OpenLB User Guide
Associated to Release 0.4 of the code

Jonas Latt

jonas.latt@gmail.com

Contents

1 OpenLB in ten minutes

1.1 What is OpenLB?

OpenLB is a common effort of people with various backgrounds in Computational Fluid Dynamics to set up a numerical framework for lattice Boltzmann simulations. The code is intended to be used both by application programmers who simply want to run a simulation with a given flow geometry, and by developers who implement their own particular dynamics. To please the first audience, the code offers a simple interface through which it is possible to set up a simulation with little effort. For the second audience, the implementation of the code is kept as simple as possible. That is, it makes use of basic concepts to implement the LB dynamics, and does not get lost in technical ramification. The idea behind this approach is that the code can be used as a framework for programmers to develop pieces of reusable code that can be readily shared 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 effect of this approach 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, the use of C++ templates leads to static genericity of the code. As a result, it is sufficient to write a single generic code that implements the various 3D lattice topologies, such as those of the D3Q15, D3Q19 and D3Q27 lattices.

1.2 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, 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.

1.3 What features are currently implemented?

Lattice Boltzmann models

BGK model for fluids	Section 4	Reference [1]
Regularized model for fluids	Section 4	Reference [2]
Multiple relaxation times	Section 4	References [3, 4]
Entropic Lattice Boltzmann	Section 4	Reference [5]
BGK with adjustable speed of sound	Section 4	References [6, 7]

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. [8])
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	

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 `MultiBlockLatticeXD`, 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.

1.4 Participants

The following persons are contributing source code to OpenLB:

Jonas Latt (Tufts Univ.): Development of the OpenLB core, integration and maintenance of added components.

Mathias Krause and Prof. Vincent Heuveline (Univ. Karlsruhe): OpenMP parallelization, Cuboid data structure for MPI parallelization, Makefile environment for compilation.

Orestis Malaspinas (EPFL): Alternative boundary conditions (Inamuro, Zou/He, Nonlinear FD), alternative LB models (Entropic LB, MRT).

Bernd Stahl (Univ. Geneva): 3D extension to MultiBlock structure for MPI parallelization, parallel version of (scalar or tensor-valued) data fields, VTK output of data.

2 OpenLB in ten lessons

2.1 Lesson 1: 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 1.2.

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

```
// CODE 1.1: Framework of an OpenLB program.
/*1*/ #include "olb2D.h"
      #ifndef OLB_PRECOMPILED // Unless precompiled version is used,
/*2*/   #include "olb2D.hh"   // include full template code
      #endif

/*3*/ using namespace olb;

      int main(int argc, char* argv[]) {
/*4*/     olbInit(&argc, &argv);
/*5*/     // Insert the central part of your code here
      }
```

Some lines in this program deserve additional comments:

`/*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`.

`/*2*/` 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 this overhead becomes too annoying during frequent development-compilation cycles, the code can be precompiled for the required data types. Although this topic is not covered in the tutorial, this short explanation should make clear what the cryptic `#ifndef OLB_PRECOMPILED` is about.

`/*3*/` All OpenLB code is contained in the namespace `std`.

`/*4*/` 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.

At this point, the code for the simulation of a fluid flow can be inserted at the place of item `/*5*/`. 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.)*

```
// CODE 1.2: to be inserted under Item /*5*/ of code 1.1.
/*5.1*/  #define LATTICE D2Q9Descriptor
/*5.2*/  typedef double T;
/*5.3*/  int nx = 20;
         int ny = 30;
         int numIter = 100;
         T omega = 1.;

/*5.4*/  BlockLattice2D<T, LATTICE> lattice(nx, ny);
/*5.5*/  BGKdynamics<T, LATTICE> bulkDynamics (
             omega,
             instances::getBulkMomenta<T,LATTICE>()
         );
/*5.6*/  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;
                 }
/*5.7*/          lattice.get(iX,iY).iniEquilibrium(rho,u);
             }
         }

         for (int iT=0; iT<numIter; ++iT) {
/*5.8*/      lattice.collide();
/*5.9*/      lattice.stream(true);
         }

/*5.10*/ ImageWriter<T> imageWriter("leeloo");
         imageWriter.writeScaledGif (
             "lesson1",
/*5.11*/     blockLattice.getDataAnalysis().computeVelocityNorm() );
```

A few explanations are again in order:

`/*5.1*/` 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.

`/*5.2*/` 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.

`/*5.3*/` Constants to specify the dimensions of the `nx`×`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.

`/*5.4*/` Instantiation of a `BlockLattice2D` object. At this point, memory for the `nx`×`ny`×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`.

`/*5.5*/` The `Dynamics` object determines the implementation of the collision step on grid nodes, in this case BGK [1]. 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.

`/*5.6*/` 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`.

`/*5.7*/` Initialize particle populations at an equilibrium distribution, with slightly increased density inside a circle of radius `r`.

`/*5.8*/` At each iteration step, the collision specified by the variable `bulkDynamics` is applied to each grid node.

`/*5.9*/` After collision follows the streaming step. The boolean argument `true` indicates that boundaries are periodic.

`/*5.10*/` 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. [9]).

`/*5.11*/` 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()` with an argument `false`. This is the default argument, and the method can therefore be invoked with no argument at all;

```
// CODE 1.3: Substitutions in CODE 1.2 to replace periodic
//   boundaries by no-slip walls
/*5.8*/      lattice.collide();
/*5.9*/      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 [10].

2.2 Lesson 2: Understand the `BlockLattice`

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

The truth is that the overhead incurred by delegating collision to an object (instead of hard-coding collision somewhere inside the loop over grid nodes) is completely irrelevant. The efficiency loss is minimal on all platforms on which OpenLB 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 in items `/*5.8*/` and `/*5.9*/` of Lesson 1. The question to ask, instead of nitpicking over object-oriented vs. non-object-oriented issues, is whether it is really necessary to step through memory twice, once to execute collision and once to execute streaming. As a matter of fact, there are several ways of avoiding this time-consuming double access to memory, one of which is implemented in OpenLB and documented in Ref. [10]. For an OpenLB user, doing this is as easy as replacing the collision-streaming sequence by a call to the method `collideAndStream()`:

```
// CODE 2.1: Substitutions in CODE 1.2 for more efficient
//   collision-streaming cycles
// /*5.8*/      lattice.collide();
// /*5.9*/      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):

```
// CODE 2.2: Direct access of 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 `CODE 2.2`:

```
// CODE 2.3: Manipulation of data through methods of a Cell
int nx, ny, someX, someY, someF;
// <...> 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 `CODE 2.3` has a distinct advantage to the approach of `CODE 2.2`: in `CODE 2.2` the data inside a `Cell<T>` is accessed directly, whereas in `CODE 2.3` 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 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.

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:

```
// CODE 2.4: 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.

2.3 Lesson 3: Define and use boundary conditions

The current OpenLB release offers five different boundary conditions for the implementation of pressure and velocity boundaries. They support boundaries that are aligned with the numerical grid, and also implement properly 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:

```
// CODE 3.1: 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.

```
// CODE 3.2: 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 `addVelocityBoundary1P` refers to a boundary whose normal is in positive y-direction (`P` stands for "positive",

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

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

Things are slightly more complicated in 3D, where edges also need 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 `addVelocityBoundary0N` 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 [2], 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 [11], and uses a finite difference scheme over adjacent neighbors to evaluate velocity gradients.

createZouHeBoundaryConditionXD
The local boundary condition introduced by Zou and He [12]. It is very accurate, especially in 2D simulations, but can have stability issues.

createInamuroBoundaryConditionXD
The local boundary condition by Inamuro *et al.* [13]. It is very accurate in 2D and 3D, but can have stability issues. In 3D, it is slower than other boundary conditions, because it solves an implicit equation at every time step.

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

It should be clear by now how powerful the abstraction mechanism of the "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:

```
// CODE 3.3: 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) {
        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>() );
        }
    }
}
```

2.4 Lesson 4: Convert between lattice and physical units

You would like to simulate a 2D flow around a quadratic obstacle of size $L \times L$, where L is some given length. The obstacle is located in the middle of a domain of height $3L$ and length $5L$. All boundaries implement Dirichlet velocity conditions: no-slip on the upper and lower boundary and a parabolic Poiseuille profile on the left and right boundary. The Reynolds number of this flow is defined with respect to the maximum velocity U in the middle of inlet and outlet, the height L of the obstacle and the viscosity ν of the fluid: $Re = U \cdot L/\nu$.

This section is about the choice of units one has to make when setting up the simulation and interpreting the numerical results. The variables used in a LB simulation (for example, the velocity one gets by computing the order-one moment of the particle populations) 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. To evaluate the data, other systems of units are however more convenient. In practice, one wants for example to play with the numerical parameters such as the discretization interval for space and time: asymptotically, the numerical result should be independent of these parameters. To be independent of lattice parameters, it is common to choose a system in which the reference length (here L) and the reference time are unity. In the present case, this implies that the reference velocity U is unity. The variables in this system of units are often called "dimensionless variables", because they are dimensionless with respect to, say, metric units of the "real" physics. In the context of LB, this is however confusing, because it is lattice units that are often considered dimensionless. In this text, the lattice-independent units are therefore called "physical units" and carry the label PU: they represent a fictitious physical world, as opposed to lattice units, labelled by LU, which represent the numerical world. To compare the simulation with a physical experiment these fictitious physical units must be further converted into metric (or othe) units. This is a trivial exercise that is up to the reader.

In physical units, the simulation introduced in this lesson has a size 3×5. Let us say that we want to represent the obstacle by $N \times N$ lattice sites. The integer number N is the discretization parameter, and its inverse $\delta_x = 1/N$ is the discrete space interval. With the help of this interval, physical units are recovered from lattice units: $x_{PU} = x_{LU} \cdot \delta_x$. In the same way, the discrete time interval δ_t recovers the physical time from the lattice time (the number of iteration steps):

$t_{PU} = t_{LU} \cdot \delta_t$. Remembering that the reference velocity is unity in physical units, this leads to the following relation between U_{LU} and δ_t: $\delta_t = U_{LU} \cdot \delta_x$.

The definition of a fluid simulation is typically split into three steps: (1) definition of the problem in physical units, (2) choice of the simulation parameters δ_x and δ_t, and (3) initialization of the variables of the simulation. Instead of δ_x and δ_t, LB people often like to specify N and U_{LU}. One reason for this is that U_{LU} is proportional to the Mach number, and its choice is important to control compressibility effects. It is however clear from the last paragraph that these two choices of lattice parameters are equivalent.

To end this discussion, an important relationship between the type of boundary condition and the number of allocated lattice sites is pointed out. Imagine that your numerical grid has a total of 5 nodes in y-direction, and that the first and the fifth node implement the dynamics of a given boundary condition. If, physically speaking, boundaries are located on top of boundary nodes (as it is the case for the default local and non-local boundary conditions in OpenLB), then the simulation contains 4 cells in y-direction. If the boundary is half-way between a boundary node and its next neighbor, the the simulation contains only 3 cells in y-direction. This fact translates into the following definition of the lattice resolution N in OpenLB. If the y-length of the system is given in physical units by $l_{y,PU}$, and on-lattice boundaries are used, then a total of $N * l_{y,PU} + 1$ cells need to be allocated in y-direction. In case of boundaries that are half-way inbetween two nodes, a total of $N * l_{y,PU} + 2$ cells need to be allocated in y-direction.

For convenience, all these conversions are handled in OpenLB by a class `LBunits`. The following code snippet shows how to configure the simulation of the 2D problem introduced at the beginning of the lesson:

```
// CODE 4.1: Use of LBunits for the setup of a 2D problem.
<...> definition of the types T and DESCRIPTOR
/*1*/   T Re       = (T)10.;
        T lx       = (T)5.;
        T ly       = (T)3.;
        int N      = 50;
        T uLattice = (T)0.02;

/*2*/   LBunits<T> units(uLattice, Re, N, lx, ly);
/*3*/   BlockLattice2D<T,DESCRIPTOR> lattice(units.getNx(), units.getNy());
        BGKdynamics<T,DESCRIPTOR> bulkDynamics (
/*4*/       units.getOmega(),
            instances::getBulkMomenta<T,DESCRIPTOR>()
        );
```

`/*1*/` Specify the Reynolds number `Re` the dimension `lx`×`ly` of the system in physical units, the resolution `N` and the reference velocity `uLattice` in lattice units.

`/*2*/` Instantiate a `LBunits` object. The same class can be used for 3D simulation, by calling the constructor with an additional parameter `lz`.

`/*3*/` The methods `getNx` and `getNy` (and `getNz` in 3D) yield the required number of grid nodes for the simulation. They accept a boolean parameter, which defaults to `false` to indicate

that the boundary condition is on-lattice. For boundaries that are half-way between nodes, use `getNx(true)` and `getNy(true)`.

`/*4*/` The method `getOmega` computes first the viscosity in lattice units, and from this the relaxation parameter ω.

2.5 Lesson 5: Extract data from a simulation

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

```
// CODE 5.1: Produce a GIF image from 2D data
// <...> Create and initialize a variable lattice
//       of type BlockLattice2D<T,D2Q9Descriptor>
DataAnalysisBase2D<T,D2Q9Descriptor> const& analysis
    = lattice.getDataAnalysis();
// Prepare for creation of an image with the colormap "earth"
ImageWriter<T> imageWriter("earth");
imageWriter.writeScaledGif("vorticity", analysis.getVorticity, 200, 200);
```

`/*1*/` 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.

`/*2*/` Prepare for creation of an image with the colormap "earth".

`/*3*/` 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:

```
// CODE 5.2: 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:

```
// CODE 5.3: 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` [14] for example is very useful for the visualization of 3D data contained in such a file.

2.6 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 `ForcedD3Q19Descriptor`. The dynamics `ForcedBGKdynamics` accesses the force term defined by these descriptors, and implements a LB dynamics with body force. The algorithm is taken from Ref. [15] 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 `forcedPoiseuille`.

2.7 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 `addPostProcessorXD`. This terminology can be somewhat confusing, because the term "post-processing" is used in the CFD community in the context of data analysis at the end of a simulation. In OpenLB, a post-processor is an operator which is applied to the lattice after each streaming step. Thus, the time-evolution of an OpenLB lattice consists of three steps: (1) local collision, (2) nearest-neighbor streaming, and (3) non-local postprocessing. Implementing the dynamics of a cell through a postprocessor is usually less efficient than when the mechanism of the `Dynamics` classes is used. It is therefore important to respect the spirit of the lattice Boltzmann method and to express the collision as a local operation whenever possible.

2.8 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:

```
// CODE 8.1: 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` (Section 8.3) and `bstep3D` (Section 8.4).

2.9 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.

2.10 Lesson 10: Run your programs on a parallel machine

Two independent approaches are offered in OpenLB to run a program on a parallel machine with distributed memory, using MPI. Both constructs can also be used to achieve memory savings, as described in the previous lesson. One construct is the `MultiBlockLatticeXD`, which inherits the interface of `BlockStructureXD`, and therefore behaves like a common, non-parallelized lattice. The other approach is the `CuboidStructureXD`, which is not constrained to the interface of a `BlockStructureXD`. One consequence of this is that you need to modify some of your code between the sequential and the parallel version. But on the other hand, this parallel construct is more general and could be used when the matrix form of the `BlockStructureXD` does not represent the geometry of the problem in a natural way.

All techniques described in the previous lessons can be used in parallel as well as in sequential program executions. Indeed, all the example programs described in Section 8 work in serial and

parallel, using the `MultiBlockLatticeXD`. The only required modification between the sequential and the parallel case is the definition of the lattice:

```
// CODE 10.1: Store and load the state of the simulation
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
```

Furthermore, the example programs `cavity2d_cuboid_mpi` (Section 8.8) and `cavity3d_cuboid_mpi` (Section 8.9) illustrate the use of the `CuboidStructureXD` to achieve parallelism.

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`.

3 BlockStructures and Cells

4 Dynamics and Postprocessing

5 Examination / evaluation of numerical data

6 Parallel program execution

7 Input / Output

8 The example programs

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

8.1 poiseuille2d

This implementation of a 2D Poiseuille flow demonstrates the setup of a basic simulation in OpenLB. The demo offers the choice of local and non-local boundaries, and the choice of implementing the inlet/outlet with a velocity or pressure boundary condition. It further shows how to extract statistics on the state of the simulation (for example the average energy), and how to produce a snapshot of the velocity field in terms of a `GIF` image.

8.2 cylinder2d

This example is only slightly more complicated than the previous one. A cylindrical obstacle, implemented via bounce-back nodes, is placed in the flow and leads to von Karman instability.

8.3 bstep2d

In the implementation of a backward facing step, the definition of the domain geometry is slightly more complicated than previously: additionally to edges and exterior corners, the boundary contains an interior node. This example shows how to define boundaries in such a case. It is furthermore shown how to use checkpointing and save the state of the simulation regularly.

8.4 bstep3d

This is the 3D version of the backward facing step. It illustrates the use of various types of boundary nodes in 3D, including surfaces, edges, and corners. The output of the simulation includes 2D images of the velocity-norm along slices, 3D VTK data that can be visualized for example with Paraview [14], and binary data that stores the state of the simulation in regular time intervals.

8.5 cavity3d

This 3D example illustrates a 3D flow in a rectangular, lid-driven cavity.

8.6 forcedPoiseuille2d

In this implementation of a Poiseuille flow, the boundaries are periodic between inlet and outlet. The flow is driven by a body force instead of a pressure gradient. This simple example illustrates the use of a body force in OpenLB.

8.7 mrt2d

Additionally to different flavors of BGK [1] and the regularized LB model [2], OpenLB offers implementations of entropic and multiple-relaxation-time (MRT) models. The present example illustrates the use of MRT. An example program for the entropic model is presently not available.

8.8 cavity2d_cuboid_mpi

The 2D lid-driven cavity, parallelized with the `CuboidStructure2D` instead of `MultiBlockStructure2D`.

8.9 cavity3d_cuboid_mpi

The 3D lid-driven cavity, parallelized with the `CuboidStructure3D` instead of `MultiBlockStructure3D`.

8.10 cavity3d_omp

The 3D lid-driven cavity, parallelized with OpenMP for shared-memory machines.

References

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

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

[3] 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.

[4] 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.

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

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

[7] 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.

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

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

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

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

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

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

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

[15] 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.

GNU Free Documentation License

Version 1.2, November 2002

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

Preamble

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

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

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

1. Applicability and definitions

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

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

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as

Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

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

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

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

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

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

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

2. Verbatim Copying

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

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

3. Copying in quantity

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

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

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

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

4. Modifications

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

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

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

D. Preserve all the copyright notices of the Document.

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

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

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

H. Include an unaltered copy of this License.

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

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

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

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

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

O. Preserve any Warranty Disclaimers.

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

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

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through

arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

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

5. Combining documents

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

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

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

6. Collections of documents

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

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

7. Aggregation with independent works

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

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

8. Translation

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

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

9. Termination

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

10. Future revisions of this license

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

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