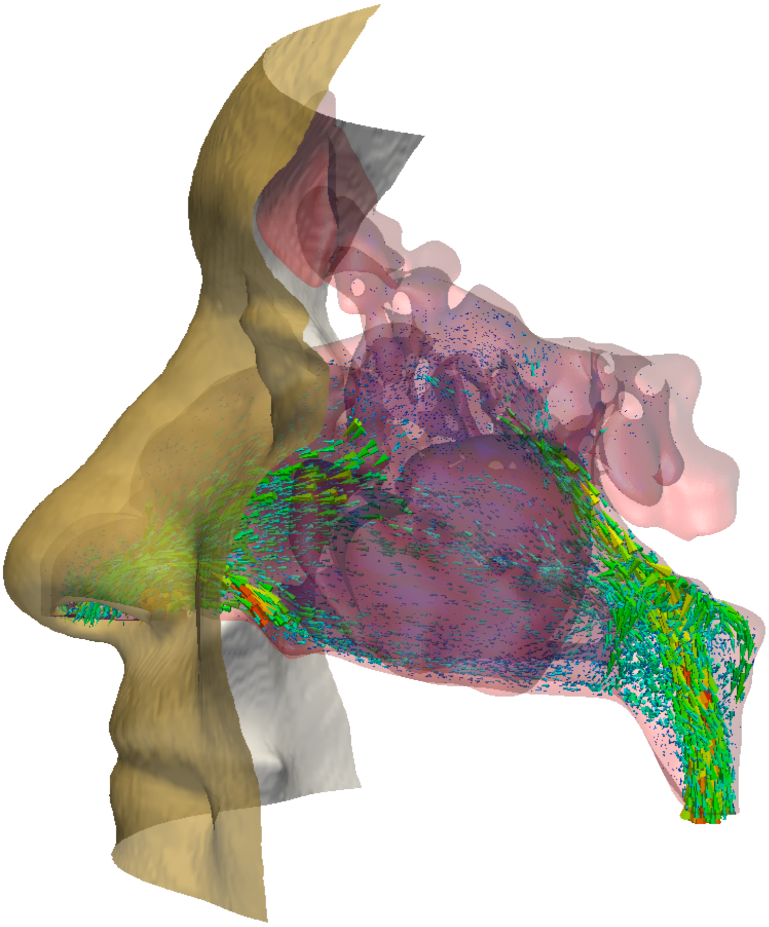
OpenLB User Guide

Associated with Release 1.5 of the Code



Authors: Adrian Kummerländer, Sam Avis, Halim Kusumaatmaja, Fedor Bukreev, Davide Dapelo, Simon Großmann, Nicolas Hafen, Claudius Holeksa, Anna Husfeldt, Julius Jeßberger, Louis Kronberg, Jan E. Marquardt, Johanna Mödl, Johannes Nguyen, Tim Pertzel, Stephan Simonis, Lukas Springmann, Nando Suntoyo, Dennis Teutscher, Mingliang Zhong, Mathias J. Krause

openlb.net

Contents

1 Introduction

1.1 Lattice Boltzmann Methods

In general, the lattice Boltzmann method (LBM) can be interpreted as a bottom-up procedure to numerically approximate a given partial differential equation (PDE). In contrast to the conventional top-down discretization of the target equation (TEQ), the LBM emerges from the space and time discretization of a discrete velocity Boltzmann-type equation. Subsequently, the solution to the TEQ is recovered in a limiting process. Different PDE can be approximated by altering certain features of the method. Exemplary LBM building blocks which induce the recovery of the aimed at transport equation are discussed in the following sections. Thorough derivations of specific LBM approaches are summarized more comprehensively for example in [35, 37].

The LB alogarithm is typically parted into two steps: collision and streaming. During the collison step every distribution function f_i at $\vec{x}$ receives a collision term Ω_i, i.e.

$$f_i^*(\vec{x},t) = f_i(\vec{x},t) + \triangle t \Omega_i, \tag{1.1}$$

where $\triangle t$ is the time step size. The most prominent collision operator introduced by Bhatnager, Gross and Krook (BGK) reads

$$\Omega_i = -\frac{1}{\tau}\left[f_i - f_i^{eq}\right]. \tag{1.2}$$

The relaxation time τ determines the speed at which the populations approach equilibrium and can for example be related to the viscosity in the hydrodynamic limit. The local equilibriuim function f_i^{eq} approximates the Maxwell–Boltzmann distribution. At the streaming step all populations are shifted to the next grid point:

$$f_i\left(\vec{x} + \triangle t \vec{c}_i, t + \triangle t\right) = f_i^*(\vec{x},t), \tag{1.3}$$

where $i = 0, 1, \ldots, q-1$ enumerates the discrete velocities $\vec{c}_i$ of dimension d contained in the underlying set $DdQq$. As a linkage from the LBM to the TEQ, the macroscopic conservative variables are obtained by respective moment summation over the populations.

The following references are suggested for further insight into LBM.

- The book THE LATTICE BOLTZMANN METHOD: PRINCIPLES AND PRACTICE [2017] by Krüger et al. [37] delivers a clear and complete introduction for beginners.
- A concise introduction is given by Mohamad [45]. The book LATTICE BOLTZMANN METHOD [2011] documents for example the formal derivation of macroscopic limit equations using Chapman-Enskop expansion.
- An LBM predecessor—the Lattice-Gas Cellular Automata (LGCA)—is extensively described in Wolf-Gladrow's book LATTICE-GAS CELLULAR AUTOMATA AND LATTICE BOLTZMANN MODELS [2000]. Starting with the derivation of LGCA, the author derives the LBM step by step. Further, a helpful interpretation of LBM is given in the beginning of the book.
- A quick overview of LBM, can be obtained from the often cited paper LATTICE BOLTZMANN METHOD FOR FLUID FLOWS [1998] by Chen and Doolen [16].

1.2 OpenLB Project

1.2.1 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).

1.2.2 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 `https://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.

1.2.3 Compiling OpenLB Programs

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 texttt Makefile contained in a default example folder. 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.

To include external libraries with a standard `#include <...>` command at the top of the .cpp file for your own OpenLB program, the respective `Makefile` should contain the necessary compiler flags. For example, the fftw3 library (which must already be installed correctly under the default compiler path on your OS) can be used by adding the `-fftw3` flag to the compilation line in the `Makefile`, i.e.: `$(CXX)$(foreach file, $(SRC), $(file:.cpp=.o))$(LDFLAGS)-L$(ROOT)/$(LIBDIR)-fftw3 -lz -o $@`.

1.2.4 Which features are currently implemented?

An excerpt of the the current features is given below. Note that an extended list is provided in [35].

Lattice Boltzmann Models

BGK model for fluids	Section 6.1.3	Reference [16]
Regularized model for fluids	Section 6.1.3	Reference [40]
Multiple Relaxation Times (MRT)	Section 6.1.3	References [18, 61]
Entropic Lattice Boltzmann	Section 6.1.3	Reference [8]
BGK with adjustable speed of sound	Section 6.1.3	References [1, 17]
BGK and MRT with Smagorinsky model	Section 6.1.3	References [47]
Porous media model	Section 6.1.3	
Power law model	Section 6.1.3	

Multiphysics Coupling

Shan-Chen two-component fluid	Section 6.6	Reference [53]
Free energy model for multicomponent fluids	Section 6.6	Reference [52]
Thermal fluid with Boussinesq approximation	Section 6.6	Reference [25]

Lattice Structures

D2Q9	This lattice is available in the precompiled library
D3Q13	This lattice requires the use of a specific dynamics object (see also Ref. [19])
D3Q15	
D3Q19	This lattice is available in the precompiled library
D3Q27	

Boundary Conditions for Straight Boundaries (Including Corners)

Regularized	local	Default choice for local boundaries
Finite difference (FD) velocity gradients	non-local	Default choice for non-local boundaries
Inamuro	local	
Zou/He	local	
Non-linear FD velocity gradients	non-local	

Boundary Conditions for Curved Boundaries

Bouzidi	non-local	first order	References [11]

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 inforamtion on the data structure design of OpenLB, we reffer to [36].

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.

1.2.5 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

Sam Avis (active): **dynamics**: multicomponent free energy model

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

Fedor Bukreev (active): **organization**: testing

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

Davide Dapelo (active): **core**: power-law unit converter, **dynamics**: Guo-Zhao porous, contributions on power-law, contributions on HLBM, **examples**: reactionFiniteDifferences2d, advectionDiffusion3d, advectionDiffusionPipe3d, **functors**: contributions on indicator and smooth indicator

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

Simon Großmann: **example**: poiseuille2dEOC, **io**: csv and gnuplot interface, **postprocessing**: eoc analysis

Nicolas Hafen (active): **particles**: core framework, surface resolved particles, coupling, dynamics, creator-functions, **dynamics**: moving porous media (HLBM), **examples**: surface resolved particle simulations

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

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

Claudius Holeksa (active) : **postProcessor**: free surface, **example**: free surface

Anna Husfeldt : **functors**: signed distance surface framework

Jonathan Jeppener-Haltenhoff: **functors**: wall shear stress, **examples**: channel3d, poiseuille3d, **core**: contributions to define field, **documentation**: user guide

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

Julius Jeßberger (active) : **core**: solver, template momenta concept, **examples**: poiseuille2d, cavity2dSolver, porousPlate3dSolver, **postprocessing**: error analysis, **utilities**: algorithmic differentiation

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

Mathias J. Krause (active): **core**: hybrid-parallelization approach, super structure, **communication**: OpenMP parallelization, cuboid data structure for MPI parallelization, load balancing, **general**: makefile environment for compilation, integration and maintenance of added components (since 2008), **boundaries**: Bouzidi boundary condition, convection, **geometry**: concept, parallelization, statistics, **io** new serializer and serializable concept, **functors**: concept, div. functors implementation, **examples**: venturi3d, aorta3d, **organization**: integration and maintenance of added components (2008-2017), project management (2006-)

Louis Kronberg (active): **core**: ade unit converter, **examples**: advectionDiffusion1d, advectionDiffusion2d, bstep2d

Adrian Kummerländer (active): **core**: SIMD CPU support, CUDA GPU support, population and field data structure, propagation pattern, vector hierarchy, cell interface, field data interface, meta descriptors, automatic code generation **dynamics**: new dynamics concept, dynamics tuple, momenta concept **communication**: block propagation, communication **functors**: lp-norm, flux, reduction, lattice indicator, error norms, refinement quality criterion, composition **boundaries**: new post processor concept, water-tightness testing and post-processor priority **general**: CI maintenance, Nix environment

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: **particles**: particulate flows, frame change

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

Jan E. Marquardt (active): **particles**: surface resolved particles, coupling, creator-functions, **dynamics**: Euler-Euler particle dynamics, **functors**: signed distance surface framework, **utilities**: algorithmic differentiation

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

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

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

Johanna Mödl (active): **core**: convection diffusion reaction dynamics, **examples**: advectionDiffusionReaction2d

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

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

Stephan Simonis (active): **core**: ade unit converter **examples**: advectionDiffusion1d, advectionDiffusion2d, advectionDiffusion3d, advectionDiffusionPipe3d, binaryShearFlow2d, fourRollMill2d, **documentation**: user guide, **dynamics**: MRT, entropic LB, free energy model

Lukas Springmann: **particles**: user-guide, unit tests

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)

Dennis Teutscher (active) : **visualisation**

Robin Trunk: **dynamics**: parallel thermal, advection diffusion models, 3D HLBM, Euler-Euler particle, multicomponent free energy model

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

1.2.6 How to Cite OpenLB

To reference to OpenLB in your publications, you can use the following entry for you .bib file:

```
@article{openlb-2020,
    author    = "Krause, M. J. and Kummerl{\"a}nder, A. and Avis, S. J.
                 and Kusumaatmaja, H. and Dapelo, D. and Klemens, F.
                 and Gaedtke, M. and Hafen, N. and Mink, A.
                 and Trunk, R. and Marquardt, J. E. and Maier, M.-L.
                 and Haussmann, M. and Simonis, S.",
    title     = "OpenLB--Open source lattice Boltzmann code",
    year      = "2020",
    journal   = "Computers \& Mathematics with Applications",
    doi       = "10.1016/j.camwa.2020.04.033"
}
```

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 8.6.

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

```
#include "olb2D.h"
#include "olb2D.hh"

using namespace olb;              // OpenLB namespaces
using namespace olb::descriptors; //

using T = double;
using DESCRIPTOR = D2Q9<>;
```

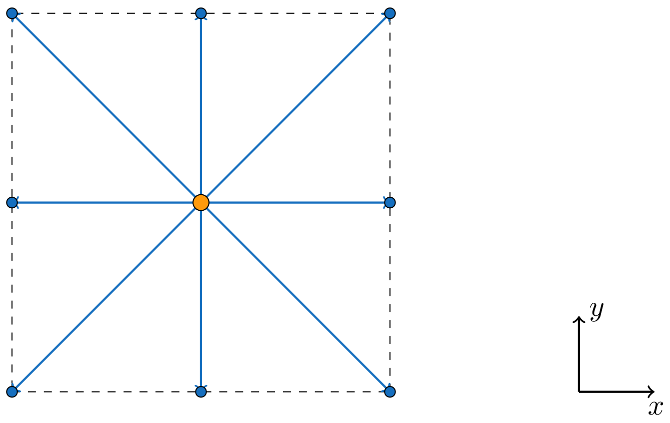


Figure 2.1: D2Q9

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 2: 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.

Line 4: All OpenLB code is contained in the namespace `olb`. The descriptors have an own namespace and define the lattice arrangement, e.g. $D2Q9$ or $D3Q19$.

Line 7: 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 8: Choice of a lattice descriptor. Lattice descriptors specify not only which lattice you are going to use, but is also used to compute the size of various dependent fields such as force vectors.

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 ex-

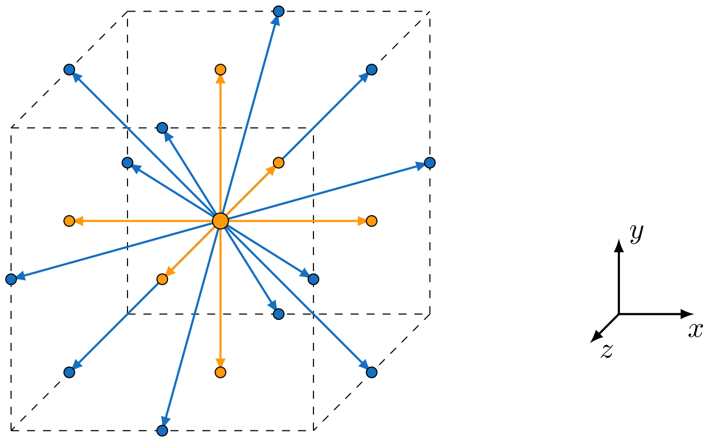


Figure 2.2: D3Q19

plain 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.

```
SuperGeometry<T,2> 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 );
  singleton::directories().setOutputDir( "./tmp/" );  // set output
```

```
    directory
OstreamManager clout( std::cout, "main" );

UnitConverterFromResolutionAndRelaxationTime<T, DESCRIPTOR> converter(
  (T)   N,                 // resolution
  (T)   relaxationTime,    // relaxation time
  (T)   charL,             // charPhysLength: reference length of simulation geometry
  (T)   1.,                // charPhysVelocity: maximal/highest expected velocity during simulation in __m / s__
  (T)   1./19230.76923,    // physViscosity: physical kinematic viscosity in __m^2 / s__
  (T)   1.                 // physDensity: physical density in __kg / m^3__
);

// Prints the converter log as console output
converter.print();
// Writes the converter log in a file
converter.write("bstep2d");

// === 2nd Step: Prepare Geometry ===
// Instantiation of a superGeometry
SuperGeometry<T,2> superGeometry( prepareGeometry(converter) );

// === 3rd Step: Prepare Lattice ===
SuperLattice<T,DESCRIPTOR> sLattice( superGeometry );
BGKdynamics<T,DESCRIPTOR> bulkDynamics (
  converter.getLatticeRelaxationFrequency(),
  instances::getBulkMomenta<T,DESCRIPTOR>()
);

//prepare Lattice and set boundaryConditions
prepareLattice( converter, sLattice, bulkDynamics, superGeometry );

// instantiate reusable functors
SuperPlaneIntegralFluxVelocity2D<T> velocityFlux( sLattice,
    converter,
    superGeometry,
    {lengthStep/2.,  heightInlet / 2.},
    {0.,  1.} );

SuperPlaneIntegralFluxPressure2D<T> pressureFlux( sLattice,
    converter,
    superGeometry,
68      {lengthStep/2.,  heightInlet / 2. },
69      {0.,  1.} );
70
71   // === 4th Step: Main Loop with Timer ===
72   clout << "starting simulation..." << std::endl;
73   Timer<T> timer( converter.getLatticeTime( maxPhysT ), superGeometry
       .getStatistics().getNvoxel() );
74   timer.start();
75
76   for ( std::size_t iT = 0; iT < converter.getLatticeTime( maxPhysT )
       ; ++iT ) {
77     // === 5th Step: Definition of Initial and Boundary Conditions
         ===
78     setBoundaryValues( converter, sLattice, iT, superGeometry );
79     // === 6th Step: Collide and Stream Execution ===
80     sLattice.collideAndStream();
81     // === 7th Step: Computation and Output of the Results ===
82     getResults( sLattice, converter, iT, superGeometry, timer,
         velocityFlux, pressureFlux );
83   }
84
85   timer.stop();
86   timer.printSummary();
87 }
```

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. [2]. For an OpenLB user, doing this is as easy as replacing the collision-streaming sequence by a call to the method `collideAndStream()`:

```
//   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, DESCRIPTOR>` 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):

```
int nx, ny, someX, someY, someF;
// <...> some code to initialize nx, ny, someX and someY
BlockLattice<T, DESCRIPTOR> lattice(nx,ny); // instantiate BlockLattice
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, DESCRIPTOR>::get()` delivers an object of type `Cell<DESCRIPTOR>`, which contains storage space for the particle populations and, if required by the `DESCRIPTOR` 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:

```
int nx, ny, someX, someY, someF;
// <...> some code to initialize nx, ny, someX and someY
BlockLattice<T, DESCRIPTOR> lattice(nx,ny); // instantiate BlockLattice
// <...> 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 computeField<FIELD>(T* ext) const Retrieve data of a field and store it 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:

```
#define D3Q19Descriptor DESCRIPTOR
int nx, ny, nz;
// <...> initialization of nx, ny, nz
BlockLattice<T,DESCRIPTOR> 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 a wide range of 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. An overview of the available boundary conditons is given by [35].

The new boundary condition system utilizes free floating functions and doesn't require a class structure. Consequently the following classes are obsolete in the current release:
`sOn/OffLatticeBoundaryConditionXD`,
`OnLatticeBoundaryConditionXD`,
`(Off)BoundaryConditionInstantiatorXD` and
`Regularized/InterpolationBoundaryManagerXD`.
Key Features of the new system are:

- Free floating design that allows for general functions like "setBoundary" to be used in multiple boundary Conditions.
- Overall slimmer design with fewer function calls and fewer loops through the block domain.
- MomentaVector and dynamicsVector is stored in `/src/core/blockLatticeStructure3D.h`
- Uncluttered function call design, which makes it easier to create new boundary conditions

The new boundaries are similarly named as in "addSlipBoundary" becomes "setSlipBoundary" in the new system.

For example, if you want to use a slip boundary condition:

Define dynamics Keep in mind that `sLattice.defineDynamics` is the same for the old and new boundary system.

Define your boundary type In this case it is the slipBoundary. To set the boundary, call the fitting setBoundaryCondition function in the following manner inside your prepareLattice function:

  - `setSlipBoundary<T,DESCRIPTOR>("superLattice", "superGeometry","MaterialNumber");`
  - The difference between the old and the new system is that every boundaryCondition needs to have the superLattice as an argument. The latticeRelaxationFrequency "omega" is usually called as the 2nd argument.

Define initial conditions

- on-lattice: `sLattice.defineRhoU(..)`
- off-lattice: `sLattice.defineRho(..),sLattice.defineUBouzidi(..)`

Define boundary values

- on-lattice: `sLattice.defineU(..)`
- off-lattice: `sLattice.defineUBouzidi(..)`

With the help of this system, 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.

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 $\mathrm{m}^2\,\mathrm{s}^{-1}$. 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 2.7. 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 ω.

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 Δx in m and time step size Δt in s are provided to the converter. From these reference values and discretization parameters, all the conversion factors and the relaxation time τ 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 Δx and Δt, 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`,

`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(
    int {N},               // resolution: number of voxels per charPhysL
    (T)   0.53,            // latticeRelaxationTime: relaxation time, have to be greater than 0.5!
    (T)   0.1,             // charPhysLength: reference length of simulation geometry
    (T)   0.2,             // charPhysVelocity: maximal/highest expected velocity during simulation in __m / s__
    (T)   0.2*2.*0.05/Re,  // physViscosity: physical kinematic viscosity in __m^2 / s__
    (T)   1.0              // physDensity: physical density in __kg / m^3__
  );
// Prints the converter log as console output
converter.print();
// Writes the converter log in a file
converter.write("converterLogFile");
// conversion from seconds to iteration steps and vice-versa
int iT = converter.getLatticeTime(maxPhysT);
T sec  = converter.getPhysTime(iT);
<...> simulation
<...> evaluation of latticeRho at the back and the front of the
```

```
       cylinder
18  T latticePressureFront = latticeRhoFront / descriptors::invCs2<T,
       DESCRIPTOR>();
19  T latticePressureBack = latticeRhoBack / descriptors::invCs2<T,
       DESCRIPTOR>();
20  T pressureDrop =   converter.getPhysPressure(latticePressureFront)
21                   - converter.getPhysPressure(latticePressureBack);
```

Listing 2.7: 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 a Simulation

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`:

T lattice.getStatistics().getAverageRho() Returns average density evaluated during the previous collision step.

T lattice.getStatistics().getAverageEnergy() Returns half the average velocity norm evaluated during the previous collision step.

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 2.8 it is shown, how to extract data out of a SuperLattice named sLattice and an SuperGeometry3D named sGeometry. The data format is a legal vtk file, that can be processed further with ParaView.

```
// 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 formata
  vtmWriter.write(geometry);
  vtmWriter.write(cuboid);
  vtmWriter.write(rank);
}
```

Listing 2.8: Extract simulation data to vtk file format.

As before mentioned, OpenLB provides functors for a bunch of data, see Listing 2.9. More details about writing simulation data can be found in Chapter 8.

```
// 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 2.9: 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 `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 2.10.

```
// velocity is an application: R^3 -> R^3
// an image in its very basic sense is an application: R^2 -> 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 2.10: Create a PPM image out of a 3D velocity functor.

This image writer provides insitu visualization which, in contrast to the `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 ϕ. The convergence is reached when the standard deviation σ of the monitored value ϕ is smaller than a given residuum ϵ times the expected value $\bar{\phi}$.

$$\sigma(\phi) = \sqrt{\frac{1}{N+1}\sum_{i=0}^{N}(\phi_i - \bar{\phi})^2} < \epsilon\bar{\phi} \tag{2.1}$$

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

$$\bar{\phi} = \frac{1}{N+1}\sum_{i=0}^{N}\phi_i \tag{2.2}$$

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

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

Listing 2.11: 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:

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

Listing 2.12: 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:

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

Listing 2.13: 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:

```
converge.takeValue(SLattice.getStatistics().getAverageEnergy(),true);
```

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

Do something like the following in the timeloop:

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

Listing 2.15: 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 are often driven by some kind of externally imposed force field. In order to optimize memory access and to minimize cache-misses, the value of this force can be stored right alongside the cell's population values. This is achieved by specifying additional fields in the lattice descriptor (see sections 6.1.2 and 6.5).

At this point we want to consider a time- and space-independent external force as a basic example. Listing 2.16 shows how such an external force can be defined for all cells of a certain material number.

```
// Define constant force
AnalyticalConst2D<T,T> force(
  8.0 * converter.getLatticeViscosity()
      * converter.getCharLatticeVelocity()
      / ( Ly*Ly ), // x-component of the force
  0.0);             // y-component of the force

// Initialize force for materials 1 and 2
superLattice.defineField<FORCE>(
  superGeometry.getMaterialIndicator({1, 2}), force);
```

Listing 2.16: Define a constant external force

This code was adapted from `examples/laminar/poiseuille2d` where just such a constant force is used to drive the channel flow. Note that the underlying `D2Q9` descriptor's field list must contain a `FORCE` field in order for `defineField<FORCE>` to work.

`SuperLattice::defineField` is just one of many template methods one can use to work with descriptor fields. e.g. we can read and write a cell's force field as follows:

```
// Get a reference to the memory location of a cell's force vector
FieldPtr<T,DESCRIPTOR,FORCE> force = cell.template getFieldPointer<
    FORCE>();
// Read a cell's force vector as an OpenLB vector value
Vector<double,2> force = cell.template getField<FORCE>();
// Set a cell's force vector to zero
cell.template setField<FORCE>(Vector<double,2>(0.0, 0.0));
```

Note that these methods work the same way for any other field that might be declared by a cell's specific descriptor.

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 `BlockStructureXD`. The most common lattice is the regular `BlockLatticeXD`, which is replaced by the `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 `CuboidStructureXD`, which does not inherit from `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 `core/latticeDescriptors.h` and `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:

```
int nx, ny;
<...> initialization of nx and ny
BlockLattice<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 2.17: Store and load the state of the simulation.

Checkpointing is also illustrated in the example programs bstep2D and bstep3D (Section 13.3.1).

2.10 Lesson 10: 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 2.18. 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++
...
```

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

Listing 2.18: Edited config.mk for MPI-parallel programs.

2.11 Lesson 11: Work with Indicators

Many of the methods covered up until this point accepted geometry and material number arguments to define their working domain. This can lead to repetition and code that is harder to read than necessary. An alternative to e.g. setting up bulk dynamics using raw material numbers is available in the form of indicator functors.

In fact most of the material number accepting operations we have covered so far use generic lattice indicators under the hood, specifically SuperIndicatorMaterial3D.

```
superLattice.defineDynamics(superGeometry, 1, &bulkDynamics);
superLattice.defineDynamics(superGeometry, 3, &bulkDynamics);
superLattice.defineDynamics(superGeometry, 4, &bulkDynamics);
// is equivalent to:
SuperIndicatorMaterial3D<T> bulkIndicator({1, 3, 4});
superLattice.defineDynamics(bulkIndicator, &bulkDynamics);
```

Listing 2.19: Indicator usage example

This can be further abstracted using SuperGeometry3D's indicator factory:

```
auto bulkIndicator = superGeometry.getMaterialIndicator({1, 3, 4});
superLattice.defineDynamics(bulkIndicator, &bulkDynamics);
```

Listing 2.20: Indicator usage in bstep3d

The advantage of this pattern is that we explicitly named materials 1, 3 and 4 as bulk materials and can reuse the indicator whenever we operate on bulk cells:

```
superLattice.defineRhoU(bulkIndicator, rho, u);
superLattice.iniEquilibrium(bulkIndicator, rho, u);
```

Listing 2.21: Indicator reusage in bstep3d

This way the bulk material domain is defined in a central place which will come in handy should we need to change them in the future.

Note that for one-off usage this can be written even more compactly:

```
superLattice.defineDynamics(
  superGeometry.getMaterialIndicator({1, 3, 4}), &bulkDynamics);
```

Listing 2.22: Inline indicator usage

This pattern of using indicators instead of raw material numbers is available for all material number accepting methods of `SuperGeometryXD`.

The methods themselves support arbitrary `SuperIndicatorFXD` instances and as such are not restricted to material indicators.

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.

To compile OpenLB at least C++14 is required. The minimum requirement for compiler is:
GCC: 5 or later
Intel compiler: 17.0 or later
Clang: 3.4 or later

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. Both can be installed with the help of the package manager *homebrew*. Install *homebrew* via:

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Install components using homebrew:

```
brew install gcc make openmpi gnuplot
```

Since `ParaView` is often used for visualization it is adviced to download the recent version from the web (https://www.paraview.org/download/).

During the usage of OpenLB some network error messages might occure. In order to avoid the reoccuring display of these, simply grant the requested network connection by clicking on the corresponding option.

3.3 Windows

The preferable appraoch is to use the Windows Subsystem for Linux (WSL) introduced in Windows 10. A guide can be found in the technical report TR4: Installing OpenLB in Windows 10 [3].

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 ϕ is called C_ϕ and the non-dimensional quantity gets the symbol ϕ^*.

To get into this topic, the book "The Lattice Boltzmann Method" [37], 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 `SuperLattice` 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 13.9.1.
2. Pre-defined geometric primitives and its combinations $(+, -, \cdot)$, see example venturi3d 13.9.5.

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.

6. Step: Construct `SuperLattice` to perform stream and collide algorithm.

```
// 1. Step: Create Indicator
STLreader<T> stlreader("filename.stl", voxelSize);
IndicatorCuboid3D<T> cylinderInFLow( extend, origin );
```

```
// 2. Step: Construct cuboidGeometry.
CuboidGeometry3D<T> cuboidGeometry(indicator, voxelSize, noOfCuboids);
// 3. Step: Construct LoadBalancer.
HeuristicLoadBalancer<T> loadBalancer(cuboidGeometry);
// 4. Step: Construct SuperGeometry.
SuperGeometry<T,3> 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 nunmer 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);
// 6. Step: Construct SuperLattice.
SuperLattice<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 computation grid the `SuperLattice`, 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 respect 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 always practical by hand e. g. if material numbers of a complex geometry are obtained from a `stl` file.

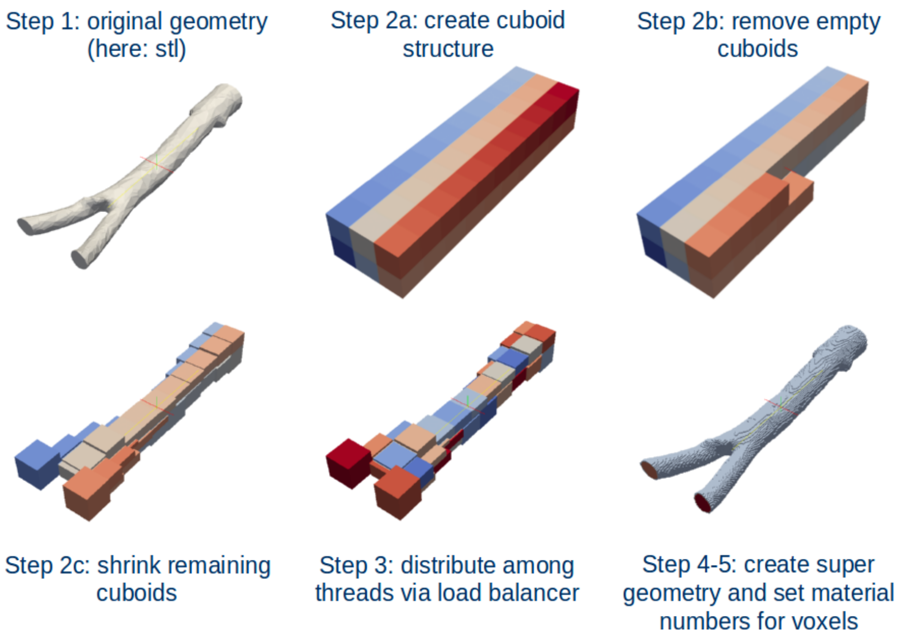

Step 1: original geometry (here: stl)
Step 2a: create cuboid structure
Step 2b: remove empty cuboids
Step 2c: shrink remaining cuboids
Step 3: distribute among threads via load balancer
Step 4-5: create super geometry and set material numbers for voxels

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 `SuperLattice`.

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

```
/// 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);
```

2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
3	1	1	1	1	1	5	5	5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
3	1	1	1	1	5	5	0	5	5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
3	1	1	1	1	5	0	0	0	5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
3	1	1	1	1	5	5	0	5	5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
3	1	1	1	1	1	5	5	5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

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).

Listing 5.2: Different rename functions to set material numbers.

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 13.9.5).

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 [4]. An introduction to modeling with FreeCAD can be found for example in `http://www.youtube.com/watch?v=geIrH1cOCzc`. 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.

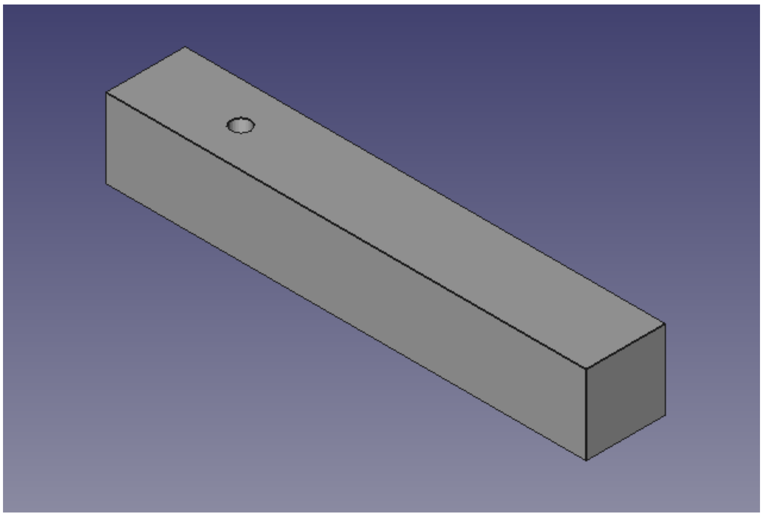


Figure 5.3: The geometry for the example `cylinder3d` from Section 13.3.3 opened in FreeCAD.

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 Ω_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 execution 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 `Cell` and is orchestrated by a `BlockLattice`, which represents a regular array of `Cells`. In each `Cell`, the q variables for the storage of the discrete velocity distribution functions f_i $(i = 0, 1, ..., q - 1)$ are contiguous in memory. Note, in terms of optimization the cells hold

$$Cell[iPop] = f_{iPop} - t_{iPop},$$

where t_{iPop} corresponds to the lattice weights. This data structure is encapsulated by

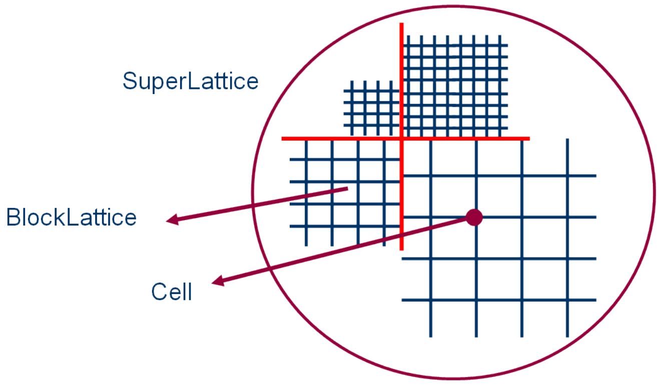


Figure 6.1: Data structures in OpenLB: A number of BlockLattices build a SuperLattice to adopt higher level software constructs like multi-block, grid refined lattices and parallelised lattices.

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

6.1.2 Descriptor

Descriptors are used to define and access LBM model specific information such as the number of dimensions and discrete velocities as well as weights and declarations of additional fields. As such a descriptor is the central definition in any OpenLB application and used troughout the code base.

```
using T = double;
using DESCRIPTOR = descriptors::D2Q9<>;

// number of spatial dimensions
const int d = descriptors::d<DESCRIPTOR>(); // == 2
// number of discrete velocities
const int q = descriptors::q<DESCRIPTOR>(); // == 9

// second discrete velocity vector
const Vector<int,2> c1 = descriptors::c<DESCRIPTOR>(1); // == {-1,1}

// weight of the first discrete velocity
const T w = descriptors::t<T,DESCRIPTOR>(0); // == 4./9.
```

OpenLB provides a rich set of such descriptors. Most of them are defined in the src/dynamics/latticeDescriptors.h Header file. Despite the central role of this concept the concrete definitions of e.g. the D2Q9 descriptor is quite compact. To illustrate this point, Listing 6.1 provides the full definition of this descriptor including all of its data.

```
template <typename... FIELDS>
struct D2Q9 : public LATTICE_DESCRIPTOR<2,9,POPULATION,FIELDS...> {
  D2Q9() = delete;
};

namespace data {

template <>
constexpr int vicinity<2,9> = 1;

template <>
constexpr int c<2,9>[9][2] = {
  { 0, 0},
  {-1, 1}, {-1, 0}, {-1,-1}, { 0,-1},
  { 1,-1}, { 1, 0}, { 1, 1}, { 0, 1}
};

template <>
constexpr int opposite<2,9>[9] = {
  0, 5, 6, 7, 8, 1, 2, 3, 4
};

template <>
constexpr Fraction t<2,9>[9] = {
  {4, 9}, {1, 36}, {1, 9}, {1, 36}, {1, 9},
  {1, 36}, {1, 9}, {1, 36}, {1, 9}
};

template <>
constexpr Fraction cs2<2,9> = {1, 3};

}
```

Listing 6.1: Definition of the D2Q9 descriptor

Many LBM based solutions to practical problems require the ability to store not just the populations of each cell but also additional data such as an external force (see e.g. section 6.5). For this reason every descriptor may be extended to include such fields:

```
using DESCRIPTOR = descriptors::D2Q9<descriptors::FORCE>;

// Check whether DESCRIPTOR contains the field FORCE
```

```
DESCRIPTOR::provides<descriptors::FORCE>(); // == true
// Get cell-local memory location of the FORCE field
const int offset = DESCRIPTOR::index<descriptors::FORCE>(); // == 9
// Get size of the descriptor's FORCE field
const int size = DESCRIPTOR::size<descriptors::FORCE>(); // == 2
```

A full list of all fields predefined by OpenLB can be found in the `src/dynamics/descriptorField.h` Header file. Note that one may add an arbitrary list of such fields to a given descriptor. Even more so it is fully supported to add fields on a per-app basis. One only needs to make sure that the type is defined prior to any custom descriptor that depends on it. e.g. a user could write inside of an app:

```
struct MY_CUSTOM_FIELD: public FIELD_BASE<42,0,0> { };
using DESCRIPTOR = D2Q9<FORCE,MY_CUSTOM_FIELD>;
```

This custom descriptor could then be used in the same way as any descriptor provided by OpenLB. This might even be preferable for very specific fields that are only used by small number of apps or specific user-provided features.

6.1.3 Dynamics

The individual blocks of the decomposed simulation domains are represented as implementations of the `BlockLattice` interface. This class executes the LB algorithm in a very traditional sense, i.e. the lattice Boltzmann equation is split into two equations, namely the *collision step*:

$$\widetilde{f}_i^h(t,\vec{r}) = f_i^h(t,\vec{r}) - \frac{1}{3\nu+1/2}\left(f_i^h(t,\vec{r}) - M^{eq}_{f_i^h}(t,\vec{r})\right) \qquad \text{in } I_h \times \Omega_h \times Q \tag{6.1}$$

and the *streaming step (propagation step)*:

$$f_i^h(t+h^2, \vec{r}+h^2\vec{v}_i) = \widetilde{f}_i^h(t,\vec{r}) \qquad \text{in } I_h \times \Omega_h \times Q\,. \tag{6.2}$$

Depending on the execution platform, all cells of the `BlockLattice` are processed either sequentially or in parallel in order to apply the local collision steps. This is followed by the non-local streaming step. The streaming step is independent of the choice of lattice Boltzmann collision steps and remains invariant. On the other hand, the collision step determines the physics of the model and can be configured by the user, by assigning configurable *dynamics* to each cell location. In this way, it is easy to implement inhomogeneous fluids which use a different type of physics from one cell to another.

This concept of per-cell dynamics is tied together in the `Dynamics` interface that is implemented by all of OpenLB's collision steps. Every particular dynamics implementation describes the various components required to model the local behavior of the assigned cell locations. Specifically, this includes the set of momenta, the equilibrium distribution as well as the actual collision operator. Fittingly, most dynamics are declared as a tuple of those three components:

```
template <typename T, typename DESCRIPTOR>
using TRTdynamics = dynamics::Tuple<
  T, DESCRIPTOR,
  momenta::BulkTuple,
  equilibria::SecondOrder,
  collision::TRT
>;
```

In this example declaration of the `TRTdynamics` we can tell at a glance that its assigned cells will expose bulk momenta reconstructed directly from the population values and relax towards the second order equilibrium using the TRT collision operator. The dynamics tuple concept is quite powerful, e.g. we may apply a forcing scheme simply by adding a single line:

```
template <typename T, typename DESCRIPTOR>
using ForcedTRTdynamics = dynamics::Tuple<
  T, DESCRIPTOR,
  momenta::BulkTuple,
  equilibria::SecondOrder,
  collision::TRT,
  forcing::Guo
>;
```

This `forcing::Guo` *combination rule* is the fourth and optional component of the dynamics tuple. Combination rules may arbitraily manipulate the previous momenta, equilibrium and collision components. In the case of Guo forcing this means that the momenta argument is shifted via `momenta::Forced` and the TRT collision operator is wrapped to force the post collision populations according to the Guo scheme.

OpenLB offers a comprehensive library of momenta, equilibria, collision operators and combination rules that can be easily combined into many different dynamics tuples. See `src/dynamics/momenta/aliases.h`, `src/dynamics/equilibrium.h`, `src/dynamics/collision.h` and `src/dynamics/forcing.h` for some examples.

As most but not all collision steps fit neatly into this framework, a `dynamics::CustomCollision` class is available and enables the combination of momenta with arbitrary collision and equilibrium implementations. One example for this are the CombinedRLBdynamics that are used as the foundation for some boundary conditions.

The full definition of the interface is available in src/dynamics/interface.h. Note that due to recent large refactoring to support execution on GPUs and SIMD CPUs, Dynamics currently contains various legacy methods that will be deprecated in future releases. New dynamics implementations should be formulated either as a dynamics::Tuple or dynamics::CustomCollision template.

6.1.4 Post Processors

While the basic concept of dynamics assigned to cells in a block lattice is 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 traversing the entire lattice a second time, applies to selected cell locations only.

While collision steps are easily parallelized due to their local nature, this doesn't hold once neighborhood access is required. For this reason, two adjacent concepts are used to group post processors: Each post processor is assigned to a *stage* and a *priority* within this stage. For example both OuterVelocityCornerProcessor3D and OuterVelocityEdgeProcessor3D are processed in the PostStream stage but the former is executed after that latter to avoid access conflicts. In turn, post processors within the same stage and priority may be executed in parallel depending on the execution platform. Note that both the number of priorities and stages may be freely customized – e.g. the free surface code introduces a number of additional stages to interleave post processing and custom communications steps.

Each post processors consists of scope and priority declarations in addition to an apply template method. For the aforementioned OuterVelocityCornerProcessor3D this is declared as follows (see src/boundary/boundaryPostProcessors3D.hh for the full implementation):

```
template<typename T, typename DESCRIPTOR,
         int xNormal, int yNormal, int zNormal>
struct OuterVelocityCornerProcessor3D {
  static constexpr OperatorScope scope = OperatorScope::PerCell;

  int getPriority() const {
    return 1;
  }

  template <typename CELL>
  void apply(CELL& cell) any_platform;
};
```

Here, the `OperatorScope::PerCell` declares that the apply function will be provided a neighborhood-enabled cell implementation for each assigned location. Other scopes such as `OperatorScope::PerBlock` (used for e.g. statistics computation) enable different access patterns. In any case, the assigned cell locations are maintained by OpenLB's post processor framework. For example

```
sLattice.addPostProcessor<PostStream>(indicator, meta::id<
    SomePerCellPostProcessor>{});
```

schedules a post processor for application to all indicated cells during the `PostStream` stage.

Note that this describes the new post processor concept adopted by OpenLB 1.5. Many existing *legacy* post processors use a different paradigm. Specifically, they are derived from `PostProcessor2D` resp. `PostProcessor3D` and override virtual methods such as `void PostProcessor2D::process(BlockLattice<T,DESCRIPTOR>&)`. All these post processors will be ported to the new approach in time. For CPU targets, legacy post processors can be used without restrictions but they are not supported on the GPU platform.

6.2 Collision Operators

A partial summary of the currently supported collision operators can be found in the recent publication on OpenLB [35]. Further, five commonly used collision schemes are compared in [29] for a typical 3D benchmark case of homogeneous isotropic turbulence. For derivations, analysis and theoretical comparisons the interested reader is referred to [37].

6.2.1 Implementation in Dynamics

As was touched upon in Section 6.1.3, local collision operators are expressed as `Dynamics` in the context of OpenLB. Specifically, the common dynamics tuple concept expresses collision operators as resusable elements alongside equilibria and momenta.

```
struct BGK {
  using parameters = typename meta::list<descriptors::OMEGA>;

  static std::string getName() {
    return "BGK";
  }

  template <typename DESCRIPTOR, typename MOMENTA, typename EQUILIBRIUM>
  struct type {
    using EquilibriumF = typename EQUILIBRIUM::template type<DESCRIPTOR,MOMENTA>;

    template <typename CELL, typename PARAMETERS, typename V=typename CELL::value_t>
    CellStatistic<V> apply(CELL& cell, PARAMETERS& parameters) any_platform {
      V fEq[DESCRIPTOR::q] { };
      const auto statistic = EquilibriumF().compute(cell, parameters, fEq);
      const V omega = parameters.template get<descriptors::OMEGA>();
      for (int iPop=0; iPop < DESCRIPTOR::q; ++iPop) {
        cell[iPop] *= V{1} - omega;
        cell[iPop] += omega * fEq[iPop];
      }
      return statistic;
    };
  };
};
```

This is the complete listing of the well known BGK collision operator that is used by many different dynamics. Each collision operator consists of three elements: A `parameters` type list of fields that are used to parametrize the collision, a `getName` method that is used to generate human readable names for dynamics tuples and a nested `type` template that contains the actual `apply` method specific to each operator.

The nested `type` template is used to enable composition into dynamics tuples and will be automatically instantiated for the required descriptor, momenta and equilibrium types. Additionally, this is used as a place for injecting partial specialization which enable usage of autogenerated CSE-optimized kernels.

As is the case for all other elements, the `apply` template method follows a fixed signature for all collision operators. Each call is provided an instance of some platform-specific implemenation of the cell concept alongside a parameters structure containing all requested values. Using these two inputs the method can perform the local collision and return the computed density and velocity magnitudes for usage in lattice statistics.

This pattern is repeated at various places of the library. Examples for other instances are equilibria and momenta elements as well as post processors. Any implementations of this style are usable on any of OpenLB's target platforms (currently this means the scalar and vectorized CPU code as well as GPU support). They are also amenable to automatic code generation.

6.3 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:

$$K = -\frac{Q\mu L}{\Delta P}, \tag{6.3}$$

where $Q = UA$ is the flow rate, U a characteristical velocity, A a cross-sectional area, μ the dynamic viscosity, L a characteristical length, ΔP 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 [13, 14], Borrvall and Petersson [10] and Pingen et al. [50], the Navier-Stokes equation is transformed (see Dornieden [21] and Stasius [57]). The discrete formulation of d describes a flow region by its permeability:

$$d = 1 - h^{dim-1}\frac{\nu_{LB}\tau_{LB}}{K} \tag{6.4}$$

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

```
#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:

```
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:

```
void setBoundaryValues(..., T physPermeability, int dim, ...){
  // d in [0,1] is a lattice-dependent porosity-value
  // depending on physical permeability K = physPermeability
  T d = converter->latticePorosity(physPermeability);
  AnalyticalConst3D<T,T> porosity(d);
  sLattice.defineField<descriptors::POROSITY>(superGeometry, 3, porosity);
  ...
}
```

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

```
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:

```
/// converts a physical permeability K to a lattice-dependent porosity d
/// (a velocity scaling factor depending on Maxwellian distribution
/// function), needs PorousBGKdynamics
T latticePorosity(T K) const
{ return 1 - pow(physLength(),getDim()-1)*getLatticeNu()*getTau()/K; }

/// converts a lattice-dependent porosity d (a velocity scaling factor
/// depending on Maxwellian distribution function) to a physical
/// permeability K, needs PorousBGKdynamics
T physPermeability(T d) const
{ return pow(physLength(),getDim()-1)*getLatticeNu()*getTau()/(1-d); }
```

6.4 Power Law Model

The two most common deviation 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

$$\mu = m\dot{\gamma}^{n-1} \,. \tag{6.5}$$

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 pawer law fluid a descriptor for dynamic `omega` must be used, such as:

```
#define DESCRIPTOR DynOmegaD2Q9Descriptor
```

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

```
AnalyticalConst2D<T,T> omega0(converter.getOmega());
sLattice.defineField<descriptors::OMEGA>(
  superGeometry.getMaterialIndicator({1,2,3,4}), omega0);
```

In the `main` function, the power law dynamics is defined:

```
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

$$\nu = \frac{1}{\rho} m\dot{\gamma}^{n-1} \,. \tag{6.6}$$

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

$$\dot{\gamma} = \sqrt{2D_{II}}\ , \tag{6.7}$$

where

$$D_{II} = \sum_{\alpha,\beta=1}^{d} E_{\alpha\beta}E_{\alpha\beta}\ , \tag{6.8}$$

where

$$E_{\alpha\beta} = -\left(1 - \frac{1}{\tau}\right)\frac{1}{2\varrho\nu}\sum_{i=0}^{q-1} f_i^h \boldsymbol{\xi}_{i\alpha}\boldsymbol{\xi}_{i\beta}\ . \tag{6.9}$$

This concept is very significant because $f_i^h\boldsymbol{\xi}_{i\alpha}\boldsymbol{\xi}_{i\beta}$ 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`

```
T computeOmega(T omega0_, T preFactor_, T rho_, T pi_[util::TensorVal
    <DESCRIPTOR >::n] );.
```

6.5 External Force

In simulations, the dynamics of a fluid are 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 additional fields in the lattice descriptor (see section 6.1.2 and lesson 7).

OpenLB implements LB dynamics with body force in `ForcedBGKdynamics`. Its algorithm is taken from Ref. [24] to guarantee second-order accuracy even when the force field is space and time dependent. Applying these `ForcedBGKdynamics` in a simulation requires that the used descriptor provides the field `descriptors::FORCE`:

```
using ForcedD2Q9Descriptor  = descriptors::D2Q9<descriptors::FORCE>;
using ForcedD3Q19Descriptor = descriptors::D3Q19<descriptors::FORCE>;
```

An example for the implementation of a LB simulation with force term is found in the code `examples/laminar/poiseuille2d`. As an alternative, the velocity shift forcing scheme developed by Shan and Chen [53] and improved by Shan and Doolen [54] is also implemented and can be accessed using `ForcedShanChenBGKdynamics`.

6.6 Multiphysics Couplings

6.6.1 Shan-Chen Model

For the simulation of both multiphase and multicomponent flow the Shan-Chen model is implemented in OpenLB. Since its first introduction [53], many variants of the model have been developed. In this implementation, there are several forcing schemes [24, 54] and interaction potentials to choose from.

6.6.2 Implementation of Shan-Chen Two-phase Fluid

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

```
SuperLattice<T, DESCRIPTOR> sLattice(superGeometry);
```

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

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

Possible choices for the dynamics are `ForcedBGKdynamics` and `ForcedShanChenBGKdynamics`.

Then the interaction potential is chosen:

```
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.

```
const T G      = -120.;
ShanChenForcedSingleComponentGenerator3D<T,DESCRIPTOR> coupling(
 G,rho0,interactionPotential);
sLattice.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):

```
sLattice.communicate();
sLattice.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`.

6.6.3 Implementation of Shan-Chen Two-component Fluid

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

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

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

```
ForcedShanChenBGKdynamics<T, DESCRIPTOR> bulkDynamics1 (
omega1, instances::getExternalVelocityMomenta<T,DESCRIPTOR>() );
ForcedShanChenBGKdynamics<T, DESCRIPTOR> bulkDynamics2 (
omega2, 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:

```
PsiEqualsRho<T,T> interactionPotential;
```

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.

```
const T G       = 3.;
ShanChenForcedGenerator3D<T,DESCRIPTOR> coupling(
 G,rho0,interactionPotential);
sLatticeOne.addLatticeCoupling(superGeometry, 1, coupling,
    sLatticeTwo);
sLatticeOne.addLatticeCoupling(superGeometry, 2, coupling,
    sLatticeTwo);
```

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:

```
sLatticeOne.communicate();
sLatticeTwo.communicate();
sLatticeOne.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-component flow are `examples/multiComponent2d` and `examples/multiComponent3d`.

6.6.4 Free Energy Model

As an alternative option for simulating multicomponent flow, the free energy model has been implemented into OpenLB, and may be used for either two or three fluid components. Examples for the binary case are given in `youngLaplaceXd` and `contactAngleXd`, while an example of the ternary case with boundaries is provided in `microFluidics2d`. These are all contained within the `multiComponent` floder

The approach taken in OpenLB is similar to that given in [52] and assumes equal densities and viscosities for each of the fluids. In the next sections the method will be outlined briefly for three components. The two component case is identical to taking the third fluid component to be zero and instead only uses two lattices.

6.6.5 The Bulk Free Energy Model

Three lattices are required to track the density, ρ, and order parameters, ϕ, and ψ. These are related to the individual component densities, C_i, by,

$$\rho = C_1 + C_2 + C_3, \qquad \phi = C_1 - C_2, \qquad \psi = C_3. \tag{6.10}$$

By considering the free energy, a force is derived to drive the fluid towards thermodynamic equilibrium. The density therefore obeys the Navier–Stokes equation with this added force. While, the equation of motion for the order parameters is the Cahn–Hilliard equation. The dynamics chosen for the first lattice must therefore include an external force, such as `ForcedBGKdynamics`, while for the second and third lattices `FreeEnergyBGKdynamics` is required.

```
ForcedBGKdynamics<T, DESCRIPTOR> bulkDynamics1 (
    omega, instances::getBulkMomenta<T,DESCRIPTOR>() );
FreeEnergyBGKdynamics<T, DESCRIPTOR> bulkDynamics23 (
    omega, gamma, instances::getBulkMomenta<T,DESCRIPTOR>() );
```

To compute the force, two lattice couplings are required. The first computes the chemical potentials for each lattice using the equations,

$$\mu_\rho = A_1 + A_2 + \frac{\alpha^2}{4}\left[(\kappa_1+\kappa_2)\left(\nabla^2\psi - \nabla^2\rho\right) + (\kappa_2-\kappa_1)\nabla^2\phi\right], \tag{6.11}$$

$$\mu_\phi = A_1 - A_2 + \frac{\alpha^2}{4}\left[(\kappa_2-\kappa_1)\left(\nabla^2\rho - \nabla^2\psi\right) - (\kappa_1+\kappa_2)\nabla^2\phi\right], \tag{6.12}$$

$$\begin{aligned}\mu_\psi = &- A_1 - A_2 + \kappa_3\psi(\psi-1)(2\psi-1) + \frac{\alpha^2}{4}\Big[(\kappa_1+\kappa_2)\nabla^2\rho \\ &- (\kappa_2-\kappa_1)\nabla^2\phi - (\kappa_1+\kappa_2+4\kappa_3)\nabla^2\psi\Big],\end{aligned} \tag{6.13}$$

where A_1 and A_2 are defined as

$$\begin{aligned}A_1 &= \frac{\kappa_1}{8}(\rho+\phi-\psi)(\rho+\phi-\psi-2)(\rho+\phi-\psi-1),\\ A_2 &= \frac{\kappa_2}{8}(\rho-\phi-\psi)(\rho-\phi-\psi-2)(\rho-\phi-\psi-1).\end{aligned}$$

The α the κ parameters are input parameters for the lattice coupling and can be used to tune the interfacial width and surface tensions. The interfacial width is given by α and the surface tensions are $\gamma_{mn} = \alpha(\kappa_m + \kappa_n)/6$.

The second lattice coupling then computes the force using,

$$\boldsymbol{F} = -\rho\boldsymbol{\nabla}\mu_\rho - \phi\boldsymbol{\nabla}\mu_\phi - \psi\boldsymbol{\nabla}\mu_\psi. \tag{6.14}$$

This depends non-locally upon the result of the first lattice coupling, and so the chemical potential must be communicated inbetween. To accomodate this, the first coupling is assigned to the ρ lattice and the force coupling is assigned to the ϕ lattice:

```
FreeEnergyChemicalPotentialGenerator3D<T, DESCRIPTOR> coupling1(
    alpha, kappa1, kappa2, kappa3 );
FreeEnergyForceGenerator3D<T, DESCRIPTOR> coupling2;

sLattice1.addLatticeCoupling<DESCRIPTOR>(
    superGeometry, 1, coupling2, {&sLattice2, &sLattice3} );
sLattice2.addLatticeCoupling<DESCRIPTOR>(
    superGeometry, 1, coupling3, {&sLattice1, &sLattice3} );
```

The following is then used in the main loop to calculate the force at each timestep:

```
sLattice1.executeCoupling();
sExternal1.communicate();
sExternal2.communicate();
sExternal3.communicate();
sLattice2.executeCoupling();
```

6.6.6 Boundaries in the Free Energy Model

Bounce-back wall boundaries with controllable contact angles may be added using the `setFreeEnergyWallBoundary` function:

```
setFreeEnergyWallBoundary<T,DESCRIPTOR>(sLattice1, superGeometry, 2,
    alpha, kappa1, kappa2, kappa3, h1, h2, h3, 1);
setFreeEnergyWallBoundary<T,DESCRIPTOR>(sLattice2, superGeometry, 2,
    alpha, kappa1, kappa2, kappa3, h1, h2, h3, 2);
setFreeEnergyWallBoundary<T,DESCRIPTOR>(sLattice3, superGeometry, 2,
    alpha, kappa1, kappa2, kappa3, h1, h2, h3, 3);
```

The final parameter, `latticeNumber`, is necessary to change each lattice differently. While the h_i parameters are related to the contact angles at the boundary. The contact angles, θ_{mn}, are given by the following relation,

$$\cos\theta_{mn} = \frac{(\alpha\kappa_n + 4h_n)^{3/2} - (\alpha\kappa_n - 4h_n)^{3/2}}{2(\kappa_m + \kappa_n)\sqrt{\alpha\kappa_n}} - \frac{(\alpha\kappa_m + 4h_m)^{3/2} - (\alpha\kappa_m - 4h_m)^{3/2}}{2(\kappa_m + \kappa_n)\sqrt{\alpha\kappa_m}}. \tag{6.15}$$

Notably, to set neutal wetting (90° angles) the values can be set to $h_i = 0$.

A demonstration of using these solid boundaries for a binary fluid case is provided in the `contactAngleXd` examples. This example compares the simulated angles to those given by equation 6.15.

Open boundary conditions can also be implented using the `setFreeEnergyInletBoundary` and `setFreeEnergyOutletBoundary` functions. These can be used to specify constant density or velocity boundaries. The first lattice is used to define the density or velocity boundary condition, while on the second and third lattices ϕ and ψ must instead be defined. For example, to set a constant velocity inlet:

```
setFreeEnergyInletBoundary(
    sLattice1, omega, inletIndicator, "velocity", 1 );
setFreeEnergyInletBoundary(
    sLattice2, omega, inletIndicator, "velocity", 2 );
setFreeEnergyInletBoundary(
    sLattice3, omega, inletIndicator, "velocity", 3 );

sLattice1.defineU( inletIndicator, 0.002 );
sLattice2.defineRho( inletIndicator, 1. );
sLattice3.defineRho( inletIndicator, 0. );
```

However, this alone is insufficient to set a constant density outlet because ρ, ϕ, and ψ are redefined by a convective boundary condition on each timestep. In this case an additional lattice coupling is required, using `FreeEnergyDensityOutletGeneratorXD`.

There are two additional requirements for open boundaries. The first is that the velocity must be coupled between the lattices using `FreeEnergyInletOutletGeneratorXD` because this is required for the collision step. The second is that the communication of the external field must now include two values. This ensures that ρ, ϕ, and ψ are properly set on block edges at the outlet. To see a full example of applying these boundary conditions see the `microFluidics2d` example.

6.6.7 Coupling Between Momentum and Energy Equations

As explained in reference [46], 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 6.16.

$$v_{shift} = v + \frac{F}{2} \tag{6.16}$$

The code snipped responsible for this shift is defined in the collision function of the file

`/dynamics/dynamics.hh` for the class ForcedBGKdynamics:

```
this->momenta . computeRhoU( cell , rho , u) ;
FieldPtr<T,DESCRIPTOR,FORCE> force = cell.template getFieldPointer<
    descriptors::FORCE>();
for ( int iVe l=0; iVel<DESCRIPTOR >::d; ++iVel ) {
  u[iVel] += force[iVel] / T{2};
}
```

Listing 6.2: 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:

```

lbm< Lattice >::addExternalForce( cell , u , omega )
```

This function follows equation 6.17, where $\tilde{f}_i$ represents the new distribution function (see reference [24] for BGK model and [22] for MRT model).

$$\bar{f}_i = f_i + (1 - \frac{\omega}{2}\{\frac{e_i - v}{c_s^2} + \frac{e_i v}{c_s^4} e_i\}\omega_i F \tag{6.17}$$

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

```
auto u = cell.template getFieldPointer<descriptors::VELOCITY>();
```

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.

```

auto u = tPartner->get(iX, iY).template getFieldPointer<descriptors::
    VELOCITY>();
blockLattice.get(iX, iY).computeU(u);
```

Listing 6.3: Velocity coupling

The new force is computed from equation 6.18, following the Boussinesq approximation:

$$F = g\rho\frac{T - T_0}{\Delta T} \tag{6.18}$$

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

```

auto force = blockLattice.get(iX, iY).template getFieldPointer<descriptors::FORCE>();
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 6.4: Computation of the Boussinesq force

6.7 Advection Diffusion Equation

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

$$\frac{\partial c}{\partial t} = \nabla \cdot (D \nabla c) - \nabla \cdot (\vec{v} c), \tag{6.19}$$

where c is the considered physical quantity (temperature, particle density), D is the diffusion coefficient and 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 [45]

$$g_i^{eq} = w_i \rho \left(1 + \frac{c_i \cdot \vec{v}}{c_s^2} \right), \tag{6.20}$$

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:

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

Listing 6.5: advection diffusion dynamics object

Additionally, a different descriptor with fewer lattice velocities is used [27]:

```
#define DESCRIPTOR AdvectionDiffusionD3Q7Descriptor
```

Listing 6.6: 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.

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

Listing 6.7: advection diffusion dynamics object

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

```
void prepareLattice(...) {
...
/// Material=3 -> boundary with constant temerature
setAdvectionDiffusionTemperatureBoundary<T,ADDESCRIPTOR>(
      sLatticeAD, omegaAD, superGeometry, 3);
...
}
```

Listing 6.8: advection diffusion descriptor

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

```
T velocity[3] = {vx,vy,vz};
...
cell.defineField<descriptors::VELOCITY>(velocity);
```

Listing 6.9: add advective velocity on a cell

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

```
AnalyticalConst3D<T,T> velocity(vel);
...
/// sets advective velocity for material 1
superLattice.defineField<descriptors::Velocity>(superGeometry, 1,
    velocity);
```

Listing 6.10: add advective velocity on a superlattice

Here, `vel` is a `std::vector<T>`.

6.7.1 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 `/boundary/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.

```
int missingNormal = 0 ;
constexpr auto 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 6.11: Collision step for a temperature boundary

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

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

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 temperature = this->momenta.computeRho( cell );
cell[missingNormal] = temperature - sum -(T)1;
```

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

```
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_{wall} at the wall is known, the value of the unknwon distribution is:

$$T_{wall} = \sum_{i=0}^{4} g_i \rightarrow g_3 = T_{wall} - (g_0 + g_1 + g_2 + g_4) \tag{6.21}$$

Adiabatic Boundary Condition

Additionally, thanks to the simplied 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 [43]).

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 too 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:

$$g_4 = g_2 \tag{6.22}$$

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

$$T_{wall} = \sum_{i=0}^{4} g_i \tag{6.23}$$

6.7.2 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 `NSlattice` or the `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 (*eps*). The listing6.12 shows how the object is defined in the main function, and how its value is updated and checked at each time step.

```
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 6.12: Check convergence

6.7.3 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 [62]), 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. $ForcedD3Q19Descriptor$, and another one for the temperature field, e.g. $AdvectionDiffusionD3Q7Descriptor$.

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:

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

Listing 6.13: 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:

```
setAdvectionDiffusionTemperatureBoundary<T,TDESCRIPTOR>(
     ADlattice, Tomega, superGeometry, 2);
```

Listing 6.14: 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:

```
NSlattice.defineField<descriptors::FORCE>( superGeometry, 1,  force )
    ;
```

Listing 6.15: 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 solving the AD equation, the temperature replaces the density variable), as well as the distribution functions corresponding to this temperature value:

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

Listing 6.16: Initialization of the temperature field

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

```
std::vector<T> zero ( 2, T( ) );
ConstAnalyticalF2D<T,T> velocity ( zero );
ADlattice.defineField<descriptors::VELOCIY>(superGeoemtry, 1,velocity
    );
```

Last step is to make the lattice ready for simulation:

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

Listing 6.17: 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.

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

Listing 6.18: 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 6.7.4.

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.

6.7.4 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:

$$Ra = \frac{g\beta}{\nu\alpha}\Delta T L^3 \tag{6.24}$$

where g is the acceleration due to gravity, β is the thermal expansion coeffcient, ν is the kinematic viscosity, α is the thermal diffusivity, ΔT is the temperature difference, and L the characteristic length.

The Prandtl number is defined as the ratio of momentum diffusivity ν to thermal diffusivity α ("Prandtl number," `Wikipedia: The Free Encyclopedia`):

$$Pr = \frac{\nu}{\alpha} \tag{6.25}$$

The differences between the converter objects for isothermal and thermal simulations (see Section 6 for more details) make it sometimes diffcult 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 them were implemented by modifying existing functors in the files
`/functors/lattice/blockLatticeLocalF3D`,
`/functors/lattice/blockLatticeIntegralF3D`,
`/functors/lattice/superLatticeLocalF3D`,
`/functors/lattice/superLatticeIntegralF3D`.

Velocity

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

$$v_{res} = \frac{v_{lattice}}{latticeU}\sqrt{Ra \cdot Pr} = \frac{v_{lattice}}{\delta t \cdot N}\sqrt{Ra \cdot Pr} \tag{6.26}$$

The lattice velocity `latticeU` is obtained from the function `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:

$$p_{phys} = p_{lattice}\frac{physcForce}{physL^{d-1}} = p_{lattice}\frac{physL^{d+1}}{physT^2}\frac{1}{physL^{d-1}} \tag{6.27}$$

$$= p_{lattice}(\frac{physL}{physT})^2 = p_{lattice}(\frac{charU}{latticeU})^2 = p_{lattice}\frac{Ra \cdot Pr}{latticeU^2} \tag{6.28}$$

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

$$p_{lattice} = \frac{\rho - 1}{3} \tag{6.29}$$

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

$$F_{phys} = F_{lattice}\frac{physL^{d+1}}{physT^2} = F_{lattice}\frac{(charL\frac{latticeL}{charL})^{d+1}}{(\frac{charL}{charU}\frac{latticeU \cdot latticeL}{charL})^2} \tag{6.30}$$

$$= F_{lattice} \cdot latticeL^{d-1}(\frac{charU}{latticeU})^2 = F_{lattice}\frac{latticeL^{d-1}}{latticeU^2}Ra \cdot Pr \tag{6.31}$$

d is the number of dimensions in the problem.

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

$$C_{F_i} = F_{i,phys}\frac{1}{\frac{1}{2}charU^2 \cdot count_i \cdot latticeL^{d-1}} = F_{i,lattice}\frac{1}{\frac{1}{2}LatticeU^2 \cdot count_i} \tag{6.32}$$

where $count_i$ is the number of cells in the surface perpendicular to the direction i of the force coeffcient computed.

6.7.5 Conduction Problems

For heat conduction problems there is no velocity field that advects the temperature (see [44] and [32]). In absence of convection, radiation and heat generation, the energy equation for a homogeneous medium is given by:

$$\frac{\partial T}{\partial t} = \alpha \Delta T \tag{6.33}$$

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:

$$g_i^{eq} = \omega_i \rho = \omega_i T \tag{6.34}$$

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:

$$\frac{\partial T}{\partial t} + v \nabla T = \alpha \Delta T \tag{6.35}$$

where α is the thermal diffusivity coeffcient.

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 [38]. 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 [42]. The temperature field distribution functions g_i are governed by the following equation:

$$g(x + e\delta_t, t + \delta_t) - g(x, t) = -N^{-1}\theta[n(x, t) - n^{(eq)}(x, t)] \tag{6.36}$$

where g and n are column vectors including the distribution functions and the moments respectively. N is an orthogonal transformation matrix, and θ is a non-negative, diagonal relaxation matrix.

The macroscopic temperature T is calculated by:

$$T = \sum_{i=0}^{4} g_i \tag{6.37}$$

The weight coeffcients for each lattice direction are given in equation 6.38.

$$\omega_i = \begin{cases} \frac{3}{5}, & \text{if } i = 0, \\ \frac{1}{10}, & \text{if } i = 1, 2, 3, 4. \end{cases} \tag{6.38}$$

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

$$n = N \cdot g \tag{6.39}$$

The transformation matrix N and its inverse matrix N^{-1} are shown in equations 6.40 and 6.41. There are some differences in the order of the columns respect to what is specified in the reference [42]. This is due to the different sequence used in numbering the velocity directions.

$$N = \begin{pmatrix} <1| \\ <e_x| \\ <e_y| \\ <5e^2-4| \\ <e_x^2-e_y^2| \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ -4 & 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & 1 & -1 \end{pmatrix} \tag{6.40}$$

$$A_{m,n} = \begin{pmatrix} \frac{1}{5} & 0 & 0 & -\frac{1}{5} & 0 \\ \frac{1}{5} & -\frac{1}{2} & 0 & \frac{1}{20} & \frac{1}{4} \\ \frac{1}{5} & 0 & -\frac{1}{2} & \frac{1}{20} & -\frac{1}{4} \\ \frac{1}{5} & \frac{1}{2} & 0 & \frac{1}{20} & -\frac{1}{4} \\ \frac{1}{5} & 0 & \frac{1}{2} & \frac{1}{20} & -\frac{1}{4} \end{pmatrix} \tag{6.41}$$

The equilibrium moments $n^{(eq)}$ are defined as:

$$n^{(eq)} = \{T, uT, vT, \varpi T, 0\}^T \tag{6.42}$$

ϖ is a constant of the D2Q5 model, which we set to -2.

The diagonal relaxation matrix θ is:

$$\theta = diag\{0, \zeta_a, \zeta_a, \zeta_e, \zeta_\nu\} \tag{6.43}$$

The first relaxation rate, corresponding to the temperature, is set to zero for simplicity, since the first moment is conserved. The relaxation rates ζ_e and ζ_ν are set to 1.5, whereas the relaxation rates ζ_a are a function of the thermal diffusivity α (6.44). The sound speed of the D2Q5 model c_s^2 is $1/5$.

$$\alpha = c_s^2(\tau_a - \frac{1}{2}) = \frac{1}{5}(\tau_a - \frac{1}{2}) \rightarrow \zeta_a = \frac{1}{\tau_a} = \frac{1}{5\alpha + \frac{1}{2}} \tag{6.44}$$

Particle Flows as Advection Diffusion Problem

The quantity c in the Advection–Diffusion equation can be considered as particle density, thereby giving an continuous ansatz for simulating particle flows. To solve for the particle distribution an additional lattice is required with an appropriate descriptor and dynamics, which are only implemented for the 3D case.

```
#define ADDESCRIPTOR ParticleAdvectionDiffusionD3Q7Descriptor
```

Listing 6.19: Advection–Diffusion descriptor for particle flows

The descriptor in Listing 6.19 allocates additional memory since for the computation of the particle velocity also the velocity of the last time step hast to be stored. This calculations also are non-local, therefore the communication of the additional data has to be ensured by an additional object, which is constructed according to Line 1 of Listing 6.20 and communicates the data by a function as shown in Line 2 of the Listing, which has to be called in the time loop.

```
1 SuperExternal3D<T, ADDESCRIPTOR, descriptors::VELOCITY> sExternal(
     superGeometry, sLatticeADE, sLatticeAD.getOverlap() );
2 sExternal.communicate();
```

Listing 6.20: 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 6.21, and some of the boundary conditions to take the diffusion coefficiant into account. Therefore a new ω_{ADE} is computed by

$$\omega_{ADE} = \left(4D \frac{U_L}{L_L U_C} + 0.5 \right)^{-1}, \tag{6.45}$$

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

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

Listing 6.21: 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 6.22, forces acting on the particle can be added like the Stokes drag force as shown in Line 2 and 3 of Listing 6.22. The implementation of new forces is straight forward, since only a new class which provides a function `applyForce(...)`, computing the force in a cell, needs to be written analgously to the existing `advDiffDragForce3D`. Finally the lattices are linked by Line 4 of Listing 6.22, which needs to be applied to the Navier–Stokes lattice for reasons of accessability.

```
AdvectionDiffusionParticleCouplingGenerator3D<T,NSDESCRIPTOR> coupling( ADDESCRIPTOR::index<descriptors::VELOCITY>());
advDiffDragForce3D<T, NSDESCRIPTOR> dragForce( converter,radius, partRho );
coupling.addForce( dragForce );
sLatticeNS.addLatticeCoupling( superGeometry, 1, coupling, sLatticeAD );
```

Listing 6.22: 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 6.23 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. [58] as well as in the examples section.

```
setExtFieldBoundary<T,ADDESCRIPTOR,descriptors::VELOCITY,descriptors::VELOCITY2>(
    sLatticeAD, superGeometry.getMaterialIndicator({2, 3, 4, 5, 6}));
```

Listing 6.23: Example of a boundary condition for the particle velocity for particle flow simulations

7 Particles

The following chapter summarizes OpenLB's functionality regarding the consideration of discrete particles in a Lagrangian framework. This includes both *sub-grid* particles assuming spherical shapes and *surface resolved* particles with arbitrary shapes. The current implementation is divided into a fully overhauled *new particle framework* (c.f. Sec. 7.1), which comprises all surface resolved functionality and the original *sub-grid (legacy) framework* (c.f. Sec. 7.2), which is limited to 3D only. As the new framework follows advances in the data concept of the lattice (c.f. Sec. 6), it provides a dimension agnostic, flexible and easily extendable implementation. While abstract template meta functionality characterizes the data handling level, accessible high-level user-functions are provided for e.g. particle creation or coupling handling. Due to those properties, the *sub-grid framework* will be included into the *new framework* in future OpenLB versions as well and is therefore tagged as *legacy code* in the current release.

7.1 New Particle Framework

To get a good overview of the new particle framework, the code of the example *settlingCube3d* (Sec. 13.5.4) is reviewed, focussing on the simulation of particles. After concentrating on the example, the file structure of the new particle framework will be explained. The example examines the settling of a cubical silica particle under the influence of gravity in surrounding water. For the first overview, the flow-chart of the example can be seen in figure 7.1.

It starts with integrating some libraries and namespaces (lines $1-13$, Listing 7.1), followed by the definition of different types (lines $15-19$, Listing 7.1), e.g. the *descriptor* and the *particle type* (in this case the new resolved particle system). Afterwards, some variables are set to a concrete value, used in the fluid and particle calculation (lines $1-14$, Listing 7.2). Particle settings include all the data to solve the equations of motion, such as the particle's starting position and density.

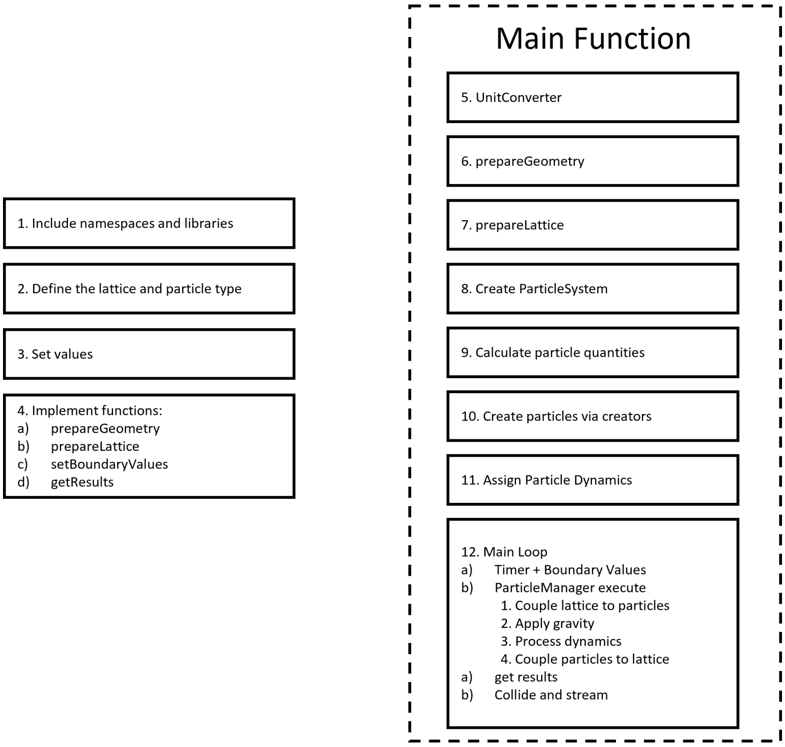

1. Include namespaces and libraries
2. Define the lattice and particle type
3. Set values
4. Implement functions:
a) prepareGeometry
b) prepareLattice
c) setBoundaryValues
d) getResults
Main Function
5. UnitConverter
6. prepareGeometry
7. prepareLattice
8. Create ParticleSystem
9. Calculate particle quantities
10. Create particles via creators
11. Assign Particle Dynamics
12. Main Loop
a) Timer + Boundary Values
b) ParticleManager execute
1. Couple lattice to particles
2. Apply gravity
3. Process dynamics
4. Couple particles to lattice
a) get results
b) Collide and stream

Figure 7.1: flow-chart of the example *settlingCube3d*

```
#include "olb3D.h"
#include "olb3D.hh"

using namespace olb;
using namespace olb::descriptors;
using namespace olb::graphics;
using namespace olb::util;
using namespace olb::particles;

typedef double T;
typedef D3Q19<POROSITY,VELOCITY_NUMERATOR,VELOCITY_DENOMINATOR> DESCRIPTOR;

//Define particleType
typedef ResolvedParticle3D PARTICLETYPE;
```

Listing 7.1: Following the new particle system with following the example settlingCube3d

```
//Particle Settings
T centerX = lengthX*.5;
T centerY = lengthY*.5;
T centerZ = lengthZ*.9;
T const cubeDensity = 2500;
T const cubeEdgeLength = 0.0025;
Vector<T,3> cubeCenter = {centerX,centerY,centerZ};
Vector<T,3> cubeOrientation = {0.,15.,0.};
Vector<T,3> cubeVelocity = {0.,0.,0.};
Vector<T,3> externalAcceleration = {.0, .0, -9.81 * (1. - physDensity
    / cubeDensity)};

// Characteristic Quantities
T const charPhysLength = lengthX;
T const charPhysVelocity = 0.15;    // Assumed maximal velocity
```

Listing 7.2: Particle settings in example settlingCube3d

Like other simulations, particle flow simulations need basic, non particle-specific functions like *prepareGeometry* or *prepareLattice*. After initializing those functions, the main function starts. The main section begins with initialization of physical units in the unit converter, which is explained in the Q&A in Sec. 14. The unit converter is followed by the preparation of the geometry using the *prepareGeometry*-function and afterwards the *prepareLattice*-function. After those general simulation functions, the particle simulation starts. First, the ParticleSystem (line 2, Listing 7.3, explained in Sec. 7.1.1) is called followed by the calculation of the particle quantities like a smoothing factor and the extent of the particles. After those calculations, the particles are created, which happens in lines $13 - 20$, in Listing 7.3. In the following lines, dynamics are assigned to the particles (lines $24 - 26$, Listing 7.3).

```
1  // Create ParticleSystem
2  ParticleSystem<T,PARTICLETYPE> particleSystem;
3
4  //Create particle manager handling coupling, gravity and particle
       dynamics
5  ParticleManager<T,DESCRIPTOR,PARTICLETYPE> particleManager(
6    particleSystem, superGeometry, sLattice, converter,
         externalAcceleration);
7
8  // Calculate particle quantities
9  T epsilon = 0.5*converter.getConversionFactorLength();
10 Vector<T,3> cubeExtend( cubeEdgeLength );
11
```

```
// Create Particle 1
creators::addResolvedCuboid3D( particleSystem, cubeCenter,
  cubeExtend, epsilon, cubeDensity, cubeOrientation );

// Create Particle 2
cubeCenter = {centerX,lengthY*0.51,lengthZ*.7};
cubeOrientation = {0.,0.,15.};
creators::addResolvedCuboid3D( particleSystem, cubeCenter,
cubeExtend, epsilon, cubeDensity, cubeOrientation );

// Create and assign resolved particle dynamics
VerletParticleDynamics<T,PARTICLETYPE> particleDynamics(converter.
    getPhysDeltaT() );
for (std::size_t iP=0; iP<particleSystem.size(); ++iP){
  particleSystem.defineDynamics( iP, &particleDynamics );
}
```

Listing 7.3: Creation of particles and assigning dynamics

Before the main loop starts in line 10, Listing 7.4, we create a timer in line 2, Listing 7.4 and set initial values to the distribution functions by calling *setBoundaryValues* in lines $6-7$, Listing 7.4. After this, the following is processed at every time step. The fluid's influence on the particles is calculated by evaluating hydrodynamic forces acting on the particle surface (line 14, Listing 7.4). Afterwards, an external acceleration, e.g. gravity, is applied onto the particles (lines 15, Listing 7.4) and the equations of motion are solved for each one in (line 16, Listing 7.4). The back coupling from the particles to the fluid follows in line 17, Listing 7.4. Finally, the main loop ends with the *getResults*-function (line 21, Listing 7.4), which prints the results to the console and writes VTK data for post-processing with ParaView (Sec. 9) at previously defined time intervals.

```
1  /// === 4th Step: Main Loop with Timer ===
2  Timer<double> timer(converter.getLatticeTime(maxPhysT),
3  superGeometry.getStatistics().getNvoxel());
4  timer.start();
5
6  /// === 5th Step: Definition of Initial and Boundary Conditions ===
7  setBoundaryValues(sLattice, converter, 0, superGeometry);
8
9  clout << "MaxIT: " << converter.getLatticeTime(maxPhysT) << std::endl
       ;
10 for (std::size_t iT = 0; iT < converter.getLatticeTime(maxPhysT)+10;
       ++iT) {
11
12 // Execute particle manager
particleManager.execute<
  couple_lattice_to_particles<T,DESCRIPTOR,PARTICLETYPE>,
  apply_gravity<T,PARTICLETYPE>,
  process_dynamics<T,PARTICLETYPE>,
  couple_particles_to_lattice<T,DESCRIPTOR,PARTICLETYPE>
>();

// Get Results
getResults(sLattice, converter, iT, superGeometry, timer,
    particleSystem );
// Collide and stream
sLattice.collideAndStream();
}
```

Listing 7.4: Main Loop with Timer

As we followed the example for particle simulation *settlingCube3d*, some functions necessary for the simulations were introduced. Therefore, in the next chapters, the underlying file-structure is examined.

As displayed in figure 7.2, particle related files can be found in the source folder (src), in the sub-folder particles. The new framework is divided into files and folders: ParticleSystem, ParticleManager, descriptor, dynamics, functions, resolved. In the following, the main subjects of those folders (black) / files (red) will be discussed.

7.1.1 Class ParticleSystem

The ParticleSystem stores all data concerning the particles in containers. Therefore, the class is used multiple times in a particle simulation. First, the ParticleSystem is created according to the desired ***PARTICLETYPE*** (lines $1-2$, Listing 7.3). However, the container of particles is empty. Therefore, we add two particles to it using creator functions in lines $9-10$ and lines $15-16$, Listing 7.3 and add dynamics via the ParticleSystem in lines $19-22$, Listing 7.3. Additionally, it is utilized in the ParticleManager (c.f. Sec. 7.1.2) to access the particles and perform predefined operations on them.

One focus of the new particle system is the separation of data and operations according to the lattice framework (c.f. Sec. 6). Therefore, only the data is stored in the ParticleSystem. For the operations, it is non-relevant and only used to store data of the calculations.

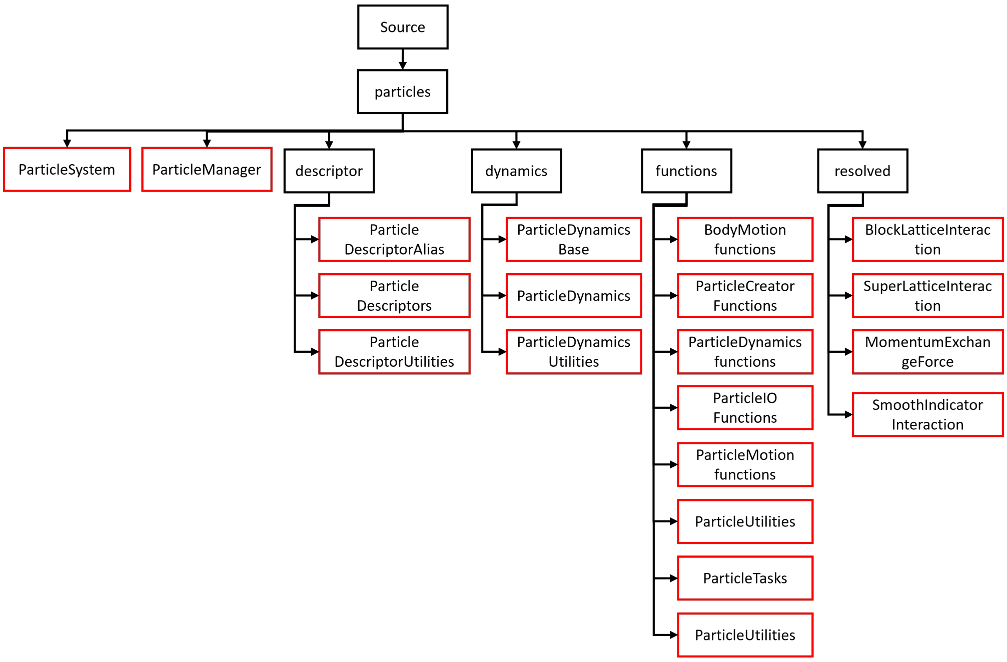


Figure 7.2: file-structure of the new particle framework

7.1.2 Class ParticleManager

The `ParticleManager` can be used to encapsulate relevant reoccurring particle tasks as e.g. the particle-lattice-coupling. After its initial instantiation by providing the access to relevant particle, lattice and set-up specific data, its *execute()* method can be called with the respective tasks specified as template arguments in the desired order. The individual tasks (included in *particleTasks.h*) provide an *execute()* method as well and a parameter set specifying the couling type and the potential embedding into a loop over all available particles. The `ParticleManager` also takes care of combining respective tasks into a single particle loop.

7.1.3 Folder resolved

In the directory *resolved*, all surface resolved specific functionality is bundled. The *blockLatticeInteraction.h* (only header file) and *blockLatticeInteraction.hh* files consist of five functions. All of those functions are needed to calculate and check the position of the particles inside the geometry. For example, the *checkSmoothIndicatorOutOfGeometry*-function checks if every part of the particle is inside the cell barrier. If the particle is partly outside of the geometry, the position needs to be changed. Another important functions is the *setBlockParticleField*, where a all cells, which are inside the particle are set as a particle field. Similar to the *blockLatticeInteraction.hh* also the *superLatticeInteraction.hh* exist. In this file the *setBlockParticleField* gets converted to the lattice structure with the function *setSuperParticleField*. The file *momentumExchangeForce.h* provides functions to calculate hydrodynamic forces on the particle's surface via an adapted momentum exchange algorithm. The file (*smoothIndicatorInteraction.h*) is needed for the simulation of the area directly at the surface of the particles.

7.1.4 Folder descriptors

The first file is the *particleDescriptorAlias.h* file, in which the alias' are given to different types of **`PARTICLETYPE`**s. In the example *settlingCube3d*, right in the beginning, the **`PARTICLETYPE`** *ResolvedParticle3D* is chosen. After the choice of this alias, the dynamics and other main properties are set. Other possible particletypes that can be chosen are *ResolvedParticle2D* or *ResolvedSphere3D*.

7.1.5 Folder dynamics

Another important part of the new particle system are the dynamics. The files are used to define those properties for the chosen particle type. For example, in the *particleDynamics.h* and *particleDynamics.hh* all dynamic functions for the particle type are implemented. Therefore, all information for calculation of dynamic values can be found here e.g. acceleration or angular acceleration. Those functions get called in the main part of simulations, e.g. in the example *settlingCube3d* (`VerletParticleDynamics`), in lines $19-23$, in Listing 7.3, as the dynamics are assigned to the particle type.

7.1.6 Folder functions

In the functions-directory, additional free functions are defined. These functions are callable anywhere in the code. The first set of files including *particleCreatorFunctions.h*, *particleCreatorFunctions2D.h*, *particleCreatorFunctions3D.h* and *particleCreatorHelperFunctions.h* concentrate on functions concerning the creation of particles with different types of surface structures. This functions are therefore called first to create particles in the desired shape. In the example *settlingCube3d*, the function *addResolvedCuboid* is called in lines $10-11$ of Listing 7.3 and creates a particle in the shape of a cuboid. Also other geometries, like circles in 2D or cylinders in 3D, can be created. All of those functions are implemented in these files.

The file (*particleMotionfunctions.h*) concentrates on the main algorithms for solving the equations of motion. Two functions exists, using different integration-types, velocity Verlet algorithm (*velocityVerletIntegration*) or Euler-Integration (*eulerIntegration*). The former function is used in the `VerletParticleDynamics` class (chapter 7.1.5) and are therefore called in the main part of the example (lines $19 - 23$, Listing 7.3). Other functions used in the `VerletParticleDynamics` (chapter 7.1.5) are the *rotationMatrix*-functions (Listing 7.5), which are implemented in the *particleDynamicsFunctions.h*. Often two functions for the same calculation exist as they need to match the dimension of a problem. Those are differentiated by partial template specialization. While the first function (lines $1 - 18$, Listing 7.5) is used for 2D-simulations, the second function (lines $20 - 45$, Listing 7.5) is used for 3D-simulations.

```
1  template<typename T>
2  struct rotation_matrix<2,T>{
3          static constexpr Vector<T,4> calculate( Vector<T,1> angle )
4          {
5          Vector<T,4> rotationMatrix;
6  
7          T cos = util::cos(angle[0]);
8          T sin = util::sin(angle[0]);
9  
10         rotationMatrix[0] = cos;
11         rotationMatrix[1] = sin;
12         rotationMatrix[2] = -sin;
13         rotationMatrix[3] = cos;
14 
15         return rotationMatrix;
16 
17         }
18 };
19 
20 template<typename T>
21 struct rotation_matrix<3,T>{
22         static constexpr Vector<T,9> calculate( Vector<T,3> angle )
23         {
24         Vector<T,9> rotationMatrix;
25 
26         T cos0 = util::cos(angle[0]);
27         T cos1 = util::cos(angle[1]);
28         T cos2 = util::cos(angle[2]);
29         T sin0 = util::sin(angle[0]);
30         T sin1 = util::sin(angle[1]);
31         T sin2 = util::sin(angle[2]);
32 
```

```
       rotationMatrix[0] = cos1*cos2;
       rotationMatrix[1] = sin0*sin1*cos2 - cos0*sin2;
       rotationMatrix[2] = cos0*sin1*cos2 + sin0*sin2;
       rotationMatrix[3] = cos1*sin2;
       rotationMatrix[4] = sin0*sin1*sin2 + cos0*cos2;
       rotationMatrix[5] = cos0*sin1*sin2 - sin0*cos2;
       rotationMatrix[6] = -sin1;
       rotationMatrix[7] = sin0*cos1;
       rotationMatrix[8] = cos0*cos1;

       return rotationMatrix;
       }
};
```

Listing 7.5: BodyMotionfunctions

The *particleDynamicsFunctions.h* also contains other important functions to simulate particle flows. Tasks are included in the *particleTasks.h* like e.g. the *couple_lattice_to_particles* or *couple_resolved_particles_to_lattice*. Both functions are used in the main loop of the example, to realise a two-way-coupling.

To sum up, many of the most important functions for the simulations of the particle flow are implemented in the *ParticleDynamicsFunctions.h*. Other functions, e.g. concerning the calculation of rotation of the particle body, are implemented in the *bodyMotionfunctions.h*. The *particleIoFunctions.h* contains functions to get the output of the calculation to the console. It consist of two important functions (*printResolvedParticleInfo* an *printResovedParticleInfoSimple*), which are used in the *getResults*-function. The *getResults*-function is called at the end of the main part of every simulation (lines $22 - 24$, Listing 7.4).

7.2 Sub-grid legacy framework

In this chapter the use of Lagrangian particles with OpenLB is shown. Similar to the `BlockLattice` and `SuperLattice` structure a `ParticleSystem3D` and `SuperParticleSystem3D` structure exists. In line 2 of Listing 7.6 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 neclected 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. [30].

```
1   // SuperParticleSystems3D
2   SuperParticleSystem3D<T,PARTICLE> supParticleSystem(superGeometry);
3   // define which properties are to be written in output data
4   SuperParticleSysVtuWriter<T,PARTICLE> supParticleWriter(
        supParticleSystem, "particles",
5     SuperParticleSysVtuWriter<T,PARTICLE>::particleProperties::
          velocity |
6     SuperParticleSysVtuWriter<T,PARTICLE>::particleProperties::mass |
7     SuperParticleSysVtuWriter<T,PARTICLE>::particleProperties::radius
           |
8     SuperParticleSysVtuWriter<T,PARTICLE>::particleProperties::active
          );
9
10  SuperLatticeInterpPhysVelocity3D<T,DESCRIPTOR> getVel(sLattice,
        converter);
11
12  auto stokesDragForce = make_shared<StokesDragForce3D<T,PARTICLE,
        DESCRIPTOR>> (getVel, converter);
13
14  // material numbers where particles should be reflected
15  std::set<int> boundMaterial = { 2, 4, 5};
16  auto materialBoundary = make_shared<MaterialBoundary3D<T, PARTICLE
        >> (superGeometry, boundMaterial);
17
18  supParticleSystem.addForce(stokesDragForce);
19  supParticleSystem.addBoundary(materialBoundary);
20  supParticleSystem.setOverlap(2. * converter.getPhysDeltaX());
```

```

\* ... *\

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

Listing 7.6: Usage of class SuperParticleSystem3D

7.2.1 Interpolation of Fluid Velocity

As the particle position $\vec{X} : I \to \Omega$ moves in the continuous domain Ω 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 $\overline{(\vec{x}_0, \dots \vec{x}_n)}$ the smallest interval containing all points in the brackets. Furthermore, let $C^n \left[\vec{a}, \vec{b}\right]$ be the vector space of continuous functions that have continuous first n derivatives in $\left[\vec{a}, \vec{b}\right]$. 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 $\hat{\vec{x}} \in \overline{(\vec{x}_0, \dots \vec{x}_n, \vec{x})}$, such that*

$$\vec{u}^F(\vec{x}) - p_n(\vec{x}) = \frac{d_{\vec{x}}^{n+1} \vec{u}^F(\widehat{\vec{x}})}{(n+1)!} \prod_{j=0}^{n} (\vec{x} - \vec{x}_j) \tag{7.1}$$

holds.

Proof. See Rannacher [51, 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

$$\begin{aligned} f(\vec{x}) - p_1(\vec{x}) &= \frac{1}{2} d_{\vec{x}}^2 \vec{u}^F(\widehat{\vec{x}})(\vec{x} - \vec{x}_0)(\vec{x} - \vec{x}_1) \\ &\leq= \frac{1}{2} d_{\vec{x}}^2 \vec{u}^F(\widehat{\vec{x}}) h^2 \end{aligned}$$

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

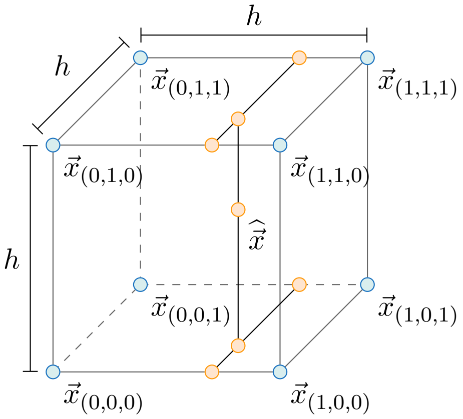


Figure 7.3: Trilinear interpolation.

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

$$\mathrm{Err}_h = \frac{\|\vec{u}_h^F - \vec{u}^{F^*}\|_{L^2(\Omega_h)}}{\|\vec{u}^{F^*}\|_{L^2(\Omega_h)}} \ .$$

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

$$\mathrm{EOC}_{i,j} = \frac{\ln(\mathrm{Err}_{h_i}/\mathrm{Err}_{h_j})}{\ln(h_i/h_j)} \ . \tag{7.2}$$

With this Krause [36, Chapter 2.3] determines an of EOC ≈ 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 $\mathbb{O}(h^2)$. This conclusion is backed up by the theoretical results obtained by [15]. 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 $\widehat{\vec{x}} \in [x_{(0,0,0)}, x_{(1,1,1)}]$ be in the cube spanned by the lattice nodes $\vec{x}_{(0,0,0)}$ and $\vec{x}_{(1,1,1)}$, see Figure 7.3 for an illustration.

We will denote by

$$\vec{d} = (d_0, d_1, d_2)^T = \hat{\vec{x}} - \vec{x}_{(0,0,0)}$$

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

$$\begin{aligned}
u_{(d,0,0)} &= u_{(0,0,0)}(h - d_0) + u_{(1,0,0)}d_0 \\
u_{(d,1,0)} &= u_{(0,1,0)}(h - d_0) + u_{(1,1,0)}d_0 \\
u_{(d,0,1)} &= u_{(0,0,1)}(h - d_0) + u_{(1,0,1)}d_0 \\
u_{(d,1,1)} &= u_{(0,1,1)}(h - d_0) + u_{(1,1,1)}d_0
\end{aligned}$$

followed by interpolation along the y-axis

$$\begin{aligned}
u_{(d,d,0)} &= u_{(d,0,0)}(h - d_1) + u_{(d,1,0)}d_1 \\
u_{(d,d,1)} &= u_{(d,0,1)}(h - d_1) + u_{(d,1,1)}d_1
\end{aligned}$$

and finally in direction of the z-axis

$$\vec{u}(\hat{\vec{x}}) = u_{(d,d,d)} = u_{(d,d,0)}(h - d_2) + u_{(d,d,1)}d_2.$$

7.2.2 Class SuperParticleSystem3D

The implementation of the particle phase follows an hierarchical ansatz, similar to the `Cell` → `BlockLattice3D` → `SuperLattice` ansatz used for the implementation of the LBM. The equivalent class in the context of Lagrangian particles are `Particle3D` → `ParticleSystem3D` → `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 `ParticleSystem3D`s, 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, SuperGeometry<T,3>& 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> vel={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 predefined 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. Finally 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(typename 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.

```
/// 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.

```
/// 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.

```
  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().

```
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.

```
  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.

7.2.3 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.

```
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++
      }
      //If not --> find new cuboid
      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.

```
/* 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 available. 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 transmissions 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_Double`s 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 `Particle3D`s 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.

Shadow Particles

If particle collisions are considered, it may happen that particles P_m with centre $\vec{X}_m \in \widetilde{\Omega}^j$ collide with particle P_n with centre $\vec{X}_n \in \widetilde{\Omega}^k$ in a different cuboid, as illustrated in Figure 7.4. Therefore P_n has to be known on $\vec{X}_m \in \widetilde{\Omega}^j$ 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 then 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.

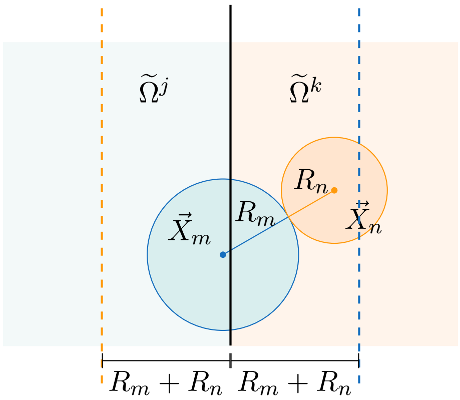


Figure 7.4: Overlap of the particle domains. Particles within a distance to of the sum of the two largest radii to a neighbour cuboid have to be transferred to this specific neighbour cuboid.

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 [5], thereby providing a nice GUI, if you are so inclined. Details on the XML format and functions are given in Section 8.8.

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 produced 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 provided 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 [6]. 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 version="1.0"?>
<VTKFile type="Collection" version="0.1" byte_order="LittleEndian">
<Collection>
<DataSet timestep="81920" group="" part="" file="data/VTM_iT0081920.vtm"/>
<DataSet timestep="163840" group="" part="" file="data/VTM_iT0163840.vtm"/>
<DataSet timestep="245760" group="" part="" file="data/VTM_iT0245760.vtm"/>
<DataSet timestep="327680" group="" part="" file="data/VTM_iT0327680.vtm"/>
<DataSet timestep="409600" group="" part="" file="data/VTM_iT0409600.vtm"/>
</Collection>
</VTKFile>
```

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.

```
<?xml version="1.0"?>
<VTKFile type="vtkMultiBlockDataSet" version="1.0" byte_order="LittleEndian">
<vtkMultiBlockDataSet>
<Block index="0" >
<DataSet index= "0" file="VTM_iT0081920iC00000.vti"></DataSet>
</Block>
<Block index="1" >
<DataSet index= "0" file="VTM_iT0081920iC00001.vti"></DataSet>
</Block>
<Block index="2" >
<DataSet index= "0" file="VTM_iT0081920iC00002.vti"></DataSet>
</Block>
<Block index="3" >
<DataSet index= "0" file="VTM_iT0081920iC00003.vti"></DataSet>
</Block>
</vtkMultiBlockDataSet>
</VTKFile>
```

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 [7], 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 `creatMasterFile()` in `iT == 0` is essential to write parallel vtk data.

8.3 CSV Writer

For some data analysis a CSV format of the data is necessary. In this case it is possible to use the CSV Writer to create these data files. The following lines show an application of the CSV Writer in the example advectionDiffusion1d (13.2.3). If one only wants to write in one data file, the filename can be given to the constructor of the CSV Writer. However the `plotFileName` parameter provides the possibility to set a new datafile with every call of this function. The `precision` parameter refers to the precision of the output data.

```
CSV<T> csv();
csv.writeDataFile(N, simulationAverage, "averageSimL2RelErr");
```

Listing 8.4: exemplary application of the CSV Writer

8.4 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 hyperplane 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 paratrized by normal (0,0,1) and centered in the mother geometry from the 3D data
BlockReduction3D2D<T> planeReduction( normVel, {0, 0, 1} );
```

Listing 8.5: 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 8.6: 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 8.7: Exemplary code using an explicitly instantiated $3D$ hyperplane lattice

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

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

Listing 8.8: 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 8.5, 8.6 and 8.7 There are two options of generating images of the processed Values in $2D$ and $3D$.

8.4.1 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.

```
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 8.9: 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:

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

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

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

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

```
ffmpeg -pattern_type glob -i ’tmp2/imageData/*.ppm’ animation.mp4
```

8.4.2 heatmap

Whereas the 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.

```
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 contur lines
jpeg_Param.colour = "rainbow";  //colour combination "grey", "pm3d",
    "blackbody" and "rainbow" can be chosen
heatmap::write(planeReduction, iT, jpeg_Param);
```

Listing 8.10: Exemplary code using the functor `heatmap` with modified parameters

The exemplary code in listing 8.10 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 13.9.5).

8.5 Gnuplot Interface

Often, for the analysis of simulations a plot of the data is required. OpenLB offers an interface which uses `Gnuplot` to create plots. Furthermore, it is possible to see the particular data that was used for the plots in realtime and to use comparison data, which are directly used in the plot.

An example for the usage from `examples/cylinder2d` is shown below.

```
// 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,
// if the png file is opened by an imageviewer it can be used as a "
    liveplot"
// optional for pdf output, use: gplot.writePDF()
gplot.writePNG();
}
```

Listing 8.11: An exemplary code to plot simulation data.

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

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

```
  ...

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

Listing 8.12: Creating plots for different time steps.

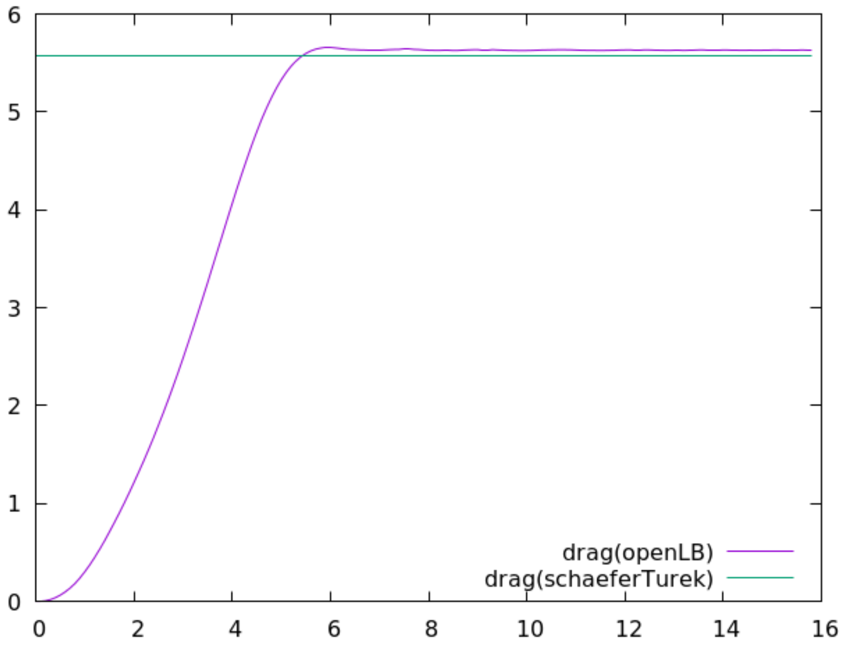


Figure 8.1: Gnuplot output of drag calculation in cylinder2d.

8.5.1 Regression with Gnuplot

Moreover, Gnuplot can be used for creating a linear regression to datasets. For instance, the analysis of the experimental order of convergence in a simulation can be executed as in the example poiseuille2dEOC.

The possible options are: Linear regression to the given data whereas it is possible to use a loglog-scaling (loglogINVERTED for inverting the x-axis). The implementation is done via the constructor of plot in the .cpp file itself as seen below:

```
static Gnuplot<T> gplot( "eoc", Gnuplot<T>::LOGLOG, Gnuplot<T>::
   LINREG);
```

The possible options for the scaling are: LINEAR (using the data as given), LOGLOG (using log of the x- and y-dataset) and LOGLOGINVERTED (using log of y-dataset and 1/log of x-dataset). For the regression type one can choose LINREG (linear Regression) and OFF (no regression).

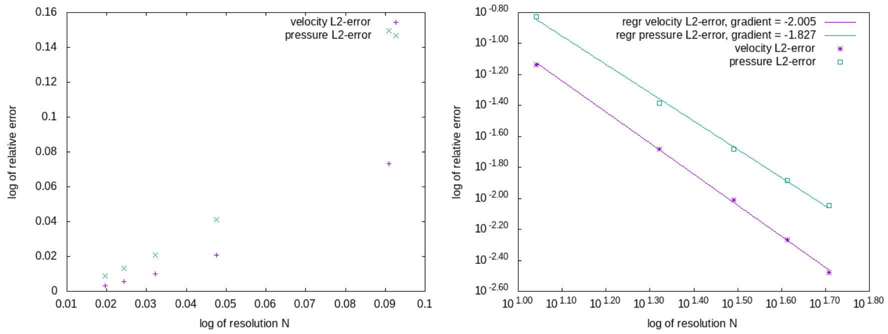


Figure 8.2: example of using regression to analyse polynomial errors (left: old, right: new implementation)

8.6 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 8.13 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.

```
1 void MyClass::print() {
2 OstreamManager clout(std::cout, "MyClass");
3 ...
4 clout << "step=" << step << "; avRho=" << avRho
5       << "; maxU=" << maxU << std::endl;
6 }
```

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
....
[prepareGeometry] Prepare Geometry ...
[SuperGeometry2D] cleaned 0 outer boundary voxel(s)
[SuperGeometry2D] cleaned 0 inner boundary voxel(s)
[SuperGeometry2D] the model is correct!
[SuperGeometryStatistics2D] materialNumber=0; count=13846; minPhysR=(0,0); maxPhysR=(5,0.75)
[SuperGeometryStatistics2D] materialNumber=1; count=92865; minPhysR=(0.0166667,0.0166667); maxPhysR=(19.9833,1.48333
[SuperGeometryStatistics2D] materialNumber=2; count=2448; minPhysR=(0,0); maxPhysR=(20,1.5)
[SuperGeometryStatistics2D] materialNumber=3; count=43; minPhysR=(0,0.783333); maxPhysR=(0,1.48333)
[SuperGeometryStatistics2D] materialNumber=4; count=89; minPhysR=(20,0.0166667); maxPhysR=(20,1.48333)
[prepareGeometry] Prepare Geometry ... OK
[prepareLattice] Prepare Lattice ...
[prepareLattice] Prepare Lattice ... OK
[main] starting simulation...
[SuperPlaneIntegralFluxVelocity2D] regionSize[m]=1.46667; flowRate[m^2/s]=0; meanVelocity[m/s]=0
[SuperPlaneIntegralFluxPressure2D] regionSize[m]=1.46667; force[N]=0; meanPressure[Pa]=0
[Timer] step=0; percent=0; passedTime=0.846; remTime=101519; MLUPs=0
[LatticeStatistics] step=0; t=0; uMax=1.49167e-154; avEnergy=0; avRho=1
[SuperPlaneIntegralFluxVelocity2D] regionSize[m]=1.46667; flowRate[m^2/s]=0; meanVelocity[m/s]=0
[SuperPlaneIntegralFluxPressure2D] regionSize[m]=1.46667; force[N]=0; meanPressure[Pa]=0
[Timer] step=300; percent=0.25; passedTime=2.503; remTime=998.697; MLUPs=17.2699
[LatticeStatistics] step=300; t=0.1; uMax=5.75006e-07; avEnergy=8.66459e-16; avRho=1
```

Listing 8.13: Terminal output of example bstep2d.

8.7 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 $\rightarrow$ 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 rad $= 2^{j-1} \cdot \text{voxelSize}, j \in \mathbb{N}$ with j such that a cube with diameter 2rad 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 rad $=$ 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).

8.8 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 tile `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`.

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

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

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

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 sqhere 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 `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.

$$Functor : \Omega \to \mathbb{R}^d, \quad d \in \mathbb{N} \tag{10.1}$$

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 `SuperLatticeF3D`. If the functor value is density, then this functor is called `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) `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.

$$AnalycialF3D : \mathbb{R}^3 \to \mathbb{R}^d, \quad d \in \mathbb{N} \tag{10.2}$$

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.

$$IndicatorF3D : \mathbb{R}^3 \to \{0,1\} \tag{10.3}$$

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.

$$SmoothIndicatorF3D : \mathbb{R}^3 \to [0,1] \tag{10.4}$$

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.

$$SuperLatticeF3D : \mathbb{N}^3 \to \mathbb{R}^d, \quad d \in \mathbb{N} \tag{10.5}$$

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.

```
// 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.

```
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.

```
// 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 13.3.4

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 showns how to compute the relative error over the whole three dimensional domain.

```
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 << "denstity-L2-error(abs)=" << absErr[0] << ";"
      << "denstity-L2-error(rel)=" << relErr[0] << std::endl;
```

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

For more detail, see the source code of example 13.3.4.

Assemble geometry with geometric primitives of type IndicatorFXD.

```
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.

```
// Creates and sets the Poiseuille inflow profile using functors
double maxVel = converter.getCharLatticeVelocity();
CirclePoiseuille3D<double> poiseuilleU(superGeometry, 3, maxVel,
    distance2Wall);
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*

```
Vector<T,3> origin;
Vector<T,3> u, v;
```

b) *origin and normal vector*

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

c) *normal vector (centered in mother cuboid)*

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

d) *circle indicator*

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

e) *arbitrary hyperplane*

```
// 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*

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

b) *grid resolution*

```
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*

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

a) *arbitrary indicator*

```
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 radius = 1.0;
IndicatorCircle2D<T> subplaneIndicator({0,0}, radius);
```

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

```
IndicatorF2D<T> subplaneIndicator...
```

Step 2: Create a `SuperF3D` functor for

a) *velocity flow*

```
SuperLatticePhysVelocity3D<T,DESCRIPTOR> f(sLattice, converter);
```

b) *pressure*

```
SuperLatticePhysPressure3D<T,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 `SuperLattice` 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

```
vFlux.print(std::string regionName,
  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

```
pFlux.print(std::string regionName,
  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:

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

10.2.7 Wall Shear Stress Functor

The *Wall Shear Stress* is defined as the parallel force per unit area exerted by a fluid on a wall. In the context of macroscopic fluid mechanics the *Wall Shear Stress* of a Newtonian fluid is given by:

$$\tau_W = \mu \frac{\partial \vec{u}}{\partial y}\bigg|_{y=0}$$

where μ is the dynamic viscosity, u is the velocity field and y the coordinate perpendicular to the wall. The *Wall Shear Stress Functor* calculates the discrete Wall Shear Stress

$$\tau_W = \boldsymbol{\sigma} \cdot \vec{n} - ((\boldsymbol{\sigma} \cdot \vec{n}) \cdot \vec{n}) \cdot \vec{n}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress tensor and $\vec{n}$ the local unit normal vector of the surface. Since the lattice stress tensor $\mathbf{\Pi}$ is not defined on boundary cells, it's read out from an adjacent fluid cell in a discrete velocity direction associated with each boundary cell. The unit normal vector is obtained by a given `IndicatorF3D` instance, which is slightly increased in size. See examples/poiseuille3d for usage details. Due to the staircase approximation of the boundary, the wall shear stress calculation is first order accurate.

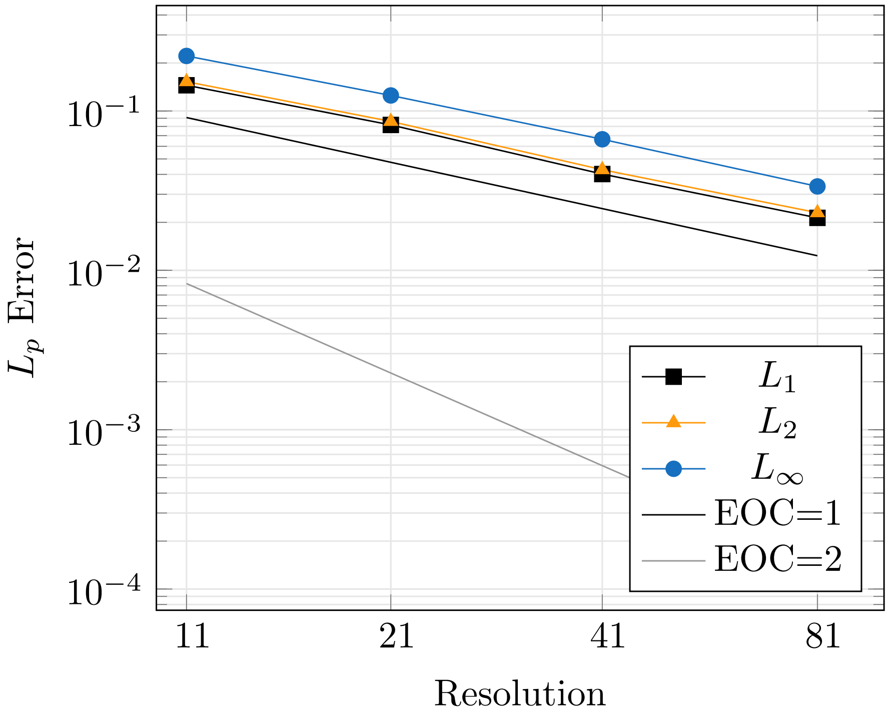


Figure 10.1: relative error of Wall Shear Stress L_p norm in poiseuille3d

10.2.8 Error Norm Functors

While relative and absolute error norms may be calculated manually using functor arithmetic (see 10.2.4) they are also available as distinct functors. As such it is preferable to utilize `SuperRelativeErrorLpNormXD` and `SuperAbsoluteErrorLpNormXD` if one uses the common definition of relative and absolute error norms:

Let `wantedF` be the simulated solution functor and `f` the analytical solution.

$$\texttt{SuperRelativeErrorLpNormXD}\text{ implements }\frac{\|\text{wantedF} - f\|_p}{\|\text{wantedF}\|_p}$$

$$\texttt{SuperAbsoluteErrorLpNormXD}\text{ implements }\|\text{wantedF} - f\|_p$$

An example of how to use these error norm functors in practice is given by the Poiseuille flow example as described in section 13.3.4.

```
Poiseuille2D<T> uSol(axisPoint, axisDirection, maxVelocity, radius);
SuperLatticePhysVelocity2D<T,DESCRIPTOR> u(sLattice, converter);
auto indicatorF = superGeometry.getMaterialIndicator(1);

SuperAbsoluteErrorL1Norm2D<T> absVelErrorNormL1(u, uSol, indicatorF);
absVelErrorNormL1(result, tmp);
clout << "velocity-L1-error(abs)=" << result[0];
SuperRelativeErrorL1Norm2D<T> relVelErrorNormL1(u, uSol, indicatorF);
relVelErrorNormL1(result, tmp);
clout << "; velocity-L1-error(rel)=" << result[0] << std::endl;
```

Listing 10.7: L1 velocity error in `poiseuille2d`

Further implementation details are touched upon in section 10.3.3.

10.2.9 Grid Refinement Metric Functors

`SuperLatticeRefinementMetricKnudsen(2,3)D` implements a automatic block-level grid refinement criterion as described by Lagrava et al. in “Automatic grid refinement criterion for lattice Boltzmann method” [39]. This criterion uses the quality of the cell-local Knudsen number approximation as measured by `SuperLatticeKnudsen*D` to judge the adequacy of the block resolution.

10.2.10 Rounding Functors

`SuperRoundingF(2,3)D` adds an option to restrict another functor by rounding its value in the way it is desired. The available options are: To the nearest integer, or only to the nearest higher/ lower integer.

10.2.11 Discretization Functors

`SuperDiscretizationF(2,3)D` allows to restrict a value into an interval. Also it is possible to set a number of equally distributed points into the interval, where the value will be restricted to.

10.3 Functor Arithmetic

10.3.1 Legacy 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. For example it facilitates the computation of relative errors.

Listing 10.8 shows basic usage of functor arithmetic to calculate the sum of two analytical functors. In this context *calculating the sum* doesn't mean that we sum specific values returned by the two functor's operators but rather that we instantiate a completely new functor as the result of the functor arithmetic expression. This new functor in turn then computes the described arithmetic expression on each call to its operator.

```
1 AnalyticalConst2D<double,double> one(1.0);
2 AnalyticalConst2D<double,double> two(2.0);
3 AnalyticalIdentity2D<double,double> tmp(one + two);
4 // or equivalent
5 AnalyticPlus2D<double,double> aPlus(one,two);
6 AnalyticalIdentity2D<double,double> tmp2(aPlus);
```

Listing 10.8: Basic showcase for arithmetic operations for AnalyticalF2D.

The remainder of this section explains the memory management concept of legacy functor arithmetic in OpenLB. It is strongly based on the example shown in Listing 10.8 and in particular on its third line.

To realize the central operation of line 3, `one + two`, the `operator+()` declared in `AnalyticalF2D<T,S>` is called by the object `one`, as shown in Figure 10.2.

A new object of type `AnalyticalPlus2D<T,S>` will be created by `operator+()` and managed using a `std::shared_ptr` which in turn is stored as a member variable

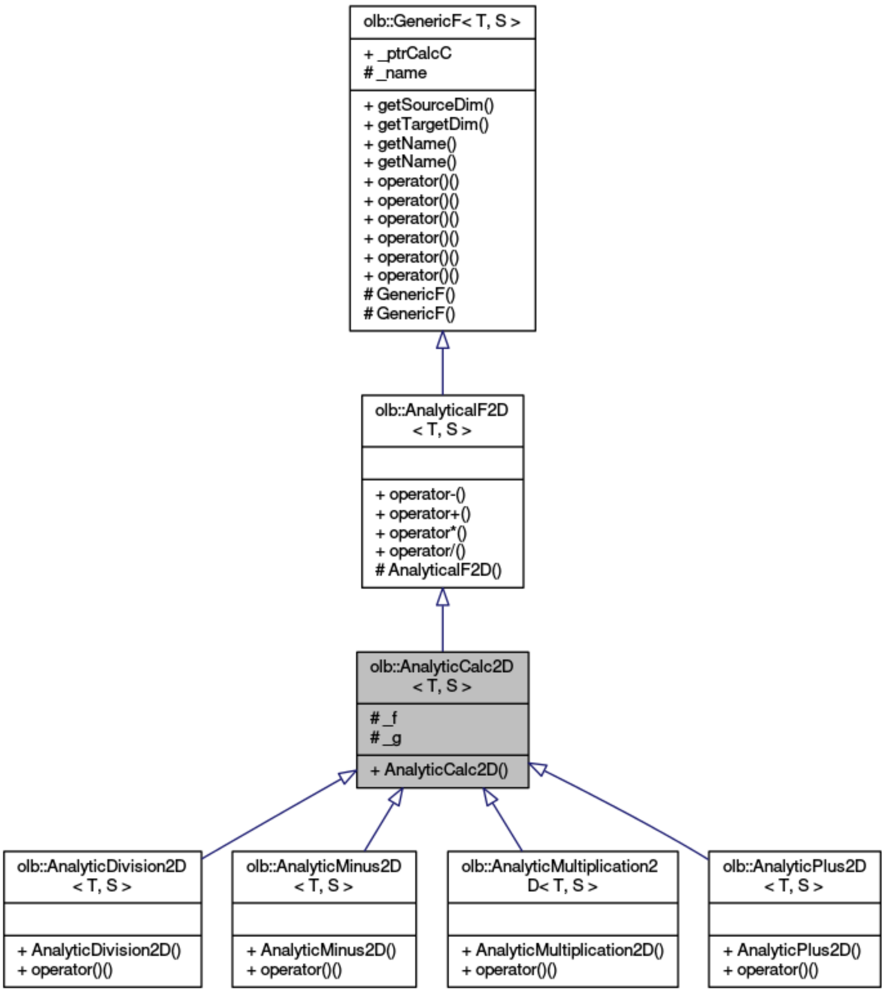


Figure 10.2: Inheritance for `AnalyticCalc2D` is shown.

of object `one`. The `std::shared_ptr` is used to free the memory allocated for the new *calculator* functor.

At this point, object `one` cares about managing the functor instance that encapsulates the arithmetic operation. However, if `one` is used for other arithmetic operations its `std::shared_ptr` may be overwritten, which can cause runtime errors. It would be more intuitive if `tmp` cared about memory management i.e. `tmp` should hold the `std::shared_ptr`. This is achieved in three steps:

First, constructing an `AnalyticalPlus2D<T,S>` object moves the `std::shared_ptr` instance from object `one` into `AnaltycialPlus2D<T,S>`. Then by constructing `tmp` the shared pointer moves once again to the newly created `AnalyticalIdentity2D<T,S>`.

Finally, tmp holds the std::shared_ptr containing the new AnalyticalPlus2D<T,S> instance. As such the lifetime of the functor arithmetic expression is tied to tmp. This makes sense as tmp wraps the only reference to the generated *arithmetic* functor.

10.3.2 Managed Functor Arithmetic

Memory management of legacy functor arithmetic as detailed in the previous section stops working in a reliable fashion as soon as a single functor instance is used in more than one arithmetic expression. Problems may also arise when ownership of the composed functor structure is transferred out of scope.

This is why it is suggested to manage more complex functor arithmetic expressions by explicitly wrapping functor instances in std::shared_ptr<F> smart pointers.

```
std::shared_ptr<SuperF3D<T>> aF(
  new SuperConst3D<T>(superStructure, {1.0, 2.0}));
std::shared_ptr<SuperF3D<T>> bF(
  new SuperConst3D<T>(superStructure, {2.0, 1.0}));
std::shared_ptr<SuperF3D<T>> cF = aF + bF;
// cF->operator() returns {3.0, 3.0}
```

Listing 10.9: Basic showcase for std::shared_ptr based functor arithmetic

Note that cF can be passed out of scope without any regard for aF and bF as managed pointers are stored internally. At first glance this new functor arithmetic may seem unnecessarily verbose for basic usage such as simply adding two functors and directly using the result. As such legacy functor arithmetic is still available for basic use cases. Usage of std::shared_ptr functor arithmetic is supported by both FunctorPtr and a DSL to ease development of more complex functor compositions.

The FunctorPtr helper template is used throughout the functor codebase to transparently accept functors independently of how their memory is managed. This means that functors managed by std::shared_ptr are accepted as arguments in any place where raw functor references were used previously. As a nice benefit FunctorPtr transparently forwards any calls of its own operator function to the operator of the underlying functor.

```
T error(FunctorPtr<SuperF3D<T>>&& f, T reference) {
  T         output[1] = { };
  const int origin[4] = {0,0,0,0};
  f(output, origin);
  return fabs(output[0] - reference);
}
```

```

std::shared_ptr<SuperF3D<T>> managedF(
  new SuperConst3D<T>(superStructure, 1.0));
SuperConst3D<T> rawF(superStructure, 1.0);

// error(managedF, 1.1) == error(rawF, 1.1) == 0.1
```

Listing 10.10: `FunctorPtr` and `std::shared_ptr` based functor arithmetic

Functor arithmetic expressions may also contain constants in addition to functors. Any scalar constant used in the context of managed functor arithmetic is implicitly wrapped into a `SuperConst3D<T>` instance.

```
std::shared_ptr<SuperF3D<T>> aF(/*...*/);
auto bF = 0.5 * aF + 2.0; // scalar multiplication and addition
```

Listing 10.11: Constant scalars in managed functor arithmetic

Constant vectors are also supported if they are explicitly passed to the `SuperConst3D<T>` constructor. Note that arithmetic operations of equidimensional functors are performed componentwise (i.e. `aF * aF` is not the scalar product).

```
std::shared_ptr<SuperF3D<T>> vectorF(
  new SuperConst3D<T>(superStructure, {1.0, 2.0, 3.0}));
auto cF = aF / vectorF; // componentwise division
```

Listing 10.12: Constant vectors in managed functor arithmetic

10.3.3 Functor Composition

Composing multiple managed functors which in turn need multiple arguments by themselves such as when calculating error norms in a reusable fashion can quickly lead to expressions that are fairly hard to read. Luckily the `functor_dsl` namespace offers a set of conveniently named helper functions in order to help deobfuscate such functor compositions.

Consider for example the following snippet which constructs and evaluates a relative error functor based on the L2 norm:

```
using namespace functor_dsl;

// decltype(wantedF) == std::shared_ptr<AnalyticalF3D<double,double>>
// decltype(f)       == std::shared_ptr<SuperF3D<double>>
```

```
5  // decltype(indicatorF) == std::shared_ptr<SuperIndicatorF3D<double>>
6
7  auto wantedLatticeF = restrict(wantedF, sLattice);
8  auto relErrorNormF  = norm<2>(wantedLatticeF - f, indicatorF))
9                      / norm<2>(wantedLatticeF, indicatorF);
10
11 const int input[4];
12 double result[1];
13 relErrorNormF->operator()(result, input);
14 std::cout << "Relative error: " << result[0] << std::endl;
```

Listing 10.13: `functor_dsl` supported functor composition

Note that lines 7 to 9 contain the full implementation of the expression

$$\frac{\|\mathrm{wantedF} - f\|_2}{\|\mathrm{wantedF}\|_2}$$

i.e. the L2-normed relative error of an arbitrary functor f as compared to the analytical solution *wantedF*.

This simplicity allows for moving even basically one-off functor compositions into reusable and easily verifiable functors whose implementation is as close to the actual mathematical definition as is reasonably possible. Correspondingly a more developed version of Listing 10.13 can be found in `SuperRelativeErrorLpNorm3D` which is used extensively by the `poiseuille3d` example to compare simulated and analytical solutions:

```
template <typename T, typename W, int P>
template <template <typename U> class DESCRIPTOR>
SuperRelativeErrorLpNorm3D<T,W,P>::SuperRelativeErrorLpNorm3D(
  SuperLattice<T,DESCRIPTOR>&      sLattice,
  FunctorPtr<SuperF3D<T,W>>&&        f,
  FunctorPtr<AnalyticalF3D<T,W>>&&   wantedF,
  FunctorPtr<SuperIndicatorF3D<T>>&& indicatorF)
  : SuperIdentity3D<T,W>([&]()
{
  using namespace functor_dsl;

  auto wantedLatticeF = restrict(wantedF.toShared(), sLattice);

  return norm<P>(wantedLatticeF-f.toShared(), indicatorF.toShared())
           / norm<P>(wantedLatticeF, indicatorF.toShared());
}())
{
```

```
18   this->getName() = "relErrorNormL" + std::to_string(P);
19 }
```

Listing 10.14: Functor composition in `SuperRelativeErrorLpNorm3D`'s constructor

Disregarding the addition of `FunctorPtr` as well as further templatization lines 12 to 15 are equivalent to the ad-hoc error norm in Listing 10.13. Also note how the actual composition happens inside of a lambda expression and is then returned to be stored by `SuperIdentity3D`. This allows for assigning composed functors their own name and renders them indistinguishable from *primitive* functors.

11 Solver Class

Quite a lot of program components are similar for each OpenLB application: e.g. the collide and stream loop is part of every simulation. The concept of a solver class is meant to perform such steps automatically, s.t. the user only has to define those steps which are specific for his/her application. Moreover, a generic interface shall be given for other programs (e.g. launch from python scripts or execution of optimization routines). For both purposes, this is work in progress and more improvements and functionalities are under development. In the following, the parts of an OpenLB program in solver style are explained. These steps are also illustrated by the examples cavity2dSolver and porousPlate3dSolver.

11.1 Structure of an OpenLB simulation in solver style

11.1.1 Parameter handling

In order to allow flexible interfaces to other programs, all parameters which are needed for simulation and interface are stored publicly in structs. For different groups of parameters (e.g. simulation/ output/ stationarity), different structs are used.

For `Simulation`, `Output` and `Stationarity`, basic versions containing the essential parameters are given by `SimulationBase`, `OutputBase`, `StationarityBase`, respectively. These can be supplemented by inheritance.

More parameter structs could be added for individualization. For instance, a `Results` struct could be used to save simulation results.

11.1.2 List parameter structs and lattices

A map of parameter structs with corresponding names is defined as a `meta::map`. Similarly, a map of lattice names and descriptors is defined. Some typical names are provided at `src/solver/names.h`; a list which is intended to be extended for individualization. The two maps are then given to the solver class as template parameters.

11.1.3 Definition of a solver class

Many standard routines for simulation are implemented in the existing class `LBSolver`. It is templatized w.r.t. maps of parameters and lattices and should therefore fit to most application cases. However, some steps (like the definition of the geometry) depend on the application and have to be defined for each application.

Therefore, an application-specific solver class is created as a child class of `LBSolver`. It has to implement the methods `prepareGeometry`, `prepareLattices`, `setInitialValues` and `setBoundaryValues`, similar to the classical app structure. Moreover, methods `getResults`, `computeResults`, `writeImages`, `writeVTK` and `writeGnuplot` can be defined if such output is desired. They are all called automatically during construction/ simulation.

The access to the parameter structs works with the tags defined above: e.g., `this->parameters(Simu` gives the maximal simulation time (which is a member of the struct `SimulationBase`). Similarly, we find access to super geometry and super lattices via `this->geometry()` and `this->lattice(LatticeName())`, respectively.

An automatic check, whether the simulation became stationary, is executed if a parameter struct with tag `Stationarity` is available (and the corresponding struct inherits from `StationarityBase`).

11.1.4 Main method

First, instances of the parameter structs and the solver class are constructed. This can be done classically, using the constructors (cf. example porousPlate3dSolver), or, if xml-reading has been implemented for all parameter structs, with the create-from xml-interface (cf. example cavity2dSolver).

Secondly, the `solve()` method of the solver instance is called in order to run the simulation.

11.2 Set up an app in solver style

In order to set up your own OpenLB application in solver style, the following steps should be followed:

- Select parameter structs. You can use existing ones or inherit from them/ define them completely new. The simulation parameters are expected to inherit from `SimulationParameters` and provide a unit converter. The output parameters should inherit from `OutputParameters`. You are free to add more parameter structs (e.g. for simulation results) yourself.

- Define a solver class. It should inherit publicly from the `LBSolver` class and implement the missing virtual methods like `prepareGeometry`.
- Define the main method. Either construct instances of the parameter structs and the solver class or use the create-from-xml interface. Then call the `solve()` method and possibly perform postprocessing.

12 Parallelization

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. 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 partioned 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 strucures. 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 8.6).

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.

12.1 Supported Platforms

OpenLB models heterogeneous parallelization using its predominant block decomposition architecture. Each individual block of a super lattice is assigned one of currently three possible target plaforms: `Platform::CPU_SISD`, `Platform::CPU_SIMD` or `Platform::GPU_CUDA`.

The availability of each platform is controlled by adding its name to the `PLATFORMS` variable in the `config.mk` build configuration. Note that further system specific changes to the compiler settings are almost certainly required for `CPU_SIMD` and `GPU_CUDA`. See the `config` folder for some examples.

Note that `CPU_SISD` must always be enabled as some host-side data structures rely on this platform.

12.1.1 SIMD

Modern CPUs offer vector instructions with a width of up to 512 bytes in the case of AVX-512. This means that 8 respectively 16 individual scalar values can be processed in a single instruction. In some situations this can significantly speed up the bulk collision step which is why OpenLB supports this option for processing its `Dynamics`.

This option is at its most powerful when combined with the `HYBRID` parallelization mode s.t. OpenMP is used to further parallelize the vectorized collision on each shared memory node. However, setting up the hybrid mode correctly on a HPC system is non trivial which is why we suggest to stick to the MPI-only mode for users unexperienced in working with HPC systems.

Once enabled in the build configuration, vectorization is applied to the dominant collision of each block lattice transparently without requiring any additional code changes.

12.1.2 GPU

General purpose graphics processing units (GPGPUs) are an ideal platform for many LBM-based simulations due to their high memory bandwidth and high degree of parallelization. OpenLB currently supports transparent usage of Nvidia GPUs via CUDA for almost all dynamics and a core set of boundary post processors.

Similarly to both other available platforms, enabling GPU support requires only adding `GPU_CUDA` to the `PLATFORMS` variable in `config.mk` and some system-specific updates to the compiler settings. MPI parallelization is fully supported for OpenLB GPU blocks, enabling simulations on multi-GPU clusters.

The set of GPU-enabled examples in OpenLB 1.5 consists of:

- `laminar/cavity(2,3)d`
- `laminar/cavity3dBenchmark`
- `laminar/cylinder(2,3)d`
- `laminar/poiseuille(2,3)d`
- `laminar/bstep(2,3)d`
- `laminar/powerLaw2d`
- `turbulence/nozzle3d`
- `turbulence/venturi3d`
- `turbulence/tgv3d`
- `advectionDiffusionReaction/advectionDiffusion3d`
- `freeSurface/(deep)fallingDrop2d`
- `freeSurface/breakingDam2d`

Note that the resolution commonly needs to be increased significantly compared to the CPU-accomodating default values in order to take full advantage of GPU acceleration. Especially on desktop-grade GPUs changing the fundamental floating point type `T` to `float` is advisable.

13 Example Programs

13.1 Example overview

All examples can be seen in the follwing table. They are listed with their related keywords to get a quick overview.

folder	example	turbulent	thermal	multi Component	multi Phase	particles	porous Media	transient flow	benchmark	showcase	STL geometry	geometry primitives	checkpointing
advectionDiffusionReaction	advectionDiffusionReaction2d								🟦			🟦	
	reactionFiniteDifferences2d								🟦			🟦	
	advectionDiffusion1d							🟦	🟦			🟦	
	advectionDiffusion2d							🟦	🟦			🟦	
	advectionDiffusion3d							🟦	🟦			🟦	
	advectionDiffusionPipe2d							🟦	🟦	🟦		🟦	
laminar	bstep2d							🟦	🟦			🟦	🟦
	bstep3d							🟦	🟦			🟦	🟦
	cavity2d							🟦	🟦			🟦	
	cavity3d							🟦	🟦			🟦	
	cylinder2d								🟦			🟩	
	cylinder3d								🟦		🟦		
	poiseuille2d								🟦			🟦	
	poiseuille2dEOC								🟦			🟦	
	poiseuille3d								🟦			🟦	
	powerLaw2d								🟦			🟦	
multiComponent	binaryShearFlow2d				🟦			🟦	🟦	🟦		🟦	
	contactAngle2d				🟦				🟦			🟦	
	contactAngle3d				🟦				🟦			🟦	
	fourRollMill2d				🟦			🟦	🟦	🟦		🟦	
	microFluidics2d			🟦				🟦		🟦		🟦	
	phaseSeperation2d				🟦			🟦				🟦	
	phaseSeperation3d				🟦			🟦				🟦	
	rayleighTaylor2d			🟦				🟦	🟦			🟦	
	rayleighTaylor3d			🟦				🟦	🟦			🟦	
	youngLaplace2d			🟦					🟦			🟦	
	youngLaplace3d			🟦					🟦			🟦	

🟦 example includes relevant subject

🟩 example includes relevant subject and is recommended for beginning

folder	example	turbulent	thermal	multi Component	multi Phase	particles	porous Media	transient flow	benchmark	showcase	STL geometry	geometry primitives	checkpointing
particles	bifurcation3d					🟦		🟦	🟦	🟦	🟦		
	dkt2d					🟦		🟦	🟦	🟦		🟦	
	magneticParticles3d					🟦		🟦		🟦		🟦	
	settlingCube3d					🟦		🟦		🟦		🟦	
porousMedia	porousPoiseuille2d						🟦		🟦			🟦	
	porousPoiseuille3d						🟦		🟦			🟦	
thermal	galliumMelting2d		🟦		🟦			🟦	🟦	🟦		🟦	
	porousPlate2d		🟦						🟦			🟦	
	porousPlate3d		🟦						🟦			🟦	
	porousPlate3dSolver		🟦						🟦			🟦	
	rayleighBernard2d	🟦	🟦					🟦	🟦	🟦		🟦	
	rayleighBernard3d	🟦	🟦					🟦	🟦	🟦		🟦	
	squareCavity2d	🟦	🟦					🟦	🟦			🟦	
	squareCavity3d		🟦						🟦			🟦	
	stefanMelting2d		🟦		🟦			🟦	🟦			🟦	
turbulent	aorta3d	🟦						🟦	🟦	🟦	🟦		
	channel3d	🟦						🟦	🟦	🟦		🟦	
	nozzle3d	🟦						🟦		🟦		🟦	
	tgv3d	🟦						🟦	🟦			🟦	
	venturi3d	🟦						🟦		🟦		🟩	

🟦🟦 example includes relevant subject

🟩 example includes relevant subject and is recommended for beginning

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

13.2 advectionDiffusionReaction

13.2.1 advectionDiffusionReaction2d

This example illustrates a steady-state chemical reaction in a plug flow reactor. One can choose two types of reaction, $A \longrightarrow C$ and $A \longleftrightarrow C$. The concentration and analytical solution along the centerline of the rectangle domain is given in `./tmp/N<resolution>/gnuplotData` as well as the error plot for the concentration along the centerline. The default configuration executes three simulation runs and the average L^2-error over the centerline is computed for each resolution. A plot of the resulting experimental order of convergence is provided in `./tmp/gnuplotData/`.

13.2.2 reactionFiniteDifferences2d

Similarly to the previous example, a simplified domain with no fluid motion and homogeneous species concentrations is simulated, but here with finite differences. The chemical reaction $|a|A \longrightarrow |b|B$ is approximated, where the reaction rate $\nu = A/t_0$ is given, where t_0 is a time conversion factor. The initial conditions are set to $A(t = 0) = 1$ and $B(t = 0) = 0$ such that an analytical solution is possible via

$$A(t) = \exp\left(-\frac{|a|\,t}{t_0}\right), \tag{13.1}$$

$$B(t) = \left|\frac{b}{a}\right| \left[1 - \exp\left(-\frac{|a|\,t}{t_0}\right)\right]. \tag{13.2}$$

By default, the executable produces a plot in `./tmp/gnuplotData/` which is given in Figure 13.1 below.

13.2.3 advectionDiffusion1d

The advectionDiffusion1d example showcases second order mesh convergence of LBM for scalar linear one-dimensional advection–diffusion equations [55] of the form

$$\partial_t \chi + \partial_x F(\chi) - \mu \partial_{xx} \chi = 0, \tag{13.3}$$

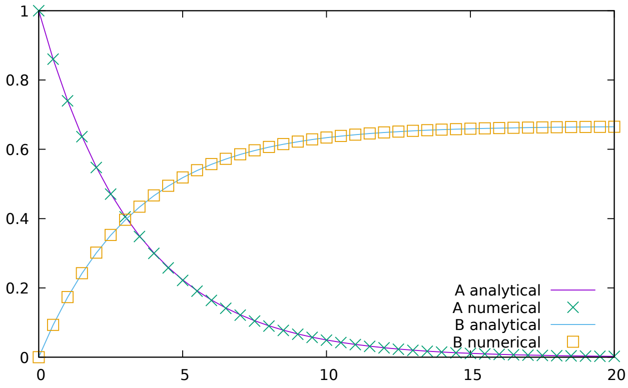


Figure 13.1: Solutions to concentration profiles in reactionFiniteDifferences2d.

where $\chi : \mathcal{X} \times \mathcal{I} \to \mathbb{R}$ is the conservative variable dependent on space $x \in \mathcal{X} \subseteq \mathbb{R}$ and time $t \in \mathcal{I} \subseteq \mathbb{R}_0^+$, $F \equiv u\chi$ is a linear function defined by the advection velocity $u \in \mathbb{R}$, and $\mu > 0$ denotes the diffusion coefficient. Hence, the LBM approximates the transport of the conservative variable χ along a one-dimensional line with periodic boundary conditions on $\mathcal{X} = [-1, 1]$. The inital pulse is defined by a sine profile, which is subsequently diffused and advected. An analytical solution at point x and time t is given by

$$\chi^\star(x,t) = \sin\left[\pi(x - ut)\right] \exp\left(-\mu\pi^2 t\right). \tag{13.4}$$

In practice the simulation uses a two-dimensional square domain which is evaluated along a centerline to obtain the desired one-dimensional result. The domain is initialised with $\chi^\star(x, t = 0)$.

Diffusive scaling is applied which results in the input parameters listed in Table 13.1. In the default setting, advectionDiffusion1d executes three simulation runs with increasing resolutions $N = 50, 100, 200$, respectively. Each simulation recovers $\mu = 1.5$ and a Péclet number of $Pe = 40/3$.

diffusive scaling $\triangle t = \triangle x^2$ for $Pe = 40/3$			
N	u_L	$\triangle x$	τ
50	0.4	0.04	5.0
100	0.2	0.02	5.0
200	0.1	0.01	5.0

Table 13.1: Default simulation parameters of advectionDiffusion1d

The output of each simulation run is stored in the `tmp/N<number>` directory. At each simulation timestep the average L2 relative Error over the centerline is computed. Said average is then stored within the respective resolution directory `./gnuplotData/data/averageL2RelError.dat`. Additionally, the program averages the values in `averageL2RelError.dat` for each simulation run, which in turn is written to the global error file `tmp/gnuplotData/data/averageSimL2RelErr.dat`. For post-processing, a python3 script can be executed via

```
python3 advectionDiffusion1dPlot.py
```

The script requires the matplotlib python package which can be installed on any platform by issuing the following commands in a terminal:

```
python3 -m pip install -U pip
python3 -m pip install -U matplotlib
```

The script generates basic error plots for every file with the file extension `.dat` in `./tmp`. Finally, a global log-log error plot with reference curves is extracted from the data contained in `averageSimL2RelErr.dat`.

13.2.4 advectionDiffusion2d

The example advectionDiffusion2d acts as a mesh-convergence test for a solution to the scalar linear *two-dimensional* advection–diffusion equation

$$\partial_t \chi + \nabla_{\vec{x}} \vec{F}(\chi) - \mu \Delta_{\vec{x}} \chi = 0, \tag{13.5}$$

where $\chi : \mathcal{X} \times \mathcal{I} \to \mathbb{R}$ is the conservative variable dependent on space $\vec{x} \in \mathcal{X} \subseteq \mathbb{R}^2$ and time $t \in \mathcal{I} \subseteq \mathbb{R}_0^+$, $\vec{F} \equiv \vec{u}\chi$ is a linear function defined by the advection velocity $\vec{u} = (u_x, u_y)^{\mathrm{T}} \in \mathbb{R}^2$, and $\mu > 0$ denotes the diffusion coefficient. Similarly, the analytical solution is given for any point $\vec{x} = (x, y)^{\mathrm{T}}$ and time t as

$$\chi^\star(x, y, t) = \sin[\pi(x - u_x t)] \sin[\pi(y - u_y t)] \exp\left(-2\mu\pi^2 t\right). \tag{13.6}$$

The simulation is executed on a square $\mathcal{X} = [-1, 1]^2$ which is periodically embedded in $\mathbb{R}^2$. An error norm over the domain measures the deviation from the analytical solution up to the timestep at which the initial pulse is diffused below 10%. For the default setting ($\mu = 0.05$ and $Pe = 100$), the outputs of three subsequent simulation runs are stored in a subfolder structure in `./tmp` and directly post-processed for visualization. A sequence of contour plots is generated with the highest computed resolution $N = 200$ and contained in `./tmp/N200/imageData`. Note that via issuing the command

```
python3 advectionDiffusion2dPlot.py
```

an error plot can be produced, which numerically validates the second order convergence in space.

13.2.5 advectionDiffusion3d

The example advectionDiffusion3d acts as a mesh-convergence test for a numerical solution to initial value problem

$$\begin{cases} \partial_t \chi(\vec{x},t) + \nabla_{\vec{x}} \vec{F}\left(\chi(\vec{x},t)\right) - \mu \Delta_{\vec{x}} \chi(\vec{x},t) = 0 & \text{in } \mathcal{X} \times \mathcal{I}, \\ \chi(\vec{x},0) \equiv \chi_0(\vec{x}) & \text{in } \mathcal{X}, \end{cases} \tag{13.7}$$

where $\chi : \mathcal{X} \times \mathcal{I} \to \mathbb{R}$ is the conservative variable dependent on space $\vec{x} \in \mathcal{X} \subseteq \mathbb{R}^3$ and time $t \in \mathcal{I} \subseteq \mathbb{R}_0^+$, $\vec{F} \equiv \vec{u}\chi$ is a linear function defined by the advection velocity $\vec{u} = (u_x, u_y, u_z)^{\mathrm{T}} \in \mathbb{R}^3$, and $\mu > 0$ denotes the diffusion coefficient. Note that the domain $\mathcal{X} = [-1,1]^3$ is periodic.

The example implements a smooth initial profile $\chi_0^{\mathrm{s}}(\vec{x})$ and an unsmooth version $\chi_0^{\mathrm{u}}(\vec{x})$. The former is a three-dimensional extrusion of the initial pulse in the advectionDiffusion2d example, such that the equation admits the analytical solution [20, 55]

$$\chi^{\star,\mathrm{s}}\left(x,y,z,t\right) = \sin\left[\pi\left(x - u_x t\right)\right] \sin\left[\pi\left(y - u_y t\right)\right] \sin\left[\pi\left(z - u_z t\right)\right] \exp\left(-3\mu\pi^2 t\right). \tag{13.8}$$

The latter comprises a Dirac delta at x_0 as initial pulse which induces a Dirac comb as superpositioned analytical solution [20]

$$\chi^{\star,\mathrm{u}}\left(x,y,z,t\right) = \frac{1}{\sqrt{4\pi\mu t}} \sum_{k\in\mathbb{Z}} \exp\left(\frac{-\left(x - x_0 - u_x t + 2k\right)^2}{4\mu t}\right) + 1 \tag{13.9}$$

For each case several error norms over the domain measure the deviations from the analytical solution. For the default setting the outputs of three subsequent simulation runs are stored in a subfolder structure in `./tmp` and directly post-processed for visualization. Via issuing the command

```
python3 advectionDiffusion3dPlot.py
```

an error plot is produced, which numerically validates the second order convergence for both initializations, under the constraints on the grid Péclet number derived in [20].

13.2.6 advectionDiffusionPipe3d

This example implements a spreading Gaussian density package advecting within a square duct pipe with velocity $\vec{U}$. The precise description of the test-case can be found in [20]. Whereas the velocity is computed via approximating the incompressible Navier–Stokes equations with a $D3Q19$ BGK LBM, the advection–diffusion equation for the density package is solved with finite differences (FD). Four different FD schemes can be employed within the example. The advantages of each of the schemes for a broad range of Pe are documented in [20].

13.3 laminar

13.3.1 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. The 2D geometry corresponds to Armaly et al. [9].

13.3.2 cavity2d, cavity2dSolver 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. The 2d-solver version illustrates the use of the solver class concept (cf. Chapter 11) together with the XML parameter interface.

13.3.3 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 [60]. 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 [4].

13.3.4 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 (see Figure 13.2). 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.

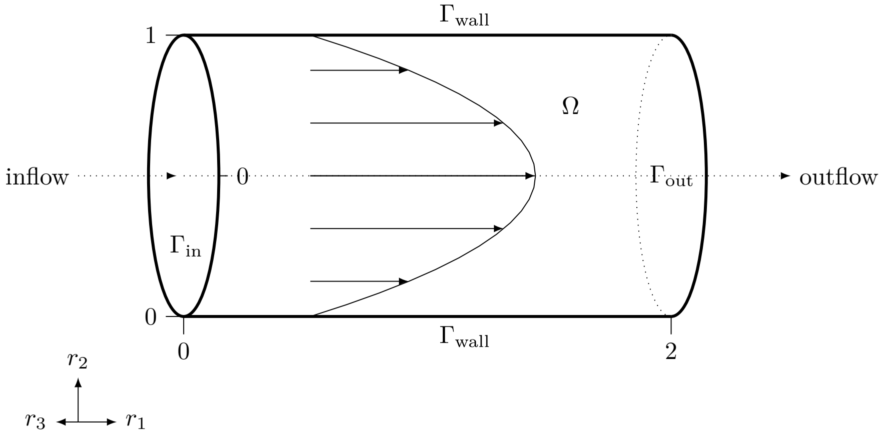


Figure 13.2: poiseuille3d geometry with boundary patches and velocity profile.

13.3.5 poiseuille2dEOC

poiseuille2dEOC is an extension of the poiseuille2d example which focuses on the experimental order of convergence of the simulation. This example basically runs the poiseuille2d simulation multiple times and analyzes the different error norms of the simulation with the gnuplot extension (cf. subsection 8.5.1).

13.3.6 powerLaw2d

This example describe 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.

13.4 multiComponent

The examples in this folder demonstrate the use of the free-energy model (section 6.6.4) for multicomponent flows.

13.4.1 binaryShearFlow2d

A circular domain of one fluid phase is immersed in a rectangle filled with another fluid phase. The top and bottom walls are moving in opposite directions, such that the droplet shaped phase is exposed to shear flow and deforms accordingly. The default parameter setting is taken from [33] and injected into the more general ternary free energy model from [52]. Both scenarios, breakup and steady state of the initial droplet, are implemented and visualilzed as vtk output.

13.4.2 contactAngle2d and contactAngle3d

In this example a semi-spherical droplet of fluid is initialised within a different fluid at a solid boundary. The contact angle is measured as the droplet comes to equilibrium. This is compared with the analytical angle predicted by the parameters set for the boundary (100 degrees for preset values).

This example demonstrates how to use the solid wetting boundaries for the free-energy model with two fluid components.

13.4.3 fourRollMill2d

Here, a spherical domain filled with one fluid phase is immersed in a square filled with another phase of equal density and viscosity. Four circle structures which represent roller sections are equidistantly distributed in the corners of the domain. The bottom left and top right cylinders begin to spin in counterclock-wise direction. Whereas the top left and bottom right cylinders spin in clock-wise direction. A velocity field of extensional type deforms the initial droplet accordingly. Dependent on the non-dimensional parameter setting in the example header, the droplet reaches steady state or breaks up.

13.4.4 microFluidics2d

This example shows a microfluidic channel creating droplets of two fluid components. Poiseuille velocity profiles are imposed at the various channel inlets, while a constant density outlet is imposed at the end of the channel to allow the droplets to exit the simulation.

This example demonstrates the use of three fluid components with the free energy model. It also shows the use of open boundary conditions, specifically velocity inlet and density outlet boundaries.

13.4.5 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.

13.4.6 rayleighTaylor2d and rayleighTaylor3d

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 H. Chen is used [53]. These examples show the usage of multicomponent flow and periodic boundaries.

13.4.7 youngLaplace2d and youngLaplace3d

In this example the two-component free energy model is used in its simplest configuration to perform a Young–Laplace pressure test. A circular or spherical domain of a fluid with radius R is immersed in another fluid. A diffusive interface forms and the pressure difference across the interface, Δp, is calculated and compared to that given by the Young–Laplace equation,

$$\Delta p = \frac{\gamma}{R} = \frac{\alpha}{6R}(\kappa_1 + \kappa_2) \qquad \text{for 2D,} \tag{13.10}$$

$$\Delta p = \frac{2\gamma}{R} = \frac{\alpha}{3R}(\kappa_1 + \kappa_2) \qquad \text{for 3D.} \tag{13.11}$$

The parameters α and κ_i are input parameters to the simulation which define the interfacial width and surface tension, γ, respectively.

The pressure difference is calculated between a point in the middle of the circular domain and a point furthest away from it in the computational domain.

13.5 particles

13.5.1 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 implenetations 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.

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.

```
SuperExternal3D<T,ADDESCRIPTOR,descriptors::VELOCITY> sExternal(
          superGeometry,
          sLatticeAD,
          sLatticeAD.getOverlap());

   ...

sExternal.communicate();
```

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 6.7.5. In this example only the Stokes drag is applied by

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

For the simulation of particles as a continuum, also new boundary conditions are required. Here `setZeroDistributionBoundary` 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 `AdvectionDiffusionConvectionBoundary` for advection–diffusion lattices can be applied, that approximates a Neumann boundary condition, for further reference see [58]. 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 `setExtFieldBoundary`.

eulerLagrange

The main task of his example is to show the using of Lagrangian particles with OpenLB. As already described in Section 7.2, similar to the `BlockLattice` and `SuperLattice` structure a `ParticleSystem` and `SuperParticleSystem` structure exists in the context of the legacy particle framework. Besides the particles the examples use the save feature of the `SuperLattice`. By

```
sLattice.save("fluidSolution")
```

and

```
sLattice.load("fluidSolution")
```

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.

13.5.2 dkt2d

OpenLB provides an alternative approach to conventional resolved particle simulation methods, referred to as the homogenised lattice Boltzmann method (HLBM). It was introduced in "Particle flow simulations with homogenised lattice Boltzmann methods" by Krause et al. [34] and extended for the simulation of 3D particles in "Towards the simulation of arbitrarily shaped 3D particles using a homogenised lattice Boltzmann method" by Trunk et al. [58]. It was eventually revisited in [59]. In this approach the porous media model, introduced into LBM by Spaid and Phelan (1997) [56], is extended by enabling the simulation of moving porous media. In order to avoid pressure fluctuations, the local porosity coefficient is used as a smoothing parameter.

The example *dkt2d* employs said approach for the sedimentation of two particles under gravity in a water-like fluid in 2D. The rectangular domain is limited by no-slip boundary conditions. This setup is usually referred to as a drafting–kissing–tumbling (DKT) phenomenon and is widely used as a reference setup for the simulation of particle dynamics submerged in a fluid. The benchmark case is e.g. described in "Drafting, kissing and tumbling process of two particles with different sizes" by Wang et al. or "The immersed boundary-lattice Boltzmann method for solving fluid-particles interaction problems" by Feng and Michaelides. For the calculation of forces a DNS approach is chosen which also leads to a back-coupling of the particle on the fluid, inducing a flow. The example demonstrates the usage of HLBM in the OpenLB framework as well as the utilisation of the Gnuplot-writer to print simulation results.

13.5.3 magneticParticles3d

Warning: This example can currently only be run sequentially!

High-gradient magnetic separation is a method to separate ferromagnetic particles from a suspension. The simulation shows the deposition of magnetic particles on a single magnetized wire and models the magnetic separation step of the complete process.

13.5.4 settlingCube3d

The case examines the settling of a cubical silica particle under gravity in a surrounding fluid. The rectangular domain is limited by no-slip boundary conditions. For the calculation of forces a DNS approach is chosen which also leads to a back-coupling of the particle on the fluid, inducing a flow. The example demonstrates the usage of HLBM in the OpenLB framework as well as the utilisation of the Gnuplot-writer to print simulation results (Section 13.5.2).

13.6 porousMedia

13.6.1 porousPoiseuille2d and porousPoiseuille3d

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. For a schematic simulation setup see Figure 13.3.

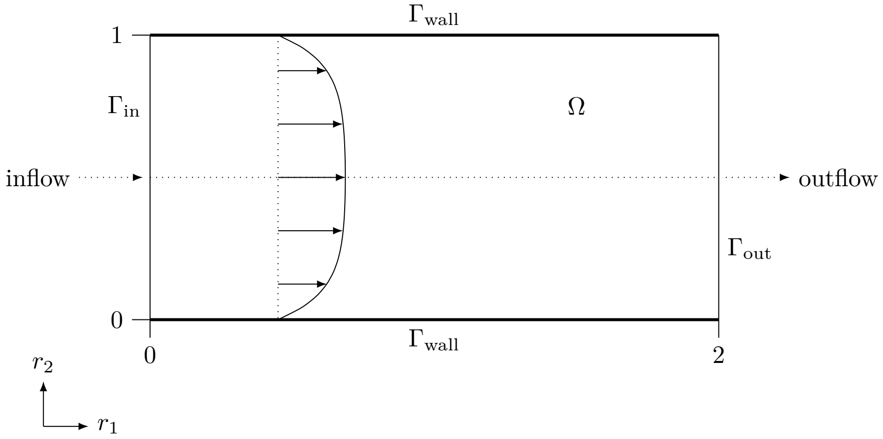


Figure 13.3: porousPoiseuille2d geometry with boundary patches and velocity profile.

13.7 reaction

13.7.1 advectionDiffusionReaction2d

This example illustrates a forward or a reversible reaction of a substance into another one (A→C or A↔C). The concentration of each substance is modeled with a one-dimensional advection diffusion equation. We consider a steady state in a plug flow reactor. This means that it is time independent and we have a constant velocity field u. The chemical reaction is modeled with a linear but coupled source term. The equations read

$$\begin{cases} u\partial_x c_A & = D\partial_x^2 c_A - k_H c_A + k_R c_C \\ u\partial_x c_C & = D\partial_x^2 c_C + k_H c_A - k_R c_C \end{cases}$$

with the diffusion coefficient $D > 0$, forth reaction rate coefficient $k_H > 0$ and backwards reaction rate coefficient $k_R > 0$ (= 0 in case of $A \rightarrow$ C). This LBM models the transport of the species concentration along a one-dimensional line on $[0, 10]$. In practice the simulation uses a two-dimensional rectangular domain which is evaluated along a centerline to obtain the desired one-dimensional result. The hight of the domain depends on the resolution which holds the number of voxels for the hight constant.

On the bottom and the top of the rectangular periodic boundaries and at the inlet and outlet the concetrations from the analytical solution are set. The solution is given by

$$c_A(x) = c_{A,0}e^{\lambda x} + \frac{k_R}{k_H + k_R} c_{A,0} \left(1 - e^{\lambda x}\right)$$
$$c_C(x) = c_{A,0} - c_A(x)$$

with $\lambda = \frac{u - \sqrt{u^2 + 4(k_H + k_R)D}}{2D}$.
Diffusive scaling is applied and a physical diffusivity of $D = 0.1$ and a flow rate of $u = 0.5$ which leads to a Péclet number of $Pe = 100$.
Every species has its own lattice and stored in a vector. In the `simulate` method we iterate over ever element of the vector `adlattices`.

In the default setting, advectionDiffusionReaction2d executes three simulation runs with increasing resolutions $N = 200, 250, 300$, respectively. The output of each simulation run is stored in the `tmp/N<number>` directory. It contains a plot `centerConcentrations.pdf` of the concentrations and the analytical solution along the centerline and an error plot of the numerical and analytical solution along the centerline `ErrorConcentration.pdf`. After each simulation has converged the average L2 relative Error over the centerline is computed. Said average is then stored within `tmp/gnuplotData/data/averageL2RelError.dat`. The order of convergence can be seen in the log-log error plot in `tmp/gnuplotData/concentration_eoc.png`. The EOC plot is only done for one species (A or C) and the species can be chosen in the return statement of the method `errorOverLine`.

One can select the reactionType `a2c` or `a2cAndBack` which automatically provides the data for modelling the reaction. It contains the number of reactions, the reaction rate coefficients `physReactionCoeff[numReactions]` (k_H, k_R), the number of species `numComponents` and their names, the stoichometric coefficients `stochCoeff` which are sorted according to the number of reactions and inside each reaction block according to the species number. Finally we assume that the reaction rate satisfies a power law depending on the concentration of the species'. The exponent is given by the reaction order `reactionOrders` which is sorted in the same way as `stochCoeff`. In the example cases these exponents are always 1.
The chemical reaction itself is represented as a source term for each Advection Diffusion equation. This source term is calculated in the `ConcentrationAdvectionDiffusionCouplingGenerator` for every species which can handle arbitrary num-

ber of species and reactions and stored in the field `SOURCE`.

13.8 thermal

13.8.1 galliumMelting2d

The solution for the melting problem (solid-liquid phase change) coupled with natural convection is found using the lattice Boltzmann method after Huang and Wu [31]. The equilibrium distribution function for the temperature is modified in order to deal with the latent-heat source term. That way, iteration steps or solving a group of linear equations is avoided, which results in enhanced efficiency. The phase interface is located by the current total enthalpy, and its movement is considered by the immersed moving boundary scheme after Noble and Torczynski [48]. This method was validated by comparison with experimental values (e.g. Gau and Viskanta [23]).

13.8.2 porousPlate2d, porousPlate3d and porousPlate3dSolver

The porous plate problem was described by [26] and [49]

To test the coupled model's accuracy and to determine it's experimental order of convergence (EOC), we use a numerical simulation the porouse 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 13.4.

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

$$u_x(y) = u_{x,0}(\frac{e^{Re \cdot y/L} - 1}{e^{Re} - 1}) \tag{13.12}$$

$$T(y) = T_0 + \Delta T = (\frac{e^{Pr \cdot Re \cdot y/L} - 1}{e^{Pr \cdot Re} - 1}) \tag{13.13}$$

Herein $u_{x,0}$ is the upper plate's velocity, $Re = \frac{u_{y,0}L}{\nu}$ the Reynolds number depending on the injected velocity $u_{y,0}$, the fluid's viscosity ν and the channel length L. The

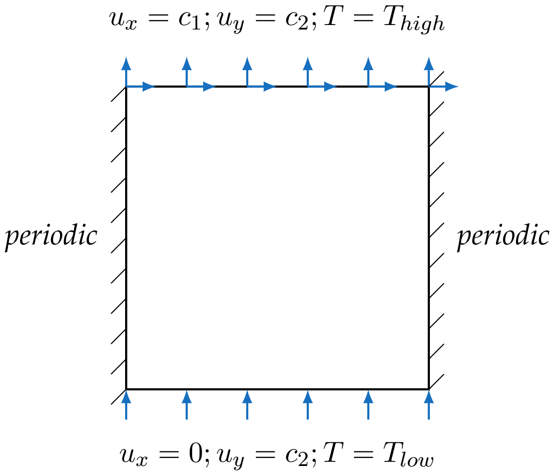

$u_x = c_1; u_y = c_2; T = T_{high}$
periodic
periodic
$u_x = 0; u_y = c_2; T = T_{low}$

Figure 13.4: Schematic representation of the porous plate's simulation setup including it's boundary conditions

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 [[26]]:

$$E = \frac{\sqrt{\sum_i |T(x_i) - T_a(x_i)|^2}}{\sqrt{\sum_i |T_a(x_i)|^2}} \quad , \tag{13.14}$$

where the summation is over the entire system, T_a is the analytical solution 13.13.

The porousPlate3dSolver example implements the same simulation, but additionally illustrates the application of the solver class concept.

13.8.3 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. [25], 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.

$$T(x,y,t=0) = T_{max}[(1-\frac{y}{L_y}) + 0.1cos(2\pi\frac{x}{L_x})] \tag{13.15}$$

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.

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

Listing 13.1: Initialization of the temperature field

In the equation 13.15, the y-dependent part of the equation matchs to the stationary solution of the problem, corresponding to a case where there is no fluid movement and

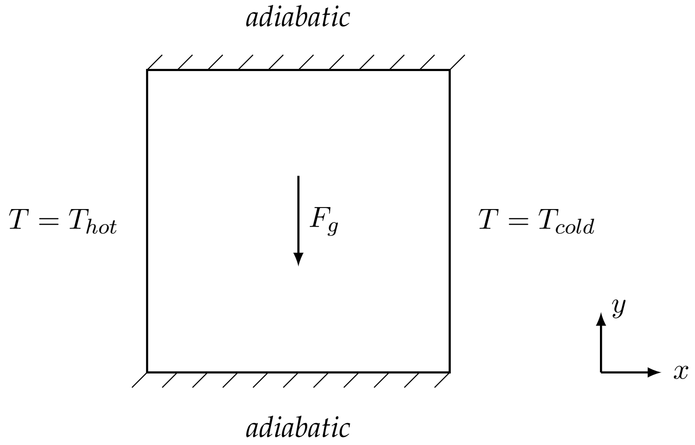


Figure 13.5: Schematic diagram of the simulation domain

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}.

13.8.4 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.

Setup

The problem considered is shown schematically in Figure 13.5. The horizontal walls of the cavity are adiabatic, while the vertical walls are kept isothermal, with the left wall at high temperature (T_{hot} = 1) and the right wall at low temperature (T_{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 [41], but it had variations over 10% with respect to the BGK model, so another formulation (reference [42]) 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.

13.8.5 stefanMelting2d

The solution for the melting problem (solid-liquid phase change) is found using the lattice Boltzmann method after Huang and Wu [31]. The equilibrium distribution function for the temperature is modified in order to deal with the latent-heat source term. That way, iteration steps or solving a group of linear equations is avoided, which results in enhanced efficiency. The phase interface is located by the current total enthalpy, and its movement is considered by the immersed moving boundary scheme after Noble and Torczynski [48]. Huang and Wu validated this method by the problem of conduction-induced melting in a semi-infinite space, comparing its results to analytical solutions.

13.9 turbulent

13.9.1 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.

13.9.2 channel3d

This example features the application of wallfunctions in a bi-periodic, fully developed turbulent channel flow for friction Reynolds numbers of $Re_\tau = 1000$ and $Re_\tau = 2000$. For the published results and further reference see [28].

13.9.3 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 ν 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`.

Figure 13.6 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)

Table 13.2: This table shows the preset simulation parameters.

parameter	value
$charL$	$1m$
$latticeL$	$\frac{1}{3}m$
$charU$	$1\frac{m}{s}$
ν	$0.00002\frac{m^2}{s}$
Re_{inlet}	5000
turbulence model	Smagorinsky

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. $rho = 1$).

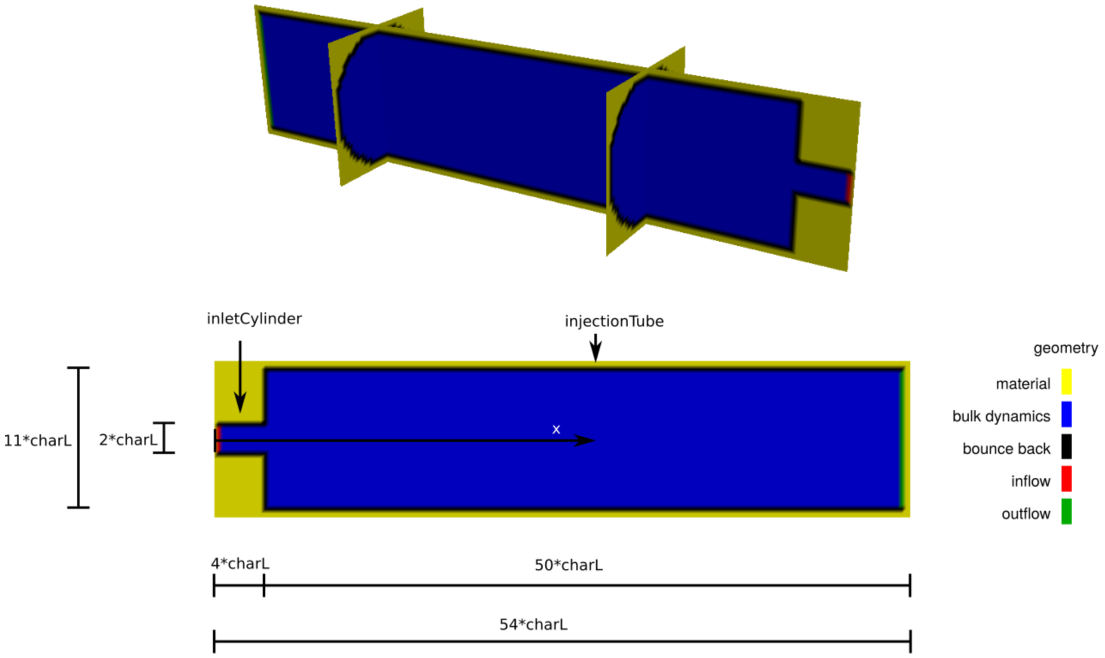


Figure 13.6: Cross section of a 3d geometry of nozzle3d in dependency of characteristic length $charL$.

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.

- 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.

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

#define Smagorinsky

...

#elif defined(Smagorinsky)
bulkDynamics = new SmagorinskyBGKdynamics<T, DESCRIPTOR>(converter.getOmega(), instances::getBulkMomenta<T, DESCRIPTOR>(),
0.04, converter.getLatticeL(), converter.physTime());
```

As an example, Figure 13.7 shows the results with preset parameters.

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.

13.9.4 tgv3d

The Taylor–Green vortex (TGV) is one of the simplest configurations to investigate the generation of small scale structures and the resulting turbulence. The cubic domain

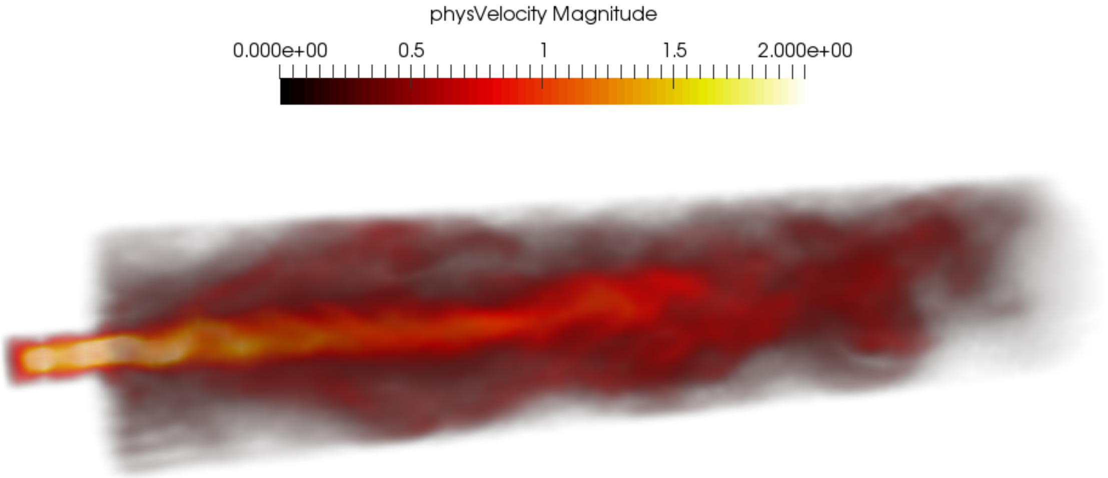


Figure 13.7: Physical velocity field after 200 seconds with preset parameters (Smagorinsky Model, $C_S = 0.15$, $latticeL = \frac{1}{3}m$, $Re_{inlet} = 5000$).

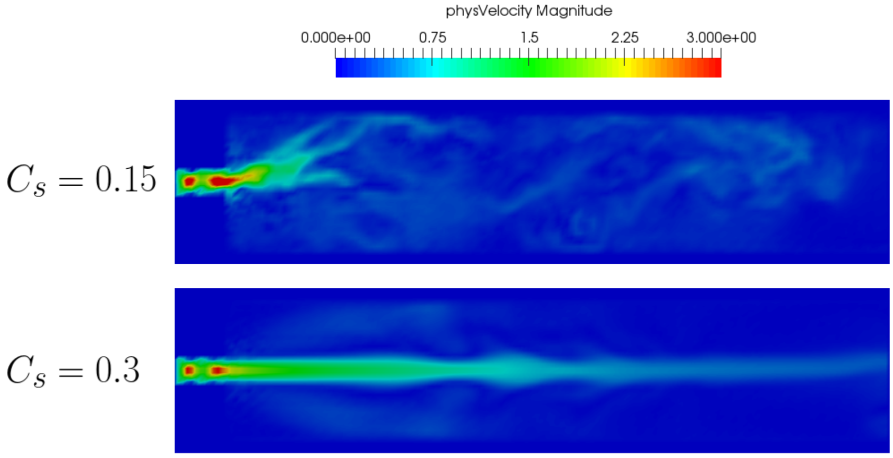


Figure 13.8: Increasing Smagorinsky Constant straightens turbulence.

$\Omega = (2\pi)^3$ with periodic boundaries and the single mode initialization contribute to the model's simplicity. In consequence, the TGV is a common benchmark case for direct numerical simulation (DNS) as well as large eddy simulation (LES). This example demonstrates the usage of different subgrid models and visualizes their effects on global turbulence quantities. The molecular dissipation rate, the eddy dissipation rate

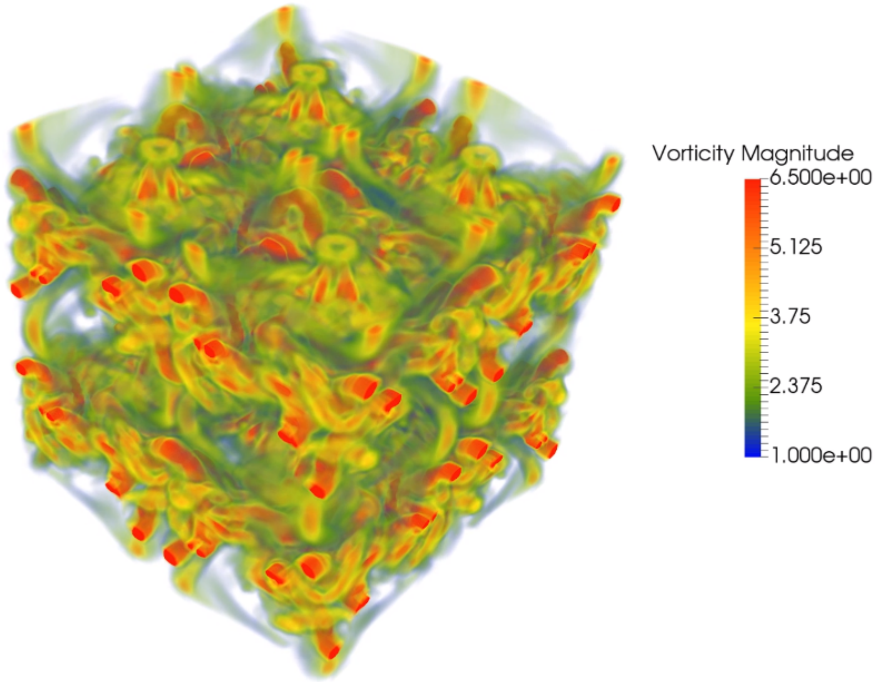


Figure 13.9: Isosurface of vorticity for the Taylor–Green vortex at $t = 12\ s$

and the effective dissipation rate are calculated and plotted over the simulation time. The results can be compared with a DNS solution published by Brachet et al. [12].

13.9.5 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 defind by certain points (position vetors) in the coordinate system and their radius. A cone-shaped cylinder needs the centre of the start an 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.

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. $rho = 1$).

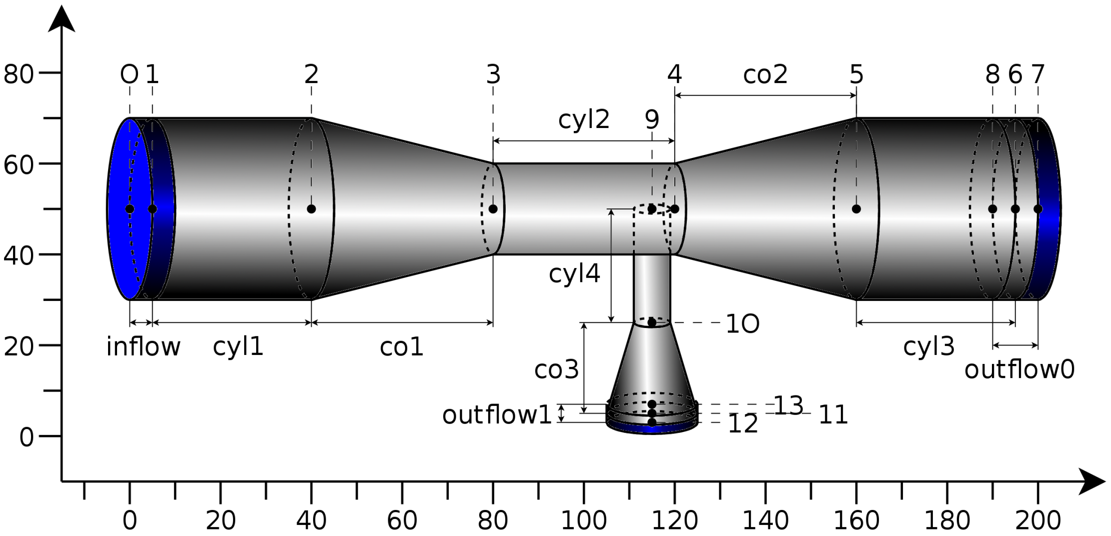


Figure 13.10: Schematic diagramm visualizing the defined points position.

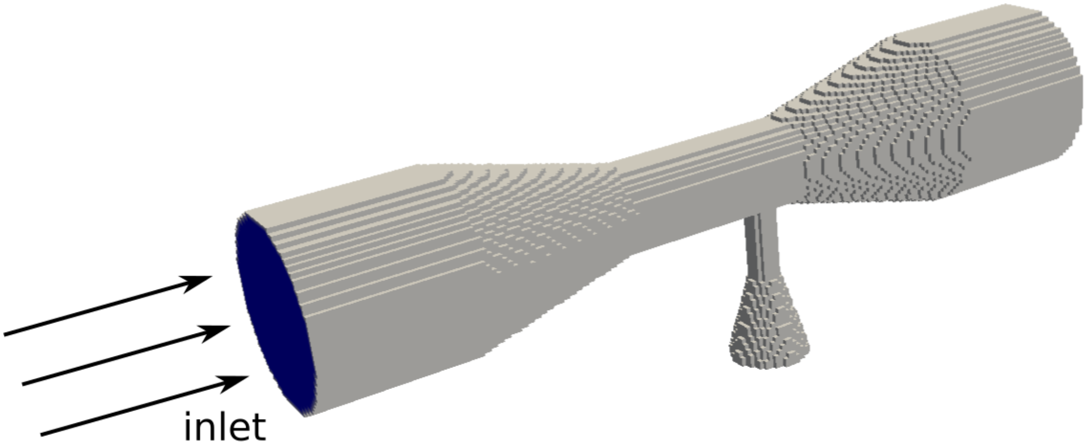


Figure 13.11: Built geometry as used in simulation.

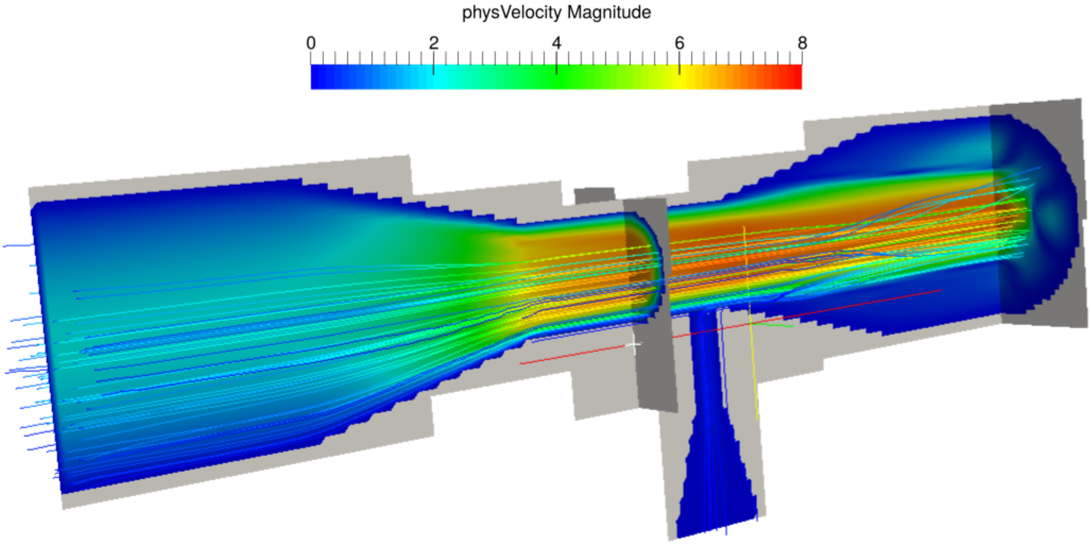


Figure 13.12: Simulation after 200 simulated time steps.

14 Q&A

In this Q&A part, some potential questions concerning the code are answered:

1. What do I need the unit converter for?

The unit converter (Listing 14.1) is used in every simulation done with OpenLB. In this class, the physical units, like length or mass, are converted to lattice units and vice versa This step is necessary to get a result in the correct physical dimensions and units.

```
    UnitConverterFromResolutionAndLatticeVelocity<T,DESCRIPTOR>
        converter(
    (int)    res,                     //resolution
    ( T )    charLatticeVelocity,     //charLatticeVelocity
    ( T )    charPhysLength,          //charPhysLength
    ( T )    charPhysVelocity,        //charPhysVelocity
    ( T )    physViscosity,           //physViscosity
    ( T )    physDensity              //physDensity
    );
    converter.print();
```

Listing 14.1: UnitConverter

For a closer look, also checkout the respective example in Sec. 2.4.

2. What are aliases used for?

Aliases are used to keep the code simpler for users and allow them to get a faster overview over the code. Aliases aren't used for all parts of the program. Especially if the user likes to change the code for a special problem, alias-functions may not be available and therefore need to be created by the user himself or normal functions can be used. To sum up, *alias*-functions aren't necessary for the simulation to work, but allow to simplify the code for a better overview for the user.

3. Why do we use "constexpr" to define some functions?

Constexpr allows you to get the output of the function at the time of compilation. The value returned is set to a constant expression and as a result the runtime can be reduced, due to the constant value. If the return value is part of an if-loop, it won't be checked more than once, as the result is already clear, after the first check. In the example below the rotation of a 2D-particle is calculated and therefore the output is set to a concrete value at compilation time, and this value stays constant.

```
static constexpr Vector<T,2> execute( Vector<T,2> input, Vector<T,4> rotationMatrix, Vector<T,2> rotationCenter = Vector<T,2>(0.,0.) )
{
        Vector<T,2> dist = input - rotationCenter;
        return Vector<T,2>(
        dist[0]*rotationMatrix[0] +
        dist[1]*rotationMatrix[2],
        dist[0]*rotationMatrix[1] +
        dist[1]*rotationMatrix[3] );
}
```

Listing 14.2: constexpr

4. What is the difference between files ending with *.h* and ending with *.hh*?

The files ending with *.h* like *particleDynamics.h* are header-files, where classes and functions are only declared. The specification (definition) happens in the *.hh* files. In the following two code-snippets of the same function, the differences in terms of specification can be compared.

```
template<typename T, typename DESCRIPTOR>
class VerletParticleDynamics : public ParticleDynamics<T, DESCRIPTOR> {
        public:
        /// Constructor
        VerletParticleDynamics( T timeStepSize );
        /// Procesisng step
        void process (Particle<T,DESCRIPTOR>& particle) override;
        private:
        T _timeStepSize;
};
```

Listing 14.3: particleDynamics.h

```
    template<typename T, typename PARTICLETYPE>
    VerletParticleDynamics<T,PARTICLETYPE>::VerletParticleDynamics ( T timeStepSize )
    : _timeStepSize( timeStepSize )
    {
        this->getName() = "VerletParticleDynamics";
    }

    template<typename T, typename PARTICLETYPE>
    void VerletParticleDynamics<T,PARTICLETYPE>::process (
    Particle<T,PARTICLETYPE>& particle )
    {
        //Calculate acceleration
        auto acceleration = getAcceleration<T,PARTICLETYPE>( particle );
        //Check for angular components
        if constexpr ( providesAngle<PARTICLETYPE>() ) {
            //Calculate angular acceleration
            auto angularAcceleration = getAngAcceleration<T,PARTICLETYPE>( particle );
            //Verlet algorithm
            particles::dynamics::velocityVerletIntegration<T, PARTICLETYPE>(
            particle, _timeStepSize , acceleration, angularAcceleration );
            //Update rotation matrix
            updateRotationMatrix<T,PARTICLETYPE>( particle );
        } else {
            //Verlet algorithm without rotation
            particles::dynamics::velocityVerletIntegration<T, PARTICLETYPE>(
            particle, _timeStepSize , acceleration );
        }
    }
```

Listing 14.4: particleDynamics.hh

Bibliography

[1] LB model with adjustable speed of sound. Technical report. http://www.openlb.net/tech-reports.

[2] How to implement your DdQq dynamics with only q variables per node. Technical report. http://www.openlb.net/tech-reports.

[3] Installing OpenLB in Windows 10/Ubuntu bash. Technical Report. http://www.openlb.net/tech-reports.

[4] FreeCAD: An Open Source parametric 3D CAD modeler. https://www.freecadweb.org/.

[5] The OpenGPI project. http://www.opengpi.org.

[6] The VTK data format documentation. http://www.vtk.org/VTK/img/file-formats.pdf.

[7] The Paraview project. http://www.paraview.org.

[8] S. Ansumali. "Minimal kinetic modeling of hydrodynamics". PhD thesis. Swiss Federal Institute of Technology Zurich, 2004.

[9] B. F. Armaly, F. Durst, J. C. F. Pereira, and B. Schönung. "Experimental and theoretical investigation of backward-facing step flow". In: *Journal of Fluid Mechanics* 127 (1983), pp. 473–496. DOI: 10.1017/S0022112083002839.

[10] T. Borrvall and J. Petersson. "Topology optimization of fluids in Stokes flow". In: *International Journal for Numerical Methods in Fluids* 41.1 (2003), pp. 77–107. DOI: 10.1002/fld.426.

[11] M. Bouzidi, M. Firdaouss, and P. Lallemand. "Momentum transfer of a Boltzmann-lattice fluid with boundaries". In: *Physics of Fluids* 13.11 (2001), pp. 3452–3459. DOI: 10.1063/1.1399290.

[12] M. E. Brachet, D. I. Meiron, S. A. Orszag, B. Nickel, R. H. Morf, and U. Frisch. "Small-scale structure of the Taylor–Green vortex". In: *Journal of Fluid Mechanics* 130 (1983), pp. 411–452.

[13] H. Brinkman. "A calculation of the viscous force exerted by a flowing fluid on a dense swarm of particles". English. In: *Applied Scientific Research* 1.1 (1949), pp. 27–34. DOI: 10.1007/BF02120313.

[14] H. Brinkman. "On the permeability of media consisting of closely packed porous particles". English. In: *Applied Scientific Research* 1.1 (1949), pp. 81–86. DOI: 10.1007/BF02120318.

[15] A. Caiazzo and M. Junk. "Asymptotic analysis of lattice Boltzmann methods for flow-rigid body interaction". In: *Progress in Computational Physics* 3 (2013), p. 91.

[16] S. Chen and G. D. Doolen. "Lattice Boltzmann Method for Fluid Flows". In: *Ann. Rev. Fluid Mech.* 30 (1998), pp. 329–364.

[17] B. Chopard, A. Dupuis, A. Masselot, and P. Luthi. "Cellular Automata and Lattice Boltzmann techniques: an approach to model and simulate complex systems". In: *Adv. Compl. Sys.* 5 (2002), pp. 103–246. DOI: 10.1142/S0219525902000602.

[18] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.-S. Luo. "Multiple-relaxation-time lattice Boltzmann models in three dimensions". In: *Phil. Trans. R. Soc. Lond. A* 360 (2002), pp. 437–451.

[19] D. d'Humières, M. Bouzidi, and P. Lallemand. "Thirteen-velocity three-dimensional lattice Boltzmann model". In: *Phys. Rev. E* 63 (2001), p. 066702. DOI: 10.1103/PhysRevE.63.066702.

[20] D. Dapelo, S. Simonis, M. J. Krause, and J. Bridgeman. "Lattice-Boltzmann coupled models for advection–diffusion flow on a wide range of Péclet numbers". In: *Journal of Computational Science* 51 (2021), p. 101363. DOI: https://doi.org/10.1016/j.jocs.2021.101363.

[21] T. Dornieden. "Optimierung von Strömungsgebieten mit adjungierten Lattice Boltzmann Methoden". Diplomarbeit. Karlsruhe Institute of Technology (KIT), 2013.

[22] A. Fakhari and M. H. Rahimian. "Phase-field modeling by the method of lattice Boltzmann equations". In: *Physical Review E* 81.3 (2010), p. 036707.

[23] C. Gau and R. Viskanta. "Melting and Solidification of a Pure Metal on a Vertikal Wall". In: *Journal of Heat Transfer* 108.1 (1986), pp. 174–181.

[24] Z. Guo, C. Zheng, and B. Shi. "Discrete lattice effects on the forcing term in the lattice Boltzmann method". In: *Phys. Rev. E* 65 (2002), p. 046308.

[25] Z. Guo, B. Shi, and C. Zheng. "A coupled lattice BGK model for the Boussinesq equations". In: *Int. J. Num. Meth. Fluids* 39 (2002), pp. 325–342. DOI: 10.1002/fld.337.

[26] Z. Guo, B. Shi, and C. Zheng. "A coupled lattice BGK model for the Boussinesq equations". In: *International Journal for Numerical Methods in Fluids* 39.4 (2002), pp. 325–342.

[27] X.-Y. L. H-B Huang and M. C. Sukop. "Numerical study of lattice Boltzmann methods for a convection-diffusion equation coupled with Navier-Stokes equations". In: *J. Phys. A: Math. Theor.* 44.5 (2011).

[28] M. Haussmann, A. C. BARRETO, G. L. KOUYI, N. Rivière, H. Nirschl, and M. J. Krause. "Large-eddy simulation coupled with wall models for turbulent channel flows at high Reynolds numbers with a lattice Boltzmann method — Application to Coriolis mass flowmeter". In: *Computers & Mathematics with Applications* 78.10 (2019), pp. 3285–3302. DOI: 10.1016/j.camwa.2019.04.033.

[29] M. Haussmann, S. Simonis, H. Nirschl, and M. J. Krause. “Direct numerical simulation of decaying homogeneous isotropic turbulence—numerical experiments on stability, consistency and accuracy of distinct lattice Boltzmann methods”. In: *International Journal of Modern Physics C* 30.09 (2019), p. 1950074.

[30] T. Henn, G. Thäter, W. Dörfler, H. Nirschl, and M. J. Krause. “Parallel dilute particulate flow simulations in the human nasal cavity”. In: *Computers & Fluids* 124 (2016), pp. 197–207.

[31] R. Huang and H. Wu. “Phase interface effects in the total enthalpy-based lattice Boltzmann model for solid–liquid phase change”. In: *Journal of Computational Physics* 294 (2015), pp. 346–362.

[32] G. Kałuża. “The numerical solution of the transient heat conduction problem using the lattice Boltzmann method”. In: *Scientific Research of the Institute of Mathematics and Computer Science* 11.1 (2012), pp. 23–30.

[33] A. Komrakova, O. Shardt, D. Eskin, and J. Derksen. “Lattice Boltzmann simulations of drop deformation and breakup in shear flow”. In: *International Journal of Multiphase Flow* 59 (2014), pp. 24–43. DOI: https://doi.org/10.1016/j.ijmultiphaseflow.2013.10.009.

[34] M. J. Krause, F. Klemens, T. Henn, R. Trunk, and H. Nirschl. “Particle flow simulations with homogenised lattice Boltzmann methods”. In: *Particuology* 34 (2017), pp. 1–13. DOI: https://doi.org/10.1016/j.partic.2016.11.001.

[35] M. J. Krause et al. “OpenLB–Open source lattice Boltzmann code”. In: *Computers & Mathematics with Applications* (2020). DOI: https://doi.org/10.1016/j.camwa.2020.04.033.

[36] M. J. Krause. “Fluid Flow Simulation and Optimisation with Lattice Boltzmann Methods on High Performance Computers: Application to the Human Respiratory System”. eng. PhD thesis. Kaiserstrasse 12, 76131 Karlsruhe, Germany: KIT, Universität Karlsruhe, 2010.

[37] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E. M. Viggen. *The Lattice Boltzmann Method*. Springer, 2017.

[38] A. Ladd and R. Verberg. “Lattice-Boltzmann simulations of particle-fluid suspensions”. In: *Journal of Statistical Physics* 104.5-6 (2001), pp. 1191–1251.

[39] D. Lagrava, O. Malaspinas, J. Latt, and B. Chopard. “Automatic grid refinement criterion for lattice Boltzmann method”. In: *ArXiv e-prints* (July 2015).

[40] J. Latt and B. Chopard. “Lattice Boltzmann Method with regularized non-equilibrium distribution functions”. In: *Math. Comp. Sim.* 72 (2006), pp. 165–168.

[41] L. Li, R. Mei, and J. F. Klausner. “Boundary conditions for thermal lattice Boltzmann equation method”. In: *Journal of Computational Physics* 237 (2013), pp. 366–395.

[42] Q. Liu and Y.-L. He. "Double multiple-relaxation-time lattice Boltzmann model for solid–liquid phase change with natural convection in porous media". In: *Physica A: Statistical Mechanics and its Applications* 438 (2015), pp. 94–106.

[43] A. Mezrhab, M. A. Moussaoui, M. Jami, H. Naji, and M. Bouzidi. "Double MRT thermal lattice Boltzmann method for simulating convective flows". In: *Physics Letters A* 374.34 (2010), pp. 3499–3507.

[44] S. C. Mishra and H. K. Roy. "Solving transient conduction and radiation heat transfer problems using the lattice Boltzmann method and the finite volume method". In: *Journal of Computational Physics* 223.1 (2007), pp. 89–107.

[45] A. A. Mohamad. *Lattice Boltzmann Method - Fundamentals and Engineering Applications with Computer Codes*. Springer-Verlag, 2011.

[46] A. Mohamad and A. Kuzmin. "A critical evaluation of force term in lattice Boltzmann method, natural convection problem". In: *International Journal of Heat and Mass Transfer* 53.5 (2010), pp. 990–996.

[47] P. Nathen, D. Gaudlitz, M. J. Krause, and J. Kratzke. "An extension of the Lattice Boltzmann Method for simulating turbulent flows around rotating geometries of arbitrary shape". In: *21st AIAA Computational Fluid Dynamics Conference*. American Institute of Aeronautics and Astronautics. 2013. DOI: doi:10.2514/6.2013-2573.

[48] D. Noble and J. Torczynski. "A lattice-Boltzmann method for partially saturated computational cells". In: *Int. J. Modern Phys. C* 9.8 (1998), pp. 1189–1202.

[49] Y. Peng, C. Shu, and Y. Chew. "Simplified thermal lattice Boltzmann model for incompressible thermal flows". In: *Physical Review E* 68.2 (2003), p. 026701.

[50] G. Pingen, A. Evgrafov, and K. Maute. "Topology optimization of flow domains using the lattice Boltzmann method". English. In: *Structural and Multidisciplinary Optimization* 34.6 (2007), pp. 507–524. DOI: 10.1007/s00158-007-0105-7.

[51] R. Rannacher. *Einfuehrung in die Numerische Mathematik (Numerik 0)*. Vorlessungsskriptum SS 2005. Universitaet Heidelberg, 2006.

[52] C. Semprebon, T. Krüger, and H. Kusumaatmaja. "A Ternary Free Energy Lattice Boltzmann Model with Tunable Surface Tensions and Contact Angles". In: *Physical Review E* 93.3 (2016), p. 033305.

[53] X. Shan and H. Chen. "Lattice Boltzmann model for simulating flows with multiple phases and components". In: *Phys. Rev. E* 47 (1993), pp. 1815–1819. DOI: 10.1103/PhysRevE.47.1815.

[54] X. Shan and G. Doolen. "Multicomponent lattice-Boltzmann model with interparticle interaction". In: *Journal of Statistical Physics* 81 (1995), pp. 379–393.

[55] S. Simonis, M. Frank, and M. J. Krause. "On relaxation systems and their relation to discrete velocity Boltzmann models for scalar advection–diffusion equations". In: *Phil. Trans. R. Soc. A* 378.2175 (2020), p. 20190400. DOI: 10.1098/rsta.2019.0400.

[56] M. A. A. Spaid and F. R. Phelan. "Lattice Boltzmann methods for modeling microscale flow in fibrous porous media". In: *Physics of Fluids* 9.9 (1997), pp. 2468–2474. DOI: 10.1063/1.869392.

[57] S. Stasius. "Identifikation von Strömungsgebieten mit adjungierten Lattice Boltzmann Methoden (ALBM)". Diplomarbeit. Karlsruhe Institute for Technology (KIT), 2014.

[58] R. Trunk, T. Henn, W. Dörfler, H. Nirschl, and M. Krause. "Inertial Dilute Particulate Fluid Flow Simulations with an Euler-Euler Lattice Boltzmann Method". In: *Journal of Computational Science* 17, Part 2 (2016), pp. 438–445. DOI: http://dx.doi.org/10.1016/j.jocs.2016.03.013.

[59] R. Trunk, T. Weckerle, N. Hafen, G. Thäter, H. Nirschl, and M. J. Krause. "Revisiting the Homogenized Lattice Boltzmann Method with Applications on Particulate Flows". In: *Computation* 9.2 (2021). DOI: 10.3390/computation9020011.

[60] S. Turek and M. Schäfer. "Benchmark computations of laminar flow around cylinder". In: *Flow Simulation with High-Performance Computers II*. Vol. 52. Notes on Numerical Fluid Mechanics. Vieweg, Jan. 1996, pp. 547–566.

[61] D. Yu, R. Mei, L.-S. Luo, and W. Shyy. "Viscous flow computations with the method of lattice Boltzmann equation". In: *Progress in Aerospace Sciences* 39.5 (2003), pp. 329–367.

[62] H. Zheng, C. Shu, and Y.-T. Chew. "A lattice Boltzmann model for multiphase flows with large density ratio". In: *Journal of Computational Physics* 218.1 (2006), pp. 353–371.

15 License

GNU Free Documentation License

Version 1.2, November 2002

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 Document 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 nothing 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 composed of pixels) generic paint programs or (for drawings) some widely available drawing 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 Document, 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 published 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 Title 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.