# 1 Introduction

## 1.1 Overview

This introduction provides a brief overview of what the *Ludwig* code does, how to obtain and build it, and how to run a sample problem. It is assumed that the reader has at least a general knowledge of Navier-Stokes hydrodynamics, complex fluids, and to some extent statistical physics. This knowledge will be required to make sense of the input and output involved in using the code. Those wanting to work on or develop the code itself will need knowledge of ANSI C, and perhaps message passing and CUDA. We assume the reader is using a Unix-based system.

### 1.1.1 Purpose of the code

The *Ludwig* code has been developed over a number of years to address specific problems in complex fluids. The underlying hydrodynamic model is based on the lattice Boltzmann equation (LBE, or just 'LB'). This itself may be used to study simple (Newtonian) fluids in a number of different scenarios, including porous media and particle suspensions. However, the code is more generally suited to complex fluids, where a number of options are available, among others: symmetric binary fluids and Brazovskii smectics, polar gels, liquid crystals, or charged fluid via a Poisson-Boltzmann equation approach. These features are added in the framework of a free energy approach, where specific compositional or orientational order parameters are evolved according to the appropriate coarse-grained dynamics, but also interact with the fluid in a fully coupled fashion.

A number of other features are catered for is some or all situations. These include fluctuating hydrodynamics, and appropriate fluctuating versions of order parameter dynamics for binary solvents and liquid crystals. Colloidal particles are available for simulation of suspensions and dispersions, and with appropriate boundary conditions binary mixtures, liquid crystals, and charged fluid. Shear flow is available for fluid-only systems implemented via Lees-Edwards sliding periodic boundaries. Not all these features play with all the others: check the specific section of the document for details.

Broadly, the code is intended for complex fluid problems at low Reynolds numbers, so there is no consideration of turbulence, high Mach number flows, high density ratio flows, and so on.

### 1.1.2 How the code works

We aim to provide a robust and portable code, written in ANSI C, which can be used to perform serial and scalable parallel simulations of complex fluid systems based around hydrodynamics via the lattice Boltzmann method. Time evolution of modelled quantities takes place on a fixed regular discrete lattice. The preferred method of dealing with the corresponding order parameter equations is by using finite difference. However, for the case of a binary fluid, a two-distribution lattice Boltzmann approach is also maintained for historical reference.

Users control the operation of the code via a plain text input file; output for various data are available. These data may be visualised using appropriate third-party software (e.g., Paraview). Specific diagnostic output may require alterations to the code.

Potential users should also note that the complex fluid simulations enabled by *Ludwig* can be time consuming, prone to instability, and provide results which are difficult to interpret. We make no apology for this: that's the way it is.

### 1.1.3 Who should read what?

This documentation is aimed largely at users, but includes information that will also be relevant for developers. The documentation discusses the underlying features of the coordinate system, hydrodynamics (LB), and other generic features, and has a separate section for each free energy. You will need to consult the section relevant to your problem.

## 1.2 Obtaining the code

Under the auspices of the CCPs (Collaborative Computational Projects), and in particular CCP5, we have a CCPForge project account at `http://ccpforge.cse.rl.ac.uk/` which provides revision control via SVN, issue tracking, and so on.

**Users:** Code releases may be downloaded via anonymous download

```
http://ccpforge.cse.rl.ac.uk/gf/project/ludwig/frs/
```

The current release is available as `ludwig-0.4.0.tar.gz`. Alternatively, registered users can obtain the code via svn checkout in the same way as developers.

**Developers:** Developers need to create an account at CCPForge, and let one of the project owners[1] know the user name so you can be added to the access list.

To download the entire code, developers need

```
svn checkout --username <user> http://ccpforge.cse.rl.ac.uk/svn/ludwig
```

where `<user>` is replaced by your CCPForge user name. You will see

```
$ ls ludwig
branches tags     trunk
```

which is the usual svn convention for trunk, branches and tags. The process for committing code is described in section 3.5.

If you look in the `trunk` directory, you should see

```
$ ls trunk
config LICENSE    mpi_s  src      tests version.h
docs   Makefile.mk README targetDP util
```

[1] Currently: `kevin@epcc.ed.ac.uk`, `ohenrich@epcc.ed.ac.uk`

# 2 Quick Start for Users

This section provides a brief overview of compiling and running the code, which should allow users to get off the ground. The section assumes the code has been downloaded and you are in the top level directory.

## 2.1 Configuration

You will need to copy one of the existing configuration files from the `config` directory to the top level directory. This file sets all the relevant local details for compilation and so on. For example:

```
$ cp config/lunix-gcc-default.mk config.mk
```

Edit the new `config.mk` file to check, or adjust, the options as required. You will need to identify your local C compiler. The example uses `gcc` in serial, and `mpicc` in parallel. For MPI in particular, local details may vary. A quick way to compile the code is via

```
$ cd tests
$ make compile-mpi-d3q19
```

This integrates a number of separate stages which are described indivually in the following sections. The resulting executable will be `./src/Ludwig.exe` at the top level.

### 2.1.1 targetDP

A `targetDP` layer is required by the code in all cases. It allows compilation for both simple CPU systems and for a number of accelerator systems using a single source code. This will make the appropriate choices depending what has been specified in the top-level `config.mk` file. From the top-level directory

```
$ cd targetDP
$ make
```

should produce the appropriate `libtarget.a` library.

### 2.1.2 Serial

First, compile the MPI stub library in the `mpi_s` directory. You should be able to build the library and run the tests using, e.g.:

```
$ make libc
gcc -Wall -I. -c mpi_serial.c
ar -cru libmpi.a mpi_serial.o
$ make testc
gcc -Wall  mpi_tests.c -L. -lmpi
./a.out
Running mpi_s tests...
Finished mpi_s tests ok.
```

Now compile the main code in the `src` directory. Compilation should look like:

```
$ make serial
make serial-d3q19
make serial-model "LB=-D_D3Q19_" "LBOBJ=d3q19.o"
make lib
```

```
make libar "INC=-I. -I../mpi_s" "LIBS= -L../mpi_s -lmpi -lm"
gcc -D_D3Q19_ -Wall -O2 -I. -I../mpi_s -c model.c
gcc -D_D3Q19_ -Wall -O2 -I. -I../mpi_s -c propagation.c
...
```

which shows that the default lattice Boltzmann model is D3Q19. Successful compilation will provide an executable `Ludwig.exe` which is linked against the MPI stub library.

#### 2.1.3 Parallel

Here, you do not need to compile the stub library. Simply compile the main code with, e.g.,:

```
$ make mpi
make mpi-d3q19
make mpi-model "LB=-D_D3Q19_" "LBOBJ=d3q19.o"
make libmpi
make libar "CC=mpicc"
mpicc -D_D3Q19_ -Wall -O2 -I. -c model.c
mpicc -D_D3Q19_ -Wall -O2 -I. -c propagation.c
...
```

Again, this should produce an executable `./Ludwig.exe` in the current directory, this time linked against the true MPI library.

### 2.2 Running an Example

#### 2.2.1 Input file

The behaviour of *Ludwig* at run time is controlled by a plain text input file which consists of a series for key value pairs. Each key value pair controls a parameter or parameters within the code, for example the system size and number of time steps, the fluid properties density and viscosity, the free energy type and associated parameters (if required), frequency and type of output, parallel decomposition, and so on.

For most keys, the associated property has some default value which will be used by the code if the relevant key is not present (or commented out) in the input file. While such default values are chosen to be at least sensible, users need to be aware that all necessary keys need to be considered for a given problem. However, many keys are irrelevant for any given problem.

A significant number of example input files are available as part of the test suite, and these can form a useful starting point for a given type of problem. We will consider one which uses some of the common keys.

#### 2.2.2 Example input

We will consider a simple example which computes the motion of a single spherical colloidal particle in an initially stationary fluid of given viscosity. The velocity may be measured as a function of time. Assume we have an executable in the `src` directory.

Make a copy of the input file

```
$ cp ../tests/regression/d3q19/serial-auto-c01.inp .
```

Inspection of this file will reveal the following: blank lines and comments — lines beginning with `#` — may be included, but are ignored at run time; other lines are parsed

as key value pairs, each pair on a separate line, and with key and value separated by one or more spaces. Values may be characters or strings, a scalar integer or 3-vector of integers, or a scalar or 3-vector of floating point numbers. Values which are 3-vectors are delineated by an underscore character (not a space), e.g., `1_2_3`, and always correspond to a Cartesian triple $(x, y, z)$. A given value is parsed appropriately in the context of the associated key.

So, the first few lines of the above file are:

```
################################################################################
#
#  Colloid velocity autocorrelation test (no noise).
#
################################################################################

N_cycles 40

################################################################################
#
#  System and MPI
#
################################################################################

size 64_64_64
grid 2_2_2
```

Here, the first key value pair `N_cycles 40` sets the number of time steps to 40, while the second, `size 64_64_64`, sets the system size to be 64 points in each of the three coordinate directions. The `grid` key relates to the parallel domain decomposition as is discussed in the following section.

### 2.2.3 Run time

The executable takes a single argument on the command line which is the name of the input file, which should be in the same directory as the executable. If no input file name is supplied, the executable will look for a default `input`. If no input file is located at all, an error to that effect will be reported.

**Serial:** If a serial executable is run in the normal way with the copy of the input file as the argument the code should take around 10–20 seconds to execute 40 steps. The fist few lines of output should look like:

```
$ ./Ludwig.exe ./serial-auto-c01.inp
Welcome to Ludwig v0.2.16 (Serial version running on 1 process)

The SVN revision details are: 2638M
Note assertions via standard C assert() are on.

Read 20 user parameters from serial-auto-c01.inp

No free energy selected

System details
--------------
System size:  64 64 64
```

```
Decomposition: 1 1 1
Local domain:  64 64 64
Periodic:      1 1 1
Halo nhalo:    1
```

This output shows that the appropriate input file has been read, and the system size set correspondingly with a number of default settings including periodic boundary conditions. "No free energy" tells us we are using a single Newtonian fluid.

Normal termination of execution is accompanied by a report of the time take by various parts of the code, and finally by

```
...
Ludwig finished normally.
```

**Parallel:** The executable compiled and linked against the MPI library can be run with the MPI launcher for the local system, often `mpirun`. For example, a run on 8 MPI tasks produces a similar output to that seen in the serial case, but reports details of the local domain decomposition:

```
$ mpirun -np 8 ./Ludwig.exe ./serial-auto-c01.inp
Welcome to Ludwig v0.1.26 (MPI version running on 8 processes)
...
System details
--------------
System size:  64 64 64
Decomposition: 2 2 2
Local domain: 32 32 32
...
```

The decomposition is controlled by the `grid` key in the input. Here, `grid 2_2_2` is consistent with the 8 MPI tasks specified on the command line, and the resulting local decomposition is 32 lattice points in each coordinate direction. If no grid is specified in the input, or a request is make for a grid which cannot be met (e.g., the product of the grid dimensions does not agree with the total number of MPI tasks available) the code will try to determine its own decomposition as best it can. If no valid parallel decomposition is available at all, the code will exit with a message to that effect.

Further details of various input key value pairs are given in relevant sections of the documentation.

### 2.2.4 Output

The standard output of the running code produces a number of aggregate quantities which allow a broad overview of the progress of the computation to be seen by the user. These include, where relevant, global statistics related to fluid density and momentum, the integrated free energy, particle-related quantities, and so on. The frequency of this information can be adjusted from the input file (see, e.g., `freq_statistics`).

Output for lattice-based quantities (for example, the velocity field $u(\mathbf{r};t)$) is via external file. This output is in serial or in parallel according to how the model is run, and may be in either ASCII or raw binary format as required. Output files are produced with time step encoded in the file, and an extension which describes the parallel output decomposition. Further, each quantity output in this way is accompanied by a metadata description with `meta` extension. For example, the two output files

```
bash-3.2$ ls dist*
dist-00000020.001-001 dist.001-001.meta
```

contain the LB distributions for time step 20 (the file is 1 of a total number of 1 in parallel), and the plain text metadata description, respectively.

To ensure output for based-based quantities is in the correct order for analysis, post-processing may be required (always if the code is run in parallel). This uses a utility provided for the purpose which required both the data and the metadata description to recombine the parallel output. This utility is described ELSEWHERE.

Output for colloidal particle data is again to file with a name encoding the time step and parallel docomposition of the output. For example,

```
bash-3.2$ ls config*
config.cds00000020.001-001
```

contains particle data for time step 20 in a format described ELSEWHERE. These data may be requested in ASCII or raw binary format.

### 2.2.5 Errors and run time failures

The code attempts to provide meaningful diagnostic error messages for common problems. Such errors include missing or incorrectly formatted input files, inconsistent input values, and unacceptable feature combinations. The code should also detect other run time problems such as insufficient memory and errors writing output files. These errors will result in termination.

Instability in the computation will often be manifested by numerically absurd values in the statistical output (ultimately `NaN` in many cases). Instability may or may not result in termination, depending on the problem. Such instability is very often related to poor parameter choices, of which there can be many combinations. Check the relevant section of the documentation for advice on reasonable starting parameters for different problems.

## 2.3 Note on Units

All computation in *Ludwig* is undertaken in "lattice units," fundamentally related to the underlying LB fluid model which expects discrete model space and time steps $\Delta x = \Delta t = 1$. The natural way to approach problems is then to ensure that appropriate dimensionless quantities are reasonable. However, "reasonable" may not equate to "matching experimental values". For example, typical flows in colloidal suspensions my exhibit Reynolds numbers as low as $O(10^{-6})$ to $O(10^{-8})$. Matching such a value in a computation may imply an impractically small time step; a solution is to let the Reynolds number rise artificially with the constraint that it remains small compared to $O(1)$. Further discussion of the issue of units is provided in, e.g., [7]. Again, consult the relevant section of the documentation for comments on specific problems.

# 3 General Information for Users and Developers

This section contains information on the design of the code, along with details of compilation an testing procedures which may be of interest to both users and developers. *Ludwig* is named for Ludwig Boltzmann (1844–1906) to reflect its background in lattice Boltzmann hydrodynamics.

## 3.1 Manifest

The top level ludwig directory should contain at least the following:

```
$ ls
bash-3.2$ ls
config LICENSE    mpi_s  src      tests version.h
docs   Makefile.mk README targetDP util
```

The code is released under a standard BSD license; the current version number is found in `version.h`. The main source is found in the `src` directory, with other relevant code in `tests` and `util`. The `targetDP` directory holds code related to the targetDP abstraction layer [17]. These documents are found in `docs`.

## 3.2 Code Overview

The code is ANSI C (1999), and can be built without any external dependencies, with the exception of the message passing interface (MPI), which is used to provide domain-decomposition based parallelism. Note that the code will also compile under NVIDIA `nvcc`, i.e., it also meets the C++ standard.

### 3.2.1 Design

The *Ludwig* code has evolved and expanded over a number of years to its present state. Its original purpose was to investigate specifically spinodal decomposition in binary fluids (see, e.g., [23]). This work used a free-energy based formulation combined with a two-distribution approach to the binary fluid problem (with lattice Boltzmann model D3Q15 at that time). This approach is retained today, albeit in a somewhat updated form. The wetting problem for binary fluid at solid surfaces has also been of consistent interest. The code has always been developed with parallel computing in mind, and has been run on a large number of different parallel machines including the Cray T3E at Edinburgh. These features, developed by J.-C. Desplat and others, were reflected in the early descriptive publication in *Comp. Phys. Comm* in 2001 [11].

The expansion of the code to include a number of additional features has occasioned significant alterations over time, and little of the original code remains. However, the fundamental idea that the code should essentially operate for high performance computers and be implemented using ANSI C with message passing via MPI has remained unchanged.

The code has a number of basic building blocks which are encapsulated in individual files.

### 3.2.2 Parallel environment

The code is designed around message passing using MPI. A stub MPI library is provided for platforms where a real MPI is not available (an increasingly rare occurrence), in

which case the code runs in serial. The parallel environment (interface defined in `pe.h`) therefore underpins the entire code and provides basic MPI infrastructure such as `info()`, which provides `printf()` functionality at the root process only. The parallel environment also provides information on version number etc. MPI parallelism is implemented via standard domain decomposition based on the computational lattice (discussed further in the following section on the coordinate system).

### 3.2.3 targetDP: threaded parallelism and vector parallelism

To allow various threaded models to be included without duplicating source code, we have developed a lightweight abstraction of the thread level parallelism, This currently supports a standard single-threaded model, OpenMP, or CUDA. targetDP ("target data parallel") allows single kernels to be written which can then be compiled for the different threaded models which may be appropriate on different platforms. targetDP can also be used as a convenient way to express vector parallelism (typically SSE or AVX depending on processor architecture). targetDP allows explicit specification of the vector length at compile time.

targetDP does not automatically identify parallelism; this must still be added appropriately by the developer. It is merely a concise way to include different threaded models. It is only available in the development version.

### 3.2.4 Coordinate system

It is important to understand the coordinate system used in the computation. This is fundamentally a regular, 3-dimensional Cartesian coordinate system . (Even if a D2Q9 LB model is employed, the code is still fundamentally 3-dimensional, so it is perhaps not as efficient for 2-dimensional problems as it might be.)

The coordinate system is centred around the (LB) lattice, with lattice spacings $\Delta x = \Delta y = \Delta z = 1$. We will refer, in general, to the lattice spacing as $\Delta x$ throughout, its generalisation to three dimensions $x, y, z$ being understood. Lattice sites in the $x-$direction therefore have unit spacing and are located at $x = 1, 2, \ldots, N_x$. The length of the system $L_x = N_x$, with the limits of the computational domain begin $x = 1/2$ and $x = L_x + 1/2$. This allows us to specify a unit control volume centred on each lattice site $x_i$ being $i-1/2$ to $i + 1/2$. This will become particularly significant for finite difference (finite volume) approaches discussed later.

Information on the coordinate system, system size and so on is encapsulated in `coords.c`, which also deals with the regular domain decomposition in parallel. Decompositions may be explicitly requested by the user in the input. or computed by the code itself at run time. `coords.h` also provides basic functionality for message passing within a standard MPI Cartesian communicator, specification of periodic boundary conditions, and so on.

Three-dimensional fields are typically stored on the lattice, but are addressed in compressed one-dimensional format. This avoids use of multidimensional arrays in C. This addressing must take account of the width of the halo region required at the edge of each sub-domain required for exchanging information in the domain decomposition. The extent of the halo region varies depending on the application required, and is selected automatically at run time.

### 3.2.5 Lattice Boltzmann hydrodynamics

The hydrodynamic core of the calculation is supplied by the lattice Boltzmann method, which was central at the time of first development. LB is also the basis for hydrodynamic solid-fluid interactions at stationary walls and for moving spherical colloids. The LB approach is described in more detail in Section 5. For general and historical references, the interested reader should consider, e.g., Succi [37].

### 3.2.6 Free energy

For complex fluids, hydrodynamics is augmented by the addition of a free energy for the problem at hand, expressed in terms of an appropriate order parameter. The order parameter may be a three dimensional field, vector field, or tensor field. For each problem type, appropriate tome evolution of the order parameter is supplied.

Coupling between the thermodynamic sector and the hydrodynamics is abstracted so that the hydrodynamic core does not require alteration for the different free energies. This is implemented via a series of call back functions in the free energy which are set appropriately at run time to correspond to that specified by the user in the input.

Different (bulk) fluid free energy choices are complemented by appropriate surface free energy contributions which typically alter the computation of order parameter gradients at solid surface. Specific gradient routines for the calculation of order parameter gradients may be selected or added by the user or developer.

Currently available free energies are:

- Symmetric binary fluid with scalar compositional order parameter $\phi(\mathbf{r})$ and related Cahn-Hilliard equation;
- Brazovskii smectics, again which scalar compositional order parameter $\phi(\mathbf{r})$;
- Polar (active) gels with vector order parameter $P_\alpha(\mathbf{r})$ and related Leslie-Erikson equation;
- Landau-de Gennes liquid crystal free energy with tensor orientational order parameter $Q_{\alpha\beta}(\mathbf{r})$, extended to apolar active fluids and related Beris-Edwards equation;
- a free energy appropriate for electrokinetics for charged fluids and related Nernst-Planck equations (also requiring the solution of the Poisson equation for the potential);
- a coupled electrokinetic binary fluid model;
- a liquid crystal emulsion free energy which couples a binary composition to the liquid crystal.

See the relevant sections on each free energy for further details.

## 3.3 Compilation

Compilation of the main code is controlled by the `config.mk` in the top-level directory. A number of example `config.mk` files are provided in the `config` directory (which can be copied and adjusted as needed). This section discusses a number of issues which influence compilation.

### 3.3.1 Dependencies on third-party software:

There is the option to use PETSC to solve the Poisson equation required in the electrokinetic problem. A rather less efficient in-built method can be used if PETSC is not available. We suggest using PETSC v3.4 or later available from Argonne National Laboratory `http://www.mcs.anl.gov/petsc/`.

The tests use the lightweight implementation of exceptions provided under the GPL Lesser General Public License by Guillermo Calvo. This is included with the source.

### 3.3.2 Makefile

The `Makefile.mk` and `config.mk` files in the top level directory control options for compilation. Before compilation, the `config.mk` file must be edited provide details of the local C compiler(s). The relevant lines are usually limited to, e.g.:

```
CC = cc
MPICC = mpicc
CFLAGS = -O2
```

where `CC` is the compiler used for serial compilation, `MPICC` is the compiler used for compilation against MPI, and `CFLAGS` provides compiler switches (often related to optimisation).

The `Makefile` provided in each subdirectory includes the `config.mk` file via the top level `Makefile.mk`. Changes to the individual Makefiles should therefore not be required.

If GPU compilation is required, the `config.mk` file should specify the local details concerning `nvcc`. This includes explicit information on the location of MPI headers and libraries if an MPI-parallel GPU version is required.

### 3.3.3 C assertions

The code makes quite a lot of use of standard C assertions, which are useful to prevent errors. They do result in a considerably slower execution in some instances, so production runs should switch off the assertions with the standard `NDEBUG` preprocessor flag. Add `-DNDEBUG` to `CFLAGS` in the `config.mk`.

### 3.3.4 Targets for serial and parallel code

The code can be compiled with or without a true MPI library. For serial execution, the MPI stub library must be compiled first (see "Quick Start for Users" Section 2.1.2). The appropriate target is:

```
bash-3.2$ make serial
make serial-d3q19
make serial-model ‘‘LB=-D_D3Q19_’’ ‘‘LBOBJ=d3q19.o’’
...
```

from which it will be seen that the default LB model is D3Q19. This is the same for the true MPI target

```
bash-3.2$ make mpi
make mpi-d3q19
make mpi-model ‘‘LB=-D_D3Q19_’’ ‘‘LBOBJ=d3q19.o’’
...
```

### 3.3.5 Targets for different LB models

Specific targets are provided if an alternative LB model is wanted. The available options are:

```
make [ serial-d2q9 | serial-d3q15 | serial-d3q19 ]
make [ mpi-d2q9 | mpi-d3q15 | mpi-d3q19 ]
```

A further set of targets is supplied if 'reverse' or structure-of-array memory ordering is wanted (e.g., for GPU computing):

```
make [ serial-d2q9r | serial-d3q15r | serial-d3q19r ]
make [ mpi-d2q9r | mpi-d3q15r | mpi-d3q19r ]
```

This will only influence efficiency of the code: results are unchanged.

## 3.4 Tests

A series of tests are provided in the `./tests` directory. These are of two types. Unit tests are generally written when units of code are first introduced to ensure basic operation is error free. The unit tests are encoded in an executable linked against the appropriate *Ludwig* library. Regression tests are introduced and updated when physical results are (broadly) validated. The regression tests use the stand-alone executable for the appropriate model, and read input files which define the different tests. Both are run regularly as the basis of a nightly test procedure, but they can be run independently, e.g., after adding or changing code.

### 3.4.1 Running the tests

A number of different `Makefile` targets are provided for running serial, parallel, or GPU tests. For each test, there is the option to run either the relevant unit tests, or regression tests, or both. For example, to run both serial unit tests and serial regression tests for the D3Q19 model, invoke

```
$ cd tests
$ make compile-run-serial
```

in the test directory. The model selection follows the naming scheme described for compilation in the previous section.

## 3.5 Additional Notes for Developers

Developers who wish to contribute code to the SVN repository please consider the following pleas concerning standards.

### 3.5.1 Coding standards

While definitive statements on style are avoided, please try to maintain some basic standards: (0) code should be strictly ANSI C99 standard; (1) code should compile without warnings when appropriate compiler flags are set (e.g., `-Wall` under `gcc`); (2) avoid long lines of code for readability reasons; (3) avoid confusing commented-out code; (4) avoid conditional pre-processor directives if possible; (5) add meaningful and descriptive comments; (6) use standard `assert()` to trap programming errors; (7) explicitly trap possible run time errors (8) add appropriate tests.

### 3.5.2 Documentation standards

Additions and alterations to the code need to be reflected in the documentation. These should be checked in at the same time as the code itself.

### 3.5.3 Protocol

Before checking in code, please follow procedure: (1) increment the patch version number in `version.h` consistent with the SVN (if a minor version increment is required, communication with other developers must be considered); (2) run at least the unit tests and the short regression tests; (3) add a note to the change log; and (4) SVN update and commit.

# 4 Further Information for Developers

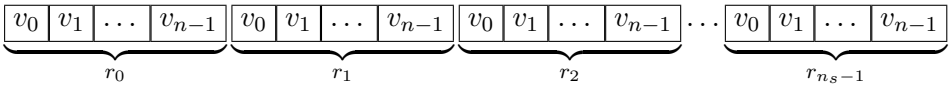
## 4.1 Address models for 3-dimensional fields

The code allows for different order of storage associated with 3-dimensional fields (scalars, vectors, and so on). For historical reasons these are referred to as 'Model' and 'Model R' options, which correspond to array-of-structures and structure-of-arrays layout, respectively. Early versions of the code for CPU were written to favour summation of LB distributions on a per lattice site basis in operations such as $\rho(\mathbf{r}) = \sum_i f_i(\mathbf{r})$. This is array-of-structures, where the $f_i$ are stored contiguously per site. Introduction of GPU versions, where memory coalescing favours the opposite memory order, were then referred to as 'Model R', the 'R' standing for 'reverse'.

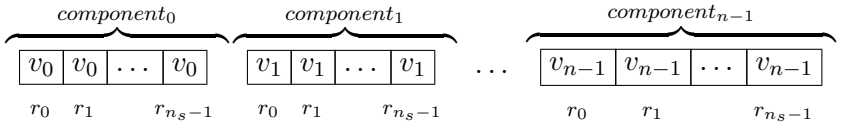
The memory layouts are discussed below. In all cases, a 3-d field occupies lattice sites with a unique spatial index determine by position, and computed via `coords_index()`. These position indices will be denoted $r_0, r_1, \ldots, r_{n_s-1}$ where $n_s$ is the total number of lattice sites (including halo points).

### 4.1.1 Rank 1 objects (to include scalar fields)

**ADDR_MODEL**: The array-of-structures or Model order for an $n-$vector field with components $v_\alpha = (v_0, v_1, \ldots, v_{n-1})$ is, schematically:



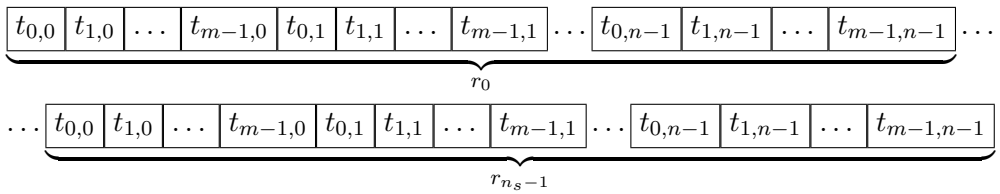

**ADDR_MODEL_R**: The Model R version is:




A scalar field has $n = 1$.

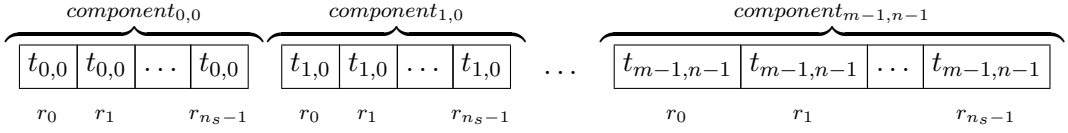
### 4.1.2 Rank 2 objects (to include dyadic tensor fields)

**ADDR_MODEL**: A general rank 2 tensor $t_{\alpha\beta}$ with components $(t_{0,0}, \ldots, t_{m-1,n-1})$ is stored as:




Dyadic tensors, for example the gradient of a vector field $\partial_\alpha v_\beta$ in three dimensions, are stored in corresponding fashion with $m = 3$ and $\partial_\alpha = (\partial_x, \partial_y, \partial_z)$.

**ADDR_MODEL_R**: The Model R version is



### 4.1.3 Compressed rank 2 objects

A symmetric tensor $S_{\alpha\beta}$ in three dimensions has six independent components. It may be convenient to store this in compressed form as a rank 1 vector $(S_{xx}, S_{xy}, S_{xz}, S_{yy}, S_{yz}, S_{zz})$ to eliminate redundent storage.

A symmetric traceless rank 2 tensor — for example, the Landau-de Gennes liquid crystal order parameter $Q_{\alpha\beta}$ — has five independent components. This is stored as a rank 1 vector with five components $(Q_{xx}, Q_{xy}, Q_{xz}, Q_{yy}, Q_{yz})$ to eliminate redundant storage. API calls are provided to expand the compressed format to the full rank-2 representation $Q_{\alpha\beta}$ and, conversely, to compress the full representation to five components.

### 4.1.4 Rank 3 objects (to include triadic tensor fields)

The general rank 3 object $t_{\alpha\beta\gamma}$ with components $(t_{0,0,0}, \ldots, t_{m-1,n-1,p-1})$ is stored in a manner which generalises from the above, i.e., with the rightmost index running fastest. Diagrams are omitted, but Model and Model R storage patterns follow the same form as seen above.

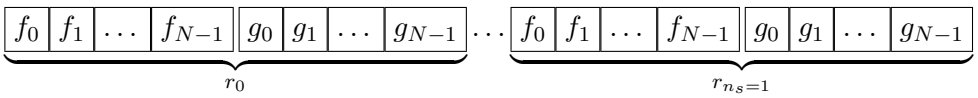
A triadic tensor, for example the general second derivative of a vector field $\partial_\alpha \partial_\beta v_\gamma$ may be stored as a rank 3 object.

### 4.1.5 Compressed rank 3 objects

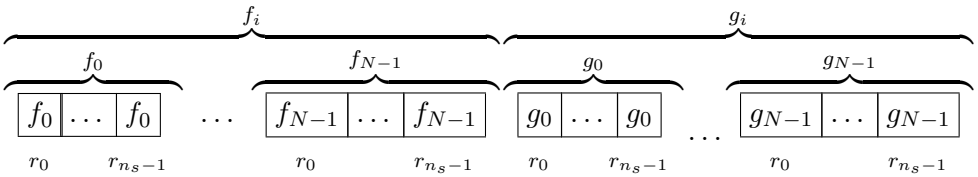
Symmetry properties may be used to reduce the storage requirement associated with rank 3 tensors. For example, the gradient of the liquid crystal order parameter $\partial_\gamma Q_{\alpha\beta}$ may be stored as a rank 2 object. The exact requirement will depend on the construction of the tensor.

### 4.1.6 LB distribution data

**ADDR_MODEL**: For the LB distributions, up to two distinct distributions can be accommodated to allow for a binary fluid implementation, although the usual situation is to have only one. The Model order for two $N$-velocity distributions $f$ and $g$ is:




**ADDR_MODEL_R**: The Model R order is:




For the single-distribution case, this is equivalent to the rank 1 vector field with $n = N$.

## 4.2 Generalised model for SIMD vectorisation

To promote parallelism at the instruction level, it is necessary to insure the innermost loop of any loop construct has an extent which is the vector length for the target architecture. The memory layout should therefore be adjusted accordingly. This means the MODEL

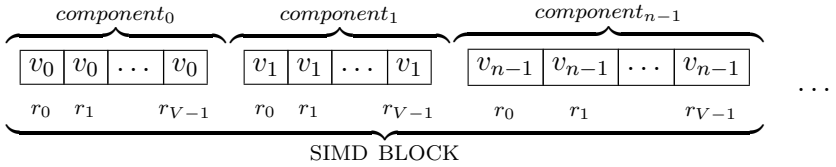
format is generalised to a (rather clumsily named) array of structures of short vectors. The length of the short vectors is the SIMD vector length.

The outermost loop in this context is always to be the loop over lattice sites, which is adjusted according to the vector length; see Section 4.3 below for practical examples.

The Model R picture, which targets coalescene, can be viewed as naturally supporting SIMD vectorisation by virtue of allowing contiguous access to individual quantities as a function of lattice index. (If SIMD vectorisation is wanted at all on GPU architecture, it can be as a means to releive register pressure.) Model R therefore remains unaltered and we concentrate on the Model picture.

### 4.2.1 Rank 1 objects

**VADDR_MODEL**: The structure of short vectors is based on the SIMD vector length which we here denote $V$:




Subsequent SIMD blocks involve lattice sites $r_V \dots r_{2V-1}$, $r_{2V} \dots r_{3V-1}$, and so on. If the SIMD vector length is unity, this collapses to the standard Model picture shown in Section 4.1.1.

The generalisation of this approach to rank 2 and rank 3 objects follows the corresponding Model implementation.

## 4.3 Addressing: How to?

To provide a transparent interface for addressing vector fields, a single API is provided which implements the addressing order selected at compile time. This interface is always based on a fixed order of indices which may include the vector index of the innermost SIMD loop if present. This allows both vectorised and non-vectorised loops to be constructed in a consistent manner.

The usual model for constructing a loop involving all lattice sites (in this case without haloes) would be, in the absence of vectorisation:

```
for (ic = 1; ic <= nlocal[X]; ic++) {
  for (jc = 1; jc <= nlocal[Y]; jc++) {
    for (kc = 1; kc <= nlocal[Z]; kc++) {

      index = coords_index(ic, jc, kc);
      /* Perform operation per lattice site ... */
    }
  }
}
```

where final array indexing is based on the coordinate index and is specific to the object at hand. To allow transparent switching between Model and Model R versions, the indexing must be abstracted. As a concrete example, the following considers a rank 1 3-vector:

```
for (ic = 1; ic <= nlocal[X]; ic++) {
```

```
  for (jc = 1; jc <= nlocal[Y]; jc++) {
    for (kc = 1; kc <= nlocal[Z]; kc++) {

      index = coords_index(ic, jc, kc);
      for (ia = 0; ia < 3; ia++) {
        iref = addr_rank1(nsites, 3, index, ia);
        array[iref] = ...;
      }
    }
  }
}
```

Here, the function `addr_rank1()` performs the appropriate arithmetic to reference the correct 1-d array element in either address model. We note that this can be implemented to operate appropriately when the SIMD vector length is an arbitrary number.

An equivalent vectorised version may be constructed as follows:

```
for (n = 0; n < nsites; n += SIMDVL) {
  index = coords_indexv(n, nextra);   /* Proposal */
  for (ia = 0; ia < 3; ia++) {
    for (iv = 0; iv < SIMDVL; iv++) {
      iref = vaddr_rank1(nsites, 3, index, ia, iv);
      array[iref] = ...;
    }
  }
}
```

Note that a different function `vaddr_rank1()` with an additional argument which is the index of the vector loop is used. Note also that vectorisaed versions must take into account the extent that the kernel extends into the halo region (here `nextra`), which involves logic to avoid lattice sites which are not required (not shown in this example).

Note that, in practice, the loop over lattice sites is further abstracted in the code to accommodate thread-level parallelism via targetDP interface.

# 5 Lattice Boltzmann Hydrodynamics

We review here the lattice Boltzmann method applied to a simple Newtonian fluid with particular emphsis on the relevant implementation in *Ludwig*.

## 5.1 The Navier Stokes Equation

We seek to solve the isothermal Navier-Stokes equations which, often written in vector form, express mass conservation

$$\partial_t \rho + \boldsymbol{\nabla}.(\rho \mathbf{u}) = 0 \tag{1}$$

and the conservation of momentum

$$\partial_t (\rho \mathbf{u}) + \boldsymbol{\nabla}.(\rho \mathbf{u}\mathbf{u}) = -\boldsymbol{\nabla} p + \eta \nabla^2 \mathbf{u} + \zeta \boldsymbol{\nabla}(\boldsymbol{\nabla}.\mathbf{u}). \tag{2}$$

Equation 1 expresses the local rate of change of the density $\rho(\mathbf{r};t)$ as the divergence of the flux of mass associated with the velocity field $\mathbf{u}(\mathbf{r};t)$. Equation 2 expresses Newton's second law for momentum, where the terms on the right hand side represent the force on the fluid.

For this work, it is more convenient to rewrite these equations in tensor notation, where Cartesian coordinates $x, y, z$ are represented by indices $\alpha$ and $\beta$, viz

$$\partial_t \rho + \nabla_\alpha (\rho u_\alpha) = 0 \tag{3}$$

and

$$\partial_t (\rho u_\alpha) + \nabla_\beta (\rho u_\alpha u_\beta) = -\nabla_\alpha p + \eta \nabla_\beta (u_\alpha \nabla_\beta + \nabla_\alpha u_\beta) + \zeta \nabla_\alpha (\nabla_\gamma u_\gamma). \tag{4}$$

Here, repeated Greek indices are understood to be summed over. The conservation law is seem better if the forcing terms of the right hand side are combined in the fluid stress $\Pi_{\alpha\beta}$ so that

$$\partial_t (\rho u_\alpha) + \nabla_\beta \Pi_{\alpha\beta} = 0. \tag{5}$$

In this case the expanded expression for the stress tensor is

$$\Pi_{\alpha\beta} = p\delta_{\alpha\beta} + \rho u_\alpha u_\beta + \eta \nabla_\alpha v_\beta + \zeta (\nabla_\gamma v_\gamma)\delta_{\alpha\beta} \tag{6}$$

where $\delta_{\alpha\beta}$ is that of Kroneker. Th Navier-Stokes equations in three dimensions have 10 degrees of freedom (or hydrodynamic modes) being $\rho$, three components of the mass flux $\rho u_\alpha$, and 6 independent modes from the (symmetric) stress tensor $\Pi_{\alpha\beta}$.

## 5.2 The Lattice Boltzmann Equation

The Navier-Stokes equation may be approximated in a discrete system by the lattice Boltzmann equation (LBE). A discrete density distribution function $f_i(\mathbf{r};t)$ at lattice points $\mathbf{r}$ and time $t$ evolves according to

$$f_i(\mathbf{r} + \mathbf{c}_i \Delta t; t + \Delta t) = f_i(\mathbf{r};t) + \sum_j \mathcal{L}_{ij}\big(f_i(\mathbf{r};t) - f_i^{\mathrm{eq}}(\mathbf{r};t)\big) \tag{7}$$

where $\mathbf{c}_i$ is the discrete velocity basis and $\Delta t$ is the discrete time step. The collision operator $\mathcal{L}_{ij}$ provides the mechanism to compute a discrete update from the non-equilibrium distribution $f_i(\mathbf{r};t) - f_i^{\mathrm{eq}}(\mathbf{r};t)$. Additional terms may be added to this equation to represent external body forces, thermal fluctuations, and so on. These additional terms are discussed in the following sections.

### 5.2.1 The distribution function and its moments

In lattice Boltzmann, the density and velocity of the continuum fluid are complemented by the distribution function $f_i(\mathbf{r};t)$ defined with reference to the discrete velocity space $c_{i\alpha}$. It is possible to relate the hydrodynamic quantities to the distribution function via its moments, that is

$$\rho(\mathbf{r};t) = \sum_i f_i(\mathbf{r};t), \quad \rho u_\alpha(\mathbf{r};t) = \sum_i f_i(\mathbf{r};t) c_{i\alpha}, \quad \Pi_{\alpha\beta}(\mathbf{r};t) = \sum_i f_i(\mathbf{r};t) c_{i\alpha} c_{i\beta}. \tag{8}$$

Here, the index of the summation is over the number of discrete velocities used as the basis, a number which will be denoted $N_{\text{vel}}$. For example, in three dimensions $N_{\text{vel}}$ is often 19 and the basis is referred to as D3Q19.

The number of moments, or modes, supported by a velocity set is exactly $N_{\text{vel}}$, and these can be written in general as

$$M^a(\mathbf{r};t) = \sum_i m_i^a f_i(\mathbf{r};t), \tag{9}$$

where the $m_i$ are the eigenvectors of the collision matrix in the LBE. For example, in the case of the density, all the $m_i^a = 1$ and the mode $M^a$ is the density $\rho = \sum_i f_i$. Note that the number of modes supported by a given basis will generally excceed the number of hydrodynamic modes; the excess modes have no direct physical interpretation and are variously referred to as non-hydrodynamic, kinetic, or ghost, modes. The ghost modes take no part in bulk hydrodynamics, but may become important in other contexts, such as thermal fluctuations and near boundaries. The distribution function can be related to the modes $M^a(\mathbf{r};t)$ via

$$f_i(\mathbf{r};t) = w_i \sum_a m_i^a N^a M^a(\mathbf{r};t). \tag{10}$$

In this equation, $w_i$ are the standard LB weights appearing in the equilibrium distribution function, while the $N^a$ are a per-mode normalising factor uniquely determined by the orthogonality condition

$$N^a \sum_i w_i m_i^a m_i^b = \delta_{ab}. \tag{11}$$

Writing the basis this way has the advantage that the equilibrium distribution projects directly into the hydrodynamic modes only. Putting it another way, we may write

$$f_i^{\text{eq}} = w_i\big(\rho + \rho c_{i\alpha} u_\alpha / c_s^2 + (c_{i\alpha}c_{i\beta} - c_s^2\delta_{\alpha\beta})(\Pi_{\alpha\beta}^{\text{eq}} - p\delta_{\alpha\beta})/2c_s^4\big) \tag{12}$$

where only (equilibrium) hydrodynamic quantities appear on the right hand side.

### 5.2.2 Collision and relaxation times

## 5.3 Model Basis Descriptions

### 5.3.1 D2Q9

The D2Q9 model in two dimensions consists one zero vector (0,0), four vectors of length unity being $(\pm 1, 0)$ and $(0, \pm 1)$, and four vectors of length $\sqrt{2}$ being $(\pm 1, \pm 1)$. The eigenvectors of the collision matrix, with associated weights and normalisers are shown in Table 1. In two dimensions there are six hydrodynamic modes and a total of three kinetic modes, or ghost modes.

| $M^a$ | $p$ | $m_i^a$ | | | | | | | | | $N^a$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\rho$ | - | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | **1** |
| $\rho c_{ix}$ | - | 0 | 1 | 1 | 1 | 0 | 0 | -1 | 1 | -1 | 3 | $c_{ix}$ |
| $\rho c_{iy}$ | - | 0 | 1 | 0 | -1 | 1 | -1 | 1 | 0 | -1 | 3 | $c_{iy}$ |
| $Q_{xx}$ | 1/3 | -1 | 2 | 2 | 2 | -1 | -1 | 2 | 2 | 2 | 9/2 | $c_{ix}c_{ix} - c_s^2$ |
| $Q_{xy}$ | - | 0 | 1 | 0 | -1 | 0 | 0 | -1 | 0 | 1 | 9 | $c_{ix}c_{iy}$ |
| $Q_{yy}$ | 1/3 | -1 | 2 | -1 | 2 | 2 | 2 | 2 | -1 | 2 | 9/2 | $c_{iy}c_{iy} - c_s^2$ |
| $\chi^1$ | - | 1 | 4 | -2 | 4 | -2 | -2 | 4 | -2 | 4 | 1/4 | $\chi^1$ |
| $J_{ix}$ | - | 0 | 4 | -2 | 4 | 0 | 0 | -4 | -2 | -4 | 3/8 | $\chi^1 \rho c_{ix}$ |
| $J_{iy}$ | - | 0 | 4 | 0 | -4 | -2 | 2 | 4 | 0 | -4 | 3/8 | $\chi^1 \rho c_{iy}$ |
| $w_i$ | 1/36 | 16 | 1 | 4 | 1 | 4 | 4 | 1 | 4 | 1 | | $w_i$ |

Table 1: Table showing the details of the basis used for the D2Q9 model in two dimensions. The nine modes $M^a$ include six hydrodynamic modes, one scalar kinetic mode $\chi^1$, and one vector kinetic mode $J_{i\alpha}$. The weights in the equilibrium distribution function are $w_i$ and the normaliser for each mode is $N^a$. The eigenvectors of the collision matrix are the columns of the transformation matrix $m_i^a$. The prefactor $p$ (where present) multiplies all the elements to the right in that row.

#### 5.3.2 D3Q15

The D3Q15 model in three dimensions consists of a set of vectors: one zero vector $(0, 0, 0)$, six vectors of length unity being $(\pm 1, 0, 0)$ cyclically permuted, and 8 vectors of length $\sqrt{3}$ being $(\pm 1, \pm 1, \pm 1)$. The eigenvalues and eigenvectors of the collision matrix used for D3Q15 are given in Table 2.

#### 5.3.3 D3Q19

The D3Q19 model in three dimensions is constructed with velocities: one zero vector $(0, 0, 0)$, three vectors of length unity being $(\pm 1, 0, 0)$ cyclically permuted, and twelve vectors of length $\sqrt{2}$ being $(\pm 1, \pm 1, 0)$ cyclically permuted. The details of the D3Q19 model are set out in Table 3.

### 5.4 Fluctuating LBE

It is possible [3] to simulate fluctuating hydrodynamics for an isothermal fluid via the inclusion of a fluctuating stress $\sigma_{\alpha\beta}$:

$$\Pi_{\alpha\beta} = p\delta_{\alpha\beta} + \rho u_\alpha u_\beta + \eta_{\alpha\beta\gamma\delta}\nabla_\gamma u_\delta + \sigma_{\alpha\beta}. \tag{13}$$

The fluctuation-dissipation theorem relates the magnitude of this random stress to the isothermal temperature and the viscosity.

In the LBE, this translates to the addition of a random contribution $\xi_i$ to the distribution at the collision stage, so that

$$\ldots + \xi_i. \tag{14}$$

For the conserved modes $\xi_i = 0$. For all the non-conserved modes, i.e., those with dissipation, the fluctuating part may be written

$$\xi_i(\mathbf{r}; t) = w_i m_i^a \hat{\xi}^a(\mathbf{r}; t) N^a \tag{15}$$

| $M^a$ | $p$ | $m_i^a$ | | | | | | | | | | | | | | | $N^a$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\rho$ | - | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | **1** |
| $\rho c_{ix}$ | - | 0 | 1 | -1 | 0 | 0 | 0 | 0 | 1 | -1 | 1 | -1 | 1 | -1 | 1 | -1 | 3 | $c_{ix}$ |
| $\rho c_{iy}$ | - | 0 | 0 | 0 | 1 | -1 | 0 | 0 | 1 | 1 | -1 | -1 | 1 | 1 | -1 | -1 | 3 | $c_{iy}$ |
| $\rho c_{iz}$ | - | 0 | 0 | 0 | 0 | 0 | 1 | -1 | 1 | 1 | 1 | 1 | -1 | -1 | -1 | -1 | 3 | $c_{iz}$ |
| $Q_{xx}$ | 1/3 | -1 | 2 | 2 | -1 | -1 | -1 | -1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 9/2 | $c_{ix}c_{ix} - c_s^2$ |
| $Q_{yy}$ | 1/3 | -1 | -1 | -1 | 2 | 2 | -1 | -1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 9/2 | $c_{iy}c_{iy} - c_s^2$ |
| $Q_{zz}$ | 1/3 | -1 | -1 | -1 | -1 | -1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 9/2 | $c_{iz}c_{iz} - c_s^2$ |
| $Q_{xy}$ | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -1 | -1 | 1 | 1 | -1 | -1 | 1 | 9 | $c_{ix}c_{iy}$ |
| $Q_{yz}$ | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | -1 | -1 | -1 | -1 | 1 | 1 | 9 | $c_{iy}c_{iz}$ |
| $Q_{zx}$ | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -1 | 1 | -1 | -1 | 1 | -1 | 1 | 9 | $c_{iz}c_{ix}$ |
| $\chi^1$ | - | -2 | 1 | 1 | 1 | 1 | 1 | 1 | -2 | -2 | -2 | -2 | -2 | -2 | -2 | -2 | 1/2 | $\chi^1$ |
| $J_{ix}$ | - | 0 | 1 | -1 | 0 | 0 | 0 | 0 | -2 | 2 | -2 | 2 | -2 | 2 | -2 | 2 | 3/2 | $\chi^1 \rho c_{ix}$ |
| $J_{iy}$ | - | 0 | 0 | 0 | 1 | -1 | 0 | 0 | -2 | -2 | 2 | 2 | -2 | -2 | 2 | 2 | 3/2 | $\chi^1 \rho c_{iy}$ |
| $J_{iz}$ | - | 0 | 0 | 0 | 0 | 0 | 1 | -1 | -2 | -2 | -2 | -2 | 2 | 2 | 2 | 2 | 3/2 | $\chi^1 \rho c_{iz}$ |
| $\chi^3$ | - | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -1 | -1 | 1 | -1 | 1 | 1 | -1 | 9 | $c_{ix}c_{iy}c_{iz}$ |
| $w_i$ | 1/72 | 16 | 8 | 8 | 8 | 8 | 8 | 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | $w_i$ |

Table 2: Table showing the details of the basis used for the D3Q15 model in three dimensions. The fifteen modes $M^a$ include two scalar kinetic modes $\chi^1$ and $\chi^3$, and one vector kinetic mode $J_{i\alpha}$. The weights in the equilibrium distribution are $w_i$ and the normaliser for each mode is $N^a$. The eigenvectors of the collision matrix are the columns of the transformation matrix $m_i^a$. The prefactor $p$ simply multiplies all elements of $m_i^a$ in that row as a convenience.

where $\hat{\xi}^a$ is a noise termwhich has a variance determined by the relaxation time for given mode

$$\left\langle \hat{\xi}^a \hat{\xi}^b \right\rangle = \frac{\tau_a + \tau_b + 1}{\tau_a \tau_b} \left\langle \delta M^a \delta M^b \right\rangle . \tag{16}$$

#### 5.4.1 Fluctuating stress

For the stress, the random contribution to the distributions is

$$\xi_i = w_i \frac{Q_{i\alpha\beta} \hat{\sigma}_{\alpha\beta}}{4c_s^2} \tag{17}$$

where $\hat{\sigma}_{\alpha\beta}$ is a symmtric matrix of random variates drawn from a Gaussian distribution with variance given by equation 16. In the case that the shear and bulk viscosities are the same, i.e., there is a single relaxation time, then the variances of the six independent components of the matrix are given by

$$\langle \hat{\sigma}_{\alpha\beta} \hat{\sigma}_{\mu\nu} \rangle = \frac{2\tau + 1}{\tau^2} (\delta_{\alpha\mu}\delta_{\beta\nu} + \delta_{\alpha\nu}\delta_{\beta\mu}). \tag{18}$$

### 5.5 Hydrodynamic Boundary Conditions

#### 5.5.1 Bounce-Back on Links

A very general method for the representation of solid objects within the LB approach was put forward by Ladd [24, 25]. Solid objects (of any shape) are defined by a boundary

| $M^a$ | $m_i^a$ | | | | | | | | | | | | | | | | | | | $N^a$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\rho$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $\rho c_{ix}$ | 0 | 1 | -1 | 0 | 0 | 0 | 0 | 1 | 1 | -1 | -1 | 1 | 1 | -1 | -1 | 0 | 0 | 0 | 0 | 3 |
| $\rho c_{iy}$ | 0 | 0 | 0 | 1 | -1 | 0 | 0 | 1 | -1 | 1 | -1 | 0 | 0 | 0 | 0 | 1 | 1 | -1 | -1 | 3 |
| $\rho c_{iz}$ | 0 | 0 | 0 | 0 | 0 | 1 | -1 | 0 | 0 | 0 | 0 | 1 | -1 | 1 | -1 | 1 | -1 | 1 | -1 | 3 |
| $Q_{ixx}$ | -1 | 2 | 2 | -1 | -1 | -1 | -1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | -1 | -1 | -1 | -1 | 9/2 |
| $Q_{iyy}$ | -1 | -1 | -1 | 2 | 2 | -1 | -1 | 2 | 2 | 2 | 2 | -1 | -1 | -1 | -1 | 2 | 2 | 2 | 2 | 9/2 |
| $Q_{izz}$ | -1 | -1 | -1 | -1 | -1 | 2 | 2 | -1 | -1 | -1 | -1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 9/2 |
| $Q_{ixy}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -1 | -1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9 |
| $Q_{ixz}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -1 | -1 | 1 | 0 | 0 | 0 | 0 | 9 |
| $Q_{iyz}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | -1 | -1 | 1 | 9 |
| $\chi^1$ | 0 | 1 | 1 | 1 | 1 | -2 | -2 | -2 | -2 | -2 | -2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3/4 |
| $\chi^1 \rho c_{ix}$ | 0 | 1 | -1 | 0 | 0 | 0 | 0 | -2 | -2 | 2 | 2 | 1 | 1 | -1 | -1 | 0 | 0 | 0 | 0 | 3/2 |
| $\chi^1 \rho c_{iy}$ | 0 | 0 | 0 | 1 | -1 | 0 | 0 | -2 | 2 | -2 | 2 | 0 | 0 | 0 | 0 | 1 | 1 | -1 | -1 | 3/2 |
| $\chi^1 \rho c_{iz}$ | 0 | 0 | 0 | 0 | 0 | -2 | 2 | 0 | 0 | 0 | 0 | 1 | -1 | 1 | -1 | 1 | -1 | 1 | -1 | 3/2 |
| $\chi^2$ | 0 | 1 | 1 | -1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | -1 | -1 | -1 | 1 | 1 | 1 | 1 | 9/4 |
| $\chi^2 \rho c_{ix}$ | 0 | 1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | -1 | 1 | 1 | 0 | 0 | 0 | 0 | 9/2 |
| $\chi^2 \rho c_{iy}$ | 0 | 0 | 0 | -1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | -1 | -1 | 9/2 |
| $\chi^2 \rho c_{iz}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | 1 | -1 | 1 | 1 | -1 | 1 | -1 | 9/2 |
| $\chi^3$ | 1 | -2 | -2 | -2 | -2 | -2 | -2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1/2 |
| $w_i$ | 12 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |

Table 3: Table showing the details of the basis used for the D3Q19 model in three dimensions. The nineteen modes $M^a$ include ten hydrodynamic modes, three scalr kinetic modes $\chi^1$, $\chi^2$, and $\chi^3$; there are also two vector kinetic modes $\chi^1 \rho c_{i\alpha}$ and $\chi^2 \rho c_{i\alpha}$. The weights in the equilibrium distribution function are $w_i$, and the normaliser for each mode is $N^a$. The eigenvectors of the collision matrix are the columns of the transformation matrix $m_i^a$.

surface which intersects some of the velocity vectors $\mathbf{c}_i$ joining lattice nodes. Sites inside are designated solid, while sites outside remain fluid. The correct boundary condition is defined by identifying *links* between fluid and solid sites, which allows those elements of the distribution which would cross the boundary at the propagation step to be "bounced-back" into the fluid. This bounce-back on links is an efficient method to obtain the correct hydrodynamic interaction between solid and fluid.

### 5.5.2 Fixed objects

### 5.5.3 Moving objects

# 6 The Input File

## 6.1 General

By default, the run time expects to find user input in a file `input` in the current working directory. If a different file name is required, its name should be provided as the sole command line argument, e.g.,

```
./Ludwig.exe input_file_name
```

If the input file is not located in the current working directory the code will terminate immediately with an error message.

When an input file is located, its content is read by a single MPI task, and its contents then broadcast to all MPI relevant tasks. The format of the file is plain ASCII text, and its contents are parsed on a line by line basis. Lines may contain the following:

- comments introduced by `#`.
- key value pairs separated by white space.

Blank lines are treated as comments. The behaviour of the code is determined by a set of key value pairs. Any given key may appear only once in the input file; unused key value pairs are not reported. If the key value pairs are not correctly formed, the code will terminate with an error message and indicate the offending input line.

Key value pairs may be present in the input file, but have no effect for any given run: they are merely ignored. Relevant control parameters for given input are reported in the standard output.

### 6.1.1 Key value pairs

Key value pairs are made up of a key — an alphanumeric string with no white space — and corresponding value following white space. Values may take on the follow forms:

```
key_string         value_string

key_integer_scalar 1
key_integer_vector 1_2_3

key_double_scalar  1.0
key_double_vector  1.0_2.0_3.0
```

Values which are strings should contain no white space. Scalar parameters may be integer values, or floating point values with a decimal point (scientific notation is also allowed). Vector parameters are introduced by a set of three values (to be interpreted as $x, y, z$ components of the vector in Cartesian coordinates) separated by an underscore. The identity of the key will specify what type of value is expected. Keys and (string) values are case sensitive.

Most keys have an associated default value which will be used if the key is not present. Some keys must be specified: an error will occur if they are missing. The remainder of this part of the guide details the various choices for key value pairs, along with any default values, and any relevant constraints.

## 6.2 The Free Energy

The choice of free energy is determined as follows:

```
free_energy  none
```

The default value is `none`, i.e., a simple Newtonian fluid is used. Possible values of the `free_energy` key are:

```
#  none                   Newtonian fluid [DEFAULT]
#  symmetric              Symmetric binary fluid (finite difference)
#  symmetric_lb           Symmetric binary fluid (two distributions)
#  brazovskii             Brazovskii smectics
#  surfactant             Surfactants
#  polar_active           Polar active gels
#  lc_blue_phase          Liquid crystal (nematics, cholesterics, BPs)
#  lc_droplet             Liquid crystal emulsions
#  fe_electro             Single fluid electrokinetics
#  fe_electro_symmetric   Binary fluid electrokinetics
```

The choice of free energy will control automatically a number of factors related to choice of order parameter, the degree of parallel communication required, and so on. Each free energy has a number of associated parameters discussed in the following sections.

Details of general (Newtonian) fluid parameters, such as viscosity, are discussed in Section 6.4.

### 6.2.1 Symmetric Binary Fluids

We recall that the free energy density is, as a function of compositional order $\phi$:

$$\tfrac{1}{2}A\phi^2 + \tfrac{1}{4}B\phi^4 + \tfrac{1}{2}\kappa(\nabla\phi)^2.$$

Parameters are introduced by (with default values):

```
free_energy symmetric
A           -0.0625                     # Default: -0.003125
B           +0.0625                     # Default: +0.003125
K           +0.04                       # Default: +0.002
```

Common usage has $A < 0$ and $B = -A$ so that $\phi^\star = \pm 1$. The parameter $\kappa$ (key `K`) controls the interfacial energy penalty and is usually positive.

### 6.2.2 Brazovskii smectics

The free energy density is:

$$\tfrac{1}{2}A\phi^2 + \tfrac{1}{4}B\phi^4 + \tfrac{1}{2}\kappa(\nabla\phi)^2 + \tfrac{1}{2}C(\nabla^2\phi)^2$$

Parameters are introduced via the keys:

```
free_energy brazovskii
A            -0.0005                    # Default: 0.0
B            +0.0005                    # Default: 0.0
K            -0.0006                    # Default: 0.0
C            +0.00076                   # Default: 0.0
```

For $A < 0$, phase separation occurs with a result depending on $\kappa$: one gets two symmetric phases for $\kappa > 0$ (cf. the symmetric case) or a lamellar phase for $\kappa < 0$. Typically, $B = -A$ and the parameter in the highest derivative $C > 0$.

### 6.2.3 Surfactants

The surfactant free energy should not be used at the present time.

### 6.2.4 Polar active gels

The free energy density is a function of vector order parameter $P_\alpha$:

$$\tfrac{1}{2}AP_\alpha P_\alpha + \tfrac{1}{4}B(P_\alpha P_\alpha)^2 + \tfrac{1}{2}\kappa(\partial_\alpha P_\beta)^2$$

There are no default parameters:

```
free_energy      polar_active
polar_active_a  -0.1                    # Default: 0.0
polar_active_b  +0.1                    # Default: 0.0
polar_active_k   0.01                   # Default: 0.0
```

It is usual to choose $B > 0$, in which case $A > 0$ gives an isotropic phase, whereas $A < 0$ gives a polar nematic phase. The elastic constant $\kappa$ (key `polar_active_k`) is positive.

### 6.2.5 Liquid crystal

The free energy density is a function of tensor order parameter $Q_{\alpha\beta}$:

$$\begin{aligned}\tfrac{1}{2}A_0(1-\gamma/3)Q_{\alpha\beta}^2 - \tfrac{1}{3}A_0\gamma Q_{\alpha\beta}Q_{\beta\delta}Q_{\delta\alpha} + \tfrac{1}{4}A_0\gamma(Q_{\alpha\beta}^2)^2 \\ +\tfrac{1}{2}\Big(\kappa_0(\epsilon_{\alpha\delta\sigma}\partial_\delta Q_{\sigma\beta} + 2q_0Q_{\alpha\beta})^2 + \kappa_1(\partial_\alpha Q_{\alpha\beta})^2\Big)\end{aligned}$$

The corresponding `free_energy` value, despite its name, is suitable for nematics and cholesterics, and not just blue phases:

```
free_energy    lc_blue_phase
lc_a0          0.01                   # Deafult: 0.0
lc_gamma       3.0                    # Default: 0.0
lc_q0          0.19635                # Default: 0.0
lc_kappa0      0.00648456             # Default: 0.0
lc_kappa1      0.00648456             # Default: 0.0
```

The bulk free energy parameter $A_0$ is positive and controls the energy scale (key `lc_a0`); $\gamma$ is positive and influences the position in the phase diagram relative to the isotropic/nematic transition (key `lc_gamma`). The two elastic constants must be equal, i.e., we enforce the single elastic constant approximation (boths keys `lc_kappa0` and `lc_kappa1` must be specified).

Other important parameters in the liquid crystal picture are:

```
lc_xi          0.7                     # Default: 0.0
lc_Gamma       0.5                     # Default: 0.0
lc_active_zeta 0.0                     # Default: 0.0
```

The first is $\xi$ (key `lc_xi`) is the effective molecular aspect ratio and should be in the range $0 < \xi < 1$. The rotational diffusion constant is $\Gamma$ (key `lc_Gamma`; not to be confused with `lc_gamma`). The (optional) apolar activity parameter is $\zeta$ (key `lc_active_zeta`).

### 6.2.6 Liquid crystal emulsion

This an interaction free energy which combines the symmetric and liquid crystal free energies. The liquid crystal free energy constant $\gamma$ becomes a function of composition via $\gamma(\phi) = \gamma_0 + \delta(1 + \phi)$, and a coupling term is added to the free energy density:

$$WQ_{\alpha\beta}\partial_\alpha\phi\partial_\beta\phi.$$

Typically, we might choose $\gamma_0$ and $\delta$ so that $\gamma(-\phi^\star) < 2.7$ and the $-\phi^\star$ phase is isotropic, while $\gamma(+\phi^\star) > 2.7$ and the $+\phi^\star$ phase is ordered (nematic, cholesteric, or blue phase). Experience suggests that a suitable choice is $\gamma_0 = 2.5$ and $\delta = 0.25$.

For anchoring constant $W > 0$, the liquid crystal anchoring at the interface is planar, while for $W < 0$ the anchoring is normal. This is set via key `lc_droplet_W`.

Relevant keys (with default values) are:

```
free_energy          lc_droplet

A                    -0.0625
B                    +0.0625
K                    +0.053

lc_a0                 0.1
lc_q0                 0.19635
lc_kappa0             0.007
lc_kappa1             0.007

lc_droplet_gamma      2.586              # Default: 0.0
lc_droplet_delta      0.25               # Default: 0.0
lc_droplet_W         -0.05               # Default: 0.0
```

Note that key `lc_gamma` is not set in this case.

## 6.3 System Parameters

Basic parameters controlling the number of time steps and the system size are:

```
N_start     0                          # Default: 0
N_cycles    100                        # Default: 0
size        128_128_1                  # Default: 64_64_64
```

A typical simulation will start from time zero (key `N_start`) and run for a certain number of time steps (key `N_cycles`). The system size (key `size`) specifies the total number of lattice sites in each dimension. If a two-dimensional system is required, the extent in the $z$-direction must be set to unity, as in the above example.

If a restart from a previous run is required, the choice of parameters may be as follows:

```
N_start     100
N_cycles    400
```

This will restart from data previously saved at time step 100, and run a further 400 cycles, i.e., to time step 500.

### 6.3.1 Parallel decomposition

In parallel, the domain decompostion is closely related to the system size, and is specified as follows:

```
size        64_64_64
grid        4_2_1
```

The `grid` key specifies the number of MPI tasks required in each coordinate direction. In the above example, the decomposition is into 4 in the $x$-direction, into 2 in the $y$-direction, while the $z$-direction is not decomposed. In this example, the local domain size per MPI task would then be $16 \times 32 \times 64$. The total number of MPI tasks available must match the total implied by `grid` (8 in the example).

The `grid` specifications must exactly divide the system size; if no decomposition is possible, the code will terminate with an error message. If the requested decomposition is not valid, or `grid` is omitted, the code will try to supply a decomposition based on the number of MPI tasks available and `MPI_Dims_create()`; this may be implementation dependent.

## 6.4 Fluid Parameters

Control parameters for a Newtonian fluid include:

```
fluid_rho0              1.0
viscosity               0.166666666666666
viscosity_bulk          0.166666666666666
isothermal_fluctuations off
temperature             0.0
```

The mean fluid density is $\rho_0$ (key `fluid_rho0`) which defaults to unity in lattice units; it is not usually necessary to change this. The shear viscosity is `viscosity` and as default value 1/6 to correspond to unit relaxation time in the lattice Boltzmann picture. Reasonable values of the shear viscosity are $0.2 > \eta > 0.0001$ in lattice units. Higher values move further into the over-relaxation region, and can result in poor behaviour. Lower values increase the Reynolds number and tend to cause problems with stability. The bulk viscosity has a default value which is equal to whatever shear viscosity has been selected. Higher values of the bulk viscosity may be set independently and can help to suppress large deviations from incompressibility and maintain numerical stability in certain situations.

If fluctuating hydrodynamics is wanted, set the value of `isothermal_fluctualtions` to `on`. The associated temperature is in lattice units: reasonable values (at $\rho_0 = 1$) are $0 < kT < 0.0001$. If the temperature is too high, local velocities will rapidly exceed the Mach number constraint and the simulation will be unstable.

# 7 Software Configuration Management Plan

## 7.1 Introduction

### 7.1.1 Purpose

This section contains information on the *Software Configuration Management Plan* for the *Ludwig* code. It is to provide users of the code a background on the way in which the code is developed, how bugs are dealt with, the way in which new features may be added, how correctness is ensured and tested, and so on. It is therefore a part of the process by which the code is maintained. The Software Configuration Management Plan will be referred to as 'The Plan' throughout the remainder of the section.

### 7.1.2 Scope

The *Ludwig* code is designed to study simple and complex fluids based around the numerical solution of the Navier-Stokes equations. Components of *Ludwig* include the main program, unit and regression tests, and a small number support libraries, and a small number of utility programs used to prepare input and post-process output. The Plan is limited to these software components.

### 7.1.3 Relationship to organisation and other projects

*Ludwig* is an on-going research project, and relies mainly on funding for the United Kingdom Engineering and Physical Sciences Research Council to provide support for staff time to work on maintenance and development.

### 7.1.4 References

This section is based on the IEEE standard for Configuration Management IEEE 828-2012 [21], and follows the format set out therein. Relevant references are included and can be found in the main References section at the end of the document.

## 7.2 Identification

The material under control of the project includes a number of sets of files: this documentation, the source code, the build and test suite which is used to monitor the status of the code, and so on. Control of project files is via a single on-line repository which uses the subversion (SVN) revision control system. The SVN repository is currently located at

`http://ccpforge.cse.rl.ac.uk/`

which is maintained at the Rutherford Appleton Laboratory on behalf of Collaborative Computational Physics 5 (The Computer Simulation of Condensed Phases). Subversion identifies different revisions by a unique revision number. A given version of a file is then uniquely identified by its path in the repository as it exists at a given revision number.

Specifically, configuration items are taken to include files in the trunk of the repository

`http://ccpforge.cse.rl.ac.uk/svn/ludwig/trunk`

Other branches in the repository are not considered to be under configuration management.

## 7.3 Responsibility and authority

While the Soft Matter Physics Group is responsible for the overall scientific direction and development of the code and concomitant priorities, all Configuration Management activities will be the responsibility of EPCC at The University of Edinburgh.

## 7.4 Project organisation

### 7.4.1 Organisations

*Ludwig* is developed by a team based in the School of Physics at The University of Edinburgh. This involves two groups: the Soft Matter Physics Group, and Edinburgh Parallel Computing Centre (EPCC). These groups collaborate with a number of workers at different Universities around the world on different aspects of the code.

### 7.4.2 Process Management

EPCC is responsible for the Software Configuration Management process. It is anticipated that the cost of this process will be small as the project team can communicate informally on a regular basis. There is currently no independent surveillance of activities to ensure compliance with The Plan.

### 7.4.3 Control

Requests for changes or additions to the code should be made via the issue tracker at CCPForge. The project team will then decide whether the change is possible and/or desirable. If so, the implementation and testing of the change will take place, and the new code committed back to the repository.

*Submitting a change request:* A description of the proposed change will be submitted to the CCPForge issue tracker as a change request to provide a record of the change process.

*Evaluating a change request:* The request, together with any additional information, will be evaluated by the SCM team. The request will be accepted, refined, or rejected as appropriate. The change owner will make necessary updates to the change record.

*Implementation of change:* Important changes to behaviour of code related to a change will be accompanied by a relevant descriptive record in the change request.

*Version control:* A `MAJOR.MINOR.PATCH` numerical version name scheme is used [2]. It is expected that bug fixes and relatively small changes will be accompanied by a unit increment in the patch level; more significant changes will be reflected at the minor version number; major reconstructions will occur rarely and incur a change of major version number.

### 7.4.4 Status

The status of changes will be tracked at the CCPForge issue tracker with a named SCM team member owning the change. The relevant tracker item will be closed when successfully implemented and tested.

The tracker is available to CPPForge member users to provide a record of changes to the code. Anonymous users will be alerted to changes via release notes.

### 7.4.5 Release management and delivery

There are currently no formal releases of the *Ludwig* code. Any release management and delivery will be via the SVN repository.

## 7.5 Plan Maintenance

The Plan is currently under review to consider organisational changes affecting the responsibilities different stakeholders.

## References


[1] Adhikari, R. J.-C. Desplat, and K. Stratford, Sliding periodic boundary conditions for lattice Boltzmann and lattice kinetic equations, `arXiv:cond-mat/0503175v1` (2005).

[2] Apache Portable Runtime Project "APR's Version Numbering". See, e.g., `https://apr.apache.org/versioning.html)` (accessed November 2015).

[3] Adhikari, R., K. Stratford. M.E. Cates, and A.J. Wagner, Fluctuating Lattice Boltzmann, *Europhys. Lett.*, **71**, 473 2005.

[4] Aidun, C.K., Y. Lu, and E.-J. Ding, Direct analysis of particulate suspensions with inertia using the discrete Boltzmann equation, *J. Fluid Mech.*, **373**, 287, 1998.

[5] G.K. Batchelor *An Introduction to Fluid Mechanics*, Cambridge University Press (1967).

[6] J.R. Blake, A spherical envelope approach to ciliary propusion, *J. Fluid Mech.*, **46**, 199, 1971.

[7] M. E. Cates, J.-C. Desplat, P. Stansell, A.J. Wagner, K. Stratford, R. Adhikari, and I. Pagonabarraga, Physical and Computational Scaling Issues in Lattice Boltzmann Simulations of Binary Fluid Mixtures, *Phil. Trans. Roy. Soc. A*, **363**, 1917 (2005).

[8] B. Chun and A.J.C. Ladd, Interpolated boundary condition for lattice Boltzmann simulations in narrow gaps, *Phys. Rev. E*, **75**, 066705, 2007.

[9] Cichocki, B., and R.B. Jones, Image representation of a spherical particle near a hard wall, *Physica A*, **258**, 273, 1998.

[10] C.-H. Chang and E.I. Franses, Adsorption dynamics of surfactants at the air/water interface: a critical review of mathematical models, data, and mechanisms, *Colloids and Surfaces A*, **100** 1, 1995.

[11] Desplat, J.-C., I. Pagonabarraga, and P. Bladon, LUDWIG: A parallel lattice-Boltzmann code for complex fluids. *Comput. Phys. Comms.*, **134**, 273, 2001.

[12] H. Diamant and D. Andelman, Kinetics of surfactant adsorption at fluid/fluid interfaces: non-ionic surfactants, *Europhys. Lett.* **34**, 575 (1996).

[13] H. Diamant and D. Andelman, Kinetics of surfactant adsorption at fluid-fluid interfaces, *J. Phys. Chem.*, **100** 13732, 1996.

[14] J.-B. Fournier and P. Galatola, Modeling planar degenerate wetting and anchoring in nematic liquid crystals, *Europhys. Lett.*, **72** 403–409 (2005).

[15] J. Eastoe and J.S. Dalton, Dynamic surface tension and adsorption mechanims of surfactants at the air-water interface, *Advances in Colloid and Interface Science*, **85** 13, 2000.

[16] I. Ginzburg and D. d'Humières, Multireflection boundary conditions for lattice Boltzmann models, *Phys. Rev. E*, **68**, 066614, 2003.

[17] A. Gray and K. Stratford, TargetDP reference.

[18] Hasimoto, H., On the periodic fundamental solutions of the Stokes equation and their application to viscous flow past a cubic array of spheres. *J. Fluid Mech.*, **5**, 317.

[19] Heemels, M.W., M.H.J. Hagen, and C.P. Lowe, Simulating solid colloidal particles using the lattice-Boltzmann method, *J. Comp. Phys.*, **164**, 48, 2000.

[20] IEEE Standard 208-2005, IEEE Standard for Software Configuration Management Plans. See `http://ieeexplore.ieee.org/xpl/standards.jsp` (accessed November 2015).

[21] IEEE Standard 828-2012. IEEE Standard for Configuration Management in Systems and Software Engineering. See `http://ieeexplore.ieee.org/xpl/standards.jsp` (accessed November 2015).

[22] Jeffrey, D.J., and Y. Onishi, Calculation of the resistance and mobility functions for the two unequal rigid spheres in low-Reynolds-number flow, *J. Fluid Mech.*, **139**, 261, 1984.

[23] Kendon, V.M., M.E. Cates, I. Pagonabarraga, J.-C. Desplat, and P. Bladon, Inertial effects in three dimensional spinodal decomposition of a symmetric binary fluid mixture: A lattice Boltzmann study, *J. Fluid Mech.*, **440**, 147 (2001).

[24] Ladd, A.J.C., Numerical simulations of particulate suspensions via a discretised Boltzmann equation. Part 1. Theoretical foundation, *J. Fluid. Mech.*, **271**, 285, 1994.

[25] Ladd, A.J.C., Numerical simulations of particulate suspensions via a discretised Boltzmann equation. Part 2. Numerical results, *J. Fluid. Mech.*, **271**, 311, 1994.

[26] Ladd, A.J.C., Sedimentation of homogenous suspensions of non-Brownian spheres, *Phys. Fluids*, **9**, 491. 1996.

[27] Ladd, A.J.C., Hydrodynamic screening in sedimentating suspensions of non-Brownian spheres, *Phys. Rev. Lett.*, **76**, 1392, 1996.

[28] Ladd, A.J.C., and R. Verberg, Lattice-Boltzmann simulations of particle-fluid suspensions, *J. Stat. Phys.*, **104**, 1191, 2001.

[29] H. Li, C. Pan, and C.T. Miller, Pore-scale investigation of viscous coupling effects for two-phase flow in porous media, *Phys. Rev. E*, **72**, 026705, 2005.

[30] M.J. Lighthill, On the squirming motion of nearly spherical deformable bodies through liquid at very small Reynolds numbers, *Comm. Pure Appl. Math.*, **5**, 109, 1952.

[31] I. Llopis Fusté, *Hydrodynamic cooperativity in micro-swimmer suspensions*, Ph.D. Thesis, University of Barcelona, 2008.

[32] Message Passing Interface Forum. MPI: A Message Passing Interface Standard Version 1.3 (2008).

[33] Nguyen, N.-Q., and A.J.C. Ladd, Lubrication corrections for lattice-Boltzmann simulations of particle suspensions, *Phys. Rev. E*, **66**, 046708, 2002.

[34] T. paapnastasiou, G. Georgiou, and A. Alexandrou, *Viscous Fluid Flow*, CRC Press, Boca Raton, Florida, 2000.

[35] Paraview. See `http://www.paraview.org/`. Accessed 2011.

[36] Rapaport, D.C., *The Art of Molecular Dynamics Simulation*, Cambridge University Press, 1995.

[37] S. Succi, *The lattice Boltzmann equation and beyond*, Oxford University Press, Oxford, 2001.

[38] M. Venuroli and E.S. Boek, Two-dimensional lattice-Boltzmann simulations of single phase flow in a pseudo two-dimensional micromodel, *Physica A*, **362**, 23, 2006.

[39] R.G.M. van der Sman and S. van der Graaf, Diffuse interface model of surfactant adsorption onto flat and droplet interfaces, *Rheol. Acta* **46** 3 (2006).

[40] O. Theissen and G. Gompper, Lattice Boltzmann study of spontaneous emulsification, *Eur. Phys. J. B*, **11** 91 (1999).

[41] A.F.H. Ward and L. Tordai, *J. Chem. Phys.* **14** 453, 1946.

[42] M. Skarabot, M. Ravnik, S. Zumer, U. Tkalec, I. Poberaj, D. Babic, N. Osterman and I. Musevic, *Phys. Rev. E* **76**, 051406 (2007).

[43] D.C. Wright and N.D. Mermin, *Rev. Mod. Phys.* **61**, 385 (1989).

[44] J. Lyklema *Fundamentals of Interface and Colloid Science* Academic Press (1995).

[45] S. Mafé, J.A. Manzanares, J. Pellicer, *J.* Electroanal. Chem. **2**41, 5 7-77 (1988).

[46] F. Capuani, I. Pagonabarraga, D. Frenkel, *J.* Chem. Phys. **1**21, 973- 986 (2004).

[47] B. Rotenberg, I. Pagonabarraga, D. Frenkel, *F*arad. Discuss. **1**44, 223-243 (2010).

[48] L.D. Landau, E.M. Lifshitz, *E*lectrodynamics of Continuous Media, §15 , 2nd ed., Pergamon Press, Oxford, UK (1984).

[49] J.R. Melcher, *C*ontinuum Electromechanics, §3.10, MIT Press, Cambridge, MA, USA (1981).
downloadable from:
`http://ocw.mit.edu/ans7870/resources/melcher/resized/cem_811.pdf`

[50] L.D. Landau, E.M. Lifshitz, *T*heory of Elasticity, §3 & §16, 3rd e d., Butterworth-Heinemann, Oxford, UK (1986).