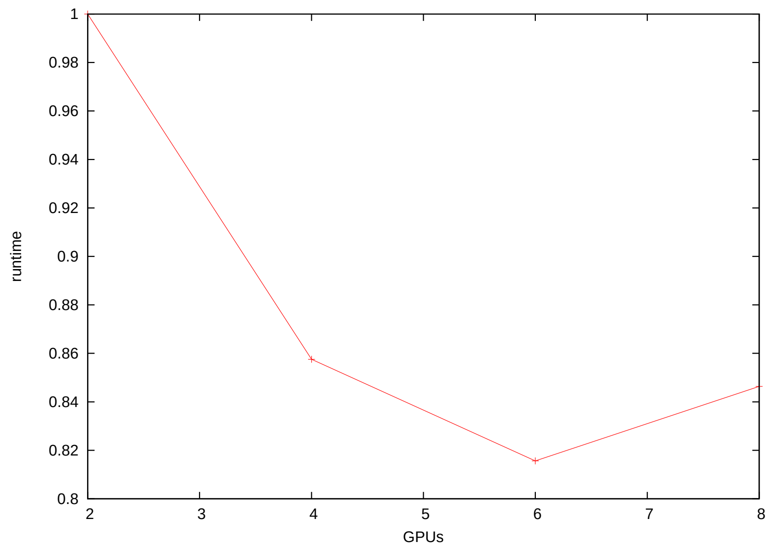

Figure 4.10: Weak scalability of distributed implementation for 2, 4, 6 and 8 GPUs, obtained by some tests in a GPU cluster with S1070 Tesla cards. The runtime is normalized with respect to the double GPU test.

The system size for strong scaling tests is $2048 \times 2048 \times 2$ and the tests were executed by 2, 4, 6, 8 GPUs, too.Figure 4.11 shows the results of the strong scalability test cases. All runtimes are normalized with respect to the double GPU test case. Again is not clear whether the system size influences the scaling behavior or not.

Of course our tests were limited in computing time and computing resources and so we were not able to run more tests and benchmarks to clarify scaling behavior exactly. Therefore both cases should be investigated a bit more in future to see which is the most efficient way to use multiple GPUs. One desirable goal would be to split out the complete runtime into three different parts: (i) data manipulation on GPU, (ii) copy data from GPU to host memory and vice versa, (iii) distribute data through PCI bus and/or interconnect. This would show which part of the application is the most time consuming one. In a future work these results can be needed to optimize code's performance to circumvent this bottleneck.

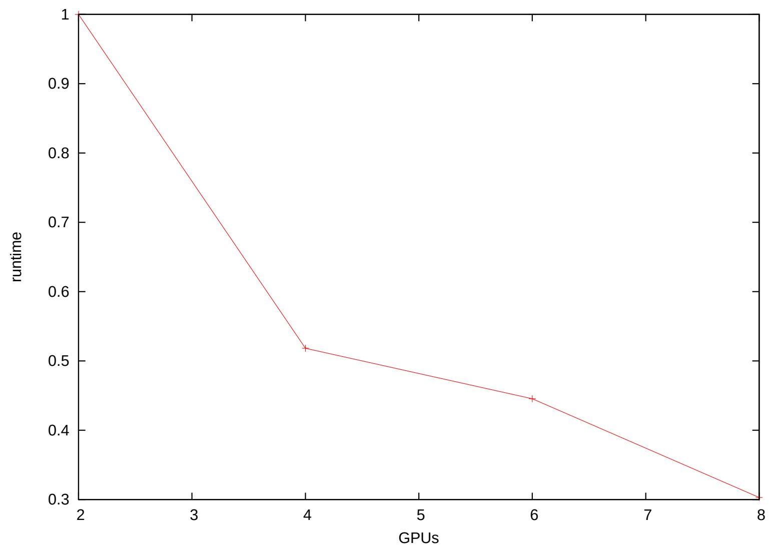

Figure 4.11: Strong scalability of distributed implementation for 2, 4, 6 and 8 GPUs, obtained by some tests in a GPU cluster with S1070 Tesla cards. The runtime is normalized with respect to the double GPU test.

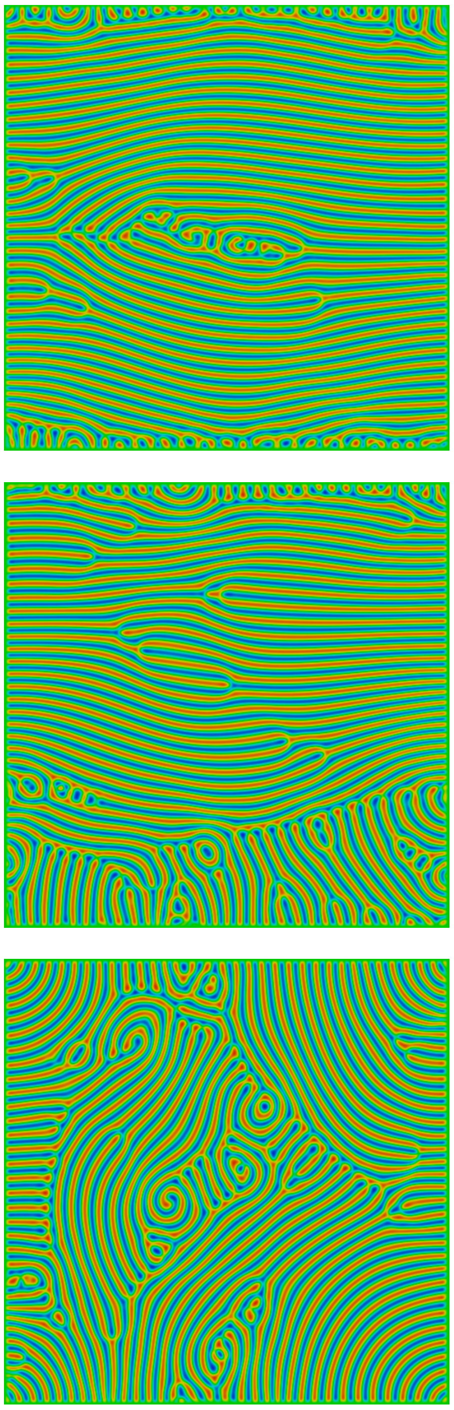

Figure 4.12: Numerical simulation of SDC onset for fixed aspect ratio $\Gamma = 40$ and different Rayleigh numbers ($Ra = 2595, 2664, 2698$) at $t = 5000t_\nu$ in a cell with no-slip boundaries. We have used parameters of Table 4.1. As you can see in the first two simulations SDC seems to be a transient state. The third simulation still evolves new spirals (even for larger times). This is the onset of SDC.

Chapter 5

Conclusions and outlook

This master thesis has shown how well pseudo-spectral methods perform on graphics cards. We have presented a way in which a simulation package can be implemented on GPUs and how it can be verified to ensure correct simulation results. Until now we can simulate large systems on a single graphics card and even larger systems on a distributed system. Considering the single graphics card implementation we can access system dimensions which have exclusively been accessible to large clusters before. Of course this is related to the massive parallelism of a GPU and the fact that no data has to be transferred to the RAM of the CPU. Up to now simulation dimensions are limited only by the size of global memory. So switching to another graphics card version like NVIDIA's Tesla series may again increase the dimension of the system which we can simulate. Since these cards have up to 8 GB of memory and our testing system has only 1.2 GB this should enable us to increase the system size by a large factor.

Furthermore we presented a way how to simulate systems which exceed the limits of a single graphics card's memory. We have implemented a distributed version of our simulation package which uses message passing interface in conjunction with CUDA to increase the performance. As shown all operation are nearly pointwise, except the fast Fourier transformation. Therefore we apply a simple version of a distributed CUDA-MPI fast Fourier transformation which uses row-by-row and column-by-column, one-dimensional transformations to implement two-dimensional transformations. The complete implementation has been verified in the same way as the single graphics cards implementation was. Of course this data transfer reduces the average computing performance of the system per time.

We have seen that there is a number of patterns which can be simulated quite well, of course one of them is the spiral defect chaos state (which was one of the goals of this thesis). This enables us to study its statistical properties in detail. It is only limited by the computing time and performance. Nevertheless this simulation package offers better insight into the complete dynamic of the system. Some of them have already been investigated in this thesis. At first we have studied onset of sprial defect chaos as function of aspect ratio and Rayleigh number. We have verified several experimental results for a specific cell geometry (in this case a simple square). Furthermore we have investigated some simple statistical properties like structure functions and energy spectra.

To simulate non-periodic, bounded domains with no-slip boundary conditions in all directions we have employed a so-called penalization method. This method introduces an additional penalization term instead of changing the complete numerical simulation schema (including its simulation domain). This method works quite well in our cases and the penalization parameter is not too small and causes in general not too many problems. We have shown that this method allows as to simulate rather complex geometries like an array of disks or any other geometry.

Of course there is still a number of open questions. We want to mention some of them here, since they may be a starting point for some future work. In general we can separate them into two different sections. The first one is considering the parallelization on multiple GPUs. Until now asynchronous data transfer is possible for CUDA and MPI. Since we use them in conjunction and there are still some limitations of CUDA and the number of host threads, it is rather time-consuming to implement a complete asynchronous data transfer covering the complete chain of data transfer from one GPU to another. This will be available with the upcoming 4.0 release of CUDA and any MPI implementation by accessing the same GPU data with multiple threads. Furthermore there are some performance modifications which can be implemented to increase performance of the distributed two-dimensional fast Fourier transformation. This will increase the performance of the complete system and allows to run simulations faster. So this topic covers the implementation of two-dimensional fast Fourier transformations on GPUs and is directed to scientific computing.

The second topic covers the physical interpretation of spiral defect chaos. Since there has only been a limited amount of time available to study statistical properties of spiral defect chaos after implementing the algorithms on a GPU, there is still a large number of topics in that area. Of course one would be to investigate of any kind of cascade further in detail: For example to study the influence of no-slip boundaries (in the horizontal plane) on statistical properties like structure functions or energy spectra and to clarify isotropy by considering the SO_2-decomposition. Another topic can be to compute $\int \hat{u}\widehat{u \cdot \nabla u} dk$ analogue to two dimensional turbulent flow (, which generates the energy cascade there in case of homogenous, isotropic turbulence).

Furthermore this simulation package offers the possibility to study new patterns in complex geometries. To introduce new geometries one only has to implement a new mask (which represents the penalization term in the algorithm). Another possibility would be to increase the number of vertical ansatz functions or to replace them by an arbitrary number of Tschebychev polynomials to be able to simulate convection at higher Rayleigh numbers (i.e. extending the code from weak turbulent to fully turbulent domain).

Particle distribution and movement can be another interesting topic. One can study the influence of spiral-defect chaos to the distribution of particles or a passive scalar field as a function time. Of course it is possible to seed the fluid with particles of a specific mass or without mass. The influence of spirals on the particle distribution with respect to the large scale is not clear.

All in all this simulation package offers many further possibilities and shows how well a pseudo-spectral method is performing on a GPU. Relatively big simulations are now possible on "small" machines. In the near future performance of GPUs will still increase even more, so that calculations on GPUs will become more and more important. Our pseudo-spectral simulation is one of the well performing examples on GPUs.

Appendix A

Discrete equations for Rayleigh-Bénard Convection

The following set of discrete equations can be obtained by a Galerkin ansatz in all directions (taking two ansatz functions in vertical direction and an arbitrary number in both horizontal direction). The basic Boussinesq equations are:

$$\frac{1}{Pr}\left(\frac{\partial}{\partial t}\Delta\Delta_2 f + \delta \cdot div(u \otimes u)\right) = -Ra\Delta_2 T' + \Delta^2\Delta_2 f \tag{A.1}$$

$$\frac{1}{Pr}\left(\frac{\partial}{\partial t}\Delta_2 g + \epsilon \cdot div(u \otimes u)\right) = \Delta\Delta_2 g \tag{A.2}$$

$$\frac{\partial T'}{\partial t} + \Delta_2 f + (u \cdot \nabla)T' = \Delta T' \tag{A.3}$$

$$\frac{1}{Pr}\left(\frac{\partial F}{\partial t} + \frac{\partial}{\partial z} < u_1 u_3 >^{xy}\right) = \frac{\partial^2}{\partial z^2}F \tag{A.4}$$

$$\frac{1}{Pr}\left(\frac{\partial G}{\partial t} + \frac{\partial}{\partial z} < u_1 u_3 >^{xy}\right) = \frac{\partial^2}{\partial z^2}G \tag{A.5}$$

The linear part of this differential equation system is given by the following equations. For T' we get:

$$\begin{aligned}
&\frac{1}{L_xL_y}\int_{-\frac{1}{2}}^{\frac{1}{2}}\int_0^{L_y}\int_0^{L_x}\partial_t T'\overline{e^{\frac{2\pi i k_1 x}{L_x}}e^{\frac{2\pi i k_2 y}{L_y}}S_n(z)}\\
&=\frac{1}{2}\partial_t\hat{T}'_{k_1,k_2,n}(t)\\
&=\frac{-(2\pi)^2|\frac{k}{L}|^2-n^2\pi^2}{2}\hat{T}'_{k_1,k_2,n}-\sum_{i=1}^{N_3}<C_i,S_n>\Big(-(2\pi)^2|\frac{k}{L}|^2\Big)\hat{f}_{k_1,k_2,i}
\end{aligned}\tag{A.6}$$

For f we get:

$$\begin{aligned}
&\frac{1}{L_xL_y}\int_{-\frac{1}{2}}^{\frac{1}{2}}\int_0^{L_y}\int_0^{L_x}\frac{1}{Pr}\frac{\partial}{\partial t}\Delta\Delta_2 f\overline{e^{\frac{2\pi i k_1 x}{L_x}}e^{\frac{2\pi i k_2 y}{L_y}}\psi_n(z)}\\
&=-(2\pi)^2|\frac{k}{L}|^2\cdot\frac{-(2\pi)^2|\frac{k}{L}|^2\partial_t\hat{f}_{k_1,k_2,n}+\sum_{i=1}^{N_3}<C_i'',C_n>\partial_t\hat{f}_{k_1,k_2,i}}{Pr}\\
&=Ra(2\pi)^2|\frac{k}{L}|^2\Big(\sum_{i=1}^{N_3}<S_i,\psi_n>\hat{T}'_{k_1,k_2,i}\Big)-(2\pi)^2|\frac{k}{L}|^2\Big((\lambda_n+(2\pi)^4|\frac{k}{L}|^4)\hat{f}_{k_1,k_2,n}\\
&\quad-2\cdot(2\pi)^2|\frac{k}{L}|^2\sum_{i=1}^{N_3}<\psi_i'',\psi_n>\hat{f}_{k_1,k_2,i}\Big)
\end{aligned}\tag{A.7}$$

For g we get:

$$\begin{aligned}
&\frac{1}{L_xL_y}\int_{-\frac{1}{2}}^{\frac{1}{2}}\int_0^{L_y}\int_0^{L_x}\Big(\frac{1}{Pr}\partial_t\Delta_2 g\Big)\overline{e^{\frac{2\pi i k_1 x}{L_x}}e^{\frac{2\pi i k_2 y}{L_y}}S_n(z)}\\
&=-(2\pi)^2|\frac{k}{L}|^2\cdot\frac{1}{Pr}\cdot\frac{1}{2}\partial_t\hat{g}_{k_1,k_2,n}=-(2\pi)^2|\frac{k}{L}|^2\cdot\frac{-(2\pi)^2|\frac{k}{L}|^2-n^2\pi^2}{2}\hat{g}_{k_1,k_2,n}
\end{aligned}\tag{A.8}$$

For F we get:

$$\begin{aligned}
\int_{-\frac{1}{2}}^{\frac{1}{2}}\Big(\frac{1}{Pr}\partial_t F\Big)\overline{S_n(z)}&=\frac{1}{Pr}\cdot\frac{1}{2}\partial_t\hat{F}_n\\
&=\frac{-n^2\pi^2}{2}\hat{F}_n
\end{aligned}\tag{A.9}$$

For G we get:

$$\begin{aligned}
&\int_{-\frac{1}{2}}^{\frac{1}{2}}\Big(\frac{1}{Pr}\partial_t G\Big)\overline{S_n(z)}\\
&=\frac{1}{Pr}\cdot\frac{1}{2}\partial_t\hat{G}_n=\frac{-n^2\pi^2}{2}\hat{G}_n
\end{aligned}\tag{A.10}$$

Written as a system of differential equations the linear part becomes (for $\hat{T}', \hat{f}$):

$$B \cdot \partial_t V = L \cdot V \tag{A.11}$$

Where B, L, V denote the following matrices:

$$B = \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 & \dots & 0 \\ 0 & \ddots & 0 & \vdots & & \vdots \\ 0 & 0 & \frac{1}{2} & 0 & \dots & 0 \\ 0 & 0 & 0 & \frac{-(2\pi)^2|\frac{k}{L}|^2}{Pr}\Big(-(2\pi)^2|\frac{k}{L}|^2 + <\psi_1'', \psi_1>\Big) & \dots & \frac{-(2\pi)^2|\frac{k}{L}|^2}{Pr} <\psi_{N_3}'', \psi_1> \\ 0 & 0 & 0 & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \frac{-(2\pi)^2|\frac{k}{L}|^2}{Pr} <\psi_1'', \psi_{N_3}> & \dots & \frac{-(2\pi)^2|\frac{k}{L}|^2}{Pr}\Big(-(2\pi)^2|\frac{k}{L}|^2 + <\psi_{N_3}'', \psi_{N_3}>\Big) \end{pmatrix} \quad V = \begin{pmatrix} \hat{T}'_{k_1,k_2,1} \\ \vdots \\ \hat{T}'_{k_1,k_2,N_3} \\ \hat{f}_{k_1,k_2,1} \\ \vdots \\ \hat{f}_{k_1,k_2,N_3} \end{pmatrix} \tag{A.12}$$

$$L = \begin{pmatrix} \frac{-(2\pi)^2|\frac{k}{L}|^2 - 1^2\pi^2}{2} & 0 & 0 & (2\pi)^2|\frac{k}{L}|^2 <\psi_1, S_1> & \dots & (2\pi)^2|\frac{k}{L}|^2 <\psi_{N_3}, S_1> \\ 0 & \ddots & 0 & \vdots & \ddots & \vdots \\ 0 & 0 & \frac{-(2\pi)^2|\frac{k}{L}|^2 - N_3^2\pi^2}{2} & (2\pi)^2|\frac{k}{L}|^2 <\psi_1, S_{N_3}> & \dots & (2\pi)^2|\frac{k}{L}|^2 <\psi_{N_3}, S_{N_3}> \\ Ra(2\pi)^2|\frac{k}{L}|^2 <S_1, \psi_1> & \dots & Ra(2\pi)^2|\frac{k}{L}|^2 <S_{N_3}, \psi_1> & -(2\pi)^2|\frac{k}{L}|^2\Big(\lambda_1^4 + (2\pi)^4|\frac{k}{L}|^4 - 2\cdot(2\pi)^2|\frac{k}{L}|^2 <\psi_1'', \psi_1>\Big) & \dots & 2\cdot(2\pi)^4|\frac{k}{L}|^4 <\psi_{N_3}'', \psi_1> \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ Ra(2\pi)^2|\frac{k}{L}|^2 <S_1, \psi_{N_3}> & \dots & Ra(2\pi)^2|\frac{k}{L}|^2 <S_{N_3}, \psi_{N_3}> & 2\cdot(2\pi)^4|\frac{k}{L}|^4 <\psi_1'', \psi_{N_3}> & \dots & -(2\pi)^2|\frac{k}{L}|^2\Big(\lambda_{N_3}^4 + (2\pi)^4|\frac{k}{L}|^4 - 2\cdot(2\pi)^2|\frac{k}{L}|^2 <\psi_{N_3}'', \psi_{N_3}>\Big) \end{pmatrix} \tag{A.13}$$

-1cm.5cm

The nonlinear operators are the advective derivative of the momentum and temperature equations:

$$
\begin{aligned}
&div(u \otimes u) \cdot (1,0,0)^T \\
&= \partial_x \sum_{n,m=1}^{N_3} \psi'_n \psi'_m u_1^1(n) u_1^1(m) + 2\psi'_n S_m u_1^1(n) u_1^2(m) + S_n S_m u_1^2(n) u_1^2(m) \\
&\quad + \partial_y \sum_{n,m=1}^{N_3} \psi'_n \psi'_m u_2^1(n) u_1^1(m) + \psi'_n S_m u_2^1(n) u_1^2(m) + S_n \psi'_m u_2^2(n) u_1^1(m) \\
&\quad + S_n S_m u_2^2(n) u_1^2(m) \\
&\quad + \partial_z \sum_{n,m=1}^{N_3} \psi_n \psi'_m u_3^1(n) u_1^1(m) + \psi_n S_m u_3^1(n) u_1^2(m) \\
&div(u \otimes u) \cdot (0,1,0)^T \\
&= \partial_x \sum_{n,m=1}^{N_3} \psi'_n \psi'_m u_1^1(n) u_2^1(m) + \psi'_n S_m u_1^2(m) u_2^1(n) + S_n \psi'_m u_1^1(m) u_2^2(n) \\
&\quad + S_n S_m u_1^2(n) u_2^2(m) \\
&\quad + \partial_y \sum_{n,m=1}^{N_3} \psi'_n \psi'_m u_2^1(n) u_2^1(m) + 2\psi'_n S_m u_2^1(n) u_2^2(m) + S_n S_m u_2^2(n) u_2^2(m) \\
&\quad + \partial_z \sum_{n,m=1}^{N_3} \psi_n \psi'_m u_3^1(n) u_2^1(m) + \psi_n S_m u_3^1(n) u_2^2(m) \\
&div(u \otimes u) \cdot (0,0,1)^T \\
&= \partial_x \sum_{n,m=1}^{N_3} \psi'_n \psi_m u_1^1(n) u_3^1(m) + S_n \psi_m u_1^2(n) u_3^1(m) \\
&\quad + \partial_y \sum_{n,m=1}^{N_3} \psi'_n \psi_m u_2^1(n) u_3^1(m) + S_n \psi_m u_2^2(n) u_3^1(m) \\
&\quad + \partial_z \sum_{n,m=1}^{N_3} \psi_n \psi_m u_3^1(n) u_3^1(m)
\end{aligned}
\tag{A.14}
$$

The projection of the nonlinear part of θ is given by:

$$\begin{aligned}
&\frac{1}{L_x L_y}\int_{-\frac{1}{2}}^{\frac{1}{2}}\int_0^{L_y}\int_0^{L_x}[u\cdot\nabla\theta]\overline{e^{\frac{2\pi i k_1 x}{L_x}}e^{\frac{2\pi i k_2 y}{L_y}}S_n(z)}\\
&\quad=\sum_{i,j=1}^{N_3} < (\psi_i' S_j), S_n > \mathcal{F}_{x,y}\{u_1^1(i)(\partial_x T_1(j))\}_{k_1,k_2}\\
&\quad+\sum_{i,j=1}^{N_3} < (S_i S_j), S_n > \mathcal{F}_{x,y}\{u_1^2(i)(\partial_x T_1(j))\}_{k_1,k_2}\\
&\quad+\sum_{i,j=1}^{N_3} < (\psi_i' S_j), S_n > \mathcal{F}_{x,y}\{u_2^1(i)(\partial_y T_1(j))\}_{k_1,k_2}\\
&\quad+\sum_{i,j=1}^{N_3} < (S_i S_j), S_n > \mathcal{F}_{x,y}\{u_2^2(i)(\partial_y T_1(j))\}_{k_1,k_2}\\
&\quad+\sum_{i,j=1}^{N_3} < (\psi_i S_j'), S_n > \mathcal{F}_{x,y}\{u_3^1(i)(T_1(j))\}_{k_1,k_2}
\end{aligned}\tag{A.15}$$

Calculate the projections where $\mathcal{F}_{x,y}$ denotes the two dimensional Fourier-Transformation with respect to the x-y plane:

$$
\begin{aligned}
&\frac{1}{L_xL_y}\int_{-\frac{1}{2}}^{\frac{1}{2}}\int_0^{L_y}\int_0^{L_x}\delta\cdot[div(u\otimes u)]\overline{e^{\frac{2\pi ik_1x}{L_x}}e^{\frac{2\pi ik_2y}{L_y}}\psi_n(z)}\\
=&-\frac{(2\pi k_1)^2}{L_x^2}\Bigg(\sum_{i,j=1}^{N_3}<\partial_z(\psi_i'\psi_j'),\psi_n>\mathcal{F}_{x,y}\{u_1^1(i)u_1^1(j)\}_{k_1,k_2}\\
&+2<\partial_z(\psi_i'S_j),\psi_n>\mathcal{F}_{x,y}\{u_1^1(i)u_1^2(j)\}_{k_1,k_2}+<\partial_z(S_iS_j),\psi_n>\mathcal{F}_{x,y}\{u_1^2(i)u_1^2(j)\}_{k_1,k_2}\Bigg)\\
&-\frac{k_1k_2(2\pi)^2}{L_xL_y}\Bigg(\sum_{i,j=1}^{N_3}<\partial_z(\psi_i'\psi_j'),\psi_n>\mathcal{F}_{x,y}\{u_2^1(i)u_1^1(j)\}_{k_1,k_2}\\
&+<\partial_z(\psi_i'S_j),\psi_n>\mathcal{F}_{x,y}\{u_2^1(i)u_1^2(j)\}_{k_1,k_2}\\
&+<\partial_z(S_i\psi_j'),\psi_n>\mathcal{F}_{x,y}\{u_2^2(i)u_1^1(j)\}_{k_1,k_2}+<\partial_z(S_iS_j),\psi_n>\mathcal{F}_{x,y}\{u_2^2(i)u_1^2(j)\}_{k_1,k_2}\Bigg)\\
&+\frac{2\pi ik_1}{L_x}\Bigg(\sum_{i,j=1}^{N_3}<\partial_z^2(\psi_i\psi_j'),\psi_n>\mathcal{F}_{x,y}\{u_3^1(i)u_1^1(j)\}_{k_1,k_2}+<\partial_z^2(\psi_iS_j),\psi_n>\mathcal{F}_{x,y}\{u_3^1(i)u_1^2(j)\}_{k_1,k_2}\Bigg)\\
&-\frac{k_1k_2(2\pi)^2}{L_xL_y}\Bigg(\sum_{i,j=1}^{N_3}<\partial_z(\psi_i'\psi_j'),\psi_n>\mathcal{F}_{x,y}\{u_1^1(i)u_2^1(j)\}_{k_1,k_2}\\
&+<\partial_z(\psi_i'S_j),\psi_n>\mathcal{F}_{x,y}\{u_1^2(j)u_2^1(i)\}_{k_1,k_2}\\
&<\partial_z(S_i\psi_j'),\psi_n>\mathcal{F}_{x,y}\{u_1^1(j)u_2^2(i)\}_{k_1,k_2}+<\partial_z(S_iS_j),\psi_n>\mathcal{F}_{x,y}\{u_1^2(i)u_2^2(j)\}_{k_1,k_2}\Bigg)\\
&-\frac{(2\pi k_2)^2}{L_y^2}\Bigg(\sum_{i,j=1}^{N_3}<\partial_z(\psi_i'\psi_j'),\psi_n>\mathcal{F}_{x,y}\{u_2^1(i)u_2^1(j)\}_{k_1,k_2}\\
&+2<\partial_z(\psi_i'S_j),\psi_n>\mathcal{F}_{x,y}\{u_2^1(i)u_2^2(j)\}_{k_1,k_2}+<\partial_z(S_iS_j),\psi_n>\mathcal{F}_{x,y}\{u_2^2(i)u_2^2(j)\}_{k_1,k_2}\Bigg)\\
&+\frac{2\pi ik_2}{L_y}\Bigg(\sum_{i,j=1}^{N_3}<\partial_z^2(\psi_i\psi_j'),\psi_n>\mathcal{F}_{x,y}\{u_3^1(i)u_2^1(j)\}_{k_1,k_2}\\
&+<\partial_z^2(\psi_iS_j),\psi_n>\mathcal{F}_{x,y}\{u_3^1(i)u_2^2(j)\}_{k_1,k_2}\Bigg)\\
&+\frac{(2\pi)^3ik_1|\frac{k}{L}|^2}{L_x}\Bigg(\sum_{i,j=1}^{N_3}<(\psi_i'\psi_j),\psi_n>\mathcal{F}_{x,y}\{u_1^1(i)u_3^1(j)\}_{k_1,k_2}\\
&+<(S_i\psi_j),\psi_n>\mathcal{F}_{x,y}\{u_1^2(i)u_3^1(j)\}_{k_1,k_2}\Bigg)\\
&+\frac{(2\pi)^3ik_2|\frac{k}{L}|^2}{L_y}\Bigg(\sum_{i,j=1}^{N_3}<(\psi_i'\psi_j),\psi_n>\mathcal{F}_{x,y}\{u_2^1(i)u_3^1(j)\}_{k_1,k_2}\\
&+<(S_i\psi_j),\psi_n>\mathcal{F}_{x,y}\{u_2^2(i)u_3^1(j)\}_{k_1,k_2}\Bigg)\\
&+(2\pi)^2|\frac{k}{L}|^2\Bigg(\sum_{i,j=1}^{N_3}<\partial_z(\psi_i,\psi_j),\psi_n>\mathcal{F}_{x,y}\{u_3^1(i)u_3^1(j)\}_{k_1,k_2}\Bigg)
\end{aligned}
\tag{A.16}
$$

The projection of the nonlinear part of g is given by:

$$
\begin{aligned}
&\frac{1}{L_xL_y}\int_{-\frac{1}{2}}^{\frac{1}{2}}\int_0^{L_y}\int_0^{L_x}\epsilon\cdot[div(u\otimes u)]\overline{e^{\frac{2\pi ik_1x}{L_x}}e^{\frac{2\pi ik_2y}{L_y}}S_n(z)}\\
&=-\frac{(2\pi)^2k_1k_2}{L_xL_y}\Bigg(\sum_{i,j=1}^{N_3}<(\psi_i'\psi_j'),S_n>\mathcal{F}_{x,y}\{u_1^1(i)u_1^1(j)\}_{k_1,k_2}\\
&\quad+2<(\psi_i'S_j),S_n>\mathcal{F}_{x,y}\{u_1^1(i)u_1^2(j)\}_{k_1,k_2}\\
&\quad+<(S_iS_j),S_n>\mathcal{F}_{x,y}\{u_1^2(i)u_1^2(j)\}_{k_1,k_2}\Bigg)\\
&\quad-\frac{(2\pi k_2)^2}{L_y^2}\Bigg(\sum_{i,j=1}^{N_3}<(\psi_i'\psi_j'),S_n>\mathcal{F}_{x,y}\{u_2^1(i)u_1^1(j)\}_{k_1,k_2}\\
&\quad+<(\psi_i'S_j),S_n>\mathcal{F}_{x,y}\{u_2^1(i)u_1^2(j)\}_{k_1,k_2}\\
&\quad+<(S_i\psi_j'),S_n>\mathcal{F}_{x,y}\{u_2^2(i)u_1^1(j)\}_{k_1,k_2}\\
&\quad+<(S_iS_j),S_n>\mathcal{F}_{x,y}\{u_2^2(i)u_1^2(j)\}_{k_1,k_2}\Bigg)\\
&\quad+\frac{2\pi ik_2}{L_y}\Bigg(\sum_{i,j=1}^{N_3}<\partial_z(\psi_i\psi_j'),S_n>\mathcal{F}_{x,y}\{u_3^1(i)u_1^1(j)\}_{k_1,k_2}\\
&\quad+<\partial_z(\psi_iS_j),S_n>\mathcal{F}_{x,y}\{u_3^1(i)u_1^2(j)\}_{k_1,k_2}\Bigg)\\
&\quad+\frac{(2\pi k_1)^2}{L_x^2}\Bigg(\sum_{i,j=1}^{N_3}<(\psi_i'\psi_j'),S_n>\mathcal{F}_{x,y}\{u_1^1(i)u_2^1(j)\}_{k_1,k_2}\\
&\quad+<(\psi_i'S_j),S_n>\mathcal{F}_{x,y}\{u_1^2(j)u_2^1(i)\}_{k_1,k_2}\\
&\quad+<(S_i\psi_j'),S_n>\mathcal{F}_{x,y}\{u_1^1(j)u_2^2(i)\}_{k_1,k_2}\\
&\quad+<(S_iS_j),S_n>\mathcal{F}_{x,y}\{u_1^2(i)u_2^2(j)\}_{k_1,k_2}\Bigg)\\
&\quad+\frac{(2\pi)^2k_1k_2}{L_xL_y}\Bigg(\sum_{i,j=1}^{N_3}<(\psi_i'\psi_j'),S_n>\mathcal{F}_{x,y}\{u_2^1(i)u_2^1(j)\}_{k_1,k_2}\\
&\quad+2<(\psi_i'S_j),S_n>\mathcal{F}_{x,y}\{u_2^1(i)u_2^2(j)\}_{k_1,k_2}\\
&\quad+<(S_iS_j),S_n>\mathcal{F}_{x,y}\{u_2^2(i)u_2^2(j)\}_{k_1,k_2}\Bigg)\\
&\quad-\frac{2\pi ik_1}{L_x}\Bigg(\sum_{i,j=1}^{N_3}<\partial_z(\psi_i\psi_j'),S_n>\mathcal{F}_{x,y}\{u_3^1(i)u_2^1(j)\}_{k_1,k_2}\\
&\quad+<\partial_z(\psi_iS_j),S_n>\mathcal{F}_{x,y}\{u_3^1(i)u_2^2(j)\}_{k_1,k_2}\Bigg)
\end{aligned}
\tag{A.17}
$$

The nonlinear parts of F and G are given by:

$$\begin{aligned}&\int_{-\frac{1}{2}}^{\frac{1}{2}} \partial_z < u_1 u_3 >^{xy} \overline{S_n(z)}\\ &= \sum_{i,j=1}^{N_3} < \partial_z(\psi_i' \psi_j), S_n > \mathcal{F}_{x,y}\{u_1^1(i) u_3^1(j)\}_{0,0}\\ &\quad + < \partial_z(S_i \psi_j), S_n > \mathcal{F}_{x,y}\{u_1^2(i) u_3^1(j)\}_{0,0}\end{aligned} \tag{A.18}$$

$$\begin{aligned}&\int_{-\frac{1}{2}}^{\frac{1}{2}} \partial_z < u_2 u_3 >^{xy} \overline{S_n(z)}\\ &= \sum_{i,j=1}^{N_3} < \partial_z(\psi_i' \psi_j), S_n > \mathcal{F}_{x,y}\{u_2^1(i) u_3^1(j)\}_{0,0}\\ &\quad + < \partial_z(S_i \psi_j), S_n > \mathcal{F}_{x,y}\{u_2^2(i) u_3^1(j)\}_{0,0}\end{aligned} \tag{A.19}$$

Appendix B

Discrete equations for Swift-Hohenberg model

The discrete equations in case of Swift-Hohenberg model are much easier since the problem “neglects” the vertical component, as it is a two-dimensional model equation. Again taking advantage of a Fourier series expansion you can easily verify that the analytic equations are diagonalized by Galerkin method and Fourier expansions. We start again from the following set of equations:

$$\frac{\partial}{\partial t}u + V \cdot \nabla u = \epsilon u - (\Delta + 1)^2 u - u^3 + \delta u^2 \tag{B.1}$$

$$V = (\partial_y \zeta, -\partial_x \zeta) \tag{B.2}$$

$$\Delta \zeta = c \cdot \hat{z} \cdot \nabla u \times \nabla(\Delta u) \tag{B.3}$$

Expanding u and ζ into Fourier series leads to:

$$\frac{\partial}{\partial t}\hat{u}(k_1, k_2) = \epsilon \hat{u}(k_1, k_2) - \left(|\frac{k}{L}|^2 + 1\right)^2 - \mathcal{F}_{x,y}\{\partial_y \zeta \partial_x u - \partial_x \zeta \partial_y u + \delta u^2 - u^3\}_{k_1,k_2} \tag{B.4}$$

$$-|\frac{k}{L}|^2 \hat{\zeta}(k_1, k_2) = c \cdot \left(\mathcal{F}_{x,y}\{\partial_x u \cdot (\partial_y \partial_x^2 u + \partial_y^3 u)\}_{k_1,k_2}\right) \tag{B.5}$$

It is obvious that these equations are more simple than the complete Boussinesq equations. Nevertheless it shows many of the pattern forming processes.

Bibliography

[1] H. Bénard, *Revue generale des Sciences*, Vol. XII, p. 1261-1309, 1900

[2] J. W. Rayleigh, *On convective currents in a horizontal layer of fluid, when the higher temperature is on the underside*, Philosophical Magazine Vol. XXXII, p. 529-546, 1916

[3] S. Chandrasekhar, *Hydrodynamic and Hydromagnetic Stability*, Dover Publications, New York, 1961

[4] M. Cross, H. Greenside, *Pattern Formation and Dynamics in Nonequilibrium Systems*, Cambridge University Press, 2009

[5] S. W. Morris, E. Bodenschatz, D. S. Cannell, G. Ahlers, *Spiral Defect Chaos in Large Aspect Ratio Rayleigh-Bénard Convection*, Physical Review Letters, Vol. 71 Nr. 13, 1993

[6] E. Bodenschatz, W. Pesch, G. Ahlers *Recent Developments in Rayleigh-Bénard Convection*, Annual Review Fluid Mechanics 2000.32 (709-778)

[7] R. Hoyle, *Pattern Formation: An Introduction to methods*, Cambridge University Press, 2006

[8] Swift, Hohenberg, *Hydromagnetic fluctuations at the convective instability*, Phys. Rev. A 15, 319-328, 1977

[9] W. Decker, *Mathematische Methoden zur Beschreibung strukturbildender Systeme - eine kritische Analyse*, Phd thesis, University Bayreuth, 1995

[10] A. Tschammer, *Nichtlineare Aspekte der Rayleigh-Bénard Konvektion in isotropen und anisotropen Fluiden*, Phd thesis, University Bayreuth, 1996

[11] L.C. Evans, *Partial Differential Equations*, Volume 19, 1997

[12] S. A. Orszag, *Numerical Simulation of Incompressible Flows Within Simple Boundaries - I. Galerkin (Spectral) Methods*, Studies in applied mathematics Vol. L No. 4, December 1971

[13] P. Angot, C. Bruneau, P. Fabrie, *A penalization method to take into account obstacles in incompressible viscous flows*, Springer Verlag 1999, Numerische Mathematik 81 (497-520)

[14] B. Fornberg, *A Practical Guide To Pseudospectral Methods*, Cambridge Monographs on Applied and Computational Mathematics, 1996

[15] B.J. Schmitt, W. Wahl, *The Navier Stokes Equation II -Theory and Numerical Methods*, Lecture Notes in Mathematics, Springer, 1992

[16] B.J. Schmitt, W. Wahl, *Decomposition of solenoidal fields into poloidal field, toroidal fields and mean flow. Applications to the Boussinesq-Equations*, lecture notes

[17] A. Kassam, Llyod N. Trefethen, *Fourth order time stepping for stiff PDEs*, SIAM Journal on Scientific Computing 26, 1214-1233, 2005

[18] Q. Du, W. Zhu, *Analysis and Applications of the Exponential Time Differencing Schemes and Their Contour Integration Modifications*, BIT Numerical Mathematics 45, 307-328, 2005

[19] H.W. Alt, *Lineare Funktional-Analysis*, Springer, 5. Ausgabe, 2006

[20] R. Schaback, H. Wendland, *Numerische Mathematik*, Springer, 5. Auflage, 2004

[21] R. Verfuerth, *Numerische Stroemungsmechanik*, lecture notes 1998/99, University Bochum

[22] Nvidia, *NVIDIA CUDA C Programming Guide*, Version 3.2, 2010

[23] Nvidia, *CUDA CuFFT Library*, Version 2.0, August 2010

[24] D. H. Bailey, *High-Precision Floating-Point Arithmetic in Scientific Computing*, Computing in Science and Engineering, 2005

[25] M. Bernaschi, M. Fatica, S. Melchionna, S. Succi, E. Kaxiras, *A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries* , Journal Concurrency and Computation: Practice and Experience Vol. 22 Issue 1, 2010

[26] D. Michea, D Komatitsch, *Accelerating a three dimensional finite-difference wave propagation code using GPU graphics cards*, Geophysical Journal International 182, p. 389-402, 2010

[27] J. A. Anderson, C. D. Lorenz, A. Travesset, *General purpose molecular dynamics simulations fully implemented on graphics processing units*, Journal of Computational Physics 227, p. 5342 - 5359, 2008