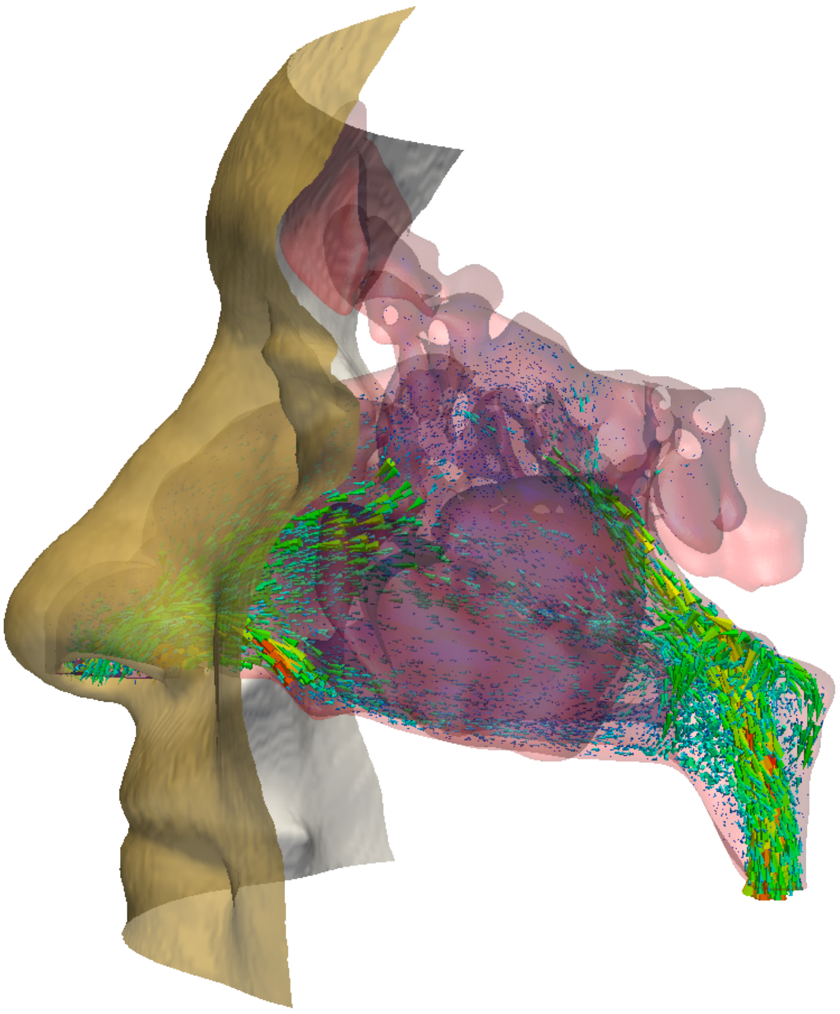
OpenLB User Guide
Associated to Release 0.9 of the Code



openlb.net

Contents

1 Preface

Aims of the user guide

2 Introduction

2.1 Fluid Flow Simulations

2.2 Lattice Boltzmann Methods

This text is directed to people who want to get in touch with *Lattice-Boltzmann Methods (LBM).*

- The most recent publication we refer to, has been written by **Erlend Magnus Viggen**. His Phd Thesis THE LATTICE BOLTZMANN METHOD: FUNDAMENTALS AND ACOUSTICS published in 2014, delivers a clear and complete introduction for beginners. In particular we want to mention Chapter 3 and 4, where he develops the fundamentals, like *theory of gas kinetics* and *the Boltzmann equation.*

- A brief an concise introduction is given by **A. A. Mohamad**. In his book LATTICE BOLTZMANN METHOD [2011], he shows in a clear way, how to get macroscopic equations from *LBM* using *Chapman-Enskop expansion.*

- The reader how want gain insight to *Lattice-Gas Cellular Automatas* - the historical origin of *LBM* - may have a look on **Dieter A. Wolf-Glodrows** book LATTICE-GAS CELLULAR AUTOMATA AND LATTICE BOLTZMANN MODELS [2000]. Starting with the ancient *Cellular Automatas*, he deploys the whole beauty of *LBM.* A nice and helpful interpretation of *LBM* is given in the beginning of the book.

- In order get a quick overview of *LBM*, we refer to often cited paper of **S. Chen** and **G. D. Doolen** LATTICE BOLTZMANN METHOD FOR FLUID FLOWS published in 1998.

2.3 The OpenLB Project

2.3.1 What is OpenLB?

OpenLB is a numerical framework for lattice Boltzmann simulations, created by students and researchers with different background 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. To please the first audience, OpenLB offers a neat interface through which it is possible to set up a simulation with little effort. For the second audience, 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. Basically, generic programming is intended 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.

2.3.2 How to get help with OpenLB?

The following resources are available for OpenLB users:

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

Forum. If you experience troubles with OpenLB, you may wish to post your concerns to the Lattice Boltzmann community on the forum at 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 like buildtype and mode of parallelization found in `Makefile.inc` can serve useful information, too.

2.3.3 How to compile OpenLB programs?

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

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. In 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 `Makefile.inc` in the root directory can be edited (it is easy to understand!) to modify the compilation process. Available options include the choice of the compiler (`GNU g++` is the default), optimization flags, and a switch between normal/debug mode, and between sequential/openmp-parallel/mpi-parallel programs.

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

2.3.4 What features are currently implemented?

Lattice Boltzmann models

BGK model for fluids	Section 6.1.3	Reference [2]
Regularized model for fluids	Section 6.1.3	Reference [3]
Multiple relaxation times (MRT)	Section 6.1.3	References [4, 5]
Entropic Lattice Boltzmann	Section 6.1.3	Reference [6]
BGK with adjustable speed of sound	Section 6.1.3	References [7, 8]
BGK and MRT with Smagorinsky model	Section 6.1.3	References [9]
Porous media model	Section 6.1.3	

Multiphysics coupling

Shan-Chen two-component fluid	Section 6.6	Reference [10]
Thermal fluid with Boussinesq approximation	Section 6.6	Reference [11]

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. [12])
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 [13]

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 `MultiBlock-LatticeXD`, both of which have similar functionality but a slightly different scope. Those advanced data structures generate a patchwork consisting of many `BlockLatticeXD` structures that are presented behind a unified interface. Applications of these structures are MPI-parallelism and memory saving simulations that do not allocate memory in chosen subdomains of the numerical grid.

Input / Output

The basic mechanism behind I/O operations in OpenLB is the serialization and unserialization of a `BlockLatticeXD` and a `DataFieldXD`. 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.

2.3.5 Participants

In 2006 the OpenLB project was initiated. Between 2006 and 2008 Jonas Latt was the project coordinator. Since 2009 Mathias J. Krause is coordinating the project. Since 2006 the following persons have contributed source code to OpenLB:

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

Vojtech Cvrcek (active): functors: 2D adaption, **dynamics**: power law, **examples**: updates

Tim Dornieden: functors: smooth start scaling

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

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

Fabian Klemens (active): functors: flux

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, **functors**: concept, div. functors implementation, **examples**: venturi3d, aorta3d

Jonas Latt: core: basic block structure, **communication**: basic parallel block lattice approach (< release 0.9), **general**: integration and maintenance of added components (2006-2008), **boundaries**: basic boundary structure, **dynamics**: basic dynamics structure, **examples**: many examples which were further developed in recent years

Marie-Luise Maier (active): particles: particulate flows

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

Cyril Masquelier (active): functors: indicator, smooth indicator

Albert Mink (active): functors: aritmethic, **io**: parallel VTK interface

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

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

Robin Trunk (active): dynamics: parallel thermal models

Peter Weisbrod (active): dynamics: parallel multi phase/component, **examples**: structure and show cases, phaseSeparationXd

Gilles Zahnd: functors: rotating frame functors

Simon Zimny: io: pre-processing: automated setting of boundary conditions

3 Using OpenLB for Applications

The general way of functioning in OpenLB follows a generic path.

1st Step: Initialization The converter between physics and lattice is set in this step. The parameters for the simulation setup are chosen here, too, if they have not already been set at the beginning.

2nd Step: Prepare Geometry It first gets the geometry either from another file (a stl file here) or from defining indicator functions. Then it creates a mesh from that, and prepares the geometry required. 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. For these tasks the function `prepareGeometry` is called. Some examples and applications which use a rather simple geometry skip this step.

3rd Step: Prepare Lattice The lattice dynamics are set here. The kinds of dynamics are chosen between the different implementations. These possibilities depend on force acting or not, the single relaxation time (BGK) used or the multi relaxation time (MRT), the simulation dimension (it can also be a 2D model), a compressible or incompressible fluid considered, and the number of neighbouring voxel chosen. The boundary condition initialisation is done in order to enable any kind of them. The lattice is then defined in the function `prepareLattice`, with the boundary condition choices for every material number, and for which materials number which dynamics are applied. It only defines the kind of boundary (like Bouzidi, bounce-back, velocity, or pressure) but not the profile function itself.

4th Step: Main Loop with Timer The timer is initialized and started, then a loop over all time steps `iT` eventually starts the simulation during which the functions `setBoundaryValues`, `collideAndStream` and `getResults` (step 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 results are printed.

5th Step: Definition of Initial and Boundary Conditions The first of the three important functions called during the loop, `setBoundaryValues`, puts into practice the boundary functions' values. In some applications it needs to refresh them during each time step, in others they 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, which performs the collision and the streaming step. If more than one lattice is used, the function is called for each of them seperately.

7th Step: Computation and Output of the Results At the end of each iteration step, the function `getResults` is called, which creates console output, `.gif` files or `.vtk` files of the results at certain timesteps.

This structuration is the very same in every OpenLB simulation, only the choices made are changing the simulation: every real modification is done in the called functions, to prepare the geometry, the converter, the lattice, and the boundary profiles. Every change has to match to OpenLB's implementation, so new models might need changes or adds in the source code. For example, the classes defined in the code are always issued from a mother-class and have to match to the inputs' functions, which may sometimes lead to unexpected issues to solve.

3.1 Lesson 1: - A Typical Application Program Structure: Implement your first OpenLB program

Unpack the OpenLB tar-ball on your system, and compile one of the example programs. If this is successful, create a directory for this tutorial at the location of your choice. Create a Makefile in this directory according to the procedure explained in Section 2.3.3.

A few lines are invariably the same from one OpenLB program to another:

Listing 1: Framework of an OpenLB program

```
1  #include "olb2D.h"
2  #ifndef OLB_PRECOMPILED // Unless precompiled version is used,
3    #include "olb2D.hh"   // include full template code
4  #endif
5
6  using namespace olb;
```

Some lines in this program deserve additional comments:

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

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

Line 6: All OpenLB code is contained in the namespace `std`.

Furthermore, for the following examples to compile, the following declarations need to be included into Listing 1 between Line 4 and 6:

```
1   #include <vector>    // Some C++ libraries which are
2   #include <cmath>     //   required for the following
3   #include <iostream>  //   examples
4   #include <iomanip>
5   #include <fstream>
6
7   using namespace olb;  // OpenLB namespaces which are
8   using namespace olb::descriptors; // accessed in the
9   using namespace olb::graphics;    // examples
10  using namespace std;  // Namespace of standard C++ library
```

At this point, the code for the simulation of a fluid flow can be inserted at the place of line 10. The following simple example represents a fluid initially at rest with a slightly increased particle density within a disk around the center. The flow is modelized through the single relaxation-time BGK model, and it evolves in a system with periodic boundaries. *(It should be pointed out that this example is only used to illustrate programming issues. The chosen initial condition does not really represent a physically meaningful state of an incompressible fluid. The example "works" because the LB model is contrived into adopting a compressible regime. Interpreting the results of a BGK model under the light of compressible flows raises however numerous issues of its own that cannot be covered here. Thus, look at the code and learn your lesson, but don't attribute too much meaning to the numerical result.)*

Listing 2: to be inserted at Line 10 of Listing 1

```
#define LATTICE D2Q9Descriptor
typedef double T;
int nx = 20;
int ny = 30;
int numIter = 100;
T omega = 1.;
T r     = 5.;

int main(int argc, char* argv[]) {
  olbInit(&argc, &argv);
  // Insert the central part of your code here
  BlockLattice2D<T, LATTICE> lattice(nx, ny);
  BGKdynamics<T, LATTICE> bulkDynamics (
      omega,
      instances::getBulkMomenta<T,LATTICE>()
  );
  lattice.defineDynamics(0,nx-1,0,ny-1, &bulkDynamics);

  for (int iX=0; iX<nx; ++iX) {
      for (int iY=0; iY<ny; ++iY) {
          T rho=1., u[2] = {0.,0.};
          if ((iX-nx/2)*(iX-nx/2) + (iY-ny/2)*(iY-ny/2) < r*r) {
              rho = 1.01;
24            }
25            lattice.get(iX,iY).iniEquilibrium(rho,u);
26        }
27  }
28
29  for (int iT=0; iT<numIter; ++iT) {
30      lattice.collide();
31      lattice.stream(true);
32  }
33
34  ImageWriter<T> imageWriter("leeloo");and the dynamic too afterwards
35  imageWriter.writeScaledGif (
36      "lesson1",
37      lattice.getDataAnalysis().getVelocityNorm() );
38 }
```

A few explanations are again in order:

Line 1: Choice of a lattice descriptor. Lattice descriptors specify not only what lattice you are going to use (for 2D simulations, the current OpenLB release gives you no choice but D2Q9 anyway), but also potentially the nature of additional scalars, such as an external force field, for which memory needs to be allocated on a grid cell.

Line 2: 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.

Lines 3-7: Constants to specify the dimensions of the `nx`$\times$`ny` lattice and the total number `numIter` of iteration steps. The relaxation parameter ω is the inverse of the relaxation time τ. It determines the value of the shear viscosity ν of the fluid.

Line 10: This line is gratuitous in sequential programs, but it is required in the context of MPI-parallelism (which is explained in Lesson 10). As a general rule, you will always want your program to be ready for both sequential and parallel executions. It is therefore good practice to include this line as a matter of routine, in all cases.

Line 12: Instantiation of a `BlockLattice2D` object. At this point, memory for the `nx`$\times$`ny`$\times$9 particle populations is allocated. If additional memory has been requested for external scalars (this is not the case here), this memory is also allocated during the instantiation of the `BlockLattice2D`.

Lines 13-16: The `Dynamics` object determines the implementation of the collision step on grid nodes, in this case BGK [2]. Objects of type `BGKdynamics` can be customized by indicating how the moments of distribution functions (particle density, velocity, etc.) should be computed. By choosing a specific `Momenta` object, one can for example implement boundary conditions in which the dynamics is the same as in the bulk, but the momenta are computed differently because of missing particle populations. In the present example, a default implementation is chosen for the computation of the momenta.

Line 17: The previously instantiated dynamics is to be used on all lattice nodes. The domain on which to instantiate the dynamics is indicated explicitly, the x-index ranging from `0` to `nx-1`, and the y-index from `0` to `ny-1`.

Line 25: Initialize particle populations at an equilibrium distribution, with slightly increased density inside a circle of radius `r`.

Line 30: At each iteration step, the collisand the dynamic too afterwardsion specified by the variable `bulkDynamics` is applied to each grid node.

Line 31: After collision follows the streaming step. The boolean argument `true` indicates that boundaries are periodic.

Line 34: The `ImageWriter` offers a means of producing 2D images of format `PPM`. If the package `ImageMagick` is installed on your machine, you can also get `GIF` images. Four colormaps are available for each of the four elements ("earth", "water", "air", "fire") and one for the fifth element "leeloo" (see Ref. [14]).

Line 37: An object of type `DataAnalysis2D` is instantiated to extract the norm of the velocity from the numerical result. From this, an image is created with help of the `ImageWriter`, by rescaling the colormap to the range of values adopted by the velocity norm in the numerical result.

You can easily observe that boundary conditions are periodic by playing around with the code and producing images at various time steps. Alternatively, no-slip walls are implemented by calling the method `BlockLattice2D::stream()` in line 28 with an argument `false`. This is the default argument, and the method can therefore be invoked with no argument at all:

Listing 3: Substitutions to replace periodic boundaries by no-slip walls

```
lattice.collide();
lattice.stream();
```

These no-slip walls are obtained through a so-called *halfway bounceback* mechanism: particle populations on boundary cells, which would leave the computational domain during streaming, stay on the cell and their value is copied to the particle population with opposite velocity vector instead. After this, the usual collision step is executed. No efficiency overhead is incurred for the implemention of this mechanism, because it is an automatic side-effect of the algorithm in OpenLB for the streaming step [15].

3.2 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 was tested so far, and it is negligible in face of other, big-picture efficiency considerations.

One such consideration is about the separation between collision and streaming at Line 28 and 29 of Listing 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 factand the dynamic too afterwards, there are several ways of avoiding this time-consuming double access to memory, one of which is implemented in OpenLB and documented in Ref. [15]. For an OpenLB user, doing this is as easy as replacing the collision-streaming sequence by a call to the method `collideAndStream()`:

Listing 4: Collision and streaming in one step for improved efficiency

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

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

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

Listing 5: Direct access to values in a BlockLattice2D

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

The method `BlockLattice2D<T, LATTICE>::get()` delivers an object of type `Cell<LATTICE>`, which contains storage space for the particle populations and, if so required by the `LATTICE` template, for additional scalars. The `Cell` offers many methods to read and manipulate the data. You are much more likely to use those methods in practice, rather than accessing particle populations directly as in Listing 5:

Listing 6: Manipulation of data through methods of a Cell

```
  int nx, ny, someX, someY, someF;and the dynamic too afterwards
// <...> some code to initialize nx, ny, someX and someY
  BlockLattice2D<T, LATTICE> 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
```

In this example, the method `Cell<T>::computeU()` computes the velocity on a cell for you, using its dynamics object. Inversely, 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.

Additionally to being more convenient, the access to the data in Listing 6 has a distinct advantage to the approach of Listing 5: in Listing 5 the data inside a `Cell<T>` is accessed directly, whereas in Listing 6 it is accessed indirectly through the dynamics object of the cell. Although direct data access works in simple data structures 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 in 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<T>` are separated into two groups, one for direct data access, and one for indirect data access through 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 library OpenLB, for example by writing a class to implement a new type of dynamics. Most subsequent lessons are written for application programmers, and the code is written with extensibility in mind, insisting for example on the possibility to run it in parallel with minimal changes.

The following is a list of some useful methods to access the data of a `Cell<T>` 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.

and the dynamic too afterwards **void computeStress (T pi[util::TensorVal⟨Lattice⟨T⟩⟩::n]) const**
Compute the off-equilibrium stress-tensor $\Pi^{(1)}$ on the cell.

void computePopulations(T* f) const
Retrieve the particle populations and store them in a q-element C-array.

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

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

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

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

void definePopulations(const T* f)
Attribute new values to all populations. The argument `f` is a C-array with `q` elements.

void defineExternalField(int pos, int size, const T* ext)
Attribute new values to all external scalars.

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

Listing 7: Instantiation of a 3D lattice

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

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.

3.3 Lesson 3: Define and use boundary conditions

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

Listing 8: Instantiation of OnLatticeBoundaryCondition

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

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

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

Listing 9: Instantiation of velocity boundary condition along lattice boundaries

```
  template<typename T>
  void velocityBoundaryBox (
      BlockLattice2D<T,D2Q9Descriptor>& lattice,
      OnLatticeBoundaryCondition2D<T,D2Q9Descriptor>& bc, T omega)
  {
      int nx = lattice.getNx();
      int ny = lattice.getNy();
      // top boundary
      bc.addVelocityBoundary1P(1,nx-2,ny-1,ny-1, omega);
      // bottom boundary
      bc.addVelocityBoundary1N(1,nx-2,   0,   0, omega);
      // left boundary
      bc.addVelocityBoundary0N(0,0, 1, ny-2, omega);
      // right boundary
```

```
        bc.addVelocityBoundary0P(nx-1,nx-1, 1, ny-2, omega);

        // Corner nodes
        bc.addExternalVelocityCornerNN(0,0, omega);
        bc.addExternalVelocityCornerNP(0,ny-1, omega);
        bc.addExternalVelocityCornerPN(nx-1,0, omega);
        bc.addExternalVelocityCornerPP(nx-1,ny-1, omega);

        // Make the lattice ready for simulation
        lattice.initialize();
    }
```

When boundary nodes are instantiated, it is necessary to specify the orientation of the boundary through the normal vector that points outside of the domain. The instruction `addVelocity-Boundary1P` refers to a boundary whose normal is in positive y-direction (`P` stands for "positive", and indexes are numbered as `0` for the x-index and `1` for the y-index). For external corners, the expression `NN` refers to any boundary vector whose opposite direction points inside the numerical domain. In this case, this boundary vector points in negative x-direction and negative y-direction. The term `External` in the method `addExternalVelocityCornerNN` refers to the fact that the domain boundaries are convex shaped. Corners of concave shaped boundaries are instantiated with methods of the form `addInternalVelocityCornerXX`, where `X` stands again for `N` or `P` and indicates the direction of a vector pointing outside the numerical domain.

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

Things are slightly more complicated in 3D, where edges also need seperate treatment. Edges are locations where two boundary surfaces that are orthogonal to each other meet. The following are typical instructions one may use in the 3D case. In 3D, the instruction `addVelocity-Boundary0N` instantiates a plane boundary domain in negative x-direction (a left boundary). It takes 6 arguments, additionally to the `omega`-argument to delimit the plane like a sub-volume with one degenerate space direction. The instruction `addExternalVelocityEdge0NP` instantiates an edge whose outward-pointing normal vector is in the `0`-plane (in the plane in which $x = 0$) and which points in negative y- and positive z-direction. Counting of indexes is cyclic: the instruction `addExternalVelocityEdge1NP` denotes an edge with normal vector in the $y = 0$-plane and with negative z- and positive x-direction. The `Edge` instructions also take 6+1 arguments, because they treat the edge like a sub-volume with two degenerate directions. In 3D, there are external and internal corners, and there are external and internal edges.

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

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

createLocalBoundaryConditionXD
This is the default local boundary condition. It implements a regularized boundary [3], which tends to be numerically stable in a last range of regimes.

createInterpBoundaryConditionXD
This is the default non-local boundary condition. It is based on the algorithm proposed by Skordos [16], and uses a finite difference scheme over adjacent neighbors to evaluate velocity gradients.

createZouHeBoundaryConditionXD
The local boundary condition introduced by Zou and He [17]. It is very accurate, especially in 2D simulations, but can have stability issues.

createInamuroBoundaryConditionXD
The local boundary condition by Inamuro *et al.* [18]. It is very accurate in 2D and 3D, but can have stability issues. In 3D, it is slower than other boundary conditions, because it solves an implicit equation at every time step.

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

It should be clear by now how powerful the abstraction mechanism of the "OnLatticeBoundaryConditionXD" objects is. With their help, 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. It bottom line is that both local and non-local boundary conditions instantiate a special dynamics object and assign it to boundary cells. Non-local boundaries additionally instantiate post-processing objects which take care of non-local aspects of the algorithm.

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

Listing 10: Implementation of a bounce-back cylinder in the domain center

```
<...> definition of the types T and DESCRIPTOR
int nx, ny, r;
<...> initialization of nx and ny, r
BlockLattice2D<T,DESCRIPTOR> lattice(nx,ny);
<...> setup of the lattice
for (int iX=0; iX<nx; ++iX) {
    for (int iY=0; iY<ny; ++iY) {and the dynamic too afterwards
        if ((iX-nx/2)*(iX-nx/2) + (iY-ny/2)*(iY-ny/2) < r*r) {
            lattice.defineDynamics(iX,iX,iY,iY,
                        &instances::getBounceBack<T,D2Q9Descriptor>() );
        }
    }
}
```

3.4 Lesson 4: Conversion between lattice and physical units

Fluid flow problems are usually given in a system of metric units. For example consider a cylinder of diameter $3cm$ in a fluid channel with average inflow velocity of $4m/s$. The fluid has a kinematic viscosity of $0,001m^2/s$. We are interested in 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 unity. Therefore when setting up a simulation a conversion directive has to be defined, which takes care of translating variables from physical units into lattice units and *vice versa*. In OpenLB all these conversions are handled by a class called LBconverter. An instance of the LBconverter is generated with some reference values of the simulation and the desired discretization parameters. 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 characteristical quantities of the fluid flow problem. In our example it is suitable to choose the cylinder's diameter as characteristic length and the average inflow speed as characteristic velocity. The converter internally builds a "dimensionless" system of units in which the characteristic values are one. The Reynolds number Re is an important parameter of this system. Furthermore two discretization parameters *latticeL* and *latticeU* are commited to the converter. *latticeL* is the discrete space intervall in physical units and from this the dimensionless discretization parameter δ_x is determined: $\delta_x = latticeL/charL$. *latticeU* sets the relation between the discretization parameters for space δ_x and time δ_t in dimensionless units: $latticeU = \delta_u = \delta_t/\delta_x$. Instead of δ_t, LB people often like to specify *latticeU*. One reason for this is that *latticeU* is proportional to the Mach number, and its choice is important to control compressibility effects.

Once the converter is initialized, its methods cand the dynamic too afterwardsan be used to convert various quantities such as velocity, 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:

Listing 11: Use of LBconverter in a 3D problem

```
<...> definition of type T
```

```
2    int dimension    = 3;
3    T latticeL       = (T) 0.003;
4    T latticeU       = (T) 0.02;
5    T charNu         = (T) 0.001;
6    T charL          = (T) 0.03;
7    T charU          = (T) 4.;
8    T charRho        = (T) 1.;
9    T pressureLevel = (T) 0.;
10
11   Lbconverter<T> converter(
12       dimension, latticeL, latticeU,
13       charNu, charL, charU, charRho, pressureLevel
14   );
15   writeLogFile(converter, "converterLog.dat");
16   cout << converter.getRe() << endl;
17   T omega = converter.getOmega();
18 <...> simulation
19 <...> evaluation of latticeRho at the back and the front of the cylinder
20   T pressureDrop =    converter.physPressure(latticeRhoFront)
21                     - converter.physPressure(latticeRhoBack);
```

Line 2: Specify discretization parameters and characteristical values.

Line 11: Instantiate a `Lbconverter` object.

Line 15: Write simulation parameters and conversion factors in a logfile.

Line 16: Print the Reynolds number `Re`.

Line 17: The method `getOmega` computes first the viscosity in lattice units, and then the relaxation parameter ω.

Line 20: The converter automatically calculates the pressure values from the local density.

3.5 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. Instead, a `BlockLatticeXD` sums them up internally, and in this way keeps track of the average density, the average energy (half the square of the velocity norm) and the maximum value of the velocity norm. Those values are accessed trough 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.

One needs to be careful though to properly interpret the value of the discrete time to which those quantities correspond. Imagine your simulation is at a discrete time t. After execution of a collision and a streaming step, it is taken from time t to time $t+1$. If after this you evaluate for example the velocity at a point through the command `lattice.get(iX,iY).computeU(velocity)`, the computed quantity lives at a time $t+1$ of the system. The values of the internal statistics, such as `lattice.getStatistics().getAverageEnergy()` correspond however to the discrete time t, because they were evaluated prior to the previous streaming step. This time shift between the state of the system and the value of the internal statistics can be confusing, and for this reason it would have made sense to avoid computing the statistics. On the other hand, keeping track of the statistics takes a neglibibly small amount of time. This feature is therefore included in OpenLB out of efficiency considerations, and out of convenience, as it offers an easy means of monitoring the well behaving of a simulation.

Lattice cells whose dynamics is bounce-back, generated by
`instances::getBounceBack<T,LATTICE>()`,
and cells that don't execute any collision step, generated by
`instances::getNoDynamics<T,LATTICE>()`
don't contribute to the internal statistics of the lattice. The same holds for subdomains for which, by using the approach taught in Lesson 9, no memory is allocated.

Often, the information provided by the statistics of a lattice in not sufficient, and you would like to treat the numerical result more generally. To do this, you can extract data cell-by-cell from the `BlockLatticeXD` and store it into a scalar- or vector/tensor-valued matrix, named `ScalarFieldXD` in the first case and `TensorFieldXD` in the second. During parallel program execution, those matrices are parallelized, which makes it very efficient to analyze large data sets on a parallel machine. The data can then be further analyzed, for example by computing reductions such as the average value. Alternatively, its content can be stored to disk in a binary VTK format for analysis with an external tool. Extraction of numerical data from a `BlockLatticeXD` into a `ScalarFieldXD` / `VectorFieldXD` is taken care of by the `DataAnalysisXD` class.

The most straightforward way of visualizing the data is to produce a 2D snapshot of a scalar field. OpenLB creates images of format `PPM`. On a system of the Unix/Linux family with the package `ImageMagick` installed, it further supports automatic conversion into the more common `GIF` format (note that `ImageMagick` is open sourced, and that it is part of all major Linux distributions). The following example illustrates how a snapshot of the vorticity distribution in a `2D` simulation is created:

Listing 12: Produce a GIF image from 2D data

```
1 // <...> Create and initialize a variable lattice
2 //        of type BlockLattice2D<T,D2Q9Descriptor>
3   DataAnalysisBase2D<T,D2Q9Descriptor> const& analysis
4       = lattice.getDataAnalysis();
5   ImageWriter<T> imageWriter("earth");
6   imageWriter.writeScaledGif("vorticity", analysis.getVorticity,
7                              200, 200);
```

Line 3: Require an analysis object from the lattice. Alternatively, an instance of the class `DataAnalysisXD` could be prepared manually. The advantage of requiring it from the lattice is that among different implementations of the class DataAnalysisXD the most efficient one is automatically picked out for you, distinguishing for example between sequential and parallel lattices.

Line 5: Prepare for creation of an image with the colormap ”earth”.

Line 6: Calculate vorticity on every cell, and visualize it as a GIF image. The colormap is rescaled to fit the range of vorticity values. The dimension of the image is rescaled to fit into a 200×200 bounding box.

Producing 2D images is also useful in 3D simulations. In this case you can extract data on a plane orthogonal to one of the coordinate axes and produce an image from it. This is done through the slice methods of data fields:

Listing 13: Produce a GIF image from 3D data

```
// <...> Create and initialize a variable lattice
//       of type BlockLattice3D<T,D3Q19Descriptor>
  DataAnalysisBase3D<T,D3Q19Descriptor> const& analysis
      = lattice.getDataAnalysis();
  ImageWriter<T> imageWriter("earth");
// Extract a slice of the plane defined by z=0
  int slicePos=0;
  imageWriter.writeScaledGif (
      "vorticity", analysis.getVorticity.sliceZ(slicePos), 200, 200 );
```

Although the computation of statistics and the production of 2D images are very useful, they are not always sufficient to extract all the required information from the simulation. When a detailed analysis is required, it makes sense to resort to an external tool that performs postprocessing of numerical data. For this, the data can be stored in a file in a VTK format. The function writeVTKData3D stores a scalar field and a vector field in the same VTK file:

Listing 14: Produce a VTK file from 3D data

```
// <...> Create and initialize a variable lattice
//       of type BlockLattice3D<T,D3Q19Descriptor>
  DataAnalysisBase3D<T,D3Q19Descriptor> const& analysis
      = lattice.getDataAnalysis();
  writeVTKData3D( "lesson5",
                  "vorticity", analysis.getVorticityNorm(),
                  "velocity", analysis.getVelocity(), 1., 1. );
```

The open source software Paraview [19] for example is very useful for the visualization of 3D data contained in such a file.

3.6 Lesson 6: Use an external force

In simulations, the dynamics of a fluid is often driven by a force field (gravity, inter-molecular 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 minimze cache-misses, the value of this force can be stored in a cell, adjacent to the particle populations. This is achieved by specifying external scalars in the lattice descriptor (see also

Lesson 7). OpenLB offers by default the two descriptors `ForcedD2Q9Descriptor` and `Forced-D3Q19Descriptor`. The dynamics `ForcedBGKdynamics` accesses the force term defined by these descriptors, and implements a LB dynamics with body force. The algorithm is taken from Ref. [20] to guarantee second-order accuracy even when the force field is space and time dependent. An example for the implementation of a LB simulation with force term is found in the code `forced-Poiseuille`.

3.7 Lesson 7: Understand in what sense OpenLB is generic

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 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 `MultiBlockLatticeXD` for parallel applications and for memory-saving applications in face of 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. It 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 is implemented through a mechanism of dynamic (run-time) genericity. In this way, the dynamics can be different from one cell to another, and it 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 `addPost-ProcessorXD`. 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.

3.8 Lesson 8: Use checkpointing in long-lasting 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 to 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. The basic commands for saving and loading data are `saveData` and `loadData`. They take as first argument the object to be serialized resp. unserialized, and as second argument the filename:

Listing 15: Store and load the state of the simulation

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

Checkpointing is also illustrated in the example programs `bstep2D` and `bstep3D` (Section 10.2).

3.9 Lesson 9: Save memory when domain boundaries are irregular

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

3.10 Lesson 10: Run your programs on a parallel machine

OpenLB programs can be executed on a parallel machine with distributed memory, based on MPI. The approach taught in this lesson uses `MultiBlockLatticeXD`, which inherits the interface of `BlockStructureXD`, and therefore behaves like a common, non-parallelized lattice. All techniques described in the previous lessons can be used with the `MultiBlockLatticeXD` as well, and thus work both in sequential and parallel programs. The only modification you are required to do, is to swith between `BlockLatticeXD` and `MultiBlockLatticeXD`. This can be achieved through a precompiler directive, as in the following code:

Listing 16: MultiBlockLattice2D for MPI-parallel programs

```
int nx, ny;
<...> initialization of nx and ny
#ifndef PARALLEL_MODE_MPI  // sequential program execution
    BlockLattice2D<T, DESCRIPTOR> lattice(converter.getNx(),
                                          converter.getNy() );
#else                      // parallel program execution
    MultiBlockLattice2D<T, DESCRIPTOR> lattice (
        createRegularDataDistribution( converter.getNx(),
                                       converter.getNy() ) );
#endif
```

In a shared-memory environment, OpenMP is an alternative to MPI for parallelism. To parallelize OpenLB with OpenMP, no code needs to be changed at all. Just modify a flag in the Makefile as described below.

To obtain parallel versions of the example programs, modify the flags `CXX` and `PARALLEL_MODE` in the file `Makefile.inc` in the OpenLB root directory. Then, enter the directory of the desired example, eliminate previously compiled libraries (`make clean; make cleanbuild`), and recompile the example by typing the command `make`.

4 Compilation

4.1 Linux

Let us start with a clean Ubuntu 12.04 LTS system. Before installing any new software, run

```
sudo apt-get update
```

to update the package lists, so that the most recent versions of the packages will be installed.

Then install the g++ compiler which you will need to compile c++ programs:

```
sudo apt-get install g++
```

To benefit from the efficient parallelization you will probably want to run the program on more than one core, so it is recommended to install Open-MPI:

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 1: Material numbers for a 2d channel flow similar to the example `cylinder2d` from Section 10.4 (1=fluid, 2=no-slip boundary, 3=velocity boundary, 4=constant pressure boundary, 5=curved boundary, 0=do nothing).

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

For visualization purposes you can use, for example, the following open source software:

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

```
sudo apt-get install imagemagick
```

Paraview is an application built on top of the *Visualization Tool Kit* (VTK) libraries which can read vti-files writen by OpenLB. With *imagemagick*, OpenLB can directly produce gif-files during simulation.

Finally, go into the root folder of OpenLB and type

```
make
```

to compile the software library and all examples. If your system is set up correctly, you should see a lot compiler messages but no errors.

4.2 Mac

4.3 Windows

5 Geometry

5.1 Material numbers

In OpenLB exists 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 1 illustrates this at the example of an external flow. The reward of using material numbers in flow simulations is to determine the fluid directions on boundary nodes automatically, since this is not always practical by hand e. g. if material numbers of a complex geometry are obtained from a `stl` file.

5.2 Indicator Functions

OpenLB provides several functors (see Section 8) for creation of basic geometric entities such as cuboids, circles, spheres, cones etc. All indicator functors inherit from `IndicatorFXD` and therefore contain the following functions:

- `std::vector<T> operator()(std::vector<S> in)`: Takes a `X`-dimensional vector `in` in SI coordinates and returns either `true` if the point is inside the geometry, `false` otherwise.
- `std::vector<S>& getMin()`: Returns the lower corner of an axis aligned bounding box.
- `std::vector<S>& getMax()`: Returns the upper corner of an axis aligned bounding box.
- `bool distance(S& distance,const std::vector<S>& origin, const std::vector<S>& direction, int iC = -1)`: Stores the `distance` of `origin` to the closest boundary in `direction` and returns `true` if found.

The geometries already implemented are:

- 2 Dimensions:
  - `IndicatorCuboid2D`
  - `IndicatorCircle2D`
- 3 Dimensions:
  - `IndicatorCuboid3D`
  - `IndicatorCircle3D`
  - `IndicatorSphere3D`
  - `IndicatorLayer3D`
  - `IndicatorCylinder3D`
  - `IndicatorCone3D`
  - `IndicatorParallelepiped3D`
  - `IndicatorIdentity3D`

These can be combined using the mathematical operators (+ union, − set difference, · intersection) to create more complex domains. Furthermore the class `STLreader` (7.4) inherites from `IndicatorF3D` and can be used in the same manner. A demonstration of this can be found in the example venturi3D (see Section 10.10).

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 voxel in the direction of the discrete normal are fluid
     voxel with material fluidM in the region where the indicator function
      is fulfilled
  void rename(int fromBcMat, int toBcMat, int fluidMat, IndicatorF3D<bool,
     T>& condition);
```

5.3 Creating a Geometry

With the information in the last section, a computational domain is created in 6 simple steps (see also Fig 2):

1. Step: Create indicator by
    - (a) Reading a stl-file.
    - (b) Pre-defined geometric primitives, cf. Section 5.2.
    - (c) Combinations of indicators $(+, -, \cdot)$.
    - (d) Other operators on indicators (e.g. extra layer for boundary).
2. Step: Construct cuboidGeometry. During construction cuboids will be automatically removed, shrinked and weighted for a good load balance.
3. Step: Construct loadBalancer.
4. Step: Construct superGeometry.
5. Step: Set *material numbers.*
6. Step: Construct superLattice.

```
/// 1. Step: Create indicator
  STLreader<T> stlreader("filename.stl", voxelSize);
  Cone3D<bool, T> cone(center1, center2, radius1, radius2);
  Layer3D<bool, T> boundaryLayer(cone, voxelSize);

/// 2. Step: Construct cuboidGeometry.
  CuboidGeometry3D<T> cuboidGeometry(indicator, voxelSize, noOfCuboids);

/// 3. Step: Construct loadBalancer.
  HeuristicLoadBalancer<T> loadBalancer(cuboidGeometry);

/// 4. Step: Construct superGeometry.
  SuperGeometry3D<T> superGeometry(cuboidGeometry, loadBalancer);

```

```
/// 5. Step: Set material numbers.
/// set material number 2 for whole geometry
  superGeometry.rename(0,2,geometryIndicator);
/// change material number from 2 to 1 for inner (fluid) cells, so that
    only boundary cells have material nunmer 2
  superGeometry.rename(2,1,1,1,1); or
  superGeometry.rename(2,1,fluidIndicator);
/// additional material numbers for other boundary conditions
  superGeometry.rename(2,3,1,inflowIndicator);
  superGeometry.rename(2,4,1,outflowIndicator0);
  superGeometry.rename(2,5,1,outflowIndicator1);

/// 6. Step: Construct superLattice.
  SuperLattice<T, DESCRIPTOR> sLattice(superGeometry);
```

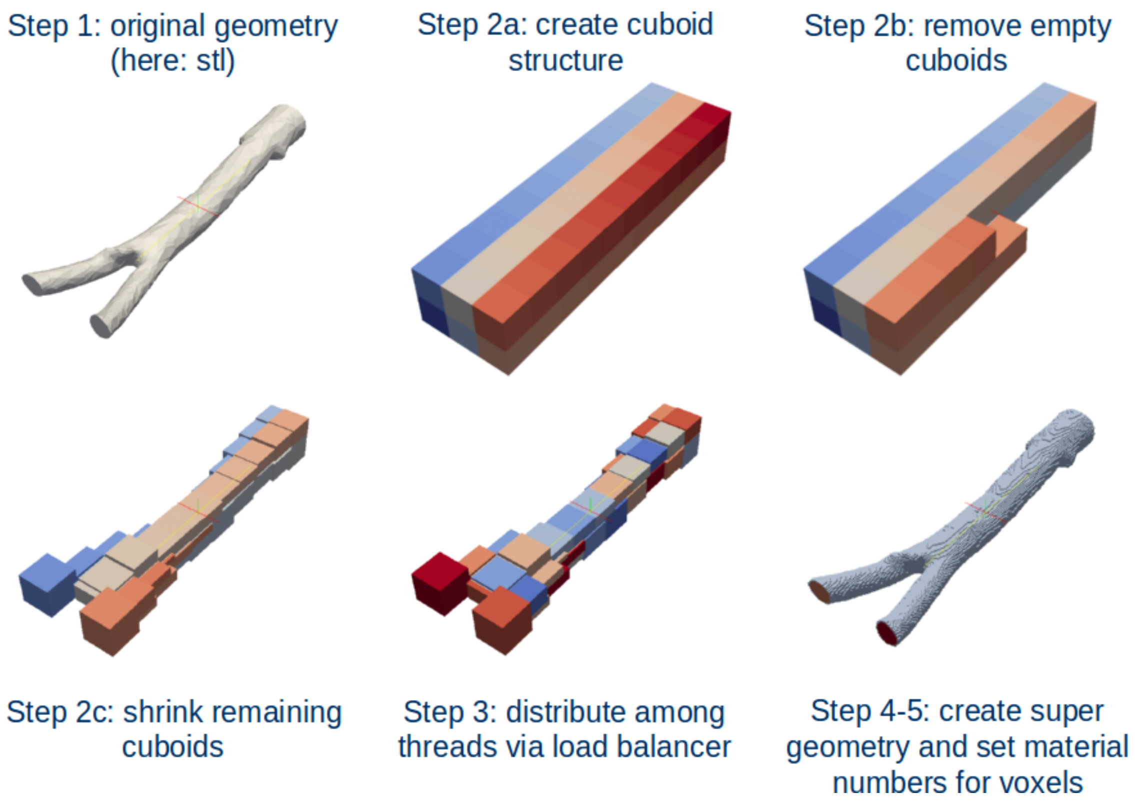

Step 1: original geometry (here: stl)
Step 2a: create cuboid structure
Step 2b: remove empty cuboids
Step 2c: shrink remaining cuboids
Step 3: distribute among threads via load balancer
Step 4-5: create super geometry and set material numbers for voxels

Figure 2: 6 steps to create a Geometry.

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 [21]. An introduction to modelling with FreeCAD can be found for example in `http://www.youtube.com/watch?v=6RxHCR7TLtI`. The general procedure is mostly similar to the following description.

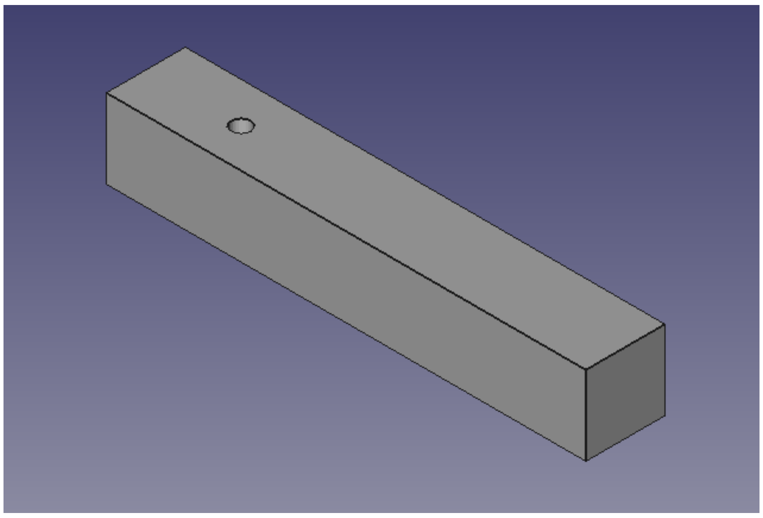


Figure 3: The geometry for the example `cylinder3d` from Section 10.4 opened in FreeCAD.

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 3 is not very difficult, the more complex geometry of a formula one car however can be a challenging and time consuming task! and the dynamic too afterwards.

6 Lattice Boltzmann Models

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 executions times on the one hand and to a sparse memory consumption on the other hand. Furthermore, it encourages a particularly efficient form of *data parallelism*, in which an array is cut into regular pieces and distributed over the nodes of a parallel machine. As a result, LB applications can be run even on large parallel machines with a particularly satisfying gain of speed.

The same spirit is adopted in the OpenLB package, in which the basic datastructure is a

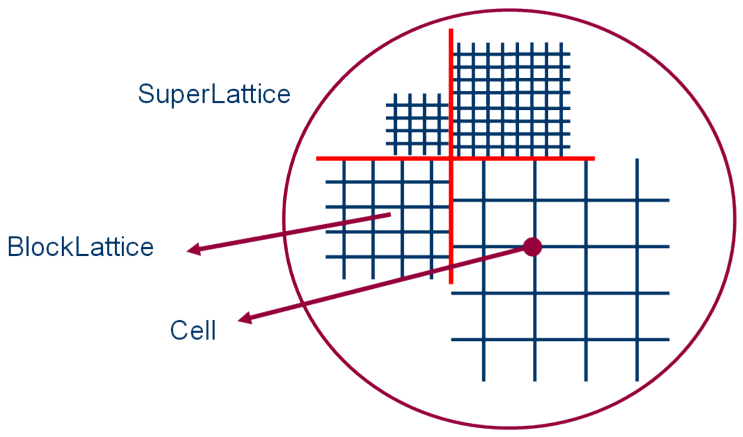


Figure 4: Data structure in OpenLB: A number of `BlockLattices` build a `SuperLattice` to adopt higher level software constructs like multi-block, grid refined lattices and parallelised lattices

`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. This data structure is encapsulated by 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 `SuperLattices`.

6.1.2 Descriptor

6.1.3 Dynamics

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

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

and the *streaming step (propagation step)*

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

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

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

6.2 Classic BGK Model

6.3 MRT Model

6.4 Porous Media Model

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

$$K = -\frac{Q\mu L}{\Delta P}, \tag{3}$$

and $Q = UA$ 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 $d = 1$, it is liquid. According to Brinkman [22, 23], Borrvall und Petersson [24] und Pingen et al. [25], the belonging Navier-Stokes equation is transformed (see Dornieden [26] and Stasius [27]). The discrete formulation of d describes a flow region by its permeability:

$$d = 1 - h^{dim-1}\frac{\nu_{LB}\tau_{LB}}{K} \tag{4}$$

τ_{LB} is the relaxation time, ν_{LB} is the discrete kinematic viscosity and h is the length. Therefore is $K \in [\nu_{LB}\tau_{LB}h^{dim-1}, \infty]$. To describe the porosity or permeability of a medium, you have to use a descriptor for porosity like

```
  #define DESCRIPTOR PorousD3Q19Descriptor
```

Be aware that the porous media model does only work in the generic compilation mode. In 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.defineExternalField(superGeometry, 3,
      DESCRIPTOR<T>::ExternalField::porosityIsAt, 1, porosity);
    ...
  }
```

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

```
  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); }
```

6.5 External Force

6.6 Multiphysics Couplings

6.7 Advection Diffusion Equation

The Advection-Diffusion-Equation is used to describe the transport of particles, energy or temperature due to advection and diffusion:

$$\frac{\partial c}{\partial t} = \nabla \cdot (D \nabla c) - \nabla \cdot (\vec{v} c) \tag{5}$$

Here c is the considered physical quantity, 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 [28]

$$g_i^{eq} = w_i \rho \left(1 + \frac{c_i \cdot \vec{v}}{c_s^2} \right), \tag{6}$$

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:

Listing 17: advection diffusion dynamics object

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

also a different descriptor with less lattice velocities is used [29]:

Listing 18: advection diffusion descriptor

```
 #define DESCRIPTOR AdvectionDiffusionD3Q7Descriptor
```

In OpenLB `D2Q5` and `D3Q7` descriptors for the Advection-Diffusion-Equation are implemented. Since this 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 to simulate e.g. a boundary with a constant temperature. To apply this condition first a `sOnLatticeBoundaryCondition3D` object for the advection diffusion boundarys has to be constructed:

Listing 19: advection diffusion dynamics object

```
sOnLatticeBoundaryCondition3D<T, DESCRIPTOR>
                                       sBoundaryConditionAD(sLattice);
int nC = sBoundaryConditionAD.get_sLattice().getLoadBalancer().size();
sBoundaryConditionAD.set_overlap(1);
for (int iC = 0; iC < nC; iC++) {
  OnLatticeAdvectionDiffusionBoundaryCondition3D<T,DESCRIPTOR>*
   ADblockBC =
    createAdvectionDiffusionBoundaryCondition3D<T,DESCRIPTOR,
    AdvectionDiffusionBGKdynamics<T, DESCRIPTOR> >
     (sBoundaryConditionAD.get_sLattice().getExtendedBlockLattice(iC));
  sBoundaryConditionAD.get_CDblockBCs().push_back(ADblockBC);
}
```

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

Listing 20: advection diffusion descriptor

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

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

Listing 21: add advective velocity on a cell

```
T velocity[3] = {vx,vy,vz};
...
cell.defineExternalField (
  DESCRIPTOR<T>::ExternalField::velocityBeginsAt,
  DESCRIPTOR<T>::ExternalField::sizeOfVelocity,
  velocity);
```

or on the whole `SuperLattice` by

Listing 22: add advective velocity on a superlattice

```
ConstAnalyticalF3D<T,T> velocity(vel);
...
/// sets advective velocity for material 1
superLattice.defineExternalField(superGeometry, 1,
  DESCRIPTOR<T>::ExternalField::velocityBeginsAt,
  DESCRIPTOR<T>::ExternalField::sizeOfVelocity,
  velocity)
```

with `vel` being an `std::vector<T>`.

7 Input / Output

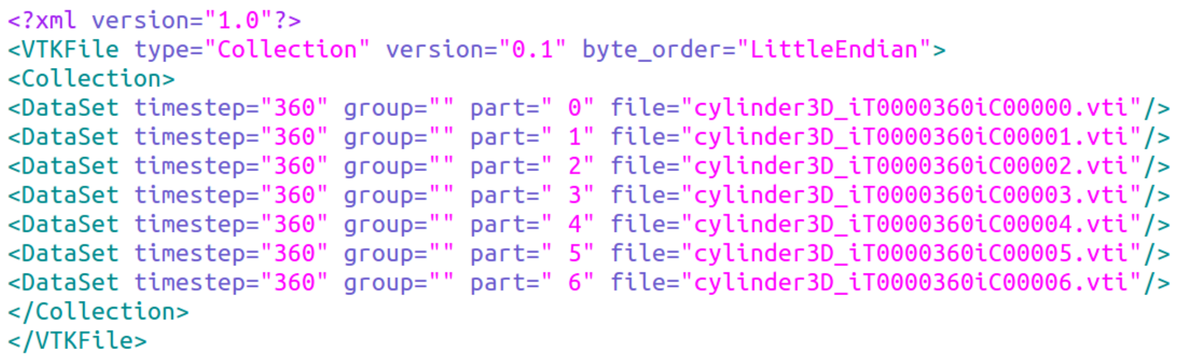
During development or even during actual simulation, it might be necessary to parametrize your program. For this case, OpenLB provides an XML parser, which can read files produced by OpenGPI [30], thereby even providing a nice GUI if you are so inclined. Details on the XML format and functions are given in Section 7.5.

The simulation data is stored in the VTK data format and can be used in addition for post-processing via 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 those 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 this output method. Therefore, OpenLB has implemented a parallel data output functionality. Every thread writes the data of its cuboids in one vti file. One ends up with several vti files, that contains partial simulation data. The key is to use an appropriate pvd file, which links the produced vti files and introduce a meaningful connection. The technical aspects are presented in Section 7.1, whereas the usage is demonstrated with an example in Section 7.2.

7.1 The output data structure in parallel simulations

OpenLB simulation data is stored in the VTK data format [31]. This format has XML structure and its data can be written either in human readable `ASCII` or binary `Base64` code. The implemented parallel output structure contains a pvd file, that consists of links to the real data written by the threads and stored in vti files. An example of a pvd file is shown in Figure 5.

For certain time steps, every thread write its data to a vti file. Since they do their work independently of each other, the writting happens in parallel. Those data files are linked in the mentioned pvd file and the hierarchical structure of the simulation data is build.



```
<?xml version="1.0"?>
<VTKFile type="Collection" version="0.1" byte_order="LittleEndian">
<Collection>
<DataSet timestep="360" group="" part=" 0" file="cylinder3D_iT0000360iC00000.vti"/>
<DataSet timestep="360" group="" part=" 1" file="cylinder3D_iT0000360iC00001.vti"/>
<DataSet timestep="360" group="" part=" 2" file="cylinder3D_iT0000360iC00002.vti"/>
<DataSet timestep="360" group="" part=" 3" file="cylinder3D_iT0000360iC00003.vti"/>
<DataSet timestep="360" group="" part=" 4" file="cylinder3D_iT0000360iC00004.vti"/>
<DataSet timestep="360" group="" part=" 5" file="cylinder3D_iT0000360iC00005.vti"/>
<DataSet timestep="360" group="" part=" 6" file="cylinder3D_iT0000360iC00006.vti"/>
</Collection>
</VTKFile>
```

Figure 5: The pvd file consists of links to the simulation data, which is stored in vti files. Since every thread writes its data to a single vti file, this program was executed by 7 threads.

It is still possible to write vtk files sequentially. This method does not need the hierarchical overhead and is much easier to use. Commonly, data like geometry, rank and cuboid is written only once during the simulation. In order to prevent big data overhead and complicated structures, the sequential routine is preferred for those functors.

7.2 Data output to VTK file format

VTK data files can be visualized and postprocessed with the free software Paraview [19], which offers a nice graphical interface. The following listing shows on the one hand, how to write VTK files sequential for an geometry and cuboid functors. On the other hand, the usage of the parallel write routine for velocity and pressure functors is shown.

```
// binary data format is default
SuperVTKwriter3D<T,DESCRIPTOR> vtkWriter("FileNameGoesHere");
// ASCII data format is obtained by
// SuperVTKwriter3D<T,DESCRIPTOR> vtkWriter("FileNameGoesHere",false);
SuperLatticePhysVelocity3D<T, DESCRIPTOR> velocity(sLattice, converter);
SuperLatticePhysPressure3D<T, DESCRIPTOR> pressure(sLattice, converter);
vtkWriter.addFunctor( velocity );
vtkWriter.addFunctor( pressure );

if (iT==0) {
  SuperLatticeGeometry3D<T, DESCRIPTOR> geometry(sLattice, superGeometry);
  SuperLatticeCuboid3D<T, DESCRIPTOR> cuboid(sLattice);
  // writes the geometry and cuboid no. to a single vti file sequentially
  vtkWriter.write(geometry);
  vtkWriter.write(cuboid);
  // mandatory to call the following write()-method
  vtkWriter.createMasterFile();
 }

  if (iT%converter.numTimeSteps(.3)==0) {
  // writes the added functors in parallel to vtk files
```

```
    vtkWriter.write(iT);
}
```

Note that the function call `creatMasterFile()` is essential to write parallel vtk data.

7.3 Console output

In OpenLB exists an extension of default ostreams that handles parallel output and prefixes each line with the name of the class that produced the output. Here is the output of one of the example programs from Section 10:

```
$ ./cylinder3d
[main] Nx=252; Ny=43; Nz=43
[BlockGeometry3D] the model is correct!
[BlockGeometry3D] wrote vti-File
[BlockGeometryStatistics3D] materialNumber=0; count=1892
[BlockGeometryStatistics3D] materialNumber=1; count=416970
[main] step=0; t=0; avEnergy=0; avRho=1; uMax=0
[reIniGeometry] step=0; scalingFactor=3.37314e-12
[main] step=50; t=2.5; avEnergy=6.5764e-08; avRho=0.999936; uMax=0.00507172
[computeResults] deltaP=-2.0906e-10
```

It is easy to determine which part of OpenLB has produced a specific message. This can be very helpful as well in debugging process as in 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 easily possible to visualize every imaginable data like convergence rate, data errors, or simple average mass density in diagrams.

```
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 one the one hand of the two words class and output, on the other hand it is quite similar to standard `cout`. The constructor gets two arguments, one describing the ostream to use, the other one setting the prefix-text. In line 4 is shown the usage of an instance of the `OstreamManager`. 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 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 benefits of `OstreamManager` is the reduction of output in parallel. Running a program using `cout` on multiple cores means normally to get one output line for each process. OstreamManager will avoid this by default and display only the output of the first processor. If that behavior is not wanted in a specific case, it can be turned off for an instance named `clout` by `clout.setMultiOutput(true)`.

Further scenarios which 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 needs of course its own instance.

7.4 Read and write .stl-files

OpenLB enables the possibility to read and write geometry data in the Standard Triangulation Language, short: stl. 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 with an radius of $\text{rad} = 2^{j-1} \cdot \text{voxelSize}, j \in \mathbb{N}$ 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 are 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).

7.5 XML parameter files

As explained in the introduction, OpenGPI provides an API to access configuration data for your application. This might come in handy to avoid multiple recompilations while searching for optimal parameters or in general development. 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 filename to the class constructor of `XMLreader`.

```
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);
```

Let us examine the code above. 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, there are now multiple methods. One is to pass a predefined variable to 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`.

8 Functors – a general concept for input and output of data

Roughly spoken, 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 to create a hierarchy and bundle ”classes of functions”. Moreover, parameters that are constant over several function evaluations need to be passed only once during instantiation.

8.1 Functors in OpenLB

In OpenLB, functors are used for a wide variety of tasks. They are divided by the unit system they are working on, making excessive use of heritage, templates and other stuff that comes along with C++.

GenericF stands on 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.

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

IndicatorF is an other subclass of GenericF and it returns a vector with elements 0 or 1. They are used to construct geometries, e.g. IndicatorSphere3D creates via an origin and radius a sphere. Its 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.

Block/SuperLatticeF is just an other subclass of GenericF. Those functors are defined on the lattice and present commonly 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.

InterpolationF establish conversion between the analytical and lattice functors. They are very important to set analytical boundary conditions, via evaluating the given analytical function on the lattice points. The reverse direction - from lattice to analytical functors - is the name-giver, since the conversion is achieved by interpolation between the lattice points.

8.2 How to use these functors?

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

Data output / data extraction Velocity, pressure, cuboids and other informations can be extracted from the lattice using predefined functors. All they need to know is a SuperLattice and converter if dimensions are wanted.

Listing 23: Code example for calculating velocity and pressure using functors.

```
// Create the data-reading functors...
SuperLatticePhysVelocity3D<T, DESCRIPTOR> velocity(&sLattice, &converter);
SuperLatticePhysPressure3D<T, DESCRIPTOR> pressure(&sLattice, &converter);
// geometries are often constructed by simple geometries
IndicatorSphere3D<bool,T> mySphere(origin,1);
```

Interpolation Interpolation is necessary to smoothly start a simulation or to obtain velocities between the computed ones on the lattice points. For the start of a simulation, the inflow velocity is smoothly increased from 0 to the desired velocity using a variable called `frac`. It is clear that frac should be 0 at the beginning of the simulation and 1 after a certain number of time steps `iTmaxStart`.

Listing 24: Code example for smoothly starting the inflow velocity in cylinder3d with a x^5 curve.

```
PolynomialStartScale<T,int> nPolynomialStartScale(iTmaxStart, T(1));
std::vector<int> iTvec(1,iT);
T frac = nPolynomialStartScale(iTvec)[0];
```

An other case for interpolation functors is the conversion of a given analytical functor such as a analytical solution, to a SuperLattice functor. Afterwards, the difference can be calculated easily with the help of the functor arithmetic. Finally, specific norms implemented as functors facilitates to talk about convergence. This application is shown in the example poiseuille2d, which is discussed in 10.8

Setting of boundary values Boundary cells are marked by a certain material number in SuperGeometry. Using a functor, velocities can be set at once on all cells of this material. First a vector is necessary that characterizes the maximum flow velocity and its directions. 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 SuperGeometry. In the last step, SuperLattice initializes all cells of a certain material given by SuperLattice with the velocities computed by the functor.

Listing 25: Code example for setting a Poiseuille velocity profile and a constant pressure boundary in cylinder3d.

```
// Creates and sets the Poiseuille inflow profile using functors
std::vector<T> maxVelocity(3,0);
maxVelocity[0] = 2.25*frac*converter.getLatticeU();
SquarePoiseuilleInflow3D<T> poiseuilleU(superGeometry, 3, maxVelocity);
sLattice.defineU(superGeometry, 3, poiseuilleU);
```

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 the area of the surface A, directed as the surface normal $\vec{n}$.

The *flux functor* calculates the discrete flux

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

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

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

So for the SuperLatticeFlux functor we need to define a surface, here a plane, and an SuperLatticeF functor.

The *plane* can be define by a circle indicator, or a starting point and a normal, or a starting point and two vectors. Optional one can set a *radius* for the plane. The *grid length* of the area can be defined, default is the lattice length. Also optional is a *material list*, so only the points with the defined material numbers are used for calculation, the default material number is 1. Next is a *SuperLatticeF functor*, which defines the quantity one wants to measure.

Step 1: define plane by
a) *a circle indicator*

```
IndicatorCircle3D<T,T> circleInd(center1, center2, center3,
                                 normal1, normal2, normal3, radius);
```

b) *a normal, a starting point and (optional) a radius*

```
std::vector<T> startingPoint, planeNormal;
T radius;
```

c) *two vectors, a starting point and (optional) a radius*

```
std::vector<T> startingPoint, planeVectorU, planeVectorV;
T radius;
```

Step 2 (optional): define grid length of the plane

```
T h = converter.getLatticeL();
```

Step 3 (optional): define material list

```
std::list<int> materials;
```

Step 4: create SuperLatticeF functor
a) *for velocity flow*

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

b) *for pressure*

```
SuperLatticePhysPressure3D<T, DESCRIPTOR> press(sLattice, converter);
```

c) *or any other SuperLatticeF functor*

```
SuperLatticeF3D<T, DESCRIPTOR> ...;
```

Step 5: create SuperLatticeFlux functor (depending on how the plane was defined)
a)*circle indicator*

```
SuperLatticeFlux3D(SuperLatticeF3D<T, DESCRIPTOR>& f,
        SuperGeometry3D<T>& sg, IndicatorCircle3D<bool,T>& circle,
        std::list<int> materials, T h = T());
```

b)*normal and startingPoint*

```
SuperLatticeFlux3D(SuperLatticeF3D<T, DESCRIPTOR>& f,
        SuperGeometry3D<T>& sg, std::vector<T>& n, std::vector<T> A,
        std::list<int> materials, T radius = T(), T h = T());
```

c)*two vectors and startingPoint*

```
SuperLatticeFlux3D(SuperLatticeF3D<T, DESCRIPTOR>& f,
        SuperGeometry3D<T>& sg, std::vector<T>& u, std::vector<T>& v,
        std::vector<T> A, std::list<int> materials, T radius = T(),
        T h = T());
```

Besides the arguments for the plane, the constructor takes 2 **necessary** and 3 **optional** arguments.

- *f:* the functor defined in **Step 4**
- *sg:* the SuperGeometry3D object
- *materials:* **default** is material number 1
- *radius:* **default** is the diameter of the geometry
- *h:* **default** is the lattice length

Step 6: get results by using the operator()

```
std::vector<int> input;
flux(input)[x];
```

- *output[0]*: flow rate, or force (if quantity has dimension 1)
- *output[1]*: size of the area
- *output[2..4]*: flow vector (ie. vector of summed up quantity)

Because in general the SuperLattice functor is either the velocity functor or the pressure functor, **Step 4** and **Step 5** can be combined. The constructors, depending on how the plane is defined, are identical to the ones used for SuperLatticeFlux, only the SuperLatticeF3D<T, DESCRIPTOR> argument is replaced by the two arguments SuperLattice3D<T, DESCRIPTOR> and LBconverter<T>.

Step 4.1): combined steps for velocity flow

```
SuperLatticePhysVelocityFlux3D<T, DESCRIPTOR>
    vFlux(SuperLattice3D<T, DESCRIPTOR> sLattice, LBconverter<T> converter,
            ...);
```

Step 4.2): combined steps for pressure

```
SuperLatticePhysPressureFlux3D<T, DESCRIPTOR>
    pFlux(SuperLattice3D<T, DESCRIPTOR> sLattice, LBconverter<T> converter,
            ...);
```

For these two functors exists a *print()* function.

Step 5.1): output for velocity functor (region size[m^2], volumetric flow rate and mean velocity)

```
vFlux.print(std::string fluxSiScale, std::string meanSiScale);
```

- *fluxSiScale:* 'ml/s' or 'l/s' or ' ' (default=m^3/s)
- *meanSiScale:* 'mm/s' or ' ' (default=m/s)

Step 5.2): output for pressure functor (region size[m^2], force and pressure)

```
pFlux.print(std::string fluxSiScale, std::string meanSiScale);
```

- *fluxSiScale:* 'MN' or 'kN' or ' ' (default=N)
- *meanSiScale:* 'mmHg' or ' ' (default=Pa)

Next are two code examples for the implementation of the flux functor in cylinder3d.

Example 1: *circle indicator, material list and SuperLatticeFlux3D*

Listing 26: Code example for getting the volumetric flow rate of the velocity flow in cylinder3d.

```
std::list<int> materials;
materials.push_back(1);
materials.push_back(6);

IndicatorCircle3D<bool,T> circleInd(2., 0.205, 0.205, 1., 1., 0., 2.);
SuperLatticePhysVelocity3D<T, DESCRIPTOR> vel(sLattice, converter);
SuperLatticeFlux3D<T, DESCRIPTOR> flux(vel, superGeometry, circleInd);

clout << "flowRate=" << flux(input)[0];
clout << "regionSize=" << flux(input)[1] << endl;
```

Example 2: *normal, startingPoint and SuperLatticePhysPressureFlux3D*

Listing 27: Code example for getting the pressure on a area in cylinder3d.

```
std::vector<T> A(3,T()), n(3,T());
A[0]=2.;A[1]=0.205;A[2]=0.205;
n[0]=1.;n[1]=1.;n[2]=0.;

SuperLatticePhysPressureFlux3D<T, DESCRIPTOR> pFlux(sLattice, converter,
        superGeometry, n, A);
pFlux.print();
```

9 Parallel program execution

Whenever possible, an OpenLB application should be written in such a way that it works well on both serial and a parallel platforms. Indeed, as applications in computational fluid dynamics require a large amount of resources, it is important to be able to switch to a parallel platform in a flexible way. This Section concentrates on parallelism on distributed memory machines using MPI, as distributed memory is the most common model on large-scale parallel machines. Furthermore, MPI parallelism has become an important option even on simple desktop computers, which quite often possess multi-core processors. In that case, you will often find that MPI is actually more efficient and/or easier to obtain in a non-commercial compiler setting than OpenMP. It is fortunately straightforward to write parallelizable application 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 which have the look and fell of serial applications, OpenLB distinguishes two classes of data. Data which is spatially distributed, such as the lattice, or scalar- respectively 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, this type of structures are referred to as data-parallel strucures. Other data types which require a small amount of storage space are duplicated on every node, and they are referred to in the following 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 7.3).

9.1 Data-parallel structures

Obtaining data-parallelism in OpenLB is as easy as using the `MultiBlockLatticeXD` instead of a `BlockLatticeXD`, a `MultiScalarFieldXD` instead of a `ScalarFieldXD`, and a `MultiVectorFieldXD` instead of a `VectorFieldXD`. In most common situations, only the case of the `BlockLatticeXD` actually needs to be treated explicitly, and this point is handled in a single line in the code, as it is for example shown in Lesson 10 (Section 3.10). Scalar- and vector-valued fields are usually generated automatically, as in the following expression:

```
// This yields an object of type ScalarFieldXD in serial,
//   and an object of type MultiScalarFieldXD in parallel
  lattice.getDataAnalysis().getVelocity();
```

The difference between the serial and the parallel case is handled transparently by addressing the data fields through through the virtual base `ScalarFieldBaseXD` respectively `VectorFieldBaseXD`, which is the same for the serial and the parallel data type:

```
// The following instruction works for in serial as well as
//   in parallel, because ScalarFieldBase2D is an abstract
//   base to both ScalarField2D and MultiScalarField2D
  ScalarFieldBase2D<T,Lattice> const& velocity
    = lattice.getDataAnalysis().getVelocity();
```

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 **??**, where it is shown how to use predefined functions for I/O operations on data-parallel structures, instead of explicit space loops.

9.2 Duplicated data types

The rule for duplicated data types is simple: all data types except for the data-parallel ones mentioned in the previous Section are duplicated. The three following rules need to be respected to ensure that the value from some input is properly duplicated over processors:

1. The call to `olbInit` at the beginning of a program ensures distribution of input from the command-line.
2. The use of `cin` ensures distribution of input from the terminal.
3. The use of `olb_ifstream` instead of `fstream` ensures distribution of input from a data file.

10 The example programs

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

10.1 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 fully automatically. As dynamics a Smagorinsky turbulent BGK model is used to stabilize the simulation for low resolutions. As output the flux at the inflow and outflow region is computed. The results has been validated by comparison with other results obtained with FEM and FVM.

10.2 bstep2d and bstep3d

The implementation of a backward facing step. It is furthermore shown how to use checkpointing to save the state of the simulation regularly.

10.3 cavity2d and cavity3d

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

10.4 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[32]. 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 [21].

10.5 multiComponent2d and multiComponent3d

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 [10]. These examples show the usage of multicomponent flow and periodic boundaries.

10.6 nozzle3d

This example examines a turbulent flow in a nozzle injection tube. At the main inlet, either a block profile or a power 1/7 profile is imposed as a Dirchlet velocity boundary condition, whereas at the outlet a Dirichlet pressure condition is set by p=0 (i.e. rho=1). The example shows the usage of turbulent models.

10.7 phaseSeparation2d and phaseSeparation3d

In these examples the simulation is initialized with a given density plus a small random number all 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.

10.8 poiseuille2d

This example examines a 2D Poseuille flow. Computation of error norms via functors is shown as well. `bgkPoiseuille2d` and `mrtPoiseuille2d` use a velocity or pressure boundary at the inlet/outlet. In `forcedPoiseuille2d` the boundaries are periodic between inlet and outlet. The flow is driven by a body force. It illustrates the use of a body force and periodic boundaries. Additionally to different flavors of BGK [2] and the regularized LB model [3], OpenLB offers implementations of entropic and multiple-relaxation-time (MRT) models. `mrtPoiseuille2d` illustrates the use of MRT. An example program for the entropic model is not yet available.

10.9 thermal2d and thermal3d

Rayleigh-Bénard convection rolls in 2D and 3D, simulated with the thermal LB model by Z. Guo e.a. [11], between a hot plate at the bottom and a cold plate at the top.

10.10 venturi3d

This example examines a steady flow in a venturi tube. At the main inlet, a Poiseuille profile is imposed as 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). The example shows the usage of the Indicator functors to build up a geometry and explains how to set boundary conditions automatically.

References

[1] The Cygwin project. `http://www.cygwin.com/`.

[2] S. Chen and G. D. Doolen. Lattice Boltzmann method for fluid flows. *Ann. Rev. Fluid Mech.*, 30:329–364, 1998.

[3] J. Latt and B. Chopard. Lattice boltzmann method with regularized non-equilibrium distribution functions. *Math. Comp. Sim.*, 72:165–168, 2006.

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

[5] D. Yu, R. Mei, L.-S. Luo, , and W. Shyy. Viscous flow computations with the method of lattice Boltzmann equation. *Prog. Aerosp. Science*, 39:329–367, 2003.

[6] Santosh Ansumali. *Minimal kinetic modeling of hydrodynamics.* PhD thesis, Swiss Federal Institute of Technology Zurich, 2004.

[7] LB model with adjustable speed of sound. Technical report. `http://www.lbmethod.org/openlb/techreports.html`.

[8] Bastien Chopard, Alexandre Dupuis, Alexandre Masselot, and Pascal Luthi. Cellular automata and lattice Boltzmann techniques: an approach to model and simulate complex systems. *Adv. Compl. Sys.*, 5:103–246, 2002.

[9] 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.

[10] Xiaowen Shan and Hudong Chen. Lattice Boltzmann model for simulating flows with multiple phases and components. *Phys. Rev. E*, 47:1815–1819, 1993.

[11] Zhaoli Guo, Baochang Shi, and Chuguang Zheng. A coupled lattice BGK model for the Boussinesq equations. *Int. J. Num. Meth. Fluids*, 39:325–342, 2002.

[12] Dominique d'Humières, M'hamed Bouzidi, and Pierre Lallemand. Thirteen-velocity three-dimensional lattice Boltzmann model. *Phys. Rev. E*, 63:066702, 2001.

[13] M'hamed Bouzidi, Mouaouia Firdaouss, and Pierre Lallemand. Momentum transfer of a Boltzmann-lattice fluid with boundaries. *Physics of Fluids*, 13(11):3452–3459, 2001.

[14] The Fifth Element. `http://en.wikipedia.org/wiki/The_Fifth_Element`.

[15] How to implement your DdQq dynamics with only q variables per node. Technical report. `http://www.lbmethod.org/openlb/techreports.html`.

[16] P. A. Skordos. Initial and boundary conditions for the lattice Boltzmann method. *Phys. Rev. E*, 48:4824–4842, 1993.

[17] Q. Zou and X. He. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Phys. Fluids*, 9:1592–1598, 1997.

[18] T. Inamuro, M. Yoshina, and F. Ogino. A non-slip boundary condition for lattice Boltzmann simulations. *Phys. Fluids*, 7:2928–2930, 1995.

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

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

[21] FreeCAD: An Open Source parametric 3D CAD modeler. `http://free-cad.sourceforge.net/`.

[22] H.C. Brinkman. On the permeability of media consisting of closely packed porous particles. *Applied Scientific Research*, 1(1):81–86, 1949.

[23] H.C. Brinkman. A calculation of the viscous force exerted by a flowing fluid on a dense swarm of particles. *Applied Scientific Research*, 1(1):27–34, 1949.

[24] Thomas Borrvall and Joakim Petersson. Topology optimization of fluids in stokes flow. *International Journal for Numerical Methods in Fluids*, 41(1):77–107, 2003.

[25] Georg Pingen, Anton Evgrafov, and Kurt Maute. Topology optimization of flow domains using the lattice boltzmann method. *Structural and Multidisciplinary Optimization*, 34(6):507–524, 2007.

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

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

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

[29] X-Y Lu H-B Huang and M C Sukop. Numerical study of lattice boltzmann methods for a convectiondiffusion equation coupled with navierstokes equations. *J. Phys. A: Math. Theor.*, 44(5), 2011.

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

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

[32] S. Turek and M. Schäfer. Benchmark computations of laminar flow around cylinder. In *Flow Simulation with High-Performance Computers II*, volume 52 of *Notes on Numerical Fluid Mechanics*, pages 547–566. Vieweg, January 1996.

GNU Free Documentation License

Version 1.2, November 2002
Copyright © 2000,2001,2002 Free Software Foundation, Inc.

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

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

Preamble

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

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

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

1. Applicability and definitions

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

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

A "**Secondary Section**" is a named appendix or a front-matter section of the 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

2. Verbatim Copying