LBHydra™ user manual

Version 0.1 (Beta)

Stuart D. C. Walsh

December 19, 2010

Contents

1 Introduction 1
- 1.1 Chapter outline . . . 2
- 1.2 Installation . . . 2
  - 1.2.1 Basic installation . . . 2
  - 1.2.2 Local installation . . . 4
  - 1.2.3 Installation without C++ Libraries . . . 4
  - 1.2.4 OpenMP . . . 4
  - 1.2.5 LBHydraGPU . . . 5
- 1.3 Quick start guide . . . 5

2 Lattice-Boltzmann Methods 7
- 2.1 Overview . . . 7
- 2.2 Lattice-Boltzmann Algorithm . . . 8
  - 2.2.1 Initial conditions . . . 9
  - 2.2.2 Boundary conditions . . . 9
  - 2.2.3 Multiple component models . . . 10

3 LBHydra Simulations 11
- 3.1 Overview . . . 11
- 3.2 Preprocessor . . . 12
- 3.3 LBHydra Simulation Engine . . . 13
- 3.4 Postprocessor . . . 14

4 Tutorials 15
- 4.1 Poiseuille flow tutorial: `example_poiseuilleFlow` . . . 16
  - 4.1.1 Preprocessor walk-through . . . 16
  - 4.1.2 Postprocessor . . . 19

4.2 Two dimensional tutorial: `example_2D` . . . 19
4.2.1 Preprocessor . . . 20
4.2.2 Postprocessor . . . 21
4.3 Dynamic library tutorial: `example_dynalibs` . . . 21
4.4 Immiscible model tutorial: `example_immiscible` . . . 21
4.4.1 Preprocessor . . . 21
4.4.2 Postprocessor . . . 23

5 Pre- and Post-processing Toolboxes 25
5.1 LBHPythonToolbox . . . 25
5.2 LBHMatlabToolbox . . . 26
5.3 Toolbox Functions . . . 26

6 Code structure 31
6.1 LBHydra Libraries . . . 31
6.2 Employing LBHydra in other C++ programs . . . 32
6.3 Key classes and functions . . . 33

7 Extending LBHydra with Dynamic Libraries 37
7.1 Dynamic libraries for Material Models . . . 37
7.1.1 Dynamic material model example . . . 39
7.2 Dynamic Lattice Libraries . . . 42
7.3 Dynamic Lattice-Function Libraries . . . 43
7.4 Accessing the dynamic libraries . . . 45

8 LBHydraGPU 47
8.1 Overview . . . 47
8.2 Required hardware and software . . . 48
8.3 LBHgpupy . . . 48

A File Formats 51
A.1 Input files . . . 51
A.1.1 Input configuration file: XXX.simulationInput . . . 51
A.1.2 Node-Material model map file: XXX.nodeMap . . . 52
A.1.3 Neighbor map: XXX.neighborMap . . . 52
A.1.4 LBData . . . 52
A.1.5 Material model data files: XXX.modelData . . . 53
A.2 Output files . . . 53
A.2.1 LBData . . . 53
A.2.2 Brick of Bytes: XXX.bob . . . 54

A.2.3 Visualization Tool Kit: XXX.vtk 54

B Material Models 55

Bibliography 57

CHAPTER 1

Introduction

This manual is a concise and self-contained guide to LBHydra, a software package for conducting lattice-Boltzmann fluid-mechanics simulations. It is primarily targeted at an audience with a basic knowledge of lattice-Boltzmann models and some experience with Matlab, Python or similar scientific programming environments. The more advanced capabilities of the package outlined in the later chapters require an understanding of the C or C++ programming languages, however, this will not be necessary for most users.

The manual has two main goals. First, it outlines the basic capabilities of the LBHydra package, and provides the tools necessary to run new simulations (Chapter 1 to Chapter 5). Second, it describes how to incorporate LBHydra's functionality into other programs, and extend the capabilities of the basic package by introducing new types of lattice-Boltzmann simulations, models and lattices (Chapter 6 to Chapter 8).

The remainder of the chapter gives a brief summary of the contents of the manual, followed by instructions for installing LBHydra, and finishes with a step-by-step quick-start guide for running the example simulations included in the LBHydra package.

We hope you enjoy this manual.

1.1 Chapter outline

Chapter 2 covers the basic concepts of Lattice Boltzmann methods and a brief introduction of the terminologies used in the manual. Chapter 3 gives an overview of the stages involved in each LBHydra simulation. At the end of this chapter, the user will have a basic understanding of the key steps involved in a typical simulation.

Chapter 4 has a walk-through of the simulation tutorials included in the LBHydra package, showing how to generate the input files, run the simulation and visualize the result. Chapter 5 discusses the toolbox functions used in these tutorials which are included in the LBHydra package.

Chapter 6 introduces the functions and classes that make up the LBHydra code, and discusses how to include these functions in other C++ programs. For Linux and Unix users, LBHydra supports dynamic libraries that may be used to link new material models, lattice functions and even lattice-Boltzmann lattice types to the main simulation engine, without having to modify the code base directly or recompile the program. The use of dynamic libraries is described in Chapter 7.

The LBHydra package also includes LBHydraGPU, an extension to the main simulation engine that allows certain simulations to be accelerated using graphics processing units. The LBHydraGPU extension is presented in Chapter 8.

In the remainder of the manual, Appendix A details the simulation file conventions, while Appendix B briefly outlines the different lattice-Boltzmann methods included in the package.

1.2 Installation

There are several installation options for LBHydra. You may choose one of the following build types based on your needs and the type of hardware that you intend to run the code on.

1.2.1 Basic installation

To install LBHydra using the basic installation option, navigate to the LBHydra folder and execute:

```
make clean; make install
```

If the build is successful, copies of the executable `lbhydra` will be created in the LBHydra directory and `/usr/local/bin`. In addition, the basic installation creates the following directories and files:

1. Library and aliases for the LBHydra lattice Boltzmann and miscellaneous function libraries:

```
/usr/local/lib/libLBHlb.so
/usr/local/lib/libLBHlb.so.x
/usr/local/lib/libLBHlb.so.x.y
/usr/local/lib/libLBHmisc.so
/usr/local/lib/libLBHmisc.so.x
/usr/local/lib/libLBHmisc.so.x.y
```

N.B. on 64 bit machines these libraries will be installed in `/usr/local/lib64/`.

2. Directories containing the header files for the LBHydra libraries.

```
/usr/local/include/lbHydra/LatticeBoltzmann/
/usr/local/include/lbHydra/Miscellaneous/
```

3. Directories containing the Matlab and Python toolboxes.

```
/usr/local/lib/matlab/LBHMatlabToolbox
/usr/local/lib/python/LBHMatlabToolbox
```

The LBHydra makefile supports the GNU make variables: `prefix`, `bindir`, `includedir` and `libdir`. The variable `prefix` (set to `/usr/local` by default) controls the common directory in which the files are copied, while `bindir`, `includedir` and `libdir` store the destination directories for the executable, header files and libraries respectively. Two additional variables are also defined `PYTHONPATH` and `MATLABPATH`, which control the destination directories for the python and matlab toolboxes.

If necessary, the default installation paths can be overridden by setting the value of these variables when running the make command *e.g.*

```
make install MATLABPATH="~/matlab"
```

On 64 bit machines the `libdir` makefile variable can be used to save the LBHydra libraries to `/usr/local/lib` if desired.

Users who merely want a copy of LBHydra installed in their home directory, are advised to use the Local installation instruction option described below.

LD_LIBRARY_PATH

To employ the LBHydra libraries in external programs or user defined dynamic libraries, the library path `/usr/local/lib/` (or `/usr/local/lib64/` for 64 bit machines) must be added to the `LD_LIBRARY_PATH` environment variable. For bash shells add the following to the `.bash_profile` file in your home directory:

`export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH;`
or for tcsh shells use:
`setenv LD_LIBRARY_PATH /usr/local/lib:$LD_LIBRARY_PATH`

1.2.2 Local installation

The basic installation procedure will attempt to install LBHydra and its associated libraries under the `/usr/local` directory. However, this directory will likely be blocked to users without root access.

To install the program in the user's home directory instead, use the local installation shell script included with the LBHydra package. Type:
`./localInstall.sh`
at the command line from within the main LBHydra directory. The LBHydra libraries and include directories and python libraries will be installed in the user's home directory under `$(HOME)/local`, the matlab library will be installed in `$(HOME)/matlab`. To use the lbHydra C++ libraries in this case, the user will need to add `$(HOME)/local/lib` to the `LD_LIBRARY_PATH` environment variable and add `-L$(HOME)/local/lib` and
`-I$(HOME)/local/include/LBHydra` to the C++ compiler flags. For 64 bit systems the library paths end in `lib64`, e.g., `-L$(HOME)/local/lib64`.

1.2.3 Installation without C++ Libraries

As mentioned above, the LBHydra libraries are only required for certain advanced features of the lbHydra package, namely to include LBHydra functions in other applications and build new user defined models. To install the LBHydra simulation engine without the C++ libraries type:
`make clean; make install-nolibs`

1.2.4 OpenMP

LBHydra uses openMP (an open source parallel programming interface) to run the main simulation engine on multiple CPU cores. The code must be linked to the openMP libraries at compile time for the multiprocessing ability to be enabled. This option is enabled by default, as the openMP libraries are available with g++ versions 4.2 and above. However, if required, the openMP option may be disabled with the following makefile commands:
`make clean; make install USEOPENMP='false'`

By default LBHydra is built using the `g++` compiler, hence the `-fopenmp` flag is passed to the compiler when using openMP. However, other flags are required to use openMP with other compilers. To use openMP with an alternative compiler

(selected by changing the `CC` makefile variable) pass the correct openMP flag to the makefile by setting the `OPENMP_FLAGS` variable.

The default number of parallel OpenMP threads used by LBHydra (generally the number of cores the CPU has) can be changed at runtime using the `-o` command line option, *e.g.*

```
lbHydra -o 4 -f anExample.simulationInput
```

OpenMP is also required to run the GPU extension of LBHydra on multiple graphics processing units (see below). For these simulations, the `-o` command line option has no effect as one openMP thread is automatically created for each GPU.

1.2.5 LBHydraGPU

LBHydraGPU is an extension to the basic LBHydra simulation engine that accesses the processing power of the computer's graphics card to increase the speed of the lattice-Boltzmann simulations. These GPU simulations require a CPU with at least one CUDA compliant nVIDIA graphics card and the CUDA nvcc compiler installed.

The GPU-based simulations also differ from the other LBHydra simulations, in that they employ a dynamic library extension to the basic LBHydra executable. This library is not compiled at the same time as the LBHydra package, but is generated during the preprocessing stage instead. This process is discussed further in Chapter 8.

1.3 Quick start guide

This quick-start guide assumes that the LBHydra package is stored in the current working directory.

Several tutorial examples (further discussed in Chapter 4) have been included in the package to acquaint the user with LBHydra. The following steps describe how to run the basic Poiseuille flow tutorial using Matlab for pre and post processing. To run the other available tutorials, simply change the directory in Step 3.

1. Enter the LBHydra directory:
   ```
   cd LBHydra
   ```

2. Compile and install the LBHydra executable:
   ```
   make install
   ```

3. Go to the tutorial directory:
   `cd tutorials/example_poiseuilleFlow`

4. Start Matlab:
   `matlab`

5. Run the simulation (within Matlab):
   `runSim`

The final step runs the Matlab script `runSim.m` which consists of three separate commands:

1. a Preprocessor script (`Preprocessor_PoiseuilleExample.m`) that generates the input files required for the simulation,

2. a command that runs the simulation using the newly created files,

3. a Postprocessor script (`Postprocessor_PoiseuilleExample.m`) that visualizes the result.

The Preprocessor script creates a cuboid lattice of 20x5x80 nodes with solid boundaries at both ends of the x axis and a pressure gradient along the z axis. The lattice-Boltzmann simulation then executes 10,000 timesteps and when completed, the Postprocessor script compares the simulated fluid velocity to that of the analytical solution. Each stage of the calculation for this and the other tutorials are explained in greater detail by the comments in the pre- and post-processor files.

CHAPTER 2

Lattice-Boltzmann Methods

This chapter gives a brief introduction to lattice-Boltzmann methods. More detailed discussion on these types of simulations can be found in the texts by Wolf-Gladrow (2000), Rothman and Zaleski (1997) and Succi (2001), and in the excellent review articles by Chen and Doolen (1998) and Aidun and Clausen (2010).

2.1 Overview

Lattice Boltzmann simulations are capable of modeling a variety of complex fluid mechanical problems (for example, complex boundary conditions, miscible and immiscible fluids and thermal effects) that are difficult or impossible to handle using other modeling methods. This makes lattice-Boltzmann methods particularly attractive for a wide range of applications in both science and engineering.

Lattice-Boltzmann methods are a "bottom-up" numerical method for modeling fluid mechanics, developed from a related numerical method known as lattice gas cellular automata. Lattice gas cellular automata are discrete, lattice-based methods for modeling fluid behavior. The fluid is described by a collection of particles that move around the lattice at fixed velocities, whose positions are updated in discrete timesteps. Like the molecules in a real gas, the simulated particles in the lattice gas only change direction upon collision with another particle or the boundary of the vessel containing the gas. By choosing the correct lattice with appropriate laws governing the particle collisions, lattice

gases are capable of representing almost the same range of fluid behavior as lattice-Boltzmann methods, with the added benefit of being slightly easier to program. However, they suffer from a major drawback – noise. Although lattice gases reproduce the correct behavior over large scales, achieving these scales often requires prohibitively large particle numbers. Lattice Boltzmann methods circumvent this problem, by replacing the discrete collection of individual particles with a description of the probability density distribution function of the particles.

In lattice-Boltzmann methods, the fluid is modeled as a set of fluid packets that move about the lattice in the same manner as the particles in lattice-gas models. However, unlike lattice-gas models, each fluid packet has an associated density represented by a real number. In place of the billiard-ball collisions used in lattice-gas models, the density of the packets is instead determined by a discrete analogue to the classical Boltzmann equation. Through this process, the fluid packet density distribution is relaxed towards a local equilibrium distribution determined from the macroscopic properties of the fluid at the node. The following section gives a more detailed overview of the basic components of the lattice-Boltzmann method and how they are implemented in LBHydra.

2.2 Lattice-Boltzmann Algorithm

A lattice is a mesh of interconnected elements or nodes. In LBHydra, each node can have multiple components, referred to as phases. Each phase has an associated set of fluid packets.

The number of fluid packets in each phase (and the velocities of the fluid packets) is dictated by the type of lattice. Different lattice types are often distinguished by the DnQm notation, where Dn denotes the number of lattice dimensions and Qm the number of fluid packet velocities. Examples of lattice types used for lattice Boltzmann simulations include the D1Q3, D2Q7, D2Q9, D3Q15, D3Q19 and D3Q27 lattices. The D2Q9 lattice, for example, is a widely-used two dimensional, nine velocity lattice based on a regular Cartesian grid of nodes. The nine lattice velocities are a single stationary velocity (0,0), velocity vectors to the four nearest neighbors (1,0),(-1,0), (0,1), (0,-1), and the four next-to-nearest neighbors (1,1),(1,-1), (-1,-1), (1,-1).

At each timestep, the distribution of fluid packets in the lattice is updated through a two-step process: a collision step and a streaming step. The collision step typically involves only those fluid packets arriving at each node, although in some cases (for example multicomponent or multiphase simulations) fluid packets at other nodes (typically the node's neighbors) may also influence the collision step. In the streaming step, the calculated resultant fluid packets at each node

are propagated to the neighboring nodes as determined by the lattice velocities.

During the collision step, collision rules assigned to the nodes determine how the fluid packets are redistributed. The form of the collision rule depends on several different factors, including: the type of lattice used; the material properties of the fluid being modeled; and whether the node represents a point on the fluid boundary (and if so the type of boundary condition implemented at the node). Collision rules also differ in the method governing the relaxation towards equilibrium. Two broad classes of collision rules are single-relaxation collision rules and multiple-relaxation collision rules. The majority of the default collision rules supplied in the LBHydra simulation are based on single-relaxation lattice-Boltzmann methods, although some multiple relaxation models are also included and more will be added in the future. User-defined single and multiple relaxation collision rules may also be added using the methods discussed in Chapter 9. In general, multiple-relaxation models and single-relaxation models are incompatible, and care should be taken to avoid combining them in different nodes of the same phase.

In LBHydra, collision rules are determined by material models assigned to each phase of each node. The material model also dictates how to interpret macroscopic physical properties from the fluid packets, as these relationships may change depending on the nature of the simulation.

2.2.1 Initial conditions

The initial conditions for lattice-Boltzmann simulations are determined by the initial distribution of fluid packets. In LBHydra, this information is provided by user-supplied input files. Python and Matlab routines for determining these fluid packet distributions based on initial pressure and/or velocity fields are provided in the package, along with example preprocessor scripts. Appropriate initial conditions can also be obtained from the output of previous LBHydra simulations, and by user generated routines using the file format descriptions given in Appendix A.

2.2.2 Boundary conditions

Lattice-Boltzmann boundary models may be either link-based or node-based. In the first case, the boundary condition is enforced by adjusting the fluid packets moving along one edge or link of the lattice. In the second case, the boundary condition is implemented by adjusting all of the fluid packets at a lattice node.

Boundary conditions are supplied in one of two ways in LBHydra. The simplest method is to introduce boundary conditions by assigning boundary material-

models to appropriate nodes in the lattice (this method may also be used to implement link-based boundary methods). A detailed discussion of the types of material models available is given in Appendix B. For more complex simulations however, for example, where the position of the boundary changes over time or where the boundary condition depends on another external simulation, it may be simpler to implement boundary conditions using either pre- or post-collision lattice functions. As the name suggests, this feature of LBHydra allows the introduction of user-defined subroutines to query and modify the lattice prior to or following each collision step. Lattice functions are discussed further in Chapter 7.

2.2.3 Multiple component models

A strength of lattice-Boltzmann models is their ability to simulate complex fluids with multiple interacting components, *e.g.* a solvent with a dissolved solute or two immiscible liquids. Lattice-Boltzmann methods model these types of fluids by assigning separate fluid packet populations to each component.

In LBHydra, material models for multiple component simulations contain additional parameters to identify the interacting phases. In these cases, care should be taken that the phases are defined in a consistent manner, for example, that two immiscible phases correctly reference each other.

CHAPTER 3

LBHydra Simulations

3.1 Overview

Most LBHydra simulations follow the same execution pipeline - prepare input, run the simulation and analyze output. As depicted by Figure 3.1, the pipeline begins with a Preprocessor script, which prepares simulation input files for the main LBHydra simulator. The LBHydra simulator then executes a given number of timesteps, producing output data periodically. This data can either be returned to the lattice-Boltzmann simulator at a later point (in the event that the code is interrupted, one can resume simulations from the stage close to where it was halted) or passed on to the Postprocessor for analysis. More sophisticated modes of execution can be provided with user generated lattice functions (Chapter 7) or by employing the LBHydra libraries and objects within the user's own code (Chapter 6).

Example pre- and post-processor Matlab and Python scripts, and additional helper functions, are provided with LBHydra to aid in generating simulation input files and process output data (Chapter 4). In addition, the `LBHMatlabToolbox` file included in the LBHydra package contains a collection of Matlab functions for use in pre- and post-processor scripts, along with equivalent Python functions in the `LBHPythonToolbox` file (Chapter 5). However, the core simulation engine does not depend on Matlab or Python for execution. We are happy to help users to develop and distribute additional pre- and post-processing routines for other computing environments.

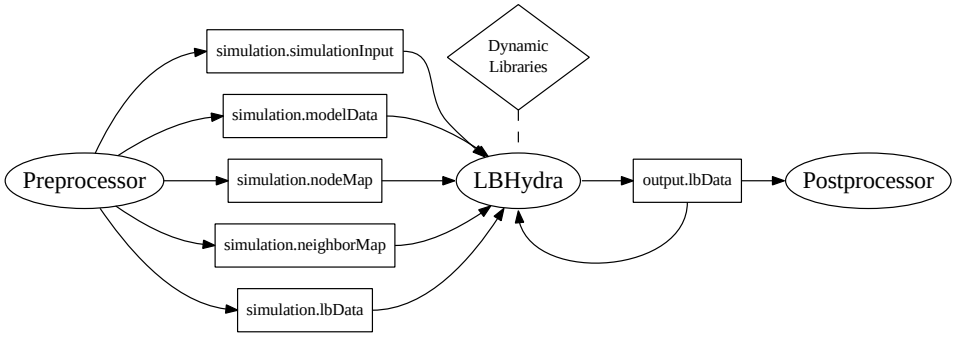


Figure 3.1: LBHydra simulation pipeline.

3.2 Preprocessor

Preprocessor scripts translate the simulation geometry, physical parameters and the initial conditions into the correct input format for the main program. This input takes the form of text (and in some cases binary) files that, once generated, can be modified directly using basic text editors. In some small scale simulations it might be feasible to create the required files directly. However, without a preprocessor script, the task of creating the input files rapidly grows in difficulty as the simulation complexity increases.

Although in general different simulations will have independent preprocessors, typically each preprocessor will complete a similar set of tasks. The general format of most preprocessors is:

1. Define simulation parameters (*e.g.* kinematic viscosity, pressure gradients etc.)

2. Generate a set of material models from those parameters (*e.g.* models controlling the fluid behavior and the boundary conditions).

3. Assign those newly created material models to the nodes and phases in the lattice. This is a color-by-numbers approach in which voxel arrays are created for each phase in the lattice (*i.e.* a three dimensional matrix of integers), and material models are assigned to the lattice by numbering the voxel array accordingly. For simple geometries the voxel array can be created algorithmically using array operations. Alternatively, more complex geometries (*e.g.* simulating flow through a porous matrix obtained from a tomography scan), the voxel array can be generated by reading in a file.

4. Assign unique IDs, coordinates and material models to each node and (in the event of a sparse matrix) generate lattice connectivity.

5. Generate initial conditions (again this can be supplied from the output of previous simulations) and assign fluid-packet density values to the nodes.

6. Write the generated data to files.

3.3 LBHydra Simulation Engine

LBHydra reads the files generated by the preprocessor and executes a predetermined number of timesteps, outputting the state of the lattice at regular intervals. The general protocol followed by LBHydra is:

- **Input simulation parameters**
  Each LBHydra simulation requires several input files. These files are supplied to the simulation engine by piping or passing (via the `-f` command line argument) a configuration file into the the program. The structure of this simulation configuration file is described in the File Specifications section in Appendix A.

- **Link dynamic libraries**
  Users have the ability to create their own material models, lattices and lattice functions, which are dynamically linked to the LBHydra simulation at runtime (Chapter 7). Paths to these dynamic libraries are specified using the `-d` command line option. The simulation searches in the directory specified by each `-d` switch and loads all the dynamic libraries found.

- **Set initial conditions**
  Next, the program sets the initial conditions for the lattice-Boltzmann simulation by loading the data from one or more files specified in the simulation input. Scripts for generating these data files are available in the LBHMatlabToolbox and LBHPythonToolbox directories.

- **Set material models**
  After the fluid packet densities have been initialized, the correct material models are set for each phase of each node. LBHydra allows users to customize their own material models, and additional material models will be added to newer versions of the code. Supported material models and their parameter strings can be tested with the `--materialModels` or `-m` command line option, *e.g.* `lbHydra -m`

- **Implement timesteps**
  Once the material models and initial conditions are loaded, the simulation is started. At each timestep the fluid packets in the lattice undergo a collision step and a streaming step. The model loops over a given number of timesteps, outputting the state of the lattice at predetermined intervals. The maximum number of timesteps and the output frequency are both specified in the input parameter file.

- **Output data**
  At predefined intervals in the simulation, data is output to a file (*e.g.* `out_001000.lbData`). The output data is a list of the different lattice Boltzmann particle density distributions for each node. Briefly, the advantages of this approach are that 1) no lattice Boltzmann data is lost allowing the simulation to be restarted from the same point [1], and 2) it provides a simulation independent means of handling the output. This last point is particularly important as the physical interpretation of the lattice-Boltzmann data may change depending on the nature of the simulation (*e.g.* in one simulation the sum of the fluid packet data may be proportional to the pressure, while in another it may represent a concentration of dissolved solute). Separating the post-processing script from the main simulation engine allows the output data to be interpretted correctly, and allows data analysis to be conducted while the simulation is running.

3.4 Postprocessor

A postprocessor script is used to analyse the output generated by the main LBHydra simulation engine. The postprocessor is a simulation-specific script whose functionality depends on the nature of the simulation and the desired analysis. Thus, unlike the preprocessor, there is no standard logic flow for the postprocessor.

Nevertheless, several postprocessing tasks are common to most LBHydra simulations, and toolbox functions for these tasks are provided with LBHydra. These include routines for importing data from LBHydra output files (in particular binary output files) and manipulating the fluid packet data once imported (*e.g.* calculating fluid densities and velocities).

[1] although other data – *e.g.* solid fractions of dissolved nodes may be. This additional information can be output using a lattice function (Chapter 7).

CHAPTER 4

Tutorials

Several tutorials are included with LBHydra to introduce the user to the application. The tutorials are located as separated directories beginning with the name "`example_`" in the `tutorials` directory.

Example	Description
`example_poiseuilleFlow`	Basic simulation setup.
`example_2D`	Use of different lattice-types (2D lattice).
`example_bodyForce`	Poiseuille flow with a body force term.
`example_dynalibs`	Use of dynamic libraries.
`example_immiscible`	Use of multiple compoment models.
`example_multirelaxation`	Use of multiple relaxation time models.
`example_porePermeability`	Flow through a porous matrix saved in a file.

The tutorial preprocessing and postprocessing scripts are provided in both Matlab and Python.[1] The tutorials are heavily documented and, for the most part, self-explanatory. To run any tutorial, follow these steps:

1. Distribute the LBHydra executable to the tutorials using the command, `make install` from the LBHydra directory (see Chapter 1 for more information).

[1]Both pre- and postprocessor Python scripts require the "numpy" (numerical python) libraries. In addition, the postprocessor scripts require the "pylab" libraries to visualise the results. The numpy library may be downloaded from `http://numpy.scipy.org`. The pylab library is part of matplotlib, which is available from `http://matplotlib.sourceforge.net`.

2. Navigate to the desired tutorial (any one of the example directories under `tutorials`).

3. Generate the tutorial's input files using a preprocessor.

   (a) Matlab: From a terminal, start Matlab with the command `matlab`. Inside the Matlab environment, type the name of the preprocessor script minus the `.m` suffix,
   `Preprocessor_XXX`

   (b) Python: Run the python preprocessor with the command
   `./Preprocessor_XXX.py`

4. Run the simulation by passing the simulation input file to the LBHydra executable with the `-f` command-line option.
   `./lbHydra -f XXX.simulationInput`

5. Interpret the output with a postprocessor:

   (a) Matlab: Inside the Matlab environment, type the name of the post-processor script minus the `.m` suffix, `Postprocessor_XXX`

   (b) Python: Run the python postprocessor with the command
   `./Postprocessor_XXX.py`

4.1 Poiseuille flow tutorial: `example_poiseuilleFlow`

The Poiseuille flow tutorial is included to demonstrate the key components of a typical simulation. It models a two dimensional flow between parallel plates. The post-processor compares the result of the simulation with the analytical solution. The pre and postprocessor examples described below are taken from the Matlab preprocessor, however, the python

4.1.1 Preprocessor walk-through

The first step of any Matlab preprocessor is to add the LBHydra Matlab toolbox to the Matlab search path. This is achieved with the following set of commands:

```
LBMatlab_Path = '../../LBHMatlabToolbox';
addpath(LBMatlab_Path);
```

Next, variables are introduced to hold the filenames for the simulation input files:

```
inputConfigurationFile = 'PoiseuilleExample.simulationInput';
nodeMapFile = 'PoiseuilleExample.nodeMap';
modelDataFile = 'PoiseuilleExample.modelData';
initialLBDataFile= 'PoiseuilleExample.lbData_bin';

outputFilePrefix = 'out';
```

... and set the simulation parameters:

```
% Lattice dimensions
nX = 20; % size of X-dimension
nY = 5; % size of Y-dimension
nZ = 80; % size of Z-dimension

% Physical parameters
deltaRho = 0.005;
kinematicViscosity = 0.16666667;

% Duration and output
maxTimeSteps = 10000;
outputFrequency = 1000;
```

The next step is to define the different material models used in the simulation. Here there are four different material models: a bounceback model used to simulate the no-flow boundary condition, a model for the fluid flow in the open channel, and density (pressure) controlled models for the boundaries on the top and bottom of the lattice

```
materialModels(1).name = 'd3q19BouncebackModel';

materialModels(2).name = 'd3q19Model';
materialModels(2).params.kinematic_viscosity = kinematicViscosity;

materialModels(3).name = 'd3q19DensityControlledModel';
materialModels(3).params.kinematic_viscosity = kinematicViscosity;
materialModels(3).params.rho = 1.0 + 0.5*deltaRho;
materialModels(3).params.boundary_norm = '0.0 0.0 -1.0';

materialModels(4).name = 'd3q19DensityControlledModel';
materialModels(4).params.kinematic_viscosity = kinematicViscosity;
materialModels(4).params.rho = 1.0 - 0.5*deltaRho;
materialModels(4).params.boundary_norm = '0.0 0.0 1.0';
```

The newly defined material models are then assigned to the lattice nodes using a color by number approach:

```
voxels = ones(nX,nY,nZ); % model 1 : LBD3Q19 bounceback model
voxels(2:end-1,:,:) = 2; % model 2 : LBD3Q19 model
voxels(2:end-1,:,1) = 3; % model 3 : density on lower boundary
voxels(2:end-1,:,end) = 4; % model 4 : density on upper boundary

nodeMap = generateNodeMaterialModelMap(voxels);
```

Once the material models have been assigned, the initial conditions for the lattice are generated. In this case, a pressure gradient is generated along the x-axis and the initial fluid densities are set to the pseudo-equilibrium populations using the `calculateLBData` function.

```
%%%%%% Create density (pressure) gradient along z axis %%%%%
minZ = min(nodeMap(:,3)); maxZ =  max(nodeMap(:,3));
densities(nodeMap(:,4)) = 1.0+deltaRho*(0.5-(nodeMap(:,3)-minZ)/nZ);

%%%%%% Convert densities to lattice-Boltzmann data %%%%%
F = calculateLBData(densities);
```

Finally the data is written to file:

```
dlmwrite(nodeMapFile, nodeMap, 'precision', 9);
writeLBData(initialLBDataFile,F);

writeMaterialModelFile(modelDataFile,materialModels);

%%%%%% Write simulation input file %%%%%%
% set input parameters
simulationInput.output_file_prefix = outputFilePrefix;
simulationInput.initial_lbdata_file = initialLBDataFile;
simulationInput.max_timesteps = maxTimeSteps;
simulationInput.output_frequency = outputFrequency;
simulationInput.node_to_material_model_map_file = nodeMapFile;
simulationInput.material_model_data_file = modelDataFile;

writeSimulationInputFile(inputConfigurationFile, simulationInput);
```

4.1.2 Postprocessor

The postprocessor for the Poiseuille example 1) loads the lattice Boltzmann data, 2) calculates the fluid velocities, and 3) compares the result to the analytical solution.

The postprocessor reads the output from the main simulation using the `readLBData` command and then calculates the fluid velocities using the `calculateVelocities` command:

```
outputFile = 'out_010000.lbData_bin';
data = readLBData(outputFile);
v = calculateVelocities(data);
```

The velocities are then plotted along with the analytical solution for comparison (Figure 4.1.2).

4.2 Two dimensional tutorial: `example_2D`

The two dimensional tutorial is equivalent to the Poiseuille example above, except that a two-dimensional nine-velocity (D2Q9) lattice is used in place of the D3Q19 lattice employed in the original Poiseuille flow simulation. This example is included to demonstrate how to invoke different lattice-types with the simulation engine.

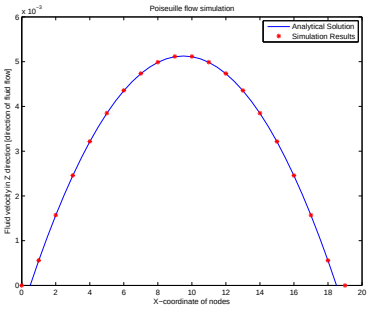


Figure 4.1: The postprocessor for the Poiseuille example tutorial compares the simulated fluid velocities to the analytical solution.

4.2.1 Preprocessor

In the preprocessor, the key differences between the standard Poiseuille flow simulation and the two dimensional simulation are:

1) The default D3Q9 lattice is replaced with the D2Q9 lattice by noting the lattice type in the simulation input file arguments:

```
simulationInput.lattice_type = 'D2Q9';
```

2) Material models for the D2Q9 lattice are used in place of the D3Q19 lattice models:

```
materialModels(1).name = 'd2q9BouncebackModel';

materialModels(2).name = 'd2q9Model';
materialModels(2).params.kinematic_viscosity = kinematicViscosity;

materialModels(3).name = 'd2q9DensityControlledModel';
materialModels(3).params.kinematic_viscosity = kinematicViscosity;
materialModels(3).params.rho = 1.0 + 0.5*deltaRho;
materialModels(3).params.boundary_norm = '0.0 -1.0 0.0';

materialModels(4).name = 'd2q9DensityControlledModel';
materialModels(4).params.kinematic_viscosity = kinematicViscosity;
materialModels(4).params.rho = 1.0 - 0.5*deltaRho;
materialModels(4).params.boundary_norm = '0.0 1.0 0.0';
```

3) The initial fluid packet density distribution is calculated using the weights for the D2Q9 lattice.

```
w = d2q9LatticeWeights;
F = calculateLBData(densities,[],w);
```

4.2.2 Postprocessor

The key difference between the postprocessor for the two dimensional simulation and that of the standard Poiseuille flow simulation is that the D2Q9 lattice directions are used to calculate the fluid velocities from the lattice data:

```
c = d2q9LatticeDirections;
v = calculateVelocities(data,c);
```

4.3 Dynamic library tutorial: `example_dynalibs`

This tutorial demonstrates how to use dynamic libraries to create user defined material models. The dynamic library in the tutorial defines a pure-streaming material-model (*i.e.* one in which the collision step has no effect). The simulation uses the model to propagate an initial density distribution (in the shape of the word LBHydra) about the lattice. Figure 4.3 shows the initial fluid packet density distribution and the density distribution after 20 timesteps.

The C++ files used to create the dynamic library are discussed in Chapter 7, along with other options for creating dynamic libraries for lattices and lattice functions.

4.4 Immiscible model tutorial: `example_immiscible`

This tutorial demonstrates how to simulate a multiple component model – in this case representing a two component immiscible fluid.

4.4.1 Preprocessor

The two phases in the simulation are noted in the simulation input parameters with the `num_phases` parameter:

```
simulationInput.num_phases = 2;
```

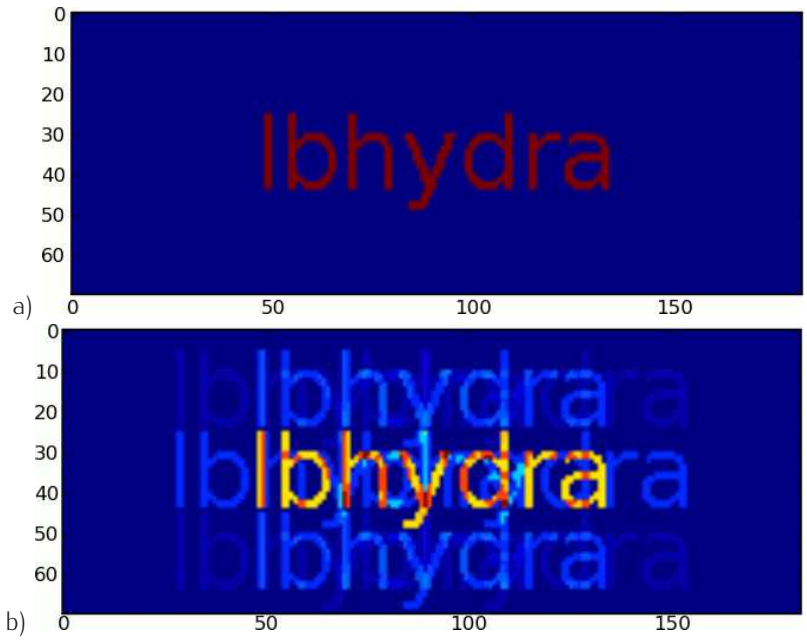


Figure 4.2: The postprocessor for the dynamic library tutorial plots the results of the pure streaming model: a) the initial density distribution and b) the distribution after 20 timesteps.

Initial fluid packet densities are calculated for both components:

```
phase0Rho = phase1AvgRho+deltaRho*(2*rand( length(nodeMap),1 ) -1);
phase1Rho = 1.0 - phase0Rho;

FDP0 = calculateLBData(phase0Rho);
FDP1 = calculateLBData(phase1Rho);

writeLBData([lbDataFilePrefix,'.p0.lbData_bin'],FDP0);
writeLBData([lbDataFilePrefix,'.p1.lbData_bin'],FDP1);
```

Multiple intial lattice data files are recorded in the simulation input file by the lattice data file prefix (instead of a single lattice data filename).

```
simulationInput.initial_lbdata_file_prefix = lbDataFilePrefix;
```

The material models note the interactions between the two phases through the `immiscible_phase_id` parameter:

```
% Model 1: Immiscible for model phase 0
materialModels(1).name = 'd3q19ImmiscibleFluidModel';
materialModels(1).params.kinematic_viscosity = nu;
materialModels(1).params.immiscible_phase_id = phaseId1;
materialModels(1).params.interaction_force_scaling_const ...
                          = interactionForceScalingConst;

% Model 2: Immiscible model for phase 1
materialModels(2).name = 'd3q19ImmiscibleFluidModel';
materialModels(2).params.kinematic_viscosity = nu;
materialModels(2).params.immiscible_phase_id = phaseId0;
materialModels(2).params.interaction_force_scaling_const ...
                          = interactionForceScalingConst;
```

Note that the phases are numbered starting from 0.

4.4.2 Postprocessor

Initially the two fluids are equally mixed except for a slight perturbation used to initiate the phase separation. As the simulation proceeds, the fluids segregate into separate regions. The postprocessor plots the densities of the two fluids along perpendicular planes within the fluid volume (Figure 4.4.2).

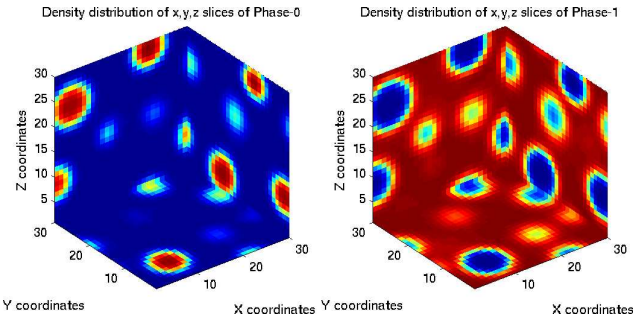


Figure 4.3: The postprocessor for the immiscible example tutorial displays the densities of the two fluids, demonstrating their segregation into separate regions (Matlab post-processor shown).

CHAPTER 5

Pre- and Post-processing Toolboxes

Python and Matlab toolboxes containing commonly used preprocessing and post-processing functions are included in the LBHydra package. The following sections show how to make these toolboxes available in Python and Matlab scripts and describe the functions in detail.

5.1 LBHPythonToolbox

Under the basic installation the `LBHPythonToolbox` is copied to `usr/local/lib/python/LBHPythonToolbox` from the main LBHydra directory. To use the toolbox, add the following to the beginning of the Python script:

```
import sys
sys.path.append('usr/local/lib/python/LBHPythonToolbox')
import lbHydra
import d3q19Lattice
```

Note that the Python modules storing the lattice directions and weights (*e.g.* `d3q19Lattice`) are separate from the main LBHydra Python toolbox and hence should be imported individually.

5.2 LBHMatlabToolbox

By default, the Matlab toolbox is copied from the main LBHydra directory and installed in the directory
`usr/local/lib/matlab/LBHMatlabToolbox`. To employ the functions in a script, add the LBHMatlabToolbox directory to the Matlab search path with the `addpath` command at the beginning of a Matlab session:
`addpath(usr/local/lib/matlab/LBHMatlabToolbox)`

Use the `help` command within Matlab for more instructions on individual functions and `help LBHMatlabToolbox` for a list of all available functions.

5.3 Toolbox Functions

The following functions are present in the Python and Matlab toolboxes.

addPhaseToMaterialModelMap

This function adds a new phase to an existing node-material model map.

Matlab, Python
`addPhaseToMaterialModelMap(mmMap,voxels)` appends the material model ids in `voxels` to the material model map `mmMap` as a new phase and returns the result.

calculateDensities

Calculate fluid densities from lattice-Boltzmann data.

Matlab, Python
`calculateDensities(lbData)` Returns the density of the lattice-Boltzmann data

calculateLBData

Calculate lattice-Boltzmann fluid packet densities from fluid density and velocity distributions.

Matlab
`calculateLBData(rho)` returns lattice-Boltzmann density data calculated from the array of fluid densities `rho`.
`calculateLBData(rho,v)` returns lattice-Boltzmann densities based on from the fluid densities `rho` and velocities `v`.

`calculateLBData(rho,[],weights)` uses the lattice weights `weights` in place of the default D3Q19 lattice weights.
`calculateLBData(rho,v,weights,directions)` uses lattice weights `weights` and lattice directions `directions` in place of the default D3Q19 lattice directions and weights.

Python
The equivalent Python function is:

```
calculateLBData(rho,vel = [],weights = d3q19Lattice.weights(),
                c = d3q19Lattice.directions())
```

calculateVelocities

A utility function to calculate fluid velocities from lattice-Boltzmann data.

Matlab
`calculateVelocities(lbdata)` calculates the velocities from D3Q19 lattice-Boltzmann data.
`calculateVelocities(lbdata,c)` uses the lattice directions (an $N \times 3$ matrix given in `c`) in place of the default D3Q19 lattice directions.

Python
The equivalent Python function call is

```
calculateVelocities(A, c = d3q19Lattice.directions())
```

dMqNLatticeDirections, dMqNLatticeWeights

These functions return directions and corresponding weights for different lattice types. Supported lattices types include: D1Q3, D2Q9, D3Q7, and D3Q19 lattices. Lattice directions are returned as a $N \times 3$ matrix, weights are returned as an $N \times 1$ vector.

Matlab

```
c = dMqNLatticeDirections();
w = dMqNLatticeWeights();
```

Python

```
c = dMqNLattice.directions()
w = dMqNLattice.weights()
```

generateNodeMaterialModelMap

Matlab,Python
`generateNodeMaterialModelMap(voxels)` uses the three dimensional array

`voxels` to create a material model map, which stores the node coordinates, the node id and the material model. Elements of `voxels` less than or equal to 0 are considered holes and are removed from the list of nodes.

generateNodeNeighborList

Creates a matrix describing the network of nodes comprising the lattice. This function is required for sparse lattices only. For dense lattices (without holes) `generateNodeMaterialModelMap` is sufficient and faster.

Matlab
`generateNodeNeighborList(voxels)` uses the three dimensional array `voxels` to create a list of node neighbors for the D3Q19 lattice. Elements of `voxels` less than or equal to 0 are considered holes and are removed from the list of nodes.
`generateNodeNeighborList(voxels,c)` uses the lattice directions in `c` in place of the default D3Q19 lattice directions.
`[nodeNbrs,nodeModelMap] = generateNodeNeighborList(voxels)`
`[nodeNbrs,nodeModelMap] = generateNodeNeighborList(voxels,c)` creates both a list of the node neighbors and a material model map, which stores the node coordinates, the node and the material model *i.e.*
`nodeMaterialModelMap = [X,Y,Z,Node,MaterialModel]`

Python
The equivalent Python function is:
`generateNodeNeighborList(voxels,c = d3q19Lattice.directions())`

inputParameterQuery

This script returns the list of supported input configuration parameters.

modelQuery

This script returns a list of supported material models and their arguments.

readBobData

This script is used to read .BOB (Brick of Bytes) output data files.
`readBobData(file,nX,nY,nZ)` reads Brick of Bytes data from a file with a brick of dimensions `nX`, `nY`, `nZ`.

readLBData

This function reads the fluid packet data represented in either text or binary format from a given file.

Matlab

`lbData = readLBData(file)` returns the lattice Boltzmann data in the file `file`. The file is assumed to be in the binary format, unless the file name ends in '.lbData' in which case a text format is assumed. The data is returned as a matrix, with one row per node and one column per lattice-Boltzmann lattice direction.

`lbData = readLBData(file,format)` reads the data from a file in the designated format. Supported format strings are 'binary' and 'text'.

Python

The equivalent Python command is

```
readLBData(filename, format = 'binary')
```

readVoxelFile

This function reads a 3D array from a voxel file – a text file containing the dimensions of a three dimensional array followed by the XY slices of the array in order along the Z axis.

Matlab, Python

`data = readVoxelFile(file)` reads the array data from the file `file` and returns a three dimensional array `data`.

Matlab only

`[nX nY nZ data] = readVoxelFile(file)` reads the voxel data from the file `file` and returns the dimensions (`nX nY nZ`) and the data as a one dimensinal vector, `data`.

writeLatticeFunctionFile

This function writes lattice function data to the specified file.

Matlab, Python

`writeLatticeFunctionFile(file,latticeFunctionStruct)` writes lattice function data to a file `file`. The argument `latticeFunctionStruct` is a Matlab structure or Python dictionary containing the name of the lattice function as a key with a string value, and other lattice function parameters as key fields with either string or numerical values, *i.e.*

```
latticeFunction.name = 'functionName';
```

```
latticeFunction.aParameter = 'stringValue';
latticeFunction.anotherParameter = 'stringvalue % with comment';
latticeFunction.aThirdParameter = numericalValue;
```

writeLBData

This function writes fluid packet densities to a file in text or binary format.

Matlab
`writeLBData(filename,data,[format])` where `data` is the matrix to be written to the file `filename` in the format specified by `format` (optional). The default format is `binary`. The format string `text` outputs the data as plain text. Any other string will write the data in the binary format.

Python
The equivalent Python command is
`writeLBData(filename, data, format = 'binary')`

writeMaterialModelFile

This function writes material model data to the specified file.

Matlab, Python
`writeMaterialModelFile(matModelFile,materialModels)` Writes the material model file `matModelFile`, using an array of Matlab structures or Python dictionaries `materialModels` with fields `number`, `name` and `params`. `materialModels(i).name` lists the name of the ith material model, while `materialModels(i).params` is a Matlab structure or Python dictionary with fields for each material model parameter, *e.g.*
`materialModels(i).params.normalized\_viscosity = '0.1667'`

writeSimulationInputFile

This function writes simulation input data to the configuration file.

Matlab, Python
`writeSimulationInputFile(simulationInputFile,simulationInput)` Writes the file `simulationInputFile` based on the Matlab structure or Python dictionary `simulationInput`. The structure or dictionary `simulationInput` contains fields representing the simulation parameters with either string or numerical values, *i.e.*

```
simulationInput.key = 'value';
simulationInput.key = 'value % comment';
simulationInput.key = value;    % with a numerical value
```

CHAPTER 6

Code structure

6.1 LBHydra Libraries

The LBHydra simulation engine and libraries are designed for Linux operating systems. It can be run on Windows operating systems also, however, the dynamic library options described in Chapter 7 are not supported.

The classes and functions used by LBHydra are divided into two libraries:

1. *Lattice-Boltzmann classes and functions*
   Library: `libLBHlb.so`
   Include directory: `lbHydra/LatticeBoltzmann/`
   This library contains classes and functions specific to the lattice-Boltzmann simulations. These include: base and derived lattices and material model classes; velocities and weights for different lattice-types; functions for reading and writing simulation data; and factory classes for user generated models, lattice functions and lattices.

2. *Miscellaneous classes and functions*
   Library: `libLBHmisc.so`
   Include directory: `lbHydra/Miscellaneous/`
   This library contains minor miscellaneous functions used throughout the code – string manipulation functions, functions for loading model parameters, timer functions, and a three dimensional vector class.

6.2 Employing LBHydra in other C++ programs

The pseudo-code below illustrates how to incorporate the lattice-Boltzmann libraries in another C++ application. The libraries make liberal use of Standard Template Library classes and functions.

```
// Include Lattice Boltzmann classes and functions
#include "LatticeBoltzmann/lbD3Q19Lattice.h"
#include "LatticeBoltzmann/lbIOParams.h"
#include "LatticeBoltzmann/lbReadWrite.h"

int main(int argc, char ** argv) {
  ...
  // Perform actions prior to the lattice-Boltzmann simulation
  ...

  // Load lattice-Boltzmann parameters from a file
  std::string filename = "aFile.simulationInput";
  LBParamStruct lbSimParams;
  loadIOParametersFromFile(filename, lbSimParams);

  // Create a new D3Q19 lattice
  LBD3Q19Lattice theLattice(lbSimParams);

  // Loop over 1000 timesteps
  for(int ts =0; ts < 1000; ++ts){

    theLattice.implementTimesteps(1); // One timestep

    ...
    // Query/Modify the lattice with set and get functions
    // eg: theLattice.getDensity(node,phase);
    //      theLattice.setIncomingVelocityDensity(node,phase,dir);
    ...

    // Output data (every timestep)
    outputLBData(theLattice, lbSimParams, ts);
  }

  ...
  // Perform actions after the lattice-Boltzmann simulation
  ...
}
```

To modify the lattice between the collision and streaming steps, the command
`theLattice.implementTimesteps(1);`
may be replaced with
```
theLattice.implementCollisionStep();
// ... Perform action here ...
theLattice.implementStreamingStep();
```

6.3 Key classes and functions

`struct LBParamStruct`

This structure stores the data contained in the simulation input files. This data can either be loaded from a file using the `loadIOParametersFromFile` command, *e.g.*
`loadIOParametersFromFile(filename, lbSimParams);`
or by setting the variables of the `LBParamStruct` directly. These variables include:

- `std::string initialLBDataFile, initialLBDataFilePrefix;`
  Naming convention for the file(s) containing the lattice-Boltzmann data. `initialLBDataFile` is used for single phase models, `initialLBDataFilePrefix` for multiphase models.
- `std::string initialLBDataFormat;`
  A string describing the data format for the input files. May be either "text" or "binary".
- `std::string nodeModelMapFile,modelDataFile,latticeGraphFile;`
  Strings containing the node-material model map filename, the model data filename and the filename of the lattice graph (the last is used in sparse lattices only).
- `std::string preCollisionFile,postCollisionFile;`
  The names of the files describing the precollision and post collision lattice functions.
- `std::string outputPrefix, outputLBDataFormat;`
  The prefix for the output files, and the output file format.
- `int maxTimesteps,outputFreq;`
  The maximum number of timesteps and the number of timesteps between output files being generated.

- `int numPhases;`
  The number of phases in the simulation.

- `std::map<std::string, std::string> paramMap;`
  A map containing other, lattice-specific arguments.

`class LBLattice`

This is the parent class for all lattice-Boltzmann lattices. The base LBLattice class is not directly instantiated, but is inherited by any class considered to be a lattice-Boltzmann lattice. The functions common to the LBLattice class and its derived classes are described in Chapter 7.

In LBHydra, lattice classes contain the fluid packet data, material models, and the simulation geometry. They support functions to query the number of nodes and phases, as well as the ability to set and get node neighbors, fluid packet densities, lattice densities and momenta, and material models.

The most important function of this class is implementTimesteps, which invokes the streaming and collision step functions for each node/phase combination.

`class LBMaterialModel`

This is the parent class for all lattice-Boltzmann material models. The base LBMaterialModel class is not directly instantiated, but is inherited by any class considered to be a material model. The functions common to the LBMaterialModel class and its derived classes are described in Chapter 7.

A material model can be though of as a collection of functions that act on fluid packet data. Perhaps the most important of these functions is the collision step, which converts a set of incoming fluid packet densities to a set of outgoing fluid packet densities. In addition the material model must contain other functions to return the number density (in most cases the sum) of the fluid packets at each node, and the first moment or "momentum" of the fluid packet data (in most cases the sum of the product of the fluid packet data with the associated lattice velocities).

`class Vect3D`

Part of the Miscellaneous Library, this class describes a three dimensional vector. It includes functions for calculating vector sums and products (dot, cross and outer) and matrix multiplication.

`function loadIOParametersFromFile`

This function parses simulation input files and stores the results in an LBParamStruct structure. It takes the form:

```
void loadIOParametersFromFile(std::string filename, LBParamStruct& lbStruct)
```

function outputLBData

This function writes the current state of the lattice to a file. It takes the form:
`void outputLBData(LBLattice lattice, LBParamStruct lbParams, int timestep)`
The output filename is in the form `PREFIX_TIMESTEP{.pX}.SUFFIX` where the prefix is determined by the arguments to LBParamStruct, the suffix is either `lbData` or `lbData_bin` depending on the output format (also determined by the LBParamStruct) and `{.pX}` is an optional part of the file name used in multiphase simulations to indicate the phase id number.

CHAPTER 7

Extending LBHydra with Dynamic Libraries

LBHydra allows users to create customized material models, lattice functions and lattices without having to modify (or recompile) the main program's source code. This is achieved by creating dynamic libraries that are linked to LBHydra at runtime. The following sections describe the C++ classes in these different dynamic libraries, and how to link the libraries the to the main LBHydra simulation engine. A tutorial demonstrating how to make custom material models with dynamic libraries is provided in the `tutorials/example_dynalibs` directory.

7.1 Dynamic libraries for Material Models

The material model class interface available for extension is:

```
class LBMaterialModel {
  public:
    LBMaterialModel (std::map<std::string,std::string> params,
                     LBLattice& lbLattice);
    void setParameters (std::map<std::string,std::string> params,
                        LBLattice& lbLattice);
    double getDensity (const double f[],int n, int p) const;
    Vect3D getMomentum (const double f[],int n, int p) const;
    void implementCollisionStep (const double fin[], double fout[],
```

```
                                    int n, int p);
    static const std::string materialModelName_;
    static const std::string materialModelArgs_;
};
```

The constructor

```
LBMaterialModel(std::map<std::string,std::string> params,
                LBLattice& lbLattice);
```

The variable `params` is a standard template library map whose keys are strings representing the names of the model parameters and values are strings representing parameter values. The set of parameters allowed by a model can be queried using the `-m` command-line option. The model's supported parameter strings are registered to the main program using the `materialModelArgs_` element of this class, which will be discussed later. The lattice structure passed as the second argument allows the material models to query the state of the lattice on construction, or store a pointer to either monitor or modify the lattice at later times in non-standard ways. For many material models this second argument is not necessary and can be ignored.

As discussed further below, the constructor should initialize default values of the material model, but delegate the task of parsing the parameters to the `setParameters()` function.

```
setParameters(std::map<std::string,std::string> params,
              LBLattice& lbLattice);
```

This command is used to set the model parameters. The arguments are the same as those described in the constructor, and in fact, `setParameters` should be invoked by the constructor on initialisation.

Separating the `setParameters` function from the constructor serves two purposes. Most importantly, it provides a means of modifying the material model parameters mid-simulation without resetting default values. In addition, for material models that inherent from a another class, it allows the derived class to modify material parameters in ways that might otherwise be prevented by the parent class's implementation.

```
getDensity(const double f[],int n, int p)
```

This function calculates the number density of fluid packets (*i.e.* the sum of fluid packets) for node `n` and phase `p`. In most cases, the node and phase identifiers (`n` and `p`) will not be needed in the calculation.

```
getMomentum(const double f[],int n, int p)
```

This function calculates the 'momentum' of the fluid packets, *i.e.* the sum of the

product of the fluid packets with the lattice velocities ($\sum f_i \mathbf{c}_i$, and return the X,Y and Z components as a 3D vector Vect3D. In most cases, the node and phase identifiers will not be needed in the calculation.

```
implementCollisionStep(const double fin[], double fout[],
                       int n, int p)
```

The most important subroutine in the material model class, this function applies the collision step to update the outgoing fluid packets, `fout`, based on the fluid packets arriving at the node, `fin`. In most cases, the node and phase identifiers (`n` and `p`) will not be needed in the calculation.

`materialModelName_`
The string used to identify the material model in the material model data file.

`materialModelArgs_`
A comma-delimited set of strings identifying the parameters supported by the material model.

Imparting dynamic behavior

Material models must be registered in the simulation engine's material model factory prior to the start of simulation. This is achieved with autoregistration. The simplest method to register a model in the factory is to use the `REGISTER_LB` compiler directive:

```
namespace{
  REGISTER_LB( materialModelClassName )
}
```

where `materialModelClassName` is the class name of the new LBMaterialModel class. The namespace directive is not strictly necessary, however it reduces the potential for name conflicts with other user-defined material models.

7.1.1 Dynamic material model example

The following code blocks are taken from the dynamic library example from the LBHydra package. The model implements a pure streaming model (*i.e.* the collision step has no effect on the simulation) to demonstrate the basic form of a third-party material model library.

myModel.h:

```
#ifndef MYMODEL_H
#define MYMODEL_H
#include "lbMaterialModel.h"

class myModel : public LBMaterialModel{
  public:
    myModel(std::map<std::string,std::string> params,
            LBLattice& lbLattice);
    ~myModel() {}; //empty

    void setParameters(std::map<std::string,std::string> params,
                       LBLattice& lbLattice);

    void implementCollisionStep(const double fin[], double fout[],
                                int nodeId, int phaseId);

    static const std::string materialModelName_;
    static const std::string materialModelArgs_;

    private:
      int numDirs_;
};
#endif
```

myModel.cpp:

```
#include "myModel.h"
#include <stdlib.h>

// lbHydra headers
#include "lbMaterialModelCatalogue.h"
#include "miscFunc.h"

const std::string myModel::materialModelName_ = "myModel";
const std::string myModel::materialModelArgs_
                              = "number_of_directions";

/// Model constructor
myModel::myModel(std::map<std::string,std::string> params,
                 LBLattice& lbLattice) {
  numDirs_ =  19;  // set default value
  this->setParameters(params, lbLattice);
}

void myModel::setParameters(std::map<std::string,std::string> params,
                            LBLattice& lbLattice) {
  // get values from input parameters
  std::map<std::string,std::string>::iterator iter;
  for (iter = params.begin(); iter!=params.end(); iter++ ) {
    const std::string& key = iter->first;
    std::string& value = iter->second;
    if ( key == "number_of_directions" ){
      numDirs_ = fromString<int>(value);
    }
  }
}

/// Collision step propagates f without alteration (pure streaming)
void myModel::implementCollisionStep(const double fin[],
                                     double fout[],
                                     int nodeId, int phaseId) {
  for(int i=0; i<numDirs_; ++i) fout[i] = fin[i];
}

/// Register the model name and args in the material model factory.
namespace{ REGISTER_LB( myModel ); }
```

The code is compiled with the following command:

```
g++  -Wall -shared -fPIC -O3 -I$(LBHYDRA_PATH)/Miscellaneous
-I$(LBHYDRA_PATH)/LatticeBoltzmann -o myModel.so myModel.cpp
```

where `$(LBHYDRA_PATH)` is the path to the LBHydra include directory.

The dynamic lattice and lattice function libraries described below can also be compiled using a similar set of commands.

7.2 Dynamic Lattice Libraries

Dynamic lattice libraries allow users to develop new lattice-Boltzmann lattice types. These libraries extend existing lattice types beyond those currently supported by the main LBHydra code (*e.g.* D3Q13 lattices) and to create new numerical methods used to store the lattice (*e.g.* sparse/dense lattice types). For example, the GPU extensions to the LBHydra code are implemented with these dynamic lattice libraries.

The lattice class interface is:

```
class LBLattice{
  public:
    LBLattice(std::map<std::string,std::string> params);

    double getIncomingVelocityDensity(int node,int phase,
                                      int dir) const;
    double getOutgoingVelocityDensity(int node,int phase,
                                      int dir) const;

    void getIncomingVelocityDensities(int node,int phase,
                                      std::vector<double>& f) const;
    void getOutgoingVelocityDensities(int node,int phase,
                                      std::vector<double>& f) const;

    void setVelocityDensity(int node, int phase,int dir,double f);
    void setIncomingVelocityDensity(int node, int phase,
                                    int dir,double f);
    void setOutgoingVelocityDensity(int node, int phase,
                                      int dir,double f);

    void setVelocityDensities(int node, int phase,
                              const std::vector<double>& f);
    void setIncomingVelocityDensities(int node, int phase,
                                      const std::vector<double>& f);
    void setOutgoingVelocityDensities(int node, int phase,
                                      const std::vector<double>& f);
```

```
    void setMaterialModelPtr(int node, int phase,
                             LBMaterialModel* materialModelPtr);

    LBMaterialModel* getMaterialModelPtr(int node, int phase);

    void setLatticeFunctionPtr(LBLatticeFunction* funcPtr);

    int size(void) const;
    int numberOfPhases(void) const;
    int numberOfLBVelocities(int phase) const;

    Vect3D getMomentum(int node,int phase) const;
    double getDensity(int node,int phase) const;

    int getNodeNeighbor(int node, int dir) const;
    std::vector<int> getNodeNeighbors(int nodeId) const;
    void setNodeNeighbor(int nodeId, int dir, int neighborId);
    void setNodeNeighbors(int nodeId, std::vector<int>& neighborList);

    void implementStreamingStep(void);
    void implementCollisionStep(void);
    void implementTimesteps(int n);

    void implementPreCollisionStep(void);
    void implementPostCollisionStep(void);
};
```

Imparting dynamic behavior

Similar to the material models, lattices are implemented in LBHydra using a lattice factory with an autoregistration method. The simplest method to register a new lattice in the factory is to invoke the `REGISTER_LB_LATTICE` compiler directive. This is achieved by including the following in the `.cpp` class file:

```
namespace{
  REGISTER_LB_LATTICE( LBLatticeClassName )
}
```

where `LBLatticeClassName` is the name of the newly created lattice class.

7.3 Dynamic Lattice-Function Libraries

Dynamic lattice-functions allow users to develop pre- and post-collision routines. These routines add additional computational steps to the traditional collision and

streaming lattice-Boltzmann stages. Potential applications for lattice functions include coupling LBHydra to external simulations, modifying the lattice fluid packet data and material models mid-simulation, and outputting additional simulation data.

The lattice-function class interface is:

```
class LBLatticeFunction {
  public:
    LBLatticeFunction (std::map<std::string,std::string> params,
                       LBLattice& lbLattice);
    void setParameters (std::map<std::string,std::string> params,
                        LBLattice& lbLattice);

    void precollisionFunction (LBLattice& theLattice);
    void postcollisionFunction (LBLattice& theLattice);

    static const std::string latticeFunctionName_;
    static const std::string latticeFunctionArgs_;
};
```

The constructor:

`LBLatticeFunction(std::map<std::string,std::string> params, LBLattice& lbLattice)`

`params` is a standard template library map whose key,value pairs are lattice function parameter names and parameter value strings respectively. The parameters allowed by a lattice function can be queried using the `--latticeFunctions` command-line option. These are registered to the main program using the `latticeFunctionArgs_` element of this class. The lattice structure is passed as the second argument to allow models access to node and phase data. The constructor delegates the task of parsing the parameters to the `setParameters()` function.

`setParameters(std::map<std::string,std::string> params, LBLattice& lbLattice)`

This is used to parse the parameters supported by the lattice function.

`precollisionFunction(LBLattice& theLattice)`

The precollision function is called after each streaming step, but before the next collision step.

`postcollisionFunction(LBLattice& theLattice)`

The postcollision function is called after each collision step, but before the next

streaming step.

`latticeFunctionName_`
The name used in the lattice function file to identify the type of lattice function.

`latticeFunctionArgs_`
A comma-delimited set of legal parameters supported in the lattice function.

Imparting dynamic behavior

As with the other dynamic classes, dynamic lattice functions are implemented in LBHydra via a factory with an autoregistration method. New functions are registered in the factory with the `REGISTER_LBF` compiler directive in the lattice function's `.cpp` class file. *e.g.*

```
namespace{
  REGISTER_LBF( LBVisualisationFunction )
}
```

7.4 Accessing the dynamic libraries

LBHydra will link to dynamic libraries in the current directory by default. It may be notified of additional dynamic library directories with the `-d` command line options, *e.g.*
`lbhydra -f anExample.simulationInput -d /path/to/dynamic/library`
Multiple `-d` command line options may be employed together to indicate multiple dynamic library locations.

A single library can contain multiple user defined classes. This approach is recommended if two or more classes are interdependent, for example if a material model requires a pre- or post-collision function to operate correctly.

CHAPTER 8

LBHydraGPU

8.1 Overview

Lattice-Boltzmann methods are well suited to implementation on single-instruction multiple data (SIMD) parallel processing environments. In particular, substantial performance increases have been achieved with lattice-Boltzmann methods (up to 40x faster) by exploiting the SIMD environment found in computer graphics processing cards.

LBHydraGPU is an extension to LBHydra package that runs lattice-Boltzmann simulations on nVIDIA graphics processing units (GPUs). LBHydraGPU is based on CUDA, a C-like programming language that allows programs to be written for graphics cards. Although CUDA is a dramatic improvement over previous generations of graphics programming tools, the GPU programming model lacks some of the flexibility of CPU based programs.

To overcome these difficulties, LBHydraGPU follows a modified version of the simulation pipeline given in Chapter 3 (Figure 8.1). After the simulation input files are generated by the preprocessor, they are passed to the LBHydraGPU code parser and compiler `LBHgpupy`. `LBHgpupy` creates a dynamic library containing a customized lattice function that enables the lattice-Boltzmann simulation to run on the GPU.

Users should be aware that LBHydraGPU is undergoing development and it is likely that parts of this module will change as the code matures. In particular, the current code parser is implemented in Python, future plans for development

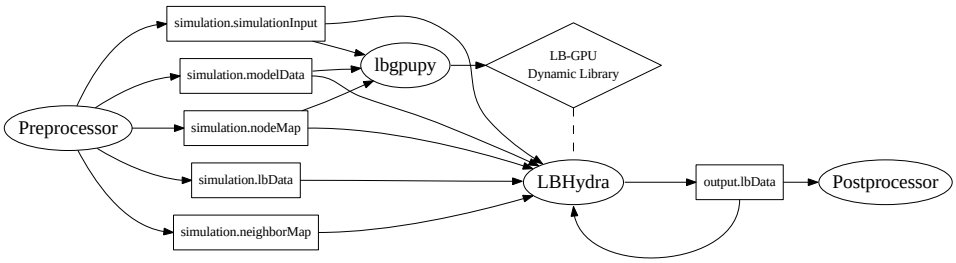


Figure 8.1: LBHydraGPU simulation pipeline.

include a C++ version of the parser integrated with the main LBHydra engine.

8.2 Required hardware and software

The LBHydraGPU code parser and compiler **LBHgpupy** requires:
1) a CUDA compliant nVIDIA graphics card
2) an installed copy of the nvcc CUDA compiler
3) Python 2.6 with the numpy (numerical python) libaries

8.3 LBHgpupy

To run the Lattice-Boltzmann GPU parser and compiler:

1. Install the main LBHydra program.
2. Generate the required input files using the pre-processor.
3. Supply the LBH-GPU parser with the required simulation input file:
   `LBHgpupy/parser.py -f GPUExample.simulationInput`
4. Compile the generated GPU code to create the dynamic library required for the GPU simulation. The name of the library will be in the form `latticeType.so` where `latticeType` is the lattice type named in the simulation input file.
   `LBHgpupy/compiler.py -f GPUExample.simulationInput`
5. Run LBHydra on the generated simulation input files.
   `lbHydra -f example.simulationInput`

Notes:

a) Steps 3 and 4 above are only required when first generating the dynamic library. If the id numbers and the parameters for the lattice models in each phase do not change, the same dynamic library can be used in multiple simulations.

b) New user-defined material models can be added to the LBHgpu parser with the `-d` option.
`LBHgpupy/parser.py -f GPUExample.simulationInput -d userModel.py`

c) In addition, for certain models it may be necessary to make user defined CUDA functions available to the dynamic library code. This is possible with the `-I` option, eg.
`LBHgpupy/parser.py -f GPUExample.simulationInput -d userModel.py -I userModelFunction.cu`

d) The compiler script assumes the command `nvcc` runs the CUDA compiler. If the system requires a path to `nvcc`, change the NVCC variable when executing the compiler script using `--makeargs`
`LBHgpupy/compiler.py -f GPUExample.simulationInput --makeargs "NVCC=/opt/cuda/bin/nvcc"`

e) Likewise, it is assumed that the CUDA libraries are located at `/usr/local/cuda`. If this is not the case (the compiler should complain that it cannot find the libraries) change the CUDAPATH variable:
`LBHgpupy/compiler.py -f GPUExample.simulationInput --makeargs "CUDAPATH=/opt/nvidia/cuda"`

f) The code for the GPU lattice Boltzmann simulation is contained in a dynamic library `XXX_LBGPU.so` . The main LBHydra code must link to this library to run the GPU simulation. By default, LBHydra will attempt to link to shared object files in the current directory. The LBHydra `-d LIBRARY_PATH` command line option can be used to specify new directories, if the dynamic library is not in the current directory.

APPENDIX A

File Formats

A.1 Input files

These files are used by lbHydra to begin the simulation.

A.1.1 Input configuration file: XXX.simulationInput

This file contains the location of the other input files and additional data regarding the output frequency, maximum numbere of timesteps etc. It is passed to the LBHydra simulation engine in one of the following ways:

- Using the command line switch `-f`, *e.g.*
  `lbHydra -f XXX.simulationInput`
- By piping the file into the simulation using the input redirection operator `<`, *e.g.*
  `lbHydra < XXX.simulationInput`

Each line of input configuration file is of the format: `<parameter>=<value>`, where `<parameter>` is the parameter name, and `<value>` the parameter value or a vector of values separated by delimiters.

An example simulation input file might contain:

```
output_file_prefix=AnExampleOutput
initial_lbdata_file=AnExample.lbData_bin
```

```
max_timesteps=10000
output_frequency=1000
node_connectivity_map_file=AnExample.neighborMap
node_to_material_model_map_file=AnExample.nodeMap
material_model_data_file=AnExample.modelData
```

A list of input parameters and the format of their values is available in the Appendix.

A.1.2 Node-Material model map file: XXX.nodeMap

This file connects the node coordinates to node IDs and the material models for each phase of the node. Each line represents a node. The format of each line is as follows:

`<X>,<Y>,<Z>,<NodeID>,<Model_Phase1>,...,<Model_PhaseK>`

The `<Model_Phase>` numbers are mapped to specific material models defined in the material model data file discussed later in this section.

A.1.3 Neighbor map: XXX.neighborMap

This file depicts how the lattice is connected, *i.e.* the neighbors of each node. This input file is not required for non-sparse lattices. Each line contains the ids of one node and its neighbors:
`<NodeID>,<NeighborID1>,<NeighborID2>,...<NeighborIDM>` where M is the number of neighbors.

A.1.4 LBData

These files describe the fluid packet densities for the lattice nodes. A different file is required for each phase. Unlike other file types, this file format appears both in the input and the output of the LBHydra simulation.

Text: XXX.lbData

The text format is structured as a simple a CSV (comma-separated value) file. Binary file output is preferred in most cases, as using the text format may impair performance and waste memory, particularly for larger simulations.
1. Each line represents the density vector of one node
2. Each line contains exactly as many elements as the number of lattice velocities
3. The naming convention is `<prefix>.p<K>.lbData` for the Kth phase
4. Format: `<Density1>,<Density2>,...,<DensityM>` where M is the number of neighbors for each node.

Binary: XXX.lbData_bin

Binary file interpreters and exporters are provided in the Python and Matlab toolboxes. The first few characters of the binary file contain plain text describing the format of each

record. The naming convention for the files is
`<Prefix>_<timestep>[.p{phaseNumber}].lbData`
where the string within the square brackets appears only if there are multiple phases involved in the simulation.

The first entry in the file gives the number of characters in the header written as an unsigned 16 bit integer (two characters), followed by the header itself. The header can be read in a normal text editor - to give an idea as to how to interpret the remainder of the file. Following the header the size in bytes of the floats and number of nodes used in the lattice are stored - written as unsigned 32 bit integers. Last, the lattice data is written to file in the order $f_{0,0} \ldots f_{0,N-1} \ldots f_{1,0} \ldots f_{1,N-1} \ldots f_{Q-1,0} \ldots f_{Q-1,N-1}$ where $f_{i,j}$ gives the fluid packet for the direction i at node j, N is the number of lattice nodes, and Q is the number of lattice directions.

A.1.5 Material model data files: XXX.modelData

This file defines the material models associated with the id numbers in the node-material model map. Each line of this file contains a material model id number, the name of the type of the material model associated with that id number and the parameters that define the material model. The same material model type may appear multiple times to represent models with different parameter values.

Formats:

```
number=<number>,name=<name>,params=<none>
number=<number>,name=<name>,params=<param1>:<value1>;...;<paramN>:<valueN>
```

For example, a simple material model data file might contain:

```
number=1,name=d3q19BouncebackModel,params=none
number=2,name=d3q19Model,params=kinematic_viscosity:0.16666667
number=3,name=d3q19DensityControlledModel,params=kinematic_viscosity:0.16666667;-
rho:1.0025;boundary_norm:0.0 0.0 -1.0
number=4,name=d3q19DensityControlledModel,params=kinematic_viscosity:0.16666667;-
rho:0.9975;boundary_norm:0.0 0.0 1.0
```

A.2 Output files

A.2.1 LBData

The format of this file is exactly the same as the input fluid densities file described in the previous section. It has to be noted that the type (*i.e.* binary/text) of output fluid densities need not match that of the input fluid density format. The naming convention is
`<outputPrefix>_<timestep>[.p{phaseNumber}].lbData`
where the string within the square brackets appears only if there are multiple phases involved in the simulation. The default output prefix is "out".

A.2.2 Brick of Bytes: XXX.bob

The Brick of Bytes (BOB) format is a simple output-only format for visualization. It is a lossy format in which values between user-supplied maximum and minumum limits is scaled to a number in the range 0 - 255 (lower or higher values are set to 0 or 255 respectively) and saved in a binary file. The BOB file format is compatible with the Hierarchical Volume Renderer (HVR) visualization software produced by the Laboratory for Computational Science and Engineering at the University of Minnesota (`http://www.lcse.umn.edu/hvr/hvr.html`).

The BOB file is read/written in the form of consecutive bytes. The general format is: `<Byte1><Byte2>...<ByteN>` where N is the number of nodes. The naming convention is `<outputPrefix>_<timestep>.<BOBParam>.bob`. The default output prefix is "out".

A.2.3 Visualization Tool Kit: XXX.vtk

The Visualization Tool Kit (vtk.org) defines a cross platform set of file formats for different data types. Lattice functions provided with LBHydra make it possible to generate VTK files during the simulation, based on the legacy structured points output format. VTK output for sparse lattices is not currently supported.

APPENDIX B

Material Models

`d2q9BouncebackModel`
Parameters: None

`d2q9DensityControlledModel`
Parameters: boundary_norm, boundary_dir, kinematic_viscosity, rho

`d2q9Model`
Parameters: kinematic_viscosity

`d2q9VelocityAndDensityControlledModel`
Parameters: velocity, rho, boundary_locations

`d3q19BouncebackModel`
Parameters: None

`d3q19ConcentrationControlledMiscibleModel`
Parameters: diffusion_coeff, fluid_phase_id, concentration

`d3q19DensityControlledModel`
Parameters: boundary_norm, boundary_dir, kinematic_viscosity, rho

`d3q19EntropyModel`
Parameters: kinematic_viscosity, collision_frequency

`d3q19ImmiscibleFluidModel`

Parameters: kinematic_viscosity, collision_frequency, immiscible_phase_id, body_force, interaction_force_scaling_const

`d3q19IncompressibleModel`
Parameters: kinematic_viscosity, collision_frequency

`d3q19MRTModel`
Parameters: Se, Sa, Sp, Sq, Sv, Sm

`d3q19MiscibleModel`
Parameters: diffusion_coeff, fluid_phase_id

`d3q19Model`
Parameters: kinematic_viscosity, collision_frequency

`d3q19VelocityAndDensityControlledModel`
Parameters: velocity, rho, boundary_locations

Bibliography

Cyrus K. Aidun and Jonathan R. Clausen. Lattice-Boltzmann method for complex flows. *Annual Review of Fluid Mechanics*, 42(1):439–472, 2010. doi: 10.1146/annurev-fluid-121108-145519.

Shiyi Chen and Gary D. Doolen. Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30(1):329–364, 1998. doi: 10.1146/annurev.fluid.30.1.329.

D. H. Rothman and S. Zaleski. *Lattice Gas Cellular Automata – Simple models of complex hydrodynamics*. Cambridge University Press, 1997.

S. Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Oxford Univ. Press, Oxford, 2001.

D. A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*. Springer, 2000.