OpenLB User Guide
Associated to Release 1.2 of the Code
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the Section entitled “GNU Free Documentation License”.
# Contents

1 Introduction .......................... 7
   1.1 Fluid Flow Simulations .......................... 7
   1.2 Lattice Boltzmann Methods .......................... 7
      1.2.1 Methods for Numerical Flow Simulation .......................... 7
   1.3 OpenLB Project .......................... 8
      1.3.1 What is OpenLB? .......................... 8
      1.3.2 Getting Help with OpenLB .......................... 9
      1.3.3 Compiling OpenLB Programs .......................... 9
      1.3.4 Which features are currently implemented? .......................... 10
      1.3.5 Project Participants .......................... 11

2 Using OpenLB for Applications .......................... 14
   2.1 Lesson 1: Getting Started - Sketch of Application .......................... 15
   2.2 Lesson 2: Understand the BlockLattice .......................... 19
   2.3 Lesson 3: Define and Use Boundary Conditions .......................... 22
   2.4 Lesson 4: UnitConverter - Lattice and Physical Units .......................... 27
   2.5 Lesson 5: Extract Data From a Simulation .......................... 29
   2.6 Lesson 6: Convergence Check .......................... 31
   2.7 Lesson 7: Use an External Force .......................... 32
   2.8 Lesson 8: Understand Genericity in OpenLB .......................... 32
   2.9 Lesson 9: Use Checkpointing for Long Duration Simulations .......................... 34
   2.10 Lesson 10: Save Memory when Domain Boundaries are Irregular .......................... 35
   2.11 Lesson 11: Run Your Programs on a Parallel Machine .......................... 35

3 Install Dependencies .......................... 37
   3.1 Linux .......................... 37
   3.2 Mac .......................... 37
   3.3 Windows .......................... 38

4 Non-dimensionalisation and Choice of Simulation Parameters .......................... 39
5 Geometry

5.1 Creating a Geometry

5.2 Setting Material Numbers

5.3 Building Geometry by Geometric Primitives

5.4 Excursion: Creating STL-files

6 Lattice Boltzmann Models and Core Data Structures

6.1 Concept - Data Organization

6.1.1 Cell - BlockLattice - SuperLattice

6.1.2 Descriptor

6.1.3 Dynamics

6.2 Classic BGK Model

6.3 MRT Model

6.4 Porous Media Model

6.5 Power Law Model

6.6 External Force

6.7 Multiphysics Couplings

6.7.1 Shan-Chen Model

6.7.2 Implementation of Shan-Chen Two-phase Fluid

6.7.3 Implementation of Shan-Chen Two-component Fluid

6.7.4 Coupling Between Momentum and Energy Equations

6.8 Advection Diffusion Equation

6.8.1 Advection Diffusion Boundary Conditions

6.8.2 Convergence Criterion

6.8.3 Creating an Application with Advection-Diffusion Dynamics

6.8.4 Obtaining Results in Thermal Simulations

6.8.5 Conduction Problems

7 Discrete Particle Method

7.1 Structure of the Particles Systems

7.2 Interpolation of Fluid Velocity

7.3 Class SuperParticleSystem3D

7.4 Implementation of the Communication Optimal Strategy

7.4.1 Shadow Particles

8 Input / Output

8.1 Output Data Structure
12.2 bifurcation3d .......................................................... 112
  12.2.1 eulerEuler .................................................. 112
  12.2.2 eulerLagrange ............................................. 113
12.3 bstep2d and bstep3d ............................................. 114
12.4 cavity2d and cavity3d .......................................... 114
12.5 cylinder2d and cylinder3d ..................................... 114
12.6 multiComponent2d and multiComponent3d .................... 114
12.7 nozzle3d .......................................................... 115
12.8 phaseSeparation2d and phaseSeparation3d ................... 117
12.9 poiseuille2d and poiseuille3d ................................ 118
12.10 porousPoiseuille2d ............................................ 118
12.11 powerLaw2d .................................................... 119
12.12 tgv3d ............................................................ 119
12.13 thermalFlows .................................................. 120
  12.13.1 porousPlate2d and porousPlate3d ....................... 120
  12.13.2 rayleighBenard2d and rayleighBenard3d ............... 121
  12.13.3 squareCavity2d and squareCavity3d .................... 122
12.14 venturi3d ........................................................ 124

13 License .............................................................. 131
1 Introduction

1.1 Fluid Flow Simulations

1.2 Lattice Boltzmann Methods

1.2.1 Methods for Numerical Flow Simulation

For the numerical result of the Lattice-Boltzmann-equation you need a phase space which discretized with a lattice. By include other physical processes could be determined flow simulation. This is based on simplified particles-microdynamik and the interaction between particles are based on the Lattice-Boltzmann-equation. The Lattice-Boltzmann-equation has to solve with discretization with introduce a lattice, which is also discretised the velocity. The whole phase space could discretized for example at 2-D with D2Q9-model. A fluidpartical could stand on the same place per time step or in the adjacent spaces, so the partical has nine velocity $\vec{c}_i$ ($i=0,...,8$, various directions).

A logarithm have two partial step: 1. collision step 2. streaming step

At the collisionstep must receive mass and impulse. To every distribution function $f_i$ at the place $\vec{x}$ is add a calculated collision term $\delta_i$:

$$ f_i^{*}(\vec{x}, t) = f_i(\vec{x}, t) + \delta_i \Omega(x, t) \quad (1.1) $$

Possible collision term is Bhatnager-Gross-Krook (BGK)

$$ \Omega_i = \frac{1}{\tau}(f_i - f_i^{eq}) \quad (1.2) $$

$\tau$ describe how fast the fluid approaches equilibrium and depends on the viscosity. $f_i^{eq}$ is the lokal equilibrium function which approximated the Maxwell-Boltzmann-distribution. At the streaming step all arrows are shifted to the next grid point:

$$ f_i(\vec{x} + \vec{c}_i \delta t, \vec{c}_i, t) = f_i^{*}(\vec{x}, t) \quad (1.3) $$

The shifted arrows form the initial situation for the next collision step.
This text is directed at people who want gain insight into *Lattice-Boltzmann Methods (LBM)*.

- The most recent publication that this documentation refers to was written by Krüger et al. [31]. The book *The Lattice Boltzmann Method* delivers a clear and complete introduction for beginners.

- A concise introduction is given by A. A. Mohamad [38]. In his book *LATTICE BOLTZMANN METHOD* [2011], he shows clearly, how one obtains macroscopic equations from *LBM* using *Chapman-Enskop expansion*.

- The readers who want insight into *Lattice-Gas Cellular Automatas* - the historical origin of *LBM* - may want to refer to Dieter A. Wolf-Glodrow’s book *LATTICE-GAS CELLULAR AUTOMATA AND LATTICE BOLTZMANN MODELS* [2000]. Starting with *Cellular Automata*, he explores the beauty of *LBM*. A helpful interpretation of *LBM* is given in the beginning of the book.

- A quick overview of *LBM*, is obtained from the often cited paper of S. Chen and G. D. Doolen *LATTICE BOLTZMANN METHOD FOR FLUID FLOWS* published in 1998.

### 1.3 OpenLB Project

#### 1.3.1 What is OpenLB?

OpenLB is a numerical framework for lattice Boltzmann simulations, created by students and researchers with different backgrounds in computational fluid dynamics. The code can be used by application programmers to implement specific flow geometries in a straightforward way, and by developers to formulate new models. For this first group of users, OpenLB offers a neat interface through which it is possible to set up a simulation with little effort. For the second group, the structure of the code is kept conceptually simple, implementing basic concepts of the lattice Boltzmann theory step-by-step. Thanks to this, the code is an excellent framework for programmers to develop pieces of reusable code that can be exchanged in the community.

One key aspect of the OpenLB code is genericity in its many facets. The core concept of generic programming is to offer a single code that can serve many purposes. On one hand, the code implements dynamic genericity through the use of object-oriented interfaces. One use of this is that the behavior of lattice sites can be modified during
program execution, to distinguish for example between bulk and boundary cells, or to modify the fluid viscosity or the value of a body force dynamically. Furthermore, C++ templates are used to achieve static genericity. As a result, it is sufficient to write a single generic code for various 3D lattice structures, such as D3Q15, D3Q19, and D3Q27 (for more information on lattice structures, see Section 6.1.2).

1.3.2 Getting Help with OpenLB

The following resources are available for OpenLB users:

**Web site.** Most recent releases of the code and documentation, including this user guide, are found on the website [http://www.openlb.net/](http://www.openlb.net/).

**Forum.** If you experience troubles with OpenLB, you may wish to post your concerns to the Lattice Boltzmann community in the forum on the OpenLB homepage.

**Bug reports.** If you think you found a bug in OpenLB, we encourage you to submit a report to bug@openlb.net. Useful bug reports include the full source code of the program in question, a description of the problem, an explanation of the circumstances under which the problem occurred, and a short description of the hardware and the compiler used. Moreover, other Makefile switches, such as buildtype and mode of parallelization found in config.mk can provide useful information too.

1.3.3 Compiling OpenLB Programs

*Note: The framework for compiling OpenLB code is based on Makefiles and has so far been tested only on platforms of the Linux/Unix family, including Mac OS X and Cygwin. If you are working under Windows and want to get started quickly, you might consider installing the free Cygwin software [1], which efficiently emulates a Posix environment under Windows.*

OpenLB consists of generic, template-based code, which needs to be included in the code of application programs, and precompiled libraries that are to be linked with the program. The installation process is light and does not require an explicit precompilation and installation of libraries. Instead, it is sufficient to unpack the source code into an arbitrary directory. Compilation of libraries is handled on-demand by the Makefile of an application program.

To get familiar with OpenLB, new users are encouraged to have a look at programs in the examples directory. Once inside one of the example directories, entering the command make will first produce libraries and then the end-user example program. This
close relationship between the production of libraries and end-user programs reflects the fact that many OpenLB users presently tend to play around with the OpenLB code as well.

The file config.mk in the root directory can be easily edited to modify the compilation process. Available options include the choice of the compiler (GNU g++ is the default), optimization flags, and a switch between normal/debug mode, and between sequential/openmp-parallel/mpi-parallel programs.

To compile your own OpenLB programs from an arbitrary directory, make a copy of a sample Makefile. Edit the `ROOT :=` entry to indicate the location of the OpenLB source, and the `OUTPUT :=` entry to explicit the name of your program, without file extension.

### 1.3.4 Which features are currently implemented?

#### Lattice Boltzmann Models

<table>
<thead>
<tr>
<th>Feature</th>
<th>Section</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGK model for fluids</td>
<td>6.1.3</td>
<td>[17]</td>
</tr>
<tr>
<td>Regularized model for fluids</td>
<td>6.1.3</td>
<td>[33]</td>
</tr>
<tr>
<td>Multiple Relaxation Times (MRT)</td>
<td>6.1.3</td>
<td>[19, 50]</td>
</tr>
<tr>
<td>Entropic Lattice Boltzmann</td>
<td>6.1.3</td>
<td>[10]</td>
</tr>
<tr>
<td>BGK with adjustable speed of sound</td>
<td>6.1.3</td>
<td>[2, 18]</td>
</tr>
<tr>
<td>BGK and MRT with Smagorinsky model</td>
<td>6.1.3</td>
<td>[40]</td>
</tr>
<tr>
<td>Porous media model</td>
<td>6.1.3</td>
<td></td>
</tr>
<tr>
<td>Power law model</td>
<td>6.1.3</td>
<td></td>
</tr>
</tbody>
</table>

#### Multiphysics Coupling

<table>
<thead>
<tr>
<th>Feature</th>
<th>Section</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shan-Chen two-component fluid</td>
<td>6.7</td>
<td>[44]</td>
</tr>
<tr>
<td>Thermal fluid with Boussinesq approximation</td>
<td>6.7</td>
<td>[24]</td>
</tr>
</tbody>
</table>

#### Lattice Structures

<table>
<thead>
<tr>
<th>Lattice</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2Q9</td>
<td>This lattice is available in the precompiled library</td>
</tr>
<tr>
<td>D3Q13</td>
<td>This lattice requires the use of a specific dynamics object (see also Ref. [20])</td>
</tr>
<tr>
<td>D3Q15</td>
<td></td>
</tr>
<tr>
<td>D3Q19</td>
<td>This lattice is available in the precompiled library</td>
</tr>
<tr>
<td>D3Q27</td>
<td></td>
</tr>
</tbody>
</table>
Boundary Conditions for Straight Boundaries (Including Corners)

<table>
<thead>
<tr>
<th>Regularized</th>
<th>Finite difference (FD) velocity gradients</th>
<th>Inamuro</th>
<th>Zou/He</th>
<th>Non-linear FD velocity gradients</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>non-local</td>
<td>local</td>
<td>local</td>
<td>non-local</td>
</tr>
<tr>
<td>Default choice for local boundaries</td>
<td>Default choice for non-local boundaries</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Boundary Conditions for Curved Boundaries

<table>
<thead>
<tr>
<th>Bouzidi</th>
<th>non-local</th>
<th>first order</th>
<th>References [12]</th>
</tr>
</thead>
</table>

Data Structures

The basic data structure used by an application programmer is the `BlockLatticeXD`. Here, the placeholder \( x \) stands for the number 2 or 3, depending on whether a 2D or 3D lattice is instantiated. A generalization of the `BlockLatticeXD` are the `CuboidStructureXD` and the `SuperLatticeXD`, both of which have similar functionality but a slightly different scope. These advanced data structures generate a patchwork consisting of many `BlockLatticeXD` structures that are presented behind a unified interface. Applications of these structures are MPI-parallel and memory saving simulations that do not allocate memory in chosen subdomains of the numerical grid. For more information on the data structure design of OpenLB, we refer to [30].

Input / Output

The basic mechanism behind I/O operations in OpenLB is the serialization and unserialization of a `BlockLatticeXD`. This mechanism is used to save the state of a simulation, and to produce VTK output for data post-processing with external tools. In both cases, the data is saved in the binary Base64 format, which ensures compact and (relatively) platform-independent data storage.

1.3.5 Project Participants

The OpenLB project was initiated in 2006. Between 2006 and 2008 Jonas Latt was the project coordinator. As of 2009, Mathias J. Krause has been coordinating the project. Since 2006 the following persons have contributed source code to OpenLB:

**Armani Arfaoui**: core: performance improvements for D3Q19 BGK collision operator
**Saada Badie**: core: performance improvements for D3Q19 BGK collision operator

**Lukas Baron**: utilities: (parallel) console output, time and performance measurement, dynamics: porous media model, functors: concept, div. functors implementation

**Vojtech Cvrcek**: functors: 2D adaptation, dynamics: power law, examples: power law, updates

**Tim Dornieden**: functors: smooth start scaling, io: vti writer

**Jonas Fietz**: io: configure file parsing based on XML, octree STL reader interface to CVMLCPP (< release 0.9), communication: heuristic load balancer

**Benjamin Förster**: core: super data implementation io: new serializer and serializable implementation, vti writer, new vti reader, functors: new discrete indicator

**Max Gaedtke**: core: unit converter dynamics: thermal, examples: thermal

**Marc Haussmann (active)**: dynamics: turbulence modelling, examples: tgv3d, io: gnuplot heatmap

**Thomas Henn**: io: voxelizer interface based on STL, particles: particulate flows

**Fabian Klemens (active)**: functors: flux, indicator-based functors io: gnuplot interface

**Jonas Kratzke**: core: unit converter, io: GUI interface based on description files and OpenGPI, boundaries: Bouzidi boundary condition


**Adrian Kummerländer (active)**: functors: Lp-norm, flux, reduction, lattice indicators

**Jonas Latt**: core: basic block structure, communication: basic parallel block lattice approach (< release 0.9), io: vti writer, general: integration and maintenance of
added components (2006-2008), **boundaries**: basic boundary structure, **dynamics**: basic dynamics structure, **examples**: numerous examples, which have been further developed in recent years, **organization**: integration and maintenance of added components (2006-2008), project management (2006-2008)

**Marie-Luise Maier (active):** particles: particulate flows, frame change

**Orestis Malaspinas:** **boundaries**: alternative boundary conditions (Inamuro, Zou/He, Nonlinear FD), **dynamics**: alternative LB models (Entropic LB, MRT)

**Cyril Masquelier:** **functors**: indicator, smooth indicator

**Albert Mink (active):** **functors**: arithmetic, io: parallel VTK interface3, zLib compression for VTK data, GifWriter, **dynamics**: radiative transport, **boundary**: diffuse reflective boundary

**Markus Mohrhard (active):** **general**: makefile environment for parallel compilation, **organization**: integration and maintenance of added components (2018-)

**Patrick Nathen (active):** **dynamics**: turbulence modeling (advanced subgrid-scale models), **examples**: nozzle3d

**Aleksandra Pachalieva (active):** **dynamics**: thermal (MRT model), **examples**: thermal (MRT model)

**Bernd Stahl:** **communication**: 3D extension to MultiBlock structure for MPI parallelization (< release 0.9), **core**: parallel version of (scalar or tensor-valued) data fields (< release 0.9), **io**: VTK output of data (< release 0.9)

**Robin Trunk (active):** **dynamics**: parallel thermal, advection diffusion models

**Peter Weisbrod:** **dynamics**: parallel multi phase/component, **examples**: structure and showcases, phaseSeparationXd

**Gilles Zahnd:** **functors**: rotating frame functors

**Asher Zarth:** **core**: vector implementation

**Simon Zimny:** **io**: pre-processing: automated setting of boundary conditions
2 Using OpenLB for Applications

The general way of functioning in OpenLB follows a generic path. The following structure is maintained throughout every OpenLB application example, to provide an common structure and guide beginners.

1st Step: Initialization
The converter between physical and lattice units is set in this step. It is also defined, where the simulation data is stored and which lattice type is used.

2nd Step: Prepare geometry
The geometry is acquired, either from another file (a .stl file) or from defining indicator functions. Then, the mesh is created and initialized based on the given geometry. This consists of classifying voxels with material numbers, according to the kind of voxels they are: an inner voxel containing fluid ruled by the fluid dynamics will have a different number than a voxel on the inflow with conditions on its velocity. The function prepareGeometry is called for these tasks. Further, the mesh is distributed over the threads to establish good scaling properties.

3rd Step: Prepare lattice
According to the material numbers of the geometry, the lattice dynamics are set here. This step characterizes the collision model and boundary behavior. The choices depend on whether a force is acting or not, the use of single relaxation time (BGK) or multiple relaxation times (MRT), the simulation dimension (it can also be a 2D model), whether compressible or incompressible fluid is considered, and the number of neighbouring voxels chosen. By the creation of a computing grid, the SuperLattice, the allocation of the required data is done as well.

4th Step: Main loop with timer
The timer is initialized and started, then a loop over all time steps iT starts the simulation, during which the functions setBoundaryValues, collideAndStream and getResults (steps 5, 6 and 7 respectively) are called repeatedly until a maximum of iterations is reached, or the simulation has converged. At the end, the timer is stopped and the summary is printed to the console.
5th Step: Definition of initial and boundary conditions The first of the three important functions called during the loop, `setBoundaryValues`, sets the slowly increasing inflow boundary condition. Since the boundary is time dependent, this happens in the main loop. In some applications, the boundaries stay the same during the whole simulation and the function doesn’t need to do anything after the very first iteration.

6th Step: Collide and stream execution Another function `collideAndStream` is called each iteration step, to perform the collision and the streaming step. If more than one lattice is used, the function is called for each lattice separately.

7th Step: Computation and output of results At the end of each iteration step, the function `getResults` is called, which creates console output, `.ppm` files or `.vti` files of the results at certain timesteps. The ideal is to get the relevant simulation data with functors and thus facilitate the post processing significantly. By passing the converter and the time step, the frequency of writing or displaying data can be chosen easily. In many applications, the console output is required more often than the vtk data.

2.1 Lesson 1: Getting Started - Sketch of Application

This section presents example `bstep2d` which can be found in the recent release of OpenLB. This well known example simulates a flow over a backward-facing step and serves as an illustration of OpenLB and its features.

In order to execute the simulation and get some results, download and unpack OpenLB on a Linux system, see Section 3.1. Then, generate a executable file by compiling the program through the command `make`. Finally, launch the simulation by `./bstep2d` and observe the terminal output, see Section 8.5.

A few lines are invariably the same for all OpenLB applications, see Listing 2.1.

```cpp
#include "olb2D.h"
#ifndef OLB_PRECOMPILED // Unless precompiled version is used,
  #include "olb2D.hh" // include full template code
#endif

#include <vector> // Some C++ library for vectors
#include <cmath> // required to compute squares, sin, ...
#include <iostream> // to write to console
#include <fstream>
```
using namespace olb; // OpenLB namespaces
using namespace olb::descriptors; //

typedef double T;
#define DESCRIPTOR D2Q9Descriptor

Listing 2.1: Framework of an OpenLB program. Fundamental properties of the simulation are defined here.

Line 1: The header file olb2D.h includes definitions for the whole 2D code present in the release. In the same way, access to 3D code is obtained by including the file olb3D.h.

Line 3: Most OpenLB code depends on template parameters. Therefore, it cannot be compiled in advance, and needs to be integrated “as is” into your programs via the file olb2D.hh or olb3D.hh respectively. Including all this code slows down compilation (2D codes may take around 10 seconds to compile, and 3D codes around 30 seconds). If this overhead becomes too annoying during frequent development-compilation cycles, the code can be precompiled for the required data types. Although this topic is not covered in the tutorial, this short explanation should clarify the meaning of the the cryptic ifndef OLB_PRECOMPILED.

Line 11: All OpenLB code is contained in the namespace olb. The descriptors have an own namespace and define the lattice arrangement, e.g. $D_{2Q9}$ or $D_{3Q19}$.

Line 14: Choice of double precision floating point arithmetic. Any other floating point type can be used, including built-in types and user-defined types which are implemented through a C++ class.

Line 15: Choice of a lattice descriptor. Lattice descriptors specify not only which lattice you are going to use (for 2D simulations, the current OpenLB release gives you no choice but $D_{2Q9}$ anyway), but also potentially the nature of additional scalars, such as an external force field, for which memory needs to be allocated on a grid cell.

The next code presents a brief overview about the structure of an OpenLB application, see Listing 2.2. It aims rather to introduce and guidelines the beginners, than explain the classes and methods in depth. Details on the shown functions can be found in the source code, this means in the bstep2d.cpp file, as well as in the following chapters.
Figure 2.1: D2Q9

Figure 2.2: D3Q19
SuperGeometry2D<T> prepareGeometry(LBconverter<T> const & converter)
{
    // create Cuboids and assign them to threads
    // create SuperGeometry
    // set material numbers
    return superGeometry;
}

void prepareLattice(...)
{
    // set dynamics for fluid and boundary lattices
    // set initial values, rho and u
}

void setBoundaryValues(...)
{
    // set Poiseuille velocity profile at inflow
    // increase inflow velocity slowly over time
}

void getResults(...)
{
    // write simulation data do vtk files and terminal
}

int main(int argc, char* argv[])
{
    // === 1st Step: Initialization ===
    olbInit(&argc, &argv);
    // set output directory
    singleton::directories().setOutputDir("./tmp/");
    OstreamManager clout(std::cout, "main");
    // create converter
    UnitConverter<T, DESCRIPTOR> converter(
        // define discretization and characteristic parameter
    );
    converter.print(); // write conversion factors to console

    // === 2nd Step: Prepare Geometry ===
    SuperGeometry2D<T> superGeometry( prepareGeometry(converter) );

    // === 3rd Step: Prepare Lattice ===
    SuperLattice2D<T, DESCRIPTOR> sLattice(superGeometry);

    BGKdynamics<T, DESCRIPTOR> bulkDynamics(
        converter.getLatticeRelaxationFrequency(),
        instances::getBulkMomenta<T, DESCRIPTOR>()
    );
Listing 2.2: A brief overview of a typically OpenLB application, bstep2d. Details on the specific functions can be found in the following chapters.

2.2 Lesson 2: Understand the BlockLattice

This second lesson starts with a response to the scream of indignation you emitted in Lesson 1, when you learned that each cell of a BlockLatticeXD carries along its own Dynamics object, and collision is triggered by some dynamic run-time mechanism. How could the OpenLB developers favor object-oriented mumbojumbo over efficiency, right there in the core of the library?

The truth is that the overhead incurred by delegating collision to an object (instead of hard-coding collision somewhere inside the loop over grid nodes) is completely irrelevant. The efficiency loss is minimal on all platforms on which OpenLB has been tested so far, and it is negligible in face of other big-picture efficiency considerations.
One such consideration is about the separation of collision and streaming in Line 63 of Listing 2.2. The question to ask, instead of nitpicking over object-oriented vs. non-object-oriented issues, is whether it is really necessary to step through memory twice; once to execute collision and once to execute streaming. As a matter of fact, there are several ways of avoiding this time-consuming double access to memory, one of which is implemented in OpenLB and documented in Ref. [3]. For an OpenLB user, doing this is as easy as replacing the collision-streaming sequence by a call to the method `collideAndStream()`:

```java
// collision-streaming cycles
lattice.collide();
// lattice.stream(true);
lattice.collideAndStream(true);
```

Listing 2.3: Collision and streaming in one step for improved efficiency

Using the method `collideAndStream` is, of course, only possible when you don’t need to compute or modify anything between collision and streaming. When this is the case, the use of this method can however reduce by as much as 40% the execution time of your code, depending on your hardware.

The `BlockLattice2D<T, LATTICE>` is basically a nx-by-ny-by-q array of variables of type `T`. The following code for example is valid (although it is bad practice, as explained below):

```java
int nx, ny, someX, someY, someF;
// <...> some code to initialize nx, ny, someX and someY
BlockLattice2D<T, LATTICE> lattice(nx, ny); // instantiate
T value = lattice.get(someX, someY)[someF]; // read values
lattice.get(someX, someY)[someF] = 0.; // write values
```

Listing 2.4: Direct access to values in a BlockLattice2D

The method `BlockLattice2D<T, LATTICE>::get()` delivers an object of type `Cell<LATTICE>`, which contains storage space for the particle populations and, if required by the `LATTICE` template, for additional scalars. The `Cell` offers many methods to read and manipulate the data. You are much more likely to use those methods in practice, rather than accessing particle populations directly as in Listing 2.4:

```java
int nx, ny, someX, someY, someF;
// <...> some code to initialize nx, ny, someX and someY
BlockLattice2D<T, LATTICE> lattice(nx, ny); // instantiate
```
// <...> some code to initialize dynamics objects of the lattice

T velocity[2];
lattice.get(someX,someY).computeU(velocity); // compute velocity
velocity[0] = 0.;
lattice.get(someX,someY).defineU(velocity); // modify velocity

Listing 2.5: Manipulation of data through methods of a Cell

In this example, the method Cell<T>::computeU() computes the velocity on a cell for you, using its dynamics object. Conversely, the method Cell<T>::defineU() modifies the velocity by translating the particle populations into space of moments, modifying the moment of the velocity, and leaving the others as they are.

In addition to being more convenient, the access to the data in Listing 2.5 has a distinct advantage to the approach of Listing 2.4. In Listing 2.4, the data inside a Cell is accessed directly, whereas in Listing 2.5, it is accessed indirectly through the dynamics object of the cell. Although direct data access works in simple data structures, such as the present BlockLattice2D, only indirect data access can be used in complicated data structures. When the code is, for example, executed in parallel, you cannot access the data directly, because it might not be found on your processor. The dynamics object, on the other hand, is smart enough to locate the data on the right processor, and to instantiate MPI communication to access it.

Generally speaking, the methods of a Cell are separated into two groups, one for direct data access, and one for indirect data access through the dynamics object. When using OpenLB as an application programmer, it is strongly recommended that you only make use of methods in the second group, in order for your code to be extensible. Methods of the first group are used by programmers who wish to extend the OpenLB library, for example by writing a class to implement a new type of dynamics. Most of the subsequent lessons are written for application programmers, and the code is written with extensibility in mind, for example, by insisting on the possibility for it to be run in parallel with minimal changes.

The following list details some useful methods to access the data of a Cell indirectly through the dynamics object:

void iniEquilibrium(T rho, const T u[Lattice<T>::d]) Initialize all particle populations at an equilibrium distribution with density rho and velocity u.

T computeRho() const Compute the particle density on the cell.

void computeU(T u[Lattice<T>::d]) const Compute the velocity on the cell.
void computeStress ( T pi[util::TensorVal(Lattice(T))::n] ) const Compute the off-equilibrium stress-tensor $\Pi^{(1)}$ on the cell.

void computeExternalField(int pos, int size, T* ext) const Retrieve the external scalars and store them in a C-array.

void defineRho(T rho) Modify the populations such that the density yields $\rho$ and the other moments are unchanged.

void defineU(const T u[Lattice(T)::d]) Modify the populations such that the velocity yields $u$ and the other moments are unchanged.

void defineStress(const T pi[util::TensorVal(Lattice(T))::n]) Modify the populations such that the tensor $\Pi^{(1)}$ yields $\pi$ and the other moments are unchanged.

The discussion of this lesson is also valid for 3D lattices, which are instantiated with the following instruction:

```cpp
#define D3Q19Descriptor LATTICE
int nx, ny, nz;
// <...> initialization of nx, ny, nz
BlockLattice3D<T,LATTICE> lattice(nx,ny,nz);
```

Listing 2.6: Instantiation of a 3D lattice

The BlockLattice2D and the BlockLattice3D have different types, because they have distinct interfaces. The method get() for example requires 2 arguments in the 2D case and 3 arguments in 3D. The Cell class, an instance of which is delivered by the method get(), is however the same in 2D and 3D, although its template is instantiated with a different lattice descriptor (e.g. D2Q9Descriptor vs. D3Q19Descriptor). The above list of methods of the Cell is therefore valid in 3D as well.

2.3 Lesson 3: Define and Use Boundary Conditions

The current OpenLB release offers five different boundary conditions for the implementation of pressure and velocity boundaries. They support boundaries that are aligned with the numerical grid, and also implement proper corner nodes in 2D and 3D, and edge nodes that connect two plane boundaries in 3D. The choice of a boundary condition is conceptually separated from the definition of the location of boundary nodes. It is therefore possible to modify the choice of the boundary condition by
changing a single instruction in a program. This instruction is the instantiation of a
OnLatticeBoundaryCondition object:

```cpp
// Instantiate 2D boundary condition
OnLatticeBoundaryCondition2D<T,D2Q9Descriptor>* boundaryCondition2D = 
createLocalBoundaryCondition2D(lattice);

// Instantiate 3D boundary condition
OnLatticeBoundaryCondition2D<T,D3Q19Descriptor>* boundaryCondition3D = 
createLocalBoundaryCondition3D(lattice);
```

Listing 2.7: Instantiation of OnLatticeBoundaryCondition.

Objects of type OnLatticeBoundaryConditionXD are used to attribute the role of
boundary node to chosen nodes of the lattice. The following code configures a lattice in
such a way that the rectangle following the lattice boundaries implements a boundary
condition on the velocity.

```cpp
template<typename T>
void velocityBoundaryBox (BlockLattice2D<T,D2Q9Descriptor>& lattice, 
OnLatticeBoundaryCondition2D<T,D2Q9Descriptor>& bc, T omega) { 
  int nx = lattice.getNx();
  int ny = lattice.getNy();
  // top boundary
  bc.addVelocityBoundary1P(1,nx-2,ny-1,ny-1, omega);
  // bottom boundary
  bc.addVelocityBoundary1N(1,nx-2, 0, 0, omega);
  // left boundary
  bc.addVelocityBoundary0N(0,0, 1, ny-2, omega);
  // right boundary
  bc.addVelocityBoundary0P(nx-1,nx-1, 1, ny-2, omega);
  // Corner nodes
  bc.addExternalVelocityCornerNN(0,0, omega);
  bc.addExternalVelocityCornerNP(0,ny-1, omega);
  bc.addExternalVelocityCornerPN(nx-1,0, omega);
  bc.addExternalVelocityCornerPP(nx-1,ny-1, omega);
  // Make the lattice ready for simulation
  lattice.initialize();
}
```
Listing 2.8: Instantiation of velocity boundary condition along lattice boundaries.

When boundary nodes are instantiated, it is necessary to specify the orientation of the boundary through the normal vector that points outside of the domain. The instruction `addVelocityBoundary1P` refers to a boundary whose normal is in positive \(y\)-direction (\(P\) stands for “positive”, and indexes are numbered as \(0\) for the \(x\)-index and \(1\) for the \(y\)-index). For external corners, the expression \(NN\) refers to any boundary vector whose opposite direction points inside the numerical domain. In this case, this boundary vector points in negative \(x\)-direction and negative \(y\)-direction. The term `External` in the method `addExternalVelocityCornerNN` refers to the fact that the domain boundaries are convex shaped. Corners of concave shaped boundaries are instantiated with methods of the form `addInternalVelocityCornerXX`, where \(X\) stands again for \(N\) or \(P\) and indicates the direction of a vector pointing outside the numerical domain.

Pressure boundaries are instantiated just as easily by replacing the word `Velocity` by `Pressure` in the methods of the `OnLatticeBoundaryCondition` object.

Things are slightly more complicated in 3D, where edges also need separate treatment. Edges are locations where two boundary surfaces that are orthogonal to each other meet. The following are typical instructions one may use in the 3D case. In 3D, the instruction `addVelocityBoundary0N` instantiates a plane boundary domain in negative \(x\)-direction (a left boundary). It takes 6 arguments, in addition to the \(omega\)-argument, in order to delimit the plane like a sub-volume with one degenerate space direction. The instruction `addExternalVelocityEdge0NP` instantiates an edge whose outward-pointing normal vector is in the \(0\)-plane (in the plane in which \(x = 0\)) and which points in negative \(y\)- and positive \(z\)-direction. Counting of indexes is cyclic: the instruction `addExternalVelocityEdge1NP` refers to an edge with normal vector in the \(y = 0\)-plane and with negative \(z\)- and positive \(x\)-direction. The Edge instructions also take 6+1 arguments, because they treat the edge like a sub-volume with two degenerate directions. In 3D, there are external and internal corners, and there are external and internal edges.

Although setting up the geometry of the numerical domain can be somewhat bothersome, especially in 3D, this is a one-time job. Once the setup is completed, specifying the required velocity and density on boundaries is straightforward. This is done through a call to the method `defineVelocity` or `defineDensity` of the corresponding cell. You may remember from LESSON 2, that on normal lattice Boltzmann nodes, these methods modify the value of particle populations in order to obtain the required
velocity/density. On boundary nodes, the rules are different. Here, particle populations are not modified (which is necessary, as you may want to change the boundary velocity during a simulation, without tampering with the particle populations). On velocity boundaries, the method `defineVelocity` modifies the required velocity value for the boundary, whereas `defineDensity` has no effect. On pressure boundaries, the method `defineVelocity` has no effect and `defineDensity` picks out the required density value on the boundary. It should be pointed out that although the domain geometry is specified piece-wise (plane per plane, edge per edge, and corner per corner), the velocity/density can be adapted individually on every node. Furthermore, accessing parameters of the boundary on a per-cell base is convenient, because it does not require the programmer to distinguish any more between plane boundaries, edges or corners. Finally, the choice of the velocity/density value is not static: it can be altered at every time step to model time-dependent boundaries.

The following is a list of available boundary conditions. Instead of showing the actual class name of the boundary condition, the list indicates the names of functions that generate the boundary condition, as these are the ones you are likely to access as an end user. The $X$ is a placeholder for 2 respectively 3, as all boundary conditions are implemented in 2D and 3D.

**createLocalBoundaryConditionXD** This is the default local boundary condition. It implements a regularized boundary [33], which tends to be numerically stable in a last range of regimes.

**createInterpBoundaryConditionXD** This is the default non-local boundary condition. It is based on the algorithm proposed by Skordos [46], and uses a finite difference scheme over adjacent neighbors to evaluate velocity gradients.

**createZouHeBoundaryConditionXD** The local boundary condition introduced by Zou and He [52]. It is very accurate, especially in 2D simulations, but can have stability issues.

**createInamuroBoundaryConditionXD** The local boundary condition by Inamuro Inamuro et al. [28]. It is very accurate in 2D and 3D, but can have stability issues. In 3D, it is slower than other boundary conditions, because it solves an implicit equation at every time step.

**createExtendedFdBoundaryConditionXD** The approach is the same as in the boundary condition generated by `createInterpBoundaryConditionXD`, but this time, non-linear velocity terms of the Chapman-Enksog expansion are taken into
account. This is rarely useful, but can make a difference in a very low Mach-
number regime.

It should be clear by now, how powerful the abstraction mechanism of the “OnLat-
ticeBoundaryConditionXD” objects is. With the help of this mechanism, one can treat
local and non-local boundary conditions the same way. Furthermore, they can be used
both for sequential and parallel program execution, as it is shown in Lesson 10. The
mechanism behind this is explained in Lesson 7. The bottom line is that both local and
non-local boundary conditions instantiate a special dynamics object and assign it to
boundary cells. Non-local boundaries additionally instantiate post-processing objects
which take care of non-local aspects of the algorithm.

This mechanism for the instantiation of boundary conditions is generic and easy to
use, but it makes sense only in quite regular geometries. In irregular geometries, even
if you agree on using a staircase approximation of domain boundaries, you will expe-
rience a hard time attributing the right boundary type to each cell. Although off-lattice
boundaries are under investigation in the OpenLB project, they are not currently avail-
able. If your irregular domain boundaries implement a no-slip condition, your current
best bet is to implement them through a fullway bounce-back dynamics. In this ap-
proach, particle populations that are opposite to each other are swapped at each iter-
ation step, and no additional collision is executed. The advantage of this procedure is
that it is independent of the orientation of the domain. The following code implements
for example a circular obstacle with no-slip walls in the center of a 2D domain:

```c++
<int> definition of the types T and DESCRIPOR

BlockLattice2D<T,DESCRIPTOR> lattice(nx,ny);

for (int iX=0; iX<nx; ++iX) {
    for (int iY=0; iY<ny; ++iY) {
        if ((iX-nx/2)*(iX-nx/2) + (iY-ny/2)*(iY-ny/2) < r*r) {
            lattice.defineDynamics(iX,iX,iY,iY,
                          &instances::getBounceBack<T,D2Q9Descriptor>() );
        }
    }
}
```

Listing 2.9: Implementation of a bounce-back cylinder in the domain center
2.4 Lesson 4: UnitConverter - Lattice and Physical Units

Fluid flow problems are usually given in a system of metric units. For example consider a cylinder of diameter 3 cm in a fluid channel with average inflow velocity of 4 m. The fluid has a kinematic viscosity of 0.001 m$^2$ s$^{-1}$. The value of interest is the pressure difference measured in Pa at the front and the back of the cylinder (with respect to the flow direction). However, the variables used in a LB simulation live in a system of lattice units, in which the distance between two lattice cells and the time interval between two iteration steps are chosen to be unity. Therefore, when setting up a simulation, a conversion directive has to be defined that takes care of scaling variables from physical units into lattice units and vice versa.

In OpenLB, all these conversions are handled by a class called UnitConverter, see Listing 2.10. An instance of the UnitConverter is generated with desired discretization parameters and reference values in SI units. It provides a set of conversion functions to enable a fast and easy way to convert between physical and lattice units. In addition, it gives information about the parameters of the fluid flow simulation, such as the Reynolds number or the relaxation parameter $\omega$.

Let’s have a closer look at the input parameters: The reference values represent characteristic quantities of the fluid flow problem. In this example, it is suitable to choose the cylinder’s diameter as characteristic length and the average inflow speed as characteristic velocity. Furthermore, two discretization parameters, namely the grid size $\Delta x$ in m and time step size $\Delta t$ in s are provided to the converter. From these reference values and discretization parameters, all the conversion factors and the relaxation time $\tau$ are calculated.

Due to the fact, that there are stability bounds for the relaxation time and the maximum occurring lattice velocity, one does not usually chose $\Delta x$ and $\Delta t$, but sets stable and accurate values for any two out of resolution, relaxation time or characteristic (maximum) lattice velocity. To make that easily available for the user of openLB, there are different constructors for the UnitConverter class: UnitConverterFromRelaxationTimeAndLatticeVelocity, UnitConverterFromResolutionAndLatticeVelocity and UnitConverterFromResolutionAndRelaxationTime.

Once the converter is initialized, its methods can be used to convert various quantities such as velocity, time, force or pressure. The function for the latter helps us to evaluate the pressure drop in our example problem, as shown in the the following code snippet:

```
UnitConverterFromResolutionAndRelaxationTime<T, DESCRIPTOR> const
converter (
Listing 2.10: Use of UnitConverter in a 3D problem.

Line 1-7: Instantiate an UnitConverter object and specify discretization parameters as well as characteristic values.

Line 9: Write simulation parameters and conversion factors to terminal.

Line 11: Write simulation parameters and conversion factors in a logfile.

Line 13 and 14: The conversion from physical units (seconds) to discrete ones (time steps) is managed by the converter.

Line 17-20: The converter automatically calculates the pressure values from the local density.
2.5 Lesson 5: Extract Data From aSimulation

When the collision step is executed, the value of the density and the velocity are computed internally, in order to evaluate the equilibrium distribution. Those macroscopic variables are however interesting for the OpenLB end-user as well, and it would be a shame to simply neglect their value after use. These values are accessed through the method `getStatistics()` of a `BlockLattice`:

\[ \text{T lattice.getStatistics().getAverageRho()} \] Returns average density evaluated during the previous collision step.

\[ \text{T lattice.getStatistics().getAverageEnergy()} \] Returns half the average velocity norm evaluated during the previous collision step.

\[ \text{T lattice.getStatistics().getMaxU()} \] Returns maximum value of the velocity norm evaluated during the previous collision step.

Often, the information provided by the statistics of a lattice in not sufficient, and more generally numerical result are required. To do this, you can get data cell-by-cell from the `BlockLatticeXD` and `SuperLatticeXD` through functors, see Chapter 10. Functors act on the underlying lattice and process its data to relevant macroscopic units, e.g. density, velocity, stress, flux, pressure and drag. Functors provide an `operator()` that instead of access stored data, computes every time it is called the data. Since OpenLB version 0.8, the concept of functors unfold not only for postprocessing, but also for boundary conditions and the generation of geometry, see Chapter 10. In Listing 2.11 it is shown, how to extract data out of a `SuperLattice3D` named `sLattice` and an `SuperGeometry3D` named `sGeometry`. The data format is a legal `vtk` file, that can be processed further with `ParaView`.

```cpp
// generate the writer object
SuperVTMWriter3D<T> vtmWriter("bstep3d");
// write every 0.2 seconds
if (iT==converter.getLatticeTime(0.2)) {
  // create functors
  SuperLatticeGeometry3D<T, DESCRIPTOR> geometry(sLattice, sGeometry);
  SuperLatticeCuboid3D<T, DESCRIPTOR> cuboid(sLattice);
  SuperLatticeRank3D<T, DESCRIPTOR> rank(sLattice);
  // write functors to file system, vtk format
  vtmWriter.write(geometry);
  vtmWriter.write(cuboid);
  vtmWriter.write(rank);
}
```

29
Listing 2.11: Extract simulation data to \texttt{vtk} file format.

As before mentioned, OpenLB provides functors for a bunch of data, see Listing 2.12. More details about writing simulation data can be found in Chapter 8.

```cpp
// Create the functors by only passing lattice and converter
SuperLatticePhysVelocity3D<T, DESCRIPTOR> velocity(sLattice, converter);
SuperLatticePhysPressure3D<T, DESCRIPTOR> pressure(sLattice, converter);
// Create functor that corresponds to material numbers
SuperLatticeGeometry3D<T, DESCRIPTOR> geometry(sLattice, superGeometry);
```

Listing 2.12: Code example for creating velocity, pressure and geometry functors.

The most straightforward and convenient way of visualizing simulation data is to produce a 2D snapshot of a scalar valued functor. This is done through the \texttt{BlockReduction3D2D}, which puts a plane into arbitrary 3D functors. Afterwards, this plane can be easily written to a image file. OpenLB creates images of format PPM as shown in Listing 2.13.

```cpp
// velocity is an application: R^3 \to R^3
// an image in its very basic sense is an application: R^2 \to R
// transformation of data is presented below

// get velocity functor
SuperLatticePhysVelocity3D<T, DESCRIPTOR> velocity(sLattice, converter);
// get scalar valued functor by applying the point wise l2 norm
SuperEuklidNorm3D<T, DESCRIPTOR> normVel( velocity );
// put a plane with normal (0,0,1) in the 3 dimensional data
BlockReduction3D2D<T> planeReduction( normVel, {0, 0, 1} );
BlockGifWriter<T> gifWriter;
// write ppm image to file system
gifWriter.write( planeReduction, iT, "vel" );
```

Listing 2.13: Create a PPM image out of a 3D velocity functor.

This image writer provides in situ visualization which, in contrast to the \texttt{vtk} writer, produces smaller data sets that can be interpreted immediately without requiring other software.
2.6 Lesson 6: Convergence Check

The class `ValueTracer` checks for time-convergence of a given scalar $\phi$. The convergence is reached when the standard deviation $\sigma$ of the monitored value $\phi$ is smaller than a given residuum $\epsilon$ times the expected value $\bar{\phi}$.

\[
\sigma(\phi) = \sqrt{\frac{1}{N+1} \sum_{i=0}^{N} (\phi_i - \bar{\phi})^2} < \epsilon \bar{\phi}
\]

(2.1)

The expected value $\phi$ is the average over the last $N$ time steps with $\phi_i := \phi(t \ast -i\Delta t)$ and time steps $\Delta t$.

\[
\bar{\phi} = \frac{1}{N+1} \sum_{i=0}^{N} \phi_i
\]

(2.2)

$N$ should be chosen as a problem specific time period. As an example $\text{charT} = \text{charL}/\text{charU}$ and $N = \text{converter.getLatticeTime}(\text{charT})$. To initialize a `ValueTracer` object use:

```cpp
util::ValueTracer<T> converge( numberTimeSteps, residuum );
```

Listing 2.14: Create a PPM image out of a 3D velocity functor.

For example, to check for convergence with a residuum of $\epsilon = 10^{-5}$ every physical second:

```cpp
util::ValueTracer<T> converge( converter.getLatticeTime(1.0), 1e-5 );
```

Listing 2.15: Create a PPM image out of a 3D velocity functor.

There for it is needed to pass the monitored value to the `ValueTracer` object every time steps by:

```cpp
for (iT = 0; iT < maxIter; ++iT) {
    ...
    converge.takeValue( monitoredValue, isVerbose );
    ...
}
```

Listing 2.16: Create a PPM image out of a 3D velocity functor.

If you like to print average value and its standard derivation every number of time steps choosen during initialization set `isVerbose` to true otherwise choose false. It is good idea to choose average energy as monitored value:
Listing 2.17: Create a PPM image out of a 3D velocity functor.

Do something like the following in the timeloop:

```cpp
if (converge.hasConverged()) {
    clout << "Simulation converged." << endl;
    break;
}
```

Listing 2.18: Create a PPM image out of a 3D velocity functor.

### 2.7 Lesson 7: Use an External Force

In simulations, the dynamics of a fluid is often driven by a force field (gravity, intermolecular interaction, etc.) which is space- and time-dependent, and which is possibly computed from an external source, independent of the LB simulation. In order to optimize memory access and to minimize cache-misses, the value of this force can be stored in a cell, adjacent to the particle populations. This is achieved by specifying external scalars in the lattice descriptor (see also Lesson 7). OpenLB offers, by default, the two descriptors `ForcedD2Q9Descriptor` and `ForcedD3Q19Descriptor`. The dynamics `ForcedBGKdynamics` accesses the force term defined by these descriptors, and implements a LB dynamics with body force. The algorithm is taken from Ref. [23] to guarantee second-order accuracy even when the force field is space and time dependent. An example for the implementation of a LB simulation with force term is found in the code examples/poiseuille2d/forced.

### 2.8 Lesson 8: Understand Genericity in OpenLB

OpenLB is a framework for the implementation of lattice Boltzmann algorithms. Although most of the code shipped with the distribution is about fluid dynamics, it is open to various types of physical models. Generally speaking, a model which makes use of OpenLB must be formulated in terms of the “local collision followed by nearest-neighbor streaming” philosophy. A current restriction to OpenLB is that the streaming step can only include nearest neighbors: there is no possibility to include larger neighborhoods within the modular framework of the library, i.e. without tampering with
OpenLB source code. Except for this restriction, one is completely free to define the topology of the neighborhood of cells, to implement an arbitrary local collision step, and to add non-local corrections for the implementation of, say, a boundary condition.

To reach this level of genericity, OpenLB distinguishes between non-modifiable core components, which you’ll always use as they are, and modular extensions. As far as these extensions are concerned, you have the choice to use default implementations that are part of OpenLB or to write your own. As a scientific developer, concentrating on these, usually quite short, extensions means that you can concentrate on the physics of your model instead of technical implementation details. By respecting this concept of modularity, you can automatically take advantage of all structural additions to OpenLB. In the current release, the most important addition is parallelism: you can run your code in parallel without (or almost without) having to care about parallelism and MPI.

The most important non-modifiable components are the lattice and the cell. You can configure their behavior, but you are not expected to write a new class which inherits from or replaces the lattice or the cell. Lattices are offered in different flavours, most of which inherit from a common interface \texttt{BlockStructureXD}. The most common lattice is the regular \texttt{BlockLatticeXD}, which is replaced by the \texttt{SuperLatticeXD} for parallel applications and for memory-saving applications when faced with irregular domain boundaries. An alternative choice for parallelism and memory savings is the \texttt{CuboidStructureXD}, which does not inherit from \texttt{BlockStructureXD}, but instead allows for more general constructs.

The modular extensions are classes that customize the behavior of core-components. An important extension of this kind is the lattice descriptor. This specifies the number of particle populations contained in a cell, and defines the lattice constants and lattice velocities, which are used to specify the neighborhood relation between a cell and its nearest neighbors. The lattice descriptor can also be used to require additional allocation of memory on a cell for external scalars, such as a force field. The integration of a lattice descriptor in a lattice happens via a template mechanism of C++. This mechanism takes place statically, i.e. before program execution, and avoids the potential efficiency loss of a dynamic, object-oriented approach. Furthermore, template specialization is used to optimize the OpenLB code specifically for some types of lattices. Because of the template-based approach, a lattice descriptor needs not inherit from some interface. Instead, you are free to simply implement a new class, inspired from the default descriptors in the files \texttt{core/latticeDescriptors.h} and \texttt{core/latticeDescriptor.hh}.
The dynamics executed by a cell are implemented through a mechanism of dynamic (run-time) genericity. In this way, the dynamics can be different from one cell to another, and can change during program execution. There are two mechanisms of this type in OpenLB, one to implement local dynamics, and one for non-local dynamics. To implement local dynamics, one needs to write a new class which inherits the interface of the abstract class `Dynamics`. The purpose of this class is to specify the nature of the collision step, as well as other important information (for example, how to compute the velocity moments on a cell). For non-local dynamics, a so-called post-processor needs to be implemented and integrated into a `BlockLatticeXD` through a call to the method `addPostProcessorXD`. This terminology can be somewhat confusing, because the term “post-processing” is used in the CFD community in the context of data analysis at the end of a simulation. In OpenLB, a post-processor is an operator which is applied to the lattice after each streaming step. Thus, the time-evolution of an OpenLB lattice consists of three steps: (1) local collision, (2) nearest-neighbor streaming, and (3) non-local postprocessing. Implementing the dynamics of a cell through a postprocessor is usually less efficient than when the mechanism of the `Dynamics` classes is used. It is therefore important to respect the spirit of the lattice Boltzmann method and to express the collision as a local operation whenever possible.

2.9 Lesson 9: Use Checkpointing for Long Duration Simulations

All types of data in OpenLB can be stored in a file or loaded from a file. This includes the data of a `BlockLatticeXD` and the data of a `ScalarFieldXD` or a `TensorFieldXD`. All these classes implement the interface `Serializable<T>`. This guarantees that they can transform their content into a data stream of type `T`, or read from such a stream. Serialization and unserialization of data is mainly used for file access, but it can be applied to different aims, such as copying data between two objects of different type. The data is stored in the ascii-based binary format `Base64`. Although `Base64`-encoded data requires 25% more storage space than when a pure binary format is used, this approach was chosen in OpenLB to enhance compatibility of the code between platforms. Saving and loading data is invoked by calling the `save` and `load` method on the object to be serialized. These methods take the filename as an optional (but recommended) argument, as shown below:

```java
int nx, ny;
<...> initialization of nx and ny
```
BlockLattice2D\(<T,\ DESCRIPTOR>\) lattice(nx, ny);
// load data from a previous simulation
lattice.load("simulation.checkpoint");
<...> run the simulation
// save data for security, to be able to take up
// the simulation at this point later
lattice.save("simulation.checkpoint");

Listing 2.19: Store and load the state of the simulation.

Checkpointing is also illustrated in the example programs bstep2D and bstep3D (Section 12.3).

2.10 Lesson 10: Save Memory when Domain Boundaries are Irregular

It is possible in OpenLB to allocate several lattices of type BlockLatticeXD and hide them behind a common interface, to treat them as the components of a larger lattice. This technique can be used to achieve parallelism, as it is described in the next lesson. Another application is the creation of lattices in which memory is allocated in selected subdomains only. This is useful for the simulation of flows with complicated domain boundaries, as no memory needs to be allocated outside the domain. An example program for this technique is under development, but is not yet available in the current release.

2.11 Lesson 11: Run Your Programs on a Parallel Machine

OpenLB programs can be executed on a parallel machine with distributed memory, based on MPI. To compile an OpenLB program for parallel execution, modify the file named config.mk, found in the OpenLB root directory, by removing the hashes before the lines: #CXX := mpic++, and #PARALLEL_MODE := MPI. The modified lines are shown in Listing 2.20. Execute make clean and make cleanbuild within the desired program directory to eliminate previously compiled libraries, and recompile the program by executing the make command. To run the program in parallel, use the command mpirun -np 2 ./cavity2d. Here -np 2 specifies the number of processors to be used.

#CXX := g++
#CXX := icpc -D__aligned__=ignored
#CXX := mpiCC
CXX := mpic++

5 ...

#PARALLEL_MODE := OFF
PARALLEL_MODE := MPI
#PARALLEL_MODE := OMP
#PARALLEL_MODE := HYBRID

Listing 2.20: Edited config.mk for MPI-parallel programs.
3 Install Dependencies

OpenLB is developed for high performance computing and hence, for Linux based operation systems. As a natural choice, the programming happens mainly on Linux systems which is as well the recommended platform to run applications.

In recent years, virtualbox turned into a powerful alternative to run a Linux operation system within Windows. In case you are a windows user, consider to run a Linux distribution in a virtual machine, e.g. virtualbox.

3.1 Linux

The developer are very proud to announce that OpenLB has no dependencies and requires only a C++ compiler alongside an OpenMPI implementation. Both can be installed on a Ubuntu system through

```
sudo apt-get install g++ openmpi-bin openmpi-doc libopenmpi-dev
```

ParaView is often used for visualization and can be installed from the Ubuntu packages as well

```
sudo apt-get install paraview
```

The very recent version has to be downloaded from the web. ParaView is an application built on top of the Visualization Tool Kit (VTK) libraries which can read VTK-files written by OpenLB.

To compile the software library and all examples, go into the root folder of OpenLB and type

```
make -j4 samples
```

If your system is set up correctly, you should see a lot compiler messages but no errors.

3.2 Mac

A working C++ and OpenMPI compiler enables you to compile and run OpenLB on Mac.
3.3 Windows

An installation guide for Windows using Cygwin can be found in the technical report TR4: Installing OpenLB in Windows/Cygwin [4]. An alternative method for running Linux on Windows through the Ubuntu Bash can be found in the technical report [5].
4 Non-dimensionalisation and Choice of Simulation Parameters

Basically, to describe a physical quantity you need a reference scale. By dividing a quantity by a variable of the same dimension, a dimensionless quantity is derived. The result is a number, which is called "the lattice value of the quantity" or the value of the amount "in lattice units", if you consider the Lattice Boltzmann Method. The reference scale is named conversion factor between two reference system. Furthermore, the conversion factor for quantity \( \phi \) is called \( C_\phi \) and the non-dimensional quantity gets the symbol \( \phi^* \).

To get into this topic, the book "The Lattice Boltzmann Method" [31], especially Chapter 7.1 and 7.2 therein, and https://en.wikipedia.org/wiki/Dimensional_analysis are strongly recommended.
5 Geometry

This chapter presents how geometry data can be loaded in or created by OpenLB. Furthermore, it is shown the concept of material numbers.

5.1 Creating a Geometry

OpenLB provides an interface for STL based geometry data and generates fully automated a regular mesh. On the other hand, geometries can be build out of geometric primitives such as cuboids, spheres and cylinders. By the implemented arithmetic that includes intersection, union and complement those primitives can be assembled very generally.

A computational domain such as the SuperLattice3D is created in 6 simple steps (see also Fig [5.1]):

1. Step: Create an Indicator3D instance by
   1. Reading an STL-file, see example aorta3d [12.1]
   2. Pre-defined geometric primitives and its combinations (+, −, ·), see example venturi3d [12.14]

2. Step: Construct CuboidGeometry3D. During construction cuboids will be automatically removed, shrunk and weighted for a good load balance.

3. Step: Construct LoadBalancer that assigns cuboids to threads.

4. Step: Construct SuperGeometry3D that links material numbers to voxels.

5. Step: Set material numbers to define dynamics for fluid and boundary.


// 1. Step: Create Indicator
STLreader< T > stlreader ("filename.stl", voxelSize);
IndicatorCuboid3D< T > cylinderInFlow( extend, origin );
CuboidGeometry3D<T> cuboidGeometry(indicator, voxelSize, noOfCuboids);

HeuristicLoadBalancer<T> loadBalancer(cuboidGeometry);

SuperGeometry3D<T> superGeometry(cuboidGeometry, loadBalancer);

// 5. Step: Set material numbers.
// set material number 2 for whole geometry
superGeometry.rename(0, 2, geometryIndicator);
// change material number from 2 to 1 for inner (fluid) cells, so
// that only boundary cells have material number 2
superGeometry.rename(2, 1, 1, 1, 1);
// or simply use an indicator that changes its lattices to one
superGeometry.rename(2, 1, fluidIndicator);
// additional material numbers for other boundary conditions
superGeometry.rename(2, 3, 1, cylinderInFlow);
superGeometry.rename(2, 4, 1, outflowIndicator0);
superGeometry.rename(2, 5, 1, outflowIndicator1);

SuperLattice3D<T, DESCRIPTOR> sLattice(superGeometry);

Listing 5.1: Create geometry based on STL or geometric primitives. All six steps are
presented briefly as source code.

The powerful application of the geometry generation of OpenLB can be demonstrated on
the example aorta3d. This example is based on a very complex geometry
and illustrated the highly user friendly and automated process from STL to computa-
tion grid the SuperLattice3D, see Figure 5.1.

5.2 Setting Material Numbers

OpenLB has a general concept for representation of a geometry. A specific number
called the material number is assigned to each cell, defining whether that cell lies on
the boundary or in the fluid domain or whether it is superfluous in the computations.
Figure 5.2 illustrates this using the example of an channel flow with an obstacle. The
different collision and stream steps on the boundary and the fluid are defined with re-
spect to the material number. The benefit of using material numbers in flow simulations
is the automatic determination of fluid directions on boundary nodes, as this is not al-
ways practical by hand e.g. if material numbers of a complex geometry are obtained
from a stl file.
Figure 5.1: Six steps to create a Geometry. It starts by reading an stl file with the help of an STLreader and end with the creation of a SuperLattice3D.

Besides creating the domain, IndicatorFXD functions can be used to set material numbers with the help of one of the rename functions in SuperGeometryXD.

```cpp
/// replace one material with another  
void rename(int fromM, int toM);
/// replace one material that fulfills an indicator functor condition with another
void rename(int fromM, int toM, IndicatorF3D<bool, T>& condition);
/// replace one material with another respecting an offset (overlap)
void rename(int fromM, int toM, unsigned offsetX, unsigned offsetY, unsigned offsetZ);
/// renames all voxels of material fromM to toM if the number of voxels given by testDirection is of material testM
void rename(int fromM, int toM, int testM, std::vector<int> testDirection);
/// renames all boundary voxels of material fromBcMat to toBcMat if two neighbour voxels in the direction of the discrete normal are fluid voxels with material fluidM in the region where the indicator function is fulfilled
void rename(int fromBcMat, int toBcMat, int fluidMat, IndicatorF3D<bool, T>& condition);
```

Listing 5.2: Different rename functions to set material numbers.
Figure 5.2: Lattice nodes of the geometry are associated to material numbers. Material number zero, one, two, three, for and five correspond to out of geometry, fluid, bounce back boundary, inflow, outflow and obstacle lattices, (1=fluid, 2=no-slip boundary, 3=velocity boundary, 4=constant pressure boundary, 5=curved boundary, 0=do nothing).

5.3 Building Geometry by Geometric Primitives

OpenLB provides several functors (see Section 10) for the creation of basic geometric primitives such as cuboids, circles, spheres, cones etc. These can be combined using the mathematical operators (+ union, − set difference, ∩ intersection) to create more complex domains. A demonstration of this can be found in the example venturi3D (see Section 12.14).

5.4 Excursion: Creating STL-files

The general process chain assumes that the geometry is already given in form of an stl file, if not created by the IndicatorFXD-functions. Simple geometries can be created using a CAD tool like FreeCAD [6]. An introduction to modeling with FreeCAD can be found for example in [http://www.youtube.com/watch?v=6RxHCR7TLtI](http://www.youtube.com/watch?v=6RxHCR7TLtI). The general procedure is mostly similar to the following description.

Firstly, a 2D drawing is created on a selected plane (e.g. the xy plane) using circles and polygons. In the next step a “height” is assigned to it in the third dimension. Several such 3d objects can be combined using operations like union, cut, intersection, rotation, trace, etc. to obtain the target geometry. Creating a square and a circle for the example cylinder3d in Figure 5.3 is not very difficult, the more complex geometry of a formula one car, however, can be a challenging and time consuming task.
Figure 5.3: The geometry for the example `cylinder3d` from Section 12.5 opened in FreeCAD.
6 Lattice Boltzmann Models and Core Data Structures

6.1 Concept - Data Organization

6.1.1 Cell - BlockLattice - SuperLattice

LBM, in its most widely accepted formulation, is executed on a regular, homogeneous lattice \( \Omega_h \) with equal grid spacing \( h \in \mathbb{R}_{>0} \) in all directions. When numerical constraints require that a given problem is solved on an inhomogeneous grid, it is common to adopt a so-called multi-block approach: the computational domain is partitioned into sub-grids with different levels of resolution, and the interface between those sub-grids is handled appropriately. This approach appears to respect the spirit of LBM well and leads to implementations that are both elegant and efficient, since the execution on a set of regular blocks is relatively fast compared to an unstructured grid representation of the whole geometry. For complex domains, a multi-block approach provides another advantage. A given domain can be represented by a certain number of regular blocks, which leads to cheap executions times on the one hand, and to a sparse memory consumption on the other hand. Furthermore, it encourages a particularly efficient form of data parallelism, in which an array is cut into regular pieces and distributed over the nodes of a parallel machine. As a result, LB applications can even be run on large parallel machines with a particularly satisfying gain of speed.

The same spirit is adopted in the OpenLB package, see Figure 6.1. The very basic data structure is called \texttt{Cell} and is orchestrated by a \texttt{BlockLattice}, which represents a regular array of \texttt{Cells}. In each \texttt{Cell}, the \( q \) variables for the storage of the discrete velocity distribution functions \( f_i \) (\( i = 0, 1, \ldots, q - 1 \)) are contiguous in memory. Note, in terms of optimization the cells hold

\[
\text{Cell}[iPop] = f_{iPop} - t_{iPop},
\]

where \( t_{iPop} \) corresponds to the lattice weights. This data structure is encapsulated by
Figure 6.1: Data structures in OpenLB: A number of BlockLattices build a SuperLattice to adopt higher level software constructs like multi-block, grid refined lattices and parallelised lattices.

a higher level, object-oriented layer. The purpose of this layer is to handle groups of BlockLattices, and to build higher level software constructs in a transparent way. Those constructs are called SuperLattice.

6.1.2 Descriptor

6.1.3 Dynamics

The core of OpenLB consists of a simple and efficient array-like construct called a BlockLattice. This object executes an LB algorithm in a very traditional sense, i.e. the lattice Boltzmann equation is split into two equations, namely the collision step:

\[
\tilde{f}_i^h(t,\vec{r}) = f_i^h(t,\vec{r}) - \frac{1}{3\nu + 1/2} \left( f_i^h(t,\vec{r}) - M_{eq}^{f_i^h}(t,\vec{r}) \right) \quad \text{in } I_h \times \Omega_h \times Q \quad (6.1)
\]

and the streaming step (propagation step):

\[
 f_i^h(t + h^2,\vec{r} + h^2\vec{v}_i) = \tilde{f}_i^h(t,\vec{r}) \quad \text{in } I_h \times \Omega_h \times Q . \quad (6.2)
\]

All Cells of the BlockLattice are iteratively parsed, and a local collision step is executed, followed by a non-local streaming step. The streaming step is independent of the choice of lattice Boltzmann dynamics and remains invariant. On the other hand, the collision step determines the physics of the model and can be configured by the user, by attributing a fully configurable dynamics object to each Cell. In this way, it is easy to implement inhomogeneous fluids which use a different type of physics from one
Cell to another. For each time step the collision and streaming step can be executed separately in two loops over all Cells or only in one. Both versions are implemented in OpenLB. Yet, for many applications the method fusing the two loops is preferable.

Although this concept of a BlockLattice is neat and should please the programmer by being conceptually close to the theory of LBM, it is not sufficiently general to address all possible issues arising in real life. As a case in point, some boundary conditions are non-local and need to access neighbouring nodes. Therefore, their implementation does not fit into the framework of a BlockLattice explained previously. The philosophy of OpenLB takes for granted that such situations, although they arise, take place in spatially confined areas only, for example the domain boundaries. They may therefore be implemented by slightly less efficient means, without spoiling the overall efficiency of the code. Their execution is taken care of by a post-processing step, which, instead of stepping over the whole lattice a second time, parses selected Cells only.

6.2 Classic BGK Model

6.3 MRT Model

6.4 Porous Media Model

The permeability parameter $K$ is a physical parameter that describes the macroscopic drag in a porous media model. For laminar flows it is defined by Darcy’s law:

$$K = -\frac{Q \mu L}{\Delta P},$$

where $Q = U A$ is the flow rate, $U$ a characteristical velocity, $A$ a cross-sectional area, $\mu$ the dynamic viscosity, $L$ a characteristical length, $\Delta P$ the pressure difference in between starting point and endpoint of the volume.

The porosity-value $d \in [0, 1]$ is a lattice-dependent value, $d = 0$ means the medium is solid and $d = 1$ denotes a liquid. According to Brinkman [14, 15], Borrvall and Petersson [11] and Pingon et al. [42], the Navier-Stokes equation is transformed (see Dornieden [21] and Stasius [47]). The discrete formulation of $d$ describes a flow region by its permeability:

$$d = 1 - h^{\text{dim}-1} \frac{\nu_{LB \tau_{LB}}}{K},$$

$\tau_{LB}$ is the relaxation time, $\nu_{LB}$ is the discrete kinematic viscosity and $h$ is the length. Therefore $K \in [\nu_{LB \tau_{LB}} h^{\text{dim}-1}, \infty]$. To describe the porosity or permeability of a medium,
a descriptor for porosity must be used, such as:

```c
#define DESCRIPTOR PorousD3Q19Descriptor
```

Be aware that the porous media model only works in the generic compilation mode. In the function `prepareLattice`, dynamics for the corresponding number of the porous material are defined for example as follows:

```c
void prepareLattice(..., Dynamics<T, DESCRIPTOR>& porousDynamics, ...
{...}

// Material=3 --> porous material
sLattice.defineDynamics(superGeometry, 3, &porousDynamics);
```

In function `setBoundaryValues`, the initial porosity value and external field is defined:

```c
void setBoundaryValues(..., T physPermeability, int dim, ...){
  // d in [0,1] is a lattice-dependent porosity-value depending on physical permeability
  T d = converter->latticePorosity(physPermeability);

  AnalyticalConst3D<T, T> porosity(d);

  sLattice.defineExternalField(superGeometry, 3,
    DESCRIPTOR<T>::ExternalField::porosityIsAt, 1, porosity);
...}
```

In the main function, the required parameters as well as the porous media dynamics are defined:

```c
int main(int argc, char* argv[]) {
...
  T physPermeability = 0.0003;
...
  PorousBGKdynamics<T, DESCRIPTOR> porousDynamics(converter->getOmega()
    ,
    instances::getBulkMomenta<T, DESCRIPTOR>());
...}
```

Additionally, the `UnitConverter` class in `src/core/units.h` provides useful functions for conversion between physical and lattice values:

```c
    /// converts a physical permeability K to a lattice-dependent porosity d
    porosity d
    /// (a velocity scaling factor depending on Maxwellian distribution
```
6.5 Power Law Model

The two most common deviations from Newton’s Law observed in real systems are pseudo-plastic fluids and dilatant fluids. By pseudo-plastic fluids the viscosity of the system decreases as the shear rate is increased. On the other hand, as the shear rate by dilatant fluids is increased, the viscosity of the system also increases. The simplest model, that describes this two type of deviations, was proposed by de Waele and Ostwald and is called the Power Law model that is defined by the viscosity as

\[ \mu = m \dot{\gamma}^n. \] (6.5)

where \( m \) is the flow consistency index, \( \dot{\gamma} \) the shear rate and \( n \) the flow behaviour index. Then

- \( n < 1 \) - pseudoplastic fluids,
- \( n = 1 \) - Newtonian fluids,
- \( n > 1 \) - dilatant fluids.

To simulate power law fluid a descriptor for dynamic \( \omega \) must be used, such as:

```cpp
#define DESCRIPTOR DynOmegaD2Q9Descriptor
```

In function `setBoundaryValues`, the initial same \( \omega \)-argument is defined:

```cpp
AnalyticalConst2D<T, T> omega0(converter.getOmega());
slattice.defineExternalField (superGeometry, 1, DESCRIPTOR<T>::ExternalField::omegaBeginsAt, 1, omega0);
slattice.defineExternalField (superGeometry, 2, DESCRIPTOR<T>::ExternalField::omegaBeginsAt, 1, omega0);
```
In the main function, the power law dynamics is defined:

```cpp
t int main(int argc, char* argv[]) {
    ...
    PowerLawBGKdynamics<T, DESCRIPTOR> bulkDynamics(converter.getOmega(), instances::getBulkMomenta<T, DESCRIPTOR>(), m, n, converter.physTime());
}
```

In 6.1 the kinematic viscosity is not more constant and then also the \( \omega \)-argument is not more constant. With using the power law model 6.5 the kinematic viscosity is computed in each step as

\[
\nu = \frac{1}{\rho} \rho m \gamma^{n-1}. \quad (6.6)
\]

The shear rate \( \dot{\gamma} \) is possible to compute with using the second invariant of the strain rate tensor \( D_{II} \)

\[
\dot{\gamma} = \sqrt{2D_{II}}, \quad (6.7)
\]

where

\[
D_{II} = \sum_{\alpha,\beta=1}^{d} E_{\alpha\beta} E_{\alpha\beta}, \quad (6.8)
\]

where

\[
E_{\alpha\beta} = -\left(1 - \frac{1}{\tau}\right) \frac{1}{2\nu} \sum_{i=0}^{q-1} f_I^i \xi_{ia} \xi_{i\beta}. \quad (6.9)
\]

This concept is very significant because \( f_I^i \xi_{ia} \xi_{i\beta} \) is usually computed during the collision process and therefore this costs in comparison to other CFD methods at almost no additional computational cost. The computation of a new \( \omega \)-argument is done in `src/dynamics/powerLawBGKdynamics.h`

```cpp
T computeOmega(T omega0_, T preFactor_, T rho_, T pi_[util::TensorVal::n苯]) ;
```
6.6 External Force

In simulations, the dynamics of a fluid is often driven by a force field (gravity, intermolecular interaction, etc.) which is space- and time-dependent, and which is possibly computed from an external source, independent of the LB simulation. In order to optimize memory access and to minimize cache-misses, the value of this force can be stored in a cell, adjacent to the particle populations. This is achieved by specifying external scalars in the lattice descriptor (see also Lesson 7). OpenLB offers by default the two descriptors ForcedD2Q9Descriptor and ForcedD3Q19Descriptor. The dynamics ForcedBGKdynamics accesses the force term defined by these descriptors, and implements a LB dynamics with body force. The algorithm is taken from Ref. [23] to guarantee second-order accuracy even when the force field is space and time dependent. An example for the implementation of a LB simulation with force term is found in the code examples/poiseuille2d/forced.

As an alternative, the velocity shift forcing scheme developed by Shan and Chen [44] and improved by Shan and Doolen [45] is also implemented and can be accessed using ForcedShanChenBGKdynamics.

6.7 Multiphysics Couplings

6.7.1 Shan-Chen Model

For the simulation of both multiphase and multicomponent flow the Shan-Chen model is implemented in OpenLB. Since its first introduction [44], many variants of the model have been developed. In this implementation, there are several forcing schemes [23, 45] and interaction potentials to choose from.

6.7.2 Implementation of Shan-Chen Two-phase Fluid

The two phases can be simulated on the same lattice instance:

```cpp
SuperLattice3D<T, DESCRIPTOR> sLattice(superGeometry);
```

Then the dynamics are chosen, which have to support external forces:

```cpp
ForcedShanChenBGKdynamics<T, DESCRIPTOR> bulkDynamics1 (omega1, instances::getExternalVelocityMomenta<T, DESCRIPTOR>());
```

Possible choices for the dynamics are ForcedBGKdynamics and ForcedShanChenBGKdynamics.
Then the interaction potential is chosen:

```cpp
ShanChen93<T, T> interactionPotential;
```

Viable interaction potentials for one component multiphase flow are ShanChen93, ShanChen94, CarnahanStarling and PengRobinson. In this model PsiEqualsRho should not be used, because this would make all the mass gather in the same place.

To enable interaction between the fluid, they have to be coupled, so the kind of coupling has to be chosen (here: ShanChenForcedSingleComponentGenerator3D) and the material numbers to which it applies. Since in the case of single component flow there is only one lattice, it is coupled with itself.

```cpp
const T G = -120.;
ShanChenForcedSingleComponentGenerator3D<T, DESCRIPTOR> coupling(G, rho0, interactionPotential);
4sLattice.addLatticeCoupling(superGeometry, 1, coupling, sLattice);
```

The interaction strength G has to be negative and the correct choice depends on the chosen interaction potential. When using PengRobinson or CarnahanStarling interaction potential, G is canceled out during computation, so the result is not affected by it (though it still has to be negative).

Finally, during the main loop the lattices have to interact with each other (or in the case of only one fluid component the lattice with itself):

```cpp
1sLattice.communicate();
1sLattice.executeCoupling();
```

These steps are placed immediately after the `collideAndStream` command.

Examples for the implementation of a LB simulation using the Shan-Chen model for two-phase flow are `examples/phaseSeparation2d` and `examples/phaseSeparation3d`.

### 6.7.3 Implementation of Shan-Chen Two-component Fluid

Two lattice instances are needed – one for each component (though there is still only one geometry):

```cpp
SuperLattice3D<T, DESCRIPTOR> sLatticeOne(superGeometry);
SuperLattice3D<T, DESCRIPTOR> sLatticeTwo(superGeometry);
```

Then the dynamics are chosen, which have to support external forces:

```cpp
ForcedShanChenBGKdynamics<T, DESCRIPTOR> bulkDynamics1 (omegal, instances::getExternalVelocityMomenta<T, DESCRIPTOR>( ));
```
Possible choices for the dynamics are ForcedBGKdynamics and ForcedShanChenBGKdynamics. One should keep in mind that tasks like definition of dynamics, external fields and initial values and the collide and stream execution have to be carried out for each lattice instance separately. The same is true for data output.

Then the interaction potential is chosen:

\[ \text{interactionPotential} = \text{PsiEqualsRho} \]

In the multicomponent case the most frequently used interaction potential is PsiEqualsRho, but ShanChen93, for example, would also be a viable choice.

To enable interaction between the fluid, they have to be coupled, so the kind of coupling has to be chosen (here: ShanChenForcedGenerator3D)and the material numbers to which it applies.

\[ \text{G} = 3. \]

The interaction strength G has to be positive. If the chosen interaction potential is PsiEqualsRho, \( G > 1 \) is needed for separation of the fluids, but it should not be much higher than 3 for stability reasons.

Finally, during the main loop the lattices have to interact with each other:

These steps are placed immediately after the collideAndStream command.

Examples for the implementation of a LB simulation using the Shan-Chen model for two-component flow are examples/multiComponent2d and examples/multiComponent3d.

6.7.4 Coupling Between Momentum and Energy Equations

As explained in reference [39], there are different schemes to couple the momentum and energy equations by means of a buoyancy force (also called Boussinesq approximation).
Some schemes add an extra force term to the collision term, other methods shift the velocity field according to Newton’s second law, and others combine an extra force term and a velocity shift. The implementation applied in OpenLB belongs to this last group of schemes.

Once the boundary values for the velocity and temperature fields are set, collision and streaming functions are called. The dynamics with an external force $F$ used for the velocity calculation (e.g. ForcedBGKdynamics) shifts the velocity before executing the collision step. The shift follows equation \ref{eq:v_shift}

$$v_{\text{shift}} = v + \frac{F}{2}$$

The code snippet responsible for this shift is defined in the collision function of the file \texttt{/dynamics/dynamics.hh} for the class ForcedBGKdynamics:

```cpp
this->momenta . computeRhoU( cell , rho , u) ;
T* force = cell .getExternal ( forceBeginsAt ) ;
for ( int iVel=0; iVel<Lattice<T>::d; ++iVel ) {
u [ iVel ] += force [ iVel ] / (T) 2 .;
}
```

Listing 6.1: Velocity shift

After the corresponding collision step using the shifted velocity, the value of the density distribution functions $f_i$ is modified by the external force with the call to the function:

```cpp
lbHelpers<T, Lattice>::addExternalForce( cell , u , omega )
```

This function follows equation \ref{eq:fi_prime} where $\tilde{f}_i$ represents the new distribution function (see reference \cite{23} for BGK model and \cite{22} for MRT model).

$$\tilde{f}_i = f_i + (1 - \frac{\omega}{2} \{ \frac{e_i - v}{e_s^2} + \frac{e_i v}{e_s^4} c_i \}) \omega_i F$$

The coupling in the collision step for the temperature field is given by the use of the velocity from the isothermal field:

```cpp
const T* u = cell .getExternal ( Lattice<T>:: ExternalField :: velocityBeginsAt )
```

The equilibrium density distribution function for the temperature has only terms of first order (see equation 3).
After the collision step, the coupling function is called `NSlattice.executeCoupling()`, where the values of the external force in the NSlattice and of the advected velocity in the ADlattice are updated.

```cpp
T* u = tPartner->get(iX, iY).getExternal(velOffset);
blockLattice.get(iX, iY).computeU(u);
```

Listing 6.2: Velocity coupling

The new force is computed from equation 6.12 following the Boussinesq approximation:

\[ F = g\rho(T - T_0) \]

(6.12)

The temperature T is obtained from the ADlattice, T_0 is the average temperature between the defined cold and hot temperatures, whereas \( \Delta T \) is the difference between the hot and cold temperatures.

```cpp
T* force = blockLattice.get(iX, iY).getExternal(forceOffset);
T temperature = tPartner->get(iX, iY).computeRho();
T rho = blockLattice.get(iX, iY).computeRho();
for(unsigned iD = 0; iD < L::d; ++iD)
{
    force[iD] = gravity * rho * (temperature - T0) / deltaTemp * dir[iD];
}
```

Listing 6.3: Computation of the Boussinesq force

### 6.8 Advection Diffusion Equation

Transport of a macroscopic density, energy or temperature is governed by the Advection-Diffusion-Equation

\[ \frac{\partial c}{\partial t} = \nabla \cdot (D \nabla c) - \nabla \cdot (\vec{v}c), \]

(6.13)

where \( c \) is the considered physical quantity (temperature, particle density), \( D \) is the diffusion coefficient and \( \vec{v} \) is a velocity field affecting \( c \). It is possible to solve this equation in terms of LBM by using an equilibrium distribution function different from the one
for the Navier-Stokes Equation \[38\]

\[ g_{i}^{eq} = w_{i}\rho \left( 1 + \frac{c_{i} \cdot \vec{v}}{c_{s}^2} \right), \tag{6.14} \]

that takes the advective transport into account. In this equation $w_i$ is a weighting factor, $c_i$ a unit vector along the lattice directions and $c_s$ the speed of sound. To use this implementation the dynamics object has to be replaced by special advection-diffusion dynamics:

```cpp
AdvectionDiffusionBGKdynamics<T, DESCRIPTOR> bulkDynamics(
    converter.getOmega(),
    instances::getBulkMomenta<T, DESCRIPTOR>());
```

Listing 6.4: advection diffusion dynamics object

Additionally, a different descriptor with fewer lattice velocities is used [26]:

```cpp
#define DESCRIPTOR AdvectionDiffusionD3Q7Descriptor
```

Listing 6.5: advection diffusion descriptor

In OpenLB D2Q5 and D3Q7 descriptors are implemented for the Advection-Diffusion Equation. Since the Advection-Diffusion Equation simulates different physical conditions than the Navier-Stokes-Equation, another set of boundary conditions is needed. A Dirichlet condition for the density is already implemented, for example to simulate a boundary with a constant temperature. To apply this condition, firstly a `sOnLatticeBoundaryCondition3D` object for the advection-diffusion boundarys has to be constructed:

```cpp
sOnLatticeBoundaryCondition3D<T, DESCRIPTOR>
sBoundaryConditionAD(sLattice);
int nC = sBoundaryConditionAD.get_sLattice().getLoadBalancer().size();

sBoundaryConditionAD.set_overlap(1);
for (int iC = 0; iC < nC; iC++) {
    OnLatticeAdvectionDiffusionBoundaryCondition3D<T, DESCRIPTOR>\* ADblockBC =
        createAdvectionDiffusionBoundaryCondition3D<T, DESCRIPTOR, AdvectionDiffusionBGKdynamics<T, DESCRIPTOR> >
        (sBoundaryConditionAD.get_sLattice().getExtendedBlockLattice(iC));
    sBoundaryConditionAD.get_CDblockBCs().push_back(ADblockBC);
}
```

56
Finally the boundary condition is set to the desired material number:

```cpp
void prepareLattice(..., SuperLattice3D<T, DESCRIPTOR>& sLattice,
                   sOnLatticeBoundaryCondition3D<T, DESCRIPTOR>& bc,
                   SuperGeometry3D<T>& superGeometry, T omega,...) {
    ...  
    /// Material=3 -> boundary with constant temperature
    bc.addTemperatureBoundary(superGeometry, 3, omega);
    ...
}
```

To apply convective transport, a velocity vector has to be passed. This can either be done individually on each cell by using:

```cpp
T velocity[3] = {vx, vy, vz};
```

Alternatively, it can be passed to the whole SuperLattice using:

```cpp
AnalyticalConst3D<T, T> velocity(vel);
```

Here, `vel` is a std::vector<T>.
6.8.1 Advection Diffusion Boundary Conditions

Dirichlet Boundary Condition

At the boundaries of a lattice, only the outgoing directions of the distribution functions are known, while those towards the domain need to be computed. Implemented in OpenLB (as explained in Section 2) are Dirichlet boundary conditions, that is, a boundary where a constant temperature is given. This boundary condition can be applied to flat walls, corners and edges (for 3-dimensional domains).

The algorithm to set a certain temperature on a wall is defined in the dynamics class Advection-DiffusionBoundariesDynamics in the file /dynamics/advectionDiffusionBoundaries.hh and works as described below.

First, the index i of the unknown distribution function $g_i$ incoming to the fluid domain is determined.

```cpp
int missingNormal = 0;
std::vector<int> missingDiagonal = util::subIndexOutgoing<L, direction, orientation>();
std::vector<int> knownIndexes = util::remainingIndexes<L>(missingDiagonal);
for (unsigned iPop = 0; iPop < missingDiagonal.size(); ++iPop)
{
    int numOfNonNullComp = 0;
    for (int iDim = 0; iDim < L::d; ++iDim)
    {
        numOfNonNullComp += abs(L::c[missingDiagonal[iPop]][iDim]);
    }
    if (numOfNonNullComp == 1)
    {
        missingNormal = missingDiagonal[iPop];
        missingDiagonal.erase(missingDiagonal.begin()+iPop);
        break;
    }
}
```

Listing 6.10: Collision step for a temperature boundary

Then, the sum of the rest of populations is computed.

```cpp
T sum = T();
for (unsigned iPop = 0; iPop < knownIndexes.size(); ++iPop)
{
    sum += cell[knownIndexes[iPop]];
}
```

58
The difference between the desired temperature value (given when setting the boundary condition) and this sum is the value assigned to the unknown distribution.

\[ T \text{ temperature} = \text{this->momenta.computeRho( cell );} \]
\[ \text{cell[missingNormal]} = \text{temperature} - \text{sum} - (T)1; \]

After that, all distribution functions are determined and a regular collision step can be executed.

\[ \text{boundaryDynamics.collide(cell, statistics);} \]

As an example, take the case of a left wall in 2D. After the streaming step all populations are known except for \( g_3 \) (see figure 1). Once the desired temperature \( T_{\text{wall}} \) at the wall is known, the value of the unknown distribution is:

\[ T_{\text{wall}} = \sum_{i=0}^{4} g_i \rightarrow g_3 = T_{\text{wall}} - (g_0 + g_1 + g_2 + g_4) \]  

(6.15)

**Adiabatic Boundary Condition**

Additionally, thanks to the simplified lattice velocities scheme used in the thermal descriptors (D2Q5 and D3Q7, see figures 1 and 2 respectively), it is possible to implement an adiabatic boundary using the bounce-back dynamics class (reference [36]).

An adiabatic boundary condition requires no heat conduction in the normal direction of the boundary. In a general situation the adiabatic boundary is set on a solid wall, meaning that the normal velocity to the wall is zero. To implement an adiabatic wall, take a 2D south wall as example. Undetermined are the distribution function \( g_4 \) and the temperature at the wall. \( g_4 \) can be computed from the distribution function in the opposite direction, in order to ensure that at the macroscopic level there is no heat conduction:

\[ g_4 = g_2 \]  

(6.16)

This procedure corresponds to the bounce-back scheme. With all the distribution functions known, the temperature at the wall can be determined from its definition:

\[ T_{\text{wall}} = \sum_{i=0}^{4} g_i \]  

(6.17)

**6.8.2 Convergence Criterion**

For thermal applications, the convergence criterion can be applied to one of the computed fields or to both of them. Generally, a value tracer on the average energy is used.
The average energy is defined proportional to the velocity squared, which makes it independent to use the \textit{NSlattice} or the \textit{ADlattice}, since both have the same velocity field.

The parameters to initialize the tracer object are the characteristic velocity of the system, the characteristic length of the system, and the desired precision of the convergence ($\epsilon_{ps}$). The listing 6.11 shows how the object is defined in the main function, and how its value is updated and checked at each time step.

```cpp
util::ValueTracer< T > converge( converter.getU( ), converter.getNy( ),
    eps );
...
for ( iT=0; iT<maxIter ; ++iT) {
  converge.takeValue( ADlattice.getStatistics( ).getAverageEnergy( ),
    true );
  ...
  if ( converge.hasConverged() ) { break; }
}
```

Listing 6.11: Check convergence

6.8.3 Creating an Application with Advection-Diffusion Dynamics

When creating a program for a thermal application, the following points must be regarded.

Lattice Descriptors

Lattice descriptors used for the AdvectionDiffusionDynamics are D2Q5 and D3Q7, which have less degrees of freedom for the velocity space.

By Chapman-Enskog expansion can be obtained that the equations do not require the fourth order isotropic lattice tensor as conventional lattice Boltzmann methods need (see [51]), therefore descriptors with less discrete velocities can be used without loss of accuracy.

It should be defined a descriptor for the velocity field that can include a external force, e.g. \textit{ForcedD3Q19Descriptor}, and another one for the temperature field, e.g. \textit{AdvectionDiffusionD3Q7Descriptor}.

60
Preparing the Geometry

In this step there is no difference with the isothermal procedure. With help from indicators and .stl-files, the desired geometry is created. Different material numbers are assigned to the cells that have different behavior.

Reading .stl-files  It is possible to read .stl-files using the OpenLB-class stlReader. There are some little differences, however, when compared to the isothermal use, due to the different converter objects. An example of its use could be:

```cpp
STLreader<T> nameIndicator("fileName.stl", converter.getDeltaX(), conversionFactor);
```

Listing 6.12: Initialization of a STLreader object

The offsets between the .stl-file and the global geometry are much easier handled if they are directly defined when creating the .stl-file, rather than trying to do it in the application code afterwards.

If a geometry is right adjusted for a grid resolution of N=100 and a conversion factor of 1, and the resolution is increased to N=200, the corresponding conversion factor also has to be increased by a factor 2 in order to keep the correct proportions of the model.

Preparing the Lattices

In a thermal application there are two independent lattices: one for the isothermal flow (usually referred to as NSlattice), and one for the thermal variables (e.g. the temperature, usually referred to as ADlattice). For each material number the desired dynamics behavior has to be defined. The implemented possibilities are instances ::

getNoDynamics (do nothing), instances :: getBounceBack (no slip), or bulkDynamics.

If corresponding, the type of boundary condition is also defined at this point. For a thermal lattice the only implemented possibility is to define a boundary with given temperature:

```cpp
TboundaryCondition.addTemperatureBoundary( superGeometry, MN, Tomega);
```

Listing 6.13: Definition of a temperature boundary

The chosen dynamics for a material number can differ between the isothermal and the thermal lattices, e.g. an obstacle with a given temperature inside a flow channel would
have a no-slip behavior for the fluid part, but be part of the bulk and have a given
temperature in the thermal lattice.

**Initialization of the Lattices (iT=0)**

**NSlattice** For all the material numbers defined as *bulkDynamics*, an initial velocity
and density has to be set (usually fluid flow at rest). Additionally, since the velocity and
the temperature field are related by a force term, an external field has to be defined. The
easiest way to do this operation is by material number:

```cpp
NSlattice.defineExternalField( superGeometry, 1, NSDESCRIPTOR<T>::ExternalField::forceBeginsAt,
                              NSDESCRIPTOR<T>::ExternalField::sizeOfForce, force );
```

Listing 6.14: Initialization of an external force field

where force is an element of type *AnalyticalF*, which can initially be set to zero.

**ADlattice** For the Advection-Diffusion lattice, an initial temperature is set (when solv-
ing the AD equation, the temperature replaces the density variable), as well as the dis-
tribution functions corresponding to this temperature value:

```cpp
T Texample = 0.5;
T zerovel[TDESCRIPTOR<T>::d] = {0., 0.};
ConstAnalyticalF2D<T,T> Example( Texample );
for ( int iPop = 0; iPop < TDESCRIPTOR<T>::q; ++iPop )
  tEqExample[ iPop ] = advectionDiffusionLbHelpers<T,TDESCRIPTOR>::quilibrium(iPop, Texample, zerovel);
ConstAnalyticalF2D<T,T> EqExample( tEqExample );
ADlattice.defineRho( superGeometry, 1, EqExample );
ADlattice.definePopulations( superGeometry, 1, EqExample );
```

Listing 6.15: Initialization of the temperature field

To apply convective transport a velocity vector has to be passed, which can be also
done by material number:

```cpp
std::vector<T> zero ( 2, T() );
ConstAnalyticalF2D<T,T> velocity ( zero );
ADlattice.defineExternalField( superGeometry, 1, TDESCRIPTOR<T>::ExternalField::velocityBeginsAt,
                              TDESCRIPTOR<T>::ExternalField::sizeOfVelocity, velocity );
```

62
Last step is to make the lattice ready for simulation:

```java
NSlattice.initialize();
ADlattice.initialize();
```

Listing 6.16: Initialization of the lattices

Setting the Boundary Conditions

If it is necessary to update the value of a boundary condition, like for example, increasing the velocity at the inflow, or changing the temperature of a boundary, it can be done following the same procedure as for the initial conditions.

Getting the Results

The desired data is saved using the VTKwriter objects, which can write the value of functors in .vti files. The functors which are usually saved are the velocity field from the NSlattice, and the temperature field (referred to as density) of the ADlattice. Thermal and isothermal information must be saved in two different objects, since they have two different descriptors.

```java
SuperLatticeVelocity2D<T, NSDESCRIPTOR> velocity( NSlattice );
SuperLatticeDensity2D<T, TDESCRIPTOR> density( ADlattice );
vtkWriterNS.addFunctor( velocity );
vtkWriterAD.addFunctor( density );
vtkWriterNS.write(iT);
vtkWriterAD.write(iT);
```

Listing 6.17: Saving results

It is important to emphasize that the data saved is in lattice units because the converter object used in the thermal simulations does not allow the conversion to physical units, but only the conversion between dimensionless and lattice units. Some conversions can be however made in order to obtain physical magnitudes, which are described later in Section 6.8.4.

Main Loop

The main loop has to include the items below.

1. **Initialization** The converter between dimensionless and lattice units is set (N, dt).
   The parameters for the simulation are chosen (Ra, Pr, Tcold, Thot, Lx, Ly, Lz).
2. **Prepare geometry**  The mesh is created, voxels are classified with different material numbers according to their behavior (inflow, outflow, etc.).

3. **Prepare lattice**  The lattice dynamics are set according to the material number assigned before. The boundary conditions are initialized: it means that the kind of boundary condition is defined at this point, but not the profile function itself. Since there are two different lattices, the definition of the dynamics and the kind of boundary conditions has to be done for each of them separately. At this point the coupling generator is also initialized (it must be always put on the NSlattice), and then it is indicated which material numbers are to be coupled with the AD-lattice.

4. **Main loop with timer**  The functions setBoundaryValues, collideAndStream, and getResults (5th, 6th and 7th steps respectively) are called repeatedly until the maximum of iterations is reached or the simulation has converged (if a convergence criteria is set).

5. **Definition of initial and boundary conditions**  It puts into practice the boundary functions’ value. In some applications it needs to refresh them during each time step, in other it is not necessary. Thermal and isothermal lattices are treated separately. As indicated before, not only velocity and density (NSlattice) and temperature (ADlattice) have to be defined, but also the external coupling forces and velocities.

6. **Collide and stream execution**  It performs the collision and the streaming step. This function is called for each of the lattices separately. After the streaming step, the coupling between the lattices (based on the Boussinesq approximation) is executed.

7. **Computation and output of the results**  It creates console and graphic output of the results at certain time steps.

6.8.4 **Obtaining Results in Thermal Simulations**

The Rayleigh and the Prandtl numbers are the dimensionless numbers which control the physics of a convection problem. The Rayleigh number for a fluid is associated with buoyancy driven flow ("Rayleigh number," Wikipedia: The Free Encyclopedia). When the Rayleigh number is below the critical value for that fluid, heat transfer is primarily
in the form of conduction; when it exceeds the critical value, heat transfer is primarily in the form of convection. For natural convection, it is defined as:

\[ Ra = \frac{g \beta \Delta T L^3}{\nu \alpha} \]  

(6.18)

where \( g \) is the acceleration due to gravity, \( \beta \) is the thermal expansion coefficient, \( \nu \) is the kinematic viscosity, \( \alpha \) is the thermal diffusivity, \( \Delta T \) is the temperature difference, and \( L \) the characteristic length.

The Prandtl number is defined as the ratio of momentum diffusivity \( \nu \) to thermal diffusivity \( \alpha \) ("Prandtl number," Wikipedia: The Free Encyclopedia):

\[ Pr = \frac{\nu}{\alpha} \]  

(6.19)

The differences between the converter objects for isothermal and thermal simulations (see Section 6 for more details) make it sometimes difficult to compute results using the algorithms already implemented in OpenLB, since most of them call elements and functions in the converter object related to physical magnitudes, which are not included in the thermal converter object.

To overcome this problem, some of the functions are re-implemented for thermal simulations, in a way that they only depend on lattice parameters and the Rayleigh and Prandtl numbers. The most important are below enumerated. Some of the were implemented by modifying existing functors in the files /functors/blockLatticeLocalF3D /blockLatticeIntegralF3D /superLatticeLocalF3D/superLatticeIntegralF3D.

**Velocity**

The resulting velocity, independent of the lattice velocity selected, can be computed as a function of the lattice velocity by:

\[ v_{res} = \frac{v_{lattice}}{latticeU} \sqrt{Ra \cdot Pr} = \frac{v_{lattice}}{\delta t \cdot N} \sqrt{Ra \cdot Pr} \]  

(6.20)

The lattice velocity \( latticeU \) is obtained from the function \( \text{converter.getU()} \) in the thermal converter object.
Pressure

The pressure in physical units is derived from the lattice pressure by using its definition from the isothermal converter object:

\[ p_{\text{phys}} = p_{\text{lattice}} \frac{\text{physForce}}{\text{physL}^{d-1}} = p_{\text{lattice}} \frac{\text{physL}^{d+1}}{\text{physT}^2} \frac{1}{\text{physL}^{d-1}} \]

\[ = p_{\text{lattice}} \left( \frac{\text{physL}}{\text{physT}} \right)^2 = p_{\text{lattice}} \left( \frac{\text{charU}}{\text{latticeU}} \right)^2 = p_{\text{lattice}} \frac{Ra \cdot Pr}{\text{latticeU}^2} \]

(6.21)

(6.22)

The lattice pressure can easily be computed from the lattice density using:

\[ p_{\text{lattice}} = \frac{\rho - 1}{3} \]

(6.23)

The physical force can also be obtained from the computed lattice force:

\[ F_{\text{phys}} = F_{\text{lattice}} \frac{\text{physL}^{d+1}}{\text{physT}^2} = F_{\text{lattice}} \left( \frac{\text{charL}}{\text{charU}} \right)^{d+1} \left( \frac{\text{charL}}{\text{charU}} \right)^2 \]

\[ = F_{\text{lattice}} \cdot \text{latticeL}^{d-1} \left( \frac{\text{charU}}{\text{latticeU}} \right)^2 = F_{\text{lattice}} \frac{\text{latticeL}^{d-1}}{\text{latticeU}^2} \frac{Ra \cdot Pr}{\text{latticeU}^2} \]

(6.24)

(6.25)

\( d \) is the number of dimensions in the problem.

For most applications can be of interest the value of the force coefficients in the different coordinate directions, which can be calculated from:

\[ C_{F_i} = F_{i,\text{phys}} \frac{1}{2 \text{charU}^2 \cdot \text{count}_i \cdot \text{latticeL}^{d-1}} = F_{i,\text{lattice}} \frac{1}{2 \text{latticeU}^2 \cdot \text{count}_i} \]

(6.26)

where \( \text{count}_i \) is the number of cells in the surface perpendicular to the direction \( i \) of the force coefficient computed.

6.8.5 Conduction Problems

For heat conduction problems there is no velocity field that advects the temperature (see [37] and [29]). In absence of convection, radiation and heat generation, the energy equation for a homogeneous medium is given by:

\[ \frac{\partial T}{\partial t} = \alpha \Delta T \]

(6.27)

A conduction simulation can be executed using an independent advection-diffusion lattice, without any velocity field coupled. In the same way as in the convection-diffusion
heat transfer, the temperature is obtained after summing the distribution functions over all directions (equation 3). The equilibrium distribution function in this case with the BGK approximation is given by:

$$g_{eq}^i = \omega_i \rho = \omega_i T$$  \hspace{1cm} (6.28)

which is equivalent to the one used in advection-diffusion simulations with the flow velocity set to zero. It means that conduction problems could be computed based on the available OpenLB code by only using a lattice with advection-diffusion dynamics and by setting the external velocity field to zero at any time.

**Multiple-Relaxation-Time (MRT)**

The implementation of the thermal lattice Boltzmann equation using the multiple-relaxation-time collision model is done similarly to the procedure used with the BGK collision model. A double MRT-LB is developed, which consists of two sets of distribution functions: an isothermal MRT model for the mass-momentum equations, and a thermal MRT model for the temperature equation. Both sets are coupled by a force term according to the Boussinesq approximation. The macroscopic governing equation for the temperature is:

$$\frac{\partial T}{\partial t} + v \nabla T = \alpha \Delta T$$  \hspace{1cm} (6.29)

where $\alpha$ is the thermal diffusivity coefficient.

The isothermal MRT model with an external force is already implemented in OpenLB for the D2Q9 and D3Q19 lattice models (dynamics class ForcedMRTdynamics). This means that only the multiple-relaxation-time thermal counterparts for 2D and 3D have to be developed.

One remark should be done about the computation of the force term in the MRT model. The existing ForcedMRTdynamics class uses the body force as described in [32]. It does not include however a velocity shift like the BGK model does. Several tests with different body forces comparing the results of the BGK model and the MRT model with and without velocity shift were conducted, showing no difference. For this reason, the ForcedMRTdynamics is left as default, without velocity shift.

**D2Q5 thermal model** The formulation for the D2Q5 thermal-MRT model is based on the reference [35]. The temperature field distribution functions $g_i$ are governed by the
The macroscopic temperature $T$ is calculated by:

$$
T = \sum_{i=0}^{4} g_i
$$

(6.31)

The weight coefficients for each lattice direction are given in equation 6.8.5.

$$
\omega_i = \begin{cases} 
\frac{3}{5} & \text{if } i = 0 \\
\frac{1}{10} & \text{if } i = 1,2,3,4 
\end{cases}
$$

The transformation matrix $N$ maps the distribution functions for the temperature $g_i$ to the corresponding moments $n_i$, i.e.,

$$
n = N \cdot g
$$

(6.32)

The transformation matrix $N$ and its inverse matrix $N^{-1}$ are shown in equations 6.33 and 6.34. There are some differences in the order of the columns respect to what is specified in the reference [35]. This is due to the different sequence used in numbering the velocity directions.
The diagonal relaxation matrix $\theta$ is:

$$\theta = \text{diag}\{\zeta_a, \zeta_a, \zeta_e, \zeta_v\}$$  (6.36)

The first relaxation rate, corresponding to the temperature, is set to zero for simplicity, since the first moment is conserved. The relaxation rates $\zeta_e$ and $\zeta_v$ are set to 1.5, whereas the relaxation rates $\zeta_a$ are a function of the thermal diffusivity $\alpha$ (6.37). The sound speed of the D2Q5 model $c_s^2$ is 1/5.

$$\alpha = c_s^2 (\tau_a - \frac{1}{2}) = \frac{1}{5} (\tau_a - \frac{1}{2}) \to \zeta_a = \frac{1}{\tau_a} = \frac{1}{5\alpha + \frac{1}{2}}$$  (6.37)
to be called in the time loop.

```cpp
SuperExternal3D<T, ADDESCRIPTOR> sExternal(superGeometry,
   sLatticeADE, ADDESCRIPTOR<T>::ExternalField::velocityBeginsAt,
   ADDESCRIPTOR<T>::ExternalField::numScalars, sLatticeAD.getOverlap () );

sExternal.communicate();
```

**Listing 6.19:** SuperExternal3D object for the communication of additional data

Although the same unit converter can be used for the Advection–Diffusion lattice, another relaxation parameter has to be handed to the dynamics, as shown in Listing 6.20, and some of the boundary conditions to take the diffusion coefficient into account. Therefore a new $\omega_{ADE}$ is computed by

$$\omega_{ADE} = \left( 4D \frac{U_L}{L_L U_C} + 0.5 \right)^{-1},$$

(6.38)

with lattice and characteristic velocity $U_L$ and $U_C$, lattice length $L_L$ as well as the desired diffusion coefficient $D$.

```cpp
ParticleAdvectionDiffusionBGKdynamics<T, ADDESCRIPTOR> bulkDynamicsAD
   ( omegaAD, instances::getBulkMomenta<T, ADDESCRIPTOR>() );
```

**Listing 6.20:** Dynamics for the simulation of particle flows the the Advection–Diffusion equation

Applying the Advection–Diffusion equation to particle flow problems requires a new dynamics due to the handling of the particle velocity by the coupling processor of the two lattices, which differs for reasons of efficiency. When constructing the coupling post-processor as shown in Listing 6.21, forces acting on the particle can be added like the Stokes drag force as shown in Line 2 and 3 of Listing 6.21. The implementation of new forces is straightforward, since only a new class which provides a function applyForce(...), computing the force in a cell, needs to be written analogously to the existing advDiffDragForce3D. Finally the lattices are linked by Line 4 of Listing 6.21, which needs to be applied to the Navier–Stokes lattice for reasons of accessibility.

```cpp
AdvectionDiffusionParticleCouplingGenerator3D<T, NSDESCRIPTION>
   coupling( ADDESCRIPTOR<T>::ExternalField::velocityBeginsAt );
advDiffDragForce3D<T, NSDESCRIPTION> dragForce( converter, radius,
   partRho );
coupling.addForce( dragForce );
```

70
Listing 6.21: Coupling of an Advection Diffusion and a Navier–Stokes lattice for particle flow simulations

For the boundary conditions the same basic objects as for the Advection–Diffusion equation can be used, however there is an additional boundary condition shown on Listing 6.22 which has to be applied at all boundaries to ensure correctness of the finite differences scheme used to compute the particle velocity.

Further information as well as results can be found in Trunk et al. [48] as well as in the examples section.

Listing 6.22: Example of a boundary condition for the particle velocity for particle flow simulations
Chapter 7: Discrete Particle Method

In this chapter the use of Lagrangian particles with OpenLB is shown.

7.1 Structure of the Particles Systems

Similar to the BlockLattice and SuperLattice structure a ParticleSystem3D and SuperParticleSystem3D structure exists. In line 2 of Listing 7.1 the SuperParticleSystem3D is instantiated, taking a SuperGeometry as parameter. In line 4 the SuperParticleSysVtuWriter is instantiated. It takes the SuperParticleSystem3D, a filename as string, and the wanted particle properties as arguments. Calling the function SuperParticleSysVtuWriter.write(int timestep) does create *.vtu files of the particles positions for the given timestep. These files can be visualized with Paraview.

Line 10 of the listing instantiates an interpolation functor for the fluids velocity, which is used in line 13 during the instantiation of StokesDragForce3D. Particles need boundary conditions also. In the listing, the simplest possible material boundary is presented. If a particle moves into a lattice node with material number 2, 4 or 5 its velocity is set to 0 and it is neglected during further computations, its state of activity is set to false. This MaterialBoundary3D is instantiated in line 16. In lines 18 and 19 the force and boundary condition are added to and stored in the respective lists in the SuperParticleSystem3D.

The actual number crunching is then done in line 25 which is positioned in the main loop of the program. The supParticleSystem.simulate(T timeStep); function integrates the particle trajectories by timeStep. Therefore all stored particle forces are computed and summed up. The particles are moved one step according to Newton’s laws. Then all stored particle boundary conditions are applied. Parallelization of the particles is done automatically.

Results of this simulation are published in Henn et al. [27].

```cpp
// SuperParticleSystems3D
SuperParticleSystem3D<T, PARTICLE> supParticleSystem(superGeometry);
// define which properties are to be written in output data
```
SuperParticleSysVtuWriter<
SuperParticleSysVtuWriter<
SuperParticleSysVtuWriter<
SuperParticleSysVtuWriter<
SuperParticleSysVtuWriter<
SuperParticleSysVtuWriter<
SuperParticleSysVtuWriter<

10
15
20
25

SuperLatticeInterpPhysVelocity3D<

auto stokesDragForce = make_shared<StokesDragForce3D<

// material numbers where particles should be reflected

auto materialBoundary = make_shared<MaterialBoundary3D<

supParticleSystem.addForce(stokesDragForce);
supParticleSystem.addBoundary(materialBoundary);
supParticleSystem.setOverlap(2. * converter.getPhysDeltaX());

\* ... *

main loop {
    supParticleSystem.simulate(converter.getPhysDeltaT());
}

Listing 7.1: Usage of class SuperParticleSystem3D

7.2 Interpolation of Fluid Velocity

As the particle position $\vec{X} : I \rightarrow \Omega$ moves in the continuous domain $\Omega$ and information on the fluid velocity can only be computed on lattice nodes $\vec{x}_i \in \Omega_h$, interpolation of the fluid velocity is necessary every time fluid-particle forces are computed. Let $\vec{u}_i^F = \vec{u}^F(\vec{x}_i)$ be the computed solution of the Navier–Stokes Equation at lattice nodes $\vec{x}_i$. Let $p \in P_n$ be the interpolating polynomial of order $n$ with $p(\vec{x}_i) = \vec{u}_i^F$ and $(\vec{x}_0, \ldots, \vec{x}_n)$ the smallest interval containing all points in the brackets. Furthermore, let $C^n[\vec{a}, \vec{b}]$ be the vector space of continuous functions that have continuous first $n$ derivatives in $[\vec{a}, \vec{b}]$. 

Then the interpolation error of the polynomial interpolation is stated by the following theorem.

**Theorem 1 (Interpolation error).** Let \( \vec{u} \in C^{n+1}\left(\vec{a},\vec{b}\right) \), \( \vec{a},\vec{b} \in \Omega \). Then for every \( \vec{x} \in \left[\vec{a},\vec{b}\right] \) there exists one \( \hat{\vec{x}} \in (\vec{x}_0, \ldots, \vec{x}_n, \vec{x}) \), such that

\[
\vec{u}_F(\vec{x}) - p_n(\vec{x}) = \frac{d_{\vec{x}}^{n+1} \vec{u}_F(\hat{\vec{x}})}{(n+1)!} \prod_{j=0}^{n} (\vec{x} - \vec{x}_j) \tag{7.1}
\]

holds.

**Proof.** See Rannacher [43, Satz 2.3]. □

Using linear \((n = 1)\) interpolation for the fluid velocity between two neighbouring lattice nodes \( \vec{a} = \vec{x}_0 \in \Omega_h, \vec{b} = \vec{x}_1 \in \Omega_h, ||\vec{x}_1 - \vec{x}_0||_2 = h \) clearly the following holds

\[
f(\vec{x}) - p_1(\vec{x}) = \frac{1}{2} d_{\vec{x}}^2 \vec{u}_F(\hat{\vec{x}})(\vec{x} - \vec{x}_0)(\vec{x} - \vec{x}_1)
\leq \frac{1}{2} d_{\vec{x}}^2 \vec{u}_F(\hat{\vec{x}})h^2
\]

and the approximation error of the linear interpolation is of order \( O(h^2) \). In the following we give reason why this order of interpolation is sufficient.

Let’s assume there exists an ideal error law of the form

\[
||\vec{u}_i^F - \vec{u}_i^{F^*}||_{L^2(\Omega_h)} = ch^\alpha,
\]

for the discrete solution \( \vec{u}_i^F \) obtained by an LBM with lattice spacing \( h \) and the analytic solution \( \vec{u}_i^{F^*} \). Then \( \alpha \in \mathbb{R}^+ \) is the to be determined order of convergence. We further define the relative error

\[
\text{Err}_h = \frac{||\vec{u}_i^F - \vec{u}_i^{F^*}||_{L^2(\Omega_h)}}{||\vec{u}_i^{F^*}||_{L^2(\Omega_h)}}.
\]

The ratio of the error laws of two distinct lattice spacings \( h_i \) and \( h_j \), forms the EOC as

\[
\text{EOC}_{i,j} = \frac{\ln(\text{Err}_{h_i}/\text{Err}_{h_j})}{\ln(h_i/h_j)} \tag{7.2}
\]

With this Krause [30, Chapter 2.3] determines an of EOC \( \approx 2 \) for the discrete solution towards the analytic solution of a stationary flow in the unit cube governed by the incompressible NSE. Therefore the order of converge of the fluid velocity obtained by an LBM can be assumed to be \( O(h^2) \). This conclusion is backed up by the theoretical
results obtained by [16]. This leads to the assumption that, each interpolation scheme of higher order than 2, would not be exhausted as the error of the incoming data is too large.

The interpolation is implemented as a trilinear interpolation using the eight nodes surrounding the particle. Let the point of interpolation \( \hat{x} \in [x(0,0,0), x(1,1,1)] \) be in the cube spanned by the lattice nodes \( x(0,0,0) \) and \( x(1,1,1) \), see Figure 7.1 for an illustration. We will denote by

\[
d = (d_0, d_1, d_2)^T = \hat{x} - x(0,0,0)
\]

the distance of the particle to the next smaller lattice node. The fluid velocities at the eight corners are named accordingly \( \tilde{u}_{(i,j,k)} \), \( i, j, k \in \{0, 1\} \). The trilinear interpolation is executed by three consecutive linear interpolations in the three different space directions. First we interpolate along the \( x \)-axis

\[
\begin{align*}
u_{(d,0,0)} &= u(0,0,0)(h - d_0) + u(1,0,0)d_0 \\
u_{(d,1,0)} &= u(0,1,0)(h - d_0) + u(1,1,0)d_0 \\
u_{(d,0,1)} &= u(0,0,1)(h - d_0) + u(1,0,1)d_0 \\
u_{(d,1,1)} &= u(0,1,1)(h - d_0) + u(1,1,1)d_0
\end{align*}
\]
followed by interpolation along the $y$-axis

\[
\begin{align*}
    u(d,d,0) &= u(d,0,0)(h - d_1) + u(d,1,0)d_1 \\
    u(d,d,1) &= u(d,0,1)(h - d_1) + u(d,1,1)d_1
\end{align*}
\]

and finally in direction of the $z$-axis

\[
\hat{\bar{u}}(\hat{x}) = u(d,d,d) = u(d,d,0)(h - d_2) + u(d,d,1)d_2.
\]

### 7.3 Class SuperParticleSystem3D

The implementation of the particle phase follows an hierarchical ansatz, similar to the Cell $\rightarrow$ BlockLattice3D $\rightarrow$ SuperLattice3D ansatz used for the implementation of the LBM. The equivalent class in the context of Lagrangian particles are Particle3D $\rightarrow$ ParticleSystem3D $\rightarrow$ SuperParticleSystem3D. The class Particle3D allocates memory for the variables of one single particle, such as its position, velocity, mass, radius and the force acting on it. It also provides the function `bool getActive()`, which returns the active state of the particle. Active particles’ positions are updated during the simulation, in contrast to non-active particles, which are only used for particle-particle interaction. The class Particle3D is intended to be inherited from, in order to provide additional properties, such as electric or magnetic charge. The particles in the domain of a specific BlockLattice3D are combined in the class ParticleSystem3D. Finally the class SuperParticleSystem3D combines all ParticleSystem3Ds, and handles the transfer of particles between them.

The concept of the class SuperParticleSystem3D is to provide an easily adaptable framework for simulation of a large number of particles arranged in and interacting with a fluid. In this context easily adaptable means that simulated forces and boundary conditions are implemented in a modular manner, such that they are easily exchangeable. Development of new forces and boundary conditions can be readily done by inheritance of provided base classes. Particle-particle interaction can be activated if necessary and deactivated to decrease simulation time. The contact detection algorithm is interchangeable. This section introduces the SuperParticleSystem3D and the mentioned properties in more detail.

The class SuperParticleSystem3D is initialised by a call to the constructor simultaneously on all PUs:
SuperParticleSystem3D(CuboidGeometry3D<T>& cuboidGeometry, 
LoadBalancer<T>& loadBalancer, SuperGeometry3D<T>& superGeometry
);

During the construction each PU instantiates one ParticleSystem3D for each local 
cuboid. Subsequently for each ParticleSystem3D a list of the ranks of PUs holding 
neighbouring cuboids is created.

Particles can be added to the SuperParticleSystem3D by a call to one of the 
addParticle() functions:

/// Add a Particle to SuperParticleSystem 
void addParticle(PARTICLETYPE<T> &p);
/// Add a number of identical Particles equally distributed in a 
given IndicatorF3D
void addParticle(IndicatorF3D<T>& ind, T mas, T rad, int no=1, std::vector<T> vel={0.,0.,0.});
/// Add a number of identical Particles equally distributed in a 
given Material Number
void addParticle(std::set<int> material, T mas, T rad, int no=1, 
std::vector<T> vec={0.,0.,0.});
/// Add Particles form a File. Save using saveToFile(std::string 
name)
void addParticlesFromFile(std::string name, T mass, T radius);

Currently there are four implementations of this class. The first adds single prede- 
fined particles, the second and third add a given number of equally distributed particles 
of the same mass and radius in an area that can be defined by either a set of material 
numbers or an indicator function. The initial particle velocity can be set optionally. Fi- 
nally particles can be added from an external file, containing their positions. In all cases 
the assignment to the correct ParticleSystem3D is carried out internally.

Particle forces and boundaries are implemented by the base classes Force3D and 
Boundary3D.

template<typename T, template<typename U> class PARTICLETYPE> 
class Force3D {
  public:
    Force3D() {
    
    virtual void applyForce(tyntype std::deque< PARTICLETYPE<T> >::iterator p, int pInt, ParticleSystem3D<T, PARTICLETYPE>& psSys )=0;
  }
}

Both classes are intended to be derived from in order to implement force and
boundary specialisations. The key function in both classes are `applyForce()` and `applyBoundary()`, which are called during each timestep of the main LBM loop. `Force3D` and `Boundary3D` specialisations are added to the `SuperParticleSystem3D` by passing a pointer to a class instantiation via a call to the respective function.

```cpp
/// Add a force to system
void addForce(std::shared_ptr<Force3D<T, PARTICLETYPE>> f);
/// Add a boundary to system
void addBoundary(std::shared_ptr<Boundary3D<T, PARTICLETYPE>> b);
```

Both functions add the passed pointer to a list of forces and boundaries, which will be looped over during the simulation step. If necessary a contact detection algorithm can be added.

```cpp
/// Set contact detection algorithm for particle-particle contact.
void setContactDetection(ContactDetection<T, PARTICLETYPE>& contactDetection);
```

A force based on contact between two particles is the contact force like described in the theory of Hertz and others and is named here as `HertzMindlinDeresiewicz3D`.

```cpp
auto hertz = make_shared < HertzMindlinDeresiewicz3D<T, PARTICLE, DESCRIPTOR>
  > (0.0003e9, 0.0003e9, 0.499, 0.499);
spSys.addForce(hertz);
```

Finally one timestep is computed by a call to the function `simulate()`.

```cpp
template<typename T, template<typename U> class PARTICLETYPE>
void SuperParticleSystem3D<T, PARTICLETYPE>::simulate(T dT)
{
  for (auto pS : _pSystems) {
    pS->_contactDetection->sort();
    pS->simulate(dT);
    pS->computeBoundary();
  }
  updateParticleDistribution();
}
```

This function contains a loop over the local `ParticleSystem3Ds` calling the local sorting algorithm and the functions `ParticleSystem3D::simulate()` and `ParticleSystem3D::computeBoundary()`. The sorting algorithm determines potential contact between particles according to the set `ContactDetection`. 

78
inline void simulate(T dT) {
    _pSys->computeForce();
    _pSys->explicitEuler(dT);
}

The inline function ParticleSystem3D::simulate() first calls the local function ParticleSystem3D::computeForce().

template<typename T, template<typename U> class PARTICLETYPE>
void ParticleSystem3D<T, PARTICLETYPE>::computeForce() {
    typename std::deque< PARTICLETYPE<T> >::iterator p;
    int pInt = 0;
    for (p = _particles.begin(); p != _particles.end(); ++p, ++pInt) {
        if (p->getActive()) {
            p->resetForce();
            for (auto f : _forces) {
                f->applyForce(p, pInt, *this);
            }
        }
    }
}

This function consists of a loop over all particles stored by the calling ParticleSystem3D. If the particle state is active, its force variable is reset to zero. Then the value computed by each previously added particle force is added to the particle’s force variable. Finally, the particle velocity and position is updated by one step of an integration method.

Returning to the function SuperParticleSystem3D::simulate(T dT) the next command in the loop is a call of the function ParticleSystem3D::computeBoundary(), which has the same structure as the ParticleSystem3D::computeForce(). After executing the loop, the function updateParticleDistribution() is called, which redistributes the particles over the ParticleSystem3Ds according to their updated position. A detailed description of this function is provided at the end of the next section.

### 7.4 Implementation of the Communication Optimal Strategy

The communication optimal strategy is implemented in the function SuperParticleSystem3D::updateParticleDistribution() already mentioned above. The function has to be called after every update of the particle positions, in order to check if the particle remained in its current cuboid, as otherwise segmentation
faults may occur during the computation of particle forces. The transfer is implemented using nonblocking operations of the MPI library.

```cpp
template<typename T, template<typename U> class PARTICLETYPE>
void SuperParticleSystem3D<T, PARTICLETYPE>::
updateParticleDistribution()
{
    /* Find particles on wrong cuboid, store in relocate and delete */
    // maps particles to their new rank
    _relocate.clear();
    for (unsigned int pS = 0; pS < _pSystems.size(); ++pS) {
        auto par = _pSystems[pS]->_particles.begin();
        while (par != _pSystems[pS]->_particles.end()) {
            // Check if particle is still in his cuboid
            if (checkCuboid(*par, 0)) {
                par++;
            } else {
                findCuboid(*par, 0);
                _relocate.insert(
                    std::make_pair(this->_loadBalancer.rank(par->getCuboid()), (*par)));
                par = _pSystems[pS]->_particles.erase(par);
            }
        }
    }
}
```

The function begins with with two nested loops. The outer loop is over all local ParticleSystem3Ds, the inner loop over the Particle3Ds of the current ParticleSystem3D. Each particle is checked if it remained in its cuboid during the last update, by the function checkCuboid(*par, 0). The first parameter of checkCuboid(*par, 0) is the particle to be tested and the second parameter is an optional spatial extension of the cuboid. If the function returns true the counter is incremented and the next particle is tested. If the function returns false the particle together with the rank of its new cuboid are copied to the std::multimap<
int, PARTICLETYPE<T> > _relocate for future treatment and removed from the std::deque<PARTICLETYPE<T> > _particles of particles.
```cpp
/* Communicate number of Particles per cuboid*/
singleton::MpiNonBlockingHelper mpiNbHelper;

/* Serialise particles */
_send_buffer.clear();
T buffer[PARTICLETYPE<T>::serialPartSize];
for (auto rN : _relocate) {
    rN.second.serialize(buffer);
    _send_buffer[rN.first].insert(_send_buffer[rN.first].end(),
        buffer, buffer+PARTICLETYPE<T>::serialPartSize);
}

The function continues by instantiating the class singleton::
MpiNonBlockingHelper, which handles memory for MPI_Request and MPI_Status
messages. Then the particles buffered in _relocate are serialised. Meaning their
data is written consecutively in memory and stored in a buffer
std::map<int, std::vector<double>> _send_buffer in preparation for the transfer.

/*Send Particles */
int noSends = 0;
for (auto rN : _rankNeighbours) {
    if (_send_buffer[rN].size() > 0) {
        ++noSends;
    }
}
mpiNbHelper.allocate(noSends);
for (auto rN : _rankNeighbours) {
    if (_send_buffer[rN].size() > 0) {
        singleton::mpi().iSend<double>(&_send_buffer[rN][0], _relocate
            .count(rN)*PARTICLETYPE<T>::serialPartSize, rN, &
            mpiNbHelper.get_mpiRequest()[k++], 1);
    }
}
singleton::mpi().barrier();

To find the number of send operations a loop over the ranks of neighbouring cuboids
is carried out, increasing the variable count each time data for a specific rank is avail-
able. Then the appropriate number of MPI_Requests is allocated. Finally the data is
sent to the respective PUs via a nonblocking MPI_Isend() and all PUs wait until the
send process is finished on each PU.

/*Receive and add particles*/
int flag = 0;
MPI_Iprobe(MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &flag,
    MPI_STATUS_IGNORE);
if (flag) {
```

for (auto rN : _rankNeighbours) {
    MPI_Status status;
    int flag = 0;
    MPI_Iprobe(rN, 1, MPI_COMM_WORLD, &flag, &status);
    if (flag) {
        int amount = 0;
        MPI_Get_count(&status, MPI_DOUBLE, &number_amount);
        T recv_buffer[amount];
        singleton::mpi().receive(recv_buffer, amount, rN, 1);
        for (int iPar=0; iPar<amount; iPar+=PARTICLETYPE<T>::serialPartSize) {
            PARTICLETYPE<T> p;
            p.unserialize(&recv_buffer[iPar]);
            if (singleton::mpi().getRank() == this->_loadBalancer.rank
                (p.getCuboid())) {
                _pSystems[this->_loadBalancer.loc(p.getCuboid())]->
                    addParticle(p);
            }
        }
    }
    }
    if (noSends > 0) {
        singleton::mpi().waitAll(mpiNbHelper);
    }
}

On the receiving side the nonblocking routine MPI_Iprobe() checks whether an incoming transmission is available. The constant MPI_ANY_SOURCE indicates that messages from all ranks are accepted. If a message is awaiting reception the flag flag is set to a nonzero value and the following switch will be true. This query is not necessary, but the following loop can be entirely skipped if no particles are transferred, which is expected to be the case most of the time.

The subsequent loop tests for each single neighbouring rank if a message awaits reception. If true the number of send MPI_Doubles is read from the status variable via an MPI_Get_count(). The appropriate memory is allocated and the message is received by wrapped call to MPI_Recv(), and written consecutively. Then new Particle3Ds are instantiated, initialised with the received data and assigned to the respective ParticleSystem3D on the updated PU. Finally, a call to MPI_Waitall() makes sure, that all MPI_Isend()s have been processed by the recipients.
7.4.1 Shadow Particles

If particle collisions are considered, it may happen that particles $P_m$ with centre $\vec{X}_m \in \tilde{\Omega}^j$ collide with particle $P_n$ with centre $\vec{X}_n \in \tilde{\Omega}^k$ in a different cuboid, as illustrated in Figure 7.2. Therefore $P_n$ has to be known on $\vec{X}_m \in \tilde{\Omega}^j$ and so-called shadow particles are introduced. Shadow particles are static particles, whose positions and velocities are not explicitly computed during the update step. Particle collision across cuboid boundaries can only occur if the distance $d = \|\vec{X}_n - \vec{X}_m\|_2$ between the participating particles is less than the sum of the two largest radii of all particles in the system. Hence the width of the particle overlap has to be at least the sum of the two largest particle radii and all particles within this overlap have to be transferred to the neighbour cuboid after each update of the particle position by an additional communication step similar to the one introduced above.

![Figure 7.2: Overlap of the particle domains. Particles within a distance to of the sum of the two largest radii to a neighbour cuboid have to be transferred to this specific neighbour cuboid.](image)
8 Input / Output

During development or even during actual simulation, it might be necessary to parametrize your program. For this case, OpenLB provides an XML parser, which can read files produced by OpenGPI [7], thereby providing a nice GUI, if you are so inclined. Details on the XML format and functions are given in Section 8.7.

The simulation data is stored in the VTK data format and can be further processed with Paraview. For output tasks that are performed only once during the simulation, it is recommended to write the data sequentially. Commonly, the geometry or cuboid information is one of these tasks. In contrast to the parallel version, it is easier to use and does not produce unnecessary data overhead. However, if the output is performed regularly in a parallel simulation, the performance may slow down using the sequential output method. Therefore, OpenLB has implemented a parallel data output functionality. At the lowest scope, every thread writes ‘vti’ files that contain the data. OpenLB writes a ‘vti’ file for every cuboid, to provide parallel data processing. Those ‘vti’ files are summarized and put together by the next hierarchy, the ‘vtm’ file. A ‘vtm’ file corresponds to the entire domain with respect to a certain time step. At the end, the different time steps are packed to a ‘pvd’ file, that is a collection of ‘vtm’ according to time steps.

The technical aspects are presented in Section 8.1 whereas the usage is demonstrated with an example in Section 8.2.

8.1 Output Data Structure

OpenLB simulation data is stored in file system according to the VTK data format [8]. This format has XML structure and the data therein is written as binary Base64 code. Additionally, the simulation data is compressed by zlib, which allows to reduce data tremendously.

On the top level, the parallel output hierarchy contains a pvd file, which consists of links to vtm files. The vtm files summarize the cuboids represented by vti files.

```xml
<?xml version="1.0"?>
```
Listing 8.1: Example of a ‘pvd’ file that points for every time step to the corresponding ‘vtm’ file. Every time step is associated to a ‘vtm’ file.

Listing 8.2: Example of a ‘vtm’ file that points to ‘vti’ files that hold data of a cuboids. Every cuboid writes its data to a ‘vti’ file, which are assembles by a ‘vtm’ file.

There is also a BlockVTKwriter that writes data sequentially. More details can be found in the source code and its documentation.

8.2 Write Simulation Data to VTK File Format

VTK data files can be visualized and postprocessed with the free software Paraview [9], which offers a nice graphical interface. The following listing shows, on the one hand, how to write VTK files sequential for a geometry and cuboid functors. On the other hand, the usage of the parallel write-routine for velocity and pressure functors is shown.

// create VTK writer object
SuperVTMwriter3D<T> vtmWriter("FileNameGoesHere");

// write only the first iteration step
if (iT==0) {
    SuperLatticeGeometry3D<T, DESCRIPTOR> geometry(sLattice,
        superGeometry);
    SuperLatticeCuboid3D<T, DESCRIPTOR> cuboid(sLattice);

    // writes the geometry and cuboids to file system, sequentially
    vtmWriter.write(geometry);
    vtmWriter.write(cuboid);
    // mandatory to call the following write()-method
    vtmWriter.createMasterFile();
}

// write every 2 sec (physical time scale)
if (iT%converter.getLatticeTime(2.)==0) {
    // create functors that process data from SuperLattice
    SuperLatticePhysVelocity3D<T, DESCRIPTOR> velocity(sLattice,
        converter);
    SuperLatticePhysPressure3D<T, DESCRIPTOR> pressure(sLattice,
        converter);

    vtmWriter.addFunctor( velocity );
    vtmWriter.addFunctor( pressure );
    // writes the added functors to file system, parallel
    vtmWriter.write(iT);
}

Listing 8.3: An exemplary code to write simulation data to file system.

Note, that the function call createMasterFile() in iT == 0 is essential to write
parallel vtk data.

8.3 Write Images Instantaneously

OpenLB is able to output image data directly. This is helpful to get a brief overview
of how the simulation is going on without using external visualization tools. Note that
only 1D data or equivalent scalar-valued data can be represented by images. Hence, for
vector-valued data, e.g. velocity, it is important to take an appropriate norm. This step
transforms the vector into a scalar and the data becomes one dimensional as required.

For 2D application it is straight forward to generate images, since every point of the
computational grid represents a pixel. However, for 3D applications this assignment
fails. OpenLB allows to reduce the 3D grid to a 2D plane by parametrizing a hyper-
plane in 3D space. The resulting 2D block lattice represents the image by assigning
lattice points to pixels.

An example of how to take a norm and how to reduce a plane is shown below:
// get the pointwise l2 norm of velocity
SuperEuklidNorm3D<T, DESCRIPTOR> normVel(velocity);
// reduce a hyperplane parametrized by normal (0,0,1) and centered in
the mother geometry from the 3D data
BlockReduction3D2D<T> planeReduction(normVel, {0, 0, 1});

Listing 8.4: An exemplary code reducing a plane in 3D

Note that internally the hyperplane is parametrized using the Hyperplane3D class. This example uses one of the helper constructors of BlockReduction3D2D to hide this detail for the common use case of parametrizing a hyperplane by a normal vector. There are further such helper constructors available if one wishes to e.g. define a hyperplane by two span vectors and its origin. However for full control over the hyperplane a Hyperplane3D instance may also be created by hand:

SuperEuklidNorm3D<T, DESCRIPTOR> normVel(velocity);
BlockReduction3D2D<T> planeReduction(
  normVel,
  // explicitly construct a 3D hyperplane
  Hyperplane3D<T>()
    .centeredIn(superGeometry.getCuboidGeometry().getMotherCuboid())
    .spannedBy({1, 0, 0}, {0, 1, 0});

BlockGifWriter<T> gifWriter;
gifWriter.write(planeReduction, iT, "vel");

Listing 8.5: Exemplary code to write images of an explicitly instantiated 3D hyperplane with

Both of these exemplary codes reduce a 3D hyperplane to a 2D lattice with 600 points on its longest side. It is possible to change this resolution either by providing it as an constructor argument to BlockReduction3D2D or by explicitly instantiating a HyperplaneLattice3D:

SuperEuklidNorm3D<T, DESCRIPTOR> normVel(velocity);
HyperplaneLattice3D<T> gifLattice(
  superGeometry.getCuboidGeometry(),
  Hyperplane3D<T>()
    .centeredIn(superGeometry.getCuboidGeometry().getMotherCuboid())
    .normalTo({0, -1, 0}),
  // resolution (floating point values are used as grid spacing instead)
  1000);
Listing 8.6: Exemplary code using an explicitly instantiated 3D hyperplane lattice

In 2D the reduction of velocity data to a block can be achieved as followed.

```cpp
SuperEuklidNorm2D<T, DESCRIPTOR> normVel( velocity );
BlockReduction2D2D<T> planeReduction( normVel );
```

---

Listing 8.7: Exemplary code reducing data in 2D

The resolution of 600 points on the longest side of the object is set as default but can be altered similarly to the listings 8.4, 8.5 and 8.6. There are two options of generating images of the processed values in 2D and 3D.

### 8.3.1 gifWriter

In this example the constructor `gifWriter` generates automatically scaled images of the PPM data type which are scaled according to the minimum and maximum value of the desired value of the time step.

```cpp
BlockReduction3D2D<T> planeReduction(normVel, gifLattice);
BlockGifWriter<T> gifWriter;
//gifWriter.write(planeReduction, 0, 0.7, iT, "vel"); //static scale
gifWriter.write( planeReduction, iT, "vel" ); // scaled
```

---

Listing 8.8: Exemplary code using `gifWriter` to create PPM files

With imagemagick’s command `convert` the PPM files generated by `gifWriter` can be combined to an animated GIF file as follows:

```bash
convert tmp/imageData/*.ppm animation.gif
```

To reduce the GIF’s file size you can use the options `fuzz` and `OptimizeFrame`, for example:

```bash
convert -fuzz 3% -layers OptimizeFrame tmp/imageData/*.ppm animation.gif
```

Even smaller files are possible with ffmpeg and conversion to MP4 video file. This could be done using a command like:

```bash
ffmpeg -pattern_type glob -i 'tmp2/imageData/*.ppm' animation.mp4
```
8.3.2 heatmap

Whereas the `gifWriter` creates only automatically scaled PPM images, the functor `heatmap` has more options to adjust the JPEG files. For this purpose the variable `plotParam` can be created and the desired modifications, e.g. minimum and maximum values of the scale, can be passed on to the optional variable.

```cpp
SuperEuklidNorm3D<T, DESCRIPTOR> normVel( velocity );
BlockReduction3D2D<T> planeReduction( normVel, {0, 0, 1} );
// write output as JPEG and changing properties
heatmap::plotParam<T> jpeg_Param;
jpeg_Param.contourlevel = 5; //setting the number of contour lines
jpeg_Param.colour = "rainbow"; //colour combination "grey", "pm3d",
"blackbody" and "rainbow" can be chosen
heatmap::write(planeReduction, iT, jpeg_Param);
```

Listing 8.9: Exemplary code using the functor `heatmap` with modified parameters

The exemplary code in listing 8.9 shows how to change the colour set and number of contour lines in the generated images. All possible adjustments are listed and used in the example `venturi3D` (see Section 12.14).

8.4 Gnuplot Interface

Often, for the analysis of simulations a plot of the data is required. OpenLB has an interface, which uses Gnuplot to create plots. Furthermore, it is possible to see the particular data needed plotting in real-time. It is also possible to use comparison data, which are directly used in the plot.

An example for the usage is shown below, and can be found in `examples/cylinder2d`.

```cpp
// Gnuplot constructor (must be static!)
// for real-time plotting: gplot("name", true) // experimental!
static Gnuplot<T> gplot( "drag" );

... 

// set data for gnuplot: input={xValue, yValue(s),
// names (optional), position of key (optional)}
gplot.setData( converter.getPhysTime( iT ), {_drag[0], 5.58},
{"drag(openLB)", "drag(schaeferTurek)", "bottom right" });

// writes a png (or optional pdf) in one file for every timestep,
```
Listing 8.10: An exemplary code to plot simulation data.

The data $\text{drag}[0]$ is calculated in the example and compared with the value 5.58. This is then plotted as shown in Fig. 8.1.

![Figure 8.1: Gnuplot output of drag calculation in cylinder2d.](image)

In order to have plots for different times following usage is recommended.

```
if ( iT%( vtkIter ) == 0 ) {
    // writes pngs: input=(name of the files (optional),
    // x range for the plot (optional))
    gplot.writePNG( iT, maxPhysT );
}
```

Listing 8.11: Creating plots for different time steps.
8.5 Console Output

In OpenLB, there is an extension of default ostreams, which handles parallel output and prefixes each line with the name of the class that produced the output. Listing 8.12 is the output of the bstep2d example.

It is easy to determine which part of OpenLB has produced a specific message. This can be very helpful in the debugging process, as well as for quickly postprocessing console output or filtering out important information without any need to go into the code. Together with OpenLB’s semi-Csv style output standard, it is possible to easily visualize any data imaginable diagrams, such as convergence rate, data errors, or simple average mass density.

```cpp
void MyClass::print() {
    OstreamManager clout(std::cout, "MyClass");
    ... 
    clout << "step=" << step << " ; avRho=" << avRho << 
        " ; maxU=" << maxU << std::endl;
}
```

Using the OstreamManager is easy and consists of two parts. First, an instance of the class OstreamManager is needed. The one created here in Line 2 is called clout like all the other instances in OpenLB. This word consists of the two words class and output. Moreover, it is quite similar to standard cout. The constructor receives two arguments: one describing the ostream to use, the other one setting the prefix-text. In line 4 the usage of an instance of the OstreamManager is shown. There is not much difference in usage between a default std::cout and an instance of OpenLB’s OstreamManager. The only thing to consider is that a normal "\n" won’t have the expected effect, so use std::endl instead.

In classes with many output producing functions however, you wouldn’t like to instantiate OstreamManager for every single function, so a central instantiation is preferred. This is done by adding a mutable OstreamManager object as a private class member and initializing it in the initialization list of each defined constructor. An example implementation of this method can be found in src/utilities/timer.{h,hh}.

Another great benefit of OstreamManager is the reduction of output in parallel. Running a program using cout on multiple cores normally means getting one line of output for each process. OstreamManager will avoid this by default and display only the output of the first processor. If this behavior is unwanted in a specific case, it can be turned off for an instance named clout by clout.setMultiOutput(true).

Further scenarios that are not yet implemented in OpenLB can make use of different
streams like the ostream std::cerr for separate error output, file streams, or something completely different. In doing so, every stream, of course, needs its own instance.
$ ./bstep2d
[LBconverter] LBconverter information
[LBconverter] characteristic values
[LBconverter] Dimension(d): dim=2
....
[prepareGeometry] Prepare Geometry ...
[SuperGeometry2D] cleaned 0 outer boundary voxel(s)
[SuperGeometry2D] cleaned 0 inner boundary voxel(s)
[SuperGeometry2D] the model is correct!
[SuperGeometryStatistics2D] updated
[SuperGeometryStatistics2D] materialNumber=0; count=4600; minPhysR=(3.35,0); maxPhysR=(5,0.75)
[SuperGeometryStatistics2D] materialNumber=1; count=92865; minPhysR=(0.0166667,0.0166667); ...
[SuperGeometryStatistics2D] materialNumber=2; count=2448; minPhysR=(0,0); maxPhysR=(20,1.5)
[SuperGeometryStatistics2D] materialNumber=3; count=43; minPhysR=(0,0.7833333); maxPhysR=(0,1.483333)
[SuperGeometryStatistics2D] materialNumber=4; count=89; minPhysR=(20,0.0166667); maxPhysR=(20,1.483333)
[main] starting simulation...
[SuperGeometryStatistics2D] updated
[Timer] step=0; percent=0; passedTime=0.109; remTime=inf; MLUPs=0
[LatticeStatistics] step=0; t=0; uMax=1.49167e-154; avEnergy=0; avRho=1
[Timer] step=300; percent=0.25; passedTime=1.199; remTime=478.401; MLUPs=26.2693
[LatticeStatistics] step=300; t=0.1; uMax=5.75006e-07; avEnergy=8.66459e-16; avRho=1

Listing 8.12: Terminal output of example bstep2d.
8.6 Read and Write STL Files

OpenLB offers the possibility to read and write geometry data in the Standard Triangulation Language, STL for short. The OpenLB class `stlReader` provides the desired functionality. In the case that the .stl-file you want to read is too large, you can use Paraview’s filter "Decimate" to reduce the number of facets.

The constructor of the class STLreader takes 2 necessary and 3 optional arguments.

```
STLreader(const std::string fName, T voxelSize, T stlSize=1,
          unsigned short int method = 2, bool verbose = false);
```

- `fName`: The filename of the STL file to be read.
- `voxelSize`: The intended spatial step size for the simulation in SI units (m).
- `stlSize`: Conversion factor if the STL file is not given in SI units. E.g. STL file in cm → stlSize = 0.01.
- `method`: Switch between methods for determining inside and outside of geometry.
  - default: fast, less stable
  - 1: slow, more stable (for untight STLs)
- `verbose`: Switch to get more output.

**Functionality:** The STL file is read and stored in the class STLmesh. A class Octree is instantiated of side-length \( r = 2^{j-1} \cdot \text{voxelSize}, j \in \mathbb{N} \) with \( j \) such that a cube with diameter 2\( r \) covers the entire STL. Intersections of triangles and the nodes of the Octree are computed and an index of the respective triangles is stored in each node. A node is a leaf if either \( r = \text{voxelSize} \) or if it does not contain any triangles.

In a second step, it is determined whether a leaf is inside the STL geometry by one of the following methods:

- (Default) One ray in Z-direction is defined for each Voxel in XY-layer. All nodes are indicated on the fly (faster, less stable).

- Define three rays (X-, Y-, Z-direction) for each leaf and count intersections with STL for each ray. Odd number of intersection means inside. The final state is decided by a majority vote (slower, more stable).
8.7 XML Parameter Files

In OpenLB essential simulation parameter can be placed in a XML. This is a useful feature, since once a program is compiled the parameter can be changed through the XML file and recompilation is redundant. As a consequence whenever parameter fitting or general simulations are wanted, this approach can help you editing only the XML file. The parsing is implemented in the the header file `io/xmlReader.h`.

The general format for the XML files is:

```
<Param>
  <Mesh>
    <lx>1</lx>
    <ly>3</ly>
  </Mesh>
  <VisualizationImages>
    <Filename>image</Filename>
  </VisualizationImages>
</Param>
```

All parameters need to be wrapped in a `<Param>` tag. To open a config file, you just pass a string with the file name to the class constructor of `XMLreader`.

```cpp
std::string fName("demo.xml");
XMLreader config(fName);

int lx, ly;
std::string imagename;
config["Mesh"].get(lx);
XMLreader meshconfig = config["Mesh"];

ly = config["Mesh"].get<int>();
config["VisualizationImages"].get<Filename>();
```

First, an `XMLreader` object `config` is created. There are multiple ways to access the configuration data. To select the tag you would like to read, you just use an associative array like syntax as shown above.

To get a specific value out of an XML parameter file, there are multiple methods. One is to pass a predefined variable to the method `get()`, which automatically converts the string in the config file to the correct type, if it is one of the basic C++ types. The other method is to call `get()` without a parameter but with the needed type as a template parameter, like `get<int>()`. For large subtrees with lots of parameters, you can also create a subobject. For this, you just have to reassign your selected subtree to a new `XMLreader`-object as is done above for `Mesh`. 

95
9 Visualization with Paraview

As already mentioned, there are several data formats that can be used in Paraview. Use ‘File – Open’ and choose the set of data you want to use. If there is a plus in front of the file name, choose this file to open the numbered collection of single files. The chosen files should now be part of the ‘Pipeline Browser’, which should be on the left hand side (if any of the panels are missing you can add them in the ‘View’ menu on the top). Click on ‘Apply’ in the ‘Properties’ panel (usually located below the ‘Pipeline Browser’) after opening.

Your data should now be visible in the center window. From within the ‘Properties’ or in one of the top tool bars, you can change the ‘Coloring’ properties, which selects what shall be displayed (e.g. physical velocity, phys pressure), which part of this choice shall be displayed (e.g. magnitude, x-value) and the way it is colored.

Make sure that ‘3D’ is part of the tool bar directly above the window where you can see your objects. If you cannot find it click on ‘2D’ which should be written instead and change it to ‘3D’ by doing this. The commands for moving your whole set of visible objects and thus changing the perspective are the following:

- Using the mouse wheel, you can zoom in and out.
- Using the right mouse button or ‘Ctrl + left mouse button’, you can move the object to the background or the foreground. In comparison to zooming in and out, this changes the level of the 3D-effect.
- Using the left mouse button allows you to turn the object.
- Clicking the mouse wheel allows you to move the object centre.

Of course you can also stick to ‘2D’, although in this case the mouse commands might change a bit.

You can visualize the temporal development of your simulation using the ‘Play’ button and the related buttons directly next to it. If you want to go to a certain time step, use the input field ‘Time’, which is also located here.

To manipulate your data in Paraview numerous so called ‘Filters’ are provided in the ‘Filters’ menu in the top bar.
9.1 Clip

With this filter, you can cut off parts of your objects, for example, to make it possible to look inside the geometry. There are several tool options to determine which part is cut off. You can choose between plane, box and sphere.

If the “wrong” side is cut off, check ‘inside out’ to make the other side visible.

Contour

Using ‘Contour’ you can show lines or planes of certain data values, which you can set.

9.2 Glyph

If you have a point data set, you can represent it as spheres using the filter ‘Glyph’ and choosing ‘Sphere’ as setting for ‘Glyph Type’. Using the resolution settings, you can smooth the surface to make the sphere look more rounded.

There are alternative ways to represent the data. As an example, arrows can be used to show the direction of a velocity. Check ‘Glyph Type’ for further possibilities.

9.3 Stream Tracer

Using the Stream Tracer allows you to draw flow lines.

Temporal Interpolator

Using this filter, you can interpolate between sets of data.

9.4 Transform

Using ‘Transform’ you can change the position and orientation of your objects, as well as the scale.
10 Functors – A General Concept for Input and Output of Data

Roughly speaking, a functor is a class that behaves like a function. Objects of a functor class perform computations by overloading the \texttt{operator()}. One big advantage of functors over functions is, that they allow the creation of a hierarchy and bundle "classes of functions". Moreover, parameters that are constant over several function evaluations only need to be passed once during instantiation.

10.1 Basic Functor Types

The functor concept is an user friendly and efficient technique to process lattice data and extract relevant data for postprocessing. In the meanwhile, OpenLB deploys the functors also for the geometry, which is a very intuitive and powerful choice.

Basically, functors are applications that operate either on the lattice \(\mathbb{N}^3\) or more generally on \(\mathbb{R}^3\). The values of such an functor may be three dimensional, e.g. velocity.

\[
\text{Functor} : \Omega \rightarrow \mathbb{R}^d, \quad d \in \mathbb{N} \quad (10.1)
\]

The nomenclature is based on the dimension of the domain. Let’s say the functor acts on a \(3d\) (super) lattice, the the functor is called \texttt{SuperLatticeF3D}. If the functor value is density, then this functor is called \texttt{SuperLatticeDensity3D}.

10.1.1 GenericF

The GenericF functor stands at the top of the hierarchy and is a virtual base class that provides interfaces. Template parameter \(S\) defines the input data type and template parameter \(T\), the output. The essential interface is the unwritten (pure virtual function) \texttt{operator()}\). Commonly, this \(()\)–operator is used as an evaluation of a certain functor, e.g. pressure at position \(x\).
10.1.2 AnalyticalF

This a subclass of GenericF for functions that lives in SI-units, e.g. for setting velocities in m/s. Parts of this class are, for example, constant, linear, interpolation and random functors, which can be evaluated by the ()–operator. There is a AnalyticalCalc class, which inherits from AnalyticalF and establishes arithmetic operations (+, −, *, /) between every type of AnalyticalF.

\[ \text{AnalyticalF}^3D : \mathbb{R}^3 \rightarrow \mathbb{R}^d, \quad d \in \mathbb{N} \]  

(10.2)

10.1.3 IndicatorF

This an other subclass of GenericF that returns a vector with elements 0 or 1. These are used to construct geometries, e.g. IndicatorSphere3D creates a sphere using an origin and radius. Evaluation returns 1, if the vector is inside the sphere and 0 elsewise. In analogy to the AnalyticalF, there are arithmetic operations as well, but with a slightly different definition. The returned object of an addition is the union, multiplication returns the intersection and subtraction represents the relative complement.

\[ \text{IndicatorF}^3D : \mathbb{R}^3 \rightarrow \{0, 1\} \]  

(10.3)

10.1.4 SmoothIndicatorF

SmoothIndicators are very similar to Indicators but their image is smooth from 0 to 1. SmoothIndicators defines a small epsilon region around the object such that is has a smooth transition form 0 to 1.

\[ \text{SmoothIndicatorF}^3D : \mathbb{R}^3 \rightarrow [0, 1] \]  

(10.4)

10.1.5 BlockLatticeF/SuperLatticeF

These functors are defined on the lattice and commonly represent the raw simulation data, e.g. pressure, velocity. SuperLattice functors are part of the parallelism layer and they delegate the calculations to the corresponding BlockLattice functors.

\[ \text{SuperLatticeF}^3D : \mathbb{N}^3 \rightarrow \mathbb{R}^d, \quad d \in \mathbb{N} \]  

(10.5)
10.1.6 InterpolationF

functors establish conversion between the analytical and lattice functors. They are very important in setting analytical boundary conditions, by evaluating the given analytical function on the lattice points. The reverse direction - from lattice to analytical functors - is where this functor receives its name, as the conversion is achieved by interpolation between the lattice points.

10.2 Application of Functors

The concept of functors benefits from generality and therefore, they are used for many applications.

10.2.1 Extract Simulation Data

Velocity, pressure and other information can be extracted from the lattice using predefined functors, see Listing 10.1. All they need to know is a SuperLatticeXD and an UnitConverter - if dimension or physical units are wanted.

```cpp
// Create functors
SuperLatticePhysVelocity3D<T, DESCRIPTOR> velocity(sLattice, converter);
SuperLatticePhysPressure3D<T, DESCRIPTOR> pressure(sLattice, converter);
```

Listing 10.1: Code example for calculating velocity and pressure using functors.

10.2.2 Define Analytic Functions

Often the inflow velocity has Poiseuille profile which is defined analytically, by means of a function. OpenLB provide analytic functors to define e.g. Poiseuille velocity profile, random values, linear and constant values.

```cpp
Poiseuille2D<T> poiseuilleU(superGeometry, 3, maxVelocity, distance2Wall);
```

Listing 10.2: Define a poiseuille velocity profile for inflow boundary condition.
10.2.3 Interpolation

Another case for interpolation functors is the conversion of a given analytical functor, such as an analytical solution to a SuperLattice functor. Afterwards, the difference can be easily calculated with the help of the functor arithmetic, see Listing 10.4. Finally, specific norms implemented as functors facilitate analysis of convergence.

```cpp
// define a analytic functor: R^3 -> R
AnalyticalConst3D<double, double> constAna(1.0);
// get analytic functor on the lattice: N^3 -> R
SuperLatticeFfromAnalyticalF3D<double, DESCRIPTOR> constLat(constAna, lattice);
```

Listing 10.3: Transition from an analytical functor to a lattice functor.

Application of this is shown in the example poiseuille2d, which is discussed in Section 12.9

10.2.4 Arithmetic and Advanced Functor Usage

Functors can be added, subtracted, ... which is a very useful and elegant method to treat data. Listing 10.4 shows how to compute the relative error over the whole three dimensional domain.

```cpp
int input[1];
double normAnaSol[1], absErr[1], relErr[1];
// define analytical solution: R^3 -> R
// for snake of simplicity it is a constant function,
// however it may be any specialization of AnalyticalF3D
AnalyticalConst3D<double, double> dSol(1.0);
// get analytical solution on the lattice: N^3 -> R
SuperLatticeFfromAnalyticalF3D<double, DESCRIPTOR> dSolLattice(dSol, lattice);
// get density out of simulation data
SuperLatticeDensity3D<T, DESCRIPTOR> d(lattice);
// compute absolute error
SuperL2Norm3D<double> dL2Norm(dSolLattice - d, superGeometry, 1);
// compute norm of solution
SuperL2Norm3D<double> dSolL2Norm(dSolLattice, superGeometry, 1);
dL2Norm(absErr, input); // access absolute error
dSolL2Norm(normAnaSol, input); // access norm of the solution
relErr[0] = absErr[0] / normAnaSol[0];
clout << "density L2error( abs ) = " << absErr[0] << " , " << "density L2error( rel ) = " << relErr[0] << std::endl;
```

101
Listing 10.4: Computation of a relative error with respect to $L^2$-norm.

For more detail, see the source code of example 12.9.

Assemble geometry with geometric primitives of type `IndicatorFXD`.

```cpp
Vector<double, 2> extendChannel(lx0, ly0);
Vector<double, 2> originChannel;
IndicatorCuboid2D<double> channel(extendChannel, originChannel);
// setup step
Vector<double, 2> extendStep(lx1, ly1);
Vector<double, 2> originStep;
IndicatorCuboid2D<double> step(extendStep, originStep);
// remove step from channel
IndicatorIdentity2D<double> channelIdent(channel-step);
```

Listing 10.5: Deploy functor arithmetic to build geometry data.

10.2.5 Setting Boundary Value

Boundary cells are marked by a certain material number in the `SuperGeometryXD`. Using a functor, velocities can be set simultaneously on all cells of this material. First, a vector that characterizes the maximum flow velocity and its directions is necessary. Then, a special functor uses this vector to initialize a Poiseuille profile. The direction can be extracted in the case of axis-parallel inflow regions automatically from the `SuperGeometryXD`. In the last step, the `SuperLattice` initializes all cells of a certain material given by the `SuperLatticeXD` with the velocities computed by the functor.

```cpp
// Creates and sets the Poiseuille inflow profile using functors
double maxVel = converter.getCharLatticeVelocity();
CirclePoiseuille3D<double> poiseuilleU(superGeometry, 3, maxVel);
sLattice.defineU(superGeometry, 3, poiseuilleU);
```

Listing 10.6: Code example for setting a Poiseuille velocity profile and a constant pressure boundary in cylinder3d.

10.2.6 Flux Functor

The flux of a quantity is defined as the rate at which this quantity passes through a fixed boundary per unit time.
As a mathematical concept, flux is represented by the surface integral of a vector field,

$$\Phi = \int \vec{F} \cdot d\vec{A}$$

where $\vec{F}$ is a vector field, and $d\vec{A}$ is an area element of the surface $A$, in the direction of the surface normal $\vec{n}$.

Flux functors calculate the discrete flux

$$\Phi_h = h^2 \sum_i \vec{f}_i \cdot \vec{n}$$

with $h$ as the grid length of the surface and $\vec{f}_i$ the vector of the quantity at grid point $i$.

As the grid of the area has to be independent from the lattice, the value of $\vec{f}_i$ will be interpolated from the surrounding lattice points.

In the general case this discrete value is calculated by SuperPlaneIntegralF3D. Note that the reduction of the relevant surface is performed by BlockReduction3D2D and that SuperPlaneIntegralF3D adds only the multiplication by the area unit as well as the normal vector for multidimensional $\vec{f}_i$.

In turn specific flux functors such as SuperPlaneIntegralFluxVelocity3D only add functor instantiation and print methods.

So, for the SuperPlaneIntegralF3D functor a surface needs to be defined. OpenLB currently supports using subsets of hyperplanes as the surfaces on which to calculate a flux.

Such a hyperplane can be defined by an origin and two span vectors, an origin and a normal vector or a 3D circle indicator. BlockReduction3D2D interpolates the full intersection of hyperplane and mother geometry. Optionally this maximal plane may be further restricted by arbitrary 2D indicators.

Note that SuperPlaneIntegralF3D as well as all specific flux functors provide a variety of constructors accepting various hyperplane parametrizations. For full control you may consider explicitly constructing a Hyperplane3D instance.

The discretization of a hyperplane parametrization (given by Hyperplane3D) into a discrete lattice is performed by HyperplaneLattice3D.

**Step 1:** Define the hyperplane by

a) **origin and two span vectors**

```cpp
Vector<T, 3> origin;
Vector<T, 3> u, v;
```
b) origin and normal vector

```cpp
Vector<T,3> origin;
Vector<T,3> normal;
```

c) normal vector (centered in mother cuboid)

```cpp
Vector<T,3> normal;
```

d) circle indicator

```cpp
IndicatorCircle3D<T> circleIndicator(center, normal, radius);
```

e) arbitrary hyperplane

```cpp
// example parametrization of a hyperplane centered in the mother
// cuboid and normal to the Z-axis
Hyperplane3D<T> hyperplane()
    .centeredIn(cuboidGeometry.getMotherCuboid())
    .normalTo({0, 0, 1});
```

**Step 1.1** (optional): Define the hyperplane discretization by

a) grid length

```cpp
T h = converter.getLatticeL();
HyperplaneLattice3D<T> hyperplaneLattice(
    cuboidGeometry,
    Hyperplane3D<T>() .originAt(origin) .spannedBy(u, v),
    h);
```

b) grid resolution

```cpp
HyperplaneLattice3D<T> hyperplaneLattice(
    cuboidGeometry,
    Hyperplane3D<T>() .originAt(origin) .spannedBy(u, v),
    600); // resolution
```

**Step 1.2** (optional): Define the flux-relevant lattice points by

a) list of material numbers

```cpp
std::vector<int> materials = {1, 2, 3};
```

a) arbitrary indicator

```cpp
SuperIndicatorF3D<T> integrationIndicator...
```
Step 1.3 (optional): Restrict the discretized intersection of hyperplane and geometry by
a) 2D circle indicator (relative to hyperplane origin)

\[ T \text{ radius} = 1.0; \]
IndicatorCircle2D\( T \) subplaneIndicator\((0,0), \text{radius}\);

a) arbitrary 2D indicator (relative to hyperplane origin)

IndicatorF2D\( T \) subplaneIndicator...

Step 2: Create a SuperF3D functor for
a) velocity flow

SuperLatticePhysVelocity3D\( T, \text{DESCRIPTOR} \) f(sLattice, converter);

b) pressure

SuperLatticePhysPressure3D\( T, \text{DESCRIPTOR} \) f(sLattice, converter);

c) any other SuperF3D functor

SuperF3D\( T \) f...

Step 3: Instantiate SuperPlaneIntegralF3D functor depending on how the hyperplane was defined and discretized.

a) using origin, two span vectors and materials list

SuperPlaneIntegralF3D\( T \) fluxF(
    f, superGeometry, origin, u, v, materials);

b) using origin, normal vector and materials list

SuperPlaneIntegralF3D\( T \) fluxF(
    f, superGeometry, origin, normal, materials);

c) using normal vector and materials list

SuperPlaneIntegralF3D\( T \) fluxF(f, superGeometry, normal, materials);

d) using 3D circle indicator and materials list

SuperPlaneIntegralF3D\( T \) fluxF(
    f, superGeometry, circleIndicator, materials);

e) using arbitrary hyperplane and integration point indicator
SuperPlaneIntegralF3D<T> fluxF(f, superGeometry, hyperplane, integrationIndicator);

f) using arbitrary hyperplane, integration point indicator and subplane indicator

SuperPlaneIntegralF3D<T> fluxF(f, superGeometry, hyperplane, integrationIndicator, subplaneIndicator);

g) using arbitrary hyperplane lattice, integration point indicator and subplane indicator

SuperPlaneIntegralF3D<T> fluxF(f, superGeometry, hyperplaneLattice, integrationIndicator, subplaneIndicator);

Step 4: Get results using operator()

int input[1]; // irrelevant
T output[5];
fluxF(output, input);

- output[0]: flow rate or plane integral (if quantity has dimension 1)
- output[1]: size of the area
- output[2..4]: flow vector (ie. vector of summed quantities)

In many cases the functor argument is either the velocity or the pressure functor. Thus Step 2 and Step 3 may be combined using SuperPlaneIntegralFluxVelocity3D respectively SuperPlaneIntegralFluxPressure3D. Their constructors are mostly identical to the ones provided by SuperPlaneIntegralF3D. In fact the only difference is that the first functor argument is replaced by references to SuperLattice3D and UnitConverter.

Step 2.1): Combined steps for velocity flux

SuperPlaneIntegralFluxVelocity3D<T> vFlux(superLattice, converter, ...);

Step 2.2): Combined steps for pressure flux

SuperPlaneIntegralFluxPressure3D<T> pFlux(superLattice, converter, ...);
Step 3.1): Output region size, volumetric flow rate and mean velocity

\[ \text{vFlux.print(} \text{std::string regionName,} \]
\[ \text{std::string fluxSiScaleName, std::string meanSiScaleName);} \]

- **fluxSiScaleName**: ‘ml/s’ or ‘l/s’ or ‘’ (default=\(m^3/s\))
- **meanSiScaleName**: ‘mm/s’ or ‘’ (default=\(m/s\))

Step 3.2): Output region size, force and mean pressure

\[ \text{pFlux.print(} \text{std::string regionName,} \]
\[ \text{std::string fluxSiScaleName, std::string meanSiScaleName);} \]

- **fluxSiScaleName**: ‘MN’ or ‘kN’ or ‘’ (default=\(N\))
- **meanSiScaleName**: ‘mmHg’ or ‘’ (default=\(Pa\))

Discrete Flux Functor

If a hyperplane is axis-aligned, flux functors may optionally be used in *discrete* mode. Passing BlockDataReductionMode::Discrete as the last argument to any plane integral or flux constructor instructs the internal BlockReduction3D2D instance to reduce the hyperplane by evaluating the underlying functor at the nearest lattice points instead of by interpolating physical positions.

Note that this imposes restrictions on the accepted hyperplane and its lattice:

- **Hyperplane3D** normal must be orthogonal to a pair of unit vectors
- **HyperplaneLattice3D** spacing must equal the distance between lattice nodes

The restriction on the hyperplane lattice spacing is fulfilled implicitly when automatic lattice parametrization is used. For example:

```cpp
// discrete flux usage in examples/aorta3d
SuperPlaneIntegralFluxVelocity3D<>
 vFluxInflow( sLattice, converter,
   superGeometry, inflow, materials, BlockDataReductionMode::
   Discrete );
```
10.3 Functor Arithmetic

Simulation data often needs heavy post-processing, in order to get relevant data. With the functor arithmetic OpenLB provides a very user friendly tool to process simulation data during simulation time. E.g. it facilitates the computation of relative errors.

```cpp
AnalyticalConst2D<
  double, double
> one(1.0);
AnalyticalConst2D<
  double, double
> two(2.0);
AnalyticalIdentity2D<
  double, double
> tmp(one + two);
// or equivalent
AnalyticPlus2D<
  double, double
> aPlus(one, two);
AnalyticalIdentity2D<
  double, double
> tmp2(aPlus);
```

Listing 10.7: Basic showcase for arithmetic operations for AnalyticalF2D.

The following explains the memory management of the functor arithmetic in OpenLB. It is strongly based on the example shown in Listing 10.7 and in particular on its third line. First, the operator+() declared in AnalyticalF2<T,S> is called by the object one, as shown in Figure 10.1. Its implementation is realized in the file analyticCalc2D.hh. Basically, there happens two things. A new object of type AnalyticalPlus2D<T,S> will be created and a shared_ptr to it, is stored into a variable of the object one. The shared_ptr is used to free the memory allocated by the new object. By now, object one cares about the arithmetic operation. However, if one is used for other arithmetic operations, its shared_ptr may be overwritten, which can causes runtime errors. It would be more intuitive if tmp cared about memory management. As a consequence, tmp should hold the shared_ptr, which is achieved in two steps. First, constructing an AnalyticalPlus2D<T,S> object, moves the shared_ptr from object one to AnalyticalPlus2D<T,S>. Then by constructing tmp the shared_ptr moves once again to the created AnalyticalIdentity2D<T,S>. Finally, tmp holds the shared_ptr and thus is responsible for the memory management.
Figure 10.1: Inheritance for AnalyticCalc2D is shown.
11 Parallel Program Execution

Whenever possible, an OpenLB application should be written in such a way that it works well on both serial and parallel platforms. As applications in computational fluid dynamics require a large amount of resources, it is essential to have the flexibility to switch to a parallel platform easily. This Section concentrates on parallelism on distributed memory machines using MPI, as distributed memory is the most common model on large-scale, parallel machines. Furthermore, MPI parallelism has become an important option even on simple desktop computers, which quite often possess multi-core processors. In this case, you will often find that MPI is actually more efficient and/or easier to obtain in a non-commercial compiler setting than OpenMP. Fortunately, it is straightforward to write parallelizable applications with OpenLB if a few basic concepts are respected. As a matter of fact, all example programs in the OpenLB distribution can be compiled with MPI and executed in parallel.

To achieve parallelism with programs that have the look and feel of serial applications, OpenLB distinguishes two classes of data. Data which is spatially distributed, such as the lattice and scalar- or vector-valued data fields, is handled through a data-parallel paradigm. The data space is partitioned into smaller regions that are distributed over the nodes of a parallel machine. In the following, these types of structures are referred to as data-parallel structures. Other data types that require a small amount of storage space are duplicated on every node. These are referred to as duplicated data. All native C++ data types are automatically duplicated by virtue of the Single-Program-Multiple-Data model of MPI. These types should be used to handle scalar values, such as the parameters of the simulation, or integral values over the solution (e.g. the average energy).

For output on the console it is strongly recommended to use OpenLB’s OstreamManager since it can help reducing output in case of parallel execution (cf Chapter 8.5).

The most important rule to respect when handling data-parallel types in application programs is to never implement explicit loops over space dimensions. Although the resulting code does yield the expected result, it is likely to run very slowly. The reason
for this is that the loops cannot be parallelized, and the code therefore runs at the speed
of a single processor, or even slower because of the implied MPI communications. An
example is given in Section 8, where it is shown how to use predefined functions for
I/O operations on data-parallel structures, instead of explicit space loops.
12 Example Programs

All the demo codes can be compiled with or without MPI, with or without OpenMP, and executed in serial or parallel.

12.1 aorta3d

In this example, the fluid flow through a bifurcation is simulated. The geometry is obtained from a mesh in STL-format. With Bouzidi boundary conditions, the curved boundary is adequately mapped and initialized entirely automatically. A Smagorinsky turbulent BGK model is used for the dynamics to stabilize the simulation for low resolutions. The output is the flux computed at the inflow and outflow region. The results have been validated through comparison with other results obtained with FEM and FVM.

12.2 bifurcation3d

The bifurcation3d example simulates particulate flow through an exemplary bifurcation of the human bronchial system. The geometry is a splitting pipe, with one inflow and two outflows. The fluid is transporting micrometer scale particles and the escape and capture rate is computed. There exist two implementations of the problem. The first one is a Euler–Euler ansatz, meaning that the fluid phase as well as the particle phase are modelled as continua. The second is an Euler–Lagrange ansatz, where the particles are modelled as discrete objects.

12.2.1 eulerEuler

In this example the particles are viewed as a continuum and described by a advection–diffusion equation. This is done similar to the thermal examples, where the temperature is the considered quantity. For particles however, inertia has to be taken into account. This is achieved by applying the Stokes drag force to the velocity field. Since for this computations also the velocity of the previous time step is required, the new descriptor
ParticleAdvectionDiffusionD3Q7Descriptor has to be used, that is capable of saving 2 velocity fields. Besides an extra lattice for the advection–diffusion equation, a SuperExternal3D structure is required to manage the communication for parallel execution.

```cpp
SuperExternal3D<T, ADDESCRIPTOR> sExternal(
    superGeometry,  
    sLatticeAD,      
    ADDESCRIPTOR<T>::ExternalField::velocityBeginsAt, 
    ADDESCRIPTOR<T>::ExternalField::numScalars, 
    sLatticeAD.getOverlap());
```

The function `communicate()` is called in the time loop and handles the communication analogue to the lattices.

Furthermore the new dynamics object ParticleAdvectionDiffusionBGKdynamics is required to access the saved velocity fields correctly and use them in an efficient way. For information on the coupling of the lattices we refer to the section on the advection–diffusion equation for particle flow problems in Chapter 6.8.5. In this example only the Stokes drag is applied by

```cpp
advDiffDragForce3D<T, NSDESCRIPTOR> dragForce( converter, radius, 
    partRho );
```

For the simulation of particles as a continuum, also new boundary conditions are required. Here addZeroDistributionBoundary represents an unidirectional outflow condition, that removes particle concentrations that cross a boundary. For the usual outflow at the bottom of the bifurcation a new ConvectionBoundary for advection–diffusion lattices can be applied, that approximates a Neumann boundary condition, for further reference see [48]. Since non-local computations (gradient is required) are performed on the the external field, also a Neumann boundary condition is required that is here implemented as addExtFieldBoundary.

**12.2.2 eulerLagrange**

The main task of his example is to show the using of Lagrangian particles with OpenLB. As already described in Chapter 7, similar to the BlockLattice and SuperLattice structure a ParticleSystem and SuperParticleSystem structure exists. Besides the particles the examples use the save feature of the SuperLattice. By
the current state of the SuperLattice can be saved and loaded again. Using this feature the startup phase for the fluid has to be computed only once.

12.3 bstep2d and bstep3d

This example implements a backward facing step. Furthermore, it is shown how checkpointing is used to regularly save the state of the simulation.

12.4 cavity2d and cavity3d

This example illustrates a flow in a cuboid, lid-driven cavity. The 2D version also shows how to use the XML parameter files and has an example description file for OpenGPI. This example is available in two different versions for sequential and parallel use.

12.5 cylinder2d and cylinder3d

This example examines a steady flow past a cylinder placed in a channel. The cylinder is offset somewhat from the center of the flow to make the steady-state symmetrical flow unstable. At the inlet, a Poiseuille profile is imposed on the velocity, whereas the outlet implements a Dirichlet pressure condition set by \( p = 0 \), inspired by [49]. For high resolution, low latticeU, and enough time to converge, the results for pressure drop, drag and lift lie within the estimated intervals for the exact results. An unsteady flow with Karman vortex street can be created by changing the Reynolds number to Re=100. The 3D version also shows the usage of the STL-reader. The model was created using the open source CAD tool FreeCAD [6].

12.6 multiComponent2d and multiComponent3d

This example demonstrates Rayleigh-Taylor instability in 2D and 3D, generated by a heavy fluid penetrating a light one. The multi-component fluid model by X. Shan and
12.7 nozzle3d

On the one hand this example describes building a cylindrical 3d geometry in OpenLB, on the other hand it examines turbulent flow in a nozzle injection tube using different turbulence models and Reynolds numbers.

For characterization different physical parameters have to be set. Resolution $N$ defines lots of physical parameters such as the velocity $charU$, the kinematic viscosity $\nu$ and two characteristic lengths $charL$ and $latticeL$. Physical length $charL$ is used to characterize the geometry and the Reynolds number. Lattice length $latticeL$ defines the mesh size and is calculated as $latticeL = charL/N$. More information about the parameter definitions are in the file units.h.

Table 12.1: This table shows the preset simulation parameters.

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$charL$</td>
<td>1m</td>
</tr>
<tr>
<td>$latticeL$</td>
<td>$\frac{1}{3}m$</td>
</tr>
<tr>
<td>$charU$</td>
<td>$\frac{1}{3}m$</td>
</tr>
<tr>
<td>$\nu$</td>
<td>0.00002 m/s</td>
</tr>
<tr>
<td>$Re_{inlet}$</td>
<td>5000</td>
</tr>
<tr>
<td>turbulence model</td>
<td>Smagorinsky</td>
</tr>
</tbody>
</table>

Figure [12.1] illustrates the geometry and the nozzle’s size as a function of the characteristic length $charL$. The nozzle consists of two circular cylinders. The inflow (red) is located left in the inletCylinder. The outflow (green) is at the right end of the injectionTube. At the main inlet, either a block profile or a power $1/7$ profile is imposed as a Dirichlet velocity boundary condition, whereas at the outlet a Dirichlet pressure condition is set by $p = 0$ (i.e. $rho = 1$).

Two vectors, origin and extend, describe the centre and normal direction of the cylinder’s circular start (origin) and end (extend) plane. The radius is defined in the function.

As mentioned before, this example examines the turbulence. The flow behavior in the inlet is characterized by the Reynolds number. The following turbulence models are based on large eddy simulation (LES). The idea behind LES is to simulate only eddies larger than a certain grid filter length, while smaller eddies are modeled. Different models are currently implemented.
Figure 12.1: Cross section of a 3d geometry of nozzle3d in dependency of characteristic length $charL$.

- The **Smagorinsky model** reduces the turbulence to a so called eddy viscosity. This viscosity depends on the Smagorinsky Constant, which has to be defined. This model has certain disadvantages at the wall.

- The **Shear-improved Smagorinsky model (SISM)** is based on the Smagorinsky model. Compared to the original model, the SISM works at the wall very well. A model specific constant has to be defined, too.

The following code shows the model selection. A model is selected, when the correlate line is uncommented. Below the model specific constants are defined. In this case the Smagorinsky Model is selected. Smagorinsky Constant is equal to 0.15.

```c++
/// Choose your turbulent model of choice

#define Smagorinsky

5 ...

#elif defined(Smagorinsky)
```

116
bulkDynamics = new SmagorinskyBGKdynamics<T, DESCRIPTOR>(converter.
    getOmega(), instances::getBulkMomenta<T, DESCRIPTOR>(),
    0.04, converter.getLatticeL(), converter.physTime());

As an example, Figure 12.2 shows the results with preset parameters.

Figure 12.2: Physical velocity field after 200 seconds with preset parameters (Smagorinsky Model, $C_S = 0.15$, $latticeL = \frac{1}{3}m$, $Re_{inlet} = 5000$).

The simulations strongly depends on the constant’s value, used in the turbulence model. But, the constant is not a general calculable value and valid for one model. It could be a function of the Reynolds number and/or another dimensionless parameter. Thus, Engineering background knowledge and experience are demanded to find physical useful values.

Generally, if the constant’s value is chosen to small, the simulation will be unphysical. If the value is to big, the turbulence will straighten turbulence.

12.8 phaseSeparation2d and phaseSeparation3d

In these examples the simulation is initialized with a given density plus small, random variation over the domain. This condition is unstable and leads to liquid-vapor phase separation. Boundaries are assumed to be periodic. These examples show the usage of multiphase flow.
12.9 poiseuille2d and poiseuille3d

For basic tests of boundary conditions, a comparison with analytical solutions is the easiest and most accurate approach. One of the fundamental applications of fluid dynamics is that of laminar flow of a Newtonian fluid in a circular pipe. This is known as Poiseuille flow. The analytical solution is easily found and is therefore a common benchmark case. It is also one of the first examples in most fluid dynamics text books for the application of the principles of fluid dynamics. The extension of the Poiseuille flow in a round pipe from 2D to 3D is trivial, consequently it is also an ideal test case for curved boundaries in 3D as well.

12.10 porousPoiseuille2d

Poiseuille flow through porous media. This implementation is the reproduction of the Guo and Zhao (2002)’s benchmark example A. The theoretical maximum velocity is calculated as in Equation 21, and the velocity profile as in Equation 23 of the original reference.
12.11 powerLaw2d

This example describes a steady non-Newtonian flow in a channel. At the inlet, a Poiseuille profile is imposed on the velocity, whereas the outlet implements a Dirichlet pressure condition set by $p = 0$.

12.12 tgv3d

The Taylor-Green-Vortex (TGV) is one of the simplest configurations, where you can investigate the generation of small structures and the resulting turbulence. The $2\pi$ periodic box domain and the single mode initial conditions contribute to the simplicity. In consequence, the TGV is a common benchmark case for Direct Numerical Simulations (DNS) and Large Eddy Simulations (LES). This example shows the usage and the effects of different subgrid scale turbulence models. The molecular dissipation rate, the eddy dissipation rate and the effective dissipation rate are calculated and plotted over the simulation time. This results can be compared with a published DNS solution by Brachet et al. [13].

Figure 12.4: Taylor-Green-Vortex vorticity plot by $t = 12$ s
The porous plate problem was described by [25] and [41]. To test the coupled model’s accuracy and to determine its experimental order of convergence (EOC), we use a numerical simulation the porous plate problem including a temperature gradient and natural convection in a square cavity. The porous plate problem describes a channel flow, where the upper cool plate moves with a constant velocity, and through the bottom warm plate a constant normal flow is injected and withdrawn at the same rate from the upper plate.

At the left and right hand side of the domain a periodic boundary condition is applied and constant velocity and temperature boundary conditions are applied to the top and bottom plates according to figure 12.5.

An analytical solution for the given steady state problem is given for the velocity and temperature distributions by:

\[ u_x(y) = u_{x,0}(e^{Re \cdot y/L - 1} - 1) \]

\[ T(y) = T_0 + \Delta T = \left( e^{Pr \cdot Re \cdot y/L} - 1 \right) \left( e^{Pr \cdot Re} - 1 \right) \]

Herein \( u_{x,0} \) is the upper plate’s velocity, \( Re = \frac{u_{x,0}L}{\nu} \) the Reynolds number depend-
ing on the injected velocity $u_y$, the fluid’s viscosity $\nu$ and the channel length $L$. The temperature difference between the hot and cold plate is given by $\Delta T = T_h - T_c$.

First we implement a couple of simulations to scale the velocity and temperature profiles for a range of the Reynolds number and Prandtl number. The relative global error described by [25]:

$$E = \sqrt{\frac{\sum_i |T(x_i) - T_a(x_i)|^2}{\sum_i |T_a(x_i)|^2}}$$

where the summation is over the entire system, $T_a$ is the analytical solution \[22\].

12.13.2 rayleighBenard2d and rayleighBenard3d

The Rayleigh-Bénard convection is a typical case of natural convection, where the lower boundary is heated and a regular pattern of convection cells is developed. This is a suitable test platform for thermal algorithms, since the driving force is a coupling between momentum and energy equations by means of a buoyancy force, which is function of the temperature, and the temperature varies spatially inside the domain.

This example demonstrates Rayleigh-Bénard convection rolls in 2D and 3D, simulated with the thermal LB model by Guo et al. [24], between a hot plate at the bottom and a cold plate at the top.

**Setup**

First case considered has an aspect ratio ($AR = Lx/Ly$) of 2, which enhances the appearance of unstable modes. The lower wall is heated with a constant temperature ($T = 1$), and the upper wall is isothermal and cold ($T = 0$). The vertical walls are set to be periodic (see figure 5).

Among the example programs implemented in OpenLB, a demo code for the Rayleigh-Bénard convection in 2D and 3D is found. This code is taken as a base for the development of most of the thermal applications. For the simulation of the Rayleigh-Bénard convection only one modification is made to the code regarding the initial conditions: to enhance the appearance of the convection cells, an instability in the domain is introduced. The available code initializes a small area near the lower boundary with a slightly higher temperature, introducing a perturbation in the system, whereas the rest of the domain is initialized with the cold temperature. In the modified code there is
no local perturbation, but the initial temperature at the domain is dependent on the space coordinates. The domain is initialized with zero velocity and a temperature field according to equation 3 by using a functor.

\[
T(x, y, t = 0) = T_{\text{max}}[(1 - \frac{y}{L_y}) + 0.1\cos(2\pi \frac{x}{L_x})]
\] (12.4)

The files created to help with the initialization of the temperature field are called `tempField.h` and `tempField2.h`. The first one computes the temperature at every point of the lattice, as a function of its macroscopic position, and then this value is applied on the lattice as the density (line 3). The second file calculates the equilibrium distribution functions for every node corresponding to the given temperature and zero velocity. Next, the populations are defined for the desired material number in line 4. The resulting temperature field is represented in figure 6.

```cpp
TemperatureField2D<T, T> Initial( converter );
TemperatureFieldPop2D<T, T> EqInitial( converter );
ADlattice.defineRho( superGeometry, 1, Initial );
ADlattice.definePopulations( superGeometry, 1, EqInitial );
```

Listing 12.1: Initialization of the temperature field

In the equation 12.4, the y-dependent part of the equation matches to the stationary solution of the problem, corresponding to a case where there is no fluid movement and the heat transfer only occurs by conduction. The cosine term introduces a disturbance in the system, which enhances the appearance of the convection cells.

Simulation Parameters

Computations where run for a range of different Rayleigh \((3 \cdot 10^3, 6 \cdot 10^5)\) and Prandtl \((0.3, 1)\) numbers, in order test how the software works. The spatial resolution was fixed to 100 cells in y-direction, and the time discretization was switched between \(10^{-3}\) and \(10^{-4}\), which give lattice velocities of 0.1 and 0.01 respectively. The convergence criterion is applied on the average energy, and it is set to a precision of \(10^{-5}\).

12.13.3 squareCavity2d and squareCavity3d

A common application for the validation of thermal models is the numerical simulation of the natural convection in a square cavity. For this configuration there is an extensive database in a wide range of Rayleigh numbers, which allows to verify the accuracy of the thermal model.
The problem considered is shown schematically in Figure 12.13.3. The horizontal walls of the cavity are adiabatic, while the vertical walls are kept isothermal, with the left wall at high temperature \( T_{\text{hot}} = 1 \) and the right wall at low temperature \( T_{\text{cold}} = 0 \).

The dynamics chosen for the velocity field is ForcedBGKdynamics, and for the temperature field AdvectionDiffusionBGKdynamics.

Simulation Parameters

Taking air at 293K as working fluid, the value of the Prandtl number is \( Pr = 0.71 \) and is kept constant. The Rayleigh number ranges from \( 10^3 \) to \( 10^6 \).

Different spatial resolutions are tested for each Rayleigh number, in order to study the grid convergence. The time-step size is adjusted so that the lattice velocity stays at the value 0.02. This ensures that the Mach number is kept at incompressible levels. The convergence criterion is set by a standard deviation of \( 10^{-6} \) in the kinetic energy.

MRT

The new implemented MRT model for thermal applications is first examined on the 2-dimensional cavity. The only setup differences to the BGK model are the lattice descriptors (ForcedMRT-D2Q9Descriptor and AdvectionDiffusionMRTD2Q5Descriptor) and the dynamics objects selected, which are now specialized for the MRT dynamics (ForcedMRTdynamics and AdvectionDiffusion-MRTdynamics).
This simulation was used as a test for different important aspects of the implementation too. First, the formulation of the MRT model, particularly the values of the transformation matrix, the relaxation times and the sound speed of the lattice. The first implementation was based in [34], but it had variations over 10% with respect to the BGK model, so another formulation (reference [35]) was selected, which gave much closer results to the BGK model. No special treatment was required to make use of the available boundary conditions.

The number of iterations required to achieve the desired precision, that is, the number of time steps until the steady-state solution is reached, was found to be usually higher for the MRT simulations. Furthermore, the execution time is between 4 and 8 times longer when compared to the BGK simulations.

12.14 venturi3d

This example examines a steady flow in a venturi tube. A Venturi tube is a cylindrical tube, which has a reduced cross-section in the middle part. At this constriction is an injection tube. As a result of the accelerating fluid in the constriction, the static pressure decreases and the injection tube’s fluid is pumped in the main tube.

The overall geometry is built with adding together single bodies. Each body’s geometry is defined by certain points (position vectors) in the coordinate system and their radius. A cone-shaped cylinder needs the centre of the start and end circle as well as the radii. Following code builds the geometry and shows the semantics.

```
/// Definition of the geometry of the venturi

//Definition of the cross-sections’ centers
Vector<T,3> C0(0,50,50);
Vector<T,3> C1(5,50,50);
Vector<T,3> C2(40,50,50);
Vector<T,3> C3(80,50,50);
Vector<T,3> C4(120,50,50);
Vector<T,3> C5(160,50,50);
Vector<T,3> C6(195,50,50);
Vector<T,3> C7(200,50,50);
Vector<T,3> C8(190,50,50);
Vector<T,3> C9(115,50,50);
Vector<T,3> C10(115,25,50);
Vector<T,3> C11(115,5,50);
Vector<T,3> C12(115,3,50);
Vector<T,3> C13(115,7,50);
```
//Definition of the radii
T radius1 = 10; // radius of the tightest part
T radius2 = 20; // radius of the widest part
T radius3 = 4;  // radius of the small exit

//Building the cylinders and cones
IndicatorCylinder3D<T> inflow(C0, C1, radius2);
IndicatorCylinder3D<T> cyl1(C1, C2, radius2);
IndicatorCone3D<T> co1(C2, C3, radius2, radius1);
IndicatorCylinder3D<T> cyl2(C3, C4, radius1);
IndicatorCone3D<T> co2(C4, C5, radius1, radius2);
IndicatorCylinder3D<T> cyl3(C5, C6, radius2);
IndicatorCylinder3D<T> outflow0(C7, C8, radius2);
IndicatorCylinder3D<T> cyl4(C9, C10, radius3);
IndicatorCone3D<T> co3(C10, C11, radius3, radius1);
IndicatorCylinder3D<T> outflow1(C12, C13, radius1);

//Addition of the cylinders to overall geometry
IndicatorIdentity3D<T> venturi(cyl1 + cyl2 + cyl3 + cyl4 + co1 + co2 + co3);

Following figure visualizes the defined point’s position.

Figure 12.7: Schematic diagramm visualizing the defined points position.

At the main inlet, a Poiseuille profile is imposed as a Dirichlet velocity boundary condition, whereas at the outlet and the minor inlet, a Dirichlet pressure condition is set by \( p = 0 \) (i.e. \( \rho = 1 \)).
Figure 12.8: Built geometry as used in simulation.

Figure 12.9: Simulation after 200 simulated time steps.
Bibliography


13 License

GNU Free Documentation License
Version 1.2, November 2002

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms
of this License. Such a notice grants a world-wide, royalty-free license, unlimited in
duration, to use that work under the conditions stated herein. The “Document”, below,
refers to any such manual or work. Any member of the public is a licensee, and is
addressed as “you”. You accept the license if you copy, modify or distribute the work
in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document
or a portion of it, either copied verbatim, or with modifications and/or translated into
another language.

A “Secondary Section” is a named appendix or a front-matter section of the Doc-
ument that deals exclusively with the relationship of the publishers or authors of the
Document to the Document’s overall subject (or to related matters) and contains noth-
ing that could fall directly within that overall subject. (Thus, if the Document is in part
a textbook of mathematics, a Secondary Section may not explain any mathematics.) The
relationship could be a matter of historical connection with the subject or with related
matters, or of legal, commercial, philosophical, ethical or political position regarding
them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated,
as being those of Invariant Sections, in the notice that says that the Document is released
under this License. If a section does not fit the above definition of Secondary then it is
not allowed to be designated as Invariant. The Document may contain zero Invariant
Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover
Texts or Back-Cover Texts, in the notice that says that the Document is released under
this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may
be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented
in a format whose specification is available to the general public, that is suitable for
revising the document straightforwardly with generic text editors or (for images com-
posed of pixels) generic paint programs or (for drawings) some widely available draw-
ing editor, and that is suitable for input to text formatters or for automatic translation to
a variety of formats suitable for input to text formatters. A copy made in an otherwise
Transparent file format whose markup, or absence of markup, has been arranged to
thwart or discourage subsequent modification by readers is not Transparent. An image
format is not Transparent if used for any substantial amount of text. A copy that is not
“Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without
markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. Copying in quantity
If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for
   authorship of the modifications in the Modified Version, together with at least
   five of the principal authors of the Document (all of its principal authors, if it has
   fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the
   publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other
   copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public
   permission to use the Modified Version under the terms of this License, in the
   form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required
   Cover Texts given in the Document’s license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled “History”, Preserve its Title, and add to it an item
   stating at least the title, year, new authors, and publisher of the Modified Version
   as given on the Title Page. If there is no section Entitled “History” in the Docu-
   ment, create one stating the title, year, authors, and publisher of the Document
   as given on its Title Page, then add an item describing the Modified Version as
   stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to
   a Transparent copy of the Document, and likewise the network locations given
   in the Document for previous versions it was based on. These may be placed in
   the “History” section. You may omit a network location for a work that was pub-
   lished at least four years before the Document itself, or if the original publisher of
   the version it refers to gives permission.

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Ti-
   tle of the section, and preserve in the section all the substance and tone of each of
   the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents,
unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate,
or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.