FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Code Generation for Lattice-Boltzmann Methods with Large Neighbourhoods

Mischa Dombrowski

Bachelorarbeit

Code Generation for Lattice-Boltzmann Methods with Large Neighbourhoods

Mischa Dombrowski

Bachelorarbeit

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	M.Sc Martin Bauer
Bearbeitungszeitraum:	02.07.2019 - 30.09.2019

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 30. September 2019 .....

Contents

List of abbreviations

LBM Lattice Boltzmann method

NSE Navier-Stokes Equation

BGK Bhatnagar–Gross–Krook

TSA Time step algorithm

MBC Maxwell-Boltzmann constraint

MRT Multiple relaxation time

CSE Common subexpression elimination

Abstract

Only nine discrete velocities are necessary for the two-dimensional lattice Boltzmann method to simulate a fluid. This restriction is enough to physically motivate the simulation but quickly reaches its limits if the initial conditions are not optimal. The introduction of large neighbourhoods increases this discretisation by including more possibilities for the populations to move during one iteration. A recently developed method to semi-automatically create these neighbourhoods is combined with code generation to implement the two-dimensional lattice Boltzmann method without any dependency on the lattice used. Additionally, the introduction of code generation leads to a significant improvement of the execution time and the generality of the implementation. The main advantages and disadvantages of using two of the most popular large neighbourhoods, namely the D2V17 and the D2V37 lattice, are analysed. The main drawback of these implementations are that they increase memory usage and computation time. In return, they increase the accuracy and stability of the simulation. This positive influence is demonstrated by comparing the different lattices using the Taylor-Green flow as a benchmark. Looking at the macroscopic values and comparing them to the analytically correct solution shows that the large velocity discretisations visibly reduce the error of the simulation with the same spatial resolution. Additionally initialising the flow with different values for the viscosity and maximum velocity successfully demonstrates how the D2V17 can be used to improve the stability.

Es werden nur neun diskrete Geschwindigkeiten für die zwei-dimensionale Lattice-Boltzmann-Methode gebraucht, um eine Flüssigkeit zu simulieren. Diese Einschränkung reicht aus, um die Simulation physikalisch zu begründen, gerät aber schnell an ihre Grenzen, wenn die Anfangsbedingungen nicht optimal sind. Das Hinzufügen von größeren Nachbarschaften erhöht diese Diskretisierung und führt dazu, dass die Populationen mehr Möglichkeiten haben, sich während eines Zeitschrittes zu bewegen. Eine kürzlich entwickelte Methode, um diese Nachbarschaften fast automatisch zu erstellen, wird mit Code Generierung kombiniert, um die zweidimensionale Lattice-Boltzmann-Methode ohne Abhängikeit zu der Diskretisierung zu implementieren. Zusätzlich führt die Einführung von Code Generierung zu einer deutlichen Verbesserung der Ausführungszeit und der Allgemeinheit der Implementierung. Die Vor- und Nachteile der Benutzung von zwei der wichtigsten großen Nachbarschaften mit den Namen D2V17 und D2V37 werden analisiert. Der Nachteil dieser Implementierungen ist eine höhere Ausführungszeit und ein höherer Speicherverbrauch. Im Gegenzug erhöhen sie die Genauigkeit und in einem Fall auch die Stabilität der Simulation. Dieser positive Effekt wird anhand des Taylor-Green Flows gezeigt. Der Vergleich der makroskopischen Werte mit der analytisch korrekten Lösung zeigt, dass Abweichungen dieser Werte für die gleiche räumliche Auflösung bei den großen Nachbarschaften deutlich geringer sind. Zusätzlich wird durch die Initialiserung des Benchmarks mit Hilfe von verschiedenen Viskositäten und maximalen Geschwindigkeiten erfolgreich gezeigt, wie der D2V17 Lattice genutzt werden kann, um die Stabilität zu erhöhen.

1 Introduction

A few months after the turn of the millennium the Clay Mathematics Institute published a list of seven questions they considered to be some of the most important unsolved problems in mathematics. They are now commonly known as the Millennium Problems of mathematics. One reason why these problems gained so much attention is that solving them, or proving them wrong, gets rewarded with one million dollars each. The only conditions were that these problems had to be important and difficult, in a sense that scientists have tried solving them for a long time [1].

For this thesis, the *Navier-Stokes existence and smoothness problem in three dimensions* stands out. The solution to this problem is supposed to further the theoretical understanding of the Navier-Stokes Equations (NSEs). They can be summarised as three equations, one for each dimension that describe the influence of external and internal forces on a fluid, paired with one equation to guarantee the conservation of mass. The problem is that it has yet to be proven that, in three-dimensional space, a smooth and physically reasonable solution exists [2].

Despite the significance of this problem, many methods that are based on the NSEs have emerged throughout the years. These methods yield satisfying results that don't require the Millennium Problem to be solved. One of these methods is the lattice Boltzmann method (LBM). Instead of

simulating the parameters part of the NSEs, like velocity or pressure it simulates their distribution. It can be shown that for the values of interest, this is a valid approach to simulate a fluid, if it is behaving sufficiently well. Turbulent flows are one example where this condition is not always satisfied resulting in stability problems of the algorithm. A few papers have been published showing how simulating those fluids can be problematic, for example [3]. Nevertheless, the LBM is already an established method and even predicted to "keep growing as a useful tool in all major fields of classical fluid dynamics" [4]. Additionally, it is very versatile and can for example also be used in other fields seemingly unrelated to fluid simulations, like image processing, where the LBM is implemented as an algorithm to detect contour lines on images, with one of the key advantages, compared to the other methods being that it is a highly parallelisable algorithm [5].

One of the key parts of the method is the discretisation of all the quantities of the system. For the discretisation of space this means, that the simulation is only defined on a grid of points, instead of at every point in space. The same goes for the discretisation of time. Similar to other numerical methods, increasing this discretisation usually leads to more accurate results.

There is another less obvious type of discretisation that has to be performed. If the values of a system are only defined at specific points and movement is introduced to these values, then it has to be guaranteed that all the distributions are on top of another point on the grid, after exactly one time step. Since this is a restriction for particle distributions to move only in a few directions, with a fixed length, in a fixed amount of time, this can be seen as the discretisation of the velocities. One possible discretisation of the velocity is to say that in two dimensions particles located on one node can only move to one of its eight neighbouring nodes or remain on the current one. Even though this might seem like a big restriction at first, these nine discrete velocities are already enough to retrieve the correct macroscopic behaviour, as shown in chapter 2.1.4.

At this point the obvious next step is to introduce another layer of velocities, or in other words a large neighbourhood, corresponding to the movement of particles to nodes one step further away. Models with a higher amount of velocities exist almost as long as the LBM itself. Most of the early proposed models came with significant disadvantages like numerical instabilities [3] and therefore have not been very commonly adapted. Recently more work has been published about new and larger velocity sets, but the standard models are still the go to models for every implementation. First and foremost the paper by Philippi et al. who found multiple suitable sets of velocities to increase the neighbourhood and the accuracy of the truncated equilibrium distribution function [6]. The meaning behind this truncation will be explained in chapter 2.1.4. Secondly D. Spiller and B. Duenweg proposed a method to semi automatically retrieve arbitrarily large velocity sets using a `Python` script [7]. The goal of this thesis is to not only implement these large neighbourhoods for the third and fourth order accurate truncation of the equilibrium distribution function and to show the advantages and disadvantages of these models, but also to add code generation to significantly speed up the execution and implementation time.

The first part of this thesis establishes the physical background necessary for the LBM. Afterwards the underlying algorithm is described for the two dimensional implementation with the usual lattice. Starting with chapter 3 the focus switches over to the introduction of more accurate models and the consequences of using these models on the implementation, performance, accuracy and stability is discussed.

2 Lattice-Boltzmann Method

The first chapter is supposed to provide the reader with an understanding of the underlying physics that are used to simulate flows with the LBM. This chapter and especially the physical derivation is influenced by [8] and is supposed to be a conclusion of the most important aspects it presents with a focus on the parts that are relevant for the rest of this thesis.

At first the two equations the algorithm is build around and their meaning is introduced which leads to the most basic implementation of the LBM. Afterwards the method is discussed, with a special focus on parts that are important for the implementation of large neighbourhoods.

2.1 Physical Introduction

One of the intriguing facts about fluid dynamics is that the governing equations which are supposed to be solved by the simulation, are easy to understand and very intuitive. The equations used to derive the LBM are based on the physical laws of conservation of mass and conservation of momentum. They are easy to understand on a large scale, further referred to as macroscale, but less obvious on a scale where only a few molecules are inside a given volume. The LBM itself, however, works in neither of those domains, but in between them.

The first equation is often referred to as the continuity equation in fluid dynamics. The equation basically describes mass conservation of a system and states that change of mass m for an arbitrary Volume V_0 must be due to flow in or out of said volume. The flow itself depends on the velocity $\boldsymbol{u}$ (bald stands for vector valued). This can be written as the following mathematical equation:

$$\frac{\partial}{\partial t}\int\limits_{V_0} \rho \, dV \; = - \oint\limits_{\partial V_0} \rho \boldsymbol{u} \, d\boldsymbol{A}$$

Here ρ stands for the density and if it is known, then the mass can be calculated by integrating over the volume. The left hand side is a simple surface integral, that calculates mass flowing through the surface of the volume V_0 and therefore the whole part that is currently of interest.

By applying Gauss's theorem, differentiating and bringing the right hand side of the equation to the left side the continuity equation is retrieved:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0$$

The second equation is called the Navier-Stokes Equation (NSE) and covers the conservation of momentum in the simulation. It can be derived in a very similar way by saying that there are three reasons why the momentum of a fluid element could change. Firstly there could be flow of momentum in or out of the observed volume. Secondly its pressure could change and lastly the momentum could change due to external body forces like gravity or centrifugal forces. Mathematically this equation can be written in terms of the velocity $\boldsymbol{u}$, the density ρ the forces $\boldsymbol{F}$ and the pressure p:

$$\frac{d}{dt}\int\limits_{V_0} \rho \boldsymbol{u} \, dV \; = - \oint\limits_{\partial V_0} \rho \boldsymbol{u}\boldsymbol{u} \, d\boldsymbol{A} \; - \oint\limits_{\partial V_0} p \, d\boldsymbol{A} \; + \int\limits_{V_0} \boldsymbol{F} \, dV$$

This can once again be simplified by using Gauss's theorem to get to the Euler equation that is used to describe an ideal fluid. Fluids in real life, however, are also affected by internal forces caused by friction, also known as viscosity. The fact that the viscosity of a fluid has to be part of this equation is pretty intuitive, since the simulation of a viscous liquid like honey should not lead to the same result as the simulation of water. To add the viscosity η to the equation it has to be written in a more general form that can afterwards be simplified as described in [8]. The resulting equation is well known as the incompressible NSE:

$$\rho \frac{D\boldsymbol{u}}{Dt} = -\nabla p + \eta \Delta \boldsymbol{u} + \boldsymbol{F}$$

2.1.1 Scale of the Lattice-Boltzmann Method

The fact that fluids can be described using different scales was already introduced. A better understanding of the different scales and the methods used to simulate fluids on them is crucial to understand the advantages of the LBM and how it improved some problems other methods have. The description of a fluid can be broken down into three different scales: macroscopic, mesoscopic and microscopic.

The first and biggest scale, called the macroscopic scale, is also the one that is of most interest. This is because it simulates fluids using quantities like density, velocity and temperature and it therefore directly solves the equations for the most important variables. This can be done by solving the continuity equation and the NSE directly. A few computational fluid dynamics methods, like finite elements or finite differences, have been implemented to solve flow related problems on this scale. These methods have in common that they are pretty difficult to implement for complex geometries, turbulent flows, and additional complexity is added when certain flow related problems arise [8].

The second scale is also the smallest and further referred to as microscopic scale. Flows can be simulated by looking at their molecules, pairs of molecules or in general particles and simulating their interactions, like collisions. Methods simulating this behaviour are generally called particle-based, but there are particle based methods working on bigger scales, as well [8]. There are a few obvious disadvantages to microscopic simulations, namely that simulating a fluid on such a small scale would lead to very small results in space, that are usually not relevant. Simply increasing the system size would therefore be incredibly time consuming.

The third scale, called mesoscopic scale is the most important one, because the LBM is based on it but probably also the least intuitive. It works in between the other two mentioned levels and instead of describing single molecules, looks at at the distribution of them. In case of the LBM this leads to the continuous distribution function $f(\boldsymbol{x}, \boldsymbol{\xi}, t)$. The distribution function describes the density of particles at the position $\boldsymbol{x}$ and time t with velocity $\boldsymbol{\xi}$. This function by itself is a full description of the system, for example one could compute the macroscopic mass density by computing the zeroth order moment.

2.1.2 Kinetic Theory

The distribution function introduced in the previous chapter will be used throughout the whole LBM and therefore it is useful to dive deeper into its properties to understand how it can represent a whole system by itself and how macroscopic values can be computed from it.

Kinetic theory states that gases, if left alone for a long enough period of time, tend to relax to their equilibrium distribution f^{eq} called the Maxwell-Boltzmann distribution.

The whole idea of the LBM is to divide the simulation of a fluid into one part, where particle distributions move from one position to another, and one part where the distribution locally relaxes towards the Maxwell-Boltzmann distribution. This relaxation towards the equilibrium distribution can be described using the Boltzmann equation. This equation captures how the fluid evolves over time and is represented by $\Omega(f)$ often called the collision operator. Its description is one of the key problems of the LBM due to the amount of different ways and difficulty levels it can be implemented [8]. The fact that this description can be applied to solve the NSEs is not obvious at all, but mandatory to physically motivate the collision operator and therefore the LBM. This is achieved by applying the Chapman-Enskog expansion to the problem, as shown for example in [9].

The most important properties this operator has to have is mass, momentum, internal and external energy conservation given by the moments of the collision operator. This leads to the simplest and most commonly used collision operator called the Bhatnagar–Gross–Krook (BGK) collision operator:

$$\Omega(f) = -\frac{1}{\tau}(f - f^{eq})$$

The motivation behind this operator is that the gas takes a specified time τ, also called the relaxation time, to reach its local equilibrium state. This parameter is closely related to the kinematic viscosity of the simulated fluid. A closer look at this relaxation time reveals some interesting

properties that have to be kept in mind during simulations. First of all, if $\tau = 1$ this operator simplifies to setting the current field to its equilibrium state. Additionally, the relaxation time is closely related to the internal friction (viscosity) of a fluid. This relation is explained in chapter 2.2.4 and is of great importance for the stability and accuracy analysis of a simulation.

2.1.3 Dimensionless Numbers

For general fluid literature but also for this thesis, dimensionless numbers play a key role in the description and the comparability of fluid simulations. They generally have no unit and carry no information about the physical size of a simulation. For this thesis the most important dimensionless number will be the Reynolds number, mainly due to it's importance for the investigation of viscous flows [10]:

$$Re = \frac{ul}{\nu}$$

There are applications for simulations with low Reynolds numbers like [11], but for this thesis the focus lies on high Reynolds numbers. The Reynolds number describes the ratio of the fluid velocity u times the characteristic length of the system l divided by the kinematic viscosity ν of the fluid. For this thesis the characteristic length will always be equal to the resolution of the simulation. There are two key takeaways to be made from this. First of if two flows with the same kinematic viscosity and the same resolution are compared, the flow with the higher Reynolds number also has higher speed. This means that high Reynolds numbers generally indicate large velocities. The second takeaway is the fact that the Reynolds number depends on the characteristic length of the system, which itself could be considered a design parameter. This means that two flows with e.g. the same kinematic viscosity but completely different velocities can have the same Reynolds number, if the characteristic length is scaled accordingly.

The important thing about dimensionless numbers is that in physics the law of similarity states that two simulations share the same physical properties, with respect to a scaling factor of the whole system, if the dimensionless numbers are the same. This is why they are such a popular tool for the comparison of flows. Another important example of a dimensionless number in the context of a simulation is the Mach number, defined by the ratio of the speed of the fluid to the speed of sound:

$$Ma = \frac{u}{c_s}$$

The speed of sound is a fixed value that is connected to the velocity sets. Flows with Mach numbers below 0.1 can be considered incompressible [8]. Additionally, the value for the Mach number divided by the Reynolds number is directly proportional to the Knudsen number. This number should be kept below one or otherwise the connection between gases and fluids using kinetic theory can no longer be established [9], but can be ignored due to the fact that this thesis focuses on application with high Reynolds numbers. Although it is important to always stay in between the limits created by these numbers, their exact value is not important when comparing two flows with the same Reynolds numbers according to [8].

2.1.4 Velocity Discretisation

In chapter 2.1.2 the connection between the NSEs and the Boltzmann equation was made. This means that instead of using the original equations, a fluid can be simulated with the newly acquired equations. Obviously, to calculate a solution for it on a computer, the Boltzmann Equation has to be discretised. Currently the system is defined for every point in time, space and for every velocity the particles might have. The same thing goes for the Maxwell-Boltzmann distribution. The step of discretisation is crucial, because it has to be proven that the solution afterwards is in fact a solution of the original problem with respect to some error that may be introduced due to truncation.

For the LBM or more specifically the Boltzmann equation three different types of discretisations are needed, namely that of the velocity, space and time. The first is the most important for this thesis, and therefore will be talked about in detail. Nonetheless, the discretisation of space and time is in general equally important and one of the reasons why the velocity discretisation is not as easy as it seems. A detailed explanation is for example given in chapter 3.5 of [8].

The most important takeaway is that during one time step Δt, the particle moves from one node to the exact position of another node if every discrete velocity is an integer multiple of $\Delta x/\Delta t$. For simplification reasons the value for Δt is from now on always chosen to be equal to one and therefore the velocities have to be integers greater or equal to zero.

The velocity discretisation is achieved by looking at the equilibrium distribution function, describing it using a finite Hermite series expansion and using the specific characteristics of this expansion to show that macroscopic moments are conserved exactly. First the force-free Boltzmann equation and the general equilibrium distribution mentioned in 2.1.2 are taken and then non-dimensionalised. Removing dimensions means that every unit has to be removed from the equation by multiplying it with the characteristic properties of the system. This leads to the following, non dimensional description of the continuous system[1]:

$$\frac{\partial f}{\partial t} + \xi_\alpha \frac{\partial f}{\partial x_\alpha} = \Omega(f) \qquad\qquad f^{eq}(\rho, \boldsymbol{u}, \theta, \boldsymbol{\xi}) = \frac{\rho}{(2\pi\theta)^{d/2}} e^{-(\boldsymbol{\xi}-\boldsymbol{u})^2/(2\theta)}$$

The left equation is the continuous force-free non-dimensional Boltzmann equation and the right one the non-dimensional equilibrium distribution. θ in the latter function is the non-dimensional temperature and $\mathbf{u}$ the fluid velocity. The appearance of temperature in the isothermal equations might seem confusing at first, but can be explained by the fact that temperature is nothing else than fast moving particles. Intuitively the distribution should change if the velocity of its particles is different.

The only restriction introduced to the collision operator is that it has to conserve the moments, as described in chapter 2.1.2. Consequently it is sufficient to show that the discretised equilibrium function conserves moments. To discretise the non-dimensional equilibrium function a Hermite series expansion is used as explained in more detail in [8]. This means that the function is rewritten in terms of a weighted base function and some of the properties of the functions are used to solve the discretisation problem. The most important characteristic becomes clear when looking at the description of the Hermite polynomials of n-th order and their base functions in d dimensions:

$$\boldsymbol{H}^{(n)}(\boldsymbol{x}) = (-1)^n \frac{1}{\omega(\boldsymbol{x})} \nabla^{(n)} \omega(\boldsymbol{x}), \qquad\qquad \omega(\boldsymbol{x}) = \frac{1}{(2\pi)^{d/2}} e^{-\boldsymbol{x}^2/2}$$

The weight function $\omega(\boldsymbol{x})$ is very similar to the equilibrium distribution. In fact, it can be written in terms of the weight function:

$$f^{eq}(\rho, \boldsymbol{u}, \theta, \boldsymbol{\xi}) = \frac{\rho}{\theta^{d/2}} \omega\left(\frac{\boldsymbol{\xi} - \boldsymbol{u}}{\sqrt{\theta}}\right)$$

The second important characteristic, also known as the Gauss-Hermite quadrature rule, is that Hermite Polynomial can be integrated exactly for a fixed but finite amount of abscissa values. In the case of integrating over the velocity space, this means that only a finite amount of velocities have to be known, which is exactly the goal.

The Hermite series expansion is the process of writing the equilibrium function as the weight function times the sum of infinitely many Hermite polynomials and their coefficients $a^{(n),eq}$. Symbolically calculated the first three coefficients of the expansion are[2]:

$$\begin{aligned} a^{(0),eq} &= \rho \\ a^{(1),eq} &= \rho u_\alpha \\ a^{(2),eq} &= \rho \left(u_\alpha u_\beta + (1-\theta)\delta_{\alpha\beta}\right) \end{aligned}$$

These coefficients correspond or are related to the moments that are supposed to be conserved. Therefore, it is only necessary to calculate the first three coefficients and the first three summands

[1] The greek indices imply Einstein's summation convention. That means reoccurring Greek indices are implicitly summed over[12].

[2] $\delta_{\alpha\beta}$ in the last equation represents the Kronecker delta.

of the Hermite series expansion to fully recover the macroscopic characteristics of the equilibrium distribution function.

So far the calculation of the coefficients still requires integration. This dependency can be removed by applying the Gauss-Hermite quadrature rule. As mentioned earlier in this section, instead of integrating over $\boldsymbol{\xi}$ to calculate one coefficient $a^{(n),eq}$ one can sum over a finite set of abscissae values $\boldsymbol{\xi}_i$.

The choice of those abscissae values and the corresponding weights is neither arbitrary, nor easy to derive. The result however is rather simple, as it leads to the most commonly used lattices for the LBM, namely the D2Q9 or more specifically to it's velocity vectors $\boldsymbol{c}_i$, weights w_i and speed of sound c_s [8]. The DdQq naming convention is used throughout all of the literature to indicate the form and type of the lattice that is used. The digit after D indicates the dimension of the lattice and therefore also the dimension of the simulation and the digit after Q indicates the amount of discrete velocities in the velocity set. Except for some more examples the values derived by the quadrature do not lead to regular and space filling lattice [6]. This means the increasing the amount of conserved moments and therefore the accuracy of the discretisation is not as easy as adding more coefficients and discrete velocities.

The third coefficient can be further simplified for non-thermal simulations by considering the fluid to be isothermal ($\theta = 1$). The complete second order accurate discrete equilibrium distribution for every discrete velocity index i corresponding to the velocity vector $\boldsymbol{c}_i$ can be calculated in the following way:

$$f_{i,2}^{eq} = w_i \rho \left(1 + \frac{c_{i\alpha} u_\alpha}{c_s^2} + \frac{u_\alpha u_\beta (c_{i\alpha} c_{i\beta} - c_s^2 \delta_{\alpha\beta})}{2 c_s^4} \right)$$

The factor c_s^2 describes the relationship between pressure and density ($p = c_s^2 \rho$) and comes from the normalisation of the abscissae values of the Hermite polynomials to integers. It therefore depends on the lattice used, which means that this value is different for some of the large neighbourhoods introduced in chapter 3.

2.2 Implementation of the Lattice-Boltzmann Method

Now that the most important physical background was established it is time to look at the most basic implementation of the two-dimensional LBM. The whole simulation can be combined into one single function that is iterated over to simulate the behaviour of a fluid for a single time step. This function and the process of calculating one iteration is further referred to as the time step algorithm (TSA). If initialised correctly the algorithm can simulate a fluid in a satisfying way which includes correctly simulating and visualising its macroscopic behaviour.

A special focus of this chapter is set on the parts of the algorithm that are different if large lattices are implemented. Almost all parts of the TSA are only effected by large neighbourhoods performance-wise. The only exceptions are the discretisation of velocity space and therefore the way the weights and velocities are represented, the calculation of the equilibrium distribution function, and to a small extent the propagation of the populations. Every other part of the algorithm can be implemented in a lattice independent fashion, without putting much additional effort into it.

2.2.1 The Time Step Algorithm

The complete LBM can be divided into two steps. First the populations collide and the local equilibrium is calculated. The second step is to propagate the newly calculated populations into the directions defined by the lattice. The TSA is the process of calculating the new density distribution from the old one in each time step, hence the name.

Each iteration starts by updating the macroscopic values used to determine the equilibrium distribution, which are the mass density and the momentum density. Then these values are used to calculate the equilibrium distribution up to a specific order of accuracy which is usually two. Afterwards the collision is simulated by applying the BGK collision operator described in 2.1.2. Finally the streaming process starts. This is the only part where particles move from one cell to a neighbouring cell. This step is important because it is different depending on the ordering of the velocity sets and the maximum range the velocities have. Increasing the size of the velocity discretisation therefore leads to a larger neighbourhood. The density in each node, or more specifically for each velocity in each node, is going to be called the density distribution f as a function of space $\boldsymbol{x}$

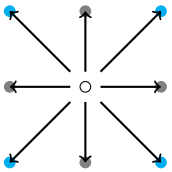


Figure 1: D2Q9 lattice

and time t and each entry its' population. Thanks to the results of chapter 2.1.4 this function is now discrete.

The discretisation of the velocity space was described in chapter 2.1.4 and it was mentioned that only a finite amount of discrete velocities is needed to simulate the fluid on a macroscopic level. This means for the density distribution function and for the system as a whole that the particles are only allowed to move in those directions. One of the most commonly used velocity sets in two dimensions is the D2Q9 lattice shown in figure 1.

The lattice consists of the stationary velocity that simulates non moving particles and eight velocities equivalent to the movement in each direction. During each iteration of the TSA the particle moves from one cell to a neighbouring cell. The populations are saved individually for each velocity to know which direction each entry is supposed to stream to. To implement the LBM one has to know the weights of those velocities. These weights can come from the derivation with the Hermite Polynomials introduced in chapter 2.1.4 but there is also a strong rule set around them regarding the isotropy of their moments. The main requirements for the moments of the weights w_i are, that every weight is positive and that isotropy up to the fifth order is given. If this is not the case, the velocity set is not suitable for the LBM as a NSE solver [8]. Solving the (1 - 6) equations corresponding to this requirement gets significantly easier if only symmetric velocity sets are considered. This implies that the weights for each velocity with the same length are equal. Therefore, discrete velocities can be combined into velocities that share the same length, further referred to as energy shells [13].

For the D2Q9 lattice this means that, since only three different lengths are possible, only three weights have to be known. The values of these weights are $w_{00} = 4/9$, $w_{10} = 1/9$ and $w_{11} = 1/36$. This syntax can be understood as if one velocity vector is exemplary for the whole energy shell. The indices denote what energy shell the weight corresponds to. For the rest of this thesis theses indices are interpreted as Cartesian coordinates with the origin in the center of each velocity set. Therefore, the velocity (1, 0) corresponds to the vector pointing one node to the right and zero nodes upwards and has the weight w_{10}. The full velocity set and the values for every discrete velocity c_i is shown in chapter 3.2.1.

$$\sum_i w_i = 1 \tag{1}$$

$$\sum_i w_i c_{i\alpha} = 0 \tag{2}$$

$$\sum_i w_i c_{i\alpha} c_{i\beta} = c_s^2 \delta_{\alpha\beta} \tag{3}$$

$$\sum_i w_i c_{i\alpha} c_{i\beta} c_{i\gamma} = 0 \tag{4}$$

$$\sum_i w_i c_{i\alpha} c_{i\beta} c_{i\gamma} c_{i\mu} = c_s^4 (\delta_{\alpha\beta}\delta_{\gamma\mu} + \delta_{\alpha\gamma}\delta_{\beta\mu} + \delta_{\alpha\mu}\delta_{\beta\gamma}) \tag{5}$$

$$\sum_i w_i c_{i\alpha} c_{i\beta} c_{i\gamma} c_{i\mu} c_{i\nu} = 0 \tag{6}$$

One consequence is that the velocity discretisation has a huge impact on the memory layout of the LBM. Two dimensional simulations are implemented using a three dimensional array. The first two dimensions correspond to the physical dimensions while the last dimension is used to separately save the distributions of each discrete velocity for one cell. This means that the size of the array

in the first two dimensions is determined by the resolution of the simulation, while the amount of values inside each cell and therefore the third dimension of the array depends on the discretisation of the velocity or more specifically on the amount of discrete velocities in the velocity set. As a result the D2Q9 has to save nine values for the discrete velocities in each cell. Each entry of one cell is considered to be the particles population of the corresponding discrete velocity and represents the density of the populations in this cell that has this velocity. The order of these velocities inside one cell is not fixed throughout literature. One possibility is to start with the velocity that has the shortest length and progressively increase it while always starting in the top left corner. This would mean that the first entry of each cell is the stationary velocity and the last one is the velocity pointing in the bottom left. The only important thing for one implementation is to keep the mapping from index to velocity consistent.

2.2.2 Updating Macroscopic Moments

In chapter 2.1.2 it was mentioned that it can be proven that the density distribution is in fact a valid and complete description of a system if macroscopic moments are of interest. The first moment of interest is the local density $\rho(\boldsymbol{x}, t)$. The calculation is done simply by evaluating the sum of all populations in one cell. The local velocity $\boldsymbol{u}(\boldsymbol{x}, t)$ can be found by multiplying each population with its discrete velocity vector $\boldsymbol{c}_i$.

$$\rho(\boldsymbol{x}, t) = \sum_i f_i(\boldsymbol{x}, t) \qquad\qquad \boldsymbol{u}(\boldsymbol{x}, t) = \sum_i \frac{\boldsymbol{c}_i f_i(\boldsymbol{x}, t)}{\rho(\boldsymbol{x}, t)}$$

The main reason why the macroscopic moments are computed is because their values have to be known for the calculation of the equilibrium distribution function. Additionally, they are a great way of visualising the behaviour of a fluid over time.

2.2.3 Calculating the Equilibrium Distribution

The kinetic theory describes how a fluid, that is left alone for long enough, tends to its local equilibrium state. This state is not to be confused with the global equilibrium of a fluid, although the same principle can be applied. In the case of fluids, the particles tend to the Maxwell-Boltzmann distribution. The discretisation of this distribution is crucial and has a great impact on the accuracy of the result. The most commonly used example of this is the following description equal to the equation derived in section 2.1.4[1]:

$$f_{i,2}^{eq}(\boldsymbol{x}, t) = w_i \rho \left(1 + \frac{\boldsymbol{u} \cdot \boldsymbol{c}_i}{c_s^2} + \frac{(\boldsymbol{u} \cdot \boldsymbol{c}_i)^2}{2c_s^4} - \frac{\boldsymbol{u}^2}{2c_s^2}\right)$$

The density ρ and velocities $\boldsymbol{u}$ are dependant of time and change during each iteration of the TSA which shows why the macroscopic moments have to be calculated. The speed of sound c_s is a fixed connected to the lattice. For the D2Q9 lattice this value is equal to $c_s = 1/\sqrt{3}$

2.2.4 Collision and Streaming

After calculating the equilibrium distribution it is possible to simulate the local behaviour of the population in each cell by calculating the collision. This can be achieved by applying a change to the current population. This change depends on the collision operator that is used for the simulation. Every implementation created in the scope of this thesis uses the BGK collision operator introduced in chapter 2.1.2, as it is the simplest one to implement and one of the most studied operators throughout the literature. Additionally, it falls under the category of single relaxation time models and therefore doesn't have the same degree of freedom when compared to other collision operators, which is a good thing because it makes the stability comparison of different lattices easier.

After simulating the collision inside a single node, the only thing missing is the actual propagation of the particles to neighbouring nodes. This is achieved during the streaming step. The combined collision and propagation step can be formulated using the following equation:

$$f_i(\boldsymbol{x} + \boldsymbol{c}_i, t + 1) = f_i(\boldsymbol{x}, t) \cdot \left(1 - \frac{1}{\tau}\right) + f_i^{eq}(\boldsymbol{x}, t) \cdot \left(\frac{1}{\tau}\right)$$

[1] $\boldsymbol{u}^2 = u_x^2 + u_y^2$ and $\boldsymbol{u} \cdot \boldsymbol{c_i} = u_x c_{ix} + u_y c_{iy}$ in two dimensions

The right hand side of this equation is the collision part. Both distribution are combined so they retrieve all important moments together. It can be shown that the parameter τ is related to the kinematic viscosity ν by the following equation:

$$\nu = c_s^2 \left(\tau - \frac{1}{2} \right)$$

This equation will become important later when comparing two different sized lattices, because they usually come with a different value for the speed of sound. To properly compare them, the kinematic viscosity of both simulations has to be the same, which means that the relaxation time has to be chosen accordingly. As a result it is usually easier for comparisons to define the relaxation rate implicitly using the kinematic viscosity of the flow.

After evaluation of the right hand side the streaming step is performed. Populations propagate to the neighbouring nodes according to their velocity vectors $\boldsymbol{c}_i$. This means that if the corresponding velocity of the current cell faces in the top right direction, the population is written in the cell to the top right. Periodic boundary conditions are applied which means that populations that leave at one side of the system re-enter it on the other side. Overall this means for the conventional implementation of the LBM that the populations can only travel to the nodes directly next to them. This behaviour is different for bigger neighbourhoods, since some of the particles propagate to nodes further away.

2.2.5 Multiple Relaxation Time

One important method that has not been talked about yet is using multiple relaxation times. The idea is applied to the collision step and the BGK collision operator, which can be seen as a simplification of a multiple relaxation time (MRT) model.

The previously explained collision was performed in population space, which means that the populations afterwards were calculated using only one relaxation time $\tau = \frac{1}{\omega}$ for all nine discrete populations. Another idea is to apply a linear transformation to this problem. Multiplying the identity matrix $\boldsymbol{I} = \boldsymbol{M}^{-1}\boldsymbol{M}$ to an equation does not change it. Therefore, the equation for the collision and the propagation can be rewritten as:

$$\begin{aligned} f_i(\boldsymbol{x} + \boldsymbol{c}_i, t+1) - f(\boldsymbol{x}, t) &= -\boldsymbol{M}^{-1}\boldsymbol{M}\omega \left[\boldsymbol{f}(\boldsymbol{x}, t) - \boldsymbol{f}^{eq}(\boldsymbol{x}, t) \right] \\ &= -\boldsymbol{M}^{-1}\omega\boldsymbol{I} \left[\boldsymbol{M}\boldsymbol{f}(\boldsymbol{x}, t) - \boldsymbol{M}\boldsymbol{f}^{eq}(\boldsymbol{x}, t) \right] \end{aligned}$$

In this case $\boldsymbol{M}$ is the transformation of the populations into moment space. Then the moments are relaxed in moment space using the relaxation matrix $\omega\boldsymbol{I}$ and finally transformed back to population space. The great thing about this procedure is that instead of using only one relaxation rate ω, nine different relaxation rates can be chosen, one for each moment. Three of these moments correspond to the conserved moments density and both velocities, therefore the relaxation rate doesn't matter. Additionally, the three second order moments determine the shear- and bulk viscosity and can't be chosen freely [14]. The other rates can be twisted for the purpose of creating more stable and accurate simulations. One example of using a MRT model to increase the stability is shown in [15]. Even though MRT models were not implemented for the benchmarks in this thesis due to the reasons mentioned in the previous chapter this whole idea will become important in chapter 3.3.2.

2.3 Implementation

It is possible to implement and simulate flows with the information provided in the previous chapter, but real simulations can be extended to enable more sophisticated simulations.

One topic that hasn't been covered yet are forces. For most real live applications it is necessary to take forces like gravity or the centrifugal force into account. Additionally, boundary conditions can potentially be implemented to handle the addition of solid walls to the simulation. Both ideas are disregarded from now on as they would not add information to this analysis of large neighbourhoods.

Within scope of this work the LBM was implemented with the following two goals in mind. The first thing is, that the program is supposed to be adaptable. There are many different stencils

the program has to work with and, while some parts of the algorithm are independent for different lattices if implemented correctly, there are some things that have to be changed. This dependency has to be removed. Switching between different lattices in the final version is as easy as changing one string. Additionally, the program has to be able to run with and without code generation. What exactly code generation means will be discussed in chapter 3.3. This fact will be used later for performance comparisons to show the benefit of using code generation.

The second point is, that two different implementations are supposed to be easily comparable. In the rest of this thesis a bunch of comparisons are made between the implementations of large neighbourhoods and using or not using code generation to quantify and analyse the difference those approaches make. The simulations run in `Python3` and the source code and installation instructions, as well as most of the visualisations can be found on GitLab [16]. The program can be split into two parts that each serve a different purpose. The first is mandatory, while the second is only used for visualisation and ease of usage. Comparisons that are done in this thesis are saved separately and can be found in the *visualize.py* file. The main part is inside of the *lbm.py* file. It consists of the class *LBM* that is used to describe the complete system and the iteration steps. Every computation that doesn't include code generation is done using *NumPy*, one of the most important `Python` packages for scientific computations [17]. The implementation of the different lattices is done using the package *lbmweights*, explained in chapter 3.2.1 [18].

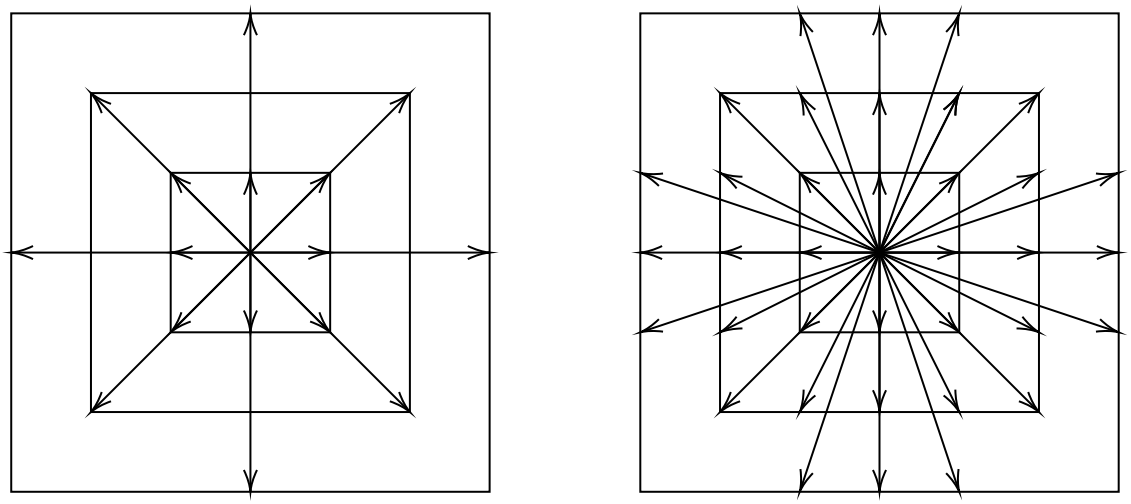


Figure 2: D2V17 (left) and D2V37 (right) lattices proposed by Philippi et al. [6]

3 Large Neighbourhoods

Now that a basic understanding of the LBM has been established, the focus of this thesis switches over to understanding and implementing large neighbourhoods. In the first part of this chapter, the meaning behind large neighbourhoods for the implementation is explained. Then code generation is added to significantly increase the performance and the resulting code is used to analyse the large neighbourhoods based on performance, stability and accuracy.

The idea of increasing the lattice size is nothing new and in general the idea is pretty obvious. Increasing the amount of discrete values or the amount of summands considered before truncation usually leads to a better result. Therefore, it is not very surprising that some larger lattices have been introduced only a few years after the initial model. One example is the lattice introduced by Fahner in 1991 with 21 discrete velocities as one of the large neighbourhood schemes for the predecessor of the LBM [19]. The D2Q17 lattice for the LBM that has been analysed by Qian et al. [20] in 1998 showed some promising advantages but was numerically unstable [3].

A big step was made by Philippi et al., who has introduced a model that includes not only the large neighbourhoods as shown in figure 2 but also a formula for the calculation of the higher order accurate equilibrium distribution function [6]. Spiller et al. went even further for the velocity discretisation by developing an algorithm that calculates velocity sets for any dimension by solving the equations necessary for the isotropy shown in chapter 2.2.1 up to an arbitrary order [13]. Chapter 3.2 summarises, refactors and extends this algorithm while using the higher order accurate formula for the equilibrium distribution to show the advantages and disadvantages of large neighbourhoods.

3.1 Implementation of Large Neighbourhoods

The term large neighbourhoods can be understood by looking at the chapter 2.2.1. It introduced the D2Q9 lattice and it was also mentioned that during the propagation step the particles are simulated to propagate to neighbouring nodes. The term neighbourhood therefore corresponds to all the nodes that are reachable by the used velocity set from the current node. If one would increase the velocity set and include velocities that reach from the current node to, for example, the node two steps to the left, this newly added node would thereafter become part of the current's node neighbourhood. This is because particle distributions with this velocity are able to directly propagate from the current node to the new node and the other way around during one single time step. In this sense the D2Q9 is regarded as having a *normal* neighbourhood and every larger velocity set has a *large* neighbourhood.

The two velocity sets used to analyse the advantages and disadvantages of large neighbourhoods in this thesis can be seen in figure 2. The left one is called D2V17. The innermost part of the lattice

is still the same, while two outer energy shells have been added. At first glance it might seem odd that the w_{20} energy shell is missing, but Philippi et al. [6] showed that this layer is not needed to calculate the equilibrium distribution up to a third order accuracy. This is also the reason for the odd naming convection. If this lattice was named D2Q17, the user might expect the lattice that consists of the D2Q9 lattice plus the two energy shells $w_{20} \neq 0$ and $w_{22} \neq 0$. Furthermore a lattice named D2Q17 already exists.

The right lattice is called D2V37 and can be considered the "minimal square lattice giving a fourth-order approximation to the continuous Boltzmann equation" [6]. To implement both it is not sufficient to change only the velocity sets, but additionally the calculation and the discretisation of the equilibrium distribution function has to change. The formula for the third and fourth order accurate equilibrium distribution function have been derived by Philippi et al [6]. The equations for the isothermal equilibrium distribution can be summarised by the following equations [15]:

$$f_{i,3}^{eq} = f_{i,2}^{eq} + w_i \rho \left[\frac{1}{6c_s^6} (\boldsymbol{u} \cdot \boldsymbol{c}_i)^3 - \frac{1}{2c_s^4} \boldsymbol{u}^2 (\boldsymbol{u} \cdot \boldsymbol{c}_i) \right] + O(\boldsymbol{u}^4)$$
$$f_{i,4}^{eq} = f_{i,3}^{eq} + w_i \rho \left[\frac{\boldsymbol{u}^4}{8c_s^4} - \frac{\boldsymbol{u}^2}{4c_s^6} (\boldsymbol{u} \cdot \boldsymbol{c}_i)^2 + \frac{1}{24c_s^8} (\boldsymbol{u} \cdot \boldsymbol{c}_i)^4 \right] + O(\boldsymbol{u}^5)$$

It can be seen that increasing the order of accuracy by one increases the complexity of the calculation but also the order of the error term $O(\boldsymbol{u})$. In order to increase the function to a third or fourth order accurate calculation, terms depending on the third or fourth power of the velocity have to be added respectively. The power of the speed of sound terms increases by two.

Changing those two parts is already basically enough to implement large neighbourhoods, if the rest of the algorithm is implemented in a lattice independent fashion.

This means, that the computation time will definitely increase for all the steps described in 2.2, though extra work for the implementation is only necessary in the *update equilibrium* part of the algorithm.

3.2 Semi-Automatic Construction of Lattice-Boltzmann Models

D. Spiller and B. Duenweg have recently been working on the semi automatic construction of velocity sets [13]. This means that they have implemented a `Python` script which has been made publicly available, and enables the user to create lattices with as much as 221 discrete velocities and theoretically even more [7]. Even though their script works nicely to find new lattices interactively, the algorithm was refactored so it is available as a `Python` module, which makes its usage for the implementation of the LBM easier. Additionally, some functions and properties were added and some parts not necessary for this thesis removed. In the following part the main idea that is described more thoroughly in their paper, is presented with a focus on the parts that differentiate both implementations.

In chapter 2.1.4 the isotropic moments up to the fifth order were introduced. These moments can be re-written in a more general form also known as the Maxwell-Boltzmann constraints (MBCs) [13]:

$$\sum_i w_i c_{i\alpha} c_{i\beta} ... c_{i\nu} = \int d^d \boldsymbol{v} f(\boldsymbol{v}) c_{i\alpha} c_{i\beta} ... c_{i\nu}$$

The assumption is made that the weights of the velocities in one velocity shell have to be equal. The amount and the squared length for each energy shell of the lattice are expected as input. The MBCs have to be fulfilled for all tensors up to the order of isotropy given by the user. These equations can be then be transformed into a linear equation system. This linear equation system describes the connection between the speed of sound and the weights. At this point the speed of sound is no longer considered a fixed parameter. They argue that the weights have to be polynomials in c_s^2 and therefore the resulting linear equation system constructs a set of polynomials, one for the weight of each velocity shell. This means that entering a value for the speed of sounds result in the exact value for the weight of each of the energy shells.

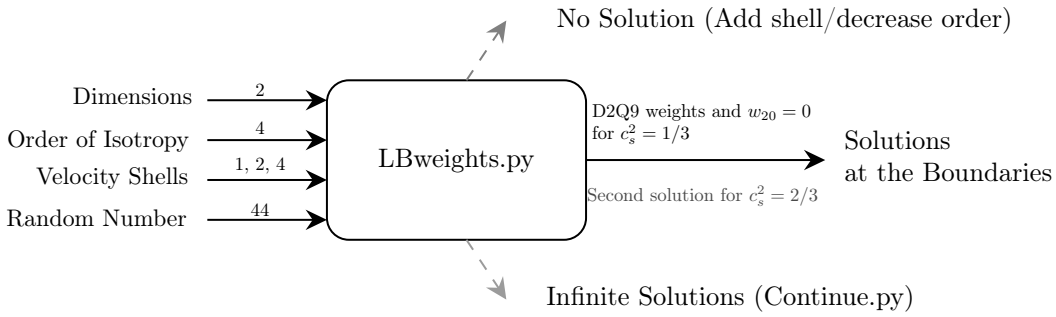


Figure 3: Derivation of the weights of the D2Q9 velocity sets using LBweights [7]. Values above the arrows on the left side are expected as input and on the right side are the output values.

The linear equation system is solved using a singular value decomposition. The shape of the matrix in the linear equation system depends on the amount of shells used as input of the script and the order of isotropy. If the order of isotropy is too high for the amount of shells provided, then the resulting equation system is underdetermined. This can be handled by another script published in the same repository but is disregarded from now on due to the fact that all of the velocity sets important for this thesis can be constructed without it. If the order of isotropy is too small no result can be guaranteed, though the script handles cases in which some singular values are small enough and can be ignored.

At this point another important restriction to these weights is considered, namely the necessity that the weights can't be negative. This means that the final part of the script finds one interval where all the polynomials are positive. This is done by looking at the real valued roots[1] of the polynomials, evaluating one value inside of every interval created by these roots and checking if the polynomial inside this interval is positive. If one interval exists in which every polynomial is positive, then every value for c_s^2 can be used as input to calculate the final weights. However, since the boundaries of this interval correspond to the roots of at least one polynomial the rational choice is to pick either the infimum or the supremum of the interval to reduce at least one weight to zero and therefore the complexity of the resulting velocity set.

One major part of the original script and paper is finding the correct input parameters to retrieve lattices that are known throughout literature and it has succeeded for some of the most important lattices of the LBM. The only difficulty is finding the right input values, most importantly the right amount and length for the velocity shells. The seed (necessary for the transformation to the linear equation system) has in most cases basically no influence but the types and the amount of shells is very important. This can sometimes lead to counter intuitive input parameters. If the shells 1 and 2 (0 is always implicitly part of the solution) are entered no solution is returned. D2Q9 lattice shows that entering the expected velocity shells with a-priori knowledge does not guarantee a result. To get the right result for this lattice, the user has to supply an additional velocity shell as input, that will later turn out to be a possible candidate for the reduction to zero, as shown in figure 3.

3.2.1 Python Module lbmweights

Having a semi-automatic tool to construct lattices that are suitable for the LBM is great, but so far it was only available as a script which made the integration more difficult than necessary. Therefore, the script was refactored without any major changes to the algorithm in order to make it usable as a `Python` package. The good thing about packages is, that they can easily be imported inside other scripts or modules. Additionally, this newly developed package provides some easy to use functions that don't require an understanding of the algorithm behind it. The resulting code, however, is a little bit slower compared to the original script due to some changes that were implemented. The reasoning behind this will be discussed later in this chapter.

The source code and an instruction on how to install the package are available online [18]. The following use case of the package is the one most similar to the original one and leads to the D2Q9

[1] Due to possible errors introduced by the algorithm roots with small enough imaginary part are also considered real

Lattice: [float]

Listing 1: Basic usage of *lbmweights*

```
from lbmweights import Lattice
lattice = Lattice(dimension=2, order=4, shell_list=[1, 2, 4], seed=44)
lattice.calculate_weights(boundary="inf")
c_s_sq, weights, velocities = lattice.velocity_set()
print(c_s_sq)  # 1/3
print(weights)  # [4/9, 1/9, 1/9, 1/9, 1/9, 1/36, 1/36, 1/36, 1/36]
print(velocities)  # [[ 0.  0. -1.  1.  0. -1.  1. -1.  1.]
                   #  [ 0. -1.  0.  0.  1. -1. -1.  1.  1.]]
```

In the first line the class *Lattice* of the `Python` package *lbmweights* is imported. This class is the main part of the package and sufficient for most use cases. Afterwards an instance of the class is created with the same parameters that are used for the original script. At this point the algorithm described by Spiller et al. has not started. This only happens once one of the high level functions like *calculate_ weights()* are called or if the user requests properties that are only available after calculation, like the weights of the lattice. The resulting weights depend on the boundary therefore it is possible to give this function a value for the speed of sound parameter or the preferred side of the boundary. In this case the infimum of the boundary, i.e. the highest value that is lower or equal to all the values inside the valid interval, is used. This value directly corresponds to the value that will later be returned as c_s_sq ($= c_s^2$). Infimum means this value will be as low as possible and Supremum means that this value will be as high as possible. If a value for c_s_sq is provided, the program will check whether or not this value lies in the valid interval and if it does, calculate the lattice for this value. Note that, due to the fact that a value inside the interval for the speed of sound is valid, but does not correspond to the root of one of the weight polynomials, no shell will be reduced and a lattice with the same order of isotropy but a higher amount of discrete velocities is created.

In the fourth line a function is called that is supposed to make the use of the lattice for the actual simulation easier. All the values that are necessary for a proper initialisation are returned so they are ready to use. If one velocity shell can be reduced to zero, the weight value for this shell is not returned and the velocities are not added to the velocity set. This line shows how the D2Q9 is indeed correctly retrieved, as the different values in the weights array and the value for the speed of sound are equal to the values shown in chapter 2.2.1 and the velocity array simply corresponds to the velocity vectors in each dimension shown in figure 1.

To verify the correct behaviour of this package, a few test cases have been implemented, that simply check that the package behaves as expected for some of lattices that have been retrieved by the original script. This includes the derivation of the most important Lattices with large neighbourhoods introduced in chapter 3.1. Additionally, some test cases are added to verify the correct behaviour for some of the edge cases like three-dimensional lattices or subshells.

Subshells are one of the main differences from the implementation of the original script to the package *lbmweights*. The definition of the velocities using a shell list is not completely unambiguous. One example is the velocity shell with squared length of 25. Both the shells corresponding to the weights w_{43} and w_{50} are possible candidates for the shell, but the weights of them can be different. The original script provides functionality to try creating a model with both subshells or interactively removing either of them. To automate this process, this information has to be provided explicitly by the user of *lbmweights* when the class is created. Another difference of the original compared to *lbmweights* is the existence of the *Continue.py* script. It provides a way to create a lattice and solve the problem if infinitely many solutions are possible. The main goal of the package, however, is not to discover new lattices, but to make the use of lattices discovered by the script easier. Therefore, it is not needed. Additionally, the last part of the algorithm, that calculates the valid intervals is implemented using *SymPy*, a `Python` package for symbolic calculations. This is done to make the implementations of some additional functionalities and the integration of code generation easier. Additionally, the *SymPy* code is more readable. The only disadvantage is that it is slightly slower than the equivalent *NumPy* implementation used in the original script, but that can be ignored because the lattice is only calculated once prior to the LBM.

All of the above mentioned reasons lead to the main advantage of *lbmweights*. The user of

this package needs minimal prior knowledge to use and implement large neighbourhoods. This is achieved by providing a few different ways of how to initialise the *Lattice* class. The three following ways described the possible initialisation schemes:

Listing 2: Initialisation schemes of *lbmweights*

```
from lbmweights import Lattice
lattice = Lattice.from_order(dimension=3, order=6)
lattice = Lattice.from_name("D3Q41-ZOT")
lattice = Lattice(dimension=3,
                  order=6,
                  shell_list=[1, 2, 3, 9, 16, 27],
                  boundary="sup",
                  unwanted_subshells=["221", "511"])
```

All of the lattices created in the example above result in the same lattice called D3Q41-ZOT as introduced by [21]. The first way of initialising a lattice requires basically no knowledge. The idea is that a user only has to enter the order of isotropy and the dimension he wants to use and if the package knows a lattice that has those properties it is returned. The second possibility requires the user to know the name of the lattice he wants to use. This is probably the most useful initialisation scheme since the names of the lattices are pretty consistent throughout literature. However, both schemes have in common that the desired lattice has to be supported by *lbmweights*. This is done by saving the input parameters that lead to the lattice with this name inside a dictionary. *lbmweights* already provides a bunch of lattices that are implemented, most of them in two dimensions. A list of all of them can be accessed by calling `Lattice.BY_NAME`.

The third method is the only way to discover new lattices. A lot of knowledge about the underlying lattice is required here, but it could be used to manually search for a new lattice. The D3Q41-ZOT lattice is a great example, because it has some extraordinary properties that have to be set in order to retrieve it. The boundary of this lattice has to be the supremum, while every other lattice known by the package is using the infimum and therefore the smallest possible value for the speed of sound that results in only positive weights. Additionally, it is the only lattice implemented that has to exclude subshells from the result. That means, while the shell 27 in the shell list should theoretically include both the subshells w_{511} and w_{333}, the former has to be removed for this program to finish. The original script asks the user what subshells are supposed to be used, while *lbmweights* takes a list of unwanted subshells as an argument. Additionally, the seeds were removed or rather fixed for the pre-defined lattices. This was done to guarantee that asking for the same lattice leads to the same result every time.

The most significant change of *lbmweights*, however, is the addition of code generation. The lattice created by the package already provides every information necessary to create a complete simulation of the single relaxation time LBM, except for boundary conditions. This means that the package is able to create the large neighbourhood LBM if only the desired order of isotropy or the name of the lattice is known. More about that in chapter 3.3.1.

3.3 Code Generation

There are a lot of reasons why a developer would want to implement the LBM in `Python`. First and foremost its syntax is pretty easy to read, making the code useful for a bigger audience. Additionally, the programmer does not have to declare data types and casting is often done in the background, and most importantly due to its popularity, a lot of open source packages exist, some of which were used throughout this work to significantly reduce the development time. One could probably make a case why other programming languages like `MATLAB` or `C++` may be better for numerical or scientific computations, but there is no definitive answer. Usually the question evolves around computation time versus development time. It can be said that `Python` is usually slower when compared to `C++`

Code generation is a way to combine the best of both worlds. The result will look like normal `Python` code, so its very accessible to people with little knowledge about the programming language itself, but the calculation of all the time consuming parts of the algorithm, most notably the whole TSA, will be done with generated `C++` code. Code generation and all the related functionalities are implemented using the package *pystencils* [22].

Listing 3: Calculating the density for a two-dimensional lattice with *lbmweights*

```
import pystencils as ps
import numpy as np
from lbmweights import Lattice
lattice = Lattice.from_name("D2Q9")

def update_density_kernel(Q):
    data = ps.fields('data({Q}):[2D]'.format(Q=Q))
    density = ps.fields('density:[2D]')
    density_formula = sum([data[0, 0](i) for i in range(Q)])
    symbolic_description = [ps.Assignment(density[0, 0], density_formula)]
    return ps.create_kernel(symbolic_description).compile()
kernel = update_density_kernel(lattice.q)

density = np.zeros((3, 3))
kernel(density=density, data=np.ones((3, 3, lattice.q),
                                     dtype=np.float64))
density  # [[9, 9, 9], [9, 9, 9], [9, 9, 9]]
```

The main purpose of this package is increasing the performance of stencil codes. Stencil codes are calculations, that only depend on arithmetic operations between one node and its relative neighbours and the same operation has to be performed on each cell. Additionally, contrary to usual loops, the order of the cells on which the calculations are performed is not fixed due to the inherently parallel nature of *pystencils*. However, this isn't much of a restriction for the LBM because every part of the it meets both criteria. The basic idea of those kernels is always the same. At first the fields are defined. They specify how the input and output should look and what dimensions of the fields are supposed to be iterated over. Afterwards the operations they are supposed to execute are added and finally the kernel is created, compiled and returned. This means that the kernel can be executed in `Python`. During every iteration one or multiple kernels are called and the generated code is executed.

Listing 3 shows how a kernel used to calculate the density of an arbitrary two-dimensional lattice can be created. Conceptually this code can be divided into three parts, indicated by the space between them. At first the packages are imported and afterwards a lattice is created. In this case the usual D2Q9 lattice is used but the kernel also works for other two-dimensional lattices.

The second part consists of the definition of the kernel and it's compilation. The function takes only one parameter which corresponds to the amount of discrete velocities of the lattice. Then two fields are created. Fields in *pystencils* are basically multidimensional arrays with the main difference that some of the elements in it are considered spatial dimensions, whereas others are referred to as index dimensions [22]. The difference becomes clear when looking at the data field, since it uses both types of dimensions. This field is initiated by a string holding the information for the dimension of both types. The amount of spatial dimensions, defined inside the square brackets, indicates the amount of physical dimensions of the underlying simulation. In this case both fields are two-dimensional. Chapter 2.2.1 described how the two-dimensional LBM needs a third dimension to store the populations of each discrete velocity. In *pystencils* implementations this third dimension is implemented as an index dimension, indicated by the round brackets inside of the string[1]. The output field however has only two spatial dimensions and no index dimensions because the density is only defined for every node, which can be seen from the formula for the calculation of the density introduced in chapter 2.2.2. For a real implementation the example shown above could be made even simpler and more general, because the *Lattice* object holds information about both the dimension and the amount of index dimensions needed. This would make the density kernel even more general, but was left out for the sake of simplicity.

Naturally the question arises why this is necessary. The *NumPy* implementation works perfectly fine with just one type of dimension. The simple answer is that the code created by this kernel doesn't iterate over index dimensions. Remember that the package is optimised for stencil codes and therefore the same operation is performed for every cell of the input field, in this case data.

[1]The curly brackets inside of the string have nothing to do with *pystencils*, but with the formatting of the string. 'data({Q})'.format(Q=9) for example becomes 'data(9)', making the whole initialisation equivalent to data = ps.field('data(9):[2D]') in the case of a D2Q9 lattice.

Listing 4: Calculating the density in *NumPy*

```
import numpy as np
data = np.ones((3, 3, 9))
density = data.sum(axis=2)  # [[9, 9, 9], [9, 9, 9], [9, 9, 9]]
```

Listing 5: Assignments for one possible propagation kernel of the D2Q9 lattice

```
symbolic_description = [
    ps.Assignment(data_out[0, 0](0), data_in[0, 0](0)),
    ps.Assignment(data_out[0, 0](1), data_in[-1, 0](1)),
        ...
    ps.Assignment(data_out[0, 0](8), data_in[1, 1](8)),
]
```

The stencil operation is only supposed to be performed on every spatial dimension, while the index dimensions can and will be used to create complex connections between different cells or entires of these cells.

The equivalent *NumPy* implementation shown in listing 4 is not different. In this case the sum is evaluated over the third dimension which leads to the same result as the *pystencils* implementation. The dimensions in *NumPy* are not named differently but they are treated differently.

After the definition of both fields, the formula used to calculate the densities has to be defined. In this case this is simply a summation over all the populations as described in 2.2.1. Note, that this summation is not arithmetic but symbolical. *pystencils* simply gets told what operation it is going to have to do, but hasn't seen an actual value, yet. The zeros inside the square brackets denote what neighbour relative to the current node is supposed to be accessed. In this case the operation is a simple summation over all the index entries of the current cell. The result of this summation is then written in the output array, again without any kind of neighbour access using an assignment. All the operations are saved as a list of Assignments, in case multiple Assignments are supposed to be executed after each other. Finally the kernel is created and then compiled. The initialisation and compilation of these kernels only has to be done once before the first iteration of the TSA starts. The compiled kernel is then saved as a callable object in `Python` making its usage just as easy and short as the *NumPy* code and the expected arguments of the kernel coincide with the fields initialised at the beginning of it.

Listing 5 further elaborates on the stencil part of this package and shows how neighbour accesses work. The assignments created can be used as a part of the propagation kernel. The values of a cell after the propagation step depend on the corresponding value of a neighbouring cell, which means they always have the same index dimension because their velocity remains the same, but the spatial dimension is different. The change of their spatial dimensions is defined by the index cell and the one discrete velocity this index cell is supposed to represent. This can be seen by looking at the indices inside of the square brackets. They always correspond to the relative location of the cell when iterating over the whole field. The last assignment for example shows the 8th discrete velocity that in this case represents the population that is supposed to move one cell in the first and one cell in the second direction. This means that during each step of the iteration over the cells, only the current cell and its values change, but the new values originate from neighbouring cells. This is the same idea as saying that for one cell all populations propagate to their corresponding neighbour. The former is called a *pull kernel* and the latter is called a *push kernel*.

This knowledge alone is already enough to create faster code in `Python` and more importantly to calculate almost everything necessary for the TSA, with the exception being boundary conditions.

3.3.1 Assignment Collection in lbmweights

pystencils has a class called *AssignmentCollection* that allows for multiple assignments to be grouped together. This class can be combined with *lbmweights* to create a collection that can be used as a fully functioning model for single relaxation time simulations with arbitrary velocity sets, where important parameters like the relaxation time or the size of the simulation can still be

chosen. By calculating the weights, the velocities and the speed of sound of the desired velocity set, the lattice class returned by *lbmweights* already contains all information needed for a kernel that takes one data field as input and outputs the data fields after one iteration of the TSA.

The major difference of the assignment collection returned by *lbmweights* compared to the kernels introduced in the previous chapter is that the kernel is compiled, while the assignment collection still has to be compiled. This means, that other things like synchronisation or data handling can be added if needed, as described in chapter 3.4.

However, there is one major change that has to be applied to the complete algorithm in order to implement the LBM using only one kernel. The normal implementation uses what is known as a *collide-stream* method. First the collision is simulated by calculating the macroscopic values and with them the discrete Maxwell-Boltzmann distribution. Afterwards the populations stream to the neighbours. This can be implemented the other way around. This means that the populations first stream to neighbouring nodes and only afterwards collide locally. Therefore, the moments retrieved after one iteration, that are for example used to visualise a fluid or quantify the accuracy, are calculated after collision instead of after streaming. It was mentioned in chapter 2.1.2 that the collision has to conserves macroscopic moments and therefore both versions are equivalent. The fact that the new algorithm starts with a streaming step instead of a collision step might seem confusing at first but can for example be justified by the fact that the field can be initialised by calculating the equilibrium distribution for some initial condition provided by the user and setting the field to this exact value. Therefore, the simulation technically starts with a collision, as well. The reason why this distinction and change to the algorithm is necessary will be explained in chapter 3.4.

3.3.2 Creation of Moment Based Models with lbmweights

Chapter 2.2.5 introduced multiple relaxation time models of the LBM by converting the populations, and more importantly, the collision step from population into moment space using a moment matrix. This matrix can be calculated directly from the chosen discrete velocity moments. The velocity moment of order zero corresponds to the density, the first order moments x and y to the velocities in each direction, the second order moments are equal to x^2, xy and y^2 and so on. The moment matrix of the D2Q9 lattice for example consists of all the discrete velocity moments of order zero up to three and the symmetrical fourth order moment x^2y^2. These moments and therefore every row of the moment matrix can be explicitly calculated [14]. The general formula for this calculation is:

$$x^a y^b = \int v_x^a v_y^b f d\boldsymbol{v} = \sum_q c_{qx}^a c_{qy}^b f_q$$

The zeroth discrete moment (a=b=0) therefore corresponds to a simple summation of all populations which is the same thing as calculating the density. Finding the correct moments for those higher order models would mean that MRT relaxation schemes could be applied to potentially further increase the stability of the LBM, just like the MRT model of the D2Q9 lattice [15]. Even more so, if the procedure of finding the right moments for larger lattices could be generalised, this would mean that *lbmweights* could not only provide large lattices up to an arbitrarily high order of isotropy but would additionally provide the right moments and therefore the moment matrix for multiple relaxation time models. Since this matrix has to be applied to the equilibrium distribution as well, and the equilibrium moments can be analytically calculated from this distribution, the choice of moments directly corresponds to the definition of how to calculate the equilibrium distribution. Therefore, the choice of these moments would ideally result in the same description on how to calculate the equilibrium distribution or another description that is not identical but has similar properties.

For the D2V17 lattice the basic idea is that since the D2Q9 included moments up to fourth order the new lattice has to include moments up to the sixth order. This results in overall 28 potential moments, which means that the resulting moment matrix is 28×17 and therefore not invertible. It is obvious that this matrix has to be invertible since that is the only condition for the matrix introduced in chapter 2.2.5. Calculating the inverse matrix is a necessity for MRT models to convert the calculated values after the collision back from moment to population space. The idea is to reduce the amount of rows in this matrix while keeping its rank up at 17 so the matrix stays invertible. The equilibrium distribution has to be accurate up to the third order, therefore none of

Listing 6: Pseudocode for the creation of MRT models for lattices with large neighbourhoods. The complete example seen in *lbmweights/mrt_moments.py* [18]

```
d2v17 = Lattice.from_name("D2V17")
selected_moments = moments_up_to_order(6, dim=2)
m = moment_matrix(selected_moments, d2v17.velocities)  # Shape (28,17)
selected_moments = remove_duplicates(m)  # Shape (22,17)
selected_moments = remove_five_more(selected_moments)  # Shape (17,17), Rank 17
method = create_with_continuous_maxwellian_eq_moments(lattice.velocities,
                                                       moment_to_relax_rate_dict,
                                                       compressible=True,
                                                       equilibrium_order=3,
                                                       c_s_sq=lattice.c_s_sq)
assert method.weights == d2v17.weights
```

the moments up to the third order should be removed. This is no problem, since the rows created by these first 10 moments are not linearly dependant and therefore they can all be part of the final matrix. Additionally, some rows created by higher order moments are completely equivalent (e.g.: x^3y and xy^3). This means that if both rows were to stay, the matrix is not going to be invertible. Therefore, one row can be removed. The resulting moment that is included can now be interpreted as half the sum of both moments, which means that the simulation is still isotropic. This only happens for the derivation of the moments of the D2V17 lattice but not for the D2V37. Overall this results in a reduced matrix of the shape 22×17, which means that five more rows have to be removed. Getting rid of the last five moments is done manually, but can potentially be automated. The idea is to start with the highest moments and symmetrically removing them (e.g.: x^6 and y^6). Afterwards it is tested if the resulting matrix still has the expected rank. If the matrix is not invertible anymore the moments removed in the last step are added again. This is done until the amount of moments is equal to the amount of rows or until only one more row has to be removed. In the latter case the same procedure is repeated with symmetrical moments(e.g: x^3y^3).

The choice of these moments can now be tested by a function of the module *lbmpy*[23]. This module allows for fast fluid simulations based on the LBM. Every method of this module is based on these moments, their corresponding equilibrium moments that can analytically be computed from the Maxwell-Boltzmann distribution, the velocity set and the relaxation rates. One of its functions allows for a model of the LBM to be created from moments and the velocity set alone.[1] The relevant parameters are the moments that have just been chosen, the velocities from the lattice returned by *lbmweights* the value for the speed of sound and the order of the equilibrium. Afterwards the function retrieves, amongst other things, the weights corresponding to the discrete velocities and the moments used as input. It turns out that if the moments chosen by the method described above are used as input for this function, the weights returned are equal to the weights of the D2V17. Unfortunately the equilibrium moments and therefore the complete method calculated this way does not coincide with the proposed lattice. Additionally, there are some other combinations of velocity moments available that lead to a non-singular matrix and the correct weights, but not to a correct model. Running a simulation with these newly calculated equilibrium moments doesn't result in a correct simulation.

Listing 6 shows pseudo code how this was implemented. The last line shows that the chosen moments together with the velocity sets lead to the exact weights of the originally proposed D2V17 model. The same idea has been implemented for the D2V37 lattice, which is equal to the original model with a small numerical error. The source code for both models retrieved by this method is located in the function *get_moments* in *lbmweights/mrt_moments.py* [18].

3.4 Performance Benchmarks

With the introduction of large neighbourhoods and the introduction of code generation there are a bunch of interesting performance benchmarks possible. Obviously the comparison of the performance of the LBM with and without code generation is one of them. Another Benchmark that is of

[1] The function is called *create_with_continuous_maxwellian_eq_moments* and is located inside the *methods* submodule

Listing 7: Assignments that can be used to create a kernel that calculates the second order accurate equilibrium. c_ix and c_iy correspond to the values of the discrete velocities in each direction and Q to the amount of discrete velocities

```
density = ps.fields("density:[2D]")
u_x = ps.fields("u_x:[2D]")
u_y = ps.fields("u_y:[2D]")
f_eq = ps.fields("f_eq({Q}):double[2D]".format(Q=Q))
symbolic_description = []
for i in range(Q):
    formula = weights[i] * density[0, 0] * (1
            + (u_x[0, 0] * c_ix[i] + u_y[0, 0] * c_iy[i]) / c_s_sq
            + (u_x[0, 0] * c_ix[i] + u_y[0, 0] * c_iy[i]) ** 2 / (2 * c_s_sq ** 2)
            - (u_x[0, 0] ** 2 + u_y[0, 0] ** 2) / (2 * c_s_sq))
    symbolic_description.append(ps.Assignment(f_eq[0, 0](i),formula))
```

interest is the performance of the D2Q9 lattice compared to the D2V17 lattice. Initially one could expect roughly twice the computation time due to the increased lattice sizes, but the calculation of the equilibrium distribution is more costly as well. Almost all of the benchmarks can be found on GitLab inside the *visualize.py* file with an instruction on how to recreate the results [16].

A first quick comparison that can be made is comparing the average execution time of one iteration of the *NumPy* implementations of both lattices. For a domain size of 100 in both directions and 1000 iterations the execution time is about three times as long. As expected both the collision and the propagation step take about two times longer, while the calculation of the equilibrium distribution takes about seven times as long. This is only true for a somewhat improved calculation of the equilibrium distribution function, though. Some recurring calculations are saved temporarily to save computation time, which has a bigger impact on the more accurate calculation due to the fact that it has more of them. The performance of the large neighbourhoods would be far worse, if this step was not performed. However, for a fair comparison of the *NumPy* and *pystencils* implementation no simplification is made in either case.

One of the great things about *pystencils* is that it is built on top of *SymPy* and the kernels are compiled before the simulation starts. As a result some operations that are difficult or time consuming to implement can be defined much more easily. Listing 7 shows that defining the assignments necessary to compute the equilibrium distribution function is as easy as writing the formula in `Python`. Obviously the generated code doesn't contain the for loop anymore, because it only defines a bunch of calculations that are supposed to be done after each other. Additionally, the values for the weights and the speed of sound are explicitly written into the generated code. Furthermore simplifications that emerge once the values for the discrete velocities are evaluated (e.g. c_ix and c_iy are both zero for the stationary velocity), are automatically removed with *SymPy*. This leads to the generated `C++` code, that can be observed by calling the *ast* property of the kernel. Obviously all of these things can be done for the *NumPy* implementation as well, but this task is very tedious and depends on the form and size of the lattices. Automating it would be the same thing as code generation but the resulting code in this case would be in `Python` and therefore slower. As a result the `Python` code should not be written inside a for loop, or if it is the whole implementation of the LBM is about ten percent slower. Consequently the calculation of the equilibrium distribution has to be implemented independently for every discrete velocity. The result are 100 lines of code for the calculation of the equilibrium distribution of one specific velocity set. Therefore, code generation is not only faster but also leads to better readability of the code.

To elaborate on this statement, the next thing that is done is comparing the performance of the *NumPy* and the *pystencils* implementation of the V17 stencils with each other. A first comparison that can be made is implementing the LBM using multiple kernels, one for every step described in chapter 2.2. The performance gain for each part of the algorithm can be seen in figure 4. The uppermost bar describes the time step algorithm, which means it is roughly the sum of all the others, because during one call of the *time_step* function, every other function gets called. This means the overall algorithm in this case is over twenty times faster. It becomes apparent from this figure that the calculation of the equilibrium distribution has become a huge bottleneck for the *NumPy*

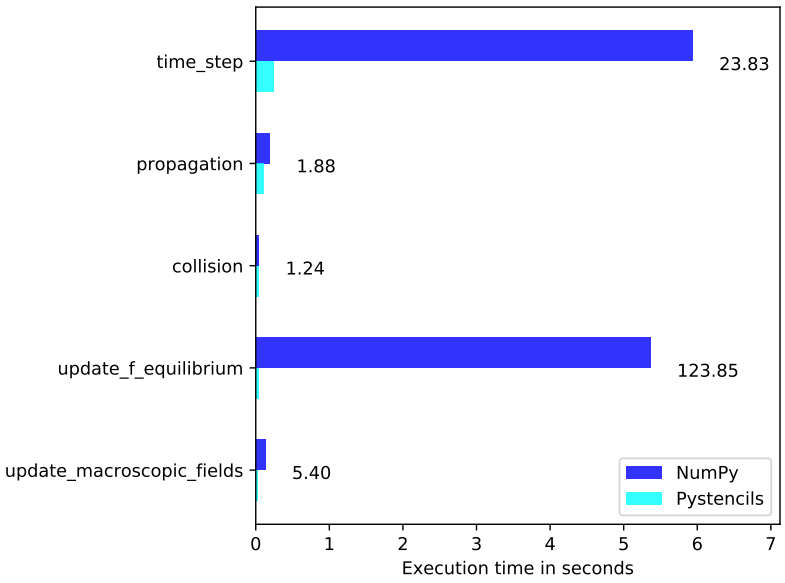


Figure 4: Performance comparison of 100 iterations of the D2V17 lattice implemented using *NumPy* and *pystencils*. The resolution is 128 in each direction and the numbers next to the bars show the performance gain due to using *pystencils*

implementation. This, however, is not true for the *pystencils* implementation whatsoever. This part of the algorithm is over 100 times faster when implemented using *pystencils*. For the complete algorithm this means that its calculation is no longer a huge bottleneck for the computation and the usage of V17 becomes significantly faster and more viable.

There is one more simplification that can be made to further increase the performance gain. Due to a simplification of the algorithm in the above example and the fact that the algorithm is implemented using multiple kernels, the above simulation is not written in pure *pystencils* code. The two main possibilities to implement the propagation are accessing the entries of cells in the corresponding direction and writing the new value in the current cell, or moving the entire array in the direction of the velocity it is supposed to represent. In pure *NumPy* the latter approach is implemented by using the function *roll*.

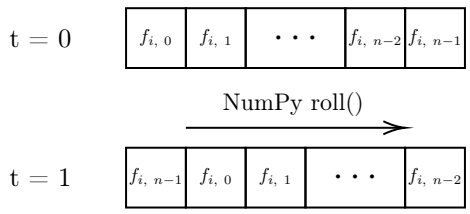


Figure 5: Periodic streaming of velocities to the right for a one dimensional field. $f_{i,x}$ is the population of the system at position x for the velocity i that represents right moving populations

This function moves every part of the complete system that corresponds to one velocity in its direction for as far as this velocity is reaching. Figure 5 shows one example of a one dimensional array that is moved one cell to the right. To apply periodic boundaries, the cells at one end that are now out of bounds are simply added on the other side of the field. Although this method is quite simple to implement, it is not very computationally efficient. *pystencils* on the other hand uses the accessing approach. This is usually no problem for the bulk cells, but becomes more problematic for boundary cells, because on a periodic field, cells on the other side of the array have to be accessed. Bulk cells are cells that are so far away from the boundary of the system that none of the discrete velocities can reach the boundaries. If at least one velocity reaches the boundary, the cell is further referred to as a boundary cell. In chapter 3.3 two possible ways how to implement kernels with neighbour access were introduced.

When *pystencils* iterates over the bulk nodes, the propagation could be achieved by pulling new values from the neighbours into the current cell, or by pushing the new values into the neigh-

bouring cells. In Figure 6 an example for the pull kernel of the D2Q9 lattice is shown with a system size of 3×4. Only one value for each cell is shown, but in a real simulation nine values would be inside each cell, one for each direction. For a bulk node, like 5 or 6, accessing the direct neighbours is no problem. Every other node needs to access at least one node on the other side of the field. Node 2, for example, needs to access some values in the cells 9, 10 and 11. This is also the reason for the necessity of the stream collision kernel introduced in 3.3.1. Pull kernels are generally possible in *pystencils* but they require the macroscopic moments of all the neighbouring nodes to be calculated to perform the collision in every neighbouring node only to pull one of the entries into the current node. This would be a huge waste of time. However, if the collision is calculated after the propagation only the equilibrium distribution in the current node has to be calculated and therefore no time is wasted compared to the original algorithm.

To prevent segmentation faults, *pystencils* only iterates over the bulk cells. As a result all the boundary nodes are not calculated. The idea how to fix this is to create a larger field, by copying cells to the other side of the field and iterating over the new bulk cells. As shown in figure 6, the new bulk cells directly correspond to the original cells and their neighbours are correct if periodic boundary condition are applied. If large neighbourhoods are implemented the amount of layers that have to be copied from one side to the other increases to the maximum value one of the velocities reaches, which is three for both the D2V17 and the D2V37 lattice. The performance comparison in figure 4 implements the second method using *NumPy*. However, there is a method how to implement this using pure *pystencils* code.

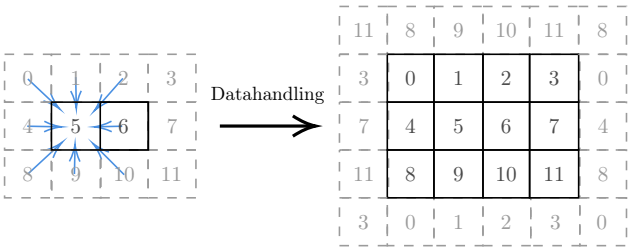


Figure 6: Implementation of datahandling for periodic boundary conditions using the D2Q9 lattice. Bulk nodes are the black blocks and ghost nodes are the grey nodes with dashed lines.

Copying values to increase the amount of bulk nodes is not an idea exclusively used for periodic boundary conditions. One additional example is to enable the calculation on clusters. One of the key advantages of the LBM is the fact that it can be implemented highly parallel, which means that someone might want to implement a calculation using multiple kernels or a graphics processing unit. As a result the field has to be separated into multiple fields that are computed separately. To implement this the local size of the field is increased by adding ghost layers. The name can be explained by the fact that they are not technically part of the system and are not iterated over, because they consist of the new boundary cells. In *pystencils* this whole idea is implemented using the *datahandling* submodule.

This is where *lbmweights* and *pystencils* meet. The assignment collection returned from *lbmweights* can now be used to create a complete simulation for kernels of arbitrary size, by adding data handling to handle boundary conditions. The order of accuracy for the calculation of the equilibrium distribution function can be specified as one of the properties of the *Lattice* objects.

The first part of listing 8 shows how a two dimensional lattice, in this case the D2V37 and the assignment collection corresponding to this lattice can be retrieved from *lbmweights*. This part already completely describes the algorithm, but is missing the field it operates on. In the second part those fields are created. The algorithm needs two fields, one to read and one to write. The shape of those fields is given by two properties of the lattice, one for the amount of values per cell and the other one that calculates how many ghost layers are needed. The latter simply takes the array for all the discrete velocities and calculates the largest value in the array, which is equal to three for the D2V37. Afterwards a copy function is created. This function is used to copy the values of the source field into ghost layers, as shown in figure 6. The *stencil* parameter for the copy function might seem confusing at first. It is used to determine what nodes are supposed to be copied. For example if the D2Q4 lattice ([9]) was used, which can be imagined as the D2Q9 without the outermost, the stationary velocity and different weights then copying fields like 11 or 8 in figure 6 would be unnecessary. Choosing a squared shaped lattice like the D2Q9 therefore works for general cases. Finally the kernel is created, compiled and the source field is initialised with some

Listing 8: D2V17 LBM using *pystencils* and *lbmweights*

```
tau = 0.55
size = (100, 100)
lattice = Lattice.from_name("D2V37")
assignment_collection = lattice.assignment_collection

dh = ps.create_data_handling(size, periodicity=(True, True))
src = dh.add_array('src', values_per_cell=lattice.q,
                   ghost_layers=lattice.necessary_ghost_layers)
dst = dh.add_array_like('dst', 'src')
copy_func = dh.synchronization_function(['src'], stencil='D2Q9')
kernel = ps.create_kernel(assignment_collection).compile()
initialize(dh)  # Some function to fill source field

def iterate():
    copy_func()
    dh.run_kernel(kernel, omega=1/tau)
    dh.swap('src', 'dst')
```

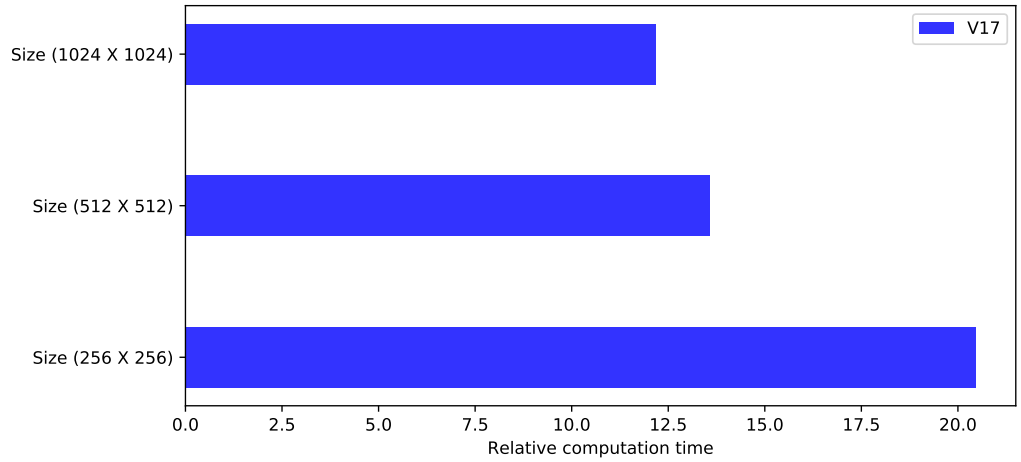


Figure 7: Average execution time of the *NumPy* implementation divided by the average execution time of the *pystencils* implementation for the D2V17 with different spatial resolutions.

value by a function that is not shown here. Every iteration therefore consists of filling the source field and its ghost layer nodes by calling the copy function, running the kernel and swapping the the fields. Swapping the fields means that all the newly calculated values are now accessible in the source field, simply implemented in C++ using a swap of pointers. The *dest* field is therefore only a temporary field used to store the new values. The *NumPy* implementation does the same thing by copying all the results from the temporary field into the actual field.

Figure 7 shows the performance gain due to the implementation using the combination of *lbmweights* and *pystencils*. The exact values are not very important, since they are highly dependant on the system used for simulation, cache sizes and different effects influenced by some optimisations that can be done for both implementations. Overall for the largest sizes, which are also the most relevant for real life application *pystencils* is about twelve times faster for the large neighbourhood lattice. The huge discrepancy is mostly due to the significantly faster calculation of the equilibrium distribution function shown in figure 4. Measuring the performance of the one-kernel implementation compared to the implementation with multiple kernels and data handling implemented in *NumPy* shows that the new implementation is about 40% faster. The most important takeaway is that *pystencils* is faster in every case. Therefore, the idea of using *NumPy* as a tool for the implementation is disregarded from now on. The integration of code generation made the execution time faster and the implementation easier. One additional advantage is the fact that the assignment collection from *lbmweights* can be used for every valid two dimensional lattice, like the D2V37 without any additional work.

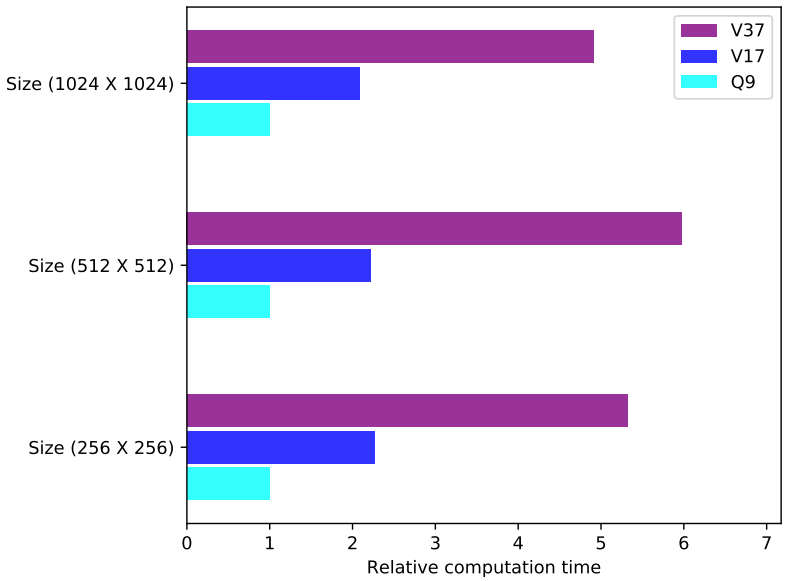


Figure 8: Average computation time of the D2V17 and the D2V37 relative to the computation time of the D2Q9 lattice over 100 iterations. Implemented using *pystencils*

.

3.4.1 Performance Benchmarking of Large Neighbourhoods

Figure 8 shows the impact of using large neighbourhoods on the computation time. The times are normalised to the computation time of the D2Q9 lattice with code generation, which means that each bar shows how much longer the simulation takes for the D2V17 or D2V37 implementation. For small simulations this value is very close to the expected increase due to the higher amount of velocities, since for 37 discrete velocities one could expect the algorithm to be about $37/9 \approx 4$ times slower than the algorithm for 9 discrete velocities. There are a lot of interpretations possible as to why the increase of relative computation time seems to increase first, and afterwards decreases. One of the main reasons is probably cache size. For large lattices, the amount of entries in one field is twice and four times as high. The smallest system size seems to still fit in the cache for all cases. Once the resolution of the simulation is increased the simulation becomes too large for the cache. Since the large benchmark values are more relevant than the small values, the implementation of the D2V17 with a third order accurate equilibrium distribution function can be considered 2.5 times slower and the implementation of the D2V37 with the fourth order accurate function about 6 times slower than the usual method. It is worth mentioning that these numbers can potentially be improved by optimising the calculation of the macroscopic values or the equilibrium distribution. One example could be including common subexpression eliminations (CSEs) using *pystencils*, which is the process of looking for recurring expressions and only calculating them once. Even though this change can be applied to all lattices, the improvement should still be noticeable due to the fact that the largest lattice has more recurring expressions.

3.5 Stability and Accuracy Benchmarking

Benchmarking the accuracy is achieved by comparing the simulated flow to its analytical solution. This means that for an initial and a boundary condition the exact values can be calculated by a formula depending on the time t. Generally speaking this only possible for very basic examples. One exception is the Taylor-Green vortex flow. Compared to most other flows with analytical solution, its initial state does not look trivial. The following equations are the analytical formulation of the Taylor-Green flow if the resolution for both dimensions is equivalent[1]:

[1]See e.g. appendix 3 of [8] for a more general description

$$\boldsymbol{u}(\boldsymbol{x},t) = u_0 \begin{pmatrix} -cos(2\pi \cdot x)sin(2\pi \cdot y) \\ sin(2\pi \cdot x)cos(2\pi \cdot y) \end{pmatrix} e^{-t/t_d}$$

$$p(\boldsymbol{x},t) = p_0 - \rho^* \frac{u_0^2}{4} \left[cos(4\pi \cdot x) + cos(4\pi \cdot y)\right] e^{-2t/t_d}$$

$$t_d = \frac{1}{2\nu \cdot (2\pi/N)^2} \quad \boldsymbol{x} \in [0,1] \times [0,1]$$

N is the resolution and u_0, p_0 and ρ^* are design parameters. There are a few takeaways from this definition. Firstly, the only part of the equation that depends on time is the amplitude. This means that the form of the macroscopic values in the simulation is not supposed to change. Therefore, the basic structure of the flow does not change over time. The values for the velocities, however, slowly become smaller.

To compare a flow with the analytical solution it has to be initiated. This can be done by evaluating the equations for $t = 0$, calculating the equilibrium distribution and initialising the whole field with this equilibrium distribution. Then the simulation starts and a few hundred iterations are computed. Afterwards the analytically correct solution can be evaluated by entering the amount of iterations as the variable t and comparing the analytical solution to the simulated one.

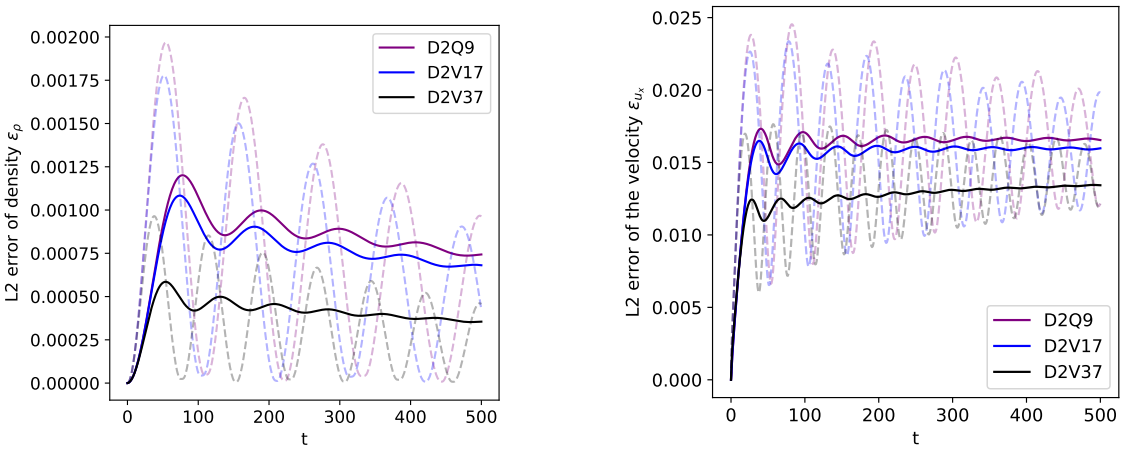
Additionally, it has to be kept in mind that the speed of sound is different for different lattices used. According to the law of similarity and the results presented in chapter 2.1.3 this should not matter as long as the Reynolds numbers are the same and the Mach numbers are similar. The Reynolds number depends on the viscosity introduced in chapter 2.2.4 and therefore on the speed of sound of the lattice. As a result two simulations with the same initialisation parameters, but different speed of sound, do not simulate the same physical system. To make up for this, one parameter has to be changed accordingly. The best way to do this is by changing the relaxation rate of a simulation in a way, that the viscosity is equivalent for all the simulations. Additionally, the pressure p_0 is related to the density by the equation $p_0 = \rho c_s^2$. Both restrictions have to be taken into account in order to accurately quantify the accuracy and stability of this benchmark flow. Obviously there are incredibly many methods and metrics to compare the accuracy of two simulations. For the rest of this work the L2 error for one of the velocities or for the density is chosen. Due to symmetrical initial conditions the choice of which velocity to choose does not matter. Generally the L2 error for a field s is defined in the following way, if the analytical solution s_a and the numerical values s_n at the time t are known [8]:

$$\epsilon_s = \sqrt{\frac{\sum_{\boldsymbol{x}} \left(s_n(\boldsymbol{x},t) - s_a(\boldsymbol{x},t)\right)^2}{\sum_{\boldsymbol{x}} s_a(\boldsymbol{x},t)}}$$

3.5.1 Accuracy of Large Neighbourhoods

One of the reasons why large neighbourhoods are not very common yet, is the fact that they are technically not needed. It was shown in chapter 2.1.4 that the necessary accuracy for the macroscopic moments is reached with the most basic lattices. However, there exists strong evidence that large neighbourhoods should be implemented. One of them is that according to Philippi et al. the fourth order accurate equilibrium distribution calculation is necessary to retrieve thermohydrodynamic moments if non isothermal flows are supposed to be simulated [6]. For the isothermal case the higher order truncation should lead to a more accurate simulation, as well. To make up for the different speed of sounds the simulation is not initiated by the relaxation time τ, but instead by with a value for the viscosity ν. Both are directly defined if one of them is given and they are related by the formula introduced in chapter 2.2.4. Figure 9 shows the accuracy benchmark for the velocity and the density. Every simulation is initialised with viscosity $\nu = 1/6$, the maximum velocity u_0 is set to 0.04, $\rho*$ to 0.1 and the resolution of the simulation to 128 cells in each direction which means that the Reynolds number is 30, p_0 is set to c_s^2 so the resulting density is one in each case, but this choice is arbitrary according to appendix 3 in [8].

The different speed of the fluctuations of the error in the figure is probably related to the different Mach numbers. Their values are fixed after setting all the other parameters. They are



(a) (b)

Figure 9: L2 error of the density (a) and the velocity (b). The dotted lines are the calculated errors in each iteration and the solid lines are the average errors.

not too far apart, so the simulations should still be comparable[1]. The average error during the first few hundred simulations is better for the large lattices, as well as the maximum error. If the resolution of the simulation is increased and the the initialisation parameters changed accordingly, all errors decrease while simultaneously keeping the structure of the plots shown in 9. This means the accuracy gain for large neighbourhoods is about the same.

3.5.2 Stability of Large Neighbourhoods

The two ways to increase the Reynolds number and therefore to test how stable a lattice can be become clear when looking at its formula introduced in chapter 2.1.3. If the size of the system is fixed, the number can be increased by increasing the maximum velocity or by making the relaxation time approach 0.5, which at the same time means that the viscosity is getting smaller.

Figure 10 shows one possible analysis of the systems stability. For every lattice the simulation runs for 1000 iterations. The error of the simulation is calculated after it finishes. The maximum L2 error is capped to one and if this value is reached the simulation is considered unstable. Additionally, if the simulation fails for one maximum velocity every simulation with the same viscosity but higher velocity is considered unstable as well.

The step from the D2Q9 lattice to the D2V17 lattice shows a significant improvement. The simulation remains stable for some viscosity values with up to 40% higher velocity. Even though the accuracy is not very impressive anymore, running a simulation with those parameters, for example $u_0 = 0.4$ and $\nu = \frac{1}{200}$ using the *lbm-large-neighbourhood* package shows that the simulation remains stable all the time and the structure of the flow is still recognisable. It is also worth mentioning that the D2V37 flow did way better than this figure suggests. Almost every set of parameters that is unstable in all cases remains stable for the longest time when using the largest velocity set.

Other papers have been published that do a similar stability analysis for the three lattices. One example is a von Neumann stability analysis of the LBM for the D2V17 and D2V37 lattices done by Siebert et al. [15]. The difference is that this analysis doesn't initialise the complete system with a benchmark flow, but looks at the answer of the system to a small perturbation from the equilibrium that is supposed to be absorbed. This analysis suggests that the stability of the D2V37 should be even better than the one of the D2V17 lattice. The reasoning behind the fact that the lattice seems to do worse for this particular benchmark flow remains a question that has to be solved in the future.

[1] Values for the Mach number are roughly 0.07, 0.065, and 0.045 in the order of increasing velocity discretisation

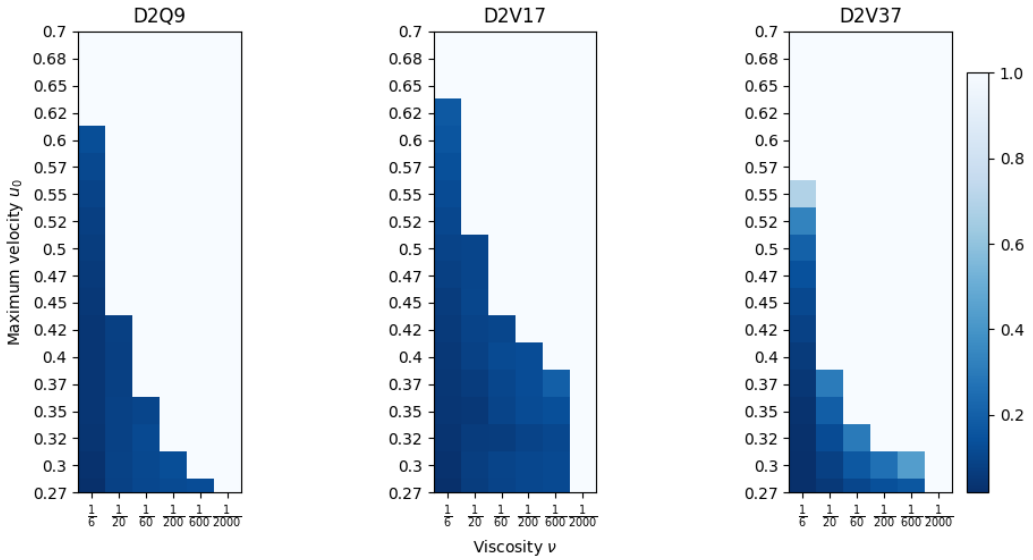


Figure 10: Stability comparison of the Taylor-Green flow for the three different lattices.

4 Conclusion and Outlook

It comes as no surprise that adding large neighbourhoods comes with both advantages and disadvantages. One problem is that the implementation of the LBM is not completely independent of its velocity discretisation. The differences between each implementation were analysed and as it turns out the only part that introduces a major difference is the calculation of the equilibrium distribution. Therefore, the LBM was implemented in a way to work with different velocity sets by using code generation. The velocity sets come from an algorithm proposed by Spiller et al. [13]. This algorithm and its corresponding script were briefly discussed, revised and used for a lattice independent implementation of the isothermal LBM. The influence of introducing code generation to the algorithm was analysed regarding its influence on the performance and it was shown that it leads to a significantly lower computation time.

As expected, the main disadvantage that comes with the implementation of large neighbourhoods is the increase in computation time and memory usage. Special focus was set on how much this computation time increases for the different implementations introduced throughout this thesis. This part showed the main advantage of using code generation, as its introduction not only increased the computation time of the program compared to the *NumPy* implementation, but it additionally improved the relative computation time of large neighbourhoods compared to normal neighbourhoods. The analysis showed that using the D2V17 is about 2.5 and the using the D2V37 model about 6 times slower than the original lattice. Applying CSE to this step could further improve these numbers.

To analyse the main advantages that are expected from the implementation of large neighbourhoods the Taylor-Green benchmark flow was used. The two main points considered were the accuracy and stability of large neighbourhoods. Looking at the accuracy of the density and the velocity for all lattices showed, that the D2V17 comes with a small advantage in accuracy while the D2V37 almost cuts the L2 error of the density in half and significantly reduced the error of the velocities. During the stability analysis, it was demonstrated how the D2V17 can be used to run simulations with a 40% higher maximum velocity as initial condition which corresponds to a significantly higher Reynolds number. The stability analysis of the largest lattice on the other hand didn't show any improvement compared to the normal velocity set, contrary to the expectations.

Additionally, a method was described on how to find moments for the MRT implementation of the LBM. The weights derived from these moments turned out to be equivalent to the real values, even though the resulting equilibrium moments were different. It would be really interesting to revise this idea and potentially formulate the algorithm to automatically create MRT models with the same discretisation of the equilibrium distribution or a similar one that shows the same properties. This would lead to an even more general form of the LBM with large neighbourhoods if only the discrete velocities are given. At the same time this method would derive the corresponding discrete equilibrium distribution moments and would therefore directly define the calculation of the equilibrium distribution function.

References

[1] Arthur M Jaffe. The millennium grand challenge in mathematics. *Notices of the AMS*, 53(6), 2006.

[2] Desciption of the navier stokes problem. http://www.claymath.org/millennium-problems/navier%E2%80%93stokes-equation. Accessed: September 30, 2019.

[3] SS Chikatamarla, CE Frouzakis, IV Karlin, AG Tomboulides, and KB Boulouchos. Lattice boltzmann method for direct numerical simulation of turbulent flows. *Journal of fluid mechanics*, 656:298–308, 2010.

[4] Sauro Succi. Lattice boltzmann 2038. *EPL (Europhysics Letters)*, 109(5):50001, 2015.

[5] Souleymane Balla-Arabé, Xinbo Gao, and Bin Wang. A fast and robust level set method for image segmentation using fuzzy clustering and lattice boltzmann method. *IEEE transactions on cybernetics*, 43(3):910–920, 2013.

[6] Paulo C Philippi, Luiz A Hegele Jr, Luís OE Dos Santos, and Rodrigo Surmas. From the continuous to the lattice boltzmann equation: The discretization problem and thermal models. *Physical Review E*, 73(5):056702, 2006.

[7] Github repository lattice-boltzmann weights. https://github.com/BDuenweg/Lattice-Boltzmann-weights. Accessed: September 30, 2019.

[8] Timm Krüger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Viggen. The lattice boltzmann method. *Springer International Publishing*, 10:978–3, 2017.

[9] Dieter A Wolf-Gladrow. *Lattice-gas cellular automata and lattice Boltzmann models: an introduction*. Springer, 2000.

[10] Miloslav Feistauer, M Feistauer, Jiří Felcman, Jiri Felcman, and Ivan Straškraba. *Mathematical and computational methods for compressible flow*. Oxford University Press on Demand, 2003.

[11] Martin Anton van der Hoef, R Beetstra, and JAM Kuipers. Lattice-boltzmann simulations of low-reynolds-number flow past mono-and bidisperse arrays of spheres: results for the permeability and drag force. *Journal of fluid mechanics*, 528:233–254, 2005.

[12] Einstein's summation convention. https://en.wikipedia.org/wiki/Einstein_notation. Accessed: September 30, 2019.

[13] Burkhard Dünweg Dominic Spiller. Semi-automatic construction of lattice boltzmann models. Private Correspondence. Version: February 7, 2019.

[14] Martin Bauer. From statistical mechanics to lattice boltzmann. Private Correspondence. 15.12.2017.

[15] DN Siebert, LA Hegele Jr, and PC Philippi. Lattice boltzmann equation linear stability analysis: Thermal and athermal models. *Physical Review E*, 77(2):026707, 2008.

[16] Gitlab repository for lbm-bigger-stencils. https://i10git.cs.fau.de/ed52egek/lbm-bigger-stencils. Accessed: September 30, 2019.

[17] Numpy website and documentation. https://www.numpy.org. Accessed: September 30, 2019.

[18] Gitlab repository for lbmweights. https://i10git.cs.fau.de/ed52egek/lbmweights. Accessed: September 30, 2019.

[19] Gerald Fahner. A multispeed model for lattice-gas hydrodynamics. *Complex Systems*, 5(1):1, 1991.

[20] Y-H Qian and Ye Zhou. Complete galilean-invariant lattice bgk models for the navier-stokes equation. *EPL (Europhysics Letters)*, 42(4):359, 1998.

[21] Shyam S Chikatamarla and Iliya V Karlin. Lattices for the lattice boltzmann method. *Physical Review E*, 79(4):046701, 2009.

[22] Code generation of stencil codes using pystencils. http://pycodegen.pages.walberla.net/pystencils/. Accessed: September 30, 2019.

[23] lbmpy - run fast fluid simulations based on the lattice-boltzmann method in python. https://i10git.cs.fau.de/pycodegen/lbmpy. Accessed: September 30, 2019.