BADChIMP manual

Espen Jettestuen Olav Aursjø

May 5, 2022

Contents

1 Executive summary

This code is developed for modeling fluid flow on the pore scale. The code can handle general geometries, non-Newtonian fluid rheologies and multi-phase systems. The code has been developed with the goal of providing a flexible environment for trying out new methods and add new physics.

The code is in active development so we will suggested to download it from our github repository[1] to get the most updated version.

There are three major parts in this manual. Chapter 2, **Introduction and background**, presents the theory behind the lattice Boltzmann method and should be a good short overview for students and researchers that want to understand the basics of the method. If you already are acquainted with he lattice Boltzmann method you could easily skip this section, although it can be useful to browse through it to get to know our notation. Chapter 3, **Code description and manual**, describes the essential parts of the code and goes through a case study line by line. Chapter 4, **Simulation of non-Newtonian rheologies**, describes how to simulate non-Newtonian rheologies using generalized expression for strain-rate-dependent viscosity.

[1]You can access it from the following link https://github.com/eje74/BADChIMP-cpp

2 Introduction and background

2.1 The Lattice Boltzmann method

The lattice Boltzmann method [9, 13, 15, 14] is historically a generalization of the lattice gas methods for fluid flow. Later, it was shown that it was a discretization of the Boltzmann equation. It is the latter derivation that have gained traction in the lattice Boltzmann community and we will just sketch out how LB is related to the Boltzmann equation.

2.1.1 Boltzmann equation

The Boltzmann equation [5] is given as

$$\partial_t f + c_i \partial_i f = \Omega, \tag{2.1}$$

where $f(t, \vec{x}, \vec{c})$ is the single-particle distribution function, $\vec{c}$ is the microscopic particle velocity, and with a collision term Ω which is a function of the distribution functions. We will also use the notation that ∂_t is the partial differential operator with respect to time, t and ∂_i is the partial differential operator with respect to the i'th Caretesian spatial coordinate, x_i. We also note that we will use Einsteins summation convention for repeated Cartesian indices.

The equilibrium distribution for f is given by the Maxwell-Boltzmann distribution

$$f^{\mathrm{MB}}(\vec{c}, \rho, \vec{u}) = \frac{\rho}{(2\pi c_s^2)^{3/2}} \exp\left(-\frac{|\vec{c} - \vec{u}|^2}{2c_s^2}\right), \tag{2.2}$$

where c_s is the sound velocity, and ρ and $\vec{u}$ are the macroscopic density and velocity, respectively. In lattice Boltzmann simulations, this is usually the BGK-collision term [4]

$$\Omega = -\frac{1}{\tau}\left(f - f^{\mathrm{MB}}\right). \tag{2.3}$$

The macroscopic quantities are defined through different moments of the distribution function,

$$\rho = \textstyle\int f \,\mathrm{d}\vec{c}, \quad \rho\vec{u} = \int \vec{c} f \,\mathrm{d}\vec{c}, \quad \rho\theta = \frac{1}{3}\int |\vec{c} - \vec{u}|^2 f \,\mathrm{d}\vec{c}. \tag{2.4}$$

A standard notation for the moment of the distribution of microscopic velocities is

$$\Pi_{i_1 \ldots i_n} = \int c_{i_1} \cdots c_{i_n} f \,\mathrm{d}\vec{c}. \tag{2.5}$$

2.1.2 Macroscopic equations

The macroscopic equations are derived from the Boltzmann equation by integration over different moments of the microscopic particle velocity coordinates. We will just illustrate the derivations using an example without any body force terms. Here we will use the integral notation $\int \mathrm{d}^3\vec{c}$ as short-hand for $\int \mathrm{d}c_x \mathrm{d}c_y \mathrm{d}c_z$. The mass conservation law follows directly from the definition in the previous section:

$$\begin{aligned}
&\int \left(\partial_t f + c_i \partial_i f\right) \mathrm{d}^3\vec{c} = \int \Omega \, \mathrm{d}^3\vec{c} = 0 \\
&\partial_t \int f \, \mathrm{d}^3\vec{c} + \partial_i \int c_i f \, \mathrm{d}^3\vec{c} = 0 \\
&\partial_t \rho + \partial_i \rho u_i = 0.
\end{aligned}$$

The equations for impulse conservation are given by

$$\begin{aligned}
&\int \left(c_i \partial_t f + c_i c_j \partial_j\right) \mathrm{d}^3\vec{c} = \int c_i \Omega \, \mathrm{d}^3\vec{c} = 0 \\
&\partial_t \int c_i f \, \mathrm{d}^3\vec{c} + \partial_j \int c_i c_j f \, \mathrm{d}^3\vec{c} = 0 \\
&\partial_t \rho u_i + \partial_j \Pi_{ij} = 0.
\end{aligned}$$

Here, we obtain a second order tensor Π_{ij} that we need to determine, which can be accomplished using the Chapman-Enskog expansion described in section 2.1.4.

2.1.3 The lattice Boltzmann numerical scheme

The procedure used to to derive the lattice Boltzmann equation from the Boltzmann equation is quite elaborate, so we will only give an executive sketch of the method with references to work that will supply the details.

The derivation can is conducted in two steps. First, the continuum of microscopic velocities is discretizied, so that we are left with an equation for a finite set of velocity distributions, f_α, needed to described the systems. Here, α is just an integer denoting which microscopic velocity distribution it represents. After this first step we are left with a coupled system of differential equations. One for each microscopic velocity, that needs to be integrated in time. The second step is to discretize this integration.

For the first step we need to rewrite and Taylor expand f^{MB} in terms of (u/c_s), i.e., the low Mach number limit,

$$\begin{aligned}
f^{\mathrm{MB}}(|\vec{u} - \vec{c}|) &= \frac{\rho}{(2\pi c_s^2)^{3/2}} \exp\left(-\frac{(\vec{u} - \vec{c})^2}{2c_s^2}\right) = \frac{\rho \exp\left(-\frac{c^2}{2c_s^2}\right)}{(2\pi c_s^2)^{3/2}} \exp\left(-\frac{u^2 - 2\vec{c}\cdot\vec{u}}{2c_s^2}\right) \\
&= \rho w(c) \left(1 + \frac{\vec{c}\cdot\vec{u}}{c_s^2} + \frac{(\vec{c}\cdot\vec{u})^2 - c_s^2 u^2}{2c_s^4}\right) + \mathcal{O}\left((u/c_s)^3\right),
\end{aligned}$$

where

$$w(c) = (2\pi c_s^2)^{-3/2} \exp\left(-\frac{c^2}{2c_s^2}\right).$$

It is this truncated expansion that is used for the equilibrium distribution in the lattice Boltzmann simulation. To derive the equilibrium values, one may now use this truncated expression to obtain the macroscopic variables. Abe [1], and He and Luo [6, 7], described that, by using Gaussian quadrature, they could rewrite the integrals over $\vec{c}$, using sums over a finite sets of microscopic velocities that gave the same lattice structure as the standard lattice Boltzmann models[1]. We can see this for ourselves by writing out the expression for

[1]They had a different approach for handling the time integration. This turned out to be a sub-optimal solution.

the equilibrium moments

$$\Pi_{i_1\ldots i_n} = \int c_{i_1}\cdots c_{i_n} f \, \mathrm{d}\vec{c} \approx \int \rho w(c) c_{i_1}\cdots c_{i_n}\left(1 + \frac{\vec{c}\cdot\vec{u}}{c_s^2} + \frac{(\vec{c}\cdot\vec{u})^2 - c_s^2 u^2}{2c_s^4}\right) \mathrm{d}^3\vec{c}. \quad (2.6)$$

Here, we can write the integral using a polynomial $P_{i_1\ldots i_n}(\vec{c})$ in $\vec{c}$, and use Gaussian quadrature to obtain

$$\int \rho w(c) P_{i_1\ldots i_n}(\vec{c}) \, \mathrm{d}\vec{c} = \sum_\alpha \rho w_\alpha P_{i_1\ldots i_n}(\vec{c}_\alpha), \quad (2.7)$$

where α is an index used to identify the microscopic velocities used in the Gaussian quadrature and $w_\alpha = w(c_\alpha)$. We only need a finite number of f-distributions to calculate the integrals. Hence, we do only need to know the evolution of the set of distribution, f_α, used in the Gaussian quadrature scheme. And, from this, we are left with the set of differential equations,

$$\partial_t f_\alpha + c_{\alpha i}\partial_i f_\alpha = \Omega_\alpha, \quad (2.8)$$

where $\Omega_\alpha = -1/\tau(f_\alpha - f_\alpha^{\mathrm{MB}})$. We also note that the Gaussion quadrature also supply us with the discrete equilibrium distribution

$$f_\alpha^{\mathrm{eq}} = w_\alpha\left(1 + \frac{c_{\alpha i}u_i}{c_s^2} + \frac{Q_{\alpha ij}u_i u_j}{2c_s^4}\right), \quad (2.9)$$

where

$$Q_{\alpha ij} = c_{\alpha i}c_{\alpha j} - c_s^2\delta_{ij}. \quad (2.10)$$

This is the standard expression used for the equilibrium distribution in LB models.This concludes the first step of the derivation of the LB method.

To begin our second step in the derivation, we note that the left hand side of the discrete Boltzmann equation, Eq. (2.8), is a 'free streaming' with velocity c_α. Thus, by integration along this velocity direction over the time interval Δt we get

$$f_\alpha(t+\Delta t, \vec{x}+\Delta t\vec{c}_\alpha) - f_\alpha(t,\vec{x}) = \\ -\frac{\Delta t}{\tau}\int_0^{\Delta t}\left(f_\alpha(t+t', \vec{x}+t'\vec{c}_\alpha) - f_\alpha^{\mathrm{MB}}(t+t', \vec{x}+t'\vec{c}_\alpha)\right)\mathrm{d}t'.$$

Using the trapezoidal rule on the right hand side, we obtain

$$f_\alpha(t+\Delta t, \vec{x}+\Delta t\vec{c}_\alpha) - f_\alpha(t,\vec{x}) \approx \\ -\frac{\Delta t}{\tau}\left(\frac{f_\alpha(t+\Delta t, \vec{x}+\Delta t\vec{c}_\alpha) - f_\alpha^{\mathrm{MB}}(t+\Delta t, \vec{x}+\Delta t\vec{c}_\alpha)}{2} + \frac{f_\alpha(t,\vec{x}) - f_\alpha^{\mathrm{MB}}(t,\vec{x})}{2}\right),$$

which is an approximation to the integral up to order $(\Delta t)^3$. In this expression, we have $f_\alpha(t+\Delta t, \vec{x}+\Delta t\vec{c}_\alpha)$ expressed using $f^{\mathrm{MB}}(t+\Delta t, \vec{x}+\Delta t\vec{c}_\alpha)$ which is an unknown quantity. To get around this problem, we make a change of variable from f_α to

$$f_\alpha^*(t,\vec{x}) = f_\alpha(t,\vec{x}) + \frac{\Delta t}{2\tau}\left(f_\alpha(t,\vec{x}) - f_\alpha^{\mathrm{MB}}(t,\vec{x})\right), \quad (2.11)$$

so that we end up with

$$f_\alpha^*(t+\Delta t, \vec{x}+\Delta t\vec{c}_\alpha) - f_\alpha^*(t,\vec{x}) = -\frac{\Delta t}{\tau}\left(f_\alpha^*(t,\vec{x}) - f_\alpha^{\mathrm{MB}}(t,\vec{x})\right). \quad (2.12)$$

The reason why this works is that $\int \Omega \, \mathrm{d}^3 c = \int c_i \Omega \, \mathrm{d}^3 c = 0$ for isolated systems, i.e., system without source terms. Hence, the redefinition of f does not change the derived density and velocity of the isolated system, and we can find these variables using f_α^* instead of f_α. We note that if we introduce mass sources and body forces we will need to add corrections to the collision terms. In the rest of this report, we will, through a renaming, denote f_α for f_α^*. This concludes the second, and last, part of the derivation. Eq. (2.12) is the equation known as *the lattice Boltzmann equation.*

2.1.4 Chapman-Enskog expansion

The Chapman-Enskog expansion [5] is a method used to identify which macroscopic equations are modeled by the Boltzmann equation. This also includes how the parameters, describing the microscopic interactions, are related to the ones emerging on macroscopic scales. The Chapman-Enskog method is referred to as an asymptotic method so it is assumed to only show behavior in the long wavelength limit. We also note that the Chapman-Enskog expansion method does not tell us if the behavior, given by the Boltzmann equation, actually converges to the identified macroscopic equations.

The expansion is based on the assumption that both differential operators, ∂_t and ∂_i, and distributions, f, can be expanded in terms that formally can be written as

$$\partial_t = \epsilon\partial_t^{(1)} + \epsilon^2\partial_t^{(2)} + \cdots , \tag{2.13}$$

$$\partial_i = \epsilon\partial_i^{(1)} + \epsilon^2\partial_i^{(2)} + \cdots , \tag{2.14}$$

and

$$f = f^{(0)} + \epsilon f^{(1)} + \epsilon^2 f^{(2)} + \cdots . \tag{2.15}$$

The ϵ indicates the influence of the term in the long wavelength limit. Here, terms with higher powers of ϵ have less influence than those with lower powers of ϵ. The goal of the Chapman-Enskog expansion is to obtain the expression for the terms in the expansion of f, as function of the macroscopic variables. So that different moments of the microscopic velocities, for instance Π_{ij}, can be expressed as functions of the macroscopic variables and their partial derivatives. For the lattice Boltzmann method, it is common to apply this method to the discretize version of the the Boltzmann equation [8, 17].

We will use the LB equation, Eq. (2.12), as an example. The expansion on the left hand side is found by first applying a Taylor expansion to the left hand side of the LB equation

$$f_\alpha(t+\Delta t, \vec{x}+\Delta t\vec{c}_\alpha) - f_\alpha(t,\vec{x}) = \sum_{n=1} \frac{(\Delta t)^n}{n!}\left((\partial_t + c_{\alpha i}\partial_i)^n f_\alpha\right)(t,\vec{x}). \tag{2.16}$$

The tedious part of the derivation is that we need to insert expressions Eqs. (2.13), (2.14), and (2.15) for ∂_t, ∂_i, and f, respectively, and then gather the terms with of the same power of ϵ. Here, we give these expressions for the three lowest levels of ϵ's

$$\begin{aligned}
&\mathcal{O}(\epsilon^0) : 0\\
&\mathcal{O}(\epsilon^1) : \left(\partial_t^{(1)} + c_{\alpha i}\partial_i^{(1)}\right) f_\alpha^{(0)}\\
&\mathcal{O}(\epsilon^2) : \left(\partial_t^{(1)} + c_{\alpha i}\partial_i^{(1)}\right) f_\alpha^{(1)} + \left(\partial_t^{(2)} + c_{\alpha i}\partial_i^{(2)}\right) f_\alpha^{(0)} + \frac{1}{2}\left(\partial_t^{(1)} + c_{\alpha i}\partial_i^{(1)}\right)^2 f_\alpha^{(0)},
\end{aligned}$$

where we have put $\Delta t = 1$ to make them more readable. The right-hand side of the equation is simpler. First of all, we identify that $f_\alpha^{\mathrm{eq}} = f_\alpha^{\mathrm{MB}}$ (see Eq. (2.9)) and we get that

$$\begin{aligned}
\mathcal{O}(\epsilon^0) :& f_\alpha^{(0)} - f_\alpha^{\text{eq}} \\
\mathcal{O}(\epsilon^1) :& -\frac{1}{\tau} f_\alpha^{(1)} \\
\mathcal{O}(\epsilon^2) :& -\frac{1}{\tau} f_\alpha^{(2)}.
\end{aligned}$$

From the $\mathcal{O}(\epsilon^0)$, we note that we can put $f_\alpha^{(0)} = f_\alpha^{\text{eq}}$. The important thing to notice here is that f_α^{eq} is a function of macroscopic variables. By matching the ϵ terms, we note that we can express $f_\alpha^{(1)}$, on the right-hand side, using only $f_\alpha^{(0)}$, on the left-hand side. This pattern is general, so that the $\mathcal{O}(\epsilon^n)$, we have that the right-hand side, consists of f's up to order n, while the left-hand side only include f's, up to order $(n-1)$. This shows that we can, in principle, derive expressions for f, up to any order in ϵ, using only macroscopic values. Going through these steps, using the expressions up to and including $\mathcal{O}(\epsilon^2)$, we can derive the Navier-Stokes equation as presented in section 2.2.1.

One important expression, from the Chapman-Enskog expansion, used in deriving models for non-Newtonian flow is the relation between $f_\alpha^{\text{neq}} = f_\alpha - f_\alpha^{\text{eq}}$ and the strain rate tensor, E_{ij},

$$\tau^{-1} f_\alpha^{\text{neq}} = -w_\alpha \left(\frac{Q_{\alpha ij}}{c_s^2} + \frac{P_{\alpha ijk} u_k}{c_s^4} \right) \rho E_{ij} - w_\alpha \left(\frac{c_{\alpha i}}{c_s^2} + \frac{Q_{\alpha ij} u_j}{c_s^4} + \frac{P_{\alpha ijk} u_j u_k}{2 c_s^6} \right) F_i, \tag{2.17}$$

where $P_{\alpha ijk} = c_{\alpha i} c_{\alpha j} c_{\alpha k} - c_s^2 \left(c_{\alpha i} \delta_{jk} + c_{\alpha j} \delta_{ik} + c_{\alpha k} \delta_{ij} \right)$.

2.1.5 Lattice structures

There are many different lattice structures that can be used in lattice Boltzmann simulations [16]. We will not list them here, but we will point out the requirements that they need to fulfill.

First of all, the notation used to describe a lattice is on the form DnQm where n is a number that tells you the number of spatial dimensions of the lattice and m tells you the number of basis vectors, $\vec{c}_\alpha$.

A basis vector needs the have the property that if you are at a node position $\vec{x}$ then $\vec{x} + \vec{c}_\alpha$ also needs to be a node position. Finally, the sets of basis vectors, $\vec{c}_\alpha$, and weights, w_α, need to fulfill the following summation rules, also called symmetries, to be able to simulation Navier-Stokes,

$$\begin{array}{ll}
\sum_\alpha w_\alpha = 1, & \sum_\alpha w_\alpha c_{\alpha i} = 0 \\
\sum_\alpha w_\alpha c_{\alpha i} c_{\alpha j} = C_2 \delta_{ij}, & \sum_\alpha w_\alpha c_{\alpha i} c_{\alpha j} c_{\alpha k} = 0 \\
\sum_\alpha w_\alpha c_{\alpha i} c_{\alpha j} c_{\alpha k} c_{\alpha l} = C_4 \left(\delta_{ij} \delta_{kl} + \delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk} \right), & \sum_\alpha w_\alpha c_{\alpha i} c_{\alpha j} c_{\alpha k} c_{\alpha l} c_{\alpha m} = 0.
\end{array} \tag{2.18}$$

2.1.6 A complete lattice Boltzmann scheme

In this section, we will supply a complete lattice Boltzmann scheme, which later is implemented in section 3.3. The equations being solved are given in section 2.2.1. The scheme is given by Eq. (2.12),

$$f_\alpha(t+1, \vec{x} + \vec{c}_\alpha) - f_\alpha(t, \vec{x}) = \Omega_\alpha, \tag{2.19}$$

where we have set $\Delta t = 1$. Now, since the system described in section 2.2.1 has both a mass source term, q, and a body force term, $\vec{F}$, we will need corrections to our collision term [2]

$$\Omega_\alpha = -\frac{1}{\tau} \left(f_\alpha - f_\alpha^{\text{eq}} \right) + \Delta\Omega_\alpha, \tag{2.20}$$

where

$$\Delta\Omega_\alpha = w_\alpha \left(1 - \frac{1}{2\tau}\right)\left(q + \frac{Q_{\alpha ij}u_i u_j}{2c_s^4}q + \frac{c_{\alpha i}F_i}{c_s^2} + \frac{Q_{\alpha ij}u_i F_j}{c_s^4}\right). \tag{2.21}$$

We will also need to add subsequent corrections to the evaluation of the density, ρ, and velocity, $\vec{u}$,

$$\rho = \sum_\alpha f_\alpha + \frac{1}{2}q$$

$$\rho u_i = \sum_\alpha c_{\alpha i} f_\alpha + \frac{1}{2}F_i.$$

Further, it can be shown, using the Chapman-Enskog expansion, that the pressure $p = c_s^2\rho$, and the kinematic viscosity $\nu = c_s^2(\tau - 1/2)$.

The standard method to implement the lattice Boltzmann equation is a two-step procedure, with a collision step and a propagation step. In the collision step, we calculate the value of the LB distribution, $\tilde{f}_\alpha$, which is copied to the neighboring node in the α-direction,

$$\tilde{f}_\alpha(t,\vec{x}) = f_\alpha(t,\vec{x}) + \Omega_\alpha(t,\vec{x}). \tag{2.22}$$

In the propagation step[2], values are copied from one neighbor node to another, following the velocity basis, so that

$$f_\alpha(t+1,\vec{x}+\vec{c}_\alpha) = \tilde{f}_\alpha(t,\vec{x}). \tag{2.23}$$

If the system has solid walls, we will get situations where the right-hand side of the above equation is not known, and we will need to supply a value using a boundary conditions. A standard method in LB models is to use a so-called bounce back scheme. The bounce back scheme that we use, in our standard case, is known as the half-way bounce back scheme. If we assume that we do not know the value of $\tilde{f}_\alpha(t,\vec{x}-\vec{c}_\alpha)$, as the node, at position $(\vec{x}-\vec{c}_\alpha)$, is inside the solid, the scheme says that we can use the value at $\vec{x}$ in the opposite direction of $\vec{c}_\alpha$, which we will call $\vec{c}_{\overline{\alpha}} = -\vec{c}_\alpha$. The propagation step then becomes

$$f_\alpha(t+1,\vec{x}) = \tilde{f}_{\overline{\alpha}}(t,\vec{x}).$$

This scheme will put a no-slip boundary condition on a wall located between positions $\vec{x}$ and $\vec{x}-\vec{c}_\alpha$.

2.2 Continuum equations

2.2.1 Fluid flow

The state of a moving fluid may be described through its velocity $\vec{u}(t,\vec{x})$, its pressure $p(t,\vec{x})$, and its mass density $\rho(t,\vec{x})$ at any position $\vec{x}$ and at any time t. These quantities are governed by an equation of state, together with mass and momentum conservation. Using the Einstein summation convention, where Latin indices denote Cartesian spatial components, mass and momentum conservation may, in component form, be described through [3, 10]

$$\partial_t\rho + \partial_i(\rho u_i) = q\,, \tag{2.24}$$

$$\rho\left(\partial_t u_i + u_j\partial_j u_i\right) = -\partial_i p + F_i + q u_i + \partial_j \sigma'_{ij} - \partial_j T_{ij}\,, \tag{2.25}$$

[2]This can also be referred to as a streaming step

where $q(t,\vec{x})$ is a mass source or sink, $p(t,\vec{x})$ is the pressure of the fluid, $F_i(t,\vec{x})$ is the component of any applied volume force, $\sigma'_{ij}(t,\vec{x})$ are the components of the deviatoric stress tensor, and T_{ij} are the components of any other stresses imposed on the fluid. For the fluid motion to be fully described, an additional equation of state is required to describe the relation between the pressure and the mass density of the fluid.

2.2.2 The deviatoric stress tensor σ'_{ij}

For Newtonian fluids, the relationship between the viscous stress and the strain rate is per definition perfectly linear, and the viscosity is independent of the state of motion or stress in the fluid. Here, the deviatoric stress tensor

$$\sigma'_{ij} = \rho\nu\left[2E_{ij} - \frac{2}{d}\partial_k u_k \delta_{ij}\right] + \xi\partial_k u_k \delta_{ij} \tag{2.26}$$

expresses the viscous stress tensor. Here, $E_{ij} = (1/2)\left[\partial_i u_j + \partial_j u_i\right]$ is the strain rate tensor, ν and ξ are, respectively, the kinematic shear viscosity and the bulk (or volume) viscosity of the fluid, and d is the number of spatial dimensions in the system.

2.2.3 Reactive Advection Diffusion

We consider here, in addition, diffusion of a quantity $\varphi(t,\vec{x})$ influenced by the external fluid flow. This quantity is the concentration, defined relative to the fluid density. It may be shown [10] that, on component form, this process is described by the equation

$$\rho(\partial_t\varphi + u_i\partial_i\varphi) = \partial_i(D\rho\partial_i\varphi) + R - \varphi^k q, \tag{2.27}$$

where D is the diffusivity and $R(t,\vec{x})$ is any given source term. Such a source term could for instance arise from the change in concentration of a chemical component due to bulk chemical reactions.

3 Code description and manual

3.1 Installation and compilation

3.1.1 Download

The code is available from the github repository:

`https://github.com/eje74/BADChIMP-cpp`

This repository can be cloned to your local machine using the following command line argument:

```
$ git clone git@github.com:eje74/BADChIMP-cpp.git
```

This assumes that the user has setup ssh to work with github. The BADChIMP code is then cloned into the folder `BADChIMP-cpp`.

3.1.2 Compilation on linux

To compile the code, run the following command-line argument

```
/BADChIMP-cpp$ ./make.sh <name_of_folder_with_main_file>
```

in the code root folder, which is `BADChIMP-cpp` if you have followed the instruction above. The argument `<name_of_folder_with_main_file>` tells in which folder the main-file to be complied is allocated. The standard case, `std_case`, is build if no argument is given. The main-file folders are sub-folders in `BADChIMP-cpp/src`. This script will make a `build` folder, run `cmake` from that folder and then run `make`. This can also be done, by hand, using the following recipe:

```
/BADChIMP-cpp$ mkdir <build-folder-name>
/BADChIMP-cpp$ cd <build-folder-name>
/BADChIMP-cpp$ cmake -DLBMAIN:STRING="<name_of_folder_with_main_file>" ./..
/BADChIMP-cpp$ make
```

3.1.3 Compilation on Windows

Make sure that open MPI is installed[1] Install cmake for Windows.[2]. Run cmake from root directory to generate Visual Studio C++ project, or simply use VSCode.

[1]See https://docs.microsoft.com/en-us/archive/blogs/windowshpc/how-to-compile-and-run-a-simple-ms-mpi-program. Download and run `msmpisetup.exe` and `msmpisdk.msi`.

[2]See https://cmake.org/

Folder name	Description
`std_case`	Standard case, single fluid Navier-Stokes solver
`two_phase`	Two phase simulator

Table 3.1: List of different main files

3.1.4 Making a new main folder

In this sub section we will go through the steps to add a new main folder, which we will call `new_main_folder` in the following, to the code repository.

The first step is to make a new sub directory in the `./BADChIMMP/src/` directory, which we in this example has called `./BADChIMMP/src/new_main_folder`. Then copy the `main.cpp` and `CMakeLists.txt` from the `std_case` folder. If you for some reason want to change the name of the main file in the new main folder, remember to also change the main file-name in `CMakeLists.txt`. This is the whole recipe for creating a new main folder. To generate a `build` folder just follow the instruction given in section 3.1 for running the `make.sh`-script.

3.2 Structure of the main file

3.2.1 Main file outline

Here is an executive overview of the main-file for a standard pore scale simulations. Each section is described in more detail on the following pages.

```
<include libraries>
<set lattice type>

int main()
{
  // SETUP
  <mpi>
  <input and output directories>
  <grid, geometry and mpi-communication>
  <read input-file>
  <macroscopic fields, i.e. density, velocity et c.>
  <boundary conditiones>
  <lattice Boltzmann velocity distributions>
  <vtk-output>

  // TIME LOOP
  for (int i = 0; i <= nIterations; i++) {
   // NODE LOOP
    for (auto nodeNo: bulkNodes) {
      <calculate macroscopic values>
      <calculate collision term>
      <propagate>
    }
    <swap data>
    <mpi-communication>
```

Variable name	description
`nD`	Number of spatial directions
`nQ`	Number of lattice directions
`c2`	$= C_2$
`c2Inv`	$= 1/C_2$
`c4`	$= C_4$
`c4Inv`	$= 1/C_4$
`c4Inv0_5`	$= 1/(2C_4)$

Table 3.2: Basic lattice structure

```
    <apply bondary conditions>
    <write to file>
  }
  <end of program rutines>
}
```

3.2.2 Lattice type

```
// SET THE LATTICE TYPE
#define LT D2Q9
```

The lattice type is set as the macro `LT`. The lattice type is a name of a static class and are defined in header files in `BADChIMP-cpp/src/lbsolver`. The header files for the different lattice types are listed in `BADChIMP-cpp/src/LBSOLVER.h`. The different lattice types are defined in separate head files with the naming convention `LBd<num dim>q<num dir>.h`. We note that in the current setup we have assumed that the last direction, that is, the direction with the larges index, is the rest direction. The naming convention for the static class is `D<num dim>Q<num dir>`. The lattice class contains a number of variables together with functions for performing simple manipulations based on the lattice structure. We will list most of them in the following tables.

The standard lattice structure constants are listed in table 3.2, besides the number of spatial and lattice directions, this also includes different expressions involving the sound speed, c_s, which is defined in section 2.1.5. We will just note the notation $C_2 = c_s^2$ and $C_4 = c_s^4$. Basis vectors and vector operations are defined using functions, but the weights, w_α, and derived quantities are simple to access through standard vector notation, see table 3.3. The variables and functions to access the basis vectors are given in table 3.4.

The function for vector and distribution manipulations are listed in table 3.5. In general, all function in this table that takes vector like input will assume that the elements can be accessed using the `[···]` notation. The functions that return vectors will return `std::valarray<lbBase_t>` type objects.

Finally, we have the constants that relates to multi-phase simulations, where we have an set of constants that is needed for the Reis and Phillips [12] implementations of color gradient methods. The constants are listed in table 3.6 and referrers to Equation. 28 in Reis and Phillips [12],

$$\left(\Omega_q^k\right)^{(2)} = \frac{A_k}{2}|\vec{F}|\left[\frac{\left(\vec{F}\cdot\vec{c}_q\right)^2}{|\vec{F}|^2} - B_q\right].$$

Variable name	description
`w0`	rest particle weight
`w1`	nearest neighbor weight
`w2`	nest nearest neighbor weight
$\vdots$	$\vdots$
`w0c2Inv`	$=$ `w0`$/C_2$
`w1c2Inv`	$=$ `w1`$/C_2$
`w2c2Inv`	$=$ `w2`$/C_2$
$\vdots$	$\vdots$
`w[q]`	$= w_q$ where $0 \leq q <$ `nQ`

Table 3.3: Lattice weights

Variable name	description
`c(q, i)`	$= c_{qi}$
`reverseDirection(q)`	$\vec{c}_{\texttt{reverseDirection(q)}} = -\vec{c}_q$
`cNorm[q]`	$= \lvert\vec{c}_q\rvert$

Table 3.4: Lattice basis vectors

Variable name	description
`dot(u, v)`	$= \vec{u} \cdot \vec{v}$
`grad(s)`	$= \nabla s$
`cDot(q, u)`	$= \vec{c}_q \cdot \vec{u}$
`cDotAll(u)`	$= \{\vec{c}_q \cdot \vec{u} \mid 0 \leq q < \texttt{nQ}\}$
`qSum(f(0, nodeNo))`	$= \sum_\alpha f_\alpha(t, \vec{x})$
`qSumC(f(0, nodeNo))`	$= \sum_\alpha \vec{c}_\alpha f_\alpha(t, \vec{x})$

Table 3.5: Functions used to work with vectors and distributions. More information is given in the text

Variable name	description
`B0`	rest particle weight
`B1`	nearest neighbor weight
`B2`	nest nearest neighbor weight
$\vdots$	$\vdots$
`B[q]`	$= B_q$ where $0 \leq q <$ `nQ`

Table 3.6: Lattice weights for the two phase color gradient method.

3.2.3 mpi (setup)

The lattice Boltzmann code uses the MPI (Message Passing Interface) library to handle parallel computation on clusters of cpu's. The standard setup code for the mpi is

```
// *********
// SETUP MPI
// *********
MPI_Init(NULL, NULL);
int nProcs;
MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
int myRank;
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
```

The variables `nProcs` and `myRank` holds the total number of processors used in the computations and the rank of the current process, respectively.

3.2.4 Input and output directories (setup)

After initializing mpi we need to define the directory paths for input and output files. In BADChIMP there are two types of input files: one relating to geometry and geometry partitioning, and another which supplies the standard input variables.

```
// *******************************
// SETUP THE INPUT AND OUTPUT PATHS
// *******************************
std::string chimpDir = "/BADChIMP-cpp/";
std::string mpiDir = chimpDir + "input/mpi/";
std::string inputDir = chimpDir + "input/";
std::string outputDir = chimpDir + "output/";
```

In the code above the base directory is saved in `chimpDir`. The code searches for the geometry related files in `mpiDir` while the standard input file is in directory `inputDir`. The output files from the simulations are written into `outputDir`.

3.2.5 Setup grid and geometry

This section of the code creates the object that handles input from the user, sets up the system geometry, node partitioning and the mpi-communication protocol. The input datafile, with `input.dat` as default name, is read into an `Input` object, while the vtklb-geometry files are read by an LBvtk-object. The grid-object, that defines the neighborhood of nodes, depends on the vtklb object. The Nodes-object, that holds information about the node type (e.g., if it is fluid, wall etc.), depends on both the grid and vtklb-objects. And finally, the BndMpi, that handles the transfer of data between processor partition boundaries, is dependent on the nodes-, grid-, and vtklb-objects.

```
// ***********************
// SETUP GRID AND GEOMETRY
// ***********************
Input input(inputDir + "input.dat");
LBvtk<LT> vtklb(mpiDir + "tmp" + std::to_string(myRank) + ".vtklb");
Grid<LT> grid(vtklb);
Nodes<LT> nodes(vtklb, grid);
BndMpi<LT> mpiBoundary(vtklb, nodes, grid);
```

In the next sections we will go through the different objects that are declared here.

3.2.6 Input

The Input-object expects an input file in a nested block structure, and uses # for comments. A block is defined by two sets of angle brackets, the first pair contains a name (and possibly a type definition), and the second pair contains a end key word which ends the block.

```
<block-name>
  # bulk content
<end>
```

Blocks can also be nested:

```
<block-nameA>
  # bulk content
  <block-nameB>
    # bulk content
  <end>
  # more bulk content
<end>
```

So, how to we add values to be read by the program? Outside of an block structure one uses the set key-word on the form

```
set name value  # e.g. set pi 3.14
```

In the program you access this value using the input-object

```
double var = input["name"];
```

Inside an block a name followed by either one or a list of numerical vales:

```
<block-name>
  nameA value              # e.g. pi 3.14
  nameB value1 value2 ...  # e.g. prime 2 3 5 7
<end>
```

These inputs are imported in the following manner,

```
double val = input["block-name"]["nameA"];
std::vector<int> valarr = input["block-name"]["nameB"];
// valarr = {2, 3, 4, 5, 7}
```

Nested blocks are accessed in a similar manner, by ordering the block names from left to right, where the left is the outermost block and the right is the innermost, e.g.,

```
<block-nameA>
  <block-nameB>
    pi 3.14
  <end>
<end>
```

is read as

```
double pi = input["block-nameA"]["block-nameB"]["pi"];
```

Values can also be added in matrix form

```
<name type>
 val(0,0)   val(0,1)   ... val(0,m-1)
 val(1,0)   val(1,1)   ... val(1,m-1)
  ...
 val(n-1,0) val(n-1,1) ... val(n-1,m-1)
<end>
```

Here, `type` can be `char` or `int`, or be omitted assuming that the entries are of floating point type. The matrix type is accessed as an vector,

```
// valmat = {val(0,0), val(0,1), ... ,val(0,m-1),
//           val(1,0), ...,val(n-1,m-1)}
std::vector<double> valmat = input["name"];
```

When using a `char` type, the data is read digit by digit and no white-space characters should separate the values which now should be a numerical digit (0-9), e.g.,

```
<binary char>
  0101
  1110
<end>
```

and is read as

```
std::vector<int> binary = input["binary"];
// binary = {0,1,0,1,1,1,1,0}
```

3.2.7 LBvtk

The `LBvtk` object reads the vtklb geometry files together with any appended data. You can use this object to get access to appended data[3]. To get access to the scalar attribute we need to use the class methods `toAttribute` and `getScalarAttribute` to access the data and to get the range of nodes number from `beginNodeNo` and `endNodeNo`. As an example we will show how to read the attribute `"name"` of `double`-type from file and print it to screen,

[3]For now we have only implemented functionality for obtaining scalar attributes.

```
vtklb.toAttribute("name");
for (int n=vtklb.beginNodeNo(); n<vtklb.endNodeNo(); ++n) {
    std::cout << vtklb.getScalarAttribute<double>() << std::endl;
}
```

Here, `toAttribute(std::string s)` takes the attribute name as a string input. `beginNodeNo()` return the first node number read and `endNodeNo()` gives the end range of node numbers. `getScalarAttribute<type>()` reads entries of numerical type `double` which is given as a template typename.

Subset data set are read using the following code:

```
vtklb.toAttribute("new_rho_x");
for (int n=0; n<vtklb.numSubsetEntries(); ++n) {
const auto ent = vtklb.getSubsetAttribure<double>();
std::cout << "Node numer = " << ent.nodeNo << "     value = " << ent.val << std::endl;
}
```

`numSubsetEntries()` gives the size of the data set and `getSubsetAttribure<'numerical type'>()` reads that data entry into a struct object with members `nodeNo` (node number) and `val` (scalar value) of the given numerical type.

3.2.8 Grid

The `Grid` object contains information about which nodes are part of a node's neighborhood and the spatial position of each node. Information about the neighborhood of a node is found by using the `neighborhood`-function,

```
auto neighbor_node = grid.neighbor(alpha, node);
// return the node number of the
// neighboring node in the alpha
// direction

auto neighbor_nodes = grid.neighbor(node);
// Returns a list of the node numbers
// to all nodes in the neighborhood
```

The position in Cartesian coordinates is found by using `pos`,

```
auto pos = grid.pos(node);
// pos = x_pos, y_pos, z_pos

auto pos_i = grid.pos(node, index);
// return the node's position's
// Cartesian index
```

To find a node number from a position array, you can use `Grid`'s `nodeNo`:

```
int node = 10;
auto pos = grid.pos(node);
int node_from_pos = grid.nodeNo(pos);
// node_from_pos = 10
```

The `getNodePos` returns the position of nodes in Cartesian coordinates as an integer vector on the from $[x, y, z]$. The function is overloaded. So, given a list of node numbers, the syntax is

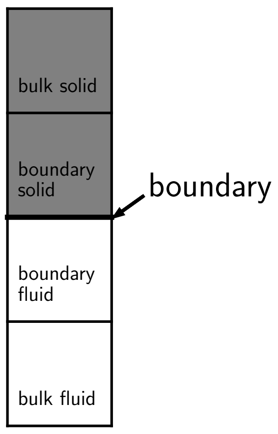


Figure 3.1: A sketch of the different node types. In this figure gray shows the solid part of the geometry and with is the fluid. The thick black line is the solid fluid boundary.

```
std::vector<int> list_of_nodes = {2, 4, 6};
auto list_of_pos = grid.getNodePos(list_of_nodes);
// list_of_pos is of type std::vector<std::vector<int>>
// If
// pos_of_node2 = {1, 1, 2}
// pos_of_node4 = {3, 1, 5}
// pos_of_node6 = {2, 5, 2}
// then
// list_of_pos = {{1, 1, 2},
//                {3, 1, 5},
//                {2, 5, 2}}
```

The function can also be used with *two* arguments: a begin node and an end node number. The function will then return the position as a vector starting with the begin node and ending with the end node. The end node's position is not part of the output vector. An example is given below:

```
int begin_node = 2;
int end_node = 4;
auto list_of_pos = grid.getNodePos(begin_node, end_node);
// list_of_pos = {pos_node2, pos_node3}
```

Finally, to get the total number of node numbers represented in grid we can use `grid.size()`. Note that this includes the potential default ghost node label.

```
auto total_number_node_labels = grid.size();
```

3.2.9 Nodes

The `Nodes` object contains more information about the nodes. Nodes comes in five different types, as far as the `Nodes`-class is concerned: the default ghost node, solid nodes of bulk

Node type	**value**
default ghost	-1
solid bulk	0
solid boundary	1
fluid boundary	2
fluid bulk	3

Table 3.7: Numerical key value for representing node types

and boundary type, and fluid nodes of bulk and boundary type (see Fig. 3.1). A fluid node is a boundary node if it has a solid node in its neighborhood, and vice versa for solid nodes. The fluid node is a bulk node if it is not at boundary node. The reason for this labeling is that a boundary node will have unknown LB distributions after the streaming step, as, at least, one velocity distribution comes from a solid node. Besides the solid and fluid nodes, we can have a default ghost node that is used as a dummy node. The default ghost node is treated as a solid node. Each node type is given a numerical value, as presented in Table 3.7. The node type value is accessed using the `getType` function. The code below prints the node type value for all nodes in the grid-object

```
for (int n=0; n < grid.size(); ++n)
{
    std::cout << "Node number " << n << " ";
    std::cout << "has the node type value " << nodes.getType(n);
    // nodes.getType(n) returns an int
    std::cout << std::endl;
}
```

The `nodes` object also contains many Boolean functions, returning true/false values for the question "is node ... of type ...":

```
int node_number = 4;

nodes.isDefault(node_number);
// return true of the node is a default ghost node, false otherwise

nodes.isBulkSolid(node_number);
// return true of the node is a bulk solid node, false otherwise

nodes.isSolidBoundary(node_number);
// return true of the node is a solid boundary node, false otherwise

nodes.isFluidBoundary(node_number);
// return true of the node is a fluid boundary node, false otherwise

nodes.isBulkFluid(node_number);
// return true of the node is a bulk fluid node, false otherwise
```

We can also ask the `nodes` object if a node is fluid or solid. A fluid node is a node that is either a fluid boundary or a bulk fluid, whereas a solid is either a solid boundary, a solid bulk or a default ghost. The class functions used for this is

```
nodes.isSolid(node_number);
// return true of the node is a solid node, false otherwise

nodes.isFluid(node_number);
// return true of the node is a fluid node, false otherwise
```

The nodes object also holds some information about the parallel partitioning of the system. Spesifically, it holds the information about which processor rank a node represents. Typically, a node that is a bulk fluid node on one processor will be a boundary fluid node on one of its neighboring processors, as it will need to get velocity distributions that are calculated on the former processor. More information about how the code handles parallelization is found in sections A and B. To obtain which processor rank a node represents we can use

```
auto node_rank = nodes.getRank(node_number);
// node_rank is set to an int containg the rank number given by mpi
```

To find out if the rank of a node is the same as the rank of the current process use

```
auto true_false = nodes.isMyRank(node_number);
// True if the node has the same rank as the current process, false otherwise
```

Finally you could ask if it is an mpi boundary node with

```
auto true_false = nodes.isMpiBoundary(node_number);
// True if node is an mpi boundary, false otherwise
```

3.2.10 BndMpi

The BndMpi class holds algorithms for transferring information between processors in a parallel setup. Most of this is done "under the hood" in the constructor when the mpiBoundary object is declared, but we need to specify in the code when we want this communication, between processors, to occur. The BndMpi class contains methods for the communication of scalar fields and velocity distribution fields. The scalar field communication is invoked by the overloaded class method communciateScalarField:

```
ScalarField rho(2, grid.size());

mpiBoundary.communciateScalarField(0, rho);
// communicates rho's 0'th field to the neighboring processors

mpiBoundary.communciateScalarField(1, rho);
// communicates rho's 1'th field to the neighboring processors

mpiBoundary.communciateScalarField(rho);
// communicates all rho's fields to the neighboring processors
// In the case above this is the same as
// mpiBoundary.communciateScalarField(0, rho);
// mpiBoundary.communciateScalarField(1, rho);
```

The ScalarField class is discussed later in this section. The communication of velocity distribution fields are similar to that of the scalar fields:

```
LbField<D2Q9> f(2, grid.size());

mpiBoundary.communicateLbField(0, f, grid);
// communicates f's 0'th field to the neighboring processors

mpiBoundary.communicateLbField(1, f, grid);
// communicates f's 1'th field to the neighboring processors

mpiBoundary.communicateLbField(f, grid);
// communicates all rho's fields to the neighboring processors
```

The `LbField` class is discussed later in this section.

3.2.11 Boundary and bulk nodes

There are several functions that can be used to partition the nodes in the different groups: `findBulkNodes`, `findSolidBndNodes`, `findFluidBndNodes`. The `findBulkNodes` returns a list of bulk node labels, where a bulk node is assumed to be a fluid node that is on the current processor. This function is defined to return the nodes that are part of the standard LB algorithm. The function is overloaded:

```
auto bulkNodes = findBulkNodes(nodes);
// returns a stl int vector with node numbers and
// takes a Nodes-object as input

std::vector<int> marker;
...
auto bulkNodesMarker = findBulkNodes(nodes, marker);
// returns a stl int vector with node numbers, and
// takes a Nodes-object and an stl int vector as input.
// Besides the above mentioned criteria, a bulk node
// will also need to be marked with a zero in the
// maker vector.
```

The `findSolidBndNodes` returns a list of node numbers for the solid boundaries nodes on the given processor. The function is not overloaded:

```
auto solidBoundaryNodes = findSolidBndNodes(nodes);
// returns a std int vector with node numbers and
// takes a Nodes-object as input
```

The `findFluidBndNodes` also returns a list of node numbers of fluid boundary nodes, but in the same way as `findBulkNodes` it is overloaded allowing for the addition of a marker vector:

```
auto fluidboundarNodes = findFluidBndNodes(nodes);
// returns a stl int vector with node numbers and
// takes a Nodes-object as input

std::vector<int> marker;
...
auto fluidboundarNodesMarker = findFluidBndNodes(nodes, marker);
```

direction	
$\vec{c}_\beta$	unknown
$-\vec{c}_\beta$	known
$\vec{c}_\gamma$	known
$-\vec{c}_\gamma$	known
$\vec{c}_\delta$	unknown
$-\vec{c}_\delta$	unknown

Table 3.8: link categories

```
// returns a stl int vector with node numbers, and
// takes a Nodes-object and an stl int vector as input.
// Besides the above mentioned criteria, a bulk node
// will also need to be marked with a zero in the
// maker vector.
```

3.2.12 Boundary conditions

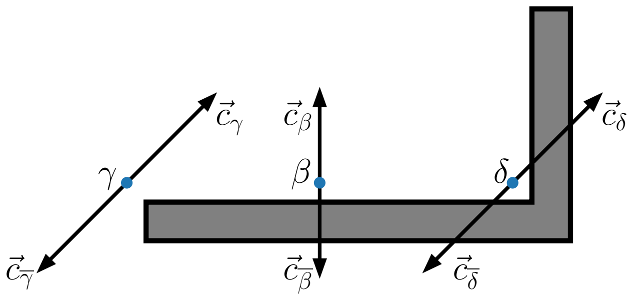
The boundary conditions are usually based on the `Boundary` class. The boundary class contains a list of boundary nodes where the all basis directions are categorized in γ, β, or δ. But first, we will expand a bit on the classification of nodes. As noted before, we call a node solid if that node does not stream a value. Hence, if a neighbor node is a solid, the distribution propagated from that node will be treated as unknown. A solid node will not alway be a solid part of the geometry. For instance, the ghost node neighbors of inlet and outlet boundaries will be treated as solids in the `Node` class. This will be used in the `Boundary` constructor which uses `Nodes`' `isSolid` to decide if a node is a solid or not.

The classification of boundary node directions is based on directions pairs. A pair is defined as a lattice direction and its reverse. As mentioned, there are three types of link pairs β, γ, or δ, which are characterized according to whether or not the values are known after streaming. That is, are values streamed from fluid nodes (known) or from solid nodes (unknown)? The categorization of links are given in Table 3.8, and illustrated in figure 3.2. In a `Boundary` object just one direction for each link is recorded. So, to get all directions we need to use the `revDir` function of the recorded directions to obtain both directions in the pair. Firstly, the `Boundary` class contains a function to get access to the boundary nodes:

```
std::vector<int> list_of_boundary_nodes;
...
// Constructor
Boundary<D2Q9> boundary(list_of_boundary_nodes, nodes, grid);

// Boundary node list
auto number_of_boundary_nodes = boundary.size();
// Holds the length of the list of boundary nodes

int boundaryNumber;
...
auto nodeNumber = boundary.nodeNo(boundaryNumber);
// nodeNumber holds the node number of the boundaryNumber-element in the
// list of boundary nodes.
```



$\vec{c}_\gamma$
$\vec{c}_\beta$
$\vec{c}_\delta$
γ
β
δ
$\vec{c}_{\overline{\gamma}}$
$\vec{c}_{\overline{\beta}}$
$\vec{c}_{\overline{\delta}}$

Figure 3.2: The figure shows the three categories of boundary links, γ, β and δ to the left, middle and right, respectively. Here, we have denoted the revere direction by using an overline-symbol over the Greek-letter label.

Secondly, the `Boundary` class contains a function to obtain the number of pairs in the given categories, and list them. To find the number of links of a given type, at a given boundary node, we can use the following functions:

```
auto number_of_beta_links_inline = boundary.nBeta(boundaryNumber);
// Returns the number of beta links

auto number_of_gamma_links_inline = boundary.nGamma(boundaryNumber);
// Returns the number of gamma links

auto number_of_delta_links_inline = boundary.nDelta(boundaryNumber);
// Returns the number of delta links
```

We note that each link consist of a pair of directions. So, to obtain the total number of directions, we will need to multiply each number-of-links variable by 2. To find the list of links contained in each category, we can use the functions listed below. We note that each element in the list returned contains only one of the directions in a link pair. For the γ and δ directions, we know that both directions are either known or unknown, respectively. But, for the β direction we have specified that the direction returned is the unknown direction.

```
std::vector<int> list_of_beta_directios = boundary.beta(boundaryNumber);
// list of the unknown direction member from each beta link pair for bondary
// member element boundaryNumber

std::vector<int> list_of_gamma_directios = boundary.gamma(boundaryNumber);
// list of one direction member from each gamma link pair for bondary member
// element boundaryNumber

std::vector<int> list_of_delta_directios = boundary.delta(boundaryNumber);
```

```
// list of one direction member from each delta link pair for bondary member
// element boundaryNumber
```

Finally we have added the function `dirRev` to find the reverse direction of lattice direction:

```
int dir;
... // dir will hold an arbitrary lattice direction
auto reverse_dir = boundary.dirRev(dir);
// reverse_dir holds the reverser direction of dir.
```

To show the full use of the `Boundary` class, we will give an example of going through all boundary nodes in the `boundary` object and list all directions classified by link-type, and know and unknown values:

```
for (int n = 0; n < boundary.size(); n++)
{
    int node = boundary.nodeNo(n); // Get node number
    std::cout << "Node number " << node << std::endl;
    // Print all beta values
    std::cout << "beta =";
    auto beta = boundary.beta(n);
    for (int q = 0; q < boundary.nBeta(n); ++q) {
        std::cout << " " << beta[q] << "(Unknown)";
        std::cout << " " << boundary.revDir( beta[n] ) << "(Known)";
    }
    std::cout << std::endl;
    // Print all gamma values
    std::cout << "gamma =";
    auto gamma = boundary.gamma(n);
    for (int q = 0; q < boundary.nGamma(n); ++q) {
       std::cout << " " << gamma[q] << "(Known)";
       std::cout << " " << boundary.revDir( gamma[q] ) << "(Known)";
    }
    std::cout << std::endl;
    // Print all delta values
    std::cout << "delta =";
    auto delta = boundary.delta(n);
    for (int q = 0; q < boundary.nDelta(n); ++q) {
       std::cout << " " << delta[n] << "(Unknown)";
       std::cout << " " << boundary.revDir( delta[n] ) << "(Unknown)";
    }
    std::cout << std::endl;
}
```

Caution: Note that a node number and the numbering of an element in the boundary node list are usually not the same number. We get the node number from the index of the boundary list by using the `nodeNo` function.

The class `HalfWayBounceBack` is used to enforce the half-way bounce-back boundary condition. The declaration of an object of this class needs a list of nodes, a `Nodes` object, and a `Grid` object:

```
auto list_of_fluid_nodes = findFluidBndNodes(nodes);
```

```
// std vecter of node numbers of the nodes that are part of the boundary

HalfWayBounceBack<D2Q9> boundaryCondition(list_of_fluid_nodes, nodes, grid);
// Declaration of the a HalfWayBounceBack object. Besides the
// list-of-fluid-nodes it takes a Nodes object and a Grids object as input.
```

For a fluid simulations, we would expect the list-of-boundary-nodes to be fluid boundary nodes. We would also assume that the solid boundary nodes are part of the system, as this will hold the bounce back part of the distributions. The half-way bounce-back is effectively a part of the propagation routine. The conceptual picture is that the distributions hits a wall, located half way between the origin node and the destination node, where it is flipped and returns to the origin as the distribution going in the opposite direction as it left. This means that it could, and should, be applied straight after the bulk node loop. The boundary condition is applied using the overloaded `apply` class function:

```
LbField<D2Q9> f(2, grid.size());
// f holds two LB distributions

boundaryCondition.apply(0, f, grid);
// Applies the half-way bounce back to field 0

boundaryCondition.apply(1, f, grid);
// Applies the half-way bounce back to field 1

boundaryCondition.apply(f, grid);
// Applies the half way bounce back to all fields represented by f
// It is equivalent to running the two functions above, in this case
```

3.2.13 Field classes

There are currently three field classes in BAChIMP: `ScalarField`, `VectorField`, and `LbField`: These represent fields of scalars, vectors, and LB distributions, respectively. The scalar and vector field objects are used to hold the fluid density and velocity, for instance. Below we see a standard initialization step in a LB simulation, where all densities are set to 1 and all velocities are set to 0.

```
// *****************
// MACROSCOPIC FIELDS
// *****************
// LT is a constant representing a lattice type
//
// Density
ScalarField rho(1, grid.size());
// Velocity
VectorField<LT> vel(1, grid.size());
// Initiate values
for (auto nodeNo: bulkNodes) {
    rho(0, nodeNo) = 1.0;
    for (int d=0; d < LT::nD; ++d)
        vel(0, d, nodeNo) = 0.0;
}
```

The `ScalarField` object declaration, `ScalarField rho(1, grid.size())`, takes two input arguments. The first is the number of fields and the second is the size of the system. In the case here, we have only one field, and the number of elements matches the size of the system. The same input arguments are needed by the `VectorField<LT>` object. We also note that a vector field is a template class. We obtain the values from a scalar object by referring to the field and node number, so that

```
auto value = rho(0, 4);
// value has the type given by lbBase_t.
```

returns the value of the node with node number 4 in the 0'th field. We note that, as is standard in C++, we begin the numbering of fields with 0. There are two way to obtain values from a `VectorField` object:

```
auto value = vel(0, 1, 4);
// Return the value of component 1 at node number 4

auto vector = vel(0, 4);
// returns a std::valarray of lbBase_t elments
// represnting the vector value at node number 4
```

The number of components of a vector in a `VectorField` object is the same as the dimension of the lattice type. To set values we use the same notation as above for the `ScalarField` and for components of vectors for the `VectorField`, but if you want to assign a vector to a node use `VectorField`'s `set` function:

```
VectorField<LT> vector(1, 1);
VectorField<LT> vectorField(1, 10);
...
vectorField.set(0, 4) = vector(0, 0);
// Sets the vector at node 4 equal the value of vector(0, 0);
```

`ScalarField` and `VectorField` objects can be written to file and read using the class functions `writeToFile` and `readFromFile` using the syntax shown below.

```
ScalarField rhoWrite(2, grid.size());
...
std::string filename = "myfile";
rhoWrite.writeToFile(filename); // Write field to file

ScalarField rhoRead(2, grid.size());
rhoRead.readFromFile(filename); // Read from file
```

The syntax is exactly the same for `VectorField` objects. Note that number of fields, number of nodes and, in the case for vector objects, spatial dimension must be the same for the written and read field objects.

The `LbField` class is similar to the `VectorField` class, except that the distribution is represented as a vector with the same number of elements as the number of basis vectors. Below we show how to initialize the distribution, $f_\alpha = w_\alpha \rho$,

```
// **********************
// LB FIELD INITIALIZATION
// **********************
```

```
LbField<LT> f(1, grid.size());  // LBfield
for (auto nodeNo: bulkNodes) {
    for (int q = 0; q < LT::nQ; ++q) {
        f(0, q, nodeNo) = LT::w[q]*rho(0, nodeNo);
    }
}
```

The declaration of a `LbField` object is equal to the declartion of a `VectorField` object, where the first argument is the number of fields and the second is the number of nodes. As for vector fields, you access the element in a `LbField` object using the field number and node number to obtain a vector, containing all distribution values at a node. Or, one can specify field number, direction and node number to access a given direction at a node:

```
auto value = f(0, 7, 2);
// Return the value of the distribution component 7
// at node number 2

auto distribution = f(0, 2);
// returns a std::valarray of lbBase_t elments
// represnting the distribution value at
// node number 4
```

For the propagation step, $f_\alpha(t+1, \vec{x}+\vec{c}_\alpha) = \tilde{f}_\alpha(t, \vec{x})$, we can use a loop over all lattice directions

```
LbField<LT> f(1, grid.size());
LbField<LT> f_tilde(1, grid.size());
...
for (auto nodeNo: bulkNodes) {
...
    for (int q = 0; q < LT::nQ; ++q) {
        f(0, q,  grid.neighbor(q, nodeNo)) = f_tilde(0, q, nodeNo);
    }
}
```

This can also be accomplished using the `propagateTo` function, taking a field number, a node number, a vector of distribution values, and a grid object as input:

```
for (auto nodeNo: bulkNodes) {
    ...
    f.propagateTo(0, nodeNo, f_tilde(0, nodeNo), grid);
}
```

`LbField` has the function `swapData` that lets one `LbField` object swap data with another `LbField` object of the the same type. This is for use int the standard pointer swapping used after propagation in a LB code. An example if given below:

```
LbField<LT> f(1, grid.size());
LbField<LT> f_tmp(1, grid.size());
...
f.swapData(f_tmp);
```

After `swapData` `f` now holds `f_tmp`'s data and vice verse.

`LbField` objects can also be, in the same manner as vector and scalar fields, written and read from files. The syntax is the same.

```
LbField<LT> fWrite(2, grid.size());
...
std::string filename = "myfile";
fWrite.writeToFile(filename); // Write field to file

LbField<LT> fRead(2, grid.size());
fRead.readFromFile(filename); // Read from file
```

Here, number of fields, number of nodes and number of lattice directions, $LT :: nQ$, must match for the written and read field object.

3.2.14 Macroscopic values and collision

The macroscopic values we need to run is the fluid density and velocity. To calculate the density, $\rho = \sum_\alpha f_\alpha$, we use the `calcRho` function:

```
// Copy of local velocity diestirubtion
auto fNode = f(0, nodeNo);
// fNode holds the lb distribution at node with
// node nummber nodeNo

auto rhoNode = calcRho<LT>(fNode);
// rhoNode holds the density
```

Similarly we will calculate the velocity, $\rho\vec{u} = \sum_\alpha \vec{c}_\alpha f_\alpha + 1/2\vec{F}$, using `calcVel`. `calcVel` returns a `std::valarray` and is an overloaded:

```
auto velNode = calcVel<LT>(fNode, rhoNode);
// returns the velocity as if the body force was zero

std::valarray<lbBase_t> force;
...
auto velNodeFroce = calcVel<LT>(fNode, rhoNode, force);
// returns the velocity adjusted for the force according
// to Guo's forcing scheme.
```

The physics of in a lattice Boltzmann simulation is governed by the collision term. Different collision terms are defined in the `Lbcollision.h` file in the `src/lbsolver/` directory. The general scheme for the collision step is to let the functions calculating the collision terms and their corrections return the results as vectors and then add them in the propagation step. For example, the function for the standard BGK-collision term looks like this:

```
inline std::valarray<lbBase_t> calcOmegaBGK(const T &f, const lbBase_t &tau,
const lbBase_t& rho, const lbBase_t& u_sq, const std::valarray<lbBase_t> &cu)
{
    std::valarray<lbBase_t> ret(DXQY::nQ);
    lbBase_t tau_inv = 1.0 / tau;
    for (int q = 0; q < DXQY::nQ; ++q)
    {
        ret[q] = -tau_inv *
           (
             f[q] - rho * DXQY::w[q]*
             (
               1.0
               + DXQY::c2Inv*cu[q]
               + DXQY::c4Inv0_5*(cu[q]*cu[q] - DXQY::c2*u_sq)
             )
           );
    }
    return ret;
}
```

As input it takes a lb distribution `f`, f_α, the collision time `tau`, τ, the density `rho`, ρ, and the two last input parameters are the derived quantities `u_sq`, $\vec{u}\cdot\vec{u}$ and the list of the vector products between the basis vectors and the velocity vector `cu`, $\vec{u}\cdot\vec{c}_\alpha$. We have added $\vec{u}\cdot\vec{u}$ and $\vec{u}\cdot\vec{c}_\alpha$ instead of only the velocity as the former quantities are also need elsewhere in the code block. We note that the function returns a vector that with the same number of elements as the number of directions. In the for-loop we calculate the collision operator, $\Omega_q = -(1/\tau)(f_q - f_q^{\mathrm{eq}})$, and we have written f_q^{eq} and the form

$$\rho w_q \left(1 + \frac{\vec{u}\cdot\vec{c}_q}{c_s^2} + \frac{1}{2c_s^4}\left((\vec{u}\cdot\vec{c}_q)^2 - c_s^2\vec{u}\cdot\vec{u}\right)\right)$$

3.3 Running a Navier-Stokes simulation

In this section we will go through the code in the main file line by line for the setup of a Navier-Stokes simulation. But first we will describe how to use the XX python script to generate a geometry read in the main file.

In the text we will use a numbering of the lines in the actual scripts such that it becomes easy to notice the difference between code listing and comments.

3.3.1 Generating a geometry file

Before we run the LB code, we will need to generate the geometry files that are used as input. The python scripts for generating the geometry files are found in the `PythonScripts` directory. The general script, that is imported into case specific geometry files, is the `vtklb` file.

A note on Cartesian directions: the notation is that the first index in an array refers to the first Cartesian coordinate, the second index refers to the second Cartesian coordinate, and so on. As is standard, the first, second and third Cartesian coordinates are also referred to as the x, y and z coordinates, respectively.

A standard example is given below and is found in the `geometry_demo.py` script:

```
1 #!/usr/bin/env python3
2 from vtklb import vtklb
3 import numpy as np
4 import matplotlib.pyplot as plt
5
```

In line 1, the script only tells the system that the file is a python-3 program and is not necessary. The `vtklb` class is imported in from the `vtklb.py` script in line 2. In lines 3 and 4, we import standard python libraries.

```
 6 # system size: nx  ny
 7 sytems_size = (40, 32)
 8 # set the size of the geometri
 9 geo = np.ones(sytems_size, dtype=int)
10 # partition the system in two
11 geo[20:, :] = 2
12 # Add solids to
13 # - top and bottom plate
14 geo[:,0] = 0
15 geo[:,-1] = 0
16 # - lower left corner
17 geo[:10, :11] = 0
18 # - upper left corner
19 geo[:10, 21:] = 0
20 # - lower right corner
21 geo[30:, :11] = 0
22 # - upper right corner
23 geo[30:, 21:] = 0
24
```

The `sytems_size`, in line 7, sets the system size as 40 grid points in the x-direction and 32 in the y direction. And, in line 9, we define the integer `geo` matrix. This matrix will inform the `vtklb` about the location of solid and fluid nodes, and also the partitioning of the system. Solid nodes will be marked by 0 and fluid nodes are marked with integers from 1 and upwards. This number tells which processor the fluid nodes belong to. This number, for the fluid nodes, is one number higher than the rank of the process it is assigned to, since 0 is already used for solid nodes (or nodes that is not a part of the system). Lines 11 to 23 sets the geometry and the partitioning of the system, as shown in figure 3.3.

```
25 # path to your badchimp folder
26 path_badchimp = "/BADCHiMP/"
27 # generate geometry input file(s)
28 vtk = vtklb(geo, "D2Q9", "x", "tmp", path_badchimp + "input/mpi/")
29
```

The `path_badchimp` holds the path to the top level BADCHiMP directory. The `vtklb` object declaration in line 28 writes the geometry information to files. *The first argument* `geo` gives the geometry, and is discussed before in this section; *the second argument* `"D2Q9"` gives the lattice type which can be a predefined type, as in the script above, or one of your own definition; *The third argument* `"x"` tells in which directions the system is periodic, if for example we want a system to be periodic in the x- and z- directions we would have given `"xz"` as the third argument; *the fourth argument* `"tmp"` sets the base of the output name used as input to the lb code; *the fifth argument* `path_badchimp + "input/mpi/"` gives the directory where the geometry files are written to. It is, of course, important that the file and directory names matches those in the LB code.

```
30 # Set initial density
31 # - get the Cartesian coordinates
```

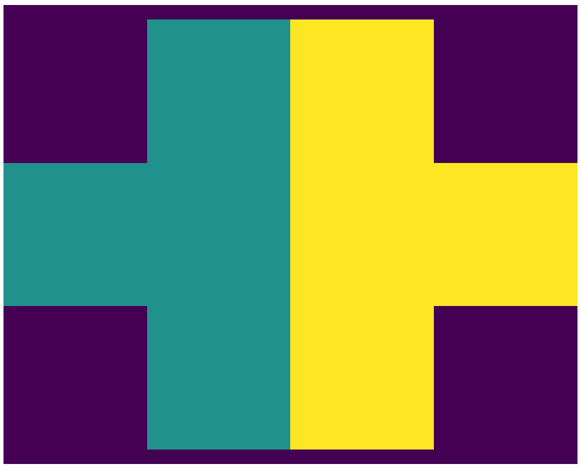


Figure 3.3: A sketch of the geometry. *Color coding*: violet are solids; turquoise are fluid nodes on processor with rank 0; yellow are fluid nodes on processor with rank 1.

```
32 X, Y = np.mgrid[0:40, 0:32]
33 # - set rho
34 rho = 1.0 + 0.1*np.sin(2*np.pi*X/40)
35
36 # Add rho to the geometry files
37 vtk.append_data_set("init_rho", rho)
```

Here, we also want to add an initialization rho. In lines 30 to 35, we define the initial `rho` (shown in figure 3.4). Here, it is important that `rho` is the same size as `geo`. In line 37, `rho` is added to input files using `vtklb`'s class function `append_data_set`, taking two arguments, where the first gives the name of the attribute, `"init_rho"`, and the second gives the actual values, `rho`.

3.3.2 A standard main file

In this section, we describe the main file in `BADChIMP/src/std_case`. The code will simulate a standard Navier-Stokes fluid simulation. We are using the BGK-collision routine, as described in the introduction, and we are using the Guo forcing scheme. The geometry and how to generate it is explained in the above section 3.3.1. The system has two spatial dimensions and we are using the D2Q9 lattice. The initial density, `rho`, is defined in the geometry input file, while the initial velocity is set to zero. The system is driven by a body force, `bodyForce`, and the boundary are of the half-way bounce back type. In the following, a code section is presented first, followed by a more detailed description of its purpose.

```
#include "../LBSOLVER.h"
#include "../IO.h"

// SET THE LATTICE TYPE
#define LT D2Q9
```

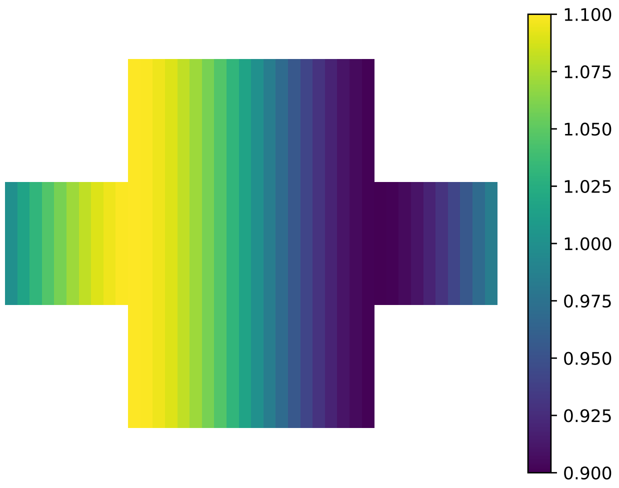


Figure 3.4: The figures shows the initial rho in LB units.

```
15
```

In line 10, we include the libraries for the classes and functions used in the lattice Boltzmann sections and to read the geometry. In line 11, we include the classes used to read the input file and write the fields to vtk-files. In line 14, we set the lattice type to D2Q9.

```
16 int main()
17 {
18   // *********
19   // SETUP MPI
20   // *********
21   MPI_Init(NULL, NULL);
22   int nProcs;
23   MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
24   int myRank;
25   MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
26
```

Line 16 starts the definition of the main function, and with that the beginning of code block that describes the LB simulations. Lines 21 to 25 initiate MPI's parallel program communication. `nProcs` holds the total number of processes used in the parallel simulation of the system, and `myRank` holds the rank of the current processor.

```
27   // ********************************
28   // SETUP THE INPUT AND OUTPUT PATHS
29   // ********************************
30   std::string chimpDir = "BADCHiMP/";
31   std::string mpiDir = chimpDir + "input/mpi/";
32   std::string inputDir = chimpDir + "input/";
33   std::string outputDir = chimpDir + "output/";
```

Line 30 to 33 set the directory paths used to find input files and where to write the output files. `chimpDir` is the path to the main code directory. `mpiDir` is the directory where the python script defining geometry writes its output. This directory needs to match the one given to the `vtklb`-python object. `inputDir` is where the input file, read by the `Input` object, is found. `outputDir` is where BADChIMP writes its output files.

```
35   // ***********************
36   // SETUP GRID AND GEOMETRY
37   // ***********************
38   Input input(inputDir + "input.dat");
39   LBvtk<LT> vtklb(mpiDir + "tmp" + std::to_string(myRank) + ".vtklb");
40   Grid<LT> grid(vtklb);
41   Nodes<LT> nodes(vtklb, grid);
42   BndMpi<LT> mpiBoundary(vtklb, nodes, grid);
43   // Set bulk nodes
44   std::vector<int> bulkNodes = findBulkNodes(nodes);
```

In lines 38 and 39, the input-file and the geometry files are read, as described in sections 3.2.6 and 3.2.7. And, in lines 40 to 42, we define the `Grid`, `Nodes`, and `BndMpi` objects described in sections 3.2.8, 3.2.9, and 3.2.10, respectively. In line 44, we assign the list of bulk nodes to `bulkNodes` which contains the nodes numbers that we will loop over in our main-loop. Node types are described in 3.2.11.

```
44   // *************
45   // READ FROM INPUT
47   // *************
48   // Number of iterations
49   int nIterations = static_cast<int>( input["iterations"]["max"]);
50   // Write interval
51   int nItrWrite = static_cast<int>( input["iterations"]["write"]);
52   // Relaxation time
53   lbBase_t tau = input["fluid"]["tau"];
54   // Body force
55   VectorField<LT> bodyForce(1, 1);
56   bodyForce.set(0, 0) = inputAsValarray<lbBase_t>(input["fluid"]["bodyforce"]);
```

In this section, we assign the values given in the input file to variables. The input-file for this case contains four values: three scalars and one vector. The file looks like this:

```
<iterations>
  max   5000   # stop simulation after
  write 100    # write interval in steps
<end>
<fluid>
  tau 0.8          # Collision time
  bodyforce 1e-5 0  # Body force
<end>
```

The file format for input files is described in section 3.2.6. Here, we see that it contains two block, the first one, `iterations`, sets the number iterations of the write interval. These are set in lines 49 and 51. The second block, `fluid`, assigns the values that are needed in the LB-method. The collision time is set in line 53 and the body force is set in line 56. Note that for `bodyForce` we must first define it (line 55) and then use `VectorField`'s `set` to assign the value.

```
59   // ******************
60   // MACROSCOPIC FIELDS
61   // ******************
62   // Density
63   ScalarField rho(1, grid.size());
64   // Initiate density from file
65   vtklb.toAttribute("init_rho");
66   for (int n=vtklb.beginNodeNo(); n < vtklb.endNodeNo(); ++n) {
67     rho(0, n) = vtklb.getScalarAttribute<lbBase_t>();
68   }
```

Here, we define the fluid density ρ as `rho` (line 63) and it will have the same number of elements as there are grid nodes. The `rho` is initialized from the values given in the geometry files. This must be read using the `vtklb` (in lines 65 to 68). The `LBvtk` class is described in section 3.2.7.

```
70   // Velocity
71   VectorField<LT> vel(1, grid.size());
72   // Initiate velocity
73   for (auto nodeNo: bulkNodes) {
74     for (int d=0; d < LT::nD; ++d)
75       vel(0, d, nodeNo) = 0.0;
76   }
```

The initial velocity is set to zero. The velocity field, $\vec{v}(t, \vec{x})$, is represented by `vel` (line 71) and is initialized to zeros through the for-loop (line 73-76).

```
78   // ******************
79   // SETUP BOUNDARY
80   // ******************
81   HalfWayBounceBack<LT> bounceBackBnd(findFluidBndNodes(nodes), nodes, grid);
```

In line 81, we define object `bounceBackBnd` that we use for the half-way bounce back boundary conditions. The function `findFluidBndNodes` returns a list of fluid boundary node number, as defined in section 3.2.11.

```
83   // *********
84   // LB FIELDS
85   // *********
86   LbField<LT> f(1, grid.size());
87   LbField<LT> fTmp(1, grid.size());
88   // initiate lb distributions
89   for (auto nodeNo: bulkNodes) {
90     for (int q = 0; q < LT::nQ; ++q) {
91       f(0, q, nodeNo) = LT::w[q]*rho(0, nodeNo);
92     }
93   }
```

This section defines and initializes the LB distributions. We define two variables `f` (line 86) and `fTmp` (line 87). `fTmp` is only used as a temporary variable to simplify the propagation step. The initial LB distribution is set, in lines 89 to 93, to $f_q(t = 0, \vec{x}) = w_q \rho(\vec{x})$.

```
 95   // **********
 96   // OUTPUT VTK
 97   // **********
 98   auto node_pos = grid.getNodePos(bulkNodes);
 99   auto global_dimensions = vtklb.getGlobaDimensions();
100   Output output(global_dimensions, outputDir, myRank,
       nProcs, node_pos);
101   output.add_file("lb_run");
102   VectorField<D3Q19> velIO(1, grid.size());
103   output["lb_run"].add_variable("rho", rho.get_data(),
       rho.get_field_index(0, bulkNodes), 1);
104   output["lb_run"].add_variable("vel", velIO.get_data(),
      velIO.get_field_index(0, bulkNodes), 3);
105   outputGeometry("lb_geo", outputDir, myRank, nProcs, nodes, grid, vtklb);
```

Lines 98 to 105 shows how to add input variables to the `Output` object `output`. The plotting has been standardized to work with only 3 dimensional vector fields which is why we have added the *ad hoc* object `velIO` to help plot the velocity. A new plotting routine has been developed but is not fully tested yet. It will soon be include into the code repository.

```
107   // *********
108   // MAIN LOOP
109   // *********
110   for (int i = 0; i <= nIterations; i++) {
```

This section introduces the beginning of the main time iteration loop. The maximum number of iterations is set by `nIterations`.

```
111     for (auto nodeNo: bulkNodes) {
112       // Copy of local velocity diestirubtion
113       const std::valarray<lbBase_t> fNode = f(0, nodeNo);
114
115       // Macroscopic values
116       const lbBase_t rhoNode = calcRho<LT>(fNode);
117       const auto velNode = calcVel<LT>(fNode, rhoNode, bodyForce(0, 0));
```

Here, the loop over all bulk nodes (line 111) is initiated. In this case, this means *all* fluid nodes. `fNode` holds a copy of the local LB distribution (line 113), and is used repeatedly in this block. The fluid density (`rhoNode`)

$$\rho = \sum_\alpha f_\alpha,$$

is calculated in line 116, and the fluid velocity (`velNode`)

$$u_i = \rho^{-1}\left(\sum_\alpha c_{\alpha i} f_\alpha + \frac{1}{2}F_i\right),$$

is calculated in line 117.

```
119       // Save density and velocity for printing
120       rho(0, nodeNo) = rhoNode;
121       vel.set(0, nodeNo) = velNode;
```

rho and vel, updated in lines 120 and 121, are used for printing the field values to file, and are not strictly necessary for the simulation to run.

```
123        // BGK-collision term
124        const lbBase_t u2 = LT::dot(velNode, velNode);
125        const std::valarray<lbBase_t> cu = LT::cDotAll(velNode);
126        const std::valarray<lbBase_t> omegaBGK
                                   = calcOmegaBGK<LT>(fNode, tau, rhoNode, u2, cu);
```

This section defines the BGK-collision term. In line 124, we calculate the square of the velocity, $\vec{u} \cdot \vec{u}$, and, in line 125, we calculate the vector object $\vec{c}_\alpha \cdot \vec{u}$. These two variables are used in calcOmegaBGK, the function that calculates the collision term as described in section 3.2.14.

```
128        // Calculate the Guo-force correction
129        const lbBase_t uF = LT::dot(velNode, bodyForce(0, 0));
130        const std::valarray<lbBase_t> cF = LT::cDotAll(bodyForce(0, 0));
131        const std::valarray<lbBase_t> deltaOmegaF
                                   = calcDeltaOmegaF<LT>(tau, cu, uF, cF);
```

This section defines the correction to the BGK-collision term due to the forcing scheme. Besides cu, that is already evaluated, we need to calculate $\vec{u} \cdot \vec{F}$ (line 129) and $\vec{c}_\alpha \cdot \vec{F}$ (line 130). The correction term

$$\Delta\Omega_\alpha = \left(1 - \frac{1}{2\tau}\right)\left(\frac{\vec{c}_\alpha \cdot \vec{F}}{c_s^2} + \frac{(\vec{c}_\alpha \cdot \vec{F})(\vec{c}_\alpha \cdot \vec{F}) - c_s^2(\vec{u} \cdot \vec{F})}{c_s^4}\right)$$

is calculated in line 131.

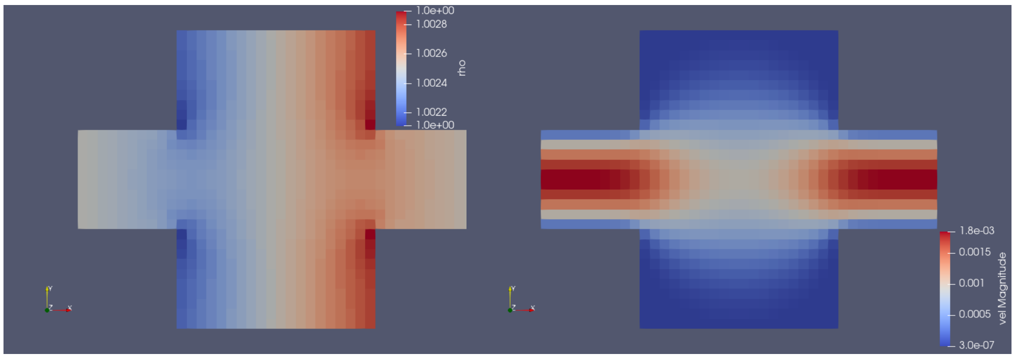
```
133        // Collision and propagation
134        fTmp.propagateTo(0, nodeNo, fNode + omegaBGK + deltaOmegaF, grid);
135
136      } // End nodes
137
138      // Swap data_ from fTmp to f;
139      f.swapData(fTmp);  // LBfield
```

In the last line of the bulk-node loop block, we will propagate the LB distribution in f to the LB distribution fTmp using the LbField function propagateTo, described in section 3.2.13. Line 136 ends the bulk-node for-loop block. To finish the propagation scheme f and fTmp swaps data on line 139 so that we can still use f as our lb distribution when we use boundary condition objects and functions.

```
141      // *******************
142      // BOUNDARY CONDITIONS
143      // *******************
144      // Mpi
145      mpiBoundary.communicateLbField(0, f, grid);
146      // Half way bounce back
147      bounceBackBnd.apply(f, grid);
```

After we have finished with the bulk nodes, we apply the boundary conditions. In line 145, we communicate with neighboring processes, as part of the parallelization scheme. The communication is handled by the `mpiBoundary` object (see section 3.2.10 for more details). The bounce back boundary scheme is conducted in line 147 with `bounceBackBnd`'s `apply` function.

```
149 // *************
150 // WRITE TO FILE
151 // *************
152     if ( ((i % nItrWrite) == 0)  ) {
153       for (auto nn: bulkNodes) {
154         velIO(0, 0, nn) = vel(0, 0, nn);
155         velIO(0, 1, nn) = vel(0, 1, nn);
156         velIO(0, 2, nn) = 0;
157       }
158       output.write("lb_run", i);
159       if (myRank==0) {
160         std::cout << "PLOT AT ITERATION : " << i << std::endl;
161       }
162     }
```



(a) Density (b) Velocity magnitude

Figure 3.5: The figures shows the fluid density in (a) and the velocity magnitude, $\sqrt{\vec{u} \cdot \vec{u}}$, in (b) for the run described in this section. All values are given in LB units.

All field added to the `output` object is plotted by using the `write` command in line 158. In line 153 to 157 we updated the values in the *ad hoc* velocity object `velIO`.

```
164   } // End iterations
165
166   MPI_Finalize();
167
168   return 0;
169 }
```

Finally, we are at the end of the main-file. The time-iteration loop block ends at line 164. The MPI communication is shut down with `MPI_Finalize` and line 168 and 169 ends the main function.

After the program has finished, all files written by the `Output` objects will be found in the folder set by `outputDir` in line 33. These files are in the vtk format, and in this project we have used paraview[4] to view and analyze the files. In figure xx we show the density and velocity magnitude for a run of the case above using the paraview program.

[4]see https://www.paraview.org/

4 Simulation of non-Newtonian rheologies

In many fluids the relation between the viscous stress and the velocity changes can be extremely non- linear, and it may very well depend on other properties of the fluid. These fluids are called non-Newtonian fluids. A subset of these, where $\sigma'_{ij} = 2\mu_{\text{eff}}(\dot{\gamma})E_{ij}$, are classified as generalized Newtonian fluid. Here, the viscosity is a function of the strain rate $\dot{\gamma} = \sqrt{2E_{ij}E_{ij}}$.

For a Carreau-model fluid

$$\sigma'_{ij} = 2\left\{\mu_\infty + (\mu_0 - \mu_\infty)\left[1 + (\lambda\dot{\gamma})^{y_0}\right]^{\frac{n-1}{y_0}}\right\}E_{ij} = 2\mu_{\text{eff}}E_{ij}\,, \tag{4.1}$$

where μ_0 is the viscosity at zero strain rate, μ_∞ is the viscosity at infinite strain rate, λ is the time constant that determines the onset of shear thinning, y_0 is a tuning parameter used to improve the viscosity match at shear rates $\dot{\gamma}/\lambda \sim 1$ (here, $y_0 = 2$), and n is the shear thinning index. This index is known to depend on the polymer concentration. the relaxation time λ normally decreases by decreasing the intrinsic viscosity.

For a Papanastasiou-model type fluid [11]

$$\sigma'_{ij} = 2\left\{\mu_p + \frac{\tau_0\left[1 - \exp\left(-m\sqrt{2E_{ij}E_{ij}}\right)\right]}{2\sqrt{2E_{ij}E_{ij}}}\right\}E_{ij} = 2\mu_{\text{eff}}E_{ij}\,, \tag{4.2}$$

where μ_p is the plastic viscosity of the yielded material, and τ_0 is yield stress.

For a Papanastasiou-model type Herschel-Bulkley fluid

$$\sigma'_{ij} = 2\left\{\mu_p\left(2E_{ij}E_{ij}\right)^{(n-1)/2} + \frac{\tau_0\left[1 - \exp\left(-m\sqrt{2E_{ij}E_{ij}}\right)\right]}{2\sqrt{2E_{ij}E_{ij}}}\right\}E_{ij} = 2\mu_{\text{eff}}E_{ij}\,, \tag{4.3}$$

4.1 Calculating the effective LB relaxation time τ for non-Newtonian rhelogies

In the LB model, it may be shown that the strain rate tensor

$$E_{ij} = \frac{1}{2\rho c_{\text{s}}^2\tau\Delta t}\widetilde{E}_{ij}, \tag{4.4}$$

where $f_\alpha^{\text{neq}} \equiv f_\alpha - f_\alpha^{\text{eq}}$ and

$$\begin{aligned}\widetilde{E}_{ij} &= -\left[\sum_\alpha f_\alpha^{\text{neq}}c_{\alpha i}c_{\alpha j} + \frac{1}{2}\big(u_iF_j + u_jF_i + u_iu_jq + c_{\text{s}}^2q\delta_{ij}\big)\Delta t\right] \\ &= -\left[\sum_\alpha f_\alpha^{\text{neq}}Q_{\alpha ij} + \frac{1}{2}\big(u_iF_j + u_jF_i + u_iu_jq\big)\Delta t\right]\end{aligned} \tag{4.5}$$

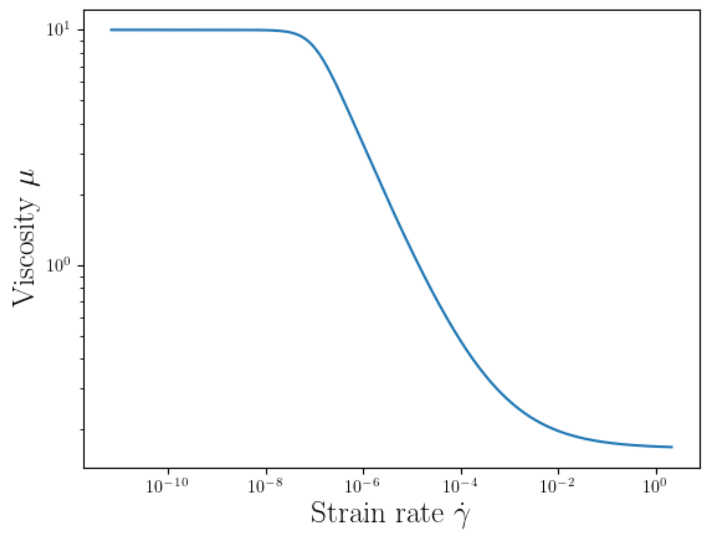


Figure 4.1: Figure of the tabulated viscosity as a function of strain rate, $\dot{\gamma} = \sqrt{2E_{ij}E_{ij}}$.

$$\tau = \frac{1}{\rho c_s^2 \Delta t} \mu_{\text{eff}} \left(\widetilde{E}_{ij} \widetilde{E}_{ij} \right) + \frac{1}{2}, \tag{4.6}$$

where μ_{eff} may be tabulated as a function of the contraction of $\widetilde{E}_{ij}$.

4.2 Running a non-Newtonian flow simulation

To run a simulation of a fluid exhibiting non-Newtonian rheology, the simulator needs the viscosity as a function of the contraction $\widetilde{E}_{ij}\widetilde{E}_{ij}$, where $\widetilde{E}_{ij}$ is defined in Eqs. (4.4) and (4.5). This is done through a file containing the tabulated viscosity values with corresponding values for $\widetilde{E}_{ij}\widetilde{E}_{ij}$. This file may generated using a Python script found in the `PythonScripts` folder found in the code root directory.

4.2.1 Python script for generating a tabulated viscosity file

The python script `viscosity_tabulationDEMO.py` shows an example of how the file containing viscosity values may be generated. Here, the Carreau rheology model (see Eq. (4.1)) is chosen as a demonstration. The Carreau model takes 5 input parameters: μ_∞, μ_0, λ, y_0, and n (see Eq. (4.1) for definitions). In the Python script, these parameters must be given in lattice Boltzmann units. Given these parameters, the script generates a file `test_rheo.dat`. The first line of this file contains the length of table. The rest of the file is divided into 2 columns: the first column contains values of $\widetilde{E}_{ij}\widetilde{E}_{ij}$, while the second column contains the corresponding effective dynamic viscosities μ_{eff}. In figure 4.1 we show the generated viscosity that we will use in the test run in this section.

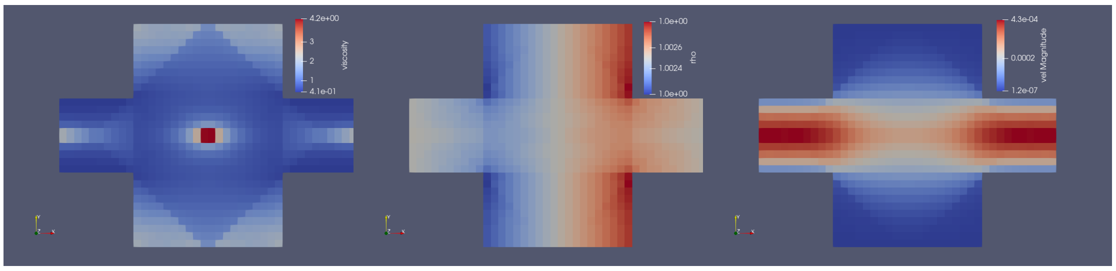
4.2.2 Key features in the main-file

Initializing chosen rheology

Following the directions given in section 4.2.1 for generating an file containing a tabulated viscosity behavior, one needs to facilitate the non-Newtonian flow behavior in the BAD-ChIMP simulator. The first step is to import the viscosity property by constructing a `GeneralizedNewtonian` object in when setting up the grid and geometry.

```
// ********************************
// SETUP THE INPUT AND OUTPUT PATHS
// ********************************
std::string chimpDir = "/BADChIMP/";
...
std::string inputDir = chimpDir + "input/";
...
// **********************
// SETUP GRID AND GEOMETRY
// **********************
...
// Read rheology
GeneralizedNewtonian<LT> carreau(inputDir + "test.dat");
```

In our example we have chosen the object name `carreau`. In the initialization step above, it takes the `test.dat` file as input. This file, which can be generated by following the directions given in section 4.2.1, must be located in the directory `inputDir`.



(a) Viscosity (b) Density (c) Velocity magnitude

Figure 4.2: The figures shows the viscosity in (a), the fluid density in (b) and the velocity magnitude, $\sqrt{\vec{u} \cdot \vec{u}}$, in (c) for the run described in this section. All values are given in LB units.

The non-Newtonian behavior is included in the main time loop through the the BGK collision term `omegaBGK` and, through the modified relaxation time `tau`, the force term `deltaOmegaF`.

```
// *********
// MAIN LOOP
// *********
...
auto omegaBGK = carreau.omegaBGK(fNode, rhoNode, velNode, u2, cu,
                                  bodyForce(0, 0), 0);
...
tau = carreau.tau();
```

```
viscosity(0, nodeNo) = carreau.viscosity();
const std::valarray<lbBase_t> deltaOmegaF = calcDeltaOmegaF<LT>(tau, cu, uF, cF);
```

Note that `carreau.omegaBGK()` must be called ahead of `carreau.tau()` in order to get the updated value of `tau`. We can also get the viscosity by using `GeneralizedNewtonian`'s `viscosity` function. The code snippets above is copied from the main-file in folder `/src/polymer_viscosity_tabulation`.

In figure 4.2 we show the viscosity, density and velocity magnitude for a Carreau rehology. The geometry and the body force is equal to the standard case presented in section 3.3.2.

5 Future work/plans

Our main plan for the numerical package developed in this project is that we aim to include all new LB methods developed in future projects into the BADChIMP code repository, so that it will become available for all users. Hopefully, this will also keep the development of the core libraries up to speed with new developments in both hardware and software, so that the code will evolve and improve as a general tool for scientific computations.

The code and user manual will continuously be improved as stated in the introduction. It is our intention that new methods and features will be added as their own chapters, so that the core of this manuscript will stay more or less the same.

The first planned extension of the code is to include more general polymer models like the C-Fene-P model. Numerical methods are currently being developed by PhD student Bjarte Hetland, as part of his thesis work, and will be include in the BADChIMP code when finished.

The lattice Boltzmann code is also used in projects on bio-cementation, turbulent flow simulations for wind turbine modeling, and three phase systems, water, oil, and CO_2, for the study of CO_2 storage in old oil reservoirs. We expect that the algorithms developed in these project will become part of the BADChIMP code.

On a more code specific level, we also plan to improve our geometry input scheme so that we could add vector and tensor fields as input. We would also like to add attributes to a small subsets of nodes to reduce the size of input files.

A Parallelization

A.1 Deadlock prevention

In the LB code each processor has a list of neighboring processors which it sends and receives data from. This list has the following properties:

1. If processor A is in processor B's list of neighbors, then processor B will be in processor A's list of neighbors.
2. Processor A is not part of its own list of neighbors.
3. A list of neighbors is sorted in ascending order, based on rank.

The send/receive code follows the structure given below:

```
for (auto neigRank: listOfNeighbors) {
    if (myRank < neigRank) {
        // SEND DATA to neigRank
        // RECEIVE DATA from neigRank
        } else {
        // RECEIVE DATA from neigRank
        // SEND DATA to neigRank
    }
}
```

We will argue, based on the Coffman conditions[1], that this structure is enough to avoid *deadlock*.

Here, we will show that the assumption of a *circular wait condition* will lead to a contradiction. We assume that there are n processes (or processors), p, waiting for each other, so that p_0 is waiting for p_1, p_1 is waiting for p_2, and so on until p_{n-1} is waiting for p_0. We call the set of processes that are part of the circular wait loop for a circular wait-set (CWS). Now, let p_k be the process with the lowest rank in a CWS. Since it has the lowest rank, it must be waiting to send to a process, p_{k+1}, with a higher rank, as given by the code structure above. And, since p_{k+1} is in p_k's list of neighbors, p_k is in p_{k+1}'s list of neighbors. p_{k+1} must either be waiting to send to, or receive from, p_{k+2}. If p_{k+2} is equal to p_k we know that p_k's rank is lower than p_{k+1}'s and p_{k+1} should be waiting to receive data from p_k. But this makes the *circular wait condition* stated above invalid. Hence, p_{k+2} must be a different process than p_k. We then know, by assumption, that the rank of p_{k+2} is higher than p_k's, which means it is in a later position in p_{k+1}'s list of neighbors (since it was sorted in ascending order). p_{k+1} should then already have been waiting to receive data from p_k, as the list of neighbors is traversed from lowest to highest values. But since p_k is

[1]See: https://en.wikipedia.org/wiki/Deadlock

already waiting to send to p_{k+1}, this will again make the *circular wait condition* false, as p_k would no longer wait to send to p_{k+1}. Thus, the assumption of a *circular wait* condition leads to a contraction, which proves, by *reductio ad absurdum*, that the proposed parallel communication protocol cannot lead to a *deadlock* situation.

B Geometry file format

This is a description of the geometry file format we have called *vtklb*. This is the format that is read by BADChIMP to include geometries and spatial varying data. This formate is based on the VTK-file format[1].

The overall tile format is given below.

```
# BADChIMP vtklb Version 0.1     <header>
one line descriptioin            <title>
ASCII | BINARY                   <data type>
DATASET type                     <geometry/topology>
...
POINT_DATA n                     <dataset attributes>
...
```

B.1 Geometry/topology

The available dataset-type for geometry/topology is the "unstructured lb grid" and "structured lb grid":

```
DATASET UNSTRUCTURED_LB_GRID
NUM_DIMENSIONS nd               <nd: int number of spatial dimension>
GLOBAL_DIMENSIONS n0 n1 ...     <Size of the bounding box of the system>
USE_ZERO_GHOST_NODE             <use node 0 as the default ghost node>
POINTS n dataType               <n: number of points>
p1_xp1_y...                     <point/node spatial position>
p2_xp2_y...
...
LATTICE nq dataType             <nq: number of basis vectors>
c0_xc0_y...                     <basis vecotor elements>
c1_xc1_y...
...
NEIGHBORS dataType              <list of neighbor nodes for each node>
i1_0i1_1...i1_(nq-1)            <node number of neighbor nodes>
i2_0i2_1...i2_(nq-1)
...
in_0in_1...in_(nq-1)
PARALLEL_COMPUTING rank         <rank: rank of the current processor>
PROCESSOR n rank                <n: number of point, rank: neighbor processor>
i0j0                            <i: node this rank, j: node neighbor rank>
i1j1
```

[1] https://vtk.org/wp-content/uploads/2015/04/file-formats.pdf

```
...
i(n-1)j(n-1)
```

Comments to entries:

`dataType`: Numrical type of entry data, i.e. `int` (integer) , `float` (float), `double` (double), ...

`NUM_DIMENSIONS`: (Optional, Default: nd=3) Specify number of spatial dimensions. In vtk it seems that all vectors and positions are given with three components.

`GLOBAL_DIMENSIONS`: Gives the size of the system in each Cartesian direction include the ghost node rim. This entry should be the same for all processors. This information is needed by the vtk-output routine.

`USE_ZERO_GHOST_NODE`: (Optional) Treat node 0 as a placeholder for a node that is not in use (i.e solid nodes). This means that we begin numbering points from 1, *not* 0. If the key word is written then this feature is enabled.

`POINT`: Same as for the vtk-format. NB if `USE_ZERO_GHOST_NODE` is enabled then the the first entry has node number 1, the next has 2 and so on. If `USE_ZERO_GHOST_NODE` is disabled the first entry has node number 0.

`LATTICE`: Description of the set of basis vector. We can check these with the ones used in the BADChIMP code and map one set to the other if that is needed.

`NEIGHBORS`: For each node there is a list of `nq` nodes. The node number corresponds to the position in the list under the `POINTS`-keyword and the position in the list of neighbors corresponds to the basic vector in the list under the `LATTICE`-keyword.

`PARALLEL_COMPUTING`: Begins the block describing the processor-to-processor communication. `rank` is the rank of the current processor.

`PROCESSOR`: Information about neighboring processor. `n` is the number of nodes at the current processor that is used to represent the nodes at a neighboring process. `rank` is the rank of the neighboring node. In the list under the keyword, the first entry, `i`, is the node number in the geometry in the current processor that represent the node with number `j` in the neighboring rank, which is the second entry on the line.

```
DATASET STRUCTURED_LB_GRID
NUM_DIMENSIONS nd               <nd: int number of spatial dimension>
GLOBAL_DIMENSIONS n0 n1 ...     <Size of the bounding box of the system>
LOCAL_DIMENSIONS n0 n1 ...      <Size of the system on the given processor
                                 excluding the rim of ghost nodes>
LOCAL_RIM_WIDTH nw              <nw: int number of rim width>
LOCAL_TO_GLOBAL_POS po p1 ...   <Sets the global position relative to
                                 the local origo>
LATTICE nq dataType             <nq: number of basis vectors>
c0_xc0_y...                     <basis vecotor elements>
c1_xc1_y...
...
PERIODIC_NODES n rank           <n: number of point, rank: neighbor processor>
i0j0                            <i: 'ghost' node, j: bulk node>
```

```
i1j1
...
PARALLEL_COMPUTING rank         <rank: rank of the current processor>
PROCESSOR n rank                <n: number of point, rank: neighbor processor>
i0j0                            <i: node this rank, j: node neighbor rank>
i1j1
...
i(n-1)j(n-1)
```

Additional comments to entries:

`PERIODIC_NODES`: Sets the nodes that are periodic on one given processor. This needs to be handled separately.

B.2 Dataset attributes

Here we use the same formalism as the vtk-file format using the keywords, e.g., `SCALARS` and `VECTORS`. This information is written after the `POINT_DATA`-keyword.

```
POINT_DATA n                     <n: number of point>
SCALARS dataName dataType        <list of scalar values>
s0                               <s: scalar numerical value of type dataType>
s1
...
s(n-1)
```

In addition to the `POINT_DATA` data attributes, we have the `POINT_DATA_SUBSET` data sets where only a certain sub-set of nodes are assigned values the data entries will then be pairs of the node number and value:

```
POINT_DATA_SUBSET
SCALARS n dataName dataType      <n: number of nodes in the subset>
i0 s0                            <i: node id>
i1 s1                            <s: scalar numerical value of type dataType>
...
i(n-1) s(n-1)
```

Python code for appending a `POINT_DATA_SUBSET`:

```
#----------------------------------------------------- Initiate the data subset section
vtk.begin_point_data_subset_section()

#----------------------------------------------------- Test case
mask = np.full(system_size, False)
mask[:, 16] = True
data1[mask] = setdata1(mask)  # Set data at the masked values
vtk.append_data_subset("subsetdata1", data1, mask)

mask[:] = False
mask[16, :] = True
data2[mask] = setdata2(mask)  # Set data at the masked values
vtk.append_data_subset("subsetdata1", data2, mask)
```

One important point is that we must initiate the point data section in the python-script. This is done only once using `begin_point_data_subset_section()` function. After that, we can add as many data-subset as we want using the function

```
vtk.append_data_subset("dataset_name", scalar_data, mask)
# dataset_name: name that is used at a lable of the data set and used
#               in BADChIMP to find the data
# scalar_data: a numpy array of scalar values with the same shape as the
#              geometry array
# mask: boolean array where True-values define the data subset.
```

B.3 Example of node numbering

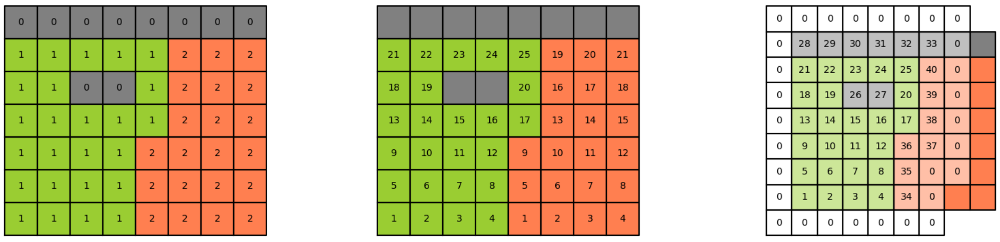


Figure B.1: Concept figure for the node numbering scheme used in BADChIMP. Left: Geometry where 0 is solid, 1 shows nodes on processor with rank 0, and 2 shows nodes on processor with rank 1. Middle: Shows the local node labels on the two processors. Right: Local labeling for processor 1 with rank 0.

Figure B.1 (left) shows an example geometry, where the green and orange areas shows the partitioning of the computational nodes between processor 1 and 2. In each processor, the fluid nodes are consecutively labeled starting from 1.

Figure B.1 (middle) shows labeling of the fluid nodes, where the solid nodes (in gray) will not be given global labels. For standard fluid flow simulations we would like to allocate memory for the solid wall nodes, but we do not need to transfer wall values between processors.

Figure B.1 (right) shows the local labeling for processor 1. Here, the labeling of the fluid nodes follow that of the fluid nodes on processor 1 illustrated in the middle figure. Note, however, that the fluid nodes that belongs to the neighboring processor are relabeled. These nodes need to be linked to the local labels on processor 2. The zero label is used as a default ghost node.

References

[1] Takashi Abe. Derivation of the Lattice Boltzmann Method by Means of the Discrete Ordinate Method for the Boltzmann Equation. *Journal of Computational Physics*, 131(1):241–246, February 1997.

[2] Olav Aursjø, Espen Jettestuen, Jan Ludvig Vinningland, and Aksel Hiorth. On the inclusion of mass source terms in a single-relaxation-time lattice Boltzmann method. *Physics of Fluids*, 30(5):057104, May 2018.

[3] George Keith Batchelor. *An introduction to fluid dynamics.* Cambridge university press, 2000.

[4] Prabhu Lal Bhatnagar, Eugene P. Gross, and Max Krook. A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems. *Physical review*, 94(3):511, 1954. Publisher: APS.

[5] Sydney Chapman and Thomas George Cowling. *The mathematical theory of non-uniform gases: an account of the kinetic theory of viscosity, thermal conduction and diffusion in gases.* Cambridge university press, 1990.

[6] Xiaoyi He and Li-Shi Luo. A priori derivation of the lattice Boltzmann equation. *Physical Review E*, 55(6):R6333–R6336, June 1997. Publisher: American Physical Society.

[7] Xiaoyi He and Li-Shi Luo. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Physical Review E*, 56(6):6811–6817, December 1997.

[8] Timm Krüger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Viggen. *The Lattice Boltzmann Method.* Springer, 2017.

[9] Timm Krüger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Viggen. *The Lattice Boltzmann method: principles and practice.* Graduate texts in physics. Springer, Cham, 2017. OCLC: 965376988.

[10] L. D. Landau and E. M. Lifshitz. *Fluid Mechanics, 2nd Ed.* Butterworth-Heinemann, 1987.

[11] Tasos C Papanastasiou. Flows of materials with yield. *Journal of Rheology*, 31(5):385–404, 1987.

[12] T Reis and T N Phillips. Lattice Boltzmann model for simulating immiscible two-phase flows. *Journal of Physics A: Mathematical and Theoretical*, 40(14):4033–4053, April 2007.

[13] D. H. Rothman and S. Zaleski. *Lattice-gas cellular automata: simple models of complex hydrodynamics*. Cambridge Univ Press, 1997.

[14] Sauro Succi. *The lattice Boltzmann equation: for fluid dynamics and beyond*. Oxford university press, 2001.

[15] M. C. Sukop and D. T. Thorne. Lattice Boltzmann modeling: an introduction for geoscientists and engineers, 2005. *Google Scholar Google Scholar Digital Library Digital Library*.

[16] Dieter A. Wolf-Gladrow. *Lattice-gas cellular automata and lattice Boltzmann models: an introduction*. Springer, 2004.

[17] Stephen Wolfram. Cellular automaton fluids 1: Basic theory. *Journal of statistical physics*, 45(3):471–526, 1986. Publisher: Springer.