DL_MESO USER MANUAL

M. A. Seaton and W. Smith

STFC Daresbury Laboratory
Daresbury, Warrington, Cheshire, WA4 4AD
United Kingdom

Version 2.5, April 2012

Contents

Acknowledgements

DL_MESO was developed under the auspices of the Engineering and Physical Sciences Research Council (EPSRC) for the EPSRC's Collaborative Computational Project for the Computer Simulation of Condensed Phases (CCP5).

The members of the CCP5 DL_MESO consortium were:

David M. Heyes, University of Surrey

Chris M. Care, Sheffield Hallam University

Peter V. Coveney, University College London

David Emerson, STFC Daresbury Laboratory

Rob English, North East Wales Institute

Andrea Ferrante, Unilever Port Sunlight

Ian Halliday, Sheffield Hallam University

John Harding, University of Sheffield

Sebastian Reich, Imperial College

Bill Smith, STFC Daresbury Laboratory

Patrick B. Warren, Unilever Port Sunlight

Julia Yeomans, Oxford University

Many other people have given advice and encouragement in the development of DL_MESO. We gratefully acknowledge the support of the following people: Maurice Leslie, Richard Wain, Alexandre Dupuis, Jonathan Chin, Michael Dupin, Weiming Liu and John Purton.

Particular thanks go to Rongshan Qin at Imperial College as the original author of the Lattice Boltzmann Equation source code (DL_MESO_LBE) and the DL_MESO graphical user interface, and Richard Anderson at STFC Daresbury Laboratory and Ard van Bergen at Novidec for their contributions to the Dissipative Particle Dynamics source code (DL_MESO_DPD).

Chapter 1

DL_MESO General Information

1.1 Description

DL_MESO is a general purpose mesoscopic simulation package developed at Daresbury Laboratory by Dr Michael Seaton under the auspices of the Engineering and Physical Sciences Research Council (EPSRC) for the EPSRC's Collaborative Computational Project for the Computer Simulation of Condensed Phases (CCP5). The package is the property of the Science and Technology Facilities Council (STFC).

DL_MESO is issued free under licence to academic institutions pursuing scientific research of a non-commercial nature. All recipients of the code must first agree to the terms and conditions of the licence and register with us to be kept aware of new developments and discovered bugs. Commercial organisations interested in acquiring the package should approach the Scientific Computing Department, STFC Daresbury Laboratory in the first instance. Daresbury Laboratory is the sole centre for distribution of the package. Under no account is it to be redistributed to third parties without consent of the owners.

DL_MESO contains two mesoscale simulation methods:

- Lattice Boltzmann Equation (included with version 1.0 and later)
- Dissipative Particle Dynamics (included with version 2.0 and later)

1.2 Functionality

The following is a list of the features that DL_MESO currently supports. Users are reminded that we are interested in hearing what other features could be usefully incorporated. We obviously have ideas of our own and CCP5 strongly influences developments, but other input would be welcome nevertheless.

1.2.1 Lattice Boltzmann Equation

DL_MESO_LBE can simulate lattice-gas systems using the Lattice Boltzmann Equation (LBE). The following properties and features are currently available:

- Multiple fluid components, solutes and coupled heat transfers[72]
- Collisions: Bhatnagar-Gross-Krook (BGK) single-relaxation-time[4] or Multiple-Relaxation-Time (MRT)[33, 8]
- Boundary conditions: Periodic, bounce-back (including stationary objects), constant pressure/velocity at planar surfaces[73]

- Mesoscale interactions: Shan-Chen pseudopotential method[55, 56], Lishchuk continuum-based method[35]
- Initial conditions can either be determined by DL_MESO_LBE or specified by the user

1.2.2 Dissipative Particle Dynamics

DL_MESO_DPD can model DPD particles ('beads') with soft or hard potential fields, along with thermostatting dissipative and random forces. The following properties and features are currently available:

- Choice of integrators/thermostats: standard Velocity Verlet, DPD Velocity Verlet[13], Lowe-Andersen[36], Peters[47] and Stoyanov-Groot[62]
- Constant volume (NVT) or constant pressure (NPT) simulations with Berendsen[2] or Langevin[29] barostats
- User selection of interaction lengths, conservative and dissipative force parameters for each species and between unlike species
- Bond stretching, angles and dihedrals between beads in user-defined 'molecules'
- Potentials: standard Groot-Warren DPD[16], density-dependent (many-body) DPD[46, 66], Lennard-Jones[30], Weeks-Chandler-Andersen[69]
- Electrostatic potentials between charged beads using a modified Ewald summation[14]
- Boundaries: Periodic, hard reflecting walls with optional short-range repulsions[50], frozen particle walls, Lees-Edwards periodic shearing boundaries[34]
- Initial conditions can either be determined by DL_MESO_DPD or specified by the user

1.3 Requirements

1.3.1 Software requirements

- Standard C++ Compiler for LBE source code, DL_MESO_LBE
- Standard Fortran90 Compiler for DPD source code, DL_MESO_DPD
- Message Passing Interface (if parallel execution required)
- JAVA 2 Version 1.4 or higher (if GUI is to be used)
- GNU Make (included in standard Unix/Linux distributions; can be installed for Windows)

1.3.2 System requirements

DL_MESO is designed to work in both serial and parallel running; it can be run on standalone machines, clusters and supercomputers. The code has been tested on Solaris, Windows XP, IBM p690+ HPCx, PowerPC 450 Blue Gene/P and Cray XT4/XT6 HECToR machines.

1.4 The DL_MESO Directory Structure

The supplied version of DL_MESO is a gzipped tar file, which unpacks as directory **dl_meso_2.x**, where x is a generation number. Beneath the top level of this directory are a number of subdirectories:

- **LBE** - containing the LBE source code
- **DPD** - containing the DPD source code
- **JAVA** - containing the GUI source code
- **MAN** - containing the DL_MESO user manual
- **DEMO** - containing test cases for DL_MESO
- **WORK** - an example 'working directory'

1.5 Disclaimer

Neither STFC, CCP5 nor any of the authors of the DL_MESO package guarantee that the package is free from error. Neither do they accept responsibility for any loss or damage that results from its use.

1.6 Copyright

1.7 Authors

Dr Michael Seaton and Prof. William Smith
Scientific Computing Department
STFC Daresbury Laboratory
Warrington
WA4 4AD
United Kingdom

1.8 Suggestions and Bug Reports

We encourage users to send suggestions for improvements and new features for DL_MESO, including bug reports and subroutines, as well as any additional test cases that demonstrate its features. All of these should be sent to `michael.seaton@stfc.ac.uk`

Chapter 2

The DL_MESO GUI

2.1 Getting Started with the DL_MESO GUI

The DL_MESO GUI offers a convenient way of using the DL_MESO package, although it is not an essential tool for those who prefer command line operation: Appendix A provides details on compiling the DL_MESO program codes manually. Working with the GUI requires the availability of Java tools, particularly the `javac` compiler and the `java` runner for Java 2 version 1.4 or later. These may be obtained from the `java.sun.com` website.

To build the GUI, proceed as follows:

- Enter the `DL_MESO/JAVA` directory.
- Type `javac *.java` to compile the source code.
- Type `jar -cfm GUI.jar manifest.mf *.class` to create the `GUI.jar` executable `JAR` file.
- Move to your working directory.
- Launch the GUI.

A Unix/Linux script called `makegui` that performs the build of the GUI can be found in the `JAVA` subdirectory.

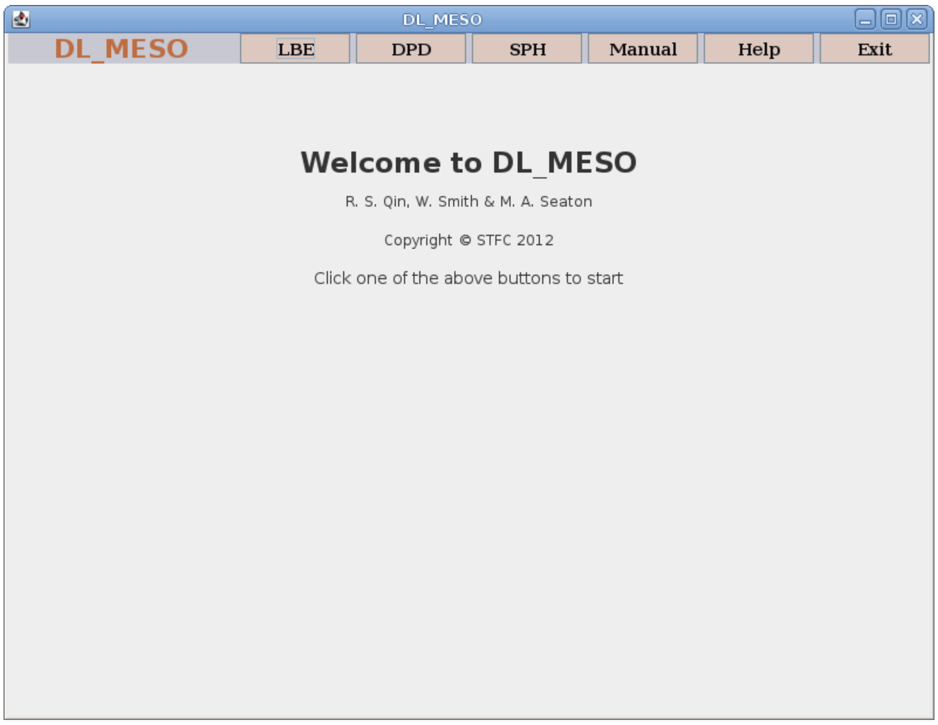
Your working directory is the directory from which you wish to work when running DL_MESO. Working there will keep any files you generate separate from the DL_MESO source files. **Note** in the current version of DL_MESO the working directory should be at the same directory level as the `JAVA` direction, i.e. within the `DL_MESO` top directory, and contain the executables of any external utilities required to set up input files and gather or process output files from simulations. An example of such a working directory (called `WORK`) is present under the `DL_MESO` top directory; this includes a makefile to compile all of the external utilties which can be invoked by the command `make -f Makefile-utils`.

In your working directory you can start the GUI with the command

- `java -jar ../JAVA/GUI.jar`

You may consider saving this command in a script for simple execution. An example script for Unix/Linux called `rungui` is present in the `WORK` subdirectory.

Figure 2.1 shows the DL_MESO GUI when it is started. Clicking the **LBE** and **DPD** buttons will produce the Lattice Boltzmann and Dissipative Particle Dynamics panels respectively, which will guide you through setting up input files, modifying and compiling the program code, running the simulation and gathering the results files



Welcome to DL_MESO

R. S. Qin, W. Smith & M. A. Seaton

Copyright © STFC 2012

Click one of the above buttons to start

Figure 2.1: DL_MESO GUI on startup

for plotting and visualization. The **SPH** button is for Smoothed Particle Hydrodynamic simulations, which will be included in future versions of DL_MESO: clicking on this button will currently produce a warning message. This user manual can be read in Adobe Acrobat Reader (if installed) by clicking the **Manual** button, while **Help** will advise you to visit the DL_MESO website at `www.ccp5.ac.uk/DL_MESO`.

2.2 Lattice Boltzmann and the DL_MESO GUI

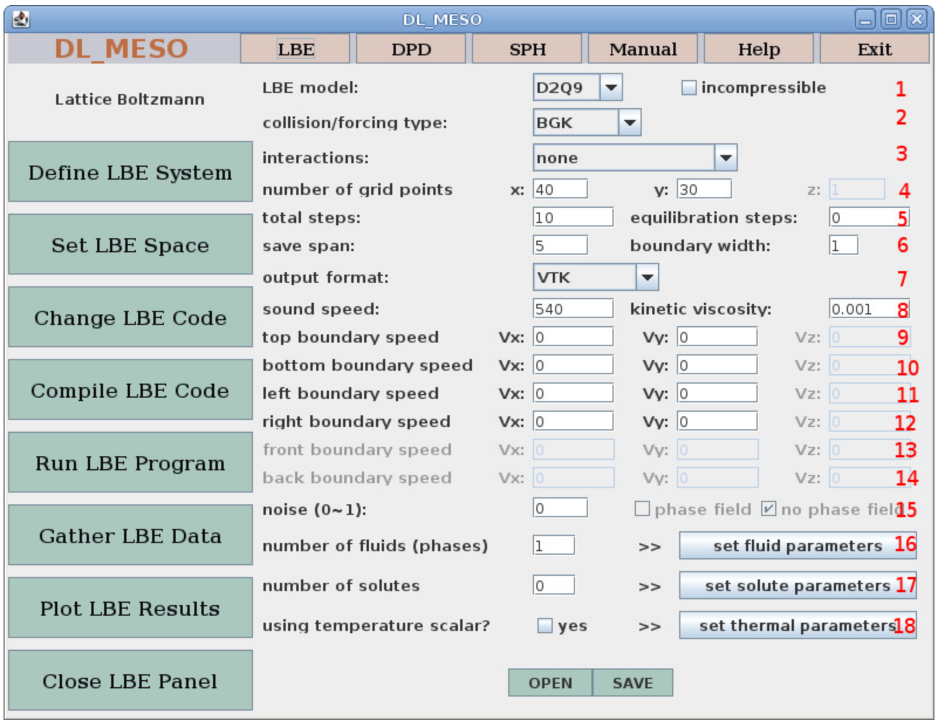
To access the LBE facilities in the DL_MESO GUI, proceed as follows:

- Click the **LBE** button to get the LBE panel.
- Click the **Define LBE System** button and supply the required information. The file `lbin.sys` will be created by the step.
- Click the **Set LBE Space** button to define the simulation space. The file `lbin.spa` will be created by this step.

2.2.1 Defining the System

Figure 2.2 shows the Define LBE System panel with the rows for data entry labelled in red numbering. The required data are as follows:

1. The required **LBE model** can be selected from the pull-down list: the D2Q9, D3Q15, D3Q19 and D3Q27 square lattice schemes are available. The tickbox can be selected to specify that the fluids in the system should be treated as **incompressible**.
2. The **collision/forcing type** for the system can be selected out of BGK, BGK with Guo forcing, MRT and MRT with Guo-like forcing.



LBE model: D2Q9 ☐ incompressible
collision/forcing type: BGK
interactions: none
number of grid points x: 40 y: 30 z: 1
total steps: 10 equilibration steps: 0
save span: 5 boundary width: 1
output format: VTK
sound speed: 540 kinetic viscosity: 0.001
top boundary speed Vx: 0 Vy: 0 Vz: 0
bottom boundary speed Vx: 0 Vy: 0 Vz: 0
left boundary speed Vx: 0 Vy: 0 Vz: 0
right boundary speed Vx: 0 Vy: 0 Vz: 0
front boundary speed Vx: 0 Vy: 0 Vz: 0
back boundary speed Vx: 0 Vy: 0 Vz: 0
noise (0~1): 0 ☐ phase field ☑ no phase field
number of fluids (phases) 1
number of solutes 0
using temperature scalar? ☐ yes

Figure 2.2: Define LBE System

3. The required mesophase **interactions** can be selected from the pull-down list: currently available options include no interactions, Shan/Chen pseudopotential interactions, Shan/Chen interactions with surface wetting and Lishchuk continuum-based interactions.

4. The **number of grid points** sets the size of the system. For 2D systems, the number of grid points in the z direction must equal 1; selecting a two-dimensional lattice model greys out this box.

5. The **total steps** and the **equilibration steps** for the simulation.

6. the **save span** (number of timesteps between system outputs) and the **boundary width** for running the parallel version of DL_MESO_LBE are given in this row. (The serial version by default automatically resets the boundary width to zero.)

7. The **output format** for system snapshots is set using this pull-down list: VTK, Legacy VTK and Plot3D.

8. The **sound speed** (c) and **kinetic viscosity** (ν) are real-life quantities for the first (main) fluid. These do not influence calculations at all but allow conversions between lattice and real units: the time step and lattice spacing are given by $\Delta t = \frac{\nu}{c^2\left(\tau_f - \frac{1}{2}\right)}$ and $\Delta x = \frac{\sqrt{3}\nu}{c\left(\tau_f - \frac{1}{2}\right)}$ respectively.

9. The specified **top boundary speed** (in lattice units), i.e. at the maximum y value for the grid. This and similar properties only need to be specified if the boundary includes a fixed velocity; any specified value will be ignored for periodic, bounce-back and fixed density conditions. Note that the z-component will be greyed out for two-dimensional systems.

10. The specified **bottom boundary speed** (in lattice units), i.e. at the minimum y value for the grid.

11. The specified **left boundary speed** (in lattice units), i.e. at the minimum x value for the grid.

12. The specified **right boundary speed** (in lattice units), i.e. at the maximum x value for the grid.

13. The specified **front boundary speed** (in lattice units), i.e. at the maximum z value for the grid. For two-dimensional systems this can be omitted and is thus greyed out.

14. The specified **back boundary speed** (in lattice units), i.e. at the minimum z value for the grid. For two-dimensional systems this can be omitted and is thus greyed out.

15. The **noise** magnitude only has an effect for initializing multiple phase simulations. DL_MESO_LBE may include either a **phase field** parameter or **no phase field** parameter for systems with multiple phases; neither mesophase algorithm requires it and thus this option is currently disabled.

16. The **number of fluids (phases)** can be increased if a multiple fluid system is to be studied: up to 6 fluids may be modelled in DL_MESO_LBE. The parameters and boundary conditions for the fluid(s) must then be set by clicking the **set fluid parameters** button – see below for more details.

17. The **number of solutes** needs changing if solute parameters are required: if the number of solutes is greater than zero (and up to 6), the number of fluids in the above row must be set to 0 or 1. If used, the parameters and boundary conditions for the solutes must be set by clicking the **set solute parameters** button – see below for more details.

18. The **using temperature scalar** box may be clicked **yes** if thermal systems are to be studied. If checked, the thermal parameters must be set by clicking the **set thermal parameters** button – see below for more details.

If a valid `lbin.sys` file already exists in the (current) working directory, the **OPEN** button can be clicked to load its information into the GUI, which can then be viewed and edited. Once all the data in this window and any pop-up windows for fluid, solute and thermal parameters are filled in, the **SAVE** button should be clicked to write the `lbin.sys` file.

2.2.1.1 Fluid, solute and thermal parameters

Examples of the pop-up windows for fluid, solute and thermal parameters can be seen in Figure 2.3 with the rows labelled in red numbering: multiple columns of dialogue boxes are made available for systems with multiple fluids and/or solutes.

For fluids, the required data are as follows:

1. **Body force x-axis**: the x-component of body force on each fluid.

2. **Body force y-axis**: the y-component of body force on each fluid.

3. **Body force z-axis**: the z-component of body force on each fluid. (Greyed out for two-dimensional systems.)

4. **Boussinesq force x-axis**: the x-component of the Boussinesq force parameter ($\vec{g}\beta$) on each fluid.

5. **Boussinesq force y-axis**: the y-component of the Boussinesq force parameter on each fluid.

6. **Boussinesq force z-axis**: the z-component of the Boussinesq force parameter on each fluid. (Greyed out for two-dimensional systems.)

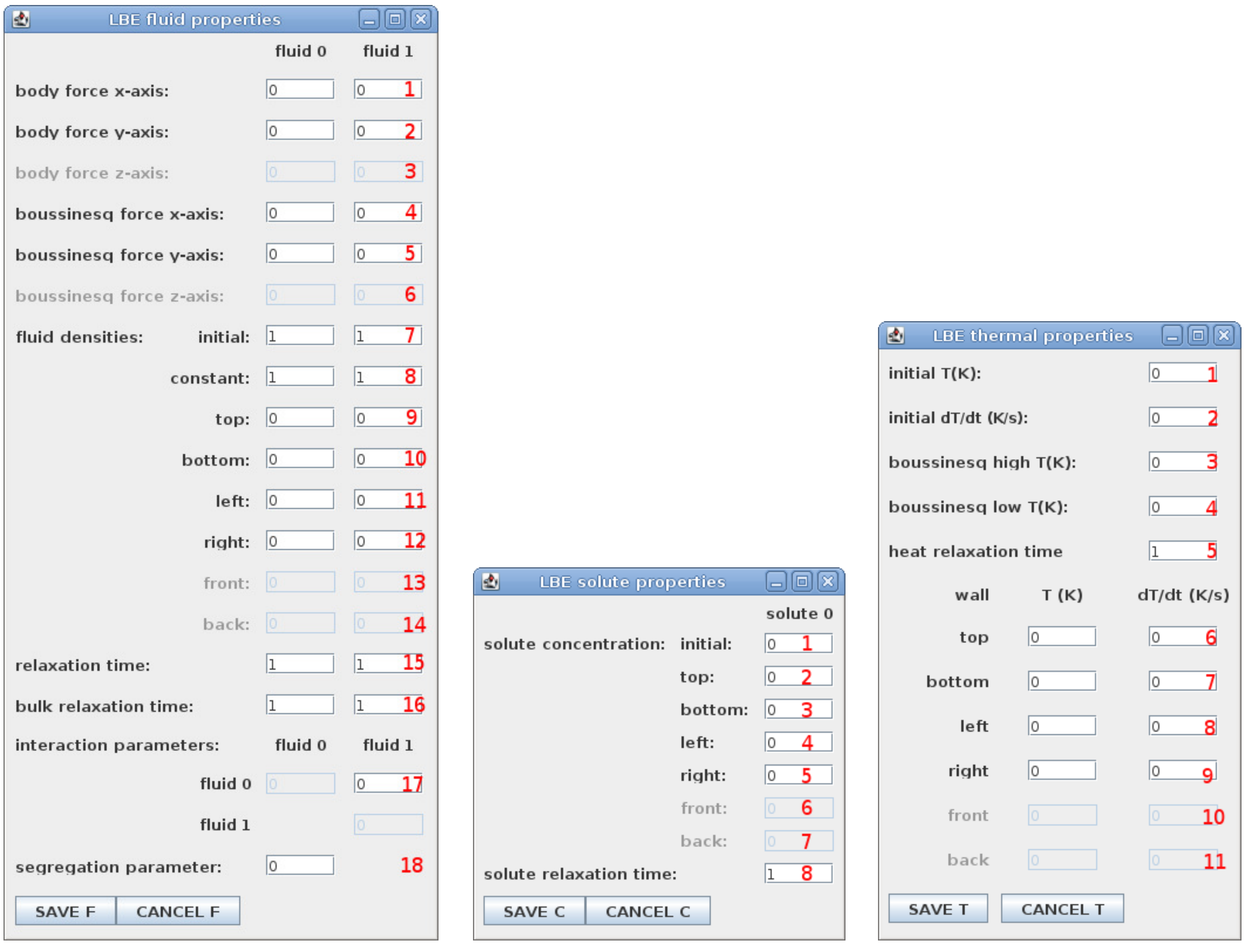
7. The **initial** fluid densities are applied throughout the system and used to initialize LBE calculations.

8. The **constant** fluid density (ρ_0) for incompressible systems: this property can also be used to define the reference densities for Shan/Chen pseudopotentials and for initialising systems with fluid drops.

9. The fluid densities at the **top** boundary.

10. The fluid densities at the **bottom** boundary.

11. The fluid densities at the **left** boundary.



LBE fluid properties

		fluid 0	fluid 1	
body force x-axis:		0	0	1
body force y-axis:		0	0	2
body force z-axis:		0	0	3
boussinesq force x-axis:		0	0	4
boussinesq force y-axis:		0	0	5
boussinesq force z-axis:		0	0	6
fluid densities:	initial:	1	1	7
	constant:	1	1	8
	top:	0	0	9
	bottom:	0	0	10
	left:	0	0	11
	right:	0	0	12
	front:	0	0	13
	back:	0	0	14
relaxation time:		1	1	15
bulk relaxation time:		1	1	16
interaction parameters:		fluid 0	fluid 1	
	fluid 0	0	0	17
	fluid 1		0	
segregation parameter:		0		18

SAVE F CANCEL F

LBE solute properties

		solute 0	
solute concentration:	initial:	0	1
	top:	0	2
	bottom:	0	3
	left:	0	4
	right:	0	5
	front:	0	6
	back:	0	7
solute relaxation time:		1	8

SAVE C CANCEL C

LBE thermal properties

initial T(K):		0	1
initial dT/dt (K/s):		0	2
boussinesq high T(K):		0	3
boussinesq low T(K):		0	4
heat relaxation time		1	5

wall	T (K)	dT/dt (K/s)	
top	0	0	6
bottom	0	0	7
left	0	0	8
right	0	0	9
front	0	0	10
back	0	0	11

SAVE T CANCEL T

(a) Fluid parameters (b) Solute parameters (c) Thermal parameters

Figure 2.3: Fluid, solute and thermal parameter pop-up windows

12. The fluid densities at the **right** boundary.

13. The fluid densities at the **front** boundary. (Greyed out for two-dimensional systems.)

14. The fluid densities at the **back** boundary. (Greyed out for two-dimensional systems.)

15. The **relaxation time** (τ_f) for each fluid: these values should be greater than 0.5 to give non-zero kinetic viscosities.

16. The **bulk relaxation time** ($\tau_{f,bulk}$) for each fluid: these values are only used in multiple-relaxation-time schemes to define the bulk viscosity and again should be greater than 0.5.

17. If more than one phase is being modelled, non-zero **interaction parameters** between the fluid species are required (g_{ab}). The Shan/Chen algorithm can accept values of g_{ab} for both $a = b$ and $a \neq b$, while the Lishchuk algorithm only requires g_{ab} for $a \neq b$.

18. If the Shan-Chen pseudopotential algorithm with wetting is to be used, **wall interaction parameters** between the fluid species and solid surfaces are required ($g_{a,wall}$). If the Lishchuk continuum-based algorithm is to be used, a non-zero **segregation parameter** (β) is required to ensure immiscible fluids separate out from each other.

Solutes require the following data in the following row numbers:

1. The **initial** concentrations of the solutes throughout the system, as used for initialization.

2. The solute concentrations at the **top** boundary.
3. The solute concentrations at the **bottom** boundary.
4. The solute concentrations at the **left** boundary.
5. The solute concentrations at the **right** boundary.
6. The solute concentrations at the **front** boundary. (Greyed out for two-dimensional systems.)
7. The solute concentrations at the **back** boundary. (Greyed out for two-dimensional systems.)
8. The **relaxation time** (τ_s) for each solute, representing diffusivities.

If selected for inclusion, the required thermal properties are:

1. The **initial T** (temperature) for the system.
2. The **initial dT/dt** (rate of change of temperature: related to heat transfers in or out) for the entire system.
3. The **Boussinesq high** reference temperature (T_h) for heat convection in the system.
4. The **Boussinesq low** reference temperature (T_l) for heat convection in the system.
5. The **heat relaxation time** (τ_t) for the system, which represents the thermal diffusivity.
6. The temperature and rate of temperature change at the **top** boundary.
7. The temperature and rate of temperature change at the **bottom** boundary.
8. The temperature and rate of temperature change at the **left** boundary.
9. The temperature and rate of temperature change at the **right** boundary.
10. The temperature and rate of temperature change at the **front** boundary. (This is greyed out for two-dimensional systems.)
11. The temperature and rate of temperature change at the **back** boundary. (This is greyed out for two-dimensional systems.)

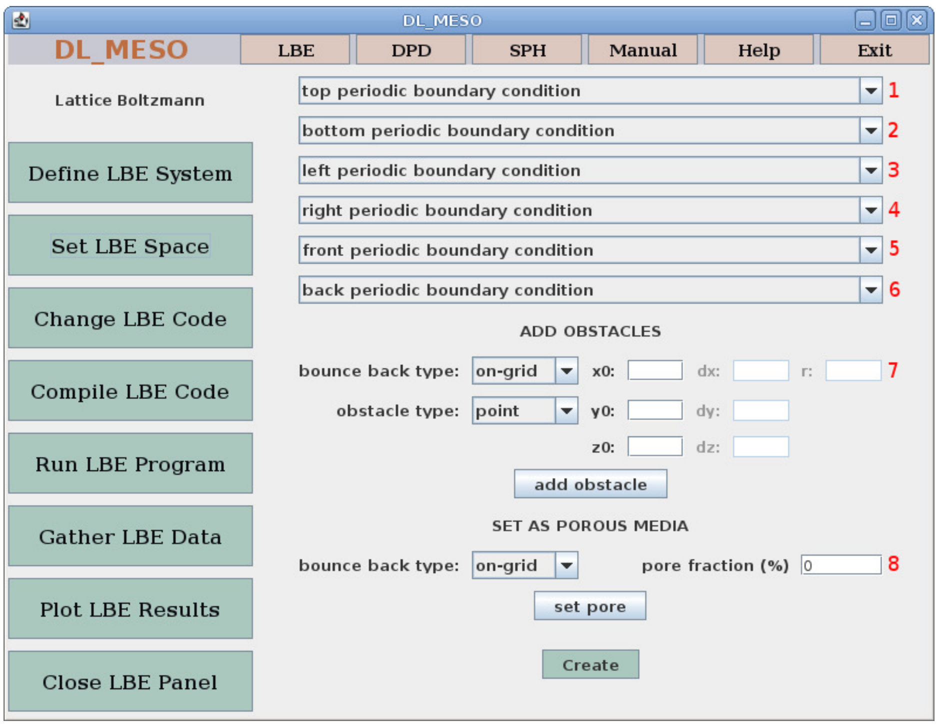
After filling in all the required values, clicking the relevant save button (**SAVE F**, **SAVE C** or **SAVE T**) will store the data in preparation for writing to the `lbin.sys` input file. The cancel buttons (**CANCEL F**, **CANCEL C** and **CANCEL T**) will close the pop-ups without saving any values.

2.2.2 Defining the Space Properties

If this option is selected before saving the LBE system data, a warning message advising that the system should be re-defined will appear.

Figure 2.4 shows the Set LBE Space panel with the rows for data entry labelled in red numbering. The following data are required:

1. The **top boundary condition** can be selected using the pull-down list from:
   - periodic
   - on-grid bounce back
   - mid-grid bounce back



top periodic boundary condition
bottom periodic boundary condition
left periodic boundary condition
right periodic boundary condition
front periodic boundary condition
back periodic boundary condition

bounce back type: on-grid x0: dx: r:
obstacle type: point y0: dy:
z0: dz:

bounce back type: on-grid pore fraction (%) 0

Figure 2.4: Set LBE Space

- fixed V (velocity), C (concentration) and T (temperature)
- fixed V and C, Neumann[1] T
- fixed V and T, Neumann C
- fixed V, Neumann C and T
- fixed P (pressure or density), C and T
- fixed P and T, Neumann C
- fixed P and C, Neumann T
- fixed P, Neumann C and T

2. The **bottom boundary condition** can be selected using the pull-down list.

3. The **left boundary condition** can be selected using the pull-down list.

4. The **right boundary condition** can be selected using the pull-down list.

5. The **front boundary condition** can be selected using the pull-down list. This pull-down list will be greyed out for two-dimensional systems.

6. The **back boundary condition** can be selected using the pull-down list. This pull-down list will be greyed out for two-dimensional systems.

7. Solid obstacles can be added to the calculation space by selecting the bounce back (on-grid or mid-grid) and obstacle types in the pull-down lists, entering its location on the grid and, if necessary, entering its size, and clicking **add obstacle**.

   - A single **point** will be located at (**x0**, **y0**, **z0**).
   - A **sphere** is centred at (**x0**, **y0**, **z0**) and has radius **r**.
   - A two-dimensional **circle** is centred at (**x0**, **y0**) and has radius **r**.

[1]For a property ϕ, DL_MESO currently only calculates $\nabla\phi = 0$ by using on-grid bounce back on the related distribution function.

- A **block** has a vertex at (**x0**, **y0**, **z0**) and has size (**dx**, **dy**, **dz**): both **z0** and **dz** can be omitted for two-dimensional blocks.

Note that lattice points well within an obstacle are set as blank sites, i.e. they will be ignored in LBE calculations.

8. The entire system can be set up as a porous solid by selecting the bounce back type, specifying a **pore fraction** and clicking **set pore** to randomly select an appropriate number of solid lattice sites.

Clicking the **Create** button will write all the lattice space data to a `lbin.spa` file; any lattice point defined more than once will hold its *latest* definition.

2.3 Dissipative Particle Dynamics and the DL_MESO GUI

To access the DPD facilities in the DL_MESO GUI, proceed as follows:

- Click the **DPD** button to get the DPD panel.
- Click the **Define DPD System** button and supply the required information. The `CONTROL` file will be created by this step.
- Note that currently no simulation space settings or molecular structure data can be entered using the GUI.
- Click **EXIT** to finish the settings.

2.3.1 Defining the System

Figure 2.5 shows the Define DPD System panel with the rows for data entry labelled in red numbering.

The required data are as follows:

1. The **job header**: a line of text up to 80 characters long describing the simulation.
2. The system **volume**: the pull-down list can be used to specify whether this is cubic or orthogonal, or whether replication of a `CONFIG` file is required (nfold). If specifying a cubic volume, the total volume should be specified, while orthogonal volumes require the sizes for all three dimensions and the nfold setting requires integer values specifying the number of replications in each dimension.
3. The target **temperature** ($k_B T$) and **pressure** (P_0) for the system. (The latter is greyed out if no barostat is to be used.)
4. The maximum **interaction cutoff** (r_c) for pairwise particle interactions and the **many-body cutoff** (r_d) for determining localized particle densities as used for many-body DPD.
5. If required, the **electrostatic cutoff** (r_e) for short-range electrostatic interactions and the **surface cutoff** (z_c) for interactions between particles and solid walls. (These are greyed out if not required.)
6. The size of the **boundary halo** for copying particle data from neighbouring subdomains or across periodic boundaries and the size of each **time step** (Δt) for integrating the equations of motion.
7. The **total steps** required for the DPD simulation and the number of time steps required to equilibrate the system (**equilibration steps**).

8. The numbers of time steps to store system variables for rolling averages (**stack interval**) and between rescaling of particle velocities to the desired system temperature during equilibration (**temp scaling interval**). The latter can be set to zero if no temperature rescaling is required.

9. The starting time step (**save start**) and the number of time steps between saves (**save interval**) of trajectory data to `HISTORY` files for later visualization. The latter can be set to zero if no trajectory data are required.

10. The numbers of time steps between printing summaries in the `OUTPUT` file (**print interval**) and outputs of statistical data (system energy, potential energies, pressure, temperature etc.) to a plottable `CORREL` file (**plot interval**). The latter value can be set to zero if no plot file is required.

11. The number of time steps between dumps of system configurations to `export` files for simulation restarts (**dump interval**) and the percentage variation in particle density (**density var**) to allow for unevenly distributed systems.

12. The **job time** is the maximum (real) time that can be spent carrying out the DPD simulation: the **close time** gives the time needed to write restart files and shut down the calculation in a controlled manner.

13. The **restart key** for the simulation: this can either be set to **none** for a new simulation, a **full restart** to continue a previous run using `export*` files, a **new run** which takes a starting state (particle positions and velocities) for a new simulation from `export*` files, and **rescaled** does the same as a new run but additionally rescales the particle velocities to give the specified system temperature.

14. The system **thermostat**: the dissipative and random forces as defined for DPD with the standard (molecular dynamics) form of the Velocity Verlet integrator (DPD/MD-VV) is the default, but recalculation of dissipative forces at the end of each step (DPD/DPD-VV)[13], the Lowe-Andersen[36], Peters[47] and Stoyanov-Groot[62] thermostats can also be selected. Values of γ for the DPD and Peters thermostats and Γ for the Lowe-Andersen and Stoyanov-Groot thermostats can be specified elsewhere for each pair of species, but an additional parameter for the Stoyanov-Groot thermostat should be set by clicking on **set thermostat** – see below for more details.

15. The system **barostat**: no barostat is used by default, but Langevin[29] and Berendsen[2] barostats are available in combination with all five thermostats. If either barostat is selected, its parameters can be set by clicking on **set barostat** and the target system pressure can be specified.

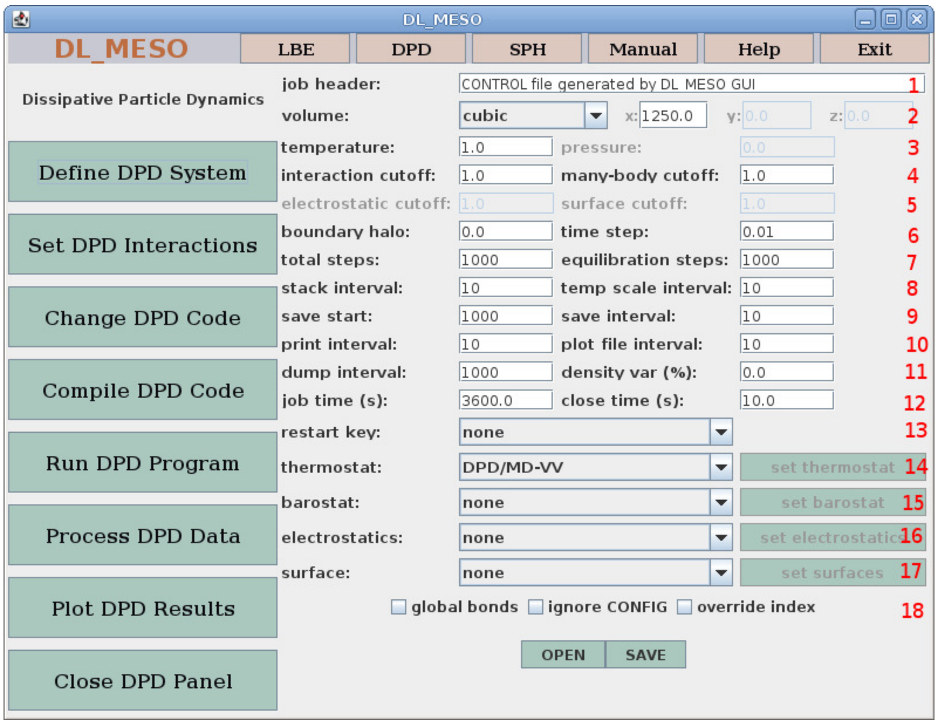
16. The **electrostatics** scheme for the simulation: an Ewald sum method with Slater-type (exponential) charge smearing[14] is available in DL_MESO_DPD. If selected, the short-range electrostatic cutoff can be edited and the parameters for the Ewald sum and charge smearing can be specified by clicking on **set electrostatics**.

17. The **surfaces** to be applied to the system: by default periodic boundary conditions are used, but alternative boundary conditions include hard walls with soft repulsions and specular reflection[50], walls of frozen beads and Lees-Edwards shearing periodic boundaries. The boundaries with the specified condition can be selected by clicking on **set surfaces**.

18. Switches to use global storage of bonds (**global bonds**), to **ignore CONFIG** files and to **override index** numbers in a `CONFIG` file can be set using these tickboxes.

Note that the DPD code uses reduced units in which the unit of length is the particle size, the unit of mass is the particle mass and the unit of energy is the primary energy parameter of the potential energy function. From these the time unit may be derived. The temperature is defined to be $\frac{2}{3}$ of the system kinetic energy.

If a valid `CONTROL` file already exists in the (current) working directory, the **OPEN** button can be clicked to load its information into the GUI, which can then be viewed and edited. The `CONTROL` file for input into DL_MESO_DPD is created by clicking the **SAVE** button.



job header: CONTROL file generated by DL MESO GUI 1

volume: cubic x: 1250.0 y: 0.0 z: 0.0 2

temperature: 1.0 pressure: 0.0 3

interaction cutoff: 1.0 many-body cutoff: 1.0 4

electrostatic cutoff: 1.0 surface cutoff: 1.0 5

boundary halo: 0.0 time step: 0.01 6

total steps: 1000 equilibration steps: 1000 7

stack interval: 10 temp scale interval: 10 8

save start: 1000 save interval: 10 9

print interval: 10 plot file interval: 10 10

dump interval: 1000 density var (%): 0.0 11

job time (s): 3600.0 close time (s): 10.0 12

restart key: none 13

thermostat: DPD/MD-VV 14

barostat: none 15

electrostatics: none 16

surface: none 17

☐ global bonds ☐ ignore CONFIG ☐ override index 18

Figure 2.5: Define DPD System

2.3.1.1 Thermostat, barostat, electrostatic and surface parameters

Examples of the pop-up windows for thermostat, barostat, electrostatic and surface parameters can be seen in Figure 2.6 with the rows labelled in red numbering: multiple columns of dialogue boxes are made available for systems with multiple species.

The thermostat pop-up window is formatted as in Figure 2.6(a):

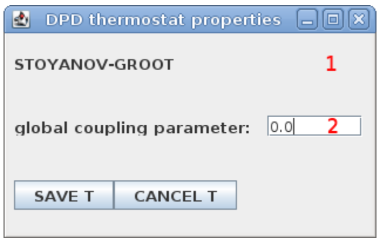
1. The type of thermostat to be used in the simulation.
2. Thermostat parameters: for the Stoyanov-Groot thermostat (currently the only type that requires an additional parameter), a **global coupling parameter** for the Nosé-Hoover part (α) is required.

Figure 2.6(b) gives the layout for the barostat pop-up window:

1. The type of barostat to be used in the simulation.
2. Barostat parameters: for the Langevin barostat, a **barostat relaxation time** (τ_p) and **piston drag coefficient** (γ_p) are required, while the Berendsen barostat requires the **compressibility/relaxation** ratio ($\frac{\beta}{\tau_p}$).
3. This check box determines whether or not an **isotropic system**, i.e. one where pressure acts uniformly in all dimensions, should be modelled. If unchecked, the barostat will act differently in each dimension and the shape of the system will change over time.

The parameters for electrostatics can be given in the pop-up window shown in Figure 2.6(c):

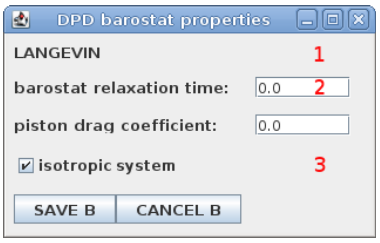
1. The type of electrostatics to be used in the simulation.
2. Electrostatic parameters: for the Ewald sum with Slater-type smearing, the **system coupling constant** (Γ), **Ewald real-space convergence** (α) and **charge smearing** (β) coefficients need to be specified.
3. The Ewald sum method also requires a reciprocal space (**k-vector**) range.



STOYANOV-GROOT

global coupling parameter: 0.0

(a) Thermostat parameters



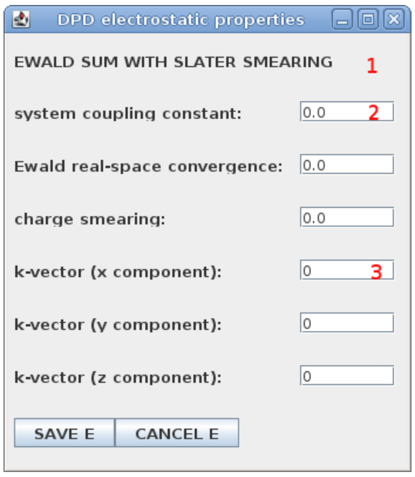
LANGEVIN

barostat relaxation time: 0.0

piston drag coefficient: 0.0

☑ isotropic system

(b) Barostat parameters



EWALD SUM WITH SLATER SMEARING

system coupling constant: 0.0

Ewald real-space convergence: 0.0

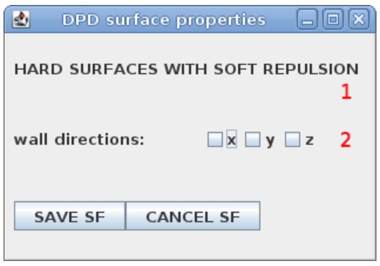
charge smearing: 0.0

k-vector (x component): 0

k-vector (y component): 0

k-vector (z component): 0

(c) Electrostatic parameters



HARD SURFACES WITH SOFT REPULSION

wall directions: ☐ x ☐ y ☐ z

(d) Surface parameters

Figure 2.6: Thermostat, barostat, electrostatic and surface pop-up windows

If non-periodic boundaries are to be used, the parameters for surfaces can be entered in the appropriate pop-up window (Figure 2.6(d)):

1. The type of surface interactions or boundary conditions to be applied.
2. **Wall directions**: if the checkbox for a particular dimension is ticked, the boundary condition will be applied to the surfaces orthogonal to the specified axis.

After filling in all the required values, clicking the relevant save button (**SAVE T**, **SAVE B**, **SAVE E** or **SAVE SF**) will store the data in preparation for writing to the `CONTROL` file. The cancel buttons (**CANCEL T**, **CANCEL B**, **CANCEL E** and **CANCEL SF**) will close the pop-ups without saving any values.

2.3.2 Defining DPD Interactions

Figure 2.7 shows the Set DPD Interactions panel with the sections for data entry labelled in red numbering. The following data are required:

1. The **number of species** is required to specify all interactions between particles in a DPD simulation. The spinner box allows the user to define up to 10 particle species, while the button **set species** opens a pop-up window for the user to enter the properties for each species and write them to a new `FIELD` file – see below for more details.
2. After the particles species have been defined, the button **set interactions** opens a pop-up window to allow the user to define non-bonded interactions between particle species and write them to the `FIELD` file.
3. The **operating system** is required before launching a command-line terminal and running the molecule generation utility `molecule-generate.cpp` to **create molecules** for the DPD simulation and write them

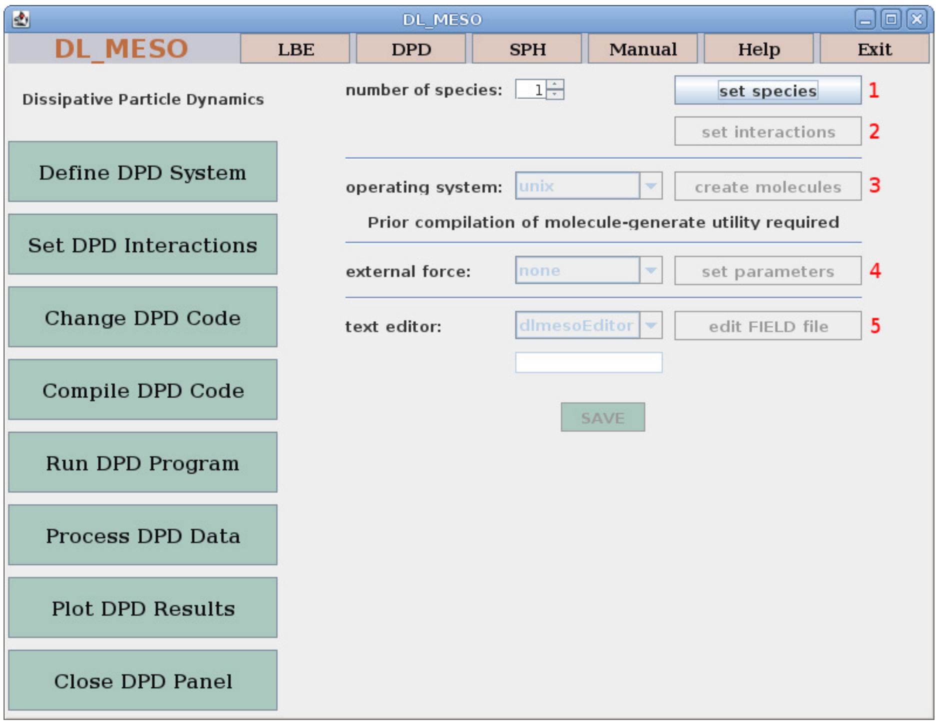


Figure 2.7: Set DPD Interactions

to the FIELD file. The utility should be compiled beforehand to give the executable `molecule.exe` (refer to Appendix B for more details).

4. It is possible to define an **external force** field on all particles in the system, using the pull-down box to define the type. Constant gravitational fields and linear shear boundaries can be defined. Clicking on the button **set parameters** opens a pop-up window to define the parameters for the external force field and write them to the FIELD file.

5. A **text editor**, including the built-in `dlmesoEditor`, may be selected using the pull-down box to view and edit the FIELD file. An alternative editor can be used by selecting 'other' and typing its name in the text box before clicking on the **edit FIELD file** button.

After the data for particle species, interactions, molecules and external fields are entered and written to the FIELD file, clicking **SAVE** will complete the file, which can still be viewed and edited afterwards using the text editor option described above.

2.3.2.1 Species, interactions and external field parameters

Examples of the pop-up windows for species, non-bonded interactions and external field parameters can be seen in Figure 2.8 with the rows labelled in red numbering.

The species pop-up window is formatted as in Figure 2.8(a), with individual columns for each species:

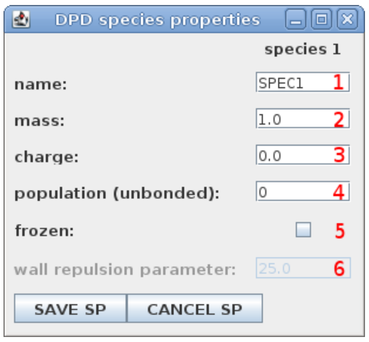
1. The **name** of each species, which can be up to 8 characters long.

2. The **mass** of a particle for the species (m_i).

3. The **charge** of a particle for the species (q_i).

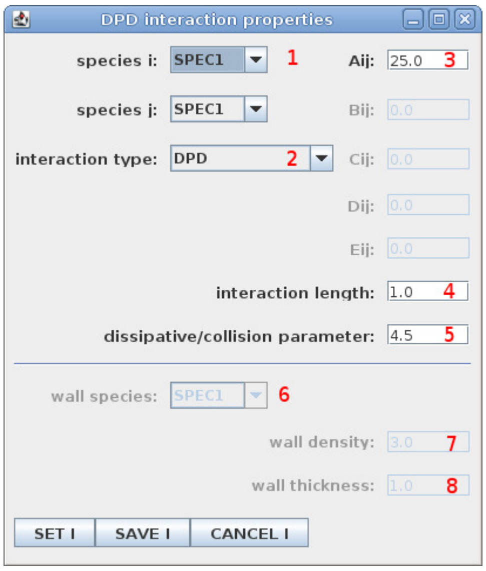
4. The number of unbonded particles of the species (**population (unbonded)**) in the system.

5. The tickbox indicates whether or not the particles for the species should be **frozen**.



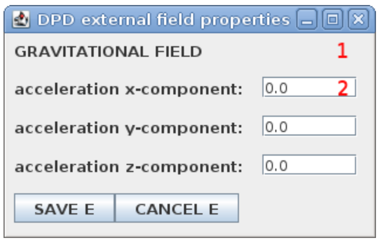
species 1
name: SPEC1
mass: 1.0
charge: 0.0
population (unbonded): 0
frozen: ☐
wall repulsion parameter: 25.0

(a) Species parameters



species i: SPEC1
species j: SPEC1
interaction type: DPD
Aij: 25.0
Bij: 0.0
Cij: 0.0
Dij: 0.0
Eij: 0.0
interaction length: 1.0
dissipative/collision parameter: 4.5
wall species: SPEC1
wall density: 3.0
wall thickness: 1.0

(b) Non-bonded interaction parameters



GRAVITATIONAL FIELD
acceleration x-component: 0.0
acceleration y-component: 0.0
acceleration z-component: 0.0

(c) External field parameters

Figure 2.8: Species, interactions and external field pop-up windows

6. If a non-periodic hard surface is defined, the **wall repulsion parameter** ($A_{wall,i}$) for the species can be specified.

Figure 2.8(b) gives the layout for the interaction pop-up window:

1. The pair of species can be selected using these pull-down boxes: the interaction parameters and type currently set for the selected species pair will be displayed.

2. The **interaction type** for the species pair: the standard DPD model by Groot and Warren[16] is the default, but many-body (density dependent) DPD[46, 66], Lennard-Jones[30] and Weeks-Chandler-Andersen 'hard sphere' models can also be selected. Note that while the Lennard-Jones and Weeks-Chandler-Andersen (WCA)[69] models are *not DPD* models, the DPD thermostat can be used with them to maintain system temperature.

3. The energy parameters for the species pair can be typed into these boxes and set using the button **SET I**. Only one energy parameter is required for standard DPD (A_{ij}), Lennard-Jones and WCA (ϵ_{ij}), while many-body DPD can use up to five: the exact number required will depend upon the model selected by the user. Note that values for these and other interaction parameters for *all species pairs* will be written to the `FIELD` file: if many-body DPD interactions are not included and mixing rules are to be used between unlike species, the file can subsequently be edited to remove extraneous definitions.

4. The maximum **interaction length** between the two species ($r_{c,ij}$ or σ_{ij}) can be typed into this box and set using the button **SET I**.

5. The dissipative factor (γ_{ij}) or collision frequency (Γ_{ij}) for the species pair (i.e. the parameter for the selected thermostat) can be typed into this box and set using the **SET I** button.

6. If a non-periodic frozen bead surface is defined, the species of beads making up the walls can be selected using this pull-down box.
7. The **wall density** of frozen beads can be typed into this box. (This is greyed out if frozen bead walls are not specified.)
8. The **wall thickness** of frozen beads can be typed into this box. (This is greyed out if frozen bead walls are not specified.)

The parameters for external force fields can be given in the pop-up window shown in Figure 2.8(c):

1. The type of external field to be used in the simulation.
2. External field parameters: for constant gravitational fields (or similar constant external force fields), the x-, y- and z-components of gravitational **acceleration** ($\vec{G}$) need to be specified. For linear shear boundaries, the x-, y- and z-components of the **boundary velocity** ($\vec{V}_w$) need to be defined, although the component orthogonal to the wall will be ignored in simulations.

After filling in all the required values, clicking the relevant save button (**SAVE SP**, **SAVE I** or **SAVE E**) will write the data to the `FIELD` file. The cancel buttons (**CANCEL SP**, **CANCEL I** and **CANCEL E**) will close the pop-ups without saving any values.

2.4 Compiling and running DL_MESO

- Compiling the LBE/DPD code may be accomplished through the compiler panel which is activated from either of the **Compile LBE Code** or **Compile DPD Code** buttons.
- The **Compile LBE Code** panel allows you to select the operating system, a C++ compiler, compiler flags and the version (serial or parallel) of the code you wish to build. If you require a C++ compiler that is not included in the pull-down list, select other and type the command for the required compiler in the neighbouring box. Clicking the **COMPILE** button will start the compilation and a message box will signal its completion.
- The **Compile DPD Code** panel allows you to select the operating system, a Fortran90 compiler, compiler flags and the version (serial or parallel) of the code you wish to build. If you require a Fortran90 compiler that is not included in the pull-down list, select other and type the command for the required compiler in the neighbouring box. The **Create Makefile** button needs to be clicked first to create a makefile in the working directory, which automates compilation and may be edited by the user. Clicking the **COMPILE** button will invoke the makefile to compile the code and a message box will signal its completion.
- If the compilation fails, you may need to edit the code. An editing panel is available for this purpose using either the **Change LBE Code** or **Change DPD code** buttons. Its function is similar to the compilation panel in operation with a choice of text editors, including one packaged with the DL_MESO GUI.
  - The files in the LBE code that can be edited include the parallel and serial main files, the lattice model file, the boundary condition file, the core routines for LBE calculations, the file for user-defined routines, the main head file and the head file for user-defined routines. Others can be edited by selecting other and typing the name of the file in the neighbouring box.
  - The files in the DPD code that may be edited include the main program, constants, global variables and the modules configuration, start (for system initialization), field (for force calculations), bond interactions, many-body DPD, surfaces and statistics. Other code files can be edited by selecting other and typing the name in the neighbouring box.

- Running the LBE/DPD code is made possible through the **Run LBE Program** or **Run DPD Program** button, which activates a panel that allows you to select the required submission command and then submit the job. You may need to create a suitable run script in your working directory beforehand if running the job in parallel.
- Collecting data from multiple processors and processing it for visualization is possible using the **Gather LBE Data** and **Process DPD Data** buttons. Note that the utilities need to be compiled in the working directory prior to use: details on this and their functions can be found in Appendix B.
- The results of LBE and DPD calculations may be plotted using the **Plot LBE Results** and **Plot DPD Results** buttons, which allows the user to select plotting and visualization applications, including those not available in the pull-down lists. Note that these need to be already installed on the workstation in use before being invoked by the GUI: if they require running from a command-line, tick the **run in terminal** box before launching the application.

2.5 Notes

- There are some inactive buttons reserved for later use.
- The GUI does not produce initial state files (`lbin.init` for LBE, `CONFIG` for DPD) prior to simulations, although there are utilities available to do this: see Appendix B for further details.
- Click **EXIT** to close down the GUI.

Part I

Lattice Boltzmann Equation (LBE)

Chapter 3

The Lattice Boltzmann Equation: Basic Theory

3.1 Introduction

The Lattice Boltzmann Equation (LBE) method is based on modelling a fluid consisting of fictional particles, which collide and move over a discrete lattice grid. This method is similar to its ancestor, Lattice Gas Cellular Automata (LGCA), but the main difference is that while LGCA represents the existence or otherwise for each particle at a grid point, LBE describes the physical state of an ensemble of particles by a single distribution function. This difference allows LBE to simulate both dilute fluids (i.e. those in which the mean free path of component particles is much larger than the lattice spacing) and condensed matter such as liquids.

The Lattice Boltzmann method uses fully discretized space, time and velocity to describe the evolution of fluid. Space is represented by a regularly distributed grid, time flow is obtained by integrating over discrete time steps and discrete velocity vectors (lattice links) are defined to ensure that a particle moves from one grid point to another without falling between them.

The Lattice Boltzmann algorithm can be summarized by the following:

- Fluid properties are mapped onto a discrete lattice.
- The physical state of the fluid at each lattice point is described by a set of particle distribution functions.
- The system evolves towards an equilibrium (or steady state) by means of a two-step process:
    1. Collision (relaxation) of the distribution function towards its local equilibrium form;
    2. Propagation of collided distribution functions along lattice links to neighbouring points.
- Macroscopic fluid variables (e.g. density, momenta) can be calculated from moments of the distribution functions.

Major benefits of the Lattice Boltzmann Equation method include the local nature of its most computationally intensive process (collision), making the method inherently and massively parallelizable, and its ability to model complex system geometries and/or fluid interactions with comparatively little additional computational cost.

3.2 Basic Definitions

Triangular and rectangular lattices are two of the most popular grid forms used in Lattice Boltzmann simulations. Triangular lattices have sixth-order rotation isotropy and have been widely applied in two-dimensional systems, e.g. the D2Q7 and D2Q13 models. Rectangular lattices have only fourth-order rotation isotropy but can more easily handle the simulation of three-dimensional systems with complex boundary conditions. The local equilibria for D2Q9 and D3Q27 lattice models can be derived *a priori* from the Maxwell equilibrium distribution. D3Q15 and D3Q19 models appear to be more popular than D3Q27 because the latter is much more expensive in terms of computing cost.

It is required that the equilibrium state should be able to reproduce elementary macroscopic fluid variables:

$$\rho = \sum_{i=0}^{q} f_i \tag{3.1}$$

$$\rho u_\alpha = \sum_{i=0}^{q} f_i e_{i\alpha} \tag{3.2}$$

where ρ is the density, f_i the ith particle distribution function, $\hat{e}_i$ the ith lattice link vector and u_α the macroscopic velocity along the α-axis.

In the Lattice Boltzmann method, the lattice link vectors $\hat{e}_i$ do not represent the thermal velocities of a particle and therefore

$$E \neq \frac{1}{2} \sum f_i \left(\hat{e}_i - \vec{u}\right)^2 \tag{3.3}$$

Equation (3.3) implies that the temperature cannot ordinarily be derived from the lattice particle distribution function and that the fluid modelled using LBE is generally *athermal.* It is possible, however, to alleviate this problem by defining a temperature at each grid point and either using a thermal lattice scheme with additional link vectors or modelling either the temperature or internal energy on an additional lattice grid.

3.3 Derivation of Equilibrium

There are two methods by which the local equilibria for the Lattice Boltzmann Equation can be constructed. The *bottom-up* method obtains the equilibrium from the Maxwell-Boltzmann equilibrium distribution. The *top-down* method constructs the equilibrium so that the required macroscopic properties can be reproduced. Only the bottom-up method is shown here; the top-down method can be found in [6].

The Maxwell-Boltzmann single particle equilibrium distribution function is

$$f^{eq} = \frac{\rho}{(2\pi\theta)^{\frac{D}{2}}} \exp\left[-\frac{\left(\vec{\xi} - \vec{u}\right)^2}{2\theta}\right] \tag{3.4}$$

where $\theta = k_B T/m$, k_B is the Boltzmann constant, T is temperature, m is molar mass, D is the space dimension, $\vec{\xi}$ is the thermal velocity and $\vec{u}$ the macroscopic velocity.

When $|\vec{\xi} - \vec{u}| \ll \sqrt{\theta}$, Equation (3.4) can be expanded into

$$f^{eq} = \frac{\rho}{(2\pi\theta)^{\frac{D}{2}}} \exp\left(-\frac{\xi^2}{2\theta}\right) \left[1 + \frac{\vec{\xi} \cdot \vec{u}}{\theta} + \frac{\left(\vec{\xi} \cdot \vec{u}\right)^2}{2\theta^2} - \frac{u^2}{2\theta}\right] \tag{3.5}$$

For a microscopic quantity $\psi(\xi)$, the associated macroscopic quantity Ψ is calculated by

$$\Psi = \int \psi(\xi) f^{eq} d\xi \tag{3.6}$$

Let $\vec{\xi} = \sqrt{2\theta}\vec{c}$, where $\vec{c}$ is a rescaled thermal velocity; the macroscopic velocity $\vec{u}$ can be similarly rescaled to $\sqrt{2\theta}\vec{u}$. Equations (3.5) and (3.6) can thus be combined to give

$$\Psi = \int e^{-c^2} \psi(c) \frac{\sqrt{2\theta}\rho}{(2\pi\theta)^{\frac{D}{2}}} \left[1 + 2\left(\vec{c} \cdot \vec{u}\right) + 2\left(\vec{c} \cdot \vec{u}\right)^2 - u^2\right] dc \tag{3.7}$$

Using Gaussian quadrature, Equation (3.7) changes into

$$\Psi = \sum_i \psi\left(c_i\right) \frac{\sqrt{2\theta}\rho}{\left(2\pi\theta\right)^{\frac{D}{2}}} w\left(c_i\right) \left[1 + 2\left(\vec{c}\cdot\vec{u}\right) + 2\left(\vec{c}\cdot\vec{u}\right)^2 - u^2\right] \tag{3.8}$$

Let

$$w_i = \frac{\sqrt{2\theta}\rho}{\left(2\pi\theta\right)^{\frac{D}{2}}} w\left(c_i\right) \tag{3.9}$$

and

$$f_i^{eq} = w_i\rho\left[1 + 2\left(\vec{c}\cdot\vec{u}\right) + 2\left(\vec{c}\cdot\vec{u}\right)^2 - u^2\right] \tag{3.10}$$

The value of $w\left(c_i\right)$ can be obtained from Gauss-Hermite integration. Equation (3.10) is the equilibrium particle distribution function in the discrete regime. w_i is called the weight factor for speed vector c_i. Equation (3.10) can also be written as

$$f_i^{eq} = w_i\rho\left[1 + \frac{3\left(\hat{e}_i\cdot\vec{u}\right)}{c^2} + \frac{9\left(\hat{e}_i\cdot\vec{u}\right)^2}{2c^4} - \frac{3u^2}{2c^2}\right] \tag{3.11}$$

where $c = \sqrt{3\theta} = \sqrt{\frac{3k_BT}{m}}$ is the modulus of the basic lattice vector and equivalent to the fluid speed of sound (e.g. for water at 20℃, $c = 367.8$ m/s).

3.4 Structural Relaxation and Macroscopic Equations

The Lattice Boltzmann method often uses the BGK (Bhatnagar, Gross and Krook) approximation[4] to describe the structural relaxation. The single particle distribution function evolves to the equilibrium state via

$$f_i\left(\vec{x} + \hat{e}_i\Delta t, t + \Delta t\right) - f_i\left(\vec{x}, t\right) = -\frac{\Delta t}{\tau_f}\left[f_i\left(\vec{x}, t\right) - f_i^{eq}\right] \tag{3.12}$$

where τ_f is called the relaxation time and is related to the kinetic viscosity of fluid. This evolution equation can be divided into two separate processes of *collision* (where t^+ denotes a time after collision has taken place)

$$f_i\left(\vec{x}, t^+\right) = f_i\left(\vec{x}, t\right) - \frac{\Delta t}{\tau_f}\left[f_i\left(\vec{x}, t\right) - f_i^{eq}\right] \tag{3.13}$$

and *propagation* (or free-streaming)

$$f_i\left(\vec{x} + \hat{e}_i\Delta t, t + \Delta t\right) = f_i\left(\vec{x}, t^+\right). \tag{3.14}$$

To derive the macroscopic equations, the left hand side of Equation (3.12) can be expanded as

$$f_i\left(\vec{x} + \hat{e}_i\Delta t, t + \Delta t\right) - f_i\left(\vec{x}, t\right) = \sum_{m=1}^{\infty} \frac{\Delta t^m}{m!}\left(\partial_t + e_{i\alpha}\partial_\alpha\right)^m f_i\left(\vec{x}, t\right) \tag{3.15}$$

Expanding the instantaneous particle distribution function around its equilibrium and retaining only the first order gives

$$f_i\left(\vec{x}, t\right) = f_i^{eq}\left(\vec{x}, t\right) - \tau_f\left(\partial_t + e_{i\alpha}\partial_\alpha\right) f_i^{eq}\left(\vec{x}, t\right) + O\left(\partial^2\right) \tag{3.16}$$

Substituting Equations (3.15) and (3.16) into the left hand side of Equation (3.12) gives the second order differential equation for the equilibrium distribution

$$\frac{f_i^{eq} - f_i}{\tau_f} = \left(\partial_t + e_{i\alpha}\partial_\alpha\right) f_i^{eq} - w_f\left(\partial_t + e_{i\alpha}\partial_\alpha\right)^2 f_i^{eq} + O\left(\partial^3\right) \tag{3.17}$$

where $w_f = \tau_f - \frac{\Delta t}{2}$.

Summing Equation (3.17) over i and ignoring the second order deriviative we obtain

$$0 = \partial_t \rho + \partial_\alpha \rho u_\alpha - w_f \partial_\beta \left(\partial \rho u_\beta + \partial_\alpha \sum_i f_i^{eq} e_{i\alpha} e_{i\beta} \right) + O\left(\partial^3\right) \tag{3.18}$$

Summing Equation (3.18) times e_i over i we obtain

$$0 = \partial_t \rho u_\alpha + \partial_\beta \sum_i f_i^{eq} e_{i\alpha} e_{i\beta} - w_f \partial_\gamma \left(\partial_t \sum_i f_i^{eq} e_{i\alpha} e_{i\gamma} + \partial_\beta \sum_i f_i^{eq} e_{i\alpha} e_{i\beta} e_{i\gamma} \right) + O\left(\delta^3\right) \tag{3.19}$$

Equation (3.19) shows that the second term in Equation (3.18) is of the third order in the derivative. Therefore we have the continuity equation to the second order of the derivative

$$\partial_t \rho + \nabla \cdot \rho \vec{u} = 0 \tag{3.20}$$

Defining the third and fourth order moments

$$\sum_i f_i^{eq} e_{i\alpha} e_{i\beta} = P_{\alpha\beta} + \rho u_\alpha u_\beta \tag{3.21}$$

$$\sum_i f_i^{eq} e_{i\alpha} e_{i\beta} e_{i\gamma} = P_{\alpha\beta} u_\gamma + P_{\alpha\gamma} u_\beta + P_{\beta\gamma} u_\alpha + \rho u_\alpha u_\beta u_\gamma \tag{3.22}$$

With these definitions, Equation (3.19) leads to the weakly compressible Navier-Stokes equation

$$\partial_t(\rho u_\alpha) + \partial_\beta(\rho u_\alpha u_\beta) = -\partial_\beta P_{\alpha\beta} + \frac{w_f D}{3} \partial_\beta \left(\frac{\delta_{\alpha\beta} - 3\partial_\alpha P_{\alpha\beta}}{D} \partial_\gamma(\rho u_\gamma) + \partial_\alpha(\rho u_\beta) + \partial_\beta(\rho u_\alpha) \right) + O\left(\partial^3\right) \tag{3.23}$$

where the kinetic viscosity is given by $\nu = \frac{w_f}{3} = \frac{1}{3}\left(\tau_f - \frac{1}{2}\right)$.

Similarly, the convection diffusion equation governing the evolution of solute transfer and the convection/conduction equation governing the evolution of thermal transfer can be obtained.

3.5 Mesoscale Interaction

The rate of change of momentum is proportional to the force

$$\rho u_\alpha\left(t + \Delta t\right) = \rho u_\alpha\left(t\right) + F_\alpha \Delta t \tag{3.24}$$

The Lattice Boltzmann Equation takes into account the effect of relaxation and it is possible to express Equation (3.24) in a new format[41]:

$$\rho u_\alpha\left(t + \Delta t\right) = \rho u_\alpha\left(t\right) + F_\alpha \tau_f \tag{3.25}$$

$\vec{F}$, with F_α as the component in the α direction, can be a long-ranged body force or any local interactions.

In the Shan-Chen pseudopotential model for multiple phases and components[55], the force on component a is defined as

$$\vec{F^a} = -\psi^a\left(\vec{x}\right) \sum_b g_{ab} \sum_i w_i \psi^b\left(\vec{x} + \hat{e}_i\right) \hat{e}_i \tag{3.26}$$

where g_{ab} is the interaction coefficient between elements a and b. ψ^a is related to the density of element a and can take many different forms, e.g.

$$\psi^a\left(\vec{x}\right) = \rho_0^a \left[1 - \exp\left(-\frac{\rho^a}{\rho_0^a}\right)\right] \tag{3.27}$$

In the Lishchuk continuum-based model for single phases and multiple components[35], the force acting between components a and b is expressed as

$$\vec{F}_{ab} = \frac{1}{2} g_{ab} K_{ab} \nabla \rho_{ab}^N \tag{3.28}$$

where $\rho^N_{ab} = \frac{\rho^a - \rho^b}{\rho^a + \rho^b}$ is a phase index between the two components and K_{ab} is the local curvature from the interface model, which can be determined from spatial gradients of the phase index. This algorithm requires a modification to the collision step: at each lattice point all fluids are collided as a single fluid, which is then segregated back out into the individual fluids.

3.6 Summary of Lattice Boltzmann Equation

Lattice Boltzmann is an established numerical methodology for handling hydrodynamics in fluid. It is particularly suited to simulating systems with complex boundary conditions and is also suitable for systems with phase transitions since mesoscale interactions can be merged into the method easily.

Chapter 4

DL_MESO_LBE Basic Definition

4.1 Lattice models

DL_MESO_LBE utilizes a right-handed Cartesian coordinate system with the x-axis from left to right in the horizontal direction, the y-axis from low to high in the vertical direction and z-axis from back to front.

D2Q9, D3Q15, D3Q19 and D3Q27 lattice models have been included. The speed vectors and weight factors are arranged to allow the use of swap algorithms for propagation[43]. The transformation matrices **T** for Multiple-Relaxation-Time (MRT) schemes are included for each lattice (except D3Q27), along with the equilibrium moments ($\vec{M}^{eq}$) expressed for the incompressible case (for the compressible case, ρ_0 is substituted with ρ), collision operators ($\vec{s}$), definition of bulk viscosity (ν') and forcing source-terms ($\vec{S}$).

D2Q9

Weight factor

i	w_i
0	$\frac{4}{9}$
2,4,6,8	$\frac{1}{9}$
1,3,5,7	$\frac{1}{36}$

Speed vector

i	$e_{i,x}$	$e_{i,y}$
0	0	0
1	-1	1
2	-1	0
3	-1	-1
4	0	-1
5	1	-1
6	1	0
7	1	1
8	0	1

$$\mathbf{T} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -4 & 2 & -1 & 2 & -1 & 2 & -1 & 2 & -1 \\ 4 & 1 & -2 & 1 & -2 & 1 & -2 & 1 & -2 \\ 0 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 1 & -2 & 1 & 0 \\ 0 & 1 & 0 & -1 & -1 & -1 & 0 & 1 & 1 \\ 0 & 1 & 0 & -1 & 2 & -1 & 0 & 1 & -2 \\ 0 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 \end{bmatrix}$$

$$\vec{M}^{eq} = \begin{pmatrix} \rho \\ e^{eq} \\ \epsilon^{eq} \\ j_x \\ q_x^{eq} \\ j_y \\ q_y^{eq} \\ p_{xx}^{eq} \\ p_{xy}^{eq} \end{pmatrix} = \begin{pmatrix} \rho \\ -2\rho + \frac{3}{\rho_0}\left(j_x^2 + j_y^2\right) \\ w_\epsilon \rho + \frac{w_{\epsilon j}}{\rho_0}(j_x^2 + j_y^2) \\ j_x \\ -j_x \\ j_y \\ -j_y \\ \frac{j_x^2 - j_y^2}{3\rho_0} \\ \frac{j_x j_y}{3\rho_0} \end{pmatrix}$$

$$\vec{S} = \begin{pmatrix} 0 \\ 6(v_x F_x + v_y F_y) \\ -6(v_x F_x + v_y F_y) \\ F_x \\ -F_x \\ F_y \\ -F_y \\ 2(v_x F_x - v_y F_y) \\ v_x F_y + v_y F_x \end{pmatrix}.$$

$$\vec{s} = \left(1, \tau_{f,bulk}^{-1}, s_2, 1, s_4, 1, s_4, \tau_f^{-1}, \tau_f^{-1}\right)^T$$

$$\nu' = \frac{1}{6}\left(\tau_{f,bulk} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t}$$

D3Q15

Weight factor

i	w_i
0	$\frac{2}{9}$
1–3, 8–10	$\frac{1}{9}$
4–7, 11–14	$\frac{1}{72}$

Speed vector

i	$e_{i,x}$	$e_{i,y}$	$e_{i,z}$
0	0	0	0
1	-1	0	0
2	0	-1	0
3	0	0	-1
4	-1	-1	-1
5	-1	-1	1
6	-1	1	-1
7	-1	1	1
8	1	0	0
9	0	1	0
10	0	0	1
11	1	1	1
12	1	1	-1
13	1	-1	1
14	1	-1	-1

$$\mathbf{T} = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-2 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\
16 & -4 & -4 & -4 & 1 & 1 & 1 & 1 & -4 & -4 & -4 & 1 & 1 & 1 & 1 \\
0 & -1 & 0 & 0 & -1 & -1 & -1 & -1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 4 & 0 & 0 & -1 & -1 & -1 & -1 & -4 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & -1 & 0 & -1 & -1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & -1 & -1 \\
0 & 0 & 4 & 0 & -1 & -1 & 1 & 1 & 0 & -4 & 0 & 1 & 1 & -1 & -1 \\
0 & 0 & 0 & -1 & -1 & 1 & -1 & 1 & 0 & 0 & 1 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 4 & -1 & 1 & -1 & 1 & 0 & 0 & -4 & 1 & -1 & 1 & -1 \\
0 & 2 & -1 & -1 & 0 & 0 & 0 & 0 & 2 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\
0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 1 & -1 & -1 & 1 \\
0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & -1 & 1 & 1 & -1 & 0 & 0 & 0 & 1 & -1 & -1 & 1
\end{bmatrix}$$

$$\vec{M}^{eq} = \begin{pmatrix} \rho \\ e^{eq} \\ \epsilon^{eq} \\ j_x \\ q_x^{eq} \\ j_y \\ q_y^{eq} \\ j_z \\ q_z^{eq} \\ 3p_{xx}^{eq} \\ p_{ww}^{eq} \\ p_{xy}^{eq} \\ p_{yz}^{eq} \\ p_{zx}^{eq} \\ m_{xyz}^{eq} \end{pmatrix} = \begin{pmatrix} \rho \\ -2\rho + \frac{(j_x^2+j_y^2+j_z^2)}{\rho_0} \\ w_\epsilon \rho + \frac{w_{\epsilon j}}{\rho_0}(j_x^2+j_y^2+j_z^2) \\ j_x \\ -\frac{7}{3}j_x \\ j_y \\ -\frac{7}{3}j_y \\ j_z \\ -\frac{7}{3}j_z \\ \frac{2j_x^2-j_y^2-j_z^2}{\rho_0} \\ \frac{j_y^2-j_z^2}{\rho_0} \\ \frac{j_x j_y}{\rho_0} \\ \frac{j_y j_z}{\rho_0} \\ \frac{j_z j_x}{\rho_0} \\ 0 \end{pmatrix}$$

$$\vec{S} = \begin{pmatrix} 0 \\ 2(v_x F_x + v_y F_y + v_z F_z) \\ -10(v_x F_x + v_y F_y + v_z F_z) \\ F_x \\ -\frac{7}{3}F_x \\ F_y \\ -\frac{7}{3}F_y \\ F_z \\ -\frac{7}{3}F_z \\ 2(2v_x F_x - v_y F_y - v_z F_z) \\ 2(v_y F_y - v_z F_z) \\ v_x F_y + v_y F_x \\ v_y F_z + v_z F_y \\ v_z F_x + v_x F_z \\ 0 \end{pmatrix}.$$

$$\vec{s} = \left(1, \tau_{f,bulk}^{-1}, s_2, 1, s_4, 1, s_4, 1, s_4, \tau_f^{-1}, \tau_f^{-1}, \tau_f^{-1}, \tau_f^{-1}, \tau_f^{-1}, s_{14}\right)^T$$

$$\nu' = \frac{2}{9}\left(\tau_{f,bulk} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t}$$

D3Q19

Weight factor

i	w_i
0	$\frac{1}{3}$
1–3, 10–12	$\frac{1}{18}$
4–9, 13–18	$\frac{1}{36}$

Speed vector

i	$e_{i,x}$	$e_{i,y}$	$e_{i,z}$
0	0	0	0
1	-1	0	0
2	0	-1	0
3	0	0	-1
4	-1	-1	0
5	-1	1	0
6	-1	0	-1
7	-1	0	1
8	0	-1	-1
9	0	-1	1
10	1	0	0
11	0	1	0
12	0	0	1
13	1	1	0
14	1	-1	0
15	1	0	1
16	1	0	-1
17	0	1	1
18	0	1	-1

$$\mathbf{T} = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-30 & -11 & -11 & -11 & 8 & 8 & 8 & 8 & 8 & 8 & -11 & -11 & -11 & 8 & 8 & 8 & 8 & 8 & 8 \\
12 & -4 & -4 & -4 & 1 & 1 & 1 & 1 & 1 & 1 & -4 & -4 & -4 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & -1 & 0 & 0 & -1 & -1 & -1 & -1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 4 & 0 & 0 & -1 & -1 & -1 & -1 & 0 & 0 & -4 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & -1 & 0 & -1 & 1 & 0 & 0 & -1 & -1 & 0 & 1 & 0 & 1 & -1 & 0 & 0 & 1 & 1 \\
0 & 0 & 4 & 0 & -1 & 1 & 0 & 0 & -1 & -1 & 0 & -4 & 0 & 1 & -1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & -1 & 0 & 0 & -1 & 1 & -1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 4 & 0 & 0 & -1 & 1 & -1 & 1 & 0 & 0 & -4 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 2 & -1 & -1 & 1 & 1 & 1 & 1 & -2 & -2 & 2 & -1 & -1 & 1 & 1 & 1 & 1 & -2 & -2 \\
0 & -4 & 2 & 2 & 1 & 1 & 1 & 1 & -2 & -2 & -4 & 2 & 2 & 1 & 1 & 1 & 1 & -2 & -2 \\
0 & 0 & 1 & -1 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 1 & -1 & 1 & 1 & -1 & -1 & 0 & 0 \\
0 & 0 & -2 & 2 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & -2 & 2 & 1 & 1 & -1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & -1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1
\end{bmatrix}$$

$$\vec{M}^{eq} = \begin{pmatrix} \rho \\ e^{eq} \\ \epsilon^{eq} \\ j_x \\ q_x^{eq} \\ j_y \\ q_y^{eq} \\ j_z \\ q_z^{eq} \\ 3p_{xx}^{eq} \\ 3\pi_{xx}^{eq} \\ p_{ww}^{eq} \\ \pi_{ww}^{eq} \\ p_{xy}^{eq} \\ p_{yz}^{eq} \\ p_{zx}^{eq} \\ m_x^{eq} \\ m_y^{eq} \\ m_z^{eq} \end{pmatrix} = \begin{pmatrix} \rho \\ -11\rho + \frac{19}{\rho_0}\left(j_x^2 + j_y^2 + j_z^2\right) \\ w_\epsilon \rho + \frac{w_{\epsilon j}}{\rho_0}(j_x^2 + j_y^2 + j_z^2) \\ j_x \\ -\frac{2}{3}j_x \\ j_y \\ -\frac{2}{3}j_y \\ j_z \\ -\frac{2}{3}j_z \\ \frac{2j_x^2 - j_y^2 - j_z^2}{\rho_0} \\ \frac{w_{xx}}{\rho_0}\left(2j_x^2 - j_y^2 - j_z^2\right) \\ \frac{j_y^2 - j_z^2}{\rho_0} \\ \frac{w_{xx}}{\rho_0}\left(j_y^2 - j_z^2\right) \\ \frac{j_x j_y}{\rho_0} \\ \frac{j_y j_z}{\rho_0} \\ \frac{j_z j_x}{\rho_0} \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\vec{S} = \begin{pmatrix} 0 \\ 38(v_x F_x + v_y F_y + v_z F_z) \\ -11(v_x F_x + v_y F_y + v_z F_z) \\ F_x \\ -\frac{2}{3}F_x \\ F_y \\ -\frac{2}{3}F_y \\ F_z \\ -\frac{2}{3}F_z \\ 2(2v_x F_x - v_y F_y - v_z F_z) \\ -(2v_x F_x - v_y F_y - v_z F_z) \\ 2(v_y F_y - v_z F_z) \\ -(v_y F_y - v_z F_z) \\ v_x F_y + v_y F_x \\ v_y F_z + v_z F_y \\ v_z F_x + v_x F_z \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

$$\vec{s} = \left(1, \tau_{f,bulk}^{-1}, s_2, 1, s_4, 1, s_4, 1, s_4, \tau_f^{-1}, s_4, \tau_f^{-1}, s_4, \tau_f^{-1}, \tau_f^{-1}, \tau_f^{-1}, s_{16}, s_{16}, s_{16}\right)^T$$

$$\nu' = \frac{2}{9}\left(\tau_{f,bulk} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t}$$

D3Q27

Weight factor

i	w_i
0	$\frac{8}{27}$
1–3, 14–16	$\frac{2}{27}$
4–9, 17–22	$\frac{1}{54}$
10–13, 23–26	$\frac{1}{216}$

Speed vector

i	$e_{i,x}$	$e_{i,y}$	$e_{i,z}$
0	0	0	0
1	-1	0	0
2	0	-1	0
3	0	0	-1
4	-1	-1	0
5	-1	1	0
6	-1	0	-1
7	-1	0	1
8	0	-1	-1
9	0	-1	1
10	-1	-1	-1
11	-1	-1	1
12	-1	1	-1
13	-1	1	1
14	1	0	0
15	0	1	0
16	0	0	1
17	1	1	0
18	1	-1	0
19	1	0	1
20	1	0	-1
21	0	1	1
22	0	1	-1
23	1	1	1
24	1	1	-1
25	1	-1	1
26	1	-1	-1

Table 4.1: Boundary condition category

value	meaning
0	liquid
10	domain boundary
11	inside solid
12	on-grid bounce-back boundary
13	mid-link bounce-back boundary
100–199	constant speed, composition and temperature boundary
200–299	constant speed, Neumann composition and temperature boundary
300–399	constant speed and composition, Neumann temperature boundary
400–499	constant speed and temperature, Neumann composition boundary
500–599	constant pressure, composition and temperature boundary
600–699	constant pressure, Neumann composition and temperature boundary
700–799	constant pressure and composition, Neumann temperature boundary
800–899	constant pressure and temperature, Neumann composition boundary

4.2 Data structure

4.2.1 Storage of particle distribution functions

For a system with a square lattice, the total number of grid points = `lbsy.nx` × `lbsy.ny` × `lbsy.nz`, where `lbsy.nx`, `lbsy.ny` and `lbsy.nz` are the numbers of grid points along the x-, y- and z-axes respectively. The grid points are arranged in a serial order of

$$g_{000}, g_{001}, \cdots g_{00\,\texttt{lbsy.nz}}, g_{010}, g_{011}, \cdots g_{0\,\texttt{lbsy.ny}\,\texttt{lbsy.nz}}, g_{100}, g_{101}, \cdots g_{\texttt{lbsy.nx}\,\texttt{lbsy.ny}\,\texttt{lbsy.nz}}$$

At each grid point, DL_MESO_LBE arranges the particle distribution functions in order of: fluid functions, solute functions, temperature functions and phase field order parameter. For example, for a D2Q9 lattice with two fluids, scalar temperature and phase field, the distribution functions are in the order of

$$f_0^0, f_1^0, f_2^0, f_3^0, f_4^0, f_5^0, f_6^0, f_7^0, f_8^0, f_0^1, f_1^1, f_2^1, f_3^1, f_4^1, f_5^1, f_6^1, f_7^1, f_8^1, T_0, T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, pf$$

Therefore the number of particle distribution functions at each grid point is

$$\texttt{lbsitelength} = (\texttt{lbsy.nf} + \texttt{lbsy.nc} + \texttt{lbsy.nt}) \times \texttt{lbsy.nq} + \texttt{lbsy.ph}$$

where `lbsy.nf`, `lbsy.nc`, `lbsy.nt`, `lbsy.nq` and `lbsy.ph` are respectively: number of fluids, number of solutes, number of temperature scalars, number of discrete speeds and number of the phase field order parameters. `lbsy.nt` and `lbsy.ph` can only take the values of 1 or 0, representing systems with or without temperature scalar and phase field. Also if `lbsy.nc` $\neq 0$, `lbsy.nf` cannot be set larger than 1.

4.2.2 Storage of space properties

The space property is represented by an integer value in DL_MESO_LBE. For example, `lbphi`[100] = 0 represents the 100th grid point as a liquid site and `lbphi`[101] = 12 shows the 101st grid point as an on-grid bounce-back boundary. Table 4.1 lists the categories of space properties.

The orientation of a solid-liquid boundary is also represented by the value of an integer. For example, a planar surface with normal vector along the y-axis is denoted by 21, while a concave corner face at the top-right-front corner is denoted by 31. It must be pointed out that only those space positions located in the surface of a face-centered cube have been included and translated in DL_MESO_LBE. Points with random orientations, e.g. 47° plane, have not been included.

The boundary condition number can be rather confusing and difficult to understand. The GUI in DL_MESO therefore includes a translator which interprets a *word* as its corresponding integer number. The *word* is made

Table 4.2: Boundary condition category

letter	**meaning**
V	Constant Velocity
P	Constant Pressure (Density)
C	Constant Solute Composition
T	Constant Temperature
B	Neumann Boundary Condition (Solute Composition or Temperature)
PS	Planar Surface
CC	Concave Corner
CE	Concave Edge
T	Normal Vector Pointing to Top
D	Normal Vector Pointing Downwards
L	Normal Vector Pointing to Left
R	Normal Vector Pointing to Right
F	Normal Vector Pointing to Front
B	Normal Vector Pointing to Back

up of defined letters as listed in Table 4.2. The boundary conditions with combinations of type and orientation are listed in Table 4.3. The *letters* are in the order of:

1. Fluid property: constant speed or constant pressure.
2. Solute property: constant composition or Neumann boundary.
3. Temperature property: isothermal (constant) or heat bath (Neumann boundary).
4. Geometric property: planar surface, concave corner or concave edge.
5. Boundary orientation: one letter for planar surface, two letters for concave corners or three letters for concave edges.

For example, a shearing planar surface facing down the y-axis with constant composition and temperature (i.e. isothermal) is represented as VCBPSD and translated as 322.

Table 4.3: Notation of boundary condition

VCTPST	121	VCTPSD	122	VCTPSL	123
VCTPSR	124	VCTPSF	125	VCTPSB	126
VCTCCTRB	127	VCTCCTLB	128	VCTCCDLB	129
VCTCCDRB	130	VCTCCTRF	131	VCTCCTLF	132
VCTCCDLF	133	VCTCCDRF	134	VCTCETR	143
VCTCETL	144	VCTCEDL	145	VCTCEDR	146
VCTCETF	147	VCTCELF	148	VCTCEDF	149
VCTCERF	150	VCTCETB	151	VCTCELB	152
VCTCEDB	153	VCTCERB	154	VBBPST	221
VBBPSD	222	VBBPSL	223	VBBPSR	224
VBBPSF	225	VBBPSB	226	VBBCCTRB	227
VBBCCTLB	228	VBBCCDLB	229	VBBCCDRB	230
VBBCCTRF	231	VBBCCTLF	232	VBBCCDLF	233
VBBCCDRF	234	VBBCETR	243	VBBCETL	244
VBBCEDL	245	VBBCEDR	246	VBBCETF	247
VBBCELF	248	VBBCEDF	249	VBBCERF	250
VBBCETB	251	VBBCELB	252	VBBCEDB	253
VBBCERB	254	VCBPST	321	VCBPSD	322
VCBPSL	323	VCBPSR	324	VCBPSF	325
VCBPSB	326	VCBCCTRB	327	VCBCCTLB	328
VCBCCDLB	329	VCBCCDRB	330	VCBCCTRF	331
VCBCCTLF	332	VCBCCDLF	333	VCBCCDRF	334
VCBCETR	343	VCBCETL	344	VCBCEDL	345
VCBCEDR	346	VCBCETF	347	VCBCELF	348
VCBCEDF	349	VCBCERF	350	VCBCETB	351
VCBCELB	352	VCBCEDB	353	VCBCERB	354
VBTPST	421	VBTPSD	422	VBTPSL	423
VBTPSR	424	VBTPSF	425	VBTPSB	426
VBTCCTRB	427	VBTCCTLB	428	VBTCCDLB	429
VBTCCDRB	430	VBTCCTRF	431	VBTCCTLF	432
VBTCCDLF	433	VBTCCDRF	434	VBTCETR	443
VBTCETL	444	VBTCEDL	445	VBTCEDR	446
VBTCETF	447	VBTCELF	448	VBTCEDF	449
VBTCERF	450	VBTCETB	451	VBTCELB	452
VBTCEDB	453	VBTCERB	454	PCTPST	521
PCTPSD	522	PCTPSL	523	PCTPSR	524
PCTPSF	525	PCTPSB	526	PCTCCTRB	527
PCTCCTLB	528	PCTCCDLB	529	PCTCCDRB	530
PCTCCTRF	531	PCTCCTLF	532	PCTCCDLF	533
PCTCCDRF	534	PCTCETR	543	PCTCETL	544
PCTCEDL	545	PCTCEDR	546	PCTCETF	547
PCTCELF	548	PCTCEDF	549	PCTCERF	550
PCTCETB	551	PCTCELB	552	PCTCEDB	553
PCTCERB	554	PBTPST	621	PBTPSD	622
PBTPSL	623	PBTPSR	624	PBTPSF	625
PBTPSB	626	PBTCCTRB	627	PBTCCTLB	628
PBTCCDLB	629	PBTCCDRB	630	PBTCCTRF	631
PBTCCTLF	632	PBTCCDLF	633	PBTCCDRF	634

Table 4.3: Notation of boundary condition (continued)

PBTCETR	643	PBTCETL	644	PBTCEDL	645
PBTCEDR	646	PBTCETF	647	PBTCELF	648
PBTCEDF	649	PBTCERF	650	PBTCETB	651
PBTCELB	652	PBTCEDB	653	PBTCERB	654
PCBPST	721	PCBPSD	722	PCBPSL	723
PCBPSR	724	PCBPSF	725	PCBPSB	726
PCBCCTRB	727	PCBCCTLB	728	PCBCCDLB	729
PCBCCDRB	730	PCBCCTRF	731	PCBCCTLF	732
PCBCCDLF	733	PCBCCDRF	734	PCBCETR	743
PCBCETL	744	PCBCEDL	745	PCBCEDR	746
PCBCETF	747	PCBCELF	748	PCBCEDF	749
PCBCERF	750	PCBCETB	751	PCBCELB	752
PCBCEDB	753	PCBCERB	754	PBBPST	821
PBBPSD	822	PBBPSL	823	PBBPSR	824
PBBPSF	825	PBBPSB	826	PBBCCTRB	827
PBBCCTLB	828	PBBCCDLB	829	PBBCCDRB	830
PBBCCTRF	831	PBBCCTLF	832	PBBCCDLF	833
PBBCCDRF	834	PBBCETR	843	PBBCETL	844
PBBCEDL	845	PBBCEDR	846	PBBCETF	847
PBBCELF	848	PBBCEDF	849	PBBCERF	850
PBBCETB	851	PBBCELB	852	PBBCEDB	853
PPBCERB	854				

4.2.3 Storage of running information

The Lattice Boltzmann component of DL_MESO defines three structures to store the system[1] information. The parameters in these structures are listed in Tables 4.4, 4.5, 4.6 and 4.7.

Table 4.4: System information

parameter	**meaning**
`lbsy.nd`	space dimension
`lbsy.nq`	number of discrete speeds
`lbsy.nf`	number of fluids
`lbsy.nc`	number of solutes
`lbsy.nt`	number of temperature scalars (0 or 1)
`lbsy.pf`	phase field order parameter
`lbsy.nx`	number of grid points in x direction
`lbsy.ny`	number of grid points in y direction
`lbsy.nz`	number of grid points in z direction (`lbsy.nz` $\equiv 1$ when `lbsy.nd` $= 2$)

4.3 The Parameters and Their Functions

Table 4.8 lists all the parameters defined in DL_MESO_LBE. The whole range parameters are named with the prefix `lb`. Because DL_MESO is an ongoing project and new parameters might be added to the package in the future, it is strongly suggested that users of DL_MESO would not name their own parameters with prefixes `lb`, `dp` or `sp`.

[1]Referring to the physical system being simulated rather than the computer system.

Table 4.5: Domain information

parameter	**meaning**
`lbdm.rank`	name of the processor
`lbdm.size`	number of processors
`lbdm.bwid`	domain boundary width (set to zero for serial running)
`lbdm.xcor`	x-coordinate of the processor
`lbdm.ycor`	y-coordinate of the processor
`lbdm.zcor`	z-coordinate of the processor (`lbdm.zcor` $\equiv 0$ when `lbsy.nd` $= 2$)
`lbdm.xdim`	number of processors along x-axis
`lbdm.ydim`	number of processors along y-axis
`lbdm.zdim`	number of processors along z-axis (`lbdm.zdim` $\equiv 1$ when `lbsy.nd` $= 2$)
`lbdm.xs`	x-coordinate of domain start position
`lbdm.xe`	x-coordinate of domain end position
`lbdm.xinner`	number of grid points along x-axis in the domain
`lbdm.xouter`	number of grid points along x-axis in the domain including the boundary
`lbdm.ys`	y-coordinate of domain start position
`lbdm.ye`	y-coordinate of domain end position
`lbdm.yinner`	number of grid points along y-axis in the domain
`lbdm.youter`	number of grid points along y-axis in the domain including the boundary
`lbdm.zs`	z-coordinate of domain start position
`lbdm.ze`	z-coordinate of domain end position
`lbdm.zinner`	number of grid points along z-axis in the domain
`lbdm.zouter`	number of grid points along z-axis in the domain including the boundary
`lbdm.touter`	total number of grid points in the domain including the boundary

Table 4.6: Neighbour information

parameter	**meaning**
`lbnb[k].rank`	processor name of neighbour `k`
`lbnb[k].spos`	start position for sending distribution function message
`lbnb[k].rpos`	start position for receiving distribution function message
`lbnb[k].bspos`	start position for sending boundary condition message
`lbnb[k].brpos`	start position for receiving boundary condition message
`k` $= 0$	right neighbour
`k` $= 1$	left neighbour
`k` $= 2$	upper neighbour
`k` $= 3$	lower neighbour
`k` $= 4$	front neighbour
`k` $= 5$	back neighbour

Table 4.7: Simulation information

parameter	**meaning**	**possible values**
`collide`	collision and forcing type	0 = BGK, 1 = BGK with Guo forcing, 2 = MRT, 3 = MRT with Guo-like forcing
`interact`	mesophase interaction type	0 = none, 1 = Shan/Chen, 2 = Shan/Chen with wetting, 3 = Lishchuk
`incompress`	incompressibility of fluids	0 = compressible fluids, 1 = incompressible fluids
`outtype`	output file type	0 = VTK, 1 = Legacy VTK, 2 = Plot3D

The notation column in Table 4.8 gives the restrictions applicable on the parameters. 'A' indicates an array of data, followed by the number of elements in the array. For example, 'A `lbsy.nf`' means the parameter is actually an array with `lbsy.nf` elements. '≥ 1' means the number must be greater or equal to one, while '1 or 0' means the value of the parameter can either be one or zero. An asterisk in the data type for the array indicates that it is allocatable.

Table 4.8: DL_MESO_LBE Parameters

function	**parameter**	**data type**	**notation**
system information	`lbsy`	structure	
domain information	`lbdm`	structure	
neighbour information	`lbnb`	structure	A 6
space dimension	`lbsy.nd`	int	
number of discrete speeds	`lbsy.nq`	int	
number of fluids	`lbsy.nf`	int	≥ 1
number of solutes	`lbsy.nc`	int	
temperature scalars	`lbsy.nt`	int	1 or 0
phase field order parameter	`lbsy.pf`	int	1 or 0
grid points in x-direction	`lbsy.nx`	int	
grid points in y-direction	`lbsy.ny`	int	
grid points in z-direction	`lbsy.nz`	int	
domain boundary width	`lbsy.bwid`	int	≥ 1
system dimension along x	`lbxsize`	double	
total calculation steps	`lbtotstep`	int	
equilibration calculation steps	`lbequstep`	int	
data save interval	`lbsave`	int	
current calculation step	`lbcurstep`	int	
steering	`lbsteer`	int	1 or 0
noise intensity	`lbnoise`	double	
evaporation limit	`lbevaplim`	double	
initial system velocity	`lbiniv`	double	A 3
top boundary velocity	`lbtopv`	double	A 3
bottom boundary velocity	`lbbotv`	double	A 3
front boundary velocity	`lbfrov`	double	A 3
back boundary velocity	`lbbakv`	double	A 3
left boundary velocity	`lblefv`	double	A 3
right boundary velocity	`lbrigv`	double	A 3
constant incompressible fluid density (ρ_0)	`lbincp`	double*	A `lbsy.nf`
initial fluid density	`lbinip`	double*	A `lbsy.nf`
top boundary fluid density	`lbtopp`	double*	A `lbsy.nf`
bottom boundary fluid density	`lbbotp`	double*	A `lbsy.nf`
front boundary fluid density	`lbfrop`	double*	A `lbsy.nf`
back boundary fluid density	`lbbakp`	double*	A `lbsy.nf`
left boundary fluid density	`lblefp`	double*	A `lbsy.nf`
right boundary fluid density	`lbrigp`	double*	A `lbsy.nf`
initial composition	`lbinic`	double*	A `lbsy.nc`
top boundary composition	`lbtopc`	double*	A `lbsy.nc`
bottom boundary composition	`lbbotc`	double*	A `lbsy.nc`
front boundary composition	`lbfroc`	double*	A `lbsy.nc`
back boundary composition	`lbbakc`	double*	A `lbsy.nc`
left boundary composition	`lblefc`	double*	A `lbsy.nc`
right boundary composition	`lbrigc`	double*	A `lbsy.nc`
initial temperature	`lbinit`	double	
top boundary temperature	`lbtopt`	double	
bottom boundary temperature	`lbbott`	double	
front boundary temperature	`lbfrot`	double	
back boundary temperature	`lbbakt`	double	
left boundary temperature	`lbleft`	double	
right boundary temperature	`lbrigt`	double	
system heating rate	`lbsysdt`	double	
top boundary heating rate	`lbtopdt`	double	

Table 4.8: DL_MESO_LBE Parameters (continued)

function	**parameter**	**data type**	**notation**
bottom boundary heating rate	`lbbotdt`	double	
front boundary heating rate	`lbfrodt`	double	
back boundary heating rate	`lbbakdt`	double	
left boundary heating rate	`lblefdt`	double	
right boundary heating rate	`lbrigdt`	double	
Boussinesq high temperature	`lbbousth`	double	
Boussinesq low temperature	`lbboustl`	double	
fluid inverse relaxation time (τ_f^{-1})	`lbtf`	double*	A `lbsy.nf`
bulk fluid inverse relaxation time ($\tau_{f,bulk}^{-1}$)	`lbtfbulk`	double*	A `lbsy.nf`
solute inverse relaxation time (τ_c^{-1})	`lbtc`	double*	A `lbsy.nc`
temperature inverse relaxation time (τ_t^{-1})	`lbtt`	double*	A `lbsy.nt`
fluid-fluid interaction parameter (g_{ab})	`lbg`	double*	A `lbsy.nf*lbsy.nf`
fluid-wall interaction parameter ($g_{a,wall}$)	`lbgwall`	double*	A `lbsy.nf`
fluid segregation parameter (β^{ab})	`lbseg`	double*	A `lbsy.nf*lbsy.nf`
body force	`lbbdforce`	double*	A `3*lbsy.nf`
Boussinesq force ($\vec{g}\beta$)	`lbbousforce`	double*	A `3*lbsy.nf`
interaction force	`lbinterforce`	double*	A `3*lbsy.nf*lbdm.touter`
distribution function	`lbf`	double*	A `lbsitelength*lbdm.touter`
temporary function	`lbft`	double*	A `lbdm.touter*(lbsy.nf+lbsy.nc)`
equilibrium distribution	`lbfeq`	double*	A `lbsy.nq`
space property	`lbphi`	int*	A `lbdm.touter`
neighbouring point property	`lbneigh`	int*	A `6*lbdm.touter`
speed vector for the model	`lbv`	int*	A `3*lbsy.nq`
index for opposing speed	`lbopv`	int*	A `lbsy.nq`
weight factor of speed vector	`lbw`	double*	A `lbsy.nq`
MRT transformation matrix	`lbtr`	double*	A `lbsy.nq*lbsy.nq`
MRT inverse transformation matrix	`lbtrinv`	double*	A `lbsy.nq*lbsy.nq`
MRT tuneable collision parameters	`lbmrts`	double	A 3
MRT tuneable equilibrium parameters	`lbmrtw`	double	A 3
number of parameters per grid point	`lbsitelength`	int	
number of grid points in yz plane	`lbyz`	int	
grid spacing	`lbdx`	double	
time step	`lbdt`	double	
speed of sound	`lbsoundv`	double	
Reynolds number	`lbreynolds`	double	
processor name	`lbdm.rank`	int	
total number of processors	`lbdm.size`	int	
x-coordinate for domain	`lbdm.xcor`	int	
y-coordinate for domain	`lbdm.ycor`	int	
z-coordinate for domain	`lbdm.zcor`	int	
number of processors along x-axis	`lbdm.xdim`	int	
number of processors along y-axis	`lbdm.ydim`	int	
number of processors along z-axis	`lbdm.zdim`	int	
x-coordinate of domain start position	`lbdm.xs`	int	
x-coordinate of domain end position	`lbdm.xe`	int	
y-coordinate of domain start position	`lbdm.ys`	int	
y-coordinate of domain end position	`lbdm.ye`	int	
z-coordinate of domain start position	`lbdm.zs`	int	
z-coordinate of domain end position	`lbdm.ze`	int	
inner[2] grid points along x	`lbdm.xinner`	int	

[2]Excluding boundary points

Table 4.8: DL_MESO_LBE Parameters (continued)

function	**parameter**	**data type**	**notation**
outer[3] grid points along x	`lbdm.xouter`	int	
inner grid points along y	`lbdm.yinner`	int	
outer grid points along y	`lbdm.youter`	int	
inner grid points along z	`lbdm.zinner`	int	
outer grid points along z	`lbdm.zouter`	int	
total grid points	`lbdm.touter`	int	
name of neighbouring processor	`lbnb[].rank`	int	
position for sending distribution message	`lbnb[].spos`	unsigned long	
position for receiving distribution message	`lbnb[].rpos`	unsigned long	
position for sending boundary message	`lbnb[].bspos`	unsigned long	
position for receiving boundary message	`lbnb[].brpos`	unsigned long	
position for sending force message	`lbnb[].fspos`	unsigned long	
position for receiving force message	`lbnb[].frpos`	unsigned long	
position for sending phase index message	`lbnb[].ispos`	unsigned long	
position for receiving phase index message	`lbnb[].irpos`	unsigned long	
message type	`lbmsg2x`	MPI_Datatype	
message type	`lbmsg2y`	MPI_Datatype	
message type	`lbmsg3x`	MPI_Datatype	
message type	`lbmsg3y`	MPI_Datatype	
message type	`lbmsg3z`	MPI_Datatype	
message type	`lbbmsg2x`	MPI_Datatype	
message type	`lbbmsg2y`	MPI_Datatype	
message type	`lbbmsg3x`	MPI_Datatype	
message type	`lbbmsg3y`	MPI_Datatype	
message type	`lbbmsg3z`	MPI_Datatype	
message type	`lbfmsg2x`	MPI_Datatype	
message type	`lbfmsg2y`	MPI_Datatype	
message type	`lbfmsg3x`	MPI_Datatype	
message type	`lbfmsg3y`	MPI_Datatype	
message type	`lbfmsg3z`	MPI_Datatype	
message type	`lbimsg2x`	MPI_Datatype	
message type	`lbimsg2y`	MPI_Datatype	
message type	`lbimsg3x`	MPI_Datatype	
message type	`lbimsg3y`	MPI_Datatype	
message type	`lbimsg3z`	MPI_Datatype	
endianness of system	`bigend`	int	
total calculation time	`totaltime`	double	
output file number	`qVersion`	int	
collision type parameter	`collide`	int	
interaction type parameter	`interact`	int	
incompressible fluid parameter	`incompress`	int	1 or 0
output file type	`outformat`	int	

[3]Including boundary points

Chapter 5

DL_MESO_LBE Features

5.1 Collision and Propagation Algorithms

The collision and propagation routines are a fundamental part of Lattice Boltzmann Equation calculations. Implementation involves the calculation of post-collisional values for the distribution functions at each lattice point (at time t^+). For the generalized form of the Lattice Boltzmann Equation with the collision operator C_i (normally in the form of a matrix):

$$f_i\left(\vec{x}, t^+\right) = f_i\left(\vec{x}, t\right) + C_i$$

and movement of these distribution functions to neighbouring lattice nodes:

$$f_i\left(\vec{x} + \hat{e}_i \Delta t, t + \Delta t\right) = f_i\left(\vec{x}, t^+\right)$$

which combine to give the governing equation for calculations.

5.1.1 Collision algorithms

The forms of collision currently available in DL_MESO_LBE are the Bhatnagar-Gross-Krook (BGK) single relaxation time[4] and Multiple Relaxation Time (MRT) schemes.

5.1.1.1 BGK single relaxation time

The BGK collision operator is defined by

$$C_i = -\frac{\Delta t}{\tau_f}\left(f_i\left(\vec{x}, t\right) - f_i^{eq}\right) \tag{5.1}$$

where τ_f is the relaxation time, related to the kinetic (shear) viscosity of fluid by

$$\nu = \frac{1}{3}\left(\tau_f - \frac{1}{2}\right)\frac{\left(\Delta x\right)^2}{\Delta t}$$

with the kinematic bulk viscosity of the fluid, ν', set equal to $\frac{2}{3}\nu$[7]. This reduces the equation for post-collisional distribution functions to

$$f_i\left(\vec{x}, t^+\right) = f_i\left(\vec{x}, t\right) - \frac{\Delta t}{\tau_f}\left(f_i\left(\vec{x}, t\right) - f_i^{eq}\right),$$

the same as Equation 3.12.

Interfacial and external forces are dealt with either by adding $\frac{\tau_f \vec{F}}{\rho}$ to the velocity of the fluid[41] when calculating the equilibrium distribution function f_i^{eq}, or by adding a forcing term to the collisional distribution function[19]:

$$F_i = \left(1 - \frac{1}{2\tau_f}\right) w_i \left[\frac{\hat{e}_i - \vec{v}}{c_s^2} + \frac{\left(\hat{e}_i \cdot \vec{v}\right)}{c_s^4}\hat{e}_i\right] \cdot \vec{F} \tag{5.2}$$

where $\vec{v}$ is defined as equal to $\vec{u} + \frac{\Delta t}{2\rho}\vec{F}$ and used in both the expression above and as the velocity for calculating equilibrium distribution functions. The former is used by default in `fCollisionBGK`, while the latter method by Guo *et al.* can be invoked using `fCollisionBGKGuo`.

5.1.1.2 Multiple Relaxation Time (MRT)

The MRT collision operator operates on a similar principle to the quasilinear Lattice Boltzmann Equation[25], which expresses collisions in terms of a square matrix with dimensions equal to the number of lattice links per grid point ($m \equiv$ `lbsy.nq`). Unlike quasilinear LBE, however, MRT collision schemes are applied to the moments for each lattice point rather than the distribution functions[33, 8], which are related to each other by

$$\vec{M} = \mathbf{T}\vec{f} \tag{5.3}$$

where $\vec{f}$ is a vector of all m distribution functions for the point, i.e. $(f_0, f_1 \dots f_{m-1})^T$, $\vec{M}$ the vector of moments (also of size m and dependent on the lattice system) and $\mathbf{T}$ the transformation matrix that renders the moments in terms of the distribution functions. Equilibrium values for the moments, $\vec{M}^{eq}$, can be determined by transforming the standard local equilibrium distribution function into moment space by

$$\vec{M}^{eq} = \mathbf{T}\vec{f}^{eq} \tag{5.4}$$

where $\vec{f}^{eq}$ is the vector of local equilibrium distribution functions: the resulting equilibrium moments can alternatively be expressed directly as functions of fluid density and velocity. Certain moments, such as density and momentum, must be conserved and their equilibrium values are set so that no changes are made. The post-collisional moments are determined by relaxation of the non-equilibrium part, i.e.

$$\vec{M}(\vec{x}, t^+) = \vec{M}(\vec{x}, t) - \mathbf{\Lambda}(\vec{M}(\vec{x}, t) - \vec{M}^{eq}(\vec{x}, t)) \tag{5.5}$$

where $\mathbf{\Lambda}$ is the collision matrix, which takes the form of a diagonal matrix of m collision parameters (represented by $\vec{s}$):

$$\mathbf{\Lambda} = diag(\vec{s}) \tag{5.6}$$

Some of the collision parameters can be specified by the user to set both kinematic and bulk viscosities, a few others can be tuned to improve simulation stability and the remainder (i.e. those for density and momentum) are fixed to conserve macroscopic hydrodynamics. If all values of s_i are set to $\frac{1}{\tau_f}$, the scheme reduces to BGK single relaxation time collisions. Since the collisional matrix is diagonal, equation 5.5 can be rewritten in terms of each moment, i.e.

$$M_i(\vec{x}, t^+) = M_i(\vec{x}, t) - s_i\left(M_i(\vec{x}, t) - M_i^{eq}(\vec{x}, t)\right) \tag{5.7}$$

Multiplying $\vec{M}(\vec{x}, t^+)$ by the inverse of the transformation matrix, $\mathbf{T}^{-1}$, gives the post-collisional distribution functions.

Interfacial and external forces can be applied either by the addition of $\tau_f\vec{F}$ to the fluid momentum[41] or by the use of a moment-transformed source term, $\vec{S}$, whose terms are defined by[49]:

$$S_i = w_i\left[\frac{\hat{e}_i - \vec{v}}{c_s^2} + \frac{(\hat{e}_i \cdot \vec{v})}{c_s^4}\hat{e}_i\right] \cdot \vec{F} \tag{5.8}$$

and applied by the following to correct the post-collisional moments:

$$\Delta\vec{M} = \left(\mathbf{I} - \tfrac{1}{2}\mathbf{\Lambda}\right)\vec{S}\Delta t \tag{5.9}$$

where $\mathbf{I}$ as an identity matrix. The above equation can be re-expressed as

$$\Delta M_i = \left(1 - \tfrac{1}{2}s_i\right)S_i\Delta t \tag{5.10}$$

The above equations after inverse transformation reduce to equation 5.2 when the collision parameters are set to $\frac{1}{\tau_f}$. All collision parameters for conserved moments should be set to unity when applying external forces.

An example can be given for the D2Q9 lattice system[33]; the moment vector is

$$\vec{M} = (\rho, e, \epsilon, j_x, q_x, j_y, q_y, p_{xx}, p_{xy})^T$$

with ρ as the density, e the energy, ϵ the square of energy, $\vec{j}$ momentum, $\vec{q}$ energy flux, p_{xx} the diagonal stress tensor component and p_{xy} the off-diagonal stress tensor component. The transformation matrix is

$$\mathbf{T} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -4 & 2 & -1 & 2 & -1 & 2 & -1 & 2 & -1 \\ 4 & 1 & -2 & 1 & -2 & 1 & -2 & 1 & -2 \\ 0 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 1 & -2 & 1 & 0 \\ 0 & 1 & 0 & -1 & -1 & -1 & 0 & 1 & 1 \\ 0 & 1 & 0 & -1 & 2 & -1 & 0 & 1 & -2 \\ 0 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 \end{bmatrix}$$

The equilibrium moment vector is

$$\vec{M}^{eq} = \left(\rho, -2\rho + \frac{3(j_x^2 + j_y^2)}{\rho}, w_\epsilon \rho + w_{\epsilon j}\frac{(j_x^2 + j_y^2)}{\rho}, j_x, -j_x, j_y, -j_y, \frac{j_x^2 - j_y^2}{3\rho}, \frac{j_x j_y}{3\rho}\right)^T$$

with w_ϵ and $w_{\epsilon j}$ as adjustable parameters: for convergence to the single relaxation time BGK scheme, these are set equal to 1 and -3 respectively. Of the 9 collision parameters available, s_0, s_3 and s_5 have no effect (except when applying external forces, when they should be set equal to one) as the associated moments are preserved and s_2, s_4 and s_6 are tuneable parameters for calculational stability with the condition that $s_4 = s_6$. The remaining parameters represent the viscosities of the fluid, i.e.

$$\begin{aligned} \nu &= \frac{1}{3}\left(\frac{1}{s_7} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t} = \frac{1}{3}\left(\frac{1}{s_8} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t} \equiv \frac{1}{3}\left(\tau_f - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t} \\ \nu' &= \frac{1}{6}\left(\frac{1}{s_1} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t} \equiv \frac{1}{6}\left(\tau_{f,bulk} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t} \end{aligned}$$

i.e. $\tau_f = \frac{1}{s_7} = \frac{1}{s_8}$ and $\tau_{f,bulk} = \frac{1}{s_1}$. If the moment-transformed source terms are to be used, the vector $\vec{S}$ for this lattice scheme is defined as

$$\vec{S} = \begin{pmatrix} 0 \\ 6(u_x F_x + u_y F_y) \\ -6(u_x F_x + u_y F_y) \\ F_x \\ -F_x \\ F_y \\ -F_y \\ 2(u_x F_x - u_y F_y) \\ u_x F_y + u_y F_x \end{pmatrix}.$$

Similar schemes are available for the D3Q15 and D3Q19 lattices (there is currently no MRT scheme available for D3Q27). The MRT schemes without source terms can be applied using `fCollisionMRT`, while the schemes with Guo-like moment-transformed source terms can be invoked using `fCollisionMRTGuo`.

5.1.2 Propagation algorithms

The simplest implementation, the *two-lattice algorithm*, involves the use of a temporary array (`lbft`) to copy post-collisional distribution functions to their new positions, which are subsequently copied back to the main distribution function array `lbf`. While this particular method is clear, easy to understand and can be applied throughout the system's lattice points in any order, its drawbacks include the use of two loops for propagation and array copying, two large arrays for the distribution functions at each lattice node and significant amounts of time expended in memory access. If the user wishes to use this method, the routine `fPropagationTwoLattice` is available.

An alternative, more efficient implementation of propagation is the *swap algorithm*[43], whereby this process is carried out by systematic swapping of pairs of collided distribution function values. To make this easier to implement, the lattice links are organised so that the conjugate link j to link i (i.e. $\hat{e}_j = -\hat{e}_i$) is equal to $i + \frac{m-1}{2}$ for $i = 1 \ldots \frac{m-1}{2}$. Looping i between 1 and $\frac{m-1}{2}$ the post-collisional distribution functions for each lattice point $f_i(\vec{x})$ are initially swapped with their conjugate values $f_j(\vec{x})$. then at each point the value $f_j(\vec{x})$ is then swapped with $f_i(\vec{x} + \hat{e}_i \Delta t)$.

These sets of swaps can be carried out either in two separate steps or in one go. The use of separate swap steps requires two sweeps through the domain, but the order in which the distribution functions are swapped does not matter and no boundary domain is necessary for serial calculations. Simultaneous swapping cannot make use of automatic periodic boundary conditions (thus a non-zero domain boundary is required) and requires lattice links to be additionally ordered so that the first half are all directed to lattice points that have previously gone through at least the first swap stage, but only a single sweep through the domain is required.

By default the serial version of DL_MESO_LBE uses the propagation routine that carries out two separate swap steps – `fPropagationSwap` – while the parallel version uses the combined-swap propagation routine `fPropagationCombinedSwap`. If the user wishes to use the combined-swap propagation routine in serial, an alternative program `slbecombine.cpp` is available which includes the following notable modifications to `slbecustom.cpp`:

- Replacement of `fSetSerialDomain` with `fSetSerialDomainBuffer`.
- Addition of `fsBoundPeriodic` and `fMarkBoundArea` immediately before `fInitializeSystem`.
- Addition of `fsPeriodic` inside main loop before calculating interaction forces (it does not matter whether this is placed before or immediately after the call for creating output files).
- Addition of `fsIndexPeriodic` inside main loop before calculating interaction forces for the Lishchuk algorithm (if these are not to be calculated, this call can be omitted).
- Addition of `fsForcePeriodic` inside main loop after calculating interaction forces (if these are not to be calculated, this call can be omitted).

5.2 Boundary conditions

To apply boundary conditions to a Lattice Boltzmann Equation simulation, the distribution functions f_i at boundary lattice points have to be modified or replaced during each time step to give the required fluid velocity or pressure/concentration/temperature. This may take place either between the collision and propagation stages or at the end of each time step: the subroutines `fPostCollBoundary` and `fPostPropBoundary` respectively are used to invoke the boundary conditions. The space property `lbphi` is used to define the boundary conditions for each lattice node in the system.

5.2.1 Periodic

Periodic boundaries are used to simulate bulk fluids sufficiently far away from the actual boundaries of a real physical system so that surface effects can be neglected. As the fluid moves out of one face of the system volume, it reappears on the opposite face with the same velocity, density etc.

DL_MESO_LBE applies periodic boundary conditions in two different ways depending on the size of the boundary domain `lbdm.bwid`. If there is no boundary domain (the default for serial running), periodic boundary conditions are automatically applied during the propagation step by using the function `fCppMod` to adjust the destination of distribution functions leaving the system so they are placed at the opposite side. No periodic boundary using this implementation needs to be defined by the space property, which can be left equal to zero as for the bulk fluid.

For systems that include a non-zero boundary domain size, the distribution functions at the edges of the actual system are copied into the buffer at the opposing sides before collisions and propagation take place. This requires the use of the space property (`lbphi[i]=10`) to determine the location of the domain buffer – which can be set up using the routine `fMarkBoundArea` – and either `fsPeriodic` and similar routines for serial running or `fNonBlockCommunication` and similar routines for running in parallel to copy the distribution functions into the buffers.

5.2.2 On-grid bounce-back

The on-grid bounce-back condition applies a no-slip condition (i.e. zero fluid velocity) at a boundary that lies halfway between grid points. This is applied after the propagation stage by reversing the distribution functions sitting on each wall node ($\vec{x}_w$), i.e.

$$f_i(\vec{x}_w, t) = f_j(\vec{x}_w, t) \tag{5.11}$$

where j is the opposite lattice link to i, i.e. $\hat{e}_j = -\hat{e}_i$. The reflection of distribution functions occurs *on-grid*. On-grid bounce-back is a first-order approximation of the boundary condition[45], i.e. the error is proportional to the lattice spacing Δx, but it is completely local (i.e. only uses distribution functions at the wall node).

5.2.3 Mid-grid bounce-back

Like the on-grid version, the mid-grid bounce-back condition applies a no-slip condition at a boundary halfway between lattice points[63]. This is applied by assigning post-collisional distribution functions to the wall node based on those values at neighbouring points, i.e.

$$f_i(\vec{x}_w, t^+) = f_j(\vec{x}_w + \hat{e}_i \Delta t, t^+). \tag{5.12}$$

This method essentially applies the actual reflection halfway between timesteps and is a spatially second-order method, although it is weakly non-local due to its use of distribution functions from neighbouring nodes.

5.2.4 Constant pressure/velocity

To specify either velocities or densities (pressures) at planar boundaries, the Zou-He method[73] is available in DL_MESO_LBE. This is based upon applying the bounce-back rule to the non-equilibrium distribution functions, i.e.

$$f_i^{(1)}(\vec{x}_w, t) = f_j^{(1)}(\vec{x}_w, t) \tag{5.13}$$

where $f_i^{(1)} = f_i - f_i^{eq}$, with the equilibrium distribution function f_i^{eq} as a function of density and velocity. This function can be used to determine the missing wall velocity or density along with the known distribution function values. For instance, for a top edge with a known velocity $\vec{v}_w$ using the D2Q9 lattice scheme (V??CEDF)[1], the wall density and missing distribution functions (all for the boundary node at $\vec{x}_w$) are given as:

$$\begin{aligned}
\rho_w &= \frac{f_0 + f_2 + f_6 + 2(f_1 + f_7 + f_8)}{1 + v_{w,y}} \\
f_4 &= f_8 - \frac{2\rho_w v_{w,y}}{3} \\
f_3 &= f_7 + \tfrac{1}{2}(f_6 - f_2) - \tfrac{1}{2}\rho_w v_{w,x} - \tfrac{1}{6}\rho_w v_{w,y} \\
f_5 &= f_1 - \tfrac{1}{2}(f_6 - f_2) + \tfrac{1}{2}\rho_w v_{w,x} - \tfrac{1}{6}\rho_w v_{w,y}
\end{aligned}$$

[1] '?' denotes any valid letters for the solute composition and temperature.

The other form, specifying the wall fluid density, requires the calculation of the wall velocity, which can be simplified by setting non-orthogonal velocity components to zero. For the analogous example at the top wall for D2Q9 (P??CEDF), the same equations for f_4, f_3 and f_5 can be used together with

$$\rho_w v_{w,x} = 0, \rho_w v_{w,y} = f_0 + f_2 + f_6 + 2(f_1 + f_7 + f_8) - \rho_w.$$

One complication for three-dimensional lattices is the requirement to apply the non-equilibrium bounce-back to all unknown distribution functions, which ordinarily overspecifies the system but can be counteracted using transverse momentum corrections for directions other than orthogonal to the boundary, which are non-zero for e.g. shearing flows. It should be noted that if the wall velocity is set to zero, the boundary condition reduces to on-grid bounce-back.

DL_MESO_LBE includes implementations of the Zou-He boundary condition for all four lattice schemes: D2Q9, D3Q15, D3Q19 and D3Q27. Only planar surfaces can be dealt with using this method; concave edges (in three-dimensions) and concave corners instead use the equilibrium distribution function for the given density/velocity and either zero velocity or the density at the nearest fluid grid point. An ‘evaporation limit’ is applied to the density as a minimum limit for constant velocity edges and corners to prevent spurious production of non-continuous fluids in multiple fluid systems: a default value of 10^{-8} is generally used but this can be overridden by the user in the `lbin.sys` file.

5.2.5 Constant solute concentration/temperature

To specify constant solute concentrations or temperatures at planar boundaries, the Inamuro method[28, 27] is used in DL_MESO_LBE. This is based upon substituting the unknown distribution functions for a boundary point with local equilibrium values, but using an adjusted solute concentration or temperature to produce the correct value for the property at that point. This concentration/temperature is obtained by substituting the equilibrium distribution function for the unknown populations into the sum of distribution functions (equal to the required concentration/temperature).

For example, for a top edge with a specified temperature T_w using the D2Q9 lattice scheme (??TCEDF), the adjustment temperature T' is given as:

$$T' = \frac{6}{1 - 3v_{w,y}} \left(T_w - h_0 - h_1 - h_2 - h_6 - h_7 - h_8\right)$$

and the missing populations ($i = 3, 4, 5$ in this case) are given by

$$h_i = w_i T' \left[1 + 3\frac{(\hat{e}_i \cdot \vec{v}_w)}{c^2}\right]$$

where $\vec{v}_w$ is the known (or calculated) velocity for all fluids at the same boundary.

DL_MESO_LBE includes implementations of the Inamuro boundary condition for all four lattice schemes. Only planar surfaces can be dealt with using this method; as for constant pressure boundaries, concave edges and corners use the equilibrium distribution function for the required concentration/temperature and the known or calculated velocity. On-grid bounce-back is used for Neumann (zero-gradient) conditions of solute concentrations and/or temperature to keep this type of boundary condition entirely local.

5.3 Mesoscale interactions

DL_MESO_LBE includes a number of algorithms that can be used to apply interactions between fluid components at the mesoscale, most commonly to model immiscible fluids. The user should take care to ensure the correct model is used for the type of system being modelled.

If `lbin.init` files are used to insert fluid drops into the simulation domain, DL_MESO_LBE includes the option of carrying out equilibration to allow the shapes of drops to settle by modelling the system without external body forces and boundaries imposing specific velocities, densities, solute concentrations or temperatures. This option can be selected using the `equilibration_step` keyword in the `lbin.sys` file.

5.3.1 Shan-Chen pseudopotential model

The Shan-Chen model[55, 56] models interactions between multiple phases and components by calculating pairwise interaction potentials. These potentials use an 'effective mass' for each component, ψ^a, which is a function of density and most frequently defined as

$$\psi^a(\vec{x}) = \rho_0^a \left[1 - \exp\left(-\frac{\rho^a(\vec{x})}{\rho_0^a}\right)\right] \tag{5.14}$$

where ρ^a is the local density of component a and ρ_0^a is the reference density for the same component. The function in this form can be used to apply phase separation for a single component, and can be changed by the user by modifying the subroutine `fCalcPotential_ShanChen`.

Defining g_{ab} as the interaction coefficient between components a and b, the overall force on component a due to interactions with other components is defined as

$$\vec{F}^a = -\psi^a(\vec{x}) \sum_b g_{ab} \sum_i w_i \psi^b(\vec{x} + \hat{e}_i)\, \hat{e}_i. \tag{5.15}$$

and any suitable forcing algorithm can be used to apply this force on a Lattice Boltzmann Equation simulation.

For a particular interaction the resulting equation of state[53] is defined as

$$P = \rho c_s^2 + \frac{1}{2} c_s^2 g_{ab} \psi^2(\rho) \tag{5.16}$$

and the interfacial tension between the components[54] is given as

$$\sigma_{ab} = -\frac{e_4 g_{ab} c_s^4 (\Delta x)^2}{2} \int_{-\infty}^{+\infty} \left(\frac{d\psi}{dz}\right)^2 dz \tag{5.17}$$

where e_4 is a lattice-dependent constant and z distance along the normal from the phase interface.

Optional fluid-solid interaction forces can be added[41] for each fluid to control its wetting:

$$\vec{F}_{wet}^a = -\psi^a(\vec{x})\, g_{a,wall} \sum_i w_i s(\vec{x} + \hat{e}_i)\, \hat{e}_i \tag{5.18}$$

where $s(\vec{x}) = 0$ for a pore (fluid site) at position $\vec{x}$ and $s(\vec{x}) = 1$ for a solid site. The interaction coefficient between component a and the wall, $g_{a,wall}$, can be given a positive (negative) value to reduce (increase) its wetting.

5.3.2 Lishchuk continuum-based model

The Lishchuk model[35, 20] is a modified form of the 'chromodynamic' model devised by Gunstensen and Rothman[17], which models interactions between multiple components by applying forces based on the existence of those components. This continuum-based model is primarily suited for systems where hydrodynamic interactions dominate and no further fluid separation takes place.

A phase field is defined between components a and b as

$$\rho_{ab}^N = \frac{\rho^a - \rho^b}{\rho^a + \rho^b}. \tag{5.19}$$

noting that $\rho_{ba}^N = -\rho_{ab}^N$. First-order spatial differentials of this quantity can be determined by means of fourth-order accurate isotropic schemes[21], e.g. for lattice points without neighbouring walls:

$$\nabla \rho_{ab}^N = \frac{1}{c_s^2 \Delta t} \sum_i w_i \hat{e}_i \rho_{ab}^N(\vec{x} + \hat{e}_i \Delta t) \tag{5.20}$$

and these can be used to determine the interfacial normal between the phases

$$\hat{n}_{ab} = \frac{\nabla \rho^N_{ab}}{\left|\nabla \rho^N_{ab}\right|} \tag{5.21}$$

which can subsequently be used to obtain the local curvature from the interface normal in the phase field

$$K_{ab} = -\nabla_S \cdot \hat{n}_{ab}. \tag{5.22}$$

The force acting between the components to give interfacial tension is given by

$$\vec{F}^{ab} = \frac{1}{2} g_{ab} K_{ab} \nabla \rho^N_{ab}. \tag{5.23}$$

Rather than colliding each fluid separately, a single 'achromatic' distribution function is defined as the sum of distribution functions for all fluids

$$f_i = \sum_a f_i^a \tag{5.24}$$

the sum of which is equal to the density of all fluids, ρ. This distribution function is collided using any valid method with all interaction forces combined together and collision operators interpolated according to local mass fraction, e.g.

$$\frac{1}{\tau_f} = \frac{1}{\rho} \sum_a \frac{\rho^a}{\tau_f^a}. \tag{5.25}$$

After the achromatic fluid is collided, the fluids are segregrated to produce post-collisional distribution functions for each fluid. This is achieved using the D'Ortona algorithm[9], which gives a non-zero boundary thickness between the fluids and reduces non-physical effects such as pinning of drops to the lattice, spatial anisotropy in interfacial tension and spurious microcurrents. The equation for the post-collisional segregated distribution function for fluid a[61] is given as

$$f_i^a\left(\vec{x}, t^+\right) = \frac{\rho^a}{\rho} f_i\left(\vec{x}, t^+\right) + \sum_{b \neq a} \beta^{ab} w_i \frac{\rho^a \rho^b}{\rho^2} \hat{e}_i \cdot \hat{n}_{ab} \tag{5.26}$$

where β^{ab} is a segregation parameter that controls the width of the diffuse boundary between phases and $\beta^{ba} = \beta^{ab}$.

This method can directly simulate a specified interfacial tension (σ_{ab}), which is related to the lattice-based parameter g_{ab} by

$$\sigma_{ab} = \frac{4 g_{ab} \nu^2 \rho_0}{c_s^4 \left(2\tau_f - 1\right)^2 \Delta x} \tag{5.27}$$

using the mean density (ρ_0), kinematic viscosity (ν) and relaxation time (τ_f) of a reference fluid (e.g. the continuous fluid for the system).

The subroutine `fCalcPhaseIndex_Lishchuk` calculates and stores the first derivatives of the phase indices, which are subsequently used for force calculations – values from neighbouring lattice points are required for calculations of interface curvature and the additional `fIndexBlockCommunication` routine will be required for parallel running – and for postcollisional fluid segregation. The `fCollision*Segregation` routines, used in place of the standard collision routines, carry out the achromatic collision of all fluids and D'Ortona segregation. No solid-fluid interactions (i.e. wetting effects) are currently available in DL_MESO_LBE using this form of mesoscopic interaction.

5.4 Diffusion and heat transfer

In a similar fashion to multiple fluid systems, the Lattice Boltzmann Equation method can be applied to problems involving diffusion and/or heat transfers by using additional distribution functions for each solute and/or temperature[27, 72].

For a system consisting of a number of dilute solutes along with a bulk fluid, the governing equation for each solute is given as

$$g_i\left(\vec{x}+\hat{e}_i\Delta t, t+\Delta t\right)-g_i\left(\vec{x},t\right)=-\frac{\Delta t}{\tau_s}\left[g_i\left(\vec{x},t\right)-g_i^{eq}\right] \tag{5.28}$$

where g_i is the distribution function for the solute and τ_s the solute relaxation time, which is related to its diffusivity

$$D=\frac{1}{3}\left(\tau_s-\frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t}$$

and the Schmidt number can be determined by

$$\text{Sc}=\frac{\nu}{D}=\frac{2\tau_f-1}{2\tau_s-1}.$$

Taking the concentration of the solute as $C_s=\sum_i g_i$, the equilibrium distribution function for the solute is given by a simpler form of Equation 3.11:

$$g_i^{eq}=w_i C_s\left[1+3\frac{(\hat{e}_i\cdot\vec{u})}{c^2}\right] \tag{5.29}$$

where the velocity used is that of the bulk fluid.

Heat transfers can be coupled to the system in a similar manner, using a thermal distribution function h_i and a thermal relaxation time τ_t, which gives the governing equation

$$h_i\left(\vec{x}+\hat{e}_i\Delta t, t+\Delta t\right)-h_i\left(\vec{x},t\right)=-\frac{\Delta t}{\tau_t}\left[h_i\left(\vec{x},t\right)-h_i^{eq}\right] \tag{5.30}$$

The temperature at each lattice point (relative to a mean value) can be determined as the sum of the distribution functions, $T=\sum_i h_i$, which can be used to determine the equilibrium distribution function

$$h_i^{eq}=w_i T\left[1+3\frac{(\hat{e}_i\cdot\vec{u})}{c^2}\right], \tag{5.31}$$

again using the bulk fluid velocity. The thermal relaxation time is related to the thermal diffusivity

$$\alpha=\frac{1}{3}\left(\tau_t-\frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t}$$

with the Prandtl number for the system determined by a ratio of relaxation times, i.e.

$$\text{Pr}=\frac{\nu}{\alpha}=\frac{2\tau_f-1}{2\tau_t-1}.$$

It should be noted that if multiple-relaxation-time (MRT) schemes are to be used, these only apply to fluids: all diffusion and heat transfer processes are calculated using the single relaxation time schemes described in this section.

5.4.1 Boussinesq approximation

The coupling of fluid flows to heat transfer described above only produces heat conduction effects. To model convective heat transfer processes, an additional force on the fluid is required to link flow to thermal transport. The most common form is the Boussinesq approximation[18], which applies a buoyancy force on fluid a proportional to the temperature difference:

$$\vec{F}^a=-\rho\vec{g}\beta^a\left(\frac{T-T_0}{T_h-T_l}\right) \tag{5.32}$$

where $\vec{g}$ is gravitational acceleration, β^a is the volumetric expansion coefficient for fluid a, T_h and T_l are respectively the maximum and minimum temperatures of the system and $T_0=\frac{1}{2}(T_h+T_l)$ is a reference temperature.

DL_MESO_LBE provides the routine `fConvectionForceBoussinesq` to calculate this force. The product of gravitational acceleration and volumetric expansion ($\vec{g}\beta^a$) for each fluid, as well as the maximum and minimum temperatures T_h and T_l, can be included in the `lbin.sys` file.

5.5 Compressible and incompressible fluids

The standard Lattice Boltzmann Equation scheme is capable of modelling compressible fluids. Incompressible fluids can be modelled by making a simple modification to the local equilibrium distribution function[24]:

$$f_i^{eq} = w_i \left[\rho + \rho_0 \left(\frac{3\left(\hat{e}_i \cdot \vec{u}\right)}{c^2} + \frac{9\left(\hat{e}_i \cdot \vec{u}\right)^2}{2c^4} - \frac{3u^2}{2c^2} \right) \right] \tag{5.33}$$

where ρ_0 is the fixed density of the incompressible fluid and the density ρ becomes an analogue to pressure ($\frac{P}{\rho_0} = c_s^2 \rho$). While Equation 3.1 is still applicable to calculate ρ, the fluid velocity is now calculated by

$$\rho_0 u_\alpha = \sum_{i=0}^{q} f_i e_{i\alpha} \tag{5.34}$$

DL_MESO_LBE allows users to select incompressible fluids by means of the variable `incompress`, whose value can be selected using the keyword `incompressible_fluids` in the `lbin.sys` file. Additional collision and equilibrium distribution function routines ending in `Incom` are included to allow the user to model incompressible fluids. All of these routines use a specified constant density for each fluid in the system (`lbincp`) as the value of ρ_0.

Chapter 6

DL_MESO_LBE Input and Output Files

6.1 Input files

All user-specified input files for DL_MESO_LBE must be in ANSI text format, with keywords (where necessary) and numerical values separated from each other with spaces or tabs.

Define system: `lbin.sys`

The use of the DL_MESO GUI is recommended for producing `lbin.sys`, although existing files of that name can also be edited. Its format consists of a keyword and an associated numerical parameter on each line separated by spaces or tabs. No allowances are made for typographical errors or abbreviations in keywords, which must be included in full and in the form described below.

Ten keywords are compulsory for all LBE simulations, as these determine the lattice scheme to be used, the number of lattices to use, and the sizes of the system and boundary regions.

keyword:	**meaning:**
space_dimension	sets the number of dimensions in the system (2 or 3)
discrete_speed	sets the number of lattice links per grid point (9, 15, 19 or 27)
number_of_fluid	sets the number of fluid lattices for the system (if modelling solutes, this must be set to 1)
number_of_solute	sets the number of solutes to be modelled
temperature_scalar	determines whether or not a lattice is needed to model heat transfers (set to 1 if needed, 0 if not)
phase_field	determines whether or not a lattice is needed to represent phase fields (set to 1 if needed, 0 if not)[1]
grid_number_x	sets the number of grid points in the x-dimension
grid_number_y	sets the number of grid points in the y-dimension
grid_number_z	sets the number of grid points in the z-dimension (if a two-dimensional system is modelled, this will be reset to 1)
domain_boundary_width	sets the size of the boundary region (if running DL_MESO in serial, this is usually reset to 0)

Additional keywords can be used to specify the algorithms for collisions, forcing and mesophase interactions, the format for output files and whether fluids are compressible or incompressible. If these are omitted, DL_MESO_LBE will assume that the fluids are compressible and subjected to BGK (single-relaxation-

[1]No multiple fluid phase scheme included in DL_MESO currently requires this lattice.

time) collisions using standard forcing without mesophase interactions. (If using the customizable versions of DL_MESO_LBE, all of these keywords may be omitted except for `incompressible_fluids`, which is required to correctly calculate fluid velocities in initialization and output files and apply boundary conditions.)

keyword:	**meaning:**
collision_type	sets the type of collisions and forcing (`BGK` (0), `BGKGuo` (1), `MRT` (2) or `MRTGuo` (3)[2])
interaction_type	sets the type of mesophase interactions[3] (`ShanChen` (1), `ShanChenWetting` (2) or `Lishchuk` (3))
output_format	sets the format for output files (`VTK` (0), `LegacyVTK` (1), `Plot3D` (2))
incompressible_fluids	determines whether or not the fluids should be incompressible (set to 0 for compressible fluids, 1 for incompressible fluids)

The following keywords can be used to specify other information, such as fluid densities, velocities, relaxation times or frequencies etc. Superfluous parameters can be omitted, while new ones would require additions to the parameter recognition loop in the `fInputParameters` subroutine in `lbpIO`. Note that if there are duplicate entries for any keyword, the value associated with the last one in the `lbin.sys` file will be used.

keyword:	**meaning:**
total_step	sets total number of timesteps for the simulation
equilibration_step	sets number of timesteps for equilibration of the simulation (without solid boundary conditions or external forcing)
save_span	sets interval for writing output files
noise_intensity	gives maximum variation in initial fluid densities for multiple fluid systems
evaporation_limit	gives minimum fluid density for non-continuous fluids when dealing with edge or corner boundaries
sound_speed	sets speed of sound for fluid 0 in real-life (i.e. non-lattice-based) units
kinetic_viscosity	sets kinematic viscosity for fluid 0 in real-life units
total_step	sets total number of timesteps for the simulation
speed_ini_n	sets initial velocity for all fluids ($n = 0$ for x-component, $n = 1$ for y-component, $n = 2$ for z-component)
speed_top_n	sets velocity at top boundary for all fluids
speed_bot_n	sets velocity at bottom boundary for all fluids
speed_lef_n	sets velocity at left boundary for all fluids
speed_rig_n	sets velocity at right boundary for all fluids
speed_fro_n	sets velocity at front boundary for all fluids
speed_bac_n	sets velocity at back boundary for all fluids
density_ini_f	sets initial density for fluid f throughout system (f between 0 and `lbsy.nf` -1)
density_inc_f	sets constant density for incompressible fluid f
density_top_f	sets density for fluid f at top boundary
density_bot_f	sets density for fluid f at bottom boundary
density_lef_f	sets density for fluid f at left boundary
density_rig_f	sets density for fluid f at right boundary
density_fro_f	sets density for fluid f at front boundary
density_bac_f	sets density for fluid f at back boundary
relaxation_fluid_f	sets relaxation time (τ_f) for fluid f
relax_freq_fluid_f	sets relaxation frequency (τ_f^{-1}) for fluid f

[2]Either the keyword or the number can be used to specify the types.
[3]If set to an unrecognised word or to 0, interactions will be switched off.

bulk_relaxation_fluid_f	sets bulk relaxation time ($\tau_{f,bulk}$) for fluid f
bulk_relax_freq_fluid_f	sets bulk relaxation frequncy ($\tau_{f,bulk}^{-1}$) for fluid f
solute_ini_s	sets initial concentration for solute s throughout system (s between 0 and `lbsy.nc` -1)
solute_top_s	sets concentration for solute s at top boundary
solute_bot_s	sets concentration for solute s at bottom boundary
solute_lef_s	sets concentration for solute s at left boundary
solute_rig_s	sets concentration for solute s at right boundary
solute_fro_s	sets concentration for solute s at front boundary
solute_bac_s	sets concentration for solute s at back boundary
relax_solute_s	sets relaxation time (τ_s) for solute s
relax_freq_solute_s	sets relaxation frequency (τ_s^{-1}) for solute s
temperature_ini	sets initial temperature throughout system
temperature_top	sets temperature at top boundary
temperature_bottom	sets temperature at bottom boundary
temperature_left	sets temperature at left boundary
temperature_right	sets temperature at right boundary
temperature_front	sets temperature at front boundary
temperature_back	sets temperature at back boundary
heating_rate_sys	sets rate of change in temperature (with time based on real-life units) throughout system
heating_rate_top	sets rate of change in temperature at top boundary
heating_rate_bottom	sets rate of change in temperature at bottom boundary
heating_rate_left	sets rate of change in temperature at left boundary
heating_rate_right	sets rate of change in temperature at right boundary
heating_rate_front	sets rate of change in temperature at front boundary
heating_rate_back	sets rate of change in temperature at back boundary
relax_thermal	sets thermal relaxation time (τ_t)
relax_freq_thermal	sets thermal relaxation frequency (τ_t^{-1})
body_force_n	sets constant external body force on fluid f: $n = 3f$ for x-component, $n = 3f+1$ for y-component, $n = 3f + 2$ for z-component
body_force_x_f	sets x-component of constant external body force on fluid f
body_force_y_f	sets y-component of constant external body force on fluid f
body_force_z_f	sets z-component of constant external body force on fluid f
boussinesq_force_n	sets Boussinesq force constant ($\vec{g}\beta$) for fluid f: $n = 3f$ for x-component, $n = 3f + 1$ for y-component, $n = 3f + 2$ for z-component
boussinesq_force_x_f	sets x-component of Boussinesq force constant ($\vec{g}\beta$) for fluid f
boussinesq_force_y_f	sets y-component of Boussinesq force constant ($\vec{g}\beta$) for fluid f
boussinesq_force_z_f	sets z-component of Boussinesq force constant ($\vec{g}\beta$) for fluid f
boussinesq_boussinesq_high	sets high reference temperature for Boussinesq convection (T_h)
boussinesq_boussinesq_low	sets low reference temperature for Boussinesq convection (T_l)
interaction_n	sets interaction parameter between fluids f_1 and f_2: $n =$ `lbsy.nf` $\times f_1 + f_2$
interaction_f_1_f_2	sets interaction parameter between fluids f_1 and f_2
segregation	sets fluid segregation parameter between all fluids species
segregation_n	sets fluid segregation parameter between fluids f_1 and f_2: $n =$ `lbsy.nf` $\times f_1 + f_2$
segregation_f_1_f_2	sets fluid segregation parameter between fluids f_1 and f_2

Define space: `lbin.spa`

The GUI is recommended for creating the `lbin.spa` file, which stores the data in the following format:

```
x,y,z,grid property
```

An empty `lbin.spa` file represents all boundaries as periodic.

Define initial condition: `lbin.init`

This optional file cannot currently be created by the GUI: the user must create this file or use the utility `lbeinitcreate` if it is required. The following format is required for each lattice point whose default velocity, fluid densities, solute concentrations or temperature needs replacing:

$$\texttt{x,y,z,u}_x\texttt{,u}_y\texttt{,u}_z\texttt{,}\rho^0\ldots\rho^{\texttt{lbsy.nf}-1}\texttt{,}c^0\ldots c^{\texttt{lbsy.nc}-1}\texttt{,}T$$

Note that three values for each grid position and velocity must be included (the values for z-components in two-dimensional simulations must be set to zero). At each grid point specified, density/concentration/temperature values must be included for *all* lattices used in calculations: the total number of values in each line must be equal to $6 + \texttt{lbsy.nf} + \texttt{lbsy.nc} + \texttt{lbsy.nt}$.

6.2 Output files

DL_MESO_LBE prints information about the simulation to the screen or standard output:

- welcome messages
- a description of the simulation to be carried out
- details of domain decomposition if running in parallel
- reports on the masses and momentum of fluids in the system at user-specified intervals
- a final summary including a calculation efficiency measure and a reminder to cite DL_MESO for any published results

This information can be directed to a file specified at the command line, e.g. by launching DL_MESO_LBE using the command

- `./lbe.exe > OUTPUT`

Snapshots of the simulation can be written in Structured Grid XML VTK, Structured Grid Legacy VTK and standard Plot3D data format, in binary format for parallel calculations and ANSI for serial. These may be modified by the user as required. The utility `lbeplot3dgather` in the `LBE/utility` directory can combine Plot3D files generated in parallel, while Parallel Structured Grid XML VTK files (`lbtout*.pvts`) that refer to the files from each processor can be created using the utility `lbevtkgather`: further details can be found in Appendix B or the `README` file in the same directory. By default *all* properties for a simulation – fluid densities, mass fractions, solute concentrations and temperatures – are written to each output file (or to individual output files for Plot3D for each property). The customizable version of DL_MESO_LBE allows users to select which properties should be written to output files.

Structured Grid XML VTK format: `lbout*.vts`

Structured Grid VTK files written by DL_MESO_LBE include the lattice dimensions (numbers of grid points in each direction), the Cartesian coordinates of the grid points in real-life units, and the same data as in Legacy VTK files. Output files produced in serial include the data between XML tags, e.g. `<DataArray>`, while those produced in parallel use the `<DataArray>` tags to refer to the starting point for the data in a stream of binary numbers inside an `<AppendedData>` tag. The latter files represent the data in each subdomain and should be retained when plotting the entire system since the parallel VTK format links to these rather than creates autonomous files for the entire system.

Legacy VTK format: `lbout*.vtk`

Legacy VTK files written by DL_MESO_LBE include the lattice dimensions (numbers of grid points in each direction), the Cartesian coordinates of the grid points in real-life units, and the following data:

- A scalar property (fluid density, mass fraction, solute concentration or scalar temperature), e.g.

$$\begin{array}{l} \rho_{0,0,0} \\ \ldots \\ \rho_{nx-1,ny-1,nz-1} \end{array}$$

- Fluid velocity

$$\begin{array}{l} U_{0,0,0} \;\; V_{0,0,0} \;\; W_{0,0,0} \\ \ldots \\ U_{nx-1,ny-1,nz-1} \;\; V_{nx-1,ny-1,nz-1} \;\; W_{nx-1,ny-1,nz-1} \end{array}$$

- The space (boundary condition) property

$$\begin{array}{l} \phi_{0,0,0} \\ \ldots \\ \phi_{nx-1,ny-1,nz-1} \end{array}$$

If all properties are to be output, they are all included in the same file for each time step under unique names.

Plot3D format

Output grid position: `lbout*.xyz`

$$\begin{array}{l} nx, ny, nz \\ x_{0,0,0}, \cdots x_{nx-1,ny-1,nz-1} \\ y_{0,0,0}, \cdots y_{nx-1,ny-1,nz-1} \\ z_{0,0,0}, \cdots z_{nx-1,ny-1,nz-1} \end{array}$$

where nx is the total number of grid points in x-direction, ny is the total number of grid points in y-direction, nz is the total number of grid points in z-direction and $(x_{i,j,k}, y_{i,j,k}, z_{i,j,k})$ is the Cartesian coordinate of grid point (i, j, k).

Output macroscopic quantities: `lbout.q`

$$nx, ny, nz$$
$$c, 1.0, Re, t$$
$$\rho_{0,0,0}, \dots \rho_{nx-1,ny-1,nz-1}, U_{0,0,0}, \dots U_{nx-1,ny-1,nz-1}$$
$$V_{0,0,0}, \dots V_{nx-1,ny-1,nz-1}, W_{0,0,0}, \dots W_{nx-1,ny-1,nz-1}$$
$$\phi_{0,0,0}, \dots \phi_{nx-1,ny-1,nz-1}$$

where c is the speed of sound for the lattice, Re the Reynolds number for the flow, t the time step, ρ the density, U the x-component of velocity, V the y-component of velocity, W the z-component of velocity and ϕ the space (boundary condition) property. (The 1.0 between the lattice speed of sound and flow Reynolds number represents the freestream angle of attack.) The density of the fluid may be replaced with its concentration or scalar temperature.

If all properties are output, each property is given a uniquely named file based on the number of fluid or solute (e.g. `lbout00dens*.q` for the density of fluid 0) and its property (`lbout*dens*.q`, `lbout*frac*.q`, `lbout*conc*.q` and `lbouttemp*.q`).

Chapter 7

DL_MESO_LBE Package Reference

7.1 Overview

DL_MESO_LBE consists of nine packages:

- `lbpMODEL`
  Contains subroutines to assign `lbw`, `lbv` and `lbopv` for D2Q9, D3Q15, D3Q19 and D3Q27 lattice models.
- `lbpBASIC`
  Contains general purpose subroutines which can be used elsewhere, e.g. a random number generator producing values between -1 and 1.
- `lbpGET`
  Contains subroutines to calculate macroscopic quantities, e.g. macroscopic density, speed and momentum.
- `lbpIO*`
  Contain subroutines to read parameters and write numerical results for plotting and visualization:
  - `lbpIO`
    Contains subroutines to read input files, calculate and write summaries.
  - `lbpIOPlot3D`
    Contains subroutines to write calculation output files in Plot3D format.
  - `lbpIOLegacyVTK`
    Contains subroutines to write calculation output files in Legacy VTK (structured grid) format.
  - `lbpIOVTK`
    Contains subroutines to write calculation output files in XML VTK (structured grid) format.
- `lbpBOUND`
  Contains subroutines for boundary conditions, e.g. calculating the particle distribution function in a shear boundary.
- `lbpFORCE`
  Contains subroutines to calculate non-constant forces, e.g. immiscible fluid-fluid interactions.
- `lbpSUB`
  Contains the most important subroutines for Lattice Boltzmann calculations, e.g. particle propagation and site collision.
- `lbpRUN*`
  Contain the major program loops to carry out Lattice Boltzmann calculations (`lbpRUNSER` for serial running, `lbpRUNPAR` for parallel).

- `lbpMPI`
  Contains all subroutines necessary for parallel computation. This package can be left out if the user uses only a single-processor workstation or a Windows PC.

It is recommended that DL_MESO users put self-defined subroutines into a package called `lbpUSER` so upgrades of DL_MESO will not interfere with their contributions.

7.2 DL_MESO_LBE Subroutines and Functions

7.2.1 main

There are two primary versions of the main DL_MESO_LBE program: serial (`slbe.cpp`) and parallel (`plbe.cpp`). These provide calls to the main loops for Lattice Boltzmann calculations in `lbpRUN*` and allow the use of input files to select collision, forcing and mesophase interaction algorithms, as well as output file formats and whether fluids are compressible or incompressible. These programs do not need to be modified by the user if the provided code features are to be used.

Alternative versions of the program, `slbecustom.cpp` and `plbecustom.cpp` for serial and parallel running respectively, are also provided. The user may wish to modify these listings to use user-defined routines and functions, as well as different outputs and formats, alternative collision routines, boundary conditions that move through the system etc. These versions reduce the number of logic statements required to run and thus may be used if greater computational efficiency is required. An additional version of the serial program is available which uses a boundary layer of lattice points (`slbecombine.cpp`). Appendix A gives more details on using these versions of the program.

7.2.2 lbpMODEL

D2Q9

- **Header records**
  `int D2Q9()`

- **Function**
  Assign the weight factor `lbw`, speed vector `lbv`, index for opposite speed vector `lbopv`, MRT transformation matrices (`lbtr` and `lbtrinv`) and tuneable parameters (`lbmrts` and `lbmrtw`) for the D2Q9 lattice model.

- **Dependencies**
  None

D3Q15

- **Header records**
  `int D3Q15()`

- **Function**
  Assign the weight factor `lbw`, speed vector `lbv`, index for opposite speed vector `lbopv`, MRT transformation matrices (`lbtr` and `lbtrinv`) and tuneable parameters (`lbmrts` and `lbmrtw`) for the D3Q15 lattice model.

- **Dependencies**
  None

D3Q19

- **Header records**
  `int D3Q19()`
- **Function**
  Assign the weight factor `lbw`, speed vector `lbv`, index for opposite speed vector `lbopv`, MRT transformation matrices (`lbtr` and `lbtrinv`) and tuneable parameters (`lbmrts` and `lbmrtw`) for the D3Q19 lattice model.
- **Dependencies**
  None

D3Q27

- **Header records**
  `int D3Q27()`
- **Function**
  Assign the weight factor `lbw`, speed vector `lbv` and the index for opposite speed vector `lbopv` for the D3Q27 lattice model.
- **Dependencies**
  None
- **Comments**
  No Multiple-Relaxation-Time (MRT) scheme is currently available for this lattice.

7.2.3 lbpBASIC

This package contains general purpose functions which are not directly related to the Lattice Boltzmann Equation method. These may be replaced with any suitable functions in C++ standard libraries.

fCppAbs

- **Header records**
  ```
  template <class T>
  T fCppAbs(T a)return (a<0)?-a:a;
  ```
- **Function**
  Calculate absolute value of number `a`.
- **Dependencies**
  None
- **Arguments**

`a`	input	any datatype
`fCppAbs`	output	same datatype as `a`

fReciprocal

- **Header records**
  ```
  template <class T>
  T fReciprocal(T a)return (a!=0)?1/a:0;
  ```

- **Function**
  Calculates reciprocal of `a` for all non-zero values; returns zero for `a = 0`.
- **Dependencies**
  None
- **Arguments**

`a`	input	any datatype
`fReciprocal`	output	same datatype as `a`

fEvapLimit

- **Header records**
```
template <class T>
T fEvapLimit(T a)return (a<lbevaplim)?0:a;
```
- **Function**
  Returns zero for values of a less than the specified evaporation limit `lbevaplim`.
- **Dependencies**
  None
- **Arguments**

`a`	input	any datatype
`fEvapLimit`	output	same datatype as `a`

- **Comments**
  This function is used to eliminate spurious production of non-continuous fluids at boundary edges and corners. The default value of `lbevaplim` (10^{-8}) can be overridden by the user.

fSwapPair

- **Header records**
```
template <class T> void fSwapPair ( T& a, T& b)
```
- **Function**
  Swaps pair of numbers, `a` and `b`.
- **Dependencies**
  None
- **Arguments**

`a`	input/output	any datatype
`b`	input/output	any datatype

fGetNumberOrdered

- **Header records** (two cases)
```
int fGetNumberOrdered(int &iox, int &ioy, int &ioz)
int fGetNumberOrdered(int &iox, int &ioy)
```
- **Function**
  Rearrange the integers in descending order.
- **Dependencies**
  None

- **Arguments**

`iox`	input/output	integer reference
`ioy`	input/output	integer reference
`ioz`	input/output	integer reference

fGetNumberOrderFixed

- **Header records** (two cases)
  `int fGetNumberOrderFixed(int &iox, int &ioy, int &ioz, int ix, int iy, int iz)`
  `int fGetNumberOrderFixed(int &iox, int &ioy, int ix, int iy)`
- **Function**
  Rearrange a set of integers so they appear in the same order as another set of integers.
- **Dependencies**
  `fGetNumberOrdered`
- **Arguments**

`iox`	input/output	integer reference
`ioy`	input/output	integer reference
`ioz`	input/output	integer reference
`ix`	input	integer
`iy`	input	integer
`iz`	input	integer

fBestGrouping

- **Header records**
  `int fBestGrouping(int totalgrid, int totalgroup, int& indigrid, int& critigroup)`
- **Function**
  Distribute grid points among the processes to give a maximum of one to the differences in the numbers of grid points.
- **Dependencies**
  None
- **Arguments**

`totalgrid`	input	integer
`totalgroup`	input	integer
`indigrid`	output	integer reference
`critigroup`	output	integer reference

- **Comments**
  The `totalgrid` grid points are distributed among `totalgroup` processes so that the first `critigroup` processes have `indigrid` grid points and the others have `indigrid-1`.

fCppMod

- **Header records** (two cases)
  `int fCppMod(int a, int b)`
  `long fCppMod(long a, long b)`
- **Function**
  Ensure that `a` is in a range between `0` and `b-1`, so that the value beyond the maximum value equals the minimum, and vice versa.

- **Dependencies**
  None

- **Arguments**

`a`	input	integer/long integer
`b`	input	integer/long integer
`fCppMod`	output	integer/long integer

- **Comments**
  `fCppMod = a-b` when `a >= b` or `fCppMod = a+b` when `a < 0`. This function is useful for periodic boundary conditions.

fPrintLine

- **Header records**
  `int fPrintLine()`

- **Function**
  Prints a line of 76 `-` characters.

- **Dependencies**
  None

fPrintDoubleLine

- **Header records**
  `int fPrintDoubleLine()`

- **Function**
  Prints a line of 76 `=` characters.

- **Dependencies**
  None

fRandom

- **Header records**
  `double fRandom()`

- **Function**
  Creates a double precision random number between -1 and 1.

- **Dependencies**
  None

- **Arguments**

`fRandom`	output	double precision

- **Comments**
  There is a built-in seed in the function, which is only activated when the function is initially called.

fBigEndian

- **Header records**
  `int fBigEndian()`
- **Function**
  Determines endianness for machine: returns 1 for big endian, 0 for little endian.
- **Dependencies**
  None

fByteSwap

- **Header records**
  `void fByteSwap(void *data, int len, int count)`
- **Function**
  Converts between endian types by swapping byte order of array `data` (with byte size per entry `len` and `count` entries).
- **Dependencies**
  None
- **Arguments**

`data`	input/output	void
`len`	input	integer
`count`	input	integer

- **Comments**
  Primarily required for writing binary files where a specific endianness is required, e.g. `.vtk` files are required in big endian.

fCheckTimeSerial

- **Header records**
  `double fCheckTimeSerial()`
- **Function**
  Outputs time in seconds since initial call.
- **Dependencies**
  None
- **Arguments**

`fCheckTimeSerial`	output	double

- **Comments**
  Obtains calculation time based on system clock; parallel calculations may obtain greater timing accuracy with `fCheckTimeMPI`.

7.2.4 lbpGET

fGetNodePosi

- **Header records** (two cases: 3D and 2D)
  `inline long fGetNodePosi(int xpos, int ypos, int zpos)`
  `inline long fGetNodePosi(int xpos, int ypos)`

- **Function**
  Calculates the position of the grid point in a one-dimensional array from its Cartesian coordinate.

- **Dependencies**
  None

- **Arguments**

`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`fGetNodePosi`	output	long integer

- **Comments**
  The calculation follows the standard C++ data structure (row-major).

fGetCoord

- **Header records** (two cases: 3D and 2D)
  `int fGetCoord(long tpos, int& xpos, int& ypos, int& zpos)`
  `int fGetCoord(long tpos, int& xpos, int& ypos)`

- **Function**
  Calculates the Cartesian coordinate of a grid point from its position in a one-dimensional array.

- **Dependencies**
  None

- **Arguments**

`xpos`	output	integer reference
`ypos`	output	integer reference
`zpos`	output	integer reference
`tpos`	input	long integer

fGetOneMassSite

- **Header records** (three cases)
  `double fGetOneMassSite(double* startpos)`
  `double fGetOneMassSite(int fpos, long tpos)`
  `double fGetOneMassSite(int fpos, int xpos, int ypos, int zpos)`

- **Function**
  Calculates the mass density of one of the fluids at a grid point.

- **Dependencies**
  None

- **Arguments**

`startpos`	input	double pointer
`fpos`	input	integer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`fGetOneMassSite`	output	double precision

- **Comments**
  Mass density is calculated according to the definition $\rho = \sum_i f_i$. In the first case, `startpos` is the start point for the summation of particle distribution functions and must be assigned correctly. The second case carries out the same calculation for the `fpos`-th fluid and `tpos`-th grid point, while the third carries it out for the `fpos`-th fluid at the grid point indicated by the Cartesian coordinate (`xpos`, `ypos`, `zpos`). The latter two are more readable to the user but a bit slower to carry out.

fGetAllMassSite

- **Header records** (three cases)
  ```
  int fGetAllMassSite(double *rho, double* startpos)
  int fGetAllMassSite(int fpos, long tpos)
  int fGetAllMassSite(int fpos, int xpos, int ypos, int zpos)
  ```
- **Function**
  Calculates the individual mass densities of all fluids at a grid point.
- **Dependencies**
  None
- **Arguments**

`startpos`	input	double pointer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`rho`	output	double precision array

- **Comments**
  Mass density is calculated according to the definition $\rho = \sum_i f_i$. In the first case, `startpos` is the start point for the summation of particle distribution functions and must be assigned correctly. The second case carries out the same calculation for the `tpos`-th grid point, while the third carries it out at the grid point indicated by the Cartesian coordinate (`xpos`, `ypos`, `zpos`). The latter two are more readable to the user but a bit slower to carry out.

fGetTotMassSite

- **Header records** (two cases)
  ```
  double fGetTotMassSite(double* startpos)
  double fGetTotMassSite(long tpos)
  ```
- **Function**
  Calculates the total mass density of all fluids at a grid point.
- **Dependencies**
  None
- **Arguments**

`startpos`	input	double pointer
`tpos`	input	long integer
`fGetTotMassSite`	output	double precision

- **Comments**
  Mass density is calculated according to the definition $\rho = \sum_i f_i$. The second case carries out the same calculation as the first but using `tpos` for the grid point.

fGetOneMassDomain

- **Header records**
  `double fGetOneMassDomain(int fpos)`
- **Function**
  Calculates the total mass of `fpos`-th fluid in the domain.
- **Dependencies**
  None
- **Arguments**

`fpos`	input	integer
`fGetOneMassDomain`	output	double precision

- **Comments**
  The total mass of the domain does not include boundary areas used for message passing or nodes used to apply boundary conditions.

fGetTotMassDomain

- **Header records**
  `double fGetTotMassDomain()`
- **Function**
  Calculates the total mass of all fluids in the domain.
- **Dependencies**
  None
- **Arguments**

`fGetTotMassDomain`	output	double precision

- **Comments**
  The total mass of the domain does not include boundary areas used for message passing or nodes used to apply boundary conditions.

fGetFracSite

- **Header records** (three cases)
  `double fGetFracSite(int fpos, double* startpos)`
  `double fGetFracSite(int fpos, long tpos)`
  `double fGetFracSite(int fpos, int xpos, int ypos, int zpos)`
- **Function**
  Calculates the mass fraction of fluid `fpos` in the site.
- **Dependencies**
  `fReciprocal`
- **Arguments**

`startpos`	input	double pointer
`fpos`	input	integer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`fGetFracSite`	output	double precision

- **Comments**
  The calculation is based on $z = \frac{\rho_i}{\sum_i \rho_i}$. This function operates in a similar way to `fGetOneMassSite` with the second and third cases slightly slower than the first but more readable.

fGetOneConcSite

- **Header records** (two cases)
  `double fGetOneConcSite(int cpos, long tpos)`
  `double fGetOneConcSite(int cpos, int xpos, int ypos, int zpos)`
- **Function**
  Calculates the concentration of solute `cpos` at the grid point.
- **Dependencies**
  `fGetOneMassSite`
- **Arguments**

`tpos`	input	long integer
`cpos`	input	integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`fGetFracSite`	output	double precision

fGetTemperatureSite

- **Header records** (two cases)
  `double fGetTemperatureSite(long tpos)`
  `double fGetTemperatureSite(long xpos, long ypos, long zpos)`
- **Function**
  Calculates the scalar temperature at the grid point.
- **Dependencies**
  `fGetOneMassSite`
- **Arguments**

`tpos`	input	long integer
`xpos`	input	long integer
`ypos`	input	long integer
`zpos`	input	long integer
`fGetFracSite`	output	double precision

fGetOneSpeedSite

- **Header records** (three cases)
  `int fGetOneSpeedSite(double *speed, double* startpos)`
  `int fGetOneSpeedSite(double *speed, int fpos, long tpos)`
  `int fGetOneSpeedSite(double *speed, int fpos, int xpos, int ypos, int zpos)`
- **Function**
  Calculates the macroscopic speed of (compressible) fluid `fpos` at the grid point.
- **Dependencies**
  `fReciprocal`

- **Arguments**

`startpos`	input	double pointer
`fpos`	input	integer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`speed`	output	double precision array

- **Comments**
The calculation is based on $v_\alpha = \frac{\sum_i f_i e_{i\alpha}}{\rho}$. The second and third cases are slightly slower than the first but more readable.

fGetOneSpeedIncomSite

- **Header records** (three cases)
`int fGetOneSpeedIncomSite(double *speed, double* startpos, double rho0)`
`int fGetOneSpeedIncomSite(double *speed, int fpos, long tpos)`
`int fGetOneSpeedIncomSite(double *speed, int fpos, int xpos, int ypos, int zpos)`

- **Function**
Calculates the macroscopic speed of incompressible fluid `fpos` at the grid point.

- **Dependencies**
`fReciprocal`

- **Arguments**

`startpos`	input	double pointer
`rho0`	input	double precision
`fpos`	input	integer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`speed`	output	double precision array

- **Comments**
The calculation is based on $v_\alpha = \frac{\sum_i f_i e_{i\alpha}}{\rho_0}$. The second and third cases are slightly slower than the first but more readable.

fGetSpeedSite

- **Header records** (three cases)
`int fGetSpeedSite(double *speed, double* startpos)`
`int fGetSpeedSite(double *speed, long tpos)`
`int fGetSpeedSite(double *speed, int xpos, int ypos, int zpos)`

- **Function**
Calculates the macroscopic speed of all (compressible) fluids at the grid point.

- **Dependencies**
`fReciprocal`

- **Arguments**

`startpos`	input	double pointer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`speed`	output	double precision array

- **Comments**
The calculation is based on $v_\alpha = \frac{\sum_i f_i e_{i\alpha}}{\rho}$. The second and third cases are more readable than the first but also slightly slower.

fGetSpeedIncomSite

- **Header records** (three cases)
`int fGetSpeedIncomSite(double *speed, double* startpos)`
`int fGetSpeedIncomSite(double *speed, long tpos)`
`int fGetSpeedIncomSite(double *speed, int xpos, int ypos, int zpos)`

- **Function**
Calculates the macroscopic speed of all incompressible fluids at the grid point.

- **Dependencies**
`fReciprocal`

- **Arguments**

`startpos`	input	double pointer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`speed`	output	double precision array

- **Comments**
The calculation is based on $v_\alpha = \frac{\sum_i f_i e_{i\alpha}}{\rho_0}$. The second and third cases are more readable than the first but also slightly slower.

fGetOneMomentSite

- **Header records** (three cases)
`int fGetOneMomentSite(double *speed, double* startpos)`
`int fGetOneMomentSite(double *speed, int fpos, long tpos)`
`int fGetOneMomentSite(double *speed, int fpos, int xpos, int ypos, int zpos)`

- **Function**
Calculates the momentum of one of the fluids at the grid point.

- **Dependencies**
None

- **Arguments**

`startpos`	input	double pointer
`fpos`	input	integer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`speed`	output	double precision array

- **Comments**
  The calculation is based on $p_\alpha = \sum_i f_i e_{i\alpha}$. The second and third cases are more readable than the first but also slightly slower.

fGetTotMomentSite

- **Header records**
  `int fGetTotMomentSite(double *speed, double* startpos)`
- **Function**
  Calculates the momentum of all fluids at the grid point.
- **Dependencies**
  None
- **Arguments**

`startpos`	input	double pointer
`speed`	output	double precision array

fGetTotMomentDomain

- **Header records**
  `int fGetTotMomentDomain(double *momentum)`
- **Function**
  Calculates the momentum of all fluids in the domain.
- **Dependencies**
  `fGetTotMomentSite`
- **Arguments**

`momentum`	output	double precision array

- **Comments**
  This function is mainly used to verify that the domain momentum along each axis is conserved.

fGetOneDirecSpeedSite

- **Header records** (three cases)

```
float fGetOneDirecSpeedSite(int dire, double* startpos)
float fGetOneDirecSpeedSite(int dire, long tpos)
float fGetOneDirecSpeedSite(int dire, int xpos, int ypos, int zpos)
```

- **Function**
  Calculates the grid speed for all (compressible) fluids along direction `dire`: `0` for x, `1` for y and `2` for z.

- **Dependencies**
  `fReciprocal`

- **Arguments**

`startpos`	input	double pointer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`dire`	input	double pointer
`fGetOneDirecSpeedSite`	output	floating point

- **Comments**
  Mainly used to output grid speed. The second and third cases are more readable than the first but also slightly slower.

fGetOneDirecSpeedIncomSite

- **Header records** (three cases)
  `float fGetOneDirecSpeedIncomSite(int dire, double* startpos)`
  `float fGetOneDirecSpeedIncomSite(int dire, long tpos)`
  `float fGetOneDirecSpeedIncomSite(int dire, int xpos, int ypos, int zpos)`

- **Function**
  Calculates the grid speed for all incompressible fluids along direction `dire`: `0` for x, `1` for y and `2` for z.

- **Dependencies**
  `fReciprocal`

- **Arguments**

`startpos`	input	double pointer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`dire`	input	double pointer
`fGetOneDirecSpeedIncomSite`	output	floating point

- **Comments**
  Mainly used to output grid speed. The second and third cases are more readable than the first but also slightly slower.

7.2.5 lbpIO

fDefineSystem

- **Header records**
  `int fDefineSystem(const char* filename = "lbin.sys")`

- **Function**
  Reads calculation parameters (lattice scheme, types of collisions and forcing, mesophase algorithms, numbers of fluids, solutes, temperature scalars, phase field order parameters, grid size) from input system file `lbin.sys`.

- **Dependencies**
  `lbin.sys` data file

- **Arguments**

`filename`	input	array of characters

- **Comments**
  The default file name is `lbin.sys`: to use different file names the argument for this routine can be changed by the user.

fPrintSystemInfo

- **Header records**
  `int fPrintSystemInfo()`

- **Function**
  Prints system information (lattice model, dimensions, boundary width, numbers of fluids, solutes and temperature scalars, collision model, forcing type and mesophase interactions).

- **Dependencies**
  None

fPrintEndEquilibration

- **Header records**
  `int fPrintEndEquilibration()`

- **Function**
  Prints message indicating end of static equilibration process.

- **Dependencies**
  None

- **Comments**
  This message is not printed if no equilibration steps are specified in the `lbin.sys` file.

fPrintDomainMass

- **Header records**
  `int fPrintDomainMass()`

- **Function**
  Calculates and prints total and individual fluid masses in domain.

- **Dependencies**
  `double fGetTotMassDomain()`
  `double fGetOneMassDomain()`

- **Comments**
  This routine only produces the masses for the entire system if running in serial; to obtain system masses in parallel running, `fPrintSystemMass()` would be needed.

fPrintDomainMomentum

- **Header records**
  `int fPrintDomainMomentum()`
- **Function**
  Calculates and prints the total fluid momentum in domain.
- **Dependencies**
  `int fGetTotMomentDomain(double *momentum)`
- **Comments**
  This routine only produces the entire system momentum if running in serial; to obtain system momentum in parallel running, `fPrintSystemMomentum()` would be needed.

fOutput

- **Header records**
  `int fOutput(const char* filename="lbout")`
- **Function**
  Outputs all system data in required format.
- **Dependencies**
  `int fOutputPlot3D()`
  `int fOutputLegacyVTK()`
  `int fOutputVTK()`
- **Comments**
  This routine is used to write output files in parallel running; a serial version of this routine, `fsOutput`, also exists.

fInputParameters

- **Header records**
  `int fInputParameters(const char* filename="lbin.sys")`
- **Function**
  Reads system parameters in from `lbin.sys` data file.
- **Dependencies**
  `lbin.sys` data file
- **Arguments**

`filename`	input	array of characters

- **Comments**
  The default file name is `lbin.sys`: to use different file names the argument for this routine can be changed by the user.

fReadSpaceParameter

- **Header records**
  `int fReadSpaceParameter(const char* filename="lbin.spa")`

- **Function**
  Reads 2D or 3D space parameters from data file `lbin.spa`.
- **Dependencies**
  `lbin.spa` data file
  `fReadSpace2D`
  `fReadSpace3D`
- **Arguments**

`filename`	input	array of characters

- **Comments**
  The default file name is `lbin.spa`: this can be changed by the user if the space data file has a different name. If system is to be equilibrated, on-grid bounce-back is initially and temporarily used in place all boundary conditions specifying fixed values for fluid velocities/densities, solute concentrations and temperatures.

fReadInitialState

- **Header records**
  `int fReadInitialState(const char* filename="lbin.init")`
- **Function**
  Reads initial velocities, densities, concentrations and temperatures from data file `lbin.init` and calculates initial distribution functions.
- **Dependencies**
  `lbin.init` data file
  `fReadInitialState2D`
  `fReadInitialState3D`
- **Arguments**

`filename`	input	array of characters

- **Comments**
  The default file name is `lbin.init`: this can be changed by the user if the initial state data file has a different name. This routine will replace the default values of all properties at specified points and should be called after `fInitializeSystem`.

fSetoffSteer

- **Header records**
  `int fSetoffSteer()`
- **Function**
  Creates a file called `notsteer` to prevent DL_MESO_LBE from creating new `lbin.sys` and `lbin.spa` files (which occurs if `notsteer` is missing).
- **Dependencies**
  None
- **Comments**
  If the user has changed the input datafiles, the code will run with new parameters.

fCheckSteer

- **Header records**
  `int fCheckSteer()`
- **Function**
  Checks for the existence of `notsteer` files: if found, then reads `lbin.sys` and `lbin.spa` files.
- **Dependencies**
  `fInputParameters("lbin.sys")`
  `fReadSpaceParameter("lbin.spa")`

7.2.6 lbpIOPlot3D

fOutputGrid

- **Header records**
  `int fOutputGrid3D(const char* filename="lbout")`
- **Function**
  Outputs grid positions for system in Plot3D format.
- **Dependencies**
  `fOutputGrid2D`
  `fOutputGrid3D`
- **Arguments**

`filename`	input	array of characters

- **Comments**
  The default output file name is `lbout*.xyz` for 3D systems and `lbout*.xy` for 2D systems.

fOutputQ

- **Header records**
  `int fOutputQ(const char* filename="lbout")`
- **Function**
  Outputs macroscopic mass densities and fractions for all fluids, concentrations for all solutes, temperature and velocity (speeds along x-, y- and z-directions) at each lattice point for system in Plot3D format.
- **Dependencies**
  `fOutputQ2D`
  `fOutputQ3D`
- **Arguments**

`filename`	input	array of characters

- **Comments**
  The default output file name is `lbout*prop*.q`, with `prop` substituted by `dens`, `frac`, `conc` or `temp` for fluid densities, mass fractions, solute concentrations or temperature respectively and preceded by the number of the fluid or solute. This can be changed by specifying an output file name when calling the routine.

fOutputQP

- **Header records**
  `int fOutputQP(const char* filename="lbout", int iprop=0)`
- **Function**
  Outputs macroscopic mass density of fluid `iprop` and velocity (speeds along x-, y- and z-directions) at each lattice point for system in Plot3D format.
- **Dependencies**
  `fOutputQP2D`
  `fOutputQP3D`
- **Arguments**

`filename`	input	array of characters
`iprop`	input	integer

- **Comments**
  The default output file name is `lbout*.q` and the density of fluid 0 is output by default. This can be changed by specifying an output file name and fluid number (up to `lbsy.nf`-1) when calling the routine.

fOutputQCA

- **Header records**
  `int fOutputQCA(const char* filename="lbout", int iprop=0)`
- **Function**
  Outputs mass fraction of fluid `iprop` and speeds along x-, y- and z-directions at each lattice point for system in Plot3D format.
- **Dependencies**
  `fOutputQCA2D`
  `fOutputQCA3D`
- **Arguments**

`filename`	input	array of characters
`iprop`	input	integer

- **Comments**
  The default output file name is `lbout*.q` and the mass fraction of fluid 0 is output by default. These can be changed by specifying an output file name and fluid number (up to `lbsy.nf`-1) when calling the routine.

fOutputQCB

- **Header records**
  `int fOutputQCB(const char* filename="lbout", int iprop=0)`
- **Function**
  Outputs concentration of solute `iprop` and fluid velocity at each lattice point for system in Plot3D format.
- **Dependencies**
  `fOutputQCB2D`
  `fOutputQCB3D`

- **Arguments**

`filename`	input	array of characters
`iprop`	input	integer

- **Comments**
  The default output file name is `lbout*.q` and the concentration of solute 0 is output by default. These can be changed by specifying an output file name and solute number (up to `lbsy.ns`−1) when calling the routine.

fOutputQT

- **Header records**
  `int fOutputQT(const char* filename="lbout")`
- **Function**
  Outputs macroscopic temperature and fluid velocity at each lattice point for system in Plot3D format.
- **Dependencies**
  `fOutputQT2D`
  `fOutputQT3D`
- **Arguments**

`filename`	input	array of characters

- **Comments**
  The default output file name is `lbout*.q`. This can be changed by specifying an output file name when calling the routine.

7.2.7 lbpIOLegacyVTK

fOutputLegacyVTK

- **Header records**
  `int fOutputLegacyVTK(const char* filename="lbout")`
- **Function**
  Outputs macroscopic mass densities and fractions for all fluids, concentrations for all solutes, temperature and velocity (speeds along x-, y- and z-directions) at each lattice point for system in Structured Grid Legacy VTK format.
- **Dependencies**
  `fOutputLegacyVTK2D`
  `fOutputLegacyVTK3D`
- **Arguments**

`filename`	input	array of characters

- **Comments**
  The default output file name is `lbout*.vtk`. This can be changed by specifying an output file name when calling the routine.

fOutputLegacyVTKP

- **Header records**
  `int fOutputLegacyVTK(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs macroscopic mass density and velocity (speeds along x-, y- and z-directions) at each lattice point of compressible fluid `iprop` for system in Structured Grid Legacy VTK format.

- **Dependencies**
  `fOutputLegacyVTKP2D`
  `fOutputLegacyVTKP3D`

- **Arguments**

`filename`	input	array of characters
`iprop`	input	integer

- **Comments**
  The default output file name is `lbout*.vtk` and the density of fluid 0 is output by default. This can be changed by specifying an output file name and fluid number (up to `lbsy.nf`−1) when calling the routine.

fOutputLegacyVTKCA

- **Header records**
  `int fOutputLegacyVTKCA(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs mass fraction of fluid `iprop` and speeds along x-, y- and z-directions at each lattice point for system in Structured Grid Legacy VTK format.

- **Dependencies**
  `fOutputLegacyVTKCA2D`
  `fOutputLegacyVTKCA3D`

- **Arguments**

`filename`	input	array of characters
`iprop`	input	integer

- **Comments**
  The default output file name is `lbout*.vtk` and the mass fraction of fluid 0 is output by default. These can be changed by specifying an output file name and fluid number (up to `lbsy.nf`−1) when calling the routine.

fOutputLegacyVTKCB

- **Header records**
  `int fOutputLegacyVTKCB(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs concentration of solute `iprop` and fluid velocity at each lattice point for system in Structured Grid Legacy VTK format.

- **Dependencies**
  `fOutputLegacyVTKCB2D`
  `fOutputLegacyVTKCB3D`

- **Arguments**

`filename`	input	array of characters
`iprop`	input	integer

- **Comments**
  The default output file name is `lbout*.vtk` and the concentration of solute 0 is output by default. These can be changed by specifying an output file name and solute number (up to `lbsy.ns`−1) when calling the routine.

fOutputLegacyVTKT

- **Header records**
  `int fOutputLegacyVTKT(const char* filename="lbout")`
- **Function**
  Outputs macroscopic temperature and fluid velocity at each lattice point for system in Structured Grid Legacy VTK format.
- **Dependencies**
  `fOutputLegacyVTKT2D`
  `fOutputLegacyVTKT3D`
- **Arguments**

`filename`	input	array of characters

- **Comments**
  The default output file name is `lbout*.vtk`. This can be changed by specifying an output file name when calling the routine.

7.2.8 lbpIOVTK

fOutputVTK

- **Header records**
  `int fOutputVTK(const char* filename="lbout")`
- **Function**
  Outputs macroscopic macroscopic mass densities and fractions for all fluids, concentrations for all solutes, temperature and velocity (speeds along x-, y- and z-directions) at each lattice point for system in Structured Grid XML VTK format.
- **Dependencies**
  `fOutputVTK2D`
  `fOutputVTK3D`
- **Arguments**

`filename`	input	array of characters

- **Comments**
  The default output file name is `lbout*.vts`. This can be changed by specifying an output file name when calling the routine.

fOutputVTKP

- **Header records**
  `int fOutputVTKP(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs macroscopic mass density of fluid `iprop` and velocity (speeds along x-, y- and z-directions) at each lattice point for system in Structured Grid XML VTK format.

- **Dependencies**
  `fOutputVTKP2D`
  `fOutputVTKP3D`

- **Arguments**

`filename`	input	array of characters
`iprop`	input	integer

- **Comments**
  The default output file name is `lbout*.vts` and the density of fluid 0 is output by default. This can be changed by specifying an output file name and fluid number (up to `lbsy.nf`-1) when calling the routine.

fOutputVTKCA

- **Header records**
  `int fOutputVTKCA(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs mass fraction of fluid `iprop` and speeds along x-, y- and z-directions at each lattice point for system in Structured Grid XML VTK format.

- **Dependencies**
  `fOutputVTKCA2D`
  `fOutputVTKCA3D`

- **Arguments**

`filename`	input	array of characters
`iprop`	input	integer

- **Comments**
  The default output file name is `lbout*.vts` and the mass fraction of fluid 0 is output by default. These can be changed by specifying an output file name and fluid number (up to `lbsy.nf`-1) when calling the routine.

fOutputVTKCB

- **Header records**
  `int fOutputVTKCB(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs concentration of solute `iprop` and fluid velocity at each lattice point for system in Structured Grid XML VTK format.

- **Dependencies**
  `fOutputVTKCB2D`
  `fOutputVTKCB3D`

- **Arguments**

`filename`	input	array of characters
`iprop`	input	integer

- **Comments**
  The default output file name is `lbout*.vts` and the concentration of solute 0 is output by default. These can be changed by specifying an output file name and solute number (up to `lbsy.ns`−1) when calling the routine.

fOutputVTKT

- **Header records**
  `int fOutputVTKT(const char* filename="lbout")`
- **Function**
  Outputs macroscopic temperature and fluid velocity at each lattice point for system in Structured Grid XML VTK format.
- **Dependencies**
  `fOutputVTKT2D`
  `fOutputVTKT3D`
- **Arguments**

`filename`	input	array of characters

- **Comments**
  The default output file name is `lbout*.vts`. This can be changed by specifying an output file name when calling the routine.

Other output routines

Subroutines with names `fsOutput*` are suitable for serial running and produce output files for entire systems. Unlike the routines listed above, these omit any domain boundary lattice points used in calculations.

Notes regarding `.q`, `.vtk` and `.vts` files

- `lbout`x`.*` is the `.q`, `.vtk` or `.vts` file at the xth saved step during serial running.
- `lbout`y`at`x`.*` is the `.q`, `.vtk` or `.vts` file at the xth saved step written by processor y during parallel running.
- `lbout` files produced using multiple processors will require gathering or simultaneous plotting: see Appendix B for more details.

7.2.9 lbpBOUND

fNextStep

- **Header records** (three cases: 3D, 2D and coordinate)
  `long fNextStep(int q, int xpos, int ypos, int zpos)`
  `long fNextStep(int q, int xpos, int ypos)`
  `long fNextStep(int dx, int dy, int dz, long tpos)`
- **Function**
  Finds particle position at the next time step when currently at (`xpos`, `ypos`, `zpos`) (or `tpos`) and moving along direction `q` or (`dx`, `dy`, `dz`).

- **Dependencies**
  `fCppMod`
- **Arguments**

`q`	input	integer
`dx`	input	integer
`dy`	input	integer
`dz`	input	integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`tpos`	input	long integer
`fNextStep`	output	long integer

fBounceBackF

- **Header records**
  `int fBounceBackF(long tpos)`
- **Function**
  Performs an on-grid bounce-back for the fluid distribution function at the `tpos`-th grid point.
- **Dependencies**
  None
- **Arguments**

`tpos`	input	long integer

- **Comments**
  This bounce-back boundary condition is carried out using `f(lbopv[i]) = f(lbv[i])`, i.e. populations are exchanged with conjugate values.

fBounceBackC

- **Header records**
  `int fBounceBackC(long tpos)`
- **Function**
  Performs an on-grid bounce-back for the solute distribution function at the `tpos`-th grid point.
- **Dependencies**
  None
- **Arguments**

`tpos`	input	long integer

- **Comments**
  This bounce-back boundary condition is carried out using `f(lbopv[i]) = f(lbv[i])`.

fBounceBackT

- **Header records**
  `int fBounceBackC(long tpos)`
- **Function**
  Performs an on-grid bounce-back for the temperature distribution function at the `tpos`-th grid point.

- **Dependencies**
  None
- **Arguments**

`tpos`	input	long integer

- **Comments**
  This bounce-back boundary condition is carried out using `f(lbopv[i]) = f(lbv[i])`.

fMidBounceBackF

- **Header records**
  `int fMidBounceBackF(long tpos)`
- **Function**
  Performs a mid-link bounce-back for the fluid distribution function at the `tpos`-th grid point.
- **Dependencies**
  None
- **Arguments**

`tpos`	input	long integer

- **Comments**
  This bounce-back boundary condition is carried out by exchanging post-collisional populations with conjugate values in neighbouring grid points.

fMidBounceBackC

- **Header records**
  `int fMidBounceBackC(long tpos)`
- **Function**
  Performs a mid-link bounce-back for the solute distribution function at the `tpos`-th grid point.
- **Dependencies**
  None
- **Arguments**

`tpos`	input	long integer

- **Comments**
  This bounce-back boundary condition is carried out by exchanging post-collisional populations with conjugate values in neighbouring grid points.

fMidBounceBackT

- **Header records**
  `int fMidBounceBackC(long tpos)`
- **Function**
  Performs a mid-link bounce-back for the temperature distribution function at the `tpos`-th grid point.
- **Dependencies**
  None

- **Arguments**

`tpos`	input	long integer

- **Comments**
  This bounce-back boundary condition is carried out by exchanging post-collisional populations with conjugate values in neighbouring grid points.

fSiteBlankF

- **Header records**
  `int fSiteBlankF(long tpos)`
- **Function**
  Sets the fluid particle distribution function at the `tpos`-th grid point to zero.
- **Dependencies**
  None
- **Arguments**

`tpos`	input	long integer

- **Comments**
  This routine is used to ensure e.g. flows inside solid boundaries are negligible.

fSiteBlankC

- **Header records**
  `int fSiteBlankC(long tpos)`
- **Function**
  Sets the solute particle distribution function at the `tpos`-th grid point to zero.
- **Dependencies**
  None
- **Arguments**

`tpos`	input	long integer

- **Comments**
  This routine is used to ensure e.g. diffusion inside a bulk solid is negligible compared to that in a liquid.

fSiteBlankT

- **Header records**
  `int fSiteBlankT(long tpos)`
- **Function**
  Sets the temperature particle distribution function at the `tpos`-th grid point to zero.
- **Dependencies**
  None
- **Arguments**

`tpos`	input	long integer

- **Comments**
  This routine is used to ensure e.g. negligible heat transfer through an insulator.

fFixedSpeedFluid

- **Header records**
  `int fFixedSpeedFluid(long tpos, int prop, double *uwall)`
- **Function**
  Calculates the particle distribution function at a fixed speed boundary.
- **Dependencies**
  Many for different lattice schemes
- **Arguments**

`tpos`	input	long integer
`prop`	input	long integer
`uwall`	output	array of doubles

- **Comments**
  Planar surface calculations are based on [73]; concave edges and corners use equilibrium boundary conditions with the density on the edge and at the grid point assumed to be equal to the values at their nearest neighbours in the bulk fluid. The array `uwall` is the velocity at the grid point for all fluids, which is subsequently used for solute concentration and temperature boundary conditions.

fFixedDensityFluid

- **Header records**
  `int fFixedDensityFluid(long tpos, int prop, double *uwall)`
- **Function**
  Calculates the particle distribution function at a fixed density boundary.
- **Dependencies**
  Many for different lattice schemes
- **Arguments**

`tpos`	input	long integer
`prop`	input	long integer
`uwall`	output	array of doubles

- **Comments**
  Planar surface calculations are based on [73]; concave edges and corners assume zero speed at the boundary. The array `uwall` is the velocity at the grid point for all fluids, which is subsequently used for solute concentration and temperature boundary conditions.

fFixedSoluteConcen

- **Header records**
  `int fFixedSoluteConcen(long tpos, int prop, double *uwall)`
- **Function**
  Calculates the particle distribution function at a fixed composition boundary.
- **Dependencies**
  Many for different lattice schemes
- **Arguments**

`tpos`	input	long integer
`prop`	input	long integer
`uwall`	input	array of doubles

- **Comments**
  Planar surface calculations are based on [28]; concave edges and corners assume zero speed at the boundary. The fluid velocity at the lattice point, given by the array `uwall`, is required for this boundary condition.

fFixedTemperature

- **Header records**
  `int fFixedTemperature(long tpos, int prop, double *uwall)`
- **Function**
  Calculates the particle distribution function at a fixed temperature boundary.
- **Dependencies**
  Many for different lattice schemes
- **Arguments**

`tpos`	input	long integer
`prop`	input	long integer
`uwall`	input	array of doubles

- **Comments**
  Planar surface calculations are based on [28]; concave edges and corners assume zero speed at the boundary. The fluid velocity at the lattice point, given by the array `uwall`, is required for this boundary condition.

fPostCollBoundary

- **Header records**
  `int fPostCollBoundary()`
- **Function**
  Calculates the particle distribution function at different boundaries after the collision step, prior to propagation.
- **Dependencies**
  Many for different lattice schemes
- **Comments**
  Algorithms for other boundary conditions can be added by the user, although care should be taken as to when they are applied in each time step: the conditions invoked in this routine are applied to post-collisional distribution functions before propagation takes place.

fPostPropBoundary

- **Header records**
  `int fPostPropBoundary()`
- **Function**
  Calculates the particle distribution function at different boundaries after propagation, prior to the next collision step.
- **Dependencies**
  Many for different lattice schemes
- **Comments**
  Algorithms for other boundary conditions can be added by the user, although care should be taken as to when they are applied in each time step: the conditions invoked in this routine apply to distribution functions after propagation.

fNeighbourBoundary

- **Header records**
  `int fNeighbourBoundary()`
- **Function**
  Determines the existence of solid boundaries in neighbouring lattice points.
- **Dependencies**
  None
- **Comments**
  Stores results in the `lbneigh` array: currently only covers orthogonal directions (i.e. no diagonals). Only needs to be called once if boundary conditions do not change during calculations.

fsPeriodic

- **Header records**
  `int fsPeriodic()`
- **Function**
  Applies periodic boundary condition for serial calculations with non-zero boundary domain widths by copying distribution functions from edges of fluid points.
- **Dependencies**
  `fsPeriodic2D`
  `fsPeriodic3D`
- **Comments**
  Serial equivalent of `fNonBlockCommunication`, essential for using the combined swap propagation routine `fPropagationCombinedSwap` in serial running.

fsBoundPeriodic

- **Header records**
  `int fsBoundPeriodic()`
- **Function**
  Applies periodic boundary condition for serial calculations with non-zero boundary domain widths by copying boundary information from edges of fluid points.
- **Dependencies**
  `fsBoundPeriodic2D`
  `fsBoundPeriodic3D`
- **Comments**
  Serial equivalent of `fBoundNonBlockCommunication`, may be required for using the combined swap propagation routine `fPropagationCombinedSwap` in serial running.

fsForcePeriodic

- **Header records**
  `int fsForcePeriodic()`

- **Function**
  Applies periodic boundary condition for serial calculations with non-zero boundary domain widths by copying interaction forces from edges of fluid points.
- **Dependencies**
  `fsForcePeriodic2D`
  `fsForcePeriodic3D`
- **Comments**
  Serial equivalent of `fForceNonBlockCommunication`, may be required for using the combined swap propagation routine `fPropagationCombinedSwap` in serial running for any system requiring non-constant forces.

fsIndexPeriodic

- **Header records**
  `int fsIndexPeriodic()`
- **Function**
  Applies periodic boundary condition for serial calculations with non-zero boundary domain widths by copying phase index spatial derivatives from edges of fluid points.
- **Dependencies**
  `fsIndexPeriodic2D`
  `fsIndexPeriodic3D`
- **Comments**
  Serial equivalent of `fIndexNonBlockCommunication`, may be required for using the combined swap propagation routine `fPropagationCombinedSwap` in serial running with the Lishchuk mesophase algorithm.

7.2.10 lbpFORCE

fInteractionForceZero

- **Header records**
  `int fInteractionForceZero()`
- **Function**
  Resets all interaction forces to zero prior to calculations.
- **Dependencies**
  None
- **Comments**
  This routine should be called before any non-constant forces (e.g. mesophase interactions) are calculated; not required if only using constant body forces such as gravity.

fCalcPotential_ShanChen

- **Header records**
  `int fCalcPotential_ShanChen()`
- **Function**
  Calculates the interaction potential $\phi^a = \rho_0^a \left(1 - \exp\left(-\frac{\rho^a}{\rho_0^a}\right)\right)$ as suggested by the Shan-Chen model for each species and lattice point.

- **Dependencies**
  `fGetOneMassSite`
- **Comments**
  The Shan-Chen model is detailed in [55, 56]. Alternative mesoscale interaction potentials based on this model can be introduced here.

fCalcInteraction_ShanChen

- **Header records**
  `int fCalcInteraction_ShanChen(int xpos, int ypos, int zpos)`
- **Function**
  Calculates particle interaction forces according to Shan-Chen model.
- **Arguments**

`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer

- **Comments**
  Further details can be found in [55, 56]. Similarly named routines to calculate interaction forces for alternative mesoscale algorithms can be added by the user.

fCalcInteraction_ShanChenWetting

- **Header records**
  `int fCalcInteraction_ShanChenWetting(int xpos, int ypos, int zpos)`
- **Function**
  Calculates particle interaction forces according to Shan-Chen model with additional fluid-solid wetting forces.
- **Arguments**

`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer

- **Comments**
  Further details can be found in [55, 56] and [41]. Similarly named routines to calculate interaction forces for alternative mesoscale algorithms can be added by the user.

fInteractionForceShanChen

- **Header records**
  `int fInteractionForceShanChen()`
- **Function**
  Calculates interaction forces for all fluids based on the Shan-Chen model[55, 56].
- **Dependencies**
  `fCalcInteraction_ShanChen`
- **Comments**
  The interaction potentials need to be calculated prior to calling this routine. Alternative mesoscale interactions can be applied using similar routines.

fInteractionForceShanChenWetting

- **Header records**
  `int fInteractionForceShanChenWetting()`
- **Function**
  Calculates interaction forces for all fluids based on the Shan-Chen model[55, 56] with additional wetting forces[41].
- **Dependencies**
  `fCalcInteraction_ShanChenWetting`
- **Comments**
  The interaction potentials need to be calculated prior to calling this routine. Alternative mesoscale interactions can be applied using similar routines.

fCalcPhaseIndex_Lishchuk

- **Header records**
  `int fCalcPhaseIndex_Lishchuk()`
- **Function**
  Calculates phase indices $\rho_{ab}^N = \frac{\rho^a - \rho^b}{\rho^a + \rho^b}$ and first-order spatial derivatives ($\nabla \rho_{ab}^N$) for all unlike fluid pairs, storing the latter for future use.
- **Dependencies**
  `fGetOneMassSite`
- **Comments**
  The Lishchuk model is detailed in [35, 20].

fCalcInteraction_Lishchuk

- **Header records**
  `int fCalcInteraction_Lishchuk(int xpos, int ypos, int zpos)`
- **Function**
  Calculates particle interaction forces according to Lishchuk model.
- **Arguments**

`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer

- **Comments**
  Further details can be found in [35, 20].

fInteractionForceLishchuk

- **Header records**
  `int fInteractionForceLishchuk()`
- **Function**
  Calculates interaction forces for all fluids based on the Lishchuk model[35, 20].

- **Dependencies**
  `fCalcInteraction_Lishchuk`
- **Comments**
  The phase indices need to be calculated prior to calling this routine.

fCalcForce_Boussinesq

- **Header records**
  `int fCalcForce_Boussinesq(long tpos, double temph, double templ)`
- **Function**
  Calculates buoyancy-driven thermal convection force according to the Boussinesq approximation.
- **Dependencies**
  `fGetOneMassSite`
  `fGetTemperatureSite`
- **Arguments**

`tpos`	input	long integer
`temph`	input	double precision
`templ`	input	double precision

- **Comments**
  The buoyancy force for compressible fluids calculated by this routine is

$$\vec{F}^a = -\vec{g}\beta^a \rho \left(\frac{T - T_0}{T_h - T_l} \right)$$

with the reference temperature $T_0 = \frac{1}{2}(T_h + T_l)$. The expression for incompressible fluids is similar with the constant fluid density ρ_0 substituted for ρ.

fConvectionForceBoussinesq

- **Header records**
  `int fConvectionForceBoussinesq(double temph, double templ)`
- **Function**
  Calculates Boussinesq thermal convection forces for all fluids based on [18].
- **Dependencies**
  `fCalcForce_Boussinesq`
- **Arguments**

`temph`	input	double precision
`templ`	input	double precision

7.2.11 lbpSUB

fWeakMemory

- **Header records**
  `inline void fWeakMemory()`
- **Function**
  Terminates calculation if system has insufficient memory.

- **Dependencies**
  None
- **Comments**
  If called, will print error message:
  `error: cannot allocate more memory`.

fMemoryAllocation

- **Header records**
  `int fMemoryAllocation()`
- **Function**
  Allocates memory for lattice Boltzmann calculations.
- **Dependencies**
  `fWeakMemory`
- **Comments**
  If memory allocation is unsuccessful, will print error message and stop calculation.

fFreeMemory

- **Header records**
  `int fFreeMemory()`
- **Function**
  Frees allocated memory.
- **Dependencies**
  None

fSetSerialDomain

- **Header records**
  `int fSetSerialDomain()`
- **Function**
  Sets domain parameters for serial running.
- **Dependencies**
  None
- **Comments**
  Default routine, sets domain boundary width `lbdm.bwid` to zero.

fSetSerialDomainBuffer

- **Header records**
  `int fSetSerialDomainBuffer()`
- **Function**
  Sets domain parameters for serial running including additional boundary points.
- **Dependencies**
  None

- **Comments**
  Similar to `fSetSerialDomain` but does not modify the domain boundary width from its user-specified value.

fStartDLMESO

- **Header records**
  `int fStartDLMESO()`
- **Function**
  Announces start of DL_MESO_LBE run.
- **Dependencies**
  None
- **Comments**
  If preferred, the call to this routine can be commented out.

fFinishDLMESO

- **Header records**
  `int fFinishDLMESO()`
- **Function**
  Announces end of DL_MESO_LBE run, prints simulation time, efficiency measure (Millions of Lattice Updates Per Second) and a message encouraging citations of DL_MESO.
- **Dependencies**
  None
- **Comments**
  If preferred, the call to this routine can be commented out.

fGetModel

- **Header records**
  `int fGetModel()`
- **Function**
  Initializes vectors `lbv`, `lbw` and `lbopv` for lattice model.
- **Dependencies**
  `D2Q9`
  `D3Q15`
  `D3Q19`
  `D3Q27`
- **Comments**
  Parameters are specified according to requested space dimension and number of discrete velocities.

fMarkBoundArea

- **Header records**
  `int fMarkBoundArea()`

- **Function**
  Denotes where boundary areas for message passing and/or periodic boundary conditions are located.
- **Dependencies**
  `int fMarkBoundArea3D()`
  `int fMarkBoundArea2D()`
- **Comments**
  Only used when boundary areas are used (primarily for parallel computing).

fGetEquilibriumF

- **Header records**
  `int fGetEquilibriumF(double *feq, double *v, double rho)`
- **Function**
  Calculates equilibrium distribution function for compressible fluid.
- **Dependencies**
  None
- **Arguments**

`feq`	output	double pointer
`v`	input	double pointer
`rho`	input	double precision

- **Comments**
  The equilibrium distribution function calculated here is

$$f^{eq} = w_i \rho \left[1 + \frac{3\left(\vec{e}_i \cdot \vec{u}\right)}{c^2} + \frac{9\left(\vec{e}_i \cdot \vec{u}\right)^2}{2c^4} - \frac{3u^2}{2c^2} \right]$$

  which is only suitable for square lattices. Other lattice models, e.g. FHP[11], would require modification or alternative versions of this routine.

fGetEquilibriumFIncom

- **Header records**
  `int fGetEquilibriumFIncom(double *feq, double *v, double rho, double rho0)`
- **Function**
  Calculates equilibrium distribution function for incompressible fluid.
- **Dependencies**
  None
- **Arguments**

`feq`	output	double pointer
`v`	input	double pointer
`rho`	input	double precision
`rho0`	input	double precision

- **Comments**
  Equilibrium distribution function calculated here is

$$f^{eq} = w_i \left\{ \rho + \rho_0 \left[\frac{3\left(e_i \cdot u\right)}{c^2} + \frac{9\left(e_i \cdot u\right)^2}{2c^4} - \frac{3u^2}{2c^2} \right] \right\},$$

  which is only suitable for square lattices. Further details can be found in [24].

fGetEquilibriumC

- **Header records**
  `int fGetEquilibriumC(double *feq, double *v, double rho)`
- **Function**
  Calculates equilibrium distribution function for solute.
- **Dependencies**
  None
- **Arguments**

`feq`	output	double pointer
`v`	input	double pointer
`rho`	input	double precision

- **Comments**
  The equilibrium distribution function calculated here is

$$g^{eq} = w_i C \left[1 + \frac{3\,(\vec{e}_i \cdot \vec{u})}{c^2} \right]$$

using the solute concentration C and the velocity of the bulk fluid[27]. This subroutine can be changed for other Lattice Boltzmann models, e.g. free energy model[64].

fGetEquilibriumT

- **Header records**
  `int fGetEquilibriumT(double *feq, double *v, double tem)`
- **Function**
  Calculates equilibrium distribution function for temperature.
- **Dependencies**
  None
- **Arguments**

`feq`	output	double pointer
`v`	input	double pointer
`tem`	input	double precision

- **Comments**
  The equilibrium distribution function calculated here is

$$h^{eq} = w_i T \left[1 + \frac{3\,(\vec{e}_i \cdot \vec{u})}{c^2} \right]$$

using the velocity of the bulk fluid[27]. This subroutine can be changed for other Lattice Boltzmann models, e.g. [23].

fGetMomentEquilibriumF

- **Header records**
  `int fGetMomentEquilibriumF(double *meq, double *p, double rho)`
- **Function**
  Calculates equilibrium distribution function in moment space for compressible fluid.

- **Dependencies**
  None

- **Arguments**

`meq`	output	double pointer
`p`	input	double pointer
`rho`	input	double precision

- **Comments**
  The equilibrium distribution function in moment space calculated here is

$$\vec{M}^{eq} = \mathbf{T}\vec{f}^{eq}$$

the exact form of which is dependent on the lattice scheme; given for D2Q9 by [33] and for D3Q15 and D3Q19 by [8]. Parameters for calculating the square of energy (ϵ) and fourth-order moments ($\pi_{\alpha\alpha}$) can be modified by the user in the `lbpMODEL` module.

fGetMomentEquilibriumFIncom

- **Header records**
  `int fGetMomentEquilibriumFIncom(double *meq, double *p, double rho, double rho0)`

- **Function**
  Calculates equilibrium distribution function in moment space for incompressible fluid.

- **Dependencies**
  None

- **Arguments**

`meq`	output	double pointer
`p`	input	double pointer
`rho`	input	double precision
`rho0`	input	double precision

- **Comments**
  The equilibrium distribution function in moment space calculated here is

$$\vec{M}^{eq} = \mathbf{T}\vec{f}^{eq}$$

the exact form of which is dependent on the lattice scheme; given for D2Q9 by [33] and for D3Q15 and D3Q19 by [8]. Parameters for calculating the square of energy (ϵ) and fourth-order moments ($\pi_{\alpha\alpha}$) can be modified by the user in the `lbpMODEL` module.

fGetMomentForce

- **Header records**
  `int fGetMomentForce(double *source, double *v, double *force)`

- **Function**
  Calculates Guo-like forcing terms in moment space.

- **Dependencies**
  None

- **Arguments**

`meq`	output	double pointer
`v`	input	double pointer
`force`	input	double pointer

- **Comments**
  The forcing terms in moment space[49] calculated here are

$$\vec{S} = \mathbf{T}\vec{w} \cdot \left[3\left(\hat{e}_i - \vec{v}\right) + 9\left(\hat{e}_i \cdot \vec{v}\right)\hat{e}_i\right] \cdot \vec{F}$$

  the exact form of which is dependent on the lattice scheme (given for D2Q9, D3Q15 and D3Q19).

fGetMRTCollide

- **Header records**
  `int fGetMRTCollide(double *collide, double omegashear, double omegabulk)`
- **Function**
  Calculates collision vector for Multiple-Relaxation-Time (MRT) scheme with specified fluid relaxation frequencies.
- **Dependencies**
  None
- **Arguments**

`collide`	output	double pointer
`omegashear`	input	double precision
`omegabulk`	input	double precision

- **Comments**
  The exact form of the collision vector $\vec{s}$ is dependent on the lattice scheme; given for D2Q9 by [33] and for D3Q15 and D3Q19 by [8]. Tuneable parameters for calculation stability can be modified by the user in the `lbpMODEL` module.

fInitializeSystem

- **Header records**
  `int fInitializeSystem()`
- **Function**
  Initializes distribution function for lattice system.
- **Dependencies**
  `fGetEquilibriumF`
  `fGetEquilibriumFIncom`
  `fGetEquilibriumC`
  `fGetEquilibriumT`
- **Comments**
  This subroutine as it stands is suitable for initializing most Lattice Boltzmann systems, although the user may modify it if it can otherwise be faster, more stable or more suitable for a particular calculation.

fSiteFluidCollisionBGK

- **Header records**
  `int fSiteFluidCollisionBGK(double* startpos, double *sitespeed, double* bodyforce)`
- **Function**
  Calculates fluid collisions at a grid point using the Bhatnagar-Grook-Krook (BGK) model for compressible fluids.

- **Dependencies**
  `fGetOneMassSite`
  `fReciprocal`
  `fGetEquilibriumF`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer

- **Comments**
  Collisions for each fluid species are carried out using the BGK single relaxation time[4]:

$$(\partial_t + e_{i\alpha}\partial_\alpha) f_i = -\frac{1}{\tau_f}\left(f_i - f_i^{eq}\right)$$

fSiteFluidCollisionBGKSegregation

- **Header records**
  `int fSiteFluidCollisionBGKSegregation(double* startpos, double *sitespeed, double* bodyforce, double* phaseindex)`

- **Function**
  Calculates collisions and phase segregation for the Lishchuk algorithm at a grid point using the Bhatnagar-Grook-Krook (BGK) model for compressible fluids.

- **Dependencies**
  `fGetAllMassSite`
  `fGetTotMassSite`
  `fReciprocal`
  `fGetEquilibriumF`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer
`phaseindex`	input	double pointer

- **Comments**
  Collisions are carried out on a single ‘achromatic’ fluid using the BGK single relaxation time[4]; fluid segregation takes place using the D’Ortona algorithm[9]:

$$f_i^a\left(\vec{x}, t^+\right) = \frac{\rho^a}{\rho} f_i\left(\vec{x}, t^+\right) + \sum_{b\neq a}\beta^{ab} w_i \frac{\rho^a\rho^b}{\rho^2}\hat{e}_i\cdot\hat{n}_{ab}.$$

fSiteFluidIncomCollisionBGK

- **Header records**
  `int fSiteFluidIncomCollisionBGK(double* startpos, double *sitespeed, double* bodyforce)`

- **Function**
  Calculates collisions at a grid point using the Bhatnagar-Grook-Krook (BGK) model for incompressible fluids.

- **Dependencies**
  `fGetOneMassSite`

`fReciprocal`
`fGetEquilibriumFIncom`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer

- **Comments**
Collisions for each fluid species are carried out using the BGK single relaxation time[4]:

$$\left(\partial_t + e_{i\alpha}\partial_\alpha\right) f_i = -\frac{1}{\tau_f}\left(f_i - f_i^{eq}\right)$$

fSiteFluidIncomCollisionBGKSegregation

- **Header records**
`int fSiteFluidIncomCollisionBGKSegregation(double* startpos, double *sitespeed, double* bodyforce, double* phaseindex)`

- **Function**
Calculates collisions and phase segregation for the Lishchuk algorithm at a grid point using the Bhatnagar-Grook-Krook (BGK) model for incompressible fluids.

- **Dependencies**
`fGetAllMassSite`
`fGetAllMassSite`
`fReciprocal`
`fGetEquilibriumFIncom`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer
`phaseindex`	input	double pointer

- **Comments**
Collisions are carried out on a single 'achromatic' fluid using the BGK single relaxation time[4]; fluid segregation takes place using the D'Ortona algorithm[9]:

$$f_i^a\left(\vec{x}, t^+\right) = \frac{\rho^a}{\rho} f_i\left(\vec{x}, t^+\right) + \sum_{b \neq a} \beta^{ab} w_i \frac{\rho^a \rho^b}{\rho^2} \hat{e}_i \cdot \hat{n}_{ab}.$$

fSiteFluidCollisionBGKGuo

- **Header records**
`int fSiteFluidCollisionBGKGuo(double* startpos, double *sitespeed, double* bodyforce)`

- **Function**
Calculates collisions at a grid point using the Bhatnagar-Grook-Krook (BGK) model with the Guo forcing term for compressible fluids.

- **Dependencies**
`fGetOneMassSite`
`fReciprocal`
`fGetEquilibriumF`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer

- **Comments**

  Collisions for each fluid species, solute and the thermal lattice are carried out using the BGK single relaxation time[4], with the following Guo forcing term[19] acting on each fluid:

$$\left(\partial_t + e_{i\alpha}\partial_\alpha\right) f_i = -\frac{1}{\tau_f}\left(f_i - f_i^{eq}\right) + \left(1 - \frac{1}{2\tau_f}\right) w_i \left[3\left(e_{i\alpha} - v_\alpha\right) + 9\left(\hat{e}_i \cdot \vec{v}\right)\right] F_\alpha$$

fSiteFluidCollisionBGKGuoSegregation

- **Header records**

  `int fSiteFluidCollisionBGKGuoSegregation(double* startpos, double *sitespeed, double* bodyforce, double* phaseindex)`

- **Function**

  Calculates collisions and phase segregation for the Lishchuk algorithm at a grid point using the Bhatnagar-Grook-Krook (BGK) model with the Guo forcing term for compressible fluids.

- **Dependencies**

  `fGetAllMassSite`
  `fGetAllMassSite`
  `fReciprocal`
  `fGetEquilibriumF`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer
`phaseindex`	input	double pointer

- **Comments**

  Collisions are carried out on a single 'achromatic' fluid using the BGK single relaxation time[4], with the Guo forcing term[19] also acting on the achromatic fluid; fluid segregation takes place using the D'Ortona algorithm[9]:

$$f_i^a\left(\vec{x}, t^+\right) = \frac{\rho^a}{\rho} f_i\left(\vec{x}, t^+\right) + \sum_{b \neq a} \beta^{ab} w_i \frac{\rho^a \rho^b}{\rho^2} \hat{e}_i \cdot \hat{n}_{ab}.$$

fSiteFluidIncomCollisionBGKGuo

- **Header records**

  `int fSiteFluidIncomCollisionBGKGuo(double* startpos, double *sitespeed, double* bodyforce)`

- **Function**

  Calculates collisions at a grid point using the Bhatnagar-Grook-Krook (BGK) model with the Guo forcing term for incompressible fluids.

- **Dependencies**

  `fGetOneMassSite`
  `fReciprocal`
  `fGetEquilibriumFIncom`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer

- **Comments**
  Collisions for each fluid species are carried out using the BGK single relaxation time[4], with the following Guo forcing term[19] acting on each fluid:

$$(\partial_t + e_{i\alpha}\partial_\alpha) f_i = -\frac{1}{\tau_f}\left(f_i - f_i^{eq}\right) + \left(1 - \frac{1}{2\tau_f}\right) w_i \left[3\left(e_{i\alpha} - v_\alpha\right) + 9\left(\hat{e}_i \cdot \vec{v}\right)\right] F_\alpha$$

fSiteFluidIncomCollisionBGKGuoSegregation

- **Header records**
  `int fSiteFluidIncomCollisionBGKGuoSegregation(double* startpos, double *sitespeed, double* bodyforce, double* phaseindex)`

- **Function**
  Calculates collisions and phase segregation for the Lishchuk algorithm at a grid point using the Bhatnagar-Grook-Krook (BGK) model with the Guo forcing term for incompressible fluids.

- **Dependencies**
  `fGetAllMassSite`
  `fGetTotMassSite`
  `fReciprocal`
  `fGetEquilibriumFIncom`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer
`phaseindex`	input	double pointer

- **Comments**
  Collisions are carried out on a single ‘achromatic’ fluid using the BGK single relaxation time[4], with the Guo forcing term[19] also acting on the achromatic fluid; fluid segregation takes place using the D’Ortona algorithm[9]:

$$f_i^a\left(\vec{x}, t^+\right) = \frac{\rho^a}{\rho} f_i\left(\vec{x}, t^+\right) + \sum_{b \neq a} \beta^{ab} w_i \frac{\rho^a \rho^b}{\rho^2} \hat{e}_i \cdot \hat{n}_{ab}.$$

fSiteFluidCollisionMRT

- **Header records**
  `int fSiteFluidCollisionMRT(double* startpos, double *sitespeed, double* bodyforce)`

- **Function**
  Calculates collisions at a grid point using Multiple-Relaxation-Time (MRT) models for compressible fluids.

- **Dependencies**
  `fGetOneMassSite`
  `fGetSpeedSite`
  `fGetMomentEquilibriumF`
  `fGetMRTCollide`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer

- **Comments**

  Collisions for each fluid species are carried out using multiple-relaxation-time (MRT) schemes[33, 8]:

$$\left(\partial_t + e_{i\alpha}\partial_\alpha\right) f_i = \mathbf{T}^{-1}\left[-\vec{s}\left(\mathbf{T}\vec{f} - \vec{M}_i^{eq}\right)\right].$$

fSiteFluidCollisionMRTSegregation

- **Header records**

  `int fSiteFluidCollisionMRTSegregation(double* startpos, double *sitespeed, double* bodyforce, double* phaseindex)`

- **Function**

  Calculates collisions and phase segregation for the Lishchuk algorithm at a grid point using Multiple-Relaxation-Time (MRT) models for compressible fluids.

- **Dependencies**

  `fGetAllMassSite`
  `fGetTotMassSite`
  `fReciprocal`
  `fGetMomentEquilibriumF`
  `fGetMRTCollide`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer
`phaseindex`	input	double pointer

- **Comments**

  Collisions are carried out on a single 'achromatic' fluid using multiple-relaxation-time (MRT) schemes[33, 8]; fluid segregation takes place using the D'Ortona algorithm[9]:

$$f_i^a\left(\vec{x}, t^+\right) = \frac{\rho^a}{\rho} f_i\left(\vec{x}, t^+\right) + \sum_{b \neq a} \beta^{ab} w_i \frac{\rho^a \rho^b}{\rho^2} \hat{e}_i \cdot \hat{n}_{ab}.$$

fSiteFluidIncomCollisionMRT

- **Header records**

  `int fSiteFluidIncomCollisionMRT(double* startpos, double *sitespeed, double* bodyforce)`

- **Function**

  Calculates collisions at a grid point using Multiple-Relaxation-Time (MRT) models for incompressible fluids.

- **Dependencies**

  `fGetOneMassSite`
  `fGetMomentEquilibriumFIncom`
  `fGetMRTCollide`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer

- **Comments**
  Collisions for each fluid species are carried out using multiple-relaxation-time (MRT) schemes[33, 8]:

$$\left(\partial_t + e_{i\alpha}\partial_\alpha\right) f_i = \mathbf{T}^{-1}\left[-\vec{s}\left(\mathbf{T}\vec{f} - \vec{M}_i^{eq}\right)\right].$$

fSiteFluidIncomCollisionMRTSegregation

- **Header records**
  `int fSiteFluidIncomCollisionMRTSegregation(double* startpos, double *sitespeed, double* bodyforce, double* phaseindex)`

- **Function**
  Calculates collisions and phase segregation for the Lishchuk algorithm at a grid point using Multiple-Relaxation-Time (MRT) models for incompressible fluids.

- **Dependencies**
  `fGetAllMassSite`
  `fGetTotMassSite`
  `fReciprocal`
  `fGetMomentEquilibriumFIncom`
  `fGetMRTCollide`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer
`phaseindex`	input	double pointer

- **Comments**
  Collisions are carried out on a single ‘achromatic’ fluid using multiple-relaxation-time (MRT) schemes[33, 8]; fluid segregation takes place using the D’Ortona algorithm[9]:

$$f_i^a\left(\vec{x}, t^+\right) = \frac{\rho^a}{\rho} f_i\left(\vec{x}, t^+\right) + \sum_{b \neq a} \beta^{ab} w_i \frac{\rho^a \rho^b}{\rho^2} \hat{e}_i \cdot \hat{n}_{ab}.$$

fSiteFluidCollisionMRTGuo

- **Header records**
  `int fSiteFluidCollisionMRTGuo(double* startpos, double *sitespeed, double* bodyforce)`

- **Function**
  Calculates collisions at a grid point using Multiple-Relaxation-Time (MRT) models coupled with Guo-like forcing terms for compressible fluids.

- **Dependencies**
  `fGetOneMassSite`
  `fGetMomentEquilibriumF`
  `fGetMomentForce`
  `fGetMRTCollide`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer

- **Comments**
Collisions for each fluid species are carried out using multiple-relaxation-time (MRT) schemes coupled with Guo-like forcing terms[49]:

$$\left(\partial_t + e_{i\alpha}\partial_\alpha\right) f_i = \mathbf{T}^{-1}\left[-\vec{s}\left(\mathbf{T}\vec{f} - \vec{M}_i^{eq}\right) + \left(\mathbf{I} - \tfrac{1}{2}\mathbf{I}\vec{s}\right)\vec{S}\right]$$

fSiteFluidCollisionMRTGuoSegregation

- **Header records**
`int fSiteFluidCollisionMRTGuoSegregation(double* startpos, double *sitespeed, double* bodyforce, double* phaseindex)`

- **Function**
Calculates collisions and phase segregation for the Lishchuk algorithm at a grid point using Multiple-Relaxation-Time (MRT) models coupled with Guo-like forcing terms for compressible fluids.

- **Dependencies**
`fGetAllMassSite`
`fGetTotMassSite`
`fReciprocal`
`fGetMomentEquilibriumF`
`fGetMomentForce`
`fGetMRTCollide`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer
`phaseindex`	input	double pointer

- **Comments**
Collisions are carried out on a single ‘achromatic’ fluid using multiple-relaxation-time (MRT) schemes coupled with Guo-like forcing terms[49]; fluid segregation takes place using the D’Ortona algorithm[9]:

$$f_i^a\left(\vec{x}, t^+\right) = \frac{\rho^a}{\rho} f_i\left(\vec{x}, t^+\right) + \sum_{b\neq a} \beta^{ab} w_i \frac{\rho^a\rho^b}{\rho^2}\hat{e}_i \cdot \hat{n}_{ab}.$$

fSiteFluidIncomCollisionMRTGuo

- **Header records**
`int fSiteFluidIncomCollisionMRTGuo(double* startpos, double *sitespeed, double* bodyforce)`

- **Function**
Calculates collisions at a grid point using Multiple-Relaxation-Time (MRT) models coupled with Guo-like forcing terms for incompressible fluids.

- **Dependencies**
`fGetOneMassSite`
`fGetMomentEquilibriumFIncom`
`fGetMomentForce`
`fGetMRTCollide`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer

- **Comments**
Collisions are carried out using multiple-relaxation-time (MRT) schemes coupled with Guo-like forcing terms[49]:

$$\left(\partial_t + e_{i\alpha}\partial_\alpha\right) f_i = \mathbf{T}^{-1}\left[-\vec{s}\left(\mathbf{T}\vec{f} - \vec{M}_i^{eq}\right) + \left(\mathbf{I} - \tfrac{1}{2}\mathbf{I}\vec{s}\right)\vec{S}\right]$$

fSiteFluidIncomCollisionMRTGuoSegregation

- **Header records**
`int fSiteFluidIncomCollisionMRTGuoSegregation(double* startpos, double *sitespeed, double* bodyforce, double* phaseindex)`

- **Function**
Calculates collisions and phase segregation for the Lishchuk algorithm at a grid point using Multiple-Relaxation-Time (MRT) models coupled with Guo-like forcing terms for incompressible fluids.

- **Dependencies**
`fGetAllMassSite`
`fGetTotMassSite`
`fReciprocal`
`fGetMomentEquilibriumFIncom`
`fGetMomentForce`
`fGetMRTCollide`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer
`phaseindex`	input	double pointer

- **Comments**
Collisions are carried out on a single 'achromatic' fluid using multiple-relaxation-time (MRT) schemes coupled with Guo-like forcing terms[49]; fluid segregation takes place using the D'Ortona algorithm[9]:

$$f_i^a\left(\vec{x}, t^+\right) = \frac{\rho^a}{\rho} f_i\left(\vec{x}, t^+\right) + \sum_{b \neq a} \beta^{ab} w_i \frac{\rho^a \rho^b}{\rho^2} \hat{e}_i \cdot \hat{n}_{ab}.$$

fSiteSoluteCollisionBGK

- **Header records**
`int fSiteSoluteCollisionBGK(double* startpos, double *sitespeed)`

- **Function**
Calculates solute collisions at a grid point using the Bhatnagar-Grook-Krook (BGK) model for compressible fluids.

- **Dependencies**
`fGetOneMassSite`
`fGetEquilibriumC`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles

- **Comments**
Collisions for each solute are carried out using the BGK single relaxation time[4]:

$$(\partial_t + e_{i\alpha}\partial_\alpha)\, g_i = -\frac{1}{\tau_s}\left(g_i - g_i^{eq}\right)$$

fSiteThermalCollisionBGK

- **Header records**
`int fSiteThermalCollisionBGK(double* startpos, double *sitespeed)`

- **Function**
Calculates thermal collisions at a grid point using the Bhatnagar-Grook-Krook (BGK) model for compressible fluids.

- **Dependencies**
`fGetOneMassSite`
`fGetEquilibriumT`

- **Arguments**

`startpos`	input	double pointer
`sitespeed`	input	array of doubles
`bodyforce`	input	double pointer

- **Comments**
Collisions for thermal currents are carried out using the BGK single relaxation time[4]:

$$(\partial_t + e_{i\alpha}\partial_\alpha)\, h_i = -\frac{1}{\tau_t}\left(h_i - h_i^{eq}\right)$$

fCollisionBGK

- **Header records**
`int fCollisionBGK()`

- **Function**
Collision steps for all compressible and incompressible fluids using BGK model.

- **Dependencies**
`fSiteFluidCollisionBGK`
`fSiteFluidIncomCollisionBGK`
`fSiteSoluteCollisionBGK`
`fSiteThermalCollisionBGK`
`fGetSpeedSite`
`fGetSpeedIncomSite`

- **Comments**
This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionBGKSegregation

- **Header records**
`int fCollisionBGKSegregation()`

- **Function**
  Collision and segregation steps for all compressible and incompressible fluids using BGK model with Lishchuk mesophase interactions.

- **Dependencies**
  `fSiteFluidCollisionBGKSegregation`
  `fSiteFluidIncomCollisionBGKSegregation`
  `fSiteSoluteCollisionBGK`
  `fSiteThermalCollisionBGK`
  `fGetOneMassSite`
  `fGetTotMassSite`
  `fGetSpeedSite`
  `fGetSpeedIncomSite`

- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionBGKGuo

- **Header records**
  `int fCollisionBGKGuo()`

- **Function**
  Collision steps for all compressible and incompressible fluids using BGK model with Guo forcing terms.

- **Dependencies**
  `fSiteFluidCollisionBGKGuo`
  `fSiteFluidIncomCollisionBGKGuo`
  `fSiteSoluteCollisionBGK`
  `fSiteThermalCollisionBGK`
  `fGetSpeedSite`
  `fGetSpeedIncomSite`

- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionBGKGuoSegregation

- **Header records**
  `int fCollisionBGKGuoSegregation()`

- **Function**
  Collision and segregation steps for all compressible and incompressible fluids using BGK model with Guo forcing terms and Lishchuk mesophase interactions.

- **Dependencies**
  `fSiteFluidCollisionBGKGuoSegregation`
  `fSiteFluidIncomCollisionBGKGuoSegregation` `fSiteSoluteCollisionBGK`
  `fSiteThermalCollisionBGK`
  `fGetOneMassSite`
  `fGetTotMassSite`
  `fGetSpeedSite`
  `fGetSpeedIncomSite`

- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionMRT

- **Header records**
  `int fCollisionMRT()`
- **Function**
  Collision steps for all compressible and incompressible fluids using MRT model.
- **Dependencies**
  `fSiteFluidCollisionMRT`
  `fSiteFluidIncomCollisionMRT`
  `fSiteSoluteCollisionBGK`
  `fSiteThermalCollisionBGK`
  `fGetSpeedSite`
  `fGetSpeedIncomSite`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionMRTSegregation

- **Header records**
  `int fCollisionMRTSegregation()`
- **Function**
  Collision and segregation steps for all compressible and incompressible fluids using MRT model with Lishchuk mesophase interactions.
- **Dependencies**
  `fSiteFluidCollisionMRTSegregation`
  `fSiteFluidIncomCollisionMRTSegregation`
  `fSiteSoluteCollisionBGK`
  `fSiteThermalCollisionBGK`
  `fGetOneMassSite`
  `fGetTotMassSite`
  `fGetSpeedSite`
  `fGetSpeedIncomSite`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionMRTGuo

- **Header records**
  `int fCollisionMRTGuo()`
- **Function**
  Collision steps for all compressible and incompressible fluids using MRT model with Guo-like forcing terms.

- **Dependencies**
  `fSiteFluidCollisionMRTGuo`
  `fSiteFluidIncomCollisionMRTGuo`
  `fSiteSoluteCollisionBGK`
  `fSiteThermalCollisionBGK`
  `fGetSpeedSite`
  `fGetSpeedIncomSite`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionMRTGuoSegregation

- **Header records**
  `int fCollisionMRTGuoSegregation()`
- **Function**
  Collision and segregation steps for all compressible and incompressible fluids using MRT model with Guo-like forcing terms and Lishchuk mesophase interactions.
- **Dependencies**
  `fSiteFluidCollisionMRTGuoSegregation`
  `fSiteFluidIncomCollisionMRTGuoSegregation`
  `fSiteSoluteCollisionBGK`
  `fSiteThermalCollisionBGK`
  `fGetOneMassSite`
  `fGetTotMassSite`
  `fGetSpeedSite`
  `fGetSpeedIncomSite`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fPropagationTwoLattice

- **Header records**
  `int fPropagationTwoLattice()`
- **Function**
  Moves lattice particles (distribution functions) to neighbouring grid points using the two-lattice algorithm.
- **Dependencies**
  None
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified. This is the least efficient propagation routine available.

fPropagationSwap

- **Header records**
  `int fPropagationSwap()`
- **Function**
  Moves lattice particles (distribution functions) to neighbouring grid points using the swap algorithm.

- **Dependencies**
  `fSwapPair`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified. Propagation is carried out by systematic swapping of post-collisional values for the distribution function, initially at each grid point and then between them (in two separate loops), as described by [43] and in section 5.1. This version can be used for either serial or parallel calculations and no boundary layer is necessary: this is the default propagation routine for serial calculations.

fPropagationCombinedSwap

- **Header records**
  `int fPropagationCombinedSwap()`
- **Function**
  Moves lattice particles (distribution functions) to neighbouring grid points using the swap algorithm.
- **Dependencies**
  `fSwapPair`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified. Propagation is carried out by systematic swapping of post-collisional values for the distribution function, initially at each grid point and then between them (in the same loop), as described by [43] and in section 5.1. This version can only be used for calculations with non-zero boundary layers: this is the default propagation routine for parallel calculations.

7.2.12 lbpMPI

This package is only required for parallel running and does not require detailed knowledge for its use. Several subroutines in this package are not described here: interested users should consult the code for further information.

fStartMPI

- **Header records**
  `int fStartMPI(int argc, char* argv[])`
- **Function**
  Starts Message Passing Interface (MPI).
- **Dependencies**
  None

fCloseMPI

- **Header records**
  `int fCloseMPI()`
- **Function**
  Closes Message Passing Interface (MPI).
- **Dependencies**
  None

fGlobalValue

- **Header records** (six cases)
  ```
  int fGlobalValue(double *vqua, int nnum)
  int fGlobalValue(int *vqua, int nnum, int *vtot)
  int fGlobalValue(int *vqua, int nnum)
  int fGlobalValue(long int *vqua, int nnum)
  int fGlobalValue(long int *vqua, int nnum, long int *vtot)
  ```
- **Function**
  Sums values from all processes and distributes the sum.

fGlobalProduct

- **Header records** (two cases)
  ```
  int fGlobalProduct(double *vqua, int nnum)
  int fGlobalProduct(int *vqua, int nnum)
  ```
- **Function**
  Multiplies together values from all processors and distributes the product.

fArrangeProcessors

- **Header records**
  `int fArrangeProcessors()`
- **Function**
  Arrange processors according to system dimensions.
- **Comments**
  Calculations are based on
  $$\frac{\texttt{lbdm.xdim}}{\texttt{lbsy.nx}} \simeq \frac{\texttt{lbdm.ydim}}{\texttt{lbsy.ny}} \simeq \frac{\texttt{lbdm.zdim}}{\texttt{lbsy.nz}}$$
  $$\texttt{lbdm.xdim} \times \texttt{lbdm.ydim} \times \texttt{lbdm.zdim} = \texttt{lbdm.size}$$

fDefineDomain

- **Header records**
  `int fDefineDomain()`
- **Function**
  Determines domain parameters for system calculation.

fDefineMessage

- **Header records**
  `int fDefineMessage()`
- **Function**
  Defines vector messages for system (distribution functions, boundary properties and interaction forces).

fDefineNeighbours

- **Header records**
  `int fDefineNeighbours()`
- **Function**
  Calculates the names of neighbouring processes and the start points for sending and receiving messages.
- **Comments**
  This subroutine must *not* be changed!

fNonBlockCommunication

- **Header records**
  `int fNonBlockCommunication()`
- **Function**
  Passes distribution function information for either 2D or 3D system.

fOutputInfo

- **Header records**
  `int fOutputInfo()`
- **Function**
  Outputs number of processes and lengths of integers and floats.
- **Comments**
  This subroutine is necessary for gathering and rearranging the `lbout` data, and produces the files `lbout.info` and `lbout.ext`.

fBoundNonBlockCommunication

- **Header records**
  `int fBoundNonBlockCommunication()`
- **Function**
  Passes boundary information for either 2D or 3D systems.

fForceNonBlockCommunication

- **Header records**
  `int fForceNonBlockCommunication()`
- **Function**
  Passes interaction force information for either 2D or 3D systems.

fIndexNonBlockCommunication

- **Header records**
  `int fIndexNonBlockCommunication()`
- **Function**
  Passes phase index information for either 2D or 3D systems.

fCheckTimeMPI

- **Header records**
  `int fCheckTimeMPI()`
- **Function**
  Outputs time in seconds since initial call.
- **Arguments**

`fCheckTimeMPI`	output	double

- **Comments**
  Obtains calculation time based on MPI wall clock.

fPrintSystemMass

- **Header records**
  `int fPrintSystemMass()`
- **Function**
  Calculates and prints total and individual fluid masses in entire system.
- **Dependencies**
  `fGetTotMassDomain`
  `fGetOneMassDomain`
  `fGlobalValue`

fPrintSystemMomentum

- **Header records**
  `int fPrintSystemMomentum()`
- **Function**
  Calculates and prints the total fluid momentum in entire system.
- **Dependencies**
  `fGetTotMomentDomain`
  `fGlobalValue`

Chapter 8

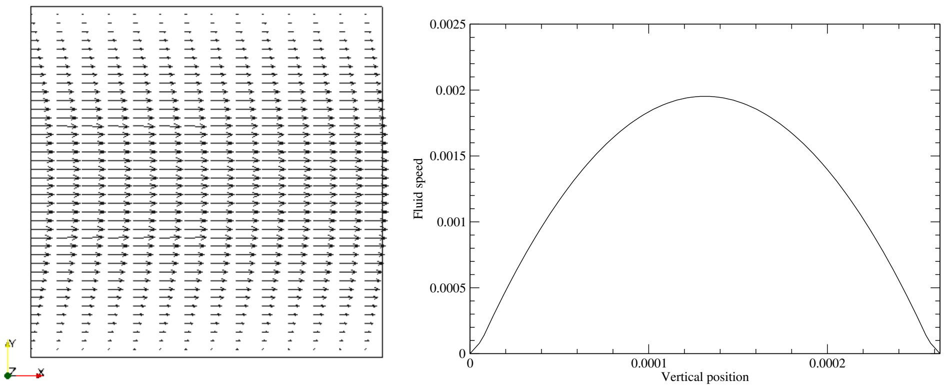
DL_MESO LBE Examples

Test cases for Lattice Boltzmann Equation simulations using DL_MESO – including the required input and sample output files – can be found in the `DEMO/LBE` subdirectory. They can be run using either the serial or parallel versions of DL_MESO_LBE.

Images of all test cases and videos for some can be found in the Example Simulations page of the DL_MESO website: a link to it can be found at `www.ccp5.ac.uk/DL_MESO`

8.1 2D_Pressure

This is a 2D simulation of a single fluid on a 42×42 grid with fixed pressure (density) boundary conditions on the left and right boundaries and solid walls (represented by bounce back) at the top and bottom. A visualization with vector glyphs and a plot of fluid speed against vertical position can be seen in Figure 8.1, which show the boundary conditions result in a laminar flow with a parabolic velocity profile.

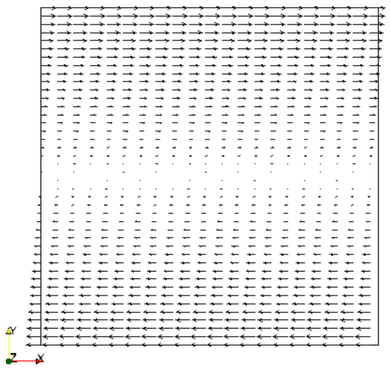


(a) Vector plot of system (b) Variation of fluid speed with vertical position

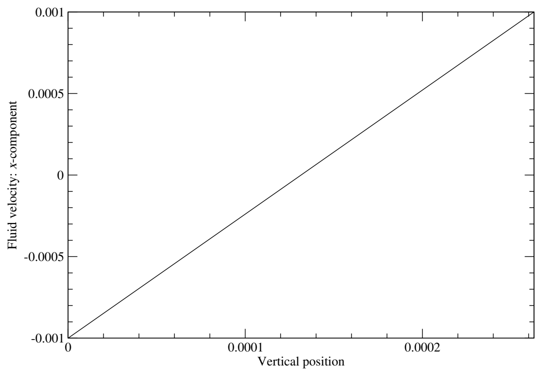
Figure 8.1: Results from LBE `2D_Pressure` test case

8.2 2D_Shear

This is a 2D simulation of a single fluid on a 42×42 grid with a shear boundary condition. The vector plot in Figure 8.2 demonstrates the ability of the applied boundary conditions to generate a linear shearing Couette flow throughout the grid, which is confirmed by the plot of horizontal velocity component as a function of vertical position at the last time step for the simulation.



(a) Vector plot of system



(b) Variation of horizontal velocity with vertical position

Figure 8.2: Results from LBE `2D_Shear` test case

8.3 2D_CylinderFlow

This is a 2D simulation of a single fluid on a 125 $\times$ 50 grid with a constant horizontal body force and a circular obstacle of radius 12, representing channel flow past an infinitely-long cylinder. Figure 8.3 shows this flow against a density map of the system, with solid black lines representing the solid boundaries (both walls and the cylinder).

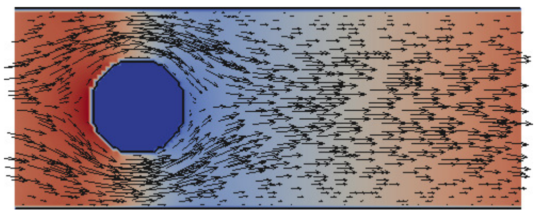


Figure 8.3: Density (scale: blue to red) and velocity vector plot from LBE `2D_CylinderFlow` test case

8.4 2D_KarmanVortex

This is a 2D simulation of a single fluid on a 250 $\times$ 50 grid with a constant horizontal body force and a circular obstacle of radius 8, representing channel flow past an infinitely-long cylinder that eventually produces a von Kármán vortex street between two solid walls. Figure 8.4 shows the flow field at the final time step: an `.AVI` video file has been rendered from the calculation and can be found in the Example Simulations page of the DL_MESO website.

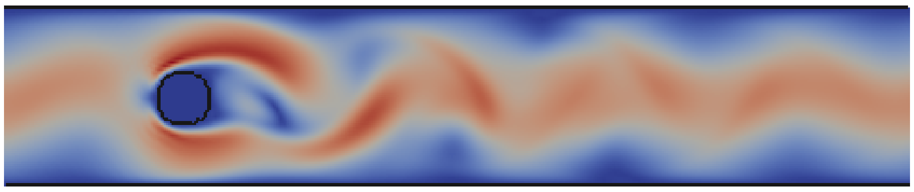


Figure 8.4: Velocity magnitude plot from LBE `2D_KarmanVortex` test case (scale: blue to red)

8.5 2D_LidCavity

This is a 2D simulation of a single incompressible fluid on a 128 $\times$ 128 grid with a shear boundary condition at the top and solid walls surrounding the other edges of the system, resulting in lid-driven cavity flow. Figure 8.5

shows the fully-developed velocity field for a Reynolds number of 100 at the final time step.

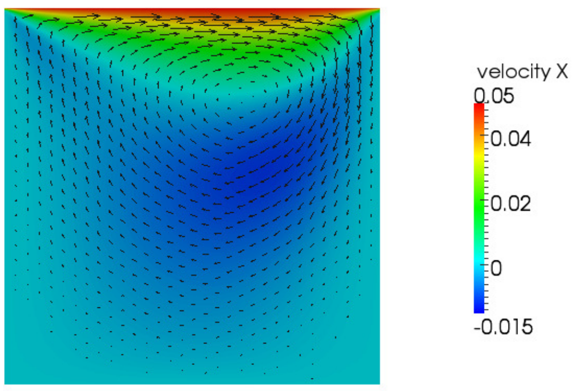


Figure 8.5: Magnitude plot of x-component velocity and velocity vector plot from LBE `2D_Lidcavity` test case

8.6 2D_RayleighBenard

This is a 2D simulation of a single fluid undergoing natural (Rayleigh-Bénard) convection on a 102×51 grid. The fluid is contained between two solid walls: the wall at the bottom of the system is maintained at a higher temperature than that at the top. Figure 8.6 shows the fully-developed temperature field at the final time step for a Prandtl number of 1 and a Rayleigh number of ~ 21250.

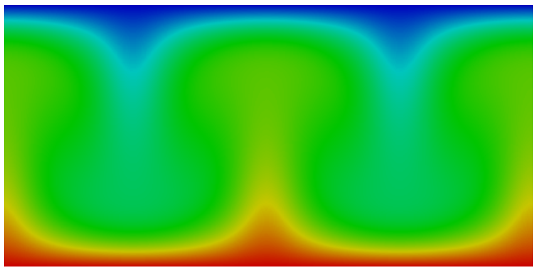
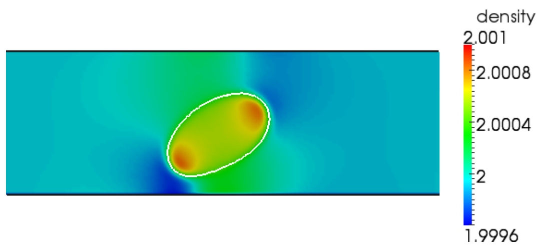


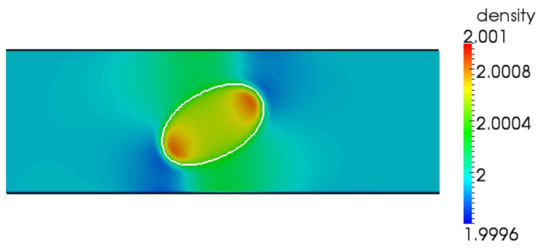
Figure 8.6: Plot of fluid temperature for LBE `2D_RayleighBenard` test case (scale: blue to red)

8.7 2D_DropShear

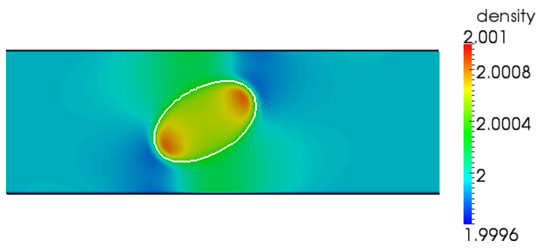
This is a 2D simulation of an initially static drop on a 150×50 grid undergoing linear shear[21] using Lishchuk continuum-based mesophase interactions with Guo forcing. The drop and continuous fluid are contained between two solid walls: after an equilibration period to allow the drop shape to settle, the wall at the top of the system is set to move horizontally while the wall at the bottom is kept stationary. Figure 8.7 shows the fluid density (pressure) field and drop positions at time steps throughout the simulation, demonstrating traverse migration (lift) due to linear shear, for a system with droplet Reynolds number of 0.135 and capillary number (ratio of inertial to interfacial stresses) of 0.147. An `.AVI` video file has been rendered from an example calculation, which can be found in the Example Simulations page of the DL_MESO website.



(a) $t = 19\ 000$



(b) $t = 87\ 500$



(c) $t = 390\ 000$

Figure 8.7: Plots of fluid density (pressure) and drop positions for LBE `2D_DropShear` test case

8.8 3D_PhaseSeparation

This is a 3D simulation of two fluids on a $100 \times 100 \times 100$ grid with periodic boundary conditions and Shan/Chen pseudopotential mesoscopic interactions that cause the fluids to separate. Figure 8.8 shows the phase separation process in a number of snapshots: two `.AVI` video files have been rendered from an example calculation (one giving a 3D view of the system, the other showing a plane normal to the y-axis) which can be found in the Example Simulations page of the DL_MESO website.

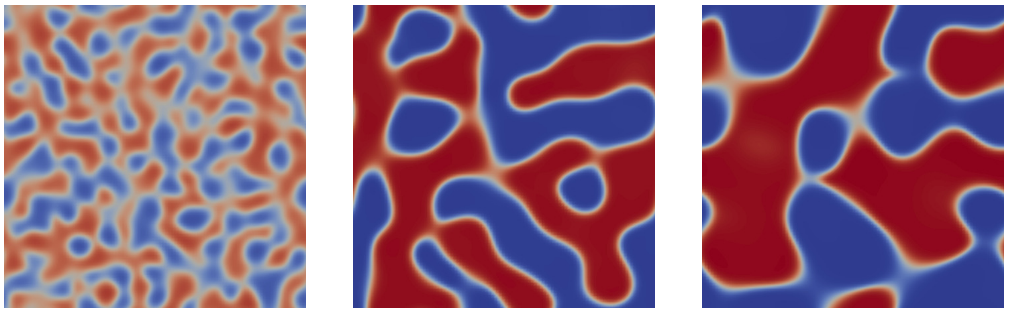
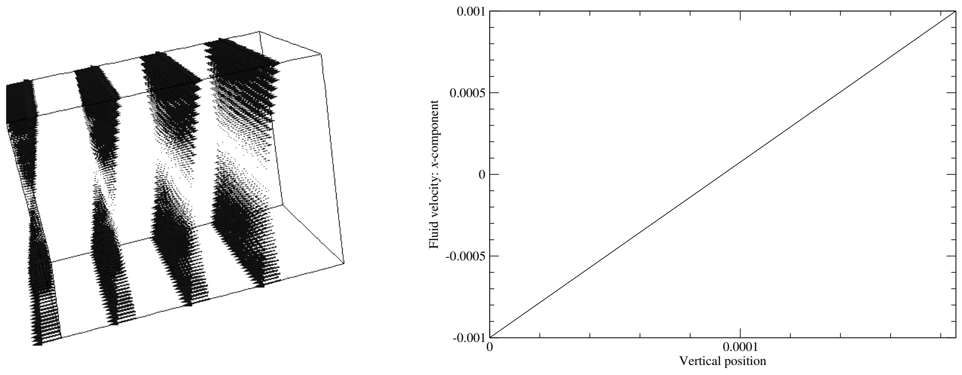

(a) $t = 400$ (b) $t = 1200$ (c) $t = 2000$

Figure 8.8: Progressive density plots in plane normal to y-axis from LBE `3D_PhaseSeparation` test case (red for fluid 0, blue for fluid 1)

8.9 3D_Shear

This is a 3D simulation of a single fluid on a $40 \times 30 \times 25$ grid with a shear boundary condition. Figure 8.9 shows a vector plot for this system, demonstrating that linear shear is generated and maintained by the moving boundaries in the planes normal to the y-axis, and a plot of the horizontal component of fluid velocity against vertical position at the last time step.



(a) Vector plot of system (b) Plot of horizontal velocity with vertical position

Figure 8.9: Results from LBE `3D_Shear` test case

8.10 3D_RayleighBenard

This is a 3D simulation of a single fluid undergoing natural (Rayleigh-Bénard) convection on a $80 \times 40 \times 80$ grid. The fluid is contained between two solid walls: the wall at the bottom of the system is maintained at a higher temperature than that at the top. Figure 8.10 shows the fully-developed temperature and convective flow fields at the final time step for a Prandtl number of 1 and a Rayleigh number of ~ 10000.

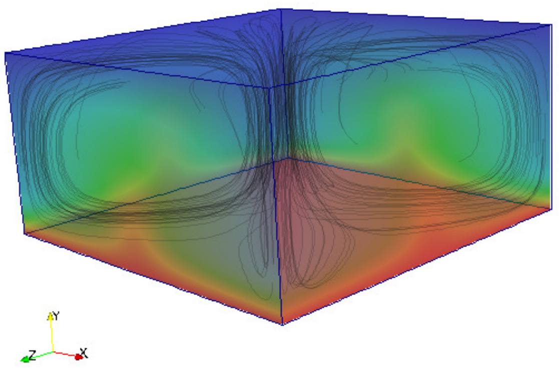


Figure 8.10: Plot of fluid temperature for LBE `3D_RayleighBenard` test case (scale: blue to red) with streamlines depicting convective flow

Part II

Dissipative Particle Dynamics (DPD)

Chapter 9

Dissipative Particle Dynamics: Basic Theory

9.1 Introduction

Dissipative Particle Dynamics (DPD) is an off-lattice, discrete particle method for modelling mesoscopic systems. It has little in common with Lattice Boltzmann methods, except in its application to systems of similar length and time scales.

The DPD method inherits its methodology from classical Molecular Dynamics (MD), particularly from Brownian Dynamics (BD). It differs from BD, however, in an important way: it is *Galilean invariant* and for this reason conserves hydrodynamic behaviour, while the BD method does not. Many systems are crucially dependent on hydrodynamic interactions and it is essential to retain this feature in the model. DPD is particularly useful for simulating systems on the near-molecular scale, such as polymers, biopolymers, lipids, emulsions and surfactants – systems in which large scale structure evolves on a time scale that is too long to be modelled effectively by MD. DPD may also be used when such systems experience shear and flow gradients.

The DPD algorithm can be summarized by the following:

- A condensed phase system may be modelled as a system of free particles interacting directly through 'soft' forces.
- The system is coupled to a heat bath via stochastic forces, which act on the particles in a pairwise manner.
- The particles also experience a damping or drag force, which also acts in a pairwise manner.
- Thermodynamic equilibrium is maintained through the balance of the stochastic and drag forces, i.e. the method satisfies the fluctuation-dissipation theorem.
- At equilibrium (or steady state) the properties of the system are calculated as averages over the individual particles, as in Molecular Dynamics.

9.2 Outline of Method

In DPD[1] the system is modelled as a system of free particles, which are spherical and interact over a range that is of the same order as their diameters. The particles can be thought of as assemblies or aggregates of molecules, such as solvent molecules or polymers, or more simply as carriers of momentum.

[1]The outline of the DPD method supplied here is based on [16].

The equations governing the time evolution in a DPD simulation resemble those of ordinary MD:

$$\frac{d\vec{v}_i}{dt} = \frac{\vec{F}_i}{m_i} \tag{9.1}$$

$$\frac{d\vec{r}_i}{dt} = \vec{v}_i \tag{9.2}$$

in which $\vec{r}_i$, $\vec{v}_i$ and $\vec{F}_i$ are the position, velocity and force of the ith particle, which has mass m_i. The force on the particle is a sum of pair forces:

$$\vec{F}_i = \sum_{j \neq i}^{N} \left(\vec{F}_{ij}^{C} + \vec{F}_{ij}^{D} + \vec{F}_{ij}^{R} \right) \tag{9.3}$$

in which $\vec{F}_{ij}^{C}$, $\vec{F}_{ij}^{D}$ and $\vec{F}_{ij}^{R}$ are the *conservative*, *drag* and *random* (or *stochastic*) pair forces respectively. Each represents the force exerted on particle i due to the presence of particle j. Additional pairwise forces may be included for more complicated systems, such as those involving chains of particles bonded together[52].

The conservative interactions are usually 'soft' (i.e. weakly interacting) so that the particles can pass by each other (or even through each other) relatively easily so that equilibrium is achieved quickly. A common form of interaction potential is an inverse parabola:

$$V(r_{ij}) = \frac{1}{2} A_{ij} r_c \left(1 - \frac{r_{ij}}{r_c}\right)^2 \tag{9.4}$$

where $r_{ij} = |\vec{r}_j - \vec{r}_i|$, r_c is a cutoff radius and A_{ij} is the interaction strength. A_{ij} may be the same for all particle pairs or may be different for different particle types.

Equation (9.4) gives rise to a repulsive force of the form

$$\vec{F}_{ij}^{C} = A_{ij} w^C(r_{ij}) \frac{\vec{r}_{ij}}{r_{ij}} = A_{ij} \left(1 - \frac{r_{ij}}{r_c}\right) \frac{\vec{r}_{ij}}{r_{ij}} \tag{9.5}$$

This is the deterministic or *conservative* force $\vec{F}_{ij}^{C}$ exerted on particle i by particle j. Note the switching function $w^C(r_{ij})$ and the force are zero when $r_{ij} > r_c$ and thus the particles have an effective diameter of 1 in units of the cutoff radius r_c.

The stochastic forces experienced by the particles is again pairwise in nature and takes the form

$$\vec{F}_{ij}^{R} = \sigma_{ij} w^R(r_{ij}) \zeta_{ij} \Delta t^{-\frac{1}{2}} \frac{\vec{r}_{ij}}{r_{ij}} \tag{9.6}$$

in which Δt is the time step and $w^R(r_{ij})$ is a switching function which imposes a finite limit on the range of the stochastic force. ζ_{ij} is a random number with zero mean and unit variance. The constant σ_{ij} is related to the temperature, as is understood from the role of the stochastic force in representing a heat bath.

Finally the particles are subject to a drag force, which depends on the relative velocity between interacting pairs of particles:

$$\vec{F}_{ij}^{D} = -\gamma_{ij} w^D(r_{ij}) (\vec{r}_{ij} \cdot \vec{v}_{ij}) \frac{\vec{r}_{ij}}{r_{ij}^2} \tag{9.7}$$

where $w^D(r_{ij})$ is once again a switching function and $\vec{v_{ij}} = \vec{v_j} - \vec{v_i}$. The constant γ_{ij} is the drag coefficient. It follows from the fluctuation-dissipation theorem that for thermodynamic equilibrium to result from this method the following relations must hold.

$$\sigma_{ij}^2 = 2\gamma_{ij} k_B T \tag{9.8}$$

$$w^D(r_{ij}) = \left[w^R(r_{ij})\right]^2 \tag{9.9}$$

In practice the switching functions are defined through

$$w^D(r_{ij}) = \left(1 - \frac{r_{ij}}{r_c}\right)^2 \qquad (r_{ij} < r_c) \tag{9.10}$$

which ensures that all interactions are switched off at the range $r_{ij} = r_c$. In many DPD simulations, the stochastic and drag coefficients are often constant for all interactions, i.e. $\sigma_{ij} \equiv \sigma$ and $\gamma_{ij} \equiv \gamma$, although this assumption does not have to apply.

9.3 Equation of state and dynamic properties

The form of the conservative force determines the equation of state for a DPD fluid, which can be derived using the virial theorem to express system pressure as follows:

$$\begin{aligned} p &= \rho k_B T + \frac{1}{3V}\left\langle \sum_{j>i} (\vec{r}_i - \vec{r}_j) \cdot \vec{F}_{ij}^C \right\rangle && (9.11) \\ &= \rho k_B T + \frac{2\pi}{3}\rho^2 \int_0^{r_c} rA\left(1 - \frac{r}{r_c}\right) g(r) r^2 \, dr && (9.12) \end{aligned}$$

where $g(r)$ is a radial distribution function for the soft sphere model[16] and ρ is the DPD particle density. For sufficiently large densities ($\rho > 2$), $g(r)$ takes the same form and the equation of state can be well-approximated by:

$$p = \rho k_B T + \alpha A \rho^2 \tag{9.13}$$

where the parameter $\alpha \approx 0.101 \pm 0.001$ has units equivalent to r_c^4. This expression permits the use of fluid compressibilities to obtain conservative force parameters for bulk fluids, e.g. for water $A \approx \frac{75 k_B T}{\rho}$. Alternative equations of state may be obtained by modifying the functional form of conservative interactions to include localized densities (i.e. many-body DPD)[46, 66].

Transport coefficients for a DPD fluid can be derived using the expressions for the drag and stochastic forces[16, 32, 40]. The kinematic viscosity can be found to be

$$\nu \approx \frac{45 k_B T}{4\pi\gamma\rho r_c^3} + \frac{2\pi\gamma\rho r_c^5}{1575} \tag{9.14}$$

while the self-diffusion coefficient is given as

$$D \approx \frac{45 k_B T}{2\pi\gamma\rho r_c^3}. \tag{9.15}$$

The ratio of these two properties, the Schmidt number ($\mathrm{Sc} = \frac{\nu}{D}$), is therefore:

$$\mathrm{Sc} \approx \frac{1}{2} + \frac{(2\pi\gamma\rho r_c^4)^2}{70875 k_B T} \tag{9.16}$$

and for values of the drag coefficient and density frequently used in DPD simulations, this value is of the order of unity, which is an appropriate magnitude for gases but three orders of magnitude too small for liquids.

This property of standard DPD does *not* rule it out for simulations of liquid phases except when hydrodynamics are important. It may also be argued that the self-diffusion of DPD particles might not correspond to that of individual molecules and thus a Schmidt number of the order 10^3 is unnecessary for modelling liquids[47]. Alternative thermostats are available which can model systems with higher Schmidt numbers[36, 62].

9.4 Derivation of Equilibrium

The derivation of the DPD algorithm is based on the Fokker-Planck equation

$$\frac{\partial \rho}{\partial t} = \mathcal{L}\rho \tag{9.17}$$

where ρ is the equilibrium distribution function and $\mathcal{L}$ is the evolution operator, which may be split into *conservative* and *dissipative* parts:

$$\mathcal{L} = \mathcal{L}^{\mathcal{C}} + \mathcal{L}^{\mathcal{D}} \tag{9.18}$$

with

$$\mathcal{L}^{\mathcal{C}} = -\sum_{i=1}^{N} \frac{\vec{p}_i}{m_i} \frac{\partial}{\partial \vec{r}_i} - \sum_{i \neq j}^{N} \vec{F}_{ij}^{C} \frac{\partial}{\partial \vec{p}_i} \tag{9.19}$$

$$\mathcal{L}^{\mathcal{D}} = \sum_{i=1}^{N} \hat{e}_{ij} \cdot \frac{\partial}{\partial \vec{p}_i} \left[\gamma w^{D} \left(\hat{e}_{ij} \cdot \vec{v}_{ij} \right) + \frac{\sigma^2}{2} \left\{ w^{R} \left(r_{ij} \right) \right\}^2 \hat{e}_{ij} \cdot \left\{ \frac{\partial}{\partial \vec{p}_i} - \frac{\partial}{\partial \vec{p}_j} \right\} \right] \tag{9.20}$$

where $\hat{e}_{ij} = \frac{\vec{r_{ij}}}{r_{ij}}$.

When $\alpha = \gamma = 0$ then Equation (9.17) becomes

$$\frac{\partial \rho}{\partial t} = \mathcal{L}^{\mathcal{C}} \rho \tag{9.21}$$

for which the equilibrium solution is evidently

$$\rho^{eq} = \frac{1}{Z} \exp \left(\frac{1}{k_B T} \left[\sum_{i=1}^{N} \frac{p_i^2}{2m_i} + \frac{1}{2} \sum_{j \neq i}^{N} \phi \left(r_{ij} \right) \right] \right) \tag{9.22}$$

which is, of course, the Boltzmann distribution function for an equilibrium system. Thus it is apparent that for the simulation based on Equation (9.17) to maintain the same distribution function, the terms in the operator $\mathcal{L}^{\mathcal{D}}$ of Equation (9.20) must sum to zero. It follows that the conditions given in Equations (9.8) and (9.9) must apply.

9.5 Summary of Dissipative Particle Dynamics

DPD is a simple method. All that is required is a system of spherical particles enclosed in a periodic box undergoing time evolution as a result of the above forces. In implementation it differs very little from Molecular Dynamics. It should be noted that all computed interactions are pairwise, which means that the principle of the conservation of momentum in the system, or 'Galilean invariance', is preserved. The conservation of momentum is required for the preservation of hydrodynamic forces.

Chapter 10

DL_MESO_DPD Basic Definition

10.1 Data structure

10.1.1 Storage of running information

DL_MESO_DPD contains storage for information on the system being modelled, the domain and neighbour information for parallel running. The parameters for these aspects of calculations can be found in Tables 10.1, 10.2 and 10.3.

It should be noted for the parameters in Table 10.1 that the fundamental units for the simulation are those of mass $[M]$, length $[L]$ and energy $[E]$: the DPD unit of time is equivalent to $[L]\sqrt{\frac{[M]}{[E]}}$ while temperature (in the form $k_B T$) is defined as two-thirds of the kinetic energy per particle.

10.1.2 Storage of particle and bond properties

The total number of particles in a system is `nsyst`, of which `nusyst` particles are 'loose', i.e. not bonded to other particles, and `nfsyst` are 'frozen', i.e. remain fixed in position but still interact with other particles. DL_MESO_DPD divides up the particles and total system volume (`volm`) between the processing units available. At any given time each process holds `nbeads` particles, including `nfbeads` frozen particles. Each process also has `nbonds` bonds, `nangles` bond angles and `ndiheds` bond dihedrals to deal with. If bonds are dealt with locally, only the bonds associated with the subdomain are calculated by each process; otherwise all processes hold information on all bonds.

The Cartesian coordinates, velocities and forces for the particles are each held in sets of three double precision arrays for x-, y- and z-components. Particle positions relative to the volume modelled by the individual processor — thus *not* absolute positions unless the serial version of DL_MESO_DPD is used — are held in arrays `xxx(i)`, `yyy(i)` and `zzz(i)` (for particle `i`). Particle velocities are held in `vxx(i)`, `vyy(i)` and `vzz(i)`. Three sets of arrays for the net forces acting on the particles are available: `fxx(i)`, `fyy(i)` and `fzz(i)` for forces that remain constant over each time step, `fvx(i)`, `fvy(i)` and `fvz(i)` for forces that may vary during the time step (e.g. drag forces for DPD Velocity Verlet integration, thermostatting force for Stoyanov-Groot thermostat), and `fcfx(i)`, `fcfy(i)` and `fcfz(i)` for corrections to forces between frozen particles (particularly long-range electrostatic forces).

The particles modelled by a particular processor have both local and global identity numbers, the latter of which are stored in the integer array `lab(i)`. DL_MESO_DPD assigns the lowest local identity numbers (i.e. between 1 and `nfbeads`) to the frozen particles in each processor's subdomain to avoid having to search for and skip over frozen particles during force integration steps, while the highest global identity numbers (from `nusyst+1` to `nsyst`) are assigned to particles belonging to molecules. When particles are copied into boundary halos, the processor numbers and local particle numbers in their original processors are stored in the integer arrays

Table 10.1: System information

parameter	**meaning**
`text`	name of calculation
`nsyst`	total number of particles
`nusyst`	total number of unbonded particles
`nfsyst`	total number of frozen particles
`nspe`	number of particle species
`nmoldef`	number of molecule types
`temp`	specified system temperature ($k_B T$)
`prszero`	specified system pressure (P_0)
`rcut`	interaction cutoff radius (r_c)
`rmbcut`	many-body interaction cutoff radius (r_d)
`relec`	short-range electrostatic interaction cutoff radius (r_e)
`srfzcut`	surface repulsion cutoff distance (z_c)
`rhalo`	size of boundary halo
`nrun`	number of calculation timesteps
`nseql`	number of equilibration timesteps
`tstep`	duration of calculation timestep (Δt)
`timjob`	maximum time available to run calculation
`tclose`	time required to shut down calculation
`kres`	calculation restart parameter
`nsbpo`	interval for printing data to `OUTPUT` file
`ltraj`	switch for saving trajectory data to `HISTORY` file(s)
`straj`	starting timestep saving trajectory data to `HISTORY` file(s)
`ntraj`	interval for saving trajectory data to `HISTORY` file(s)
`nstk`	size of statistical data stack
`ltemp`	switch for temperature scaling before equilibration
`nsbts`	interval for temperature scaling
`lcorr`	switch for saving statistical data to `CORREL` file
`iscorr`	interval for saving statistical data to `CORREL` file
`itype`	integrator/thermostat selection
`btype`	barostat selection
`etype`	electrostatic algorithm selection
`srftype`	surface boundary selection
`lbond`	switch for modelling bonds between particles
`langle`	switch for modelling bond angles
`ldihed`	switch for modelling bond dihedrals
`lgbnd`	switch for globally storing all bond data
`lisoprs`	switch for isotropic variation of system dimensions with pressure

`lmp(i)` and `loc(i)` respectively. Numbers representing species and molecule types are stored in arrays `ltp(i)` and `ltm(i)`, which are used to assign particle masses and the names of particles and molecules respectively to the arrays `weight(i)`, `atmnam(i)` and `molnam(i)`.

All of these arrays are allocatable and their sizes set equal to `maxdim`, which is an estimate of the maximum possible number of particles likely to be stored in each processor based on the total number of particles, the numbers of available link cells and the number of processors available. Since the calculation for this parameter makes the assumption that the particle density is constant throughout the system, possible variations in density can be specified by the user and taken into account when calculating `maxdim`. A similar parameter, `maxpair`, is also calculated to determine the maximum possible number of particle pair interactions and used to define the maximum sizes of arrays for storing information for thermostats that correct particle velocities after force integration, i.e. Lowe-Andersen, Peters and Stoyanov-Groot.

Bonded particles are listed by global identity numbers in the integer array `bndtbl(i,j)` for bond `i`, with `j=1` representing the first particle in the pair, `j=2` for the second and `j=3` giving the user-defined bond type. The location of the first particle in each bond pair determines the processing unit which holds this data; thus movement of this reference particle across processes also causes the bond list entry to be transferred with it.

Table 10.2: Domain information

parameter	**meaning**
`idnode`	name of the processor
`nodes`	number of processors
`idx`	x-coordinate of the processor
`idy`	y-coordinate of the processor
`idz`	z-coordinate of the processor
`npx`	number of processors along x-axis
`npy`	number of processors along y-axis
`npz`	number of processors along z-axis
`volm`	volume of system
`dimx`	size of system in x-dimension
`dimy`	size of system in y-dimension
`dimz`	size of system in z-dimension
`sidex`	size of domain in x-dimension
`sidey`	size of domain in y-dimension
`sidez`	size of domain in z-dimension
`delx`	absolute x-coordinate of domain origin
`dely`	absolute y-coordinate of domain origin
`delz`	absolute z-coordinate of domain origin
`nlx`	number of link cells in domain along x-axis
`nly`	number of link cells in domain along y-axis
`nlz`	number of link cells in domain along z-axis
`wdthx`	link cell size in x-dimension
`wdthy`	link cell size in y-dimension
`wdthz`	link cell size in z-dimension
`nlewx`	number of electrostatic link cells in domain along x-axis
`nlewy`	number of electrostatic link cells in domain along y-axis
`nlewz`	number of electrostatic link cells in domain along z-axis
`wdthewx`	electrostatic link cell size in x-dimension
`wdthewy`	electrostatic link cell size in y-dimension
`wdthewz`	electrostatic link cell size in z-dimension

Table 10.3: Neighbour information

parameter	**meaning**
`map(k)`	processor name of neighbour `k`
`k` = 1	left neighbour
`k` = 2	right neighbour
`k` = 3	lower neighbour
`k` = 4	upper neighbour
`k` = 5	back neighbour
`k` = 6	front neighbour

Bond angles and bond dihedrals are stored in similar tables, `angtbl(i,j)` (particles `j` from 1 to 3, angle type at `j=4`) and `dhdtbl(i,j)` (particles `j` from 1 to 4, dihedral type at `j=5`) respectively, using the second particle in each triple or quadruple as the reference particle.

Prior to force calculations, a list of bonded particles in each process domain and – if calculating bond forces locally – the surrounding boundary halo is constructed, `lblclst(i,j)`, to allow DL_MESO_DPD to find the local number for a particle (`j=2`) from its global number (`j=1`) using a binary search. This list may include duplicates for the same global particle number; the local numbers giving the shortest distance between pairs of particles are selected and used.

10.2 The Parameters and Their Functions

Table 10.4 lists all the globally used parameters defined in DL_MESO_DPD, as given in the `constants`, `variables` and `numeric_container` modules. Because DL_MESO is an ongoing project and new parameters might be added to the package in the future, it is strongly recommended that users of DL_MESO check the names of any self-defined variables whenever the package is updated to reduce the possibility of duplications causing unexpected errors.

The notation column in Table 10.4 gives the restrictions applicable on the parameters. 'A' indicates an array of data, followed by the number of elements in the array. For example, 'A `maxdim`' means the parameter is actually an array with `maxdim` elements (numbered from 1 to `maxdim`). '≥ 1' means the number must be greater or equal to one, while for a Boolean parameter 'T or F' means its value can either be `.true.` or `.false.`. An asterisk in the data type for the array indicates that it is allocatable and defined during the run.

Table 10.4: DL_MESO_DPD Parameters

function	parameter	data type	notation
kind parameter for double precision numbers	dp	integer	
kind parameter for long integers	li	integer	
maximum word length	mxword	integer	
I/O channel for reading input files	nread	integer	
I/O channel for writing OUTPUT file	nwrite	integer	
I/O channel for reading export* files	nrtin	integer	
I/O channel for writing export* files	nrtout	integer	
I/O channel for writing CORREL file	nsave	integer	
I/O channel for writing HISTORY* files	nhist	integer	
number of bytes per real number	lword1	integer	
number of bytes per double precision number	lword2	integer	
value of π	pi	real(KIND=dp)	
value of $\sqrt{\pi}$	rtpi	real(KIND=dp)	
conversion factor from degrees to radians	degrad	real(KIND=dp)	
conversion factor from radians to degrees	raddeg	real(KIND=dp)	
conversion factor from energy to temperature	fkt	real(KIND=dp)	
square root of 12 (for random force calculations)	rt12	real(KIND=dp)	
convergence error for Langevin barostat	langeps	real(KIND=dp)	
name of processing unit	idnode	integer	
number of processing units	nodes	integer	
filename for restart files	exportname	character(10)	
switch for temperature scaling	ltemp	logical	T or F
switch for reading CONFIG file	lconfig	logical	T or F
switch for writing CORREL file	lcorr	logical	T or F
switch for writing HISTORY* files	ltraj	logical	T or F
switch for modelling bonds	lbond	logical	T or F
switch for modelling bond angles	langle	logical	T or F
switch for modelling bond dihedrals	ldihed	logical	T or F
switch for global holding of bond information	lgbnd	logical	T or F
switch for defining variable force arrays	lvarfc	logical	T or F
switch for isotropic variations of volume with pressure	lisoprs	logical	T or F
switch for ignoring global bead numbers in CONFIG file	ligindex	logical	T or F
duplications of CONFIG file in x direction	nfoldx	integer	
duplications of CONFIG file in y direction	nfoldy	integer	
duplications of CONFIG file in z direction	nfoldz	integer	
total number of duplications of CONFIG file	nfold	integer	
data key for CONFIG file	levcfg	integer	
periodic boundary key for CONFIG file	imcon	integer	
printout selection for OUTPUT file	outsel	integer	
maximum number of particles	maxdim	integer	
maximum number of pairwise interactions	maxpair	integer	
maximum message buffer size	maxbuf	integer	
maximum number of particles per molecule	mxmolsize	integer	
maximum number of bonds per molecule	mxbond	integer	
maximum number of angles per molecule	mxangles	integer	
maximum number of dihedrals per molecule	mxdiheds	integer	
maximum number of interaction parameters	mxprm	integer	
density variation for non-uniform system distributions	dvar	real(KIND=dp)	
name of DL_MESO_DPD calculation	text	character(80)	
number of time steps for calculation	nrun	integer	
interval for writing OUTPUT file	nsbpo	integer	≥ 1
interval for writing CORREL file	iscorr	integer	
starting time step for writing HISTORY* files	straj	integer	
interval for writing HISTORY* files	ntraj	integer	
restart file (export*) creation interval	ndump	integer	
temperature scaling interval	nsbts	integer	
number of equilibration time steps	nseql	integer	
calculation restart parameter	kres	integer	
number of species	nspe	integer	≥ 1
number of potentials	npot	integer	≥ 1
number of defined molecule types	nmoldef	integer	
number of defined bond types	nbonddef	integer	$\leq$ mxbonddef
number of defined bond angle types	nangdef	integer	$\leq$ mxbonddef
number of defined bond dihedral types	ndhddef	integer	$\leq$ mxbonddef
size of statistical data stack	nstk	integer	≥ 1
total number of particles in system	nsyst	integer	
total number of unbonded particles in system	nusyst	integer	
total number of frozen particles in system	nfsyst	integer	
total number of particles per unit cell	nsystcell	integer	
total number of unbonded particles per unit cell	nusystcell	integer	
total number of frozen particles per unit cell	nfsystcell	integer	
total number of molecules per unit cell	nummol	integer	
total number of bonds per unit cell	numbond	integer	
total number of bond angles per unit cell	numang	integer	
total number of bond dihedrals per unit cell	numdhd	integer	
time step number	nstep	integer	
force calculation time accumulator	timfrc	real(KIND=dp)	
step time accumulator	timstp	real(KIND=dp)	
specified system temperature ($k_B T$)	temp	real(KIND=dp)	
size of time step (Δt)	tstep	real(KIND=dp)	
halo boundary size	rhalo	real(KIND=dp)	
interaction cutoff radius (r_c)	rcut	real(KIND=dp)	> 0
square of interaction cutoff radius	rct2	real(KIND=dp)	> 0

Table 10.4: DL_MESO_DPD Parameters (continued)

function	parameter	data type	notation
many-body DPD cutoff radius (r_d)	`rmbcut`	real(KIND=dp)	
square of many-body DPD cutoff radius	`rmbct2`	real(KIND=dp)	
short-range electrostatic cutoff (r_e)	`relec`	real(KIND=dp)	
square of electrostatic cutoff	`rel2`	real(KIND=dp)	
surface cutoff length (z_c)	`srfzcut`	real(KIND=dp)	
square of surface cutoff length	`srfzct2`	real(KIND=dp)	
maximum calculation time	`timjob`	real(KIND=dp)	
calculation close time	`tclose`	real(KIND=dp)	
system volume	`volm`	real(KIND=dp)	
size of system in x-dimension	`dimx`	real(KIND=dp)	
size of system in y-dimension	`dimy`	real(KIND=dp)	
size of system in z-dimension	`dimz`	real(KIND=dp)	
size of unit cell in x-dimension	`dimxcell`	real(KIND=dp)	
size of unit cell in y-dimension	`dimycell`	real(KIND=dp)	
size of unit cell in z-dimension	`dimzcell`	real(KIND=dp)	
number of domain cells in x	`npx`	integer	≥ 1
number of domain cells in y	`npy`	integer	≥ 1
number of domain cells in z	`npz`	integer	≥ 1
list of neighbouring processes	`map`	integer	A 6
x position of domain cell	`idx`	integer	
y position of domain cell	`idy`	integer	
z position of domain cell	`idz`	integer	
position of domain cell origin in system volume (x dimension)	`delx`	real(KIND=dp)	
position of domain cell origin in system volume (y dimension)	`dely`	real(KIND=dp)	
position of domain cell origin in system volume (z dimension)	`delz`	real(KIND=dp)	
domain cell length in x-direction	`sidex`	real(KIND=dp)	
domain cell length in y-direction	`sidey`	real(KIND=dp)	
domain cell length in z-direction	`sidez`	real(KIND=dp)	
number of particles in domain cell	`nbeads`	integer	
number of frozen particles in domain cell	`nfbeads`	integer	
number of link cells in domain cell (x dimension)	`nlx`	integer	
number of link cells in domain cell (y dimension)	`nly`	integer	
number of link cells in domain cell (z dimension)	`nlz`	integer	
number of link cells in domain cell and boundary halo (x dimension)	`nlx2`	integer	
number of link cells in domain cell and boundary halo (y dimension)	`nly2`	integer	
number of link cells in domain cell and boundary halo (z dimension)	`nlz2`	integer	
link cell length in x-direction	`wdthx`	real(KIND=dp)	
link cell length in y-direction	`wdthy`	real(KIND=dp)	
link cell length in z-direction	`wdthz`	real(KIND=dp)	
number of link cells for electrostatics in domain cell (x dimension)	`nlewx`	integer	
number of link cells for electrostatics in domain cell (y dimension)	`nlewy`	integer	
number of link cells for electrostatics in domain cell (z dimension)	`nlewz`	integer	
number of link cells for electrostatics in domain cell and boundary halo (x dimension)	`nlewx2`	integer	
number of link cells for electrostatics in domain cell and boundary halo (y dimension)	`nlewy2`	integer	
number of link cells for electrostatics in domain cell and boundary halo (z dimension)	`nlewz2`	integer	
electrostatic link cell length in x-direction	`wdthewx`	real(KIND=dp)	
electrostatic link cell length in y-direction	`wdthewy`	real(KIND=dp)	
electrostatic link cell length in z-direction	`wdthewz`	real(KIND=dp)	
species name	`namspe`	character(LEN=8)*	A `nspe`
potential interaction type	`ktype`	integer*	A `npot`
species particle mass	`amass`	real(KIND=dp)*	A `nspe`
species particle charge	`chge`	real(KIND=dp)*	A `nspe`
species frozen status	`lfrzn`	integer*	A `nspe`
interaction parameter storage	`vvv`	real(KIND=dp)*	A `mxprm,npot`
Lennard-Jones long-range potential correction	`clr`	real(KIND=dp)	A 2
charged frozen particle correction to system stress tensor	`strcfz`	real(KIND=dp)	A 9
charged frozen particle correction to system virial	`vrlcfz`	real(KIND=dp)	A 3
charged frozen particle correction to system potential energy	`potcfz`	real(KIND=dp)	
integrator/thermostat type	`itype`	integer	
dissipative coefficient (γ)/collision frequency (Γ)	`gamma`	real(KIND=dp)*	A `npot`
random force parameter (σ)[1]/probability of velocity rescaling ($\Gamma\Delta t$)	`sigma`	real(KIND=dp)*	A `npot`
Stoyanov-Groot Nosé-Hoover coupling parameter (α)	`alphasg`	real(KIND=dp)	
thermostat pair list[2]: particle i	`pairlsti`	integer*	A `maxpair`
thermostat pair list: particle j	`pairlstj`	integer*	A `maxpair`
thermostat pair list: Maxwell distributed velocity (v_{ij}°)	`veleq`	real(KIND=dp)*	A `maxpair`
number of particle pairs in thermostat pair list	`npair`	integer	
barostat type	`btype`	integer	
barostat target pressure (P_0)	`prszero`	real(KIND=dp)	
barostat parameter a	`abaro`	real(KIND=dp)	
barostat parameter b	`bbaro`	real(KIND=dp)	
x component of piston velocity	`upx`	real(KIND=dp)	
y component of piston velocity	`upy`	real(KIND=dp)	
z component of piston velocity	`upz`	real(KIND=dp)	
x component of piston velocity at previous timestep	`upx1`	real(KIND=dp)	
y component of piston velocity at previous timestep	`upy1`	real(KIND=dp)	
z component of piston velocity at previous timestep	`upz1`	real(KIND=dp)	
piston mass (W_g)	`psmass`	real(KIND=dp)	
x component of piston force	`fpx`	real(KIND=dp)	
y component of piston force	`fpy`	real(KIND=dp)	
z component of piston force	`fpz`	real(KIND=dp)	

[1]This incorporates the time step for Velocity Verlet integration and is thus equal to $\sqrt{\frac{2\gamma k_B T}{\Delta t}}$.

[2]The pair list arrays are not allocated for the DPD thermostat (MD-VV or DPD-VV).

Table 10.4: DL_MESO_DPD Parameters (continued)

function	parameter	data type	notation
instantaneous virial	ivrl	real(KIND=dp)	A 3
Langevin random force parameter (σ_p)	sigmalang	real(KIND=dp)	
electrostatic algorithm type	etype	integer	
electrostatic coupling parameter (Γ)	gammaelec	real(KIND=dp)	
total system charge ($\sum_i q_i$)	qchg	real(KIND=dp)	
Ewald real-space convergence parameter (α)	alphaew	real(KIND=dp)	
reciprocal of real-space convergence parameter	ralphaew	real(KIND=dp)	
maximum reciprocal vector size in x dimension (k_1^{max})	kmax1	integer	≥ 1
maximum reciprocal vector size in y dimension (k_2^{max})	kmax2	integer	≥ 1
maximum reciprocal vector size in z dimension (k_3^{max})	kmax3	integer	≥ 1
Ewald self-interaction correction	engsic	real(KIND=dp)	
charged system correction	qfixv	real(KIND=dp)	
Slater charge smearing coefficient (β)	betaew	real(KIND=dp)	
bond interaction parameter a	aabond	real(KIND=dp)*	A nbonddef
bond interaction parameter b	bbbond	real(KIND=dp)*	A nbonddef
bond interaction parameter c	ccbond	real(KIND=dp)*	A nbonddef
bond interaction parameter d	ddbond	real(KIND=dp)*	A nbonddef
angle interaction parameter a	aaang	real(KIND=dp)*	A nbonddef
angle interaction parameter b	bbang	real(KIND=dp)*	A nbonddef
angle interaction parameter c	ccang	real(KIND=dp)*	A nbonddef
angle interaction parameter d	ddang	real(KIND=dp)*	A nbonddef
dihedral interaction parameter a	aadhd	real(KIND=dp)*	A nbonddef
dihedral interaction parameter b	bbdhd	real(KIND=dp)*	A nbonddef
dihedral interaction parameter c	ccdhd	real(KIND=dp)*	A nbonddef
dihedral interaction parameter d	dddhd	real(KIND=dp)*	A nbonddef
bond types	bdtype	integer*	A nbonddef
bond angle types	angtype	integer*	A nbonddef
bond dihedral types	dhdtype	integer*	A nbonddef
molecule isomer switch	moliso	logical*	A nbonddef
bond table	bndtbl	integer*	A numbond,3
bond angle table	angtbl	integer*	A numang,4
bond dihedral table	dhdtbl	integer*	A numdhd,5
global/local particle number list	lblclst	integer*	A maxdim,2
number of bonds in table	nbonds	integer	
number of bond angles in table	nangles	integer	
number of bond dihedrals in table	ndiheds	integer	
number of entries in global/local particle number list	nlist	integer	
species population of unbonded particles	nspec	integer*	A nspe
species population of bonded particles	nspecmol	integer	A mxspe
molecule type population	nmol	integer*	A nmoldef
bead numbers in molecule types	nbdmol	integer	A mxmoldef
molecule name	nammol	character(8)*	A 0:nmoldef
species number for molecule insertion	mlstrtspe	integer*	A nmoldef,mxmolsize
x coordinate for molecule insertion	mlstrtxxx	real(KIND=dp)*	A nmoldef,mxmolsize
y coordinate for molecule insertion	mlstrtyyy	real(KIND=dp)*	A nmoldef,mxmolsize
z coordinate for molecule insertion	mlstrtzzz	real(KIND=dp)*	A nmoldef,mxmolsize
cube size for molecule insertion	cbsize	real(KIND=dp)*	A nmoldef
number of bonds for molecule type	nbond	integer*	A nmoldef
number of bond angles for molecule type	nangle	integer*	A nmoldef
number of bond dihedrals for molecule type	ndihed	integer*	A nmoldef
bond table storage for molecule insertion	bdinp1	integer*	A nmoldef,mxbonds
bond table storage for molecule insertion	bdinp2	integer*	A nmoldef,mxbonds
bond table storage for molecule insertion	bdinp3	integer*	A nmoldef,mxbonds
angle table storage for molecule insertion	anginp1	integer*	A nmoldef,mxbonds
angle table storage for molecule insertion	anginp2	integer*	A nmoldef,mxbonds
angle table storage for molecule insertion	anginp3	integer*	A nmoldef,mxbonds
angle table storage for molecule insertion	anginp4	integer*	A nmoldef,mxbonds
dihedral table storage for molecule insertion	dhdinp1	integer*	A nmoldef,mxbonds
dihedral table storage for molecule insertion	dhdinp2	integer*	A nmoldef,mxbonds
dihedral table storage for molecule insertion	dhdinp3	integer*	A nmoldef,mxbonds
dihedral table storage for molecule insertion	dhdinp4	integer*	A nmoldef,mxbonds
dihedral table storage for molecule insertion	dhdinp5	integer*	A nmoldef,mxbonds
localized densities	rhomb	real(KIND=dp)*	A maxdim, nspe
surface type	srftype	integer	
surface switch for boundary normal to x-axis	srfx	integer	
surface switch for boundary normal to y-axis	srfy	integer	
surface switch for boundary normal to z-axis	srfz	integer	
switches for surfaces in current node	srflgc	logical	A 6
surface repulsion parameters A_{wall}	aasrf	real(KIND=dp)*	A nspe
species of frozen beads for surface	frzwspe	integer	
number of frozen beads in x dimension for wall normal to x-axis	npxfwx	integer	
number of frozen beads in y dimension for wall normal to x-axis	npxfwy	integer	
number of frozen beads in z dimension for wall normal to x-axis	npxfwz	integer	
number of frozen beads in x dimension for wall normal to y-axis	npyfwx	integer	
number of frozen beads in y dimension for wall normal to y-axis	npyfwy	integer	
number of frozen beads in z dimension for wall normal to y-axis	npyfwz	integer	
number of frozen beads in x dimension for wall normal to z-axis	npzfwx	integer	
number of frozen beads in y dimension for wall normal to z-axis	npzfwy	integer	
number of frozen beads in z dimension for wall normal to z-axis	npzfwz	integer	
frozen bead density of walls	frzwdens	real(KIND=dp)	
width of frozen bead wall normal to x-axis	frzwxwid	real(KIND=dp)	
width of frozen bead wall normal to y-axis	frzwywid	real(KIND=dp)	
width of frozen bead wall normal to z-axis	frzwzwid	real(KIND=dp)	
x component of external body acceleration per particle	bdfrcx	real(KIND=dp)	

Table 10.4: DL_MESO_DPD Parameters (continued)

function	**parameter**	**data type**	**notation**
y component of external body acceleration per particle	`bdfrcy`	real(KIND=dp)	
z component of external body acceleration per particle	`bdfrcz`	real(KIND=dp)	
x component of Lees-Edwards shearing velocity	`shrvx`	real(KIND=dp)	
y component of Lees-Edwards shearing velocity	`shrvy`	real(KIND=dp)	
z component of Lees-Edwards shearing velocity	`shrvz`	real(KIND=dp)	
x component of Lees-Edwards shearing displacement	`shrdx`	real(KIND=dp)	
y component of Lees-Edwards shearing displacement	`shrdy`	real(KIND=dp)	
z component of Lees-Edwards shearing displacement	`shrdz`	real(KIND=dp)	
force x-component on particle	`fxx`	real(KIND=dp)*	A `maxdim`
force y-component on particle	`fyy`	real(KIND=dp)*	A `maxdim`
force z-component on particle	`fzz`	real(KIND=dp)*	A `maxdim`
variable force x-component on particle	`fvx`[3]	real(KIND=dp)*	A `maxdim`
variable force y-component on particle	`fvy`	real(KIND=dp)*	A `maxdim`
variable force z-component on particle	`fvz`	real(KIND=dp)*	A `maxdim`
corrective force x-component on charged frozen particle	`fcfx`	real(KIND=dp)*	A `maxdim`
corrective force y-component on charged frozen particle	`fcfy`	real(KIND=dp)*	A `maxdim`
corrective force z-component on charged frozen particle	`fcfz`	real(KIND=dp)*	A `maxdim`
velocity x-component of particle	`vxx`	real(KIND=dp)*	A `maxdim`
velocity y-component of particle	`vyy`	real(KIND=dp)*	A `maxdim`
velocity z-component of particle	`vzz`	real(KIND=dp)*	A `maxdim`
Cartesian coordinate x for particle	`xxx`	real(KIND=dp)*	A `maxdim`
Cartesian coordinate y for particle	`yyy`	real(KIND=dp)*	A `maxdim`
Cartesian coordinate z for particle	`zzz`	real(KIND=dp)*	A `maxdim`
particle global identity number	`lab`	integer*	A `maxdim`
particle species number	`ltp`	integer*	A `maxdim`
particle molecule type number	`ltm`	integer*	A `maxdim`
particle link cell population number	`lct`	integer*	A `maxdim`
particle link cell number	`link`	integer*	A `maxdim`
particle local domain cell identity number	`loc`	integer*	A `maxdim`
particle domain cell number	`lmp`	integer*	A `maxdim`
particle molecule type number	`ltm`	integer*	A `maxdim`
species name for particle	`atmnam`	character(8)*	A `maxdim`
molecule name for particle	`molnam`	character(8)*	A `maxdim`
particle mass	`weight`	real(KIND=dp)*	A `maxdim`
potential energy accumulator	`pe`	real(KIND=dp)	
virial accumulator	`vir`	real(KIND=dp)	
stress tensor accumulator	`stress`	real(KIND=dp)	A 9
kinetic energy accumulator	`tke`	real(KIND=dp)	
bond potential energy accumulator	`be`	real(KIND=dp)	
angle potential energy accumulator	`ae`	real(KIND=dp)	
dihedral potential energy accumulator	`de`	real(KIND=dp)	
electrostatic potential energy accumulator	`ee`	real(KIND=dp)	
bond length accumulator	`bdlng`	real(KIND=dp)	
bond length maximum value	`bdlmax`	real(KIND=dp)	
bond length minimum value	`bdlmin`	real(KIND=dp)	
bond angle accumulator	`bdang`	real(KIND=dp)	
bond dihedral accumulator	`bddhd`	real(KIND=dp)	
average system potential energy	`avepe`	real(KIND=dp)	
average system virial	`avevir`	real(KIND=dp)	
average system kinetic energy	`avetke`	real(KIND=dp)	
average system total energy	`avete`	real(KIND=dp)	
average system pressure	`aveprs`	real(KIND=dp)	
average system volume	`avevlm`	real(KIND=dp)	
average system temperature	`avettp`	real(KIND=dp)	
average system bond potential energy	`avebe`	real(KIND=dp)	
average system angle potential energy	`aveae`	real(KIND=dp)	
average system dihedral potential energy	`avede`	real(KIND=dp)	
average system electrostatic potential energy	`aveee`	real(KIND=dp)	
system potential energy fluctuation	`flcpe`	real(KIND=dp)	
system virial fluctuation	`flcvir`	real(KIND=dp)	
system kinetic energy fluctuation	`flcke`	real(KIND=dp)	
system total energy fluctuation	`flcte`	real(KIND=dp)	
system pressure fluctuation	`flcprs`	real(KIND=dp)	
system volume fluctuation	`flcvlm`	real(KIND=dp)	
system temperature fluctuation	`flcttp`	real(KIND=dp)	
system bond potential energy fluctuation	`flcbe`	real(KIND=dp)	
system angle potential energy fluctuation	`flcae`	real(KIND=dp)	
system dihedral potential energy fluctuation	`flcde`	real(KIND=dp)	
system electrostatic potential energy fluctuation	`flcee`	real(KIND=dp)	
system potential energy accumulator	`zumpe`	real(KIND=dp)	
system virial accumulator	`zumvir`	real(KIND=dp)	
system kinetic energy accumulator	`zumtke`	real(KIND=dp)	
system volume accumulator	`zumvlm`	real(KIND=dp)	
system bond potential energy accumulator	`zumbe`	real(KIND=dp)	
system angle potential energy accumulator	`zumae`	real(KIND=dp)	
system dihedral potential energy accumulator	`zumde`	real(KIND=dp)	
system electrostatic potential energy accumulator	`zumee`	real(KIND=dp)	
system potential energy at current step	`stppe`	real(KIND=dp)	
system virial at current step	`stpvir`	real(KIND=dp)	
system kinetic energy at current step	`stptke`	real(KIND=dp)	
system total energy at current step	`stptke`	real(KIND=dp)	

[3] These are only allocated if variable forces are required for e.g. DPD Velocity Verlet integration and Stoyanov-Groot thermostat.

Table 10.4: DL_MESO_DPD Parameters (continued)

function	**parameter**	**data type**	**notation**
system pressure at current step	`stpprs`	real(KIND=dp)	
system volume at current step	`stpvlm`	real(KIND=dp)	
system temperature at current step	`stpttp`	real(KIND=dp)	
system bond potential energy at current step	`stpbe`	real(KIND=dp)	
system angle potential energy at current step	`stpae`	real(KIND=dp)	
system dihedral energy at current step	`stpde`	real(KIND=dp)	
system electrostatic energy at current step	`stpee`	real(KIND=dp)	
system mean bond length at current step	`stpbdl`	real(KIND=dp)	
system maximum bond length at current step	`stpbdmx`	real(KIND=dp)	
system minimum bond length at current step	`stpbdmn`	real(KIND=dp)	
system mean bond angle at current step	`stpang`	real(KIND=dp)	
system mean bond dihedral at current step	`stpdhd`	real(KIND=dp)	
rolling average system potential energy	`ravpe`	real(KIND=dp)	
rolling average system virial	`ravvir`	real(KIND=dp)	
rolling average system kinetic energy	`ravtke`	real(KIND=dp)	
rolling average system total energy	`ravte`	real(KIND=dp)	
rolling average system pressure	`ravprs`	real(KIND=dp)	
rolling average system volume	`ravvlm`	real(KIND=dp)	
rolling average system temperature	`ravttp`	real(KIND=dp)	
rolling average system bond potential energy	`ravbe`	real(KIND=dp)	
rolling average system angle potential energy	`ravae`	real(KIND=dp)	
rolling average system dihedral potential energy	`ravde`	real(KIND=dp)	
rolling average system electrostatic potential energy	`ravee`	real(KIND=dp)	
stage number for data stack	`nav`	integer	
data stack for potential energy	`stkpe`	real(KIND=dp)*	A `nstk`
data stack for virial	`stkvir`	real(KIND=dp)*	A `nstk`
data stack for kinetic energy	`stktke`	real(KIND=dp)*	A `nstk`
data stack for volume	`stkvlm`	real(KIND=dp)*	A `nstk`
data stack for bond potential energy	`stkbe`	real(KIND=dp)*	A `nstk`
data stack for angle potential energy	`stkae`	real(KIND=dp)*	A `nstk`
data stack for dihedral potential energy	`stkde`	real(KIND=dp)*	A `nstk`
data stack for electrostatic potential energy	`stkee`	real(KIND=dp)*	A `nstk`
`duni` random number generator state	`uni`	integer	A 102
`mtrnd` random number generator state	`mt`	integer	A 0:624

Chapter 11

DL_MESO_DPD Features

11.1 Domain decomposition and linked-list cell calculations

The Domain Decomposition (DD) strategy is one of several ways to parallelize particle-based simulations[57]. Its basis is the division of the simulated system into equal-sized spatial blocks or domains, each of which is allocated to a specific processing unit of a parallel computer. The arrays defining the coordinates, velocities and forces for all N particles in the system are divided into sub-arrays of size $\approx \frac{N}{P}$ on each of the P processing units, with the particles allocated geometrically among them. In order for the strategy to work efficiently, the simulated system should possess a reasonably uniform density so that each processing unit is allocated as equal a portion of particle data as possible. The computation of forces and integration of the equations of motion are shared (more or less) equally between the processing units and to a large extent can be computed independently on each unit. While tricky to program, this method is conceptually simple and particularly suited to large-scale simulations.

The DD strategy which underpins DL_MESO_DPD is based on the link cell algorithm[26], which requires a relatively short-ranged cutoff for interparticle potentials and forces. There is a need for processing units to exchange 'halo data', i.e. sending the contents of link cells at the boundaries of each domain to neighbouring units so each may have all the necessary information to compute pairwise forces acting on the particles in its allotted domain. Similarly the force and virial contributions from particles in boundary halos need to be returned to their original processing units for summation. The link cell algorithm is also applied in serial by duplicating system data to create the boundary halo across periodic boundaries.

The size of the boundary halo – which can be specified by the user in the `CONTROL` file – should not be greater than the minimum system dimension per domain; for good parallel performance, it is recommended that the halo size should be no larger than one-third of the smallest subdomain dimension. The value of `maxdim` calculated after reading the input files (in `config_module`) gives the maximum sizes of force, velocity and position arrays. This value should be large enough to hold all particles in each domain *plus* any particles in boundary halos, including duplicates when running in serial or using smaller numbers of processing units. If the density of the system is likely to be uneven, the user can increase the size of `maxdim` by specifying an additional density variation in the `CONTROL` file.

11.1.1 Intramolecular interactions

Intramolecular interactions may be handled in two different ways: either (1) locally with each processing unit being allocated a subset of bonds to deal with (including bonds across neighbouring units), or (2) globally with all units holding all bond data and sharing bonded particle positions, each carrying out all bond calculations and appropriately allocating forces to local particles. The former method may require larger boundary halo sizes for the bond lengths being simulated but is more efficient for larger numbers of molecules and processing units, while the latter method requires the sharing of information between all units but does not require halo information and is guaranteed to find all bonds.

Bookkeeping arrays (`bndtbl`, `angtbl` and `dhdtbl`) list all particles involved in bonded interactions according to global index numbers and point to appropriate arrays of parameters to define the potential. If the 'key' bonded particle for a bond[1] moves from one processing unit to another, the entry in the bookkeeping array is also moved. At each time step a list of bonded particles in each domain (`lblclst`) is created to relate global index numbers to the local index numbers used by the processing unit in force, velocity and coordinate arrays. This global/local index list is sorted by global index number to allow cross-referencing to local index numbers by means of a binary search.

11.1.2 Electrostatic interactions

For systems with periodic boundary conditions DL_MESO_DPD uses the Ewald sum to calculate Coulombic interactions (see Section 11.5). Calculation of the real space component (in routine `ewald_real_slater`) uses the same link cell algorithm as for other pairwise interactions, albeit using a larger cutoff radius (r_e) and requires a larger boundary halo than for standard pairwise interactions.

11.2 Thermostats and integration algorithms

The integration algorithms in DL_MESO_DPD are based on the second-order Velocity Verlet (VV) scheme[67], which yields the positions, velocities and forces of particles at the same time and is generally used in molecular dynamics simulations. This algorithm has two stages. The first stage advances the particle velocities to time $t + \frac{1}{2}\Delta t$ by integrating the forces and uses the new half-step velocities to advance the position to time $t + \Delta t$:

$$\vec{v}_i\left(t + \tfrac{1}{2}\Delta t\right) = \vec{v}_i\left(t\right) + \frac{\Delta t}{2}\frac{\vec{F}_i\left(t\right)}{m_i} \tag{11.1}$$

$$\vec{r}_i\left(t + \Delta t\right) = \vec{r}_i\left(t\right) + \Delta t \vec{v}_i\left(t + \tfrac{1}{2}\Delta t\right) \tag{11.2}$$

The positions at the end of the time step allow the forces to be recalculated, before the second stage of the algorithm is applied to advance the half-step velocities to the end of the time step by integrating with the new force:

$$\vec{v}_i\left(t + \Delta t\right) = \vec{v}_i\left(t + \tfrac{1}{2}\Delta t\right) + \frac{\Delta t}{2}\frac{\vec{F}_i\left(t + \Delta t\right)}{m_i} \tag{11.3}$$

Five thermostatting algorithms are currently available in DL_MESO_DPD: two variants of the standard DPD thermostat and three alternative schemes which apply velocity corrections to the particles after force integration. The algorithm can be selected in the `CONTROL` file using the directive **ensemble** with the keyword **nvt** for constant volume simulations or **npt** for constant pressure simulations. Dissipative force parameters and collision frequencies can be specified for each interacting species pair in the `FIELD` file. Frozen particles are involved in thermostatting algorithms due to the contributions they make to system virials and pressure; however they are excluded from the force integration algorithm and their velocities are reset to their previous values (usually zero).

11.2.1 DPD thermostat with standard Velocity Verlet integration (MD-VV) (`mdvv`)

This algorithm uses the drag (dissipative) and random forces, $\vec{F}_{ij}^D$ and $\vec{F}_{ij}^R$ respectively as described in Chapter 9, as the system thermostat, i.e. the thermostatting force $\vec{F}_{ij}^T = \vec{F}_{ij}^D + \vec{F}_{ij}^R$. This thermostatting force is combined with all other forces between particles – pairwise conservative (standard and/or density-dependent), bonding, electrostatic, planar surface, external (body) forces – and integrated using the standard Velocity Verlet integrator.

[1]This is the first referenced particle in stretching bonds and the second for bond angles and dihedrals.

The combination of the DPD thermostat with the standard MD-type VV algorithm is the simplest and least time-consuming thermostatting algorithm available in DL_MESO_DPD. (If no ensemble type is selected in the `CONTROL` file, DL_MESO_DPD will use this algorithm by default.) The drag force does, however, depend upon particle velocities and is therefore only approximated using the mid-step values: this frequently produces a system temperature higher than that specified by the user and requires a small time step Δt to reduce the offset to tolerable levels.

11.2.2 DPD thermostat with DPD Velocity Verlet integration (DPD-VV) (`dpdvv`)

As with the MD-VV scheme, this algorithm uses the drag and random forces as the system thermostat, which are combined with all other forces before being integrated using the Velocity Verlet scheme. The drag force is subsequently recalculated after the second stage using the velocities at the end of the time step[13].

The recalculation of drag forces after force integration helps to alleviate the temperature offset produced by the MD-VV, and hence larger time steps may be used for reasonable temperature control. It does require the re-use of the linked-list cells and inter-processor communications to recalculate the drag forces, which can significantly increase the time required per time step compared to the MD-VV scheme.

11.2.3 Lowe-Andersen thermostat (`lowe`)

The Lowe-Andersen thermostat[36] is an alternative to the use of drag and random forces in the DPD thermostat, which uses a variant of the Andersen thermostat[1]. After all other forces (conservative, bonding etc.) are integrated using the Velocity Verlet scheme, a random sample of particle pairs have their relative velocity replaced by a value from a Maxwellian distribution, i.e.

$$v_{ij}^{\circ} = \zeta_{ij}\sqrt{\frac{k_B T}{\mu_{ij}}} \tag{11.4}$$

where $\mu_{ij} = \frac{m_i m_j}{m_i + m_j}$ is the reduced mass between the two particles. The velocities of particles i and j thus become:

$$\vec{v}_i = \vec{v}_i - \frac{\mu_{ij}}{m_i}\left(-\left(\hat{e}_{ij}\cdot\vec{v}_{ij}\right) + v_{ij}^{\circ}\right)\hat{e}_{ij} \tag{11.5}$$

$$\vec{v}_j = \vec{v}_j + \frac{\mu_{ij}}{m_i}\left(-\left(\hat{e}_{ij}\cdot\vec{v}_{ij}\right) + v_{ij}^{\circ}\right)\hat{e}_{ij} \tag{11.6}$$

The probability of a particle pair being thermostatted is equal to $\Gamma\Delta t$, where Γ is defined as the collision frequency (with a maximum effective value of $\frac{1}{\Delta t}$), and the velocity corrections to particle pairs are applied in a random order to prevent biasing.

The above pairwise correction of velocities is equivalent to applying a thermostatting force equal to

$$\vec{F}_{ij}^{T} = \frac{\mu_{ij}}{\Delta t}\left(-\left(\hat{e}_{ij}\cdot\vec{v}_{ij}\right) + v_{ij}^{\circ}\right)\hat{e}_{ij} \tag{11.7}$$

and thus a virial correction of $-\vec{F}_{ij}^{T}\cdot\vec{r}_{ij}$ is applied for each particle pair being thermostatted.

The viscosity and self-diffusion generated by this thermostat for a single species are

$$\nu = \frac{\pi\rho\Gamma r_c^5}{75m} \tag{11.8}$$

$$D = \frac{k_B T \tau_D}{m} \tag{11.9}$$

where τ_D is the decay time for velocity correlations and inversely proportional to the collision frequency. The Schmidt number is therefore proportional to $\frac{\Gamma^2}{k_B T}$ and can thus reach values up to $\mathcal{O}(10^7)$.

This thermostat is suited to systems with higher viscosities and low diffusitivies while giving the correct system temperature for a wide range of time step sizes (within numerical errors due to Velocity Verlet force integration).

Its implementation in parallel running uses a replicated data strategy to carry out the velocity corrections: this requires additional memory on each processing unit for the velocities of all particles and the data required to modify the velocities of particle pairs. The efficiency of the Lowe-Andersen thermostat thus decreases with increasing numbers of particles in the entire system and Γ.

11.2.4 Peters thermostat (`peters`)

The Peters thermostat[47] is a modification of the Lowe-Andersen thermostat that reduces to standard DPD as the time step tends to zero. After integrating all forces using the Velocity Verlet scheme, all particle pairs have their velocities modified (in a random order) using:

$$\vec{v}_i = \vec{v}_i - \frac{1}{m_i}\left(-a_{ij}\left(\hat{e}_{ij}\cdot\vec{v}_{ij}\right)\Delta t + b_{ij}\zeta_{ij}\sqrt{\Delta t}\right)\hat{e}_{ij} \tag{11.10}$$

$$\vec{v}_j = \vec{v}_j + \frac{1}{m_i}\left(-a_{ij}\left(\hat{e}_{ij}\cdot\vec{v}_{ij}\right)\Delta t + b_{ij}\zeta_{ij}\sqrt{\Delta t}\right)\hat{e}_{ij} \tag{11.11}$$

where the coefficients a_{ij} and b_{ij} are chosen so that

$$b_{ij} = \sqrt{2k_B T a_{ij}\left(1 - \frac{a_{ij}\Delta t}{2\mu_{ij}}\right)}.$$

To ensure that the thermostat both reduces to the DPD thermostat as the time step reduces to zero and is not restricted by the choice of time step, the coefficients are chosen as follows:

$$a_{ij} = \frac{\mu_{ij}}{\Delta t}\left(1 - \exp\left[-\frac{\gamma_{ij}\omega(r_{ij})\Delta t}{\mu_{ij}}\right]\right) \tag{11.12}$$

$$b_{ij} = \sqrt{\frac{k_B T\mu_{ij}}{\Delta t}\left(1 - \exp\left[-\frac{2\gamma_{ij}\omega(r_{ij})\Delta t}{\mu_{ij}}\right]\right)} \tag{11.13}$$

The above velocity corrections give an equivalent thermostatting force of

$$\vec{F}_{ij}^T = \left(-a_{ij}\left(\hat{e}_{ij}\cdot\vec{v}_{ij}\right) + \frac{b_{ij}\zeta_{ij}}{\sqrt{\Delta t}}\right)\hat{e}_{ij} \tag{11.14}$$

and a correction to the virial of $-\vec{F}_{ij}^T\cdot\vec{r}_{ij}$ is also applied for each particle pair.

This thermostat can be used with larger time steps than the standard DPD thermostat but with similarly low system viscosities. As for the Lowe-Andersen thermostat, its implementation in parallel running uses a replicated data strategy to carry out the velocity corrections, which requires additional memory per processing unit for storing the velocities of all particles in the system and the data required to modify them. The efficiency of the Peters thermostat therefore depends upon the total number of particles in the system: since all particle pairs are modified, calculation times for this thermostat may be comparable to those for the Lowe-Andersen thermostat when $\Gamma\Delta t \approx 1$.

11.2.5 Stoyanov-Groot thermostat (`stoyanov`)

The Stoyanov-Groot thermostat[62] is a combination of the Lowe-Andersen thermostat and a Galilean-invariant Nosé-Hoover thermostat which acts locally and on pairs of particles. During force calculations after the first Velocity Verlet stage, the choice to use either the Lowe-Andersen or Nosé-Hoover thermostats for each particle pair is made at random; the Lowe-Andersen thermostat is selected with a probability of $\Gamma\Delta t$. The system temperature is also determined in terms of relative velocities for all particle pairs, i.e.

$$k_B T^* = \frac{\sum_{i>j}\psi^T(r_{ij})\mu_{ij}\vec{v}_ij^2}{3\sum_{i>j}\psi^T(r_{ij})} \tag{11.15}$$

where $\psi^T(r_{ij})$ is a smearing function for the temperature, chosen to reduce to zero when $r_{ij} > r_c$: by default this is set as $\psi^T(r_{ij}) = 1$ for $r < r_c$. For all particle pairs that are to be subjected to the Nosé-Hoover thermostat, an additional thermostatting force is included:

$$\vec{F}_{ij}^T = -\alpha w^T(r_{ij}) \left(1 - \frac{k_B T^*}{k_B T}\right) [\vec{v}_{ij} \cdot \hat{e}_{ij}]\, \hat{e}_{ij} \tag{11.16}$$

with α as a system-wide thermostat coupling parameter and $w^T(r_{ij})$ as a switching function, which by default is equivalent to $w^R(r_{ij}) = 1 - \frac{r_{ij}}{r_c}$ for standard DPD. All other particle pairs are thermostatted using the standard Lowe-Andersen method. A virial correction of $-\vec{F}_{ij}^T \cdot \vec{r}_{ij}$ is also made for each particle pair.

This thermostat can produce a wide range of system viscosities and diffusivities with good temperature control and hydrodynamics, using the collision frequency Γ to obtain the required Schmidt number. The replicated data strategy is again used for the Lowe-Andersen part, which requires memory in each processing unit to store the velocities of all particles and the data for particle pair modification using the Lowe-Andersen scheme: the Nosé-Hoover scheme calculates the thermostatting forces locally.

11.3 Barostats

In addition to a thermostat, a barostat may be included in simulations to obtain a desired average pressure (P_0) by adjusting the size (and shape) of the simulation cell. DL_MESO_DPD includes two such algorithms: a Langevin-type barostat[29] and the Berendsen barostat[2], both of which have been coupled to all five available thermostats.

The isotropic pressure in a system is calculated using the virial theorem:

$$P(t) = \frac{1}{3V(t)} \left[\sum_i m_i v_i^2(t) + \sum_i \vec{F}_i(t) \cdot \vec{r}_i(t)\right] \tag{11.17}$$

while for anisotropic orthorhombic systems the pressure in dimension α, related to the instantaneous stress tensor component $\sigma_{\alpha\alpha}(t)$, is defined as

$$P_\alpha(t) = \frac{1}{V(t)} \left[\sum_i m_i v_{i,\alpha}^2(t) + \sum_i F_{i,\alpha}(t) r_{i,\alpha}(t)\right]. \tag{11.18}$$

In both equations, the instantaneous values required for barostats include only the interaction forces (e.g. soft pairwise interactions, bonds, electrostatics): they do not include virial contributions from thermostatting, which are included in reported values of system pressure.

All barostat definitions are expressed for the more general anisotropic case: these can be applied for isotropic systems by setting $P_x(t)$, $P_y(t)$ and $P_z(t)$ all equal to $P(t)$. The barostat can be selected in the `CONTROL` file using the directive **ensemble npt**: the barostat type should be specified after the coupled thermostat. The target system pressure can also be specified in the same file using the directive **pres**sure. By default the barostat is assumed to act isotropically, although the `CONTROL` file directive **no isotro**py can be used to apply anisotropy. Frozen particles are moved when a barostat is applied but their positions relative to the dimensions of the system remain constant during calculations.

11.3.1 Langevin barostat (`langevin`)

The governing equation for the Langevin barostat on an orthorhombic simulation cell[29] is the force exerted by the piston (expressed as the time-derivative of its momentum $p_{g,\alpha} = W_g u_{g,\alpha}$):

$$\dot{p}_{g,\alpha} = V(P_\alpha - P_0) + \frac{1}{N_f} \sum_i m_i v_i^2 - \gamma_p p_{g,\alpha} + \sigma_p \zeta_{p,\alpha} \Delta t^{-\frac{1}{2}} \tag{11.19}$$

where N_f is the number of degrees of freedom: for a three-dimensional box containing N moving (i.e. non-frozen) particles, $N_f = 3(N-1)$. γ_p and $\sigma_p \equiv \sqrt{\frac{2}{3}\gamma_p W_g k_B T}$ are respectively the drag and random coefficients for the piston and $\zeta_{p,\alpha}$ is a Gaussian random number for dimension α (this is set to the same value for all three dimensions if operating isotropically). When both γ_p and σ_p are set to zero, the Langevin barostat reduces to the **extended system** method.

The subsequent simulation cell size L_α can be determined by

$$\dot{L}_\alpha = \frac{p_{g,\alpha} L_\alpha}{W_g} = u_{g,\alpha} L_\alpha \tag{11.20}$$

with the barostat mass W_g chosen to be equal to $N k_B T \tau_p^2$, where τ_p is the characteristic barostat time and should be set equal to between $\frac{2}{\gamma_p}$ and $\frac{10}{\gamma_p}$.

The velocities and positions of the particles are calculated by integration of slightly modified differential equations:

$$\frac{dv_{i,\alpha}}{dt} = \frac{F_{i,\alpha}}{m_i} - u_{g,\alpha} v_{i,\alpha} - \frac{1}{N_f} v_{i,\alpha} \sum_\alpha u_{g,\alpha} \tag{11.21}$$

$$\frac{dr_{i,\alpha}}{dt} = v_{i,\alpha} + u_{g,\alpha} r_{i,\alpha} \tag{11.22}$$

where the force on particle i, $\vec{F}_i$, includes any thermostatting forces: the time integral of these forces can be determined for all thermostat types.

The implementation of this barostat is carried out using the Velocity Verlet scheme to integrate the equations of motion for both the particles and the barostat. The first Velocity Verlet stage integrates the forces on the particles

$$v_{i,\alpha}\left(t + \tfrac{1}{2}\Delta t\right) = v_{i,\alpha}(t) + \frac{\Delta t}{2} \frac{F_{i,\alpha}(t)}{m_i} - \Delta t u_{g,\alpha} v_{i,\alpha} \tag{11.23}$$

which is followed by a similar integration for the barostat velocity:

$$u_{g,\alpha}\left(t + \tfrac{1}{2}\Delta t\right) = u_{g,\alpha}(t) + \frac{\Delta t}{2} \frac{F_{g,\alpha}(t)}{W_g} \tag{11.24}$$

before the positions and simulation box dimensions are updated:

$$r_{i,\alpha}(t+\Delta t) = \exp\left(u_{g,\alpha}\left(t+\tfrac{1}{2}\Delta t\right)\Delta t\right)\left\{r_{i,\alpha}(t) + \Delta t v_{i,\alpha}\left(t+\tfrac{1}{2}\Delta t\right)\right\} \tag{11.25}$$

$$L_\alpha(t+\Delta t) = \exp\left(u_{g,\alpha}\left(t+\tfrac{1}{2}\Delta t\right)\Delta t\right) L_\alpha(t) \tag{11.26}$$

At this point the forces at the end of the time step are calculated (including thermostatting forces), along with the system pressure. Since the the barostat force requires correct velocities for both the particles and barostat, an iterative procedure is required which begins by calculating an initial guess for the barostat velocity at the end of the time step:

$$u_{g,\alpha}^{(0)}(t+\Delta t) = u_{g,\alpha}(t-\Delta t) + \frac{2F_{g,\alpha}(t)\Delta t}{W_g}$$

Each iteration starts by calculating the particle velocities in a slightly modified second Velocity Verlet step:

$$v_{i,\alpha}^{(n+1)}(t+\Delta t) = \frac{\exp\left(u_{g,\alpha}\left(t+\tfrac{1}{2}\Delta t\right)\Delta t\right) v_{i,\alpha}\left(t+\tfrac{1}{2}\Delta t\right) + \frac{\Delta t}{2}\frac{F_{i,\alpha}(t+\Delta t)}{m_i}}{1 + u_{g,\alpha}^{(n)}\Delta t} \tag{11.27}$$

The barostat force is then calculated using the same Gaussian random numbers for each iteration:

$$F_{g,\alpha}^{(n+1)}(t+\Delta t) = V(P_\alpha - P_0) + \frac{1}{N_f}\sum_i m_i \left(v_i^{(n+1)}\right)^2 - \gamma_p u_{g,\alpha}^{(n)} W_g + \sigma_p \zeta_{p,\alpha} \tag{11.28}$$

and the barostat velocity is recalculated:

$$u_{g,\alpha}^{(n+1)}(t+\Delta t) = u_{g,\alpha}\left(t+\tfrac{1}{2}\Delta t\right) + \frac{F_{g,\alpha}^{(n+1)}(t+\Delta t)\Delta t}{2W_g} \tag{11.29}$$

Equations 11.27 to 11.29 are repeated until convergence in particle velocities is achieved, i.e. when

$$\frac{\sum_i \left(\vec{v}_i^{(n+1)}(t+\Delta t) - \vec{v}_i^{(n)}(t+\Delta t)\right)^2}{3N_f} < \epsilon$$

with ϵ as a numerical tolerance (set to 10^{-6} by default). This normally takes a few iterations per time step without requiring recalculation of particle forces or rescaling of particle coordinates.

11.3.2 Berendsen barostat (`berendsen`)

The governing equation for the Berendsen barostat[2] is a simple differential equation for the pressure:

$$\frac{dP_\alpha}{dt} = \frac{P_0 - P_\alpha}{\tau_p} \tag{11.30}$$

which can be solved to give a scaling factor for the simulation volume, $\vec{\eta}(t)$:

$$\eta_\alpha(t) = 1 - \frac{\beta \Delta t}{\tau_p}\left(P_0 - P_\alpha(t)\right) \tag{11.31}$$

where β is the isothermal compressibility of the system. The exact value of this property is not critical to the algorithm, since it relies on the ratio $\frac{\beta}{\tau_p}$.

The barostat is implemented using a variant of the Velocity Verlet algorithm; after the midstep velocities are determined, the scaling factor for time t is used to modify the particle positions and resize the simulation volume

$$r_{i,\alpha}(t+\Delta t) = \eta_\alpha(t) r_{i,\alpha}(t) + \Delta t v_{i,\alpha}\left(t + \tfrac{1}{2}\Delta t\right) \tag{11.32}$$

$$L_\alpha(t+\Delta t) = \eta_\alpha(t) L_\alpha(t) \tag{11.33}$$

The remainder of the Velocity Verlet algorithm is unchanged, although the scaling factor for the beginning of the next time step can be calculated at this point using Equation 11.31. No iteration is required for this barostat.

11.4 Particle-particle interactions

Pairwise particle interaction parameters in DL_MESO_DPD are specified in the `FIELD` file for each species pair using the directive **interact**ions. Interaction parameter values and lengthscales are stored in the array `vvv(1:npot, 1:mxprm)` in preparation for DPD calculations, the maximum number of parameters `mxprm` dependent on the unbonded potential models in use.

If interaction parameters between different particle species are not specified in the `FIELD` file, these can be determined by mixing rules. Energy and dissipative parameters (e.g. $A_{\alpha\beta}$ and $\gamma_{\alpha\beta}$ for DPD) can be determined for unlike particle pairs as geometric means of these parameters for same-species interactions, e.g.

$$A_{\alpha\beta} = \sqrt{A_{\alpha\alpha} A_{\beta\beta}}$$

while interaction lengths are set to the arithmetic mean, e.g.

$$r_{c,\alpha\beta} = \frac{r_{c,\alpha\alpha} + r_{c,\beta\beta}}{2}$$

It should be noted that interaction lengths have to be less than or equal to the maximum interaction cut-off radius r_c (which applies for dissipative and random force interactions): if the maximum interaction cut-off radius is not specified in the `CONTROL` file, the maximum specified value of $r_{c,\alpha\beta}$ will be used. Frozen particles are included in all interactions but the resultant forces on these particles are not subsequently integrated.

Four types of pairwise interactions between particles are available in DL_MESO_DPD: Lennard-Jones, Weeks-Chandler-Andersen, Groot-Warren (standard) DPD and many-body (density-dependent) DPD. In the case that many-body DPD interactions are used for any particle pair, mixing rules cannot be used and thus interaction parameters for *all* particle pairs must be specified by the user.

11.4.1 Lennard-Jones (`lj`)

The Lennard-Jones potential[30] is a mathematically simple model that approximates interactions (both attractive and repulsive) between pairs of neutral atoms or molecules:

$$U(r_{ij}) = 4\epsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right] \tag{11.34}$$

where ϵ_{ij} is the depth of the potential well and σ_{ij} is the finite distance at which the potential is zero between particles i and j. This potential and its related force are calculated for all interparticle distances (r_{ij}) less than the interaction cutoff radius r_c. Long-range system-wide corrections to the potential and virial are required:

$$U^{lr} = \frac{4\pi}{V} \sum_{\alpha} \sum_{\beta \geq \alpha} (2 - \delta_{\alpha\beta}) N_\alpha N_\beta \epsilon_{\alpha\beta} \left(\frac{\sigma_{\alpha\beta}^{12}}{9r_c^9} - \frac{\sigma_{\alpha\beta}^{6}}{3r_c^3} \right) \tag{11.35}$$

$$\mathcal{W}^{lr} = -\frac{4\pi}{V} \sum_{\alpha} \sum_{\beta \geq \alpha} (2 - \delta_{\alpha\beta}) N_\alpha N_\beta \epsilon_{\alpha\beta} \left(\frac{12\sigma_{\alpha\beta}^{12}}{9r_c^9} - \frac{2\sigma_{\alpha\beta}^{6}}{r_c^3} \right) \tag{11.36}$$

where $\delta_{\alpha\beta}$ is the Kronecker delta (1 when $\alpha = \beta$, 0 when $\alpha \neq \beta$) and N_α the total number of particles of species α. These corrections multiplied by the volume are stored in the array `clr` to eliminate the need to adjust them if the system volume is changed by a barostat.

11.4.2 Weeks-Chandler-Andersen (`wca`)

The Weeks-Chandler-Andersen potential[69] is a modification of the Lennard-Jones potential to produce purely repulsive, short-range interactions:

$$U(r_{ij}) = 4\epsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right] + \epsilon_{ij}. \tag{11.37}$$

This interaction is applied for interparticle distances up to $2^{\frac{1}{6}}\sigma_{ij}$, which should be less than or equal to the interaction cutoff radius r_c. No long-range corrections to potential energy or virials are required.

11.4.3 Standard DPD (`dpd`)

The Groot-Warren (standard) form of DPD[16] uses the following purely repulsive, soft potential:

$$U\left(r_{ij}\right) = \frac{1}{2} A_{ij} r_{c,ij} \left(1 - \frac{r_{ij}}{r_{c,ij}} \right)^2. \tag{11.38}$$

This conservative interaction is applied for interparticle distances up to $r_{c,ij}$, which should be less than or equal to the maximum value r_c.

11.4.4 Many-body DPD (`mdpd`)

The conservative force in standard DPD depends only upon the species interacting and the interparticle separation, which yields a quadratic equation of state. Many-body DPD[46, 66] is a method of providing alternative thermodynamic behaviours to DPD particles by making conservative forces additionally dependent on local densities.

The free energy of an inhomogeneous system with density $\rho(r)$ can be defined as the following in both continuous and ensemble-averaged discrete forms:

$$\mathcal{F} = \int d\vec{r} \rho(\vec{r}) \psi(\overline{\rho}(\vec{r})) \tag{11.39}$$

$$= \left\langle \sum_i \psi(\overline{\rho}(\vec{r_i})) \right\rangle \tag{11.40}$$

where $\psi(\rho)$ is the free energy per particle in a homogeneous system and $\overline{\rho}(\vec{r_i})$ is a function related to the density at (and near) the position of particle i $(\vec{r_i})$. The latter can be approximated by a function dependent on the positions of particles close to particle i $(\tilde{\rho}_i)$ to allow the calculation of an instantaneous free-energy:

$$\tilde{\mathcal{F}} = \sum_i \psi(\tilde{\rho}_i).$$

The effective (conservative) force on particle i can be obtained from the spatial derivative of the free energy, although only the excess part (equivalent to the potential energy U) is required since the kinetic motion of particles automatically accounts for the ideal contribution:

$$\vec{F}_i^C = -\frac{\partial \tilde{\mathcal{F}}^{ex}(\{\vec{r}_k\})}{\partial \vec{r}_i} = -\sum_j \frac{\partial \psi^{ex}(\tilde{\rho}_j)}{\partial \vec{r}_i} \tag{11.41}$$

The force can also be expressed in terms of pairwise interactions, taking a form similar to the standard DPD conservative force (Equation 9.5):

$$\vec{F}_{ij}^C = \left(\frac{\partial \psi^{ex}(\tilde{\rho}_i)}{\partial \tilde{\rho}_i} + \frac{\partial \psi^{ex}(\tilde{\rho}_j)}{\partial \tilde{\rho}_j}\right) w^C(r_{ij}) \frac{\vec{r}_{ij}}{r_{ij}} \tag{11.42}$$

The local-density approximation can be defined as a weighted average of instantaneous densities:

$$\begin{aligned}
\tilde{\rho}_i &= \int d\vec{r}\, w^\rho(\mid \vec{r} - \vec{r_i} \mid)\rho(\vec{r}, \{\vec{r}_k\}) \\
&= \sum_{j \neq i} \int d\vec{r}\, w^\rho(\mid \vec{r} - \vec{r_i} \mid)\delta(\vec{r} - \vec{r}_j) \\
\tilde{\rho}_i &= \sum_{j \neq i} w^\rho(r_{ij}) && (11.43)
\end{aligned}$$

with $w^\rho(r)$ as the weight function vanishing beyond a cutoff r_d (which can be equal to r_c or smaller) and normalized so that $\int_0^\infty 4\pi r^2 w^\rho(r) dr = 1$. The most frequently used form for the weight function is

$$w^\rho(r_{ij}) = \frac{15}{2\pi r_d^3}\left(1 - \frac{r_{ij}}{r_d}\right)^2 \qquad (r_{ij} < r_d) \tag{11.44}$$

which reduces to standard DPD when the excess free energy per particle is set to $\psi^{ex}(\tilde{\rho}) = \frac{\pi}{30}A\tilde{\rho}$.

Multiple-component many-body DPD is also possible by defining partial local densities, e.g. for component α

$$\tilde{\rho}_i^\alpha = \sum_{j \in \alpha, j \neq i} w^\rho(r_{ij}) \tag{11.45}$$

and generalizing Equation 11.42:

$$\vec{F}_{ij}^C = \left(\frac{\partial \psi_{c(i)}^{ex}(\{\tilde{\rho}^\alpha\}_i)}{\partial \tilde{\rho}_{c(j)}} + \frac{\partial \psi_{c(j)}^{ex}(\{\tilde{\rho}^\alpha\}_j)}{\partial \tilde{\rho}_{c(i)}}\right) w^C(r_{ij}) \frac{\vec{r}_{ij}}{r_{ij}} \tag{11.46}$$

with $c(i)$ as the component to which particle i belongs (e.g. if $i \in \alpha$, $c(i) = \alpha$) and $\{\tilde{\rho}^\alpha\}_i$ as the set of local densities of different components at the position of particle i.

The `manybody_module` includes a routine (`local_density`) to calculate local densities for each species using Equation 11.45: the overall density for each particle can be obtained by a simple sum over all species. The user can modify the routines `manybody_force` and `manybody_potential` to apply their own choices for many-body forces and potentials respectively. Up to five many-body interaction parameters per species pair can be specified in the `FIELD` file.

The many-body DPD example provided with DL_MESO_DPD produces a van der Waals-like equation of state and can be used to model vapour/liquid interfaces[68]. The potential (excess free energy) per particle is given by:

$$\psi^{ex}(\tilde{\rho}) = \frac{\pi}{30}A_{ij}\overline{\overline{\rho}} + \frac{\pi r_d^4}{30}B_{ij}\tilde{\rho}^2 \tag{11.47}$$

where $\overline{\overline{\rho}}$ is equivalent to $\tilde{\rho}$ but with the cutoff set to r_c instead of r_d. The associated pairwise force is equal to

$$\vec{F}_{ij}^C = \left[A_{ij}\left(1 - \frac{r_{ij}}{r_{c,ij}}\right) + B_{ij}\left(\rho_i + \rho_j\right)\left(1 - \frac{r_{ij}}{r_d}\right)\right]\frac{\vec{r_{ij}}}{r_{ij}} \tag{11.48}$$

In the routine provided, the terms with A_{ij} for both force and potential are calculated as though they are standard DPD. By setting $A_{ij} < 0$ and $B_{ij} > 0$, a vapour/liquid mixture can be modelled and its equation of state for a single component is given as

$$p = \rho k_B T + \alpha A \rho^2 + 2\alpha B r_d^4 \left(\rho^3 - c\rho^2 + d\right)$$

where $\alpha \approx 0.101$, c and d are numerical offsets. Parameters A_{ij} and B_{ij} for each interacting pair of species can be specified in the `FIELD` file.

11.5 Long-ranged Electrostatic (Coulombic) Potentials

Compared to other interactions in DPD, electrostatic interactions act over considerably longer ranges, which can also include periodic images of the system. The governing equation for finding the electric potential is the Poisson equation, shown here in dimensionless form[15]:

$$\nabla \cdot (p(\vec{r})\nabla\varphi) = -\Gamma\rho \tag{11.49}$$

where φ is the electric potential, ρ the charge density (concentration of cations minus concentration of anions per unit volume), $p(\vec{r})$ the local polarizability relative to a reference medium (e.g. water) and Γ the coupling constant for the reference medium. The latter is given by

$$\Gamma = \frac{e^2}{k_B T \epsilon_0 \epsilon_r r_c}$$

with e as the electron charge, ϵ_0 the dielectric constant of a vacuum and ϵ_r the relative permittivity of the reference medium. For water at room temperature (298K) with N_m molecules per DPD particle, $\Gamma \approx 20.00 N_m^{-\frac{1}{3}}$. Alternatively, the total electrostatic potential energy can be expressed as a sum of Coulombic energies (which also include periodic images), i.e.

$$U = \frac{\Gamma}{4\pi}\sum_i \sum_{j>i} \frac{q_i q_j}{|r_{ij}|} \tag{11.50}$$

11.5.1 Standard Ewald sum with exponential charge smearing (ewald)

The method currently used in DL_MESO_DPD to determine the electrostatic potential is an Ewald summation[14]:

$$\begin{aligned} U &\equiv q\varphi \\ &= U^{sr} + U^{lr} + U^{sc} + U^{cc} \end{aligned} \tag{11.51}$$

where U^{sr} is a short-range potential energy term that sums quickly in real space, U^{lr} is a long-range term that sums quickly in Fourier or reciprocal space, U^{sc} is a self-energy correction term and U^{cc} is a correction for systems with a net charge.

While the original form of the short-range electrostatic potential uses point charges, this cannot be used unmodified for DPD simulations: soft beads used in combination with unlike point charges would collapse on top of each other, forming infinitely strong ion pairs. The charges are therefore spread out over a finite volume using a smearing charge distribution $f(r)$. The current approach uses a Slater-type distribution, i.e.

$$f(r) = \frac{q}{\pi\lambda^3}\exp\left(-\frac{2r}{\lambda}\right) \tag{11.52}$$

where λ is the decay length of the charge. This gives a potential energy between charged particles i and j of

$$U(r_{ij}) = \frac{\Gamma q_i q_j}{4\pi r_{ij}}\left[1 - (1 + \beta r_{ij})\exp(-2\beta r_{ij})\right] \tag{11.53}$$

and the corresponding electrostatic force is

$$\vec{F}_{ij}^{E}(r_{ij}) = \frac{\Gamma q_i q_j}{4\pi r_{ij}^2}\left[1 - \exp(-2\beta r_{ij})\left(1 + 2\beta r_{ij}(1 + \beta r_{ij})\right)\right]\frac{\vec{r}_{ij}}{r_{ij}} \tag{11.54}$$

where $\beta \equiv \frac{r_c}{\lambda}$. For large particle separations these expressions reduce down to the standard expressions for point charges and thus the reciprocal space part of the Ewald sum can be calculated without modification.

The real space terms for the Ewald sum are modifications of the above expressions, evaluated for particle pairs when the separation is less than the electrostatic short-range (real space) cutoff, r_e. The short-range potential energy between particles i and j is given as

$$U_{ij}^{sr} = \frac{\Gamma q_i q_j}{4\pi r_{ij}}\text{erfc}(\alpha r_{ij})\left(1 - (1 + \beta r_{ij})\exp(-2\beta r_{ij})\right) \tag{11.55}$$

and the pairwise force is

$$\vec{F}_{ij}^{E,sr} = \frac{\Gamma q_i q_j}{4\pi r_{ij}^2}\left(\frac{2\alpha r_{ij}}{\sqrt{\pi}}\exp(-\alpha^2 r_{ij}^2) + \text{erfc}(\alpha r_{ij})\right)\left(1 - \exp(-2\beta r_{ij})\left(1 + 2\beta r_{ij}(1 + \beta r_{ij})\right)\right)\frac{\vec{r}_{ij}}{r_{ij}} \tag{11.56}$$

where α is a convergence parameter that controls the real space contribution, chosen to give negligible contributions beyond the real-space cutoff. If a calculational precision of ϵ is required for the Ewald sum, the required value of α is equal to $\frac{\sqrt{|\ln(\epsilon r_e)|}}{r_e}$.

The long-range term for the Ewald sum requires the reciprocal vector $\vec{k}$, which for an orthogonal periodic simulation box of dimensions $L_x \times L_y \times L_z$ is given by

$$\vec{k} = \begin{bmatrix} \frac{2\pi k_1}{L_x} \\ \frac{2\pi k_2}{L_y} \\ \frac{2\pi k_3}{L_z} \end{bmatrix}$$

where k_1, k_2 and k_3 are integers (positive and negative) from zero to large values specified by the user as k_1^{max}, k_2^{max} and k_3^{max} respectively for x-, y- and z-dimensions: the maximum reciprocal vector $\vec{k}_{max}$ using the maximum k values can also be defined. Adjustments can be made to this vector to account for shearing boundaries[70].

The long-range potential energy term for the entire system is given by

$$U_{tot}^{lr} = \frac{\Gamma}{2V}\sum_{\vec{k}\neq 0}^{\vec{k}_{max}}\frac{\exp\left(-\frac{k^2}{4\alpha^2}\right)}{k^2}\left|\sum_j q_j \exp\left(-i\vec{k}\cdot\vec{r}_j\right)\right|^2 \tag{11.57}$$

where i is the imaginary constant ($\sqrt{-1}$). Differentiation of the potential gives the long-range electrostatic force on a particle:

$$\vec{F}_j^{E,lr} = -\frac{\Gamma q_j}{V}\sum_{\vec{k}\neq 0}^{\vec{k}_{max}} i\vec{k}\exp\left(i\vec{k}\cdot\vec{r}_j\right)\frac{\exp\left(-\frac{k^2}{4\alpha^2}\right)}{k^2}\sum_n q_n \exp\left(-i\vec{k}\cdot\vec{r}_n\right) \tag{11.58}$$

Charged frozen particles may interact with non-frozen particles but interactions between frozen particles must be excluded. While the real space electrostatic potential and forces between charged frozen particles can be ignored, pairwise correction terms are required to remove their contributions in reciprocal space. The potential energy to be removed between a pair of charged frozen particles with Slater-like charge distributions is given as

$$U_{ij}^{lr,corr} = \frac{\Gamma q_i q_j}{4\pi r_{ij}}\text{erf}(\alpha r_{ij})\left(1 - (1 + \beta r_{ij})\exp(-2\beta r_{ij})\right) \tag{11.59}$$

while the pairwise force is

$$\vec{F}_{ij}^{E,lr,corr} = \frac{\Gamma q_i q_j}{4\pi r_{ij}^2}\left(\text{erf}(\alpha r_{ij}) - \frac{2\alpha r_{ij}}{\sqrt{\pi}}\exp(-\alpha^2 r_{ij}^2)\right)\left(1 - \exp(-2\beta r_{ij})\left(1 + 2\beta r_{ij}(1 + \beta r_{ij})\right)\right)\frac{\vec{r}_{ij}}{r_{ij}}. \tag{11.60}$$

Even though the forces on frozen particles are ignored during integration, these do contribute to the system potential energy, virial and stress tensor terms and should therefore be evaluated.

The self-energy correction term is constant for a given system for all time steps and is equal to

$$U_{tot}^{sc} = -\frac{\Gamma\alpha}{4\pi^{\frac{3}{2}}} \sum_i q_i^2 \tag{11.61}$$

while the system charge correction is

$$U_{tot}^{cc} = -\frac{\Gamma}{8\alpha^2 V} \left(\sum_i q_i \right)^2 . \tag{11.62}$$

No additional forces are required for these terms.

In DL_MESO_DPD, the module `ewald_module` contains the most important routines for calculating electrostatic interactions using the algorithm described above. The real space (short-range) component is calculated (in routine `ewald_real_slater`) using a link cell algorithm with a cutoff radius of r_e; a larger boundary halo than for standard pairwise interactions is therefore required. The reciprocal space (long-range) component is calculated using the scheme described by [58], parallelized by distribution over atomic sites, which requires global summations but is more efficient in terms of memory usage than distribution of $\vec{k}$ vectors. The routine `ewald_reciprocal` also adds the self-interaction and charged system corrections, which are calculated using the routine `elecgen` in `config_module`. The routine `ewald_frozen_slater` calculates the corrections to forces, potential energy, virial and stress tensors that are required to exclude interactions between charged frozen particles: these corrections only need to be calculated once if the volume is held constant (and thus the frozen particles do not move) but have to be recalculated at each time step if a barostat is applied.

This method can be invoked by using the directives **ewald**, **perm**ittivity and **smear** in the `CONTROL` file: the first is used to set the real-space convergence parameter (α) and k-space vector range, the second to set the permittivity coupling constant (Γ) and the third with the keyword **slater** used to set the smearing coefficient β.

11.6 Bond interactions between particles

Molecules of particles bonded together can be included in calculations using a `FIELD` file to define the properties and topologies of the bonds, angles and dihedrals between them. These data are used in the `start` subroutine to add the specified numbers of molecules into the system before DPD calculations commence and to create tables listing the particles that are included in bonds, angles and dihedrals.

11.6.1 Stretching bonds

DL_MESO_DPD can model four forms of bond potential (and corresponding force) between specified particles, all of which are functions of the distance between them. The available bond forms between particles i and j are as follows:

1. Harmonic (Hookean/Fraenkel) bond:

$$U(r_{ij}) = \frac{\kappa}{2} (r_{ij} - r_0)^2 \tag{11.63}$$

2. (Shifted) Finitely Extendible Non-linear Elastic (FENE) bond:

$$U(r_{ij}) = \begin{cases} -\frac{1}{2}\kappa r_{max}^2 \ln \left[1 - \frac{(r_{ij}-r_0)^2}{r_{max}^2}\right] & r_{ij} < r_0 + r_{max} \\ \infty & r_{ij} \geq r_0 + r_{max} \end{cases} \tag{11.64}$$

3. Marko-Siggia Worm-Like Chain (WLC)[37]:

$$U(r_{ij}) = \begin{cases} \frac{k_B T}{2A_p}\left[\frac{1}{2\left(1-\frac{r_{ij}}{r_{max}}\right)} - \frac{1}{2\left(1+\frac{r_{ij}}{r_{max}}\right)} + \frac{r_{ij}^2}{r_{max}^2}\right] & r_{ij} < r_{max} \\ \infty & r_{ij} \geq r_{max} \end{cases} \tag{11.65}$$

4. Morse potential bond[44]:

$$U(r_{ij}) = D_e \left[1 - \exp\left(-\beta\left(r_{ij} - r_0\right)\right)\right]^2 \tag{11.66}$$

where $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$, r_0 is an equilibrium bond length, r_{max} the maximum specified bond length or extension, κ is a spring force constant, A_p the persistence length of a wormlike chain, D_e the potential well depth and β the potential 'width'.

The force on particle i due to a bond potential is obtained from the general formula:

$$\vec{F}_i = -\frac{1}{r_{ij}}\left[\frac{\partial}{\partial r_{ij}} U(r_{ij})\right]\vec{r}_{ij} \tag{11.67}$$

with the force acting on particle j equal to the negative of this, and the virial contribution from the stretching bond given by

$$\mathcal{W} = -\vec{r}_{ij} \cdot \vec{F}_i, \tag{11.68}$$

with only *one* contribution per bond[2].

11.6.2 Bond angles

DL_MESO_DPD includes three methods for modelling potentials and forces between three bonded particles due to the angle formed between them, θ_{ijk}. The potentials are given as follows:

1. Harmonic:

$$U(\theta_{ijk}) = \frac{\kappa}{2}\left(\theta_{ijk} - \theta_0\right)^2 \tag{11.69}$$

2. Harmonic cosine:

$$U(\theta_{ijk}) = \frac{\kappa}{2}\left(\cos\theta_{ijk} - \cos\theta_0\right)^2 \tag{11.70}$$

3. Cosine:

$$U(\theta_{ijk}) = A\left[1 + \cos\left(m\theta_{ijk} - \delta\right)\right] \tag{11.71}$$

where A and κ are angle force constants, m is the multiplicity, δ the angle at minimum potential and θ_0 an equilibrium bond angle.

The angle across particles i, j and k can be determined from the bond vectors $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$ and $\vec{r}_{kj} = \vec{r}_k - \vec{r}_j$:

$$\theta_{ijk} = \cos^{-1}\left\{\frac{\vec{r}_{ij} \cdot \vec{r}_{kj}}{r_{ij} r_{kj}}\right\} \tag{11.72}$$

The most general form for the bond angle potential is given thus:

$$U(\theta_{ijk}, r_{ij}, r_{kj}) = A(\theta_{ijk})\, S(r_{ij})\, S(r_{kj})\, S(r_{ik}) \tag{11.73}$$

[2] This expression is also used for the virial contribution from the standard DPD pairwise forces, i.e. Equation (9.3), again only applying a single contribution per particle pair.

with $A(\theta)$ as a purely angular function and $S(r)$ a screening or truncation function. The force on particle ℓ in dimension α is thus given by:

$$\begin{aligned}
F_\ell^\alpha &= -\frac{\partial}{\partial x_\alpha} U(\theta_{ijk}, r_{ij}, r_{kj}) && (11.74)\\
&= -S(r_{ij}) S(r_{kj}) S(r_{ik}) \frac{\partial}{\partial r_\ell^\alpha} A(\theta_{ijk})\\
&\quad -A(\theta_{ijk}) S(r_{kj}) S(r_{ik}) (\delta_{\ell i} - \delta_{\ell j}) \frac{r_{ij}^\alpha}{r_{ij}} \frac{\partial}{\partial r_{ij}} S(r_{ij})\\
&\quad -A(\theta_{ijk}) S(r_{ij}) S(r_{ik}) (\delta_{\ell k} - \delta_{\ell j}) \frac{r_{kj}^\alpha}{r_{kj}} \frac{\partial}{\partial r_{kj}} S(r_{kj})\\
&\quad -A(\theta_{ijk}) S(r_{ij}) S(r_{kj}) (\delta_{\ell k} - \delta_{\ell i}) \frac{r_{ik}^\alpha}{r_{ik}} \frac{\partial}{\partial r_{ik}} S(r_{ik}) && (11.75)
\end{aligned}$$

with $\delta_{ab} = 1$ if $a = b$ and $\delta_{ab} = 0$ if $a \neq b$. In the absence of screening terms, the above formula reduces to

$$\begin{aligned}
F_\ell^\alpha &= -\frac{\partial}{\partial r_\ell^\alpha} A(\theta_{ijk}) && (11.76)\\
&= \frac{1}{\sin\theta_{ijk}} \frac{\partial}{\partial \theta_{ijk}} A(\theta_{ijk}) \times\\
&\left\{ (\delta_{\ell j} - \delta_{\ell i}) \frac{r_{kj}^\alpha}{r_{ij} r_{kj}} + (\delta_{\ell j} - \delta_{\ell k}) \frac{r_{ij}^\alpha}{r_{ij} r_{kj}} - \cos(\theta_{ijk}) \left[(\delta_{\ell j} - \delta_{\ell i}) \frac{r_{ij}^\alpha}{r_{ij}^2} + (\delta_{\ell j} - \delta_{\ell k}) \frac{r_{kj}^\alpha}{r_{kj}^2} \right] \right\} && (11.77)
\end{aligned}$$

The contribution to the virial from the angle is given by

$$\mathcal{W} = -\left(\vec{r}_{ij} \cdot \vec{F}_i + \vec{r}_{kj} \cdot \vec{F}_k \right) \tag{11.78}$$

which is equal to zero for bond angle potentials without screening terms[59].

11.6.3 Bond dihedrals

Three potential models for bond dihedrals along particles i, j, k and l are provided in DL_MESO_DPD as follows:

1. Cosine torsion:
$$U(\phi_{ijkl}) = A\left[1 + \cos(m\phi_{ijkl} - \delta)\right] \tag{11.79}$$

2. Harmonic:
$$U(\phi_{ijkl}) = \frac{\kappa}{2} (\phi_{ijkl} - \phi_0)^2 \tag{11.80}$$

3. Harmonic cosine:
$$U(\phi_{ijkl}) = \frac{\kappa}{2} (\cos\phi_{ijkl} - \cos\phi_0)^2 \tag{11.81}$$

where A and κ are dihedral force constants, m is the multiplicity, δ the dihedral at minimum potential and ϕ_0 an equilibrium bond dihedral.

The dihedral angle across all four particles (or between planes ij and kl) is given by

$$\phi_{ijkl} = \cos^{-1} B(\vec{r}_{ij}, \vec{r}_{jk}, \vec{r}_{kl}) \tag{11.82}$$

where

$$B(\vec{r}_{ij}, \vec{r}_{jk}, \vec{r}_{kl}) = \frac{(\vec{r}_{ij} \times \vec{r}_{jk}) \cdot (\vec{r}_{jk} \times \vec{r}_{kl})}{|\vec{r}_{ij} \times \vec{r}_{jk}||\vec{r}_{jk} \times \vec{r}_{kl}|} \tag{11.83}$$

which gives a negative value for ϕ_{ijkl} if the vector $(\vec{r}_{ij} \times \vec{r}_{jk}) \cdot (\vec{r}_{jk} \times \vec{r}_{kl})$ is in the same direction as the bond vector $\vec{r}_{jk}$ and positive if in the opposite direction.

The force on particle ℓ acting in the α direction is given by

$$F_\ell^\alpha = -\frac{\partial}{\partial x_\alpha} U(\phi_{ijkl}) \tag{11.84}$$
$$= \frac{1}{\sin\phi_{ijkl}} \frac{\partial}{\partial \phi_{ijkl}} U(\phi_{ijkl}) \frac{\partial}{\partial r_\ell^\alpha} B(\vec{r}_{ij}, \vec{r}_{jk}, \vec{r}_{kl}). \tag{11.85}$$

Using the following definition

$$\left[\vec{a}\ \vec{b}\right]_\alpha \equiv \sum_\beta (1 - \delta_{\alpha\beta}) a^\beta b^\beta$$

the derivative of $B(\vec{r}_{ij}, \vec{r}_{jk}, \vec{r}_{kl})$ is given by

$$\frac{\partial}{\partial r_\ell^\alpha} B(\vec{r}_{ij}, \vec{r}_{jk}, \vec{r}_{kl}) = \frac{1}{|\vec{r}_{ij} \times \vec{r}_{jk}||\vec{r}_{jk} \times \vec{r}_{kl}|} \frac{\partial}{\partial r_\ell^\alpha} \{(\vec{r}_{ij} \times \vec{r}_{jk}) \cdot (\vec{r}_{jk} \times \vec{r}_{kl})\}$$
$$-\frac{\cos\phi_{ijkl}}{2} \left\{ \frac{1}{|\vec{r}_{ij} \times \vec{r}_{jk}|^2} \frac{\partial}{\partial r_\ell^\alpha} |\vec{r}_{ij} \times \vec{r}_{jk}|^2 + \frac{1}{|\vec{r}_{jk} \times \vec{r}_{kl}|^2} \frac{\partial}{\partial r_\ell^\alpha} |\vec{r}_{jk} \times \vec{r}_{kl}|^2 \right\} \tag{11.86}$$

with

$$\begin{aligned} \frac{\partial}{\partial r_\ell^\alpha} \{(\vec{r}_{ij} \times \vec{r}_{jk}) \cdot (\vec{r}_{jk} \times \vec{r}_{kl})\} = \; & r_{ij}^\alpha \left([\vec{r}_{jk}\vec{r}_{jk}]_\alpha (\delta_{\ell k} - \delta_{\ell l}) + [\vec{r}_{jk}\vec{r}_{kl}]_\alpha (\delta_{\ell k} - \delta_{\ell j}) \right) + \\ & r_{jk}^\alpha \left([\vec{r}_{ij}\vec{r}_{jk}]_\alpha (\delta_{\ell l} - \delta_{\ell k}) + [\vec{r}_{jk}\vec{r}_{kl}]_\alpha (\delta_{\ell j} - \delta_{\ell i}) \right) + \\ & r_{kl}^\alpha \left([\vec{r}_{ij}\vec{r}_{jk}]_\alpha (\delta_{\ell k} - \delta_{\ell j}) + [\vec{r}_{jk}\vec{r}_{jk}]_\alpha (\delta_{\ell i} - \delta_{\ell j}) \right) + \\ & 2r_{jk}^\alpha [\vec{r}_{ij}\vec{r}_{kl}]_\alpha (\delta_{\ell l} - \delta_{\ell k}) \end{aligned} \tag{11.87}$$

$$\begin{aligned} \frac{\partial}{\partial r_\ell^\alpha} |\vec{r}_{ij} \times \vec{r}_{jk}|^2 = \; & 2r_{ij}^\alpha \left([\vec{r}_{jk}\vec{r}_{jk}]_\alpha (\delta_{\ell j} - \delta_{\ell i}) + [\vec{r}_{ij}\vec{r}_{jk}]_\alpha (\delta_{\ell j} - \delta_{\ell k}) \right) + \\ & 2r_{jk}^\alpha \left([\vec{r}_{ij}\vec{r}_{ij}]_\alpha (\delta_{\ell k} - \delta_{\ell j}) + [\vec{r}_{ij}\vec{r}_{jk}]_\alpha (\delta_{\ell i} - \delta_{\ell j}) \right) \end{aligned} \tag{11.88}$$

$$\begin{aligned} \frac{\partial}{\partial r_\ell^\alpha} |\vec{r}_{jk} \times \vec{r}_{kl}|^2 = \; & 2r_{kl}^\alpha \left([\vec{r}_{jk}\vec{r}_{jk}]_\alpha (\delta_{\ell l} - \delta_{\ell k}) + [\vec{r}_{jk}\vec{r}_{kl}]_\alpha (\delta_{\ell j} - \delta_{\ell k}) \right) + \\ & 2r_{jk}^\alpha \left([\vec{r}_{kl}\vec{r}_{kl}]_\alpha (\delta_{\ell k} - \delta_{\ell j}) + [\vec{r}_{jk}\vec{r}_{kl}]_\alpha (\delta_{\ell k} - \delta_{\ell l}) \right) \end{aligned} \tag{11.89}$$

It can be shown both algebraically and thermodynamically that the dihedral makes no contribution to the virial[59].

Improper dihedrals — which limit the geometry of molecules — can be applied using the same procedure as standard dihedrals and no distinction is made between them in DL_MESO_DPD.

11.7 Surface interactions

The default boundaries for a simulation box are periodic, i.e. particles leaving the system are replaced at the opposite face with the same velocity. Certain systems may require alternative boundary conditions and DL_MESO_DPD can include these at the system boundaries. The directive **surf**ace in the `CONTROL` file can be used to specify the type of surface interaction (`hard`, `frozen` or `shear`) and which surface(s) are to be included (`srfx`, `srfy`, `srfz`: each set to 0 for periodic boundaries and 1 or greater for other types). Parameters required for wall-particle interactions can be specified using the same directive in the `FIELD` file. The module `surface_module` includes routines to set up and apply boundary conditions at those surfaces.

Care should be taken to ensure that the initial configuration does not include bonds crossing any boundary that will be non-periodic. DL_MESO_DPD will ensure this is the case for simulations starting from scratch but cannot check for bonds crossing non-periodic boundaries in `CONFIG` files.

11.7.1 Hard reflecting boundaries (`hard`)

This boundary condition is applied by using a combination of specular (free-slip) reflection at the system boundaries and a soft short-range wall repulsion to reduce density oscillations[50]. The reflection is achieved by moving any particle leaving the system at a particular boundary back into it and inverting the velocity component normal to the wall (but maintaining the tangential momentum), which is achieved with the routine `surfacebounce`. The soft short-range wall repulsion is given by

$$U_{wall}(z) = \tfrac{1}{2} A_{wall,\alpha} z_c \left(1 - \frac{z}{z_c}\right)^2, \qquad (z < z_c) \tag{11.90}$$

where $A_{wall,\alpha}$ is the repulsive force magnitude with species α, z is the distance between the particle and the wall and z_c the surface repulsion range. The repulsion is applied at the same time as all pairwise forces using the routine `hardreflect`. Since the walls are assumed to be non-porous, no interactions across them are included and thus boundary halos adjacent to walls are not used.

The repulsive force magnitudes between the boundary and each species are specified in the `FIELD` file using the directive **surf**ace, while the same directive in the `CONTROL` file is used to determine the surface repulsion cutoff and which boundary surfaces should be hard and reflecting.

11.7.2 Frozen bead walls (`frozen`)

This boundary condition is applied by adding layers of frozen beads which do not move during the simulation but still interact with all other particles. If appropriate choices for the density of frozen beads in the walls and interactions between frozen and non-frozen particles are made, non-porous boundaries with no slip conditions can be obtained, albeit at the cost of density fluctuations near the walls[51].

To use this boundary condition, a species of frozen beads needs to be specified in the `FIELD` file, along with the interaction types and parameters between this species and all others. The directive **surf**ace is used in the `CONTROL` file to specify which boundary surfaces should include frozen beads, while the same directive in the `FIELD` file identifies the frozen bead species, the bead density and thickness of the wall regions: the number of beads required is automatically determined and the size of the system is adjusted to include the additional wall regions, i.e. the user does not have to include the walls in the system dimensions given in the `CONTROL` file.

This boundary condition can only be set up in this manner for new simulations, either starting from scratch or using a `CONFIG` file. If creating a `CONFIG` file from a previous simulation, users are advised to use the hard reflecting boundary condition described above to ensure molecules remain within the required non-periodic boundaries. Simulations with frozen bead walls can be restarted but these walls must already be included in the restart (`export*`) files and cannot subsequently be added.

11.7.3 Shearing periodic walls (`shear`)

Shearing walls moving at a specified velocity are applied using the Lees-Edwards boundary condition[34]: each particle that moves through the otherwise periodic boundary has its velocity modified and is shifted by a distance related to the wall velocity, i.e.

$$\Delta \vec{x}_w = \vec{V}_w t \tag{11.91}$$

where $\vec{V}_w$ is the velocity of the moving boundary. Interactions between pairs of particles across the periodic boundary (taking the positional shift into account) are still calculated, but pairwise thermostats are not applied across this boundary to avoid quenching the modification of particle velocities, particularly with high values for dissipative force coefficients (γ) or collision frequencies (Γ), and maintain the correct shear rate[5].

The directive **surf**ace is used in the `CONTROL` file to specify which boundaries should move, while the **extern**al directive in the `FIELD` file is used to specify the velocity of the moving walls. This boundary condition is only applied after equilibration has taken place.

Chapter 12

DL_MESO_DPD Input and Output Files

12.1 Input files

All user-specified input files for DL_MESO_DPD must be in ANSI text format, with keywords (where necessary) and numerical values separated from each other with spaces or commas: tabs are currently *not* recognised by the parsing utilities.

Define system: `CONTROL`

The `CONTROL` file contains the control variables for running a DPD simulation and is read by the subroutine `read_control` in `config_module`. Such files can can be obtained either via use of the DL_MESO GUI or by editing existing files of that name, such as those in the `DEMO/DPD` directory. These consist primarily of directives: character strings that appear as the first entry of a data record and invoke a particular operation or provide numerical parameters. Extra options may be added by the inclusion of keywords to qualify a particular directive. Directives can be included in any order except for the simulation name (up to 80 characters long) on the first line of the file and the **finish** directive which marks the end of the file.

A list of the directives available follows, with bold type specifying the minimum number of letters required by DL_MESO_DPD. Some directives may include optional words or parameters as indicated by brackets.

directive:	**meaning:**
boundary **halo** f	set size of boundary halo (overriding default values determined from interaction cutoff, maximum bond lengths and short-range electrostatics) as f length units
close time f	set job closure time to f seconds
cutoff f	set maximum interaction cutoff radius, r_c, to f length units
densvar f	allow for local variation of $\approx f$ % in the system density of particles (useful for non-homogeneous or non-equilibrium simulations, default $f = 0$
electrostatic **cut**off f	set required short-range electrostatic cutoff radius, r_e, to f length units (default $f = r_c$)
ensemble nvt mdvv	select NVT ensemble, DPD thermostat with standard MD-like Velocity Verlet integration (default ensemble if otherwise not specified)
ensemble nvt dpdvv	select NVT ensemble, DPD thermostat with DPD Velocity Verlet integration
ensemble nvt lowe	select NVT ensemble, Lowe-Andersen thermostat
ensemble nvt peters	select NVT ensemble, Peters thermostat
ensemble nvt stoyanov α	select NVT ensemble, Stoyanov-Groot thermostat with coupling parameter α

ensemble npt Q **lang**evin f_1 f_2	select NPT ensemble, thermostat type Q (i.e. *mdvv*, *dpdvv*, *lowe* or *peters*) with Langevin barostat, setting relaxation time (τ_p) and viscosity parameter (γ_p) as f_1 and f_2 respectively
ensemble npt stoyanov α **lang**evin f_1 f_2	select NPT ensemble, Stoyanov-Groot thermostat with coupling parameter α and Langevin barostat with relaxation time f_1 and viscosity parameter f_2
ensemble npt Q **bere**ndsen f	select NPT ensemble, thermostat type Q with Berendsen barostat, setting ratio of compressibility to relaxation time, $\frac{\beta}{\tau_p}$, to f
ensemble npt stoyanov α **bere**ndsen f	select NPT ensemble, Stoyanov-Groot thermostat with coupling parameter α and Berendsen barostat with compressibility/relaxation time ratio f
equilibration (steps) n	equilibrate system for the first n timesteps (default $n = 0$)
ewald (sum) α k_1 k_2 k_3	calculate electrostatic forces using Ewald sum with real-space convergence parameter α and reciprocal space (k-vector) range (k_1, k_2, k_3)
finish	close the `CONTROL` file (last data record)
global bonds	calculate bonded interactions globally by storing bond data on all processors and sharing bonded particle positions (default: calculate bonded interactions locally)
job time f	set maximum job time to f seconds
manybody **cut**off f	set required many-body DPD interaction radius, r_d, to f length units (default $f = r_c$)
ndump n	write restart data to `export` files every n timesteps (default $n = 1000$)
nfold i j k	option to create volumetrically expanded version of current system (described by `CONFIG` and `FIELD` files) by replicating `CONFIG` file's contents (i, j, k) times while preserving topology of `FIELD` file
no config	ignore contents of `CONFIG` file and create initial configuration based purely on `FIELD` file
no electrostatics	ignore electrostatics in simulation
no index	ignore particles' indices as read from `CONFIG` file and set their indexing according to order of reading
no isotropy	switch off isotropy for barostat (i.e. allow uneven contractions and expansions of simulation volume)
permittivity (constant) f	set permittivity constant for system, Γ, to f
pressure f	set required system pressure to f (target pressure for constant pressure ensembles)
print (every) n	print system data every n timesteps
rcut f	see **cut**off
restart	restart job from end point of previous run (i.e. continue current simulation using `export` files)
restart noscale	restart job from previous run without rescaling to system temperature (i.e. begin a new simulation from older run without temperature reset)
restart scale	restart job from previous run after rescaling to system temperature (i.e. begin a new simulation from older run with temperature reset)
scale (temperature) (every) n	rescale system temperature every n steps during equilibration
smear slater f	apply Slater-type (exponential) charge smearing with coefficient, β, set to f
stack (size) n	set rolling average stack to n timesteps
stats (every) n	accumulate statistics data and write to `CORREL` file every n timesteps
steps n	run simulation for n timesteps
surface **cut**off f	set required surface repulsive range, z_c, to f length units (default $f = r_c$)
surface **hard** i	set hard adsorbing walls orthogonal to i-axis (`x`, `y`, `z`) if specified (multiple walls can be specified if separated with spaces or commas)

surface **froz**en i	set frozen bead walls orthogonal to i-axis (`x`, `y`, `z`) if specified (multiple walls can be specified if separated with spaces or commas). Note that this can only be used for brand new simulations: this directive is ignored if a simulation is restarted.
surface **shea**r i	set moving Lees-Edwards periodic walls orthogonal to i-axis (`x`, `y`, `z`) if specified (a single wall can be specified and only the first specified dimension will be used)
temperature f	set required simulation temperature (k_BT) to f (target temperature for constant temperature ensembles)
trajectory (i) j	write trajectory data to `HISTORY` file(s) with controls: i = start timestep for dumping configurations (default: equilibration time), j = timestep interval between configurations
timestep f	set timestep to f time units
volume f_1 (f_2 f_3)	set system size to either cubic volume f_1 or orthorhombic dimensions (f_1, f_2, f_3)

While not every directive has to be included in the `CONTROL` file for a valid simulation and many hold default values if unspecified, the following are mandatory and must be set to values greater than zero:

- **cut**off
- **temp**erature
- **timestep**
- **vol**ume (if no `CONFIG` file is available)

Superfluous parameters and switches for particular systems (e.g. specified pressure for constant volume simulations) can be safely omitted from the `CONTROL` file without causing runtime problems. If the user wishes to include new directives in the `CONTROL` file, modifications to the parameter recognition loop in the `read_config` subroutine (`config_module`) will be required.

Define interactions: `FIELD`

The `FIELD` file contains the species and force field information required for both bonded and unbonded interactions, and is read by the `read_field` and `scan_field` subroutines in `config_module`. Apart from the name of the simulation (up to 80 characters) in the first line, this file contains a number of directives, each indicating the type and number of interactions to follow.

The species information *must* be provided first, as this will be required to specify interaction data (which can be included in any order), using the directive **species** n. This indicates that data for n species are to follow, each species given in a single line using the following format:

name	a8	name of species
mass	real	particle mass for species
charge	real	particle charge for species
populations	integer	unbonded population of species
frozen	integer	determines whether particles of this species are frozen (1) or not (0)

The unbonded population can be omitted for species which are wholly contained in molecules, as can the frozen particle parameter for unfrozen species.

Non-bonded interactions are specified with the directive **interact**ions n, with n pairwise interactions to follow; each is given in a single line using the format:

species 1	a8	name of species 1
species 2	a8	name of species 2
key	a4	interaction key, see Table 12.2
variable 1	real	interaction parameter, see Table 12.2
variable 2	real	interaction parameter, see Table 12.2
variable 3	real	interaction parameter, see Table 12.2
variable 4	real	interaction parameter, see Table 12.2
variable 5	real	interaction parameter, see Table 12.2
variable 6	real	interaction parameter, see Table 12.2
variable 7	real	interaction parameter, see Table 12.2

If any interactions are many-body DPD, the interactions for *all* possible species pairs must be specified in the `FIELD` file and values for all parameters must be given, even if not all of them are required for the many-body DPD model in use. Otherwise only like-like (same species, i.e. $i = j$) interactions are required, as any missing interaction data can be derived using mixing rules. If using the Lowe-Andersen or Stoyanov-Groot thermostats, the dissipative factor γ_{ij} should be replaced with the bath collision frequency Γ_{ij}.

Table 12.2: Non-bonded interactions

key	interaction type	Parameters (1-7)						
lj	Lennard-Jones	ϵ_{ij}	σ_{ij}	γ_{ij}	-	-	-	-
wca	Weeks-Chandler-Andersen	ϵ_{ij}	σ_{ij}	γ_{ij}	-	-	-	-
dpd	Groot-Warren DPD	A_{ij}	$r_{c,ij}$	γ_{ij}	-	-	-	-
mdpd	Many-body DPD	A_{ij}	B_{ij}	C_{ij}	D_{ij}	E_{ij}	$r_{c,ij}$	γ_{ij}

Molecules are specified using the directive **molecul**es n, with data for n molecule types to follow. Immediately after this directive, the following records are included to define each molecule type:

1. Molecule name
   which can be a character string of up to 8 characters in length

2. **nummols** n
   where n is the number of times a molecule of this type appears in the system. This is followed by the data for the molecule type:

3. **bead**s n
   where n gives the number of beads (particles) in this molecule type. A number of records follow for each bead:

name	a8	name of species
x	real	relative x-coordinate for bead
y	real	relative y-coordinate for bead
z	real	relative z-coordinate for bead

The relative coordinates are used to define the initial shape of the molecule when it is inserted into the system: these are not used if an initial configuration is already available.

4. **no iso**mer
   indicates that the molecule shape should not be reflected or otherwise modified when added to the system. This directive is optional and should be left out if no restrictions on molecule insertion are to apply.

5. **bond**s n
   where n gives the number of flexible bonds in the molecule. Each of the subsequent n records contains:

bond key	a4	potential key, see Table 12.3
index 1 (i)	integer	first bead index in bond
index 2 (j)	integer	second bead index in bond
variable 1	real	potential parameter, see Table 12.3
variable 2	real	potential parameter, see Table 12.3
variable 3	real	potential parameter, see Table 12.3
variable 4	real	potential parameter, see Table 12.3

   Note that the bead indices are those arising from numbering each bead in the molecule from 1 to the number specified in the **bead**s directive for this molecule. The same numbering scheme applies for all descriptions of the molecule: DL_MESO_DPD will itself construct the global numbers for all particles in the system.

6. **angle**s n
   where n gives the number of angle bonds in the molecule. Each of the n records following contains:

angle key	a4	potential key, see Table 12.4
index 1 (i)	integer	first bead index in bond angle
index 2 (j)	integer	second bead index in bond angle (central site)
index 3 (k)	integer	third bead index in bond angle
variable 1	real	potential parameter, see Table 12.4
variable 2	real	potential parameter, see Table 12.4
variable 3	real	potential parameter, see Table 12.4
variable 4	real	potential parameter, see Table 12.4

   Angle-based parameters, e.g. θ_0, should be given in degrees. This directive and associated data records need not be specified if the molecule contains no bond angles.

7. **dihed**rals n
   where n gives the number of dihedral interactions in the molecule. Each of the following n records contains:

dihedral key	a4	potential key, see Table 12.5
index 1 (i)	integer	first bead index in bond dihedral
index 2 (j)	integer	second bead index in bond dihedral (central site)
index 3 (k)	integer	third bead index in bond dihedral
index 4 (l)	integer	fourth bead index in bond dihedral
variable 1	real	potential parameter, see Table 12.5
variable 2	real	potential parameter, see Table 12.5
variable 3	real	potential parameter, see Table 12.5
variable 4	real	potential parameter, see Table 12.5

Angle-based parameters, e.g. ϕ_0, should be given in degrees. This directive and associated data records need not be specified if the molecule contains no bond dihedrals.

8. **finish**
   indicates the end of details for a molecule type. Each subsequent molecule type can be entered after this directive, beginning with its name and ending with the **finish** directive.

Table 12.3: Bond potentials

key	potential type	Variables (1-4)				functional form
harm	Harmonic	κ	r_0	-	-	$U(r) = \frac{1}{2}\kappa(r - r_0)^2$
fene	(Shifted) FENE	κ	r_0	r_{max}	-	$U(r) = -0.5\kappa r_{max} \ln \left[1 - \left(\frac{r-r_0}{r_{max}^2}\right)^2\right] : r < r_{max} + r_0$ $U(r) = \infty : r \geq r_{max} + r_0$
wlc	Worm-like chain	A_p	r_{max}	-	-	$U(r) = \frac{k_B T}{2A_p}\left[\frac{1}{2\left(1-\frac{r}{r_{max}}\right)} - \frac{1}{2\left(1+\frac{r}{r_{max}}\right)} + \frac{r^2}{r_{max}^2}\right] : r < r_{max}$ $U(r) = \infty : r \geq r_{max}$
mors	Morse	D_e	r_0	β	-	$U(r) = D_e[1 - \exp(-\beta(r - r_0))]^2$

Table 12.4: Bond angle potentials

key	potential type	Variables (1-4)				functional form
harm	Harmonic	κ	θ_0	-	-	$U(\theta) = \frac{1}{2}\kappa(\theta - \theta_0)^2$
hcos	Harmonic cosine	κ	θ_0	-	-	$U(\theta) = \frac{1}{2}\kappa(\cos\theta - \cos\theta_0)^2$
cos	Cosine	A	m	δ	-	$U(\theta) = A\left[1 + \cos\left(m\theta - \delta\right)\right]$

Table 12.5: Bond dihedral potentials

key	potential type	Variables (1-4)				functional form
cos	Cosine torsion	A	m	δ	-	$U(\phi) = A[1 + \cos(m\phi - \delta)]$
harm	Harmonic	κ	ϕ_0	-	-	$U(\phi) = \frac{1}{2}\kappa(\phi - \phi_0)^2$
hcos	Harmonic cosine	κ	ϕ_0	-	-	$U(\phi) = \frac{1}{2}\kappa(\cos\phi - \cos\phi_0)^2$

Surface interactions can be specified using the directive **surf**ace. If hard adsorbing surfaces are to be used, this directive should be followed by entries specifying the soft short-range repulsions for all species:

name	a8	name of species
A_{wall}	real	soft short-range repulsion between species and wall

while if frozen bead surfaces are in use, the **surf**ace directive should be followed by a single line specifying the properties for the walls to be constructed:

name	a8	name of frozen bead species
ρ_{wall}	real	density of frozen beads in wall regions
x_{wall}	real	thickness of wall region

External fields are flagged by the directive **extern**al, followed by a line with a keyword indicating the type of field to be applied and the field parameters. A gravitational field can be specified using the keyword **grav** and three real values representing the x-, y- and z-components of gravitational acceleration, i.e.

$$\texttt{grav}\ G_x\ G_y\ G_z$$

If using the Lees-Edwards shearing boundary condition, the velocity of the walls in dimension α can be specified using the keyword **shear** and three real values representing the x-, y- and z-components for the velocity at $x_\alpha = L_\alpha$, i.e.

$$\texttt{shear}\ V_{w,x}\ V_{w,y}\ V_{w,z}$$

Note that the velocity at $x_\alpha = 0$ will be equal in magnitude but opposite in direction, and that the velocity component for dimension α will be ignored.

The `FIELD` file must be closed with the directive **close**.

If molecules are to be included in the system, the supplied C++ program `molecule-generate` in the directory `DPD/utility` can be used to either create a new `FIELD` file with the required data or append it to a pre-existing file: see Appendix B for more details. Example files in the `DEMO/DPD` directory can be examined for this purpose.

Define initial state: `CONFIG`

An optional `CONFIG` file can be included to define the initial state of the system, which can include the positions, velocities and forces for each particle[1]. This file is read by the subroutine `read_config` in `start_module` and scanned by the subroutine `scan_config` in `config_module`.

At the beginning of the file, five lines of information (of which the first two are mandatory) have to be included:

- The simulation name (80 characters)
- The `CONFIG` file key `levcfg` (integer), the periodic boundary key `imcon` (integer), the number of particles in the file (integer, optional) and the configuration energy (real, optional)
- The x-, y- and z-components for the x-axis vector (real, optional)
- The x-, y- and z-components for the y-axis vector (real, optional)
- The x-, y- and z-components for the z-axis vector (real, optional)

The file key `levcfg` is set depending on the information available for each particle: 0 for positions only, 1 for positions and velocities or 2 for positions, velocities and forces. If particle velocities are not specified, these are generated at random to produce a distribution corresponding to the required system temperature, while unknown forces are set to zero. The simulation name, number of particles in the file and configuration energy are not read by DL_MESO_DPD and can thus be ignored (although the line for the simulation name must remain). If axes vectors are included in the `CONFIG` file and the value of `imcon` is greater than zero, these will be read on the assumption that the simulation volume is orthorhombic (the only possible shape available in DL_MESO_DPD).

Each particle is represented by a block record, with at least two lines of information:

[1]This file is formatted identically to `CONFIG` files used in DL_POLY[60, 65], except that the origin is set as the back bottom left corner of the simulation volume instead of the centre.

- The species name (8 characters) or number (integer) and the global particle number (integer, optional)
- The x-, y- and z-coordinates for the particle (real)
- The x-, y- and z-components of particle velocity (real, if `levcfg`> 0)
- The x-, y- and z-components of force on the particle (real, if `levcfg`> 1)

If global particle numbers are not included or the **no ind**ex option is invoked in the `CONTROL` file, these are generated automatically for the particles in the order specified by the `CONFIG` file. Care should be taken that any particles belonging to molecules are numbered correctly, since the bond information is assigned in an identical fashion to unspecified systems, i.e. numbering after all loose particles in the relative order specified by the `FIELD` file. If the **nfold** option is invoked in the `CONTROL` file, DL_MESO_DPD will duplicate the given configuration in each Cartesian direction and assign global particle numbers to the enlarged system in a similar fashion, i.e. unbonded particles precede bonded ones and molecules are ordered according to the `FIELD` file.

`CONFIG` files can be created from restart files of previous simulations using the supplied Fortran90 program `exportconfig` in the directory `DPD/utility`; see Appendix B for more details. Frozen particle walls, if specified in the `CONTROL` file, can be added to systems with `CONFIG` files (with or without duplication) but users have to ensure that any molecules do not cross boundaries where frozen particle walls will be placed: no checks are available to prevent this from happening but `CONFIG` files could be created from previous simulations involving hard adsorbing surfaces.

12.2 Output files

General output file: `OUTPUT`

This ANSI text file is generated by all DPD calculations and contains:

- The system and bond/angle/dihedral properties used for calculations.
- Domain decomposition details (Parallel version only).
- The starting positions and velocities of a particle sample.
- The calculation time, current values and rolling averages for the total energy, potential energy (total, electrostatic, and from bond stretching, angles and dihedrals), virial, kinetic energy, pressure and temperature every `nsbpo` time steps.
- Final averages and fluctuations (standard deviations) over all time steps after equilibration.
- The final positions and velocities of a particle sample.
- Elapsed and average times for the calculation.

Restart file(s): `export*`

Each processing unit produces a restart file with a name beginning with `export` every `ndump` time steps. This binary file contains the following information for the time step:

- Name of DPD calculation.
- Numbers of particles, bonds, angles and dihedrals in the processing unit.
- Specified temperature, number of time steps, system volume.

- Current state of random number generators.
- Particle global identity numbers, species and molecule numbers
- Bond, angle and dihedral tables.
- Particle Cartesian coordinates, velocities, forces and virials.
- Current time step and statistical properties (current values, cumulative sums and fluctuations, rolling averages and stacks).

In combination, the `export*` files can restore a stopped DPD calculation. They can also be used to generate a plot at the given time step or a `CONFIG` file for subsequent simulations using the utilities `exportimage` and `exportconfig` respectively in the `DPD/utility` directory; Appendix B gives instructions for their use.

Trajectory file(s): `HISTORY*`

If the **traj**ectory option is specified in the `CONTROL` file, each processing unit will generate a trajectory file with a name beginning with `HISTORY` every `ntraj` time steps starting from timestep `straj`. Each binary file contains some of the system properties — including an identity number for each molecule — followed by the positions and velocities for each particle in the domain.

The `HISTORY*` files can be used with the utilities `traject` and `trajectselected` in the `DPD/utility` directory to produce plottable VMD files, with sets of bonded particles represented as residues. The utility `local` in the same directory can calculate localized properties (e.g. temperature, composition) from the same files and produces VTK files with these properties as cell data. Appendix B gives instructions for their use.

Statistical data file: `CORREL`

If the parameter `lcorr` is set to true, DL_MESO_DPD will generate an ANSI text file containing statistical data every `iscorr` time steps, which can later be imported into a spreadsheet or used by graph-plotting software. The formatting of the data varies depending on which kinds of interactions (bonds, angles, dihedrals, electrostatics) were used and whether a barostat was applied, based on the overall format (in a single line):

$$t \;\; E_{tot} \;\; E_{pot,tot} \;\; E_{pot,elec} \;\; E_{pot,bond} \;\; E_{pot,angle} \;\; E_{pot,dihed}$$
$$P \;\; \sigma_{xx} \;\; \sigma_{xy} \;\; \sigma_{xz} \;\; \sigma_{yx} \;\; \sigma_{yy} \;\; \sigma_{yz} \;\; \sigma_{zx} \;\; \sigma_{zy} \;\; \sigma_{zz} \;\; V \;\; T$$
$$\langle r_{bond} \rangle \;\; r_{bond,max} \;\; r_{bond,min} \;\; \langle \theta_{angle} \rangle \;\; \langle \phi_{dihed} \rangle$$

where t is the time, E energy (*tot* denoting total, *pot* potential), P pressure, σ_{ij} stress tensor, V volume, T temperature, $\langle r_{bond} \rangle$ the mean bond length, $r_{bond,max}$ and $r_{bond,min}$ the maximum and minimum bond lengths, $\langle \theta_{angle} \rangle$ the mean bond angle (in degrees) and $\langle \phi_{dihed} \rangle$ the mean bond dihedral (in degrees). Any property which does not vary or is not measured during the simulation, e.g. volume for NVT ensembles, is omitted from each line of data.

Chapter 13

DL_MESO_DPD Package Reference

13.1 Overview

DL_MESO_DPD consists of seventeen Fortran90 modules, which should be compiled in the following order prior to the main program itself:

- `constants`
  Contains the constants and parameters required by DL_MESO_DPD.
- `variables`
  Contains the globally available variables and arrays required by DL_MESO_DPD.
- `numeric_container`
  Contains random number generators and other general-purpose functions (e.g. scale sum, complementary error function).
- `comms_module`
  Contains all subroutines necessary for parallel computation.
- `error_module`
  Contains subroutines to print error messages and close down DL_MESO_DPD in a controlled manner.
- `parse_utils`
  Contains functions to read in text data from input files.
- `surface_module`
  Contains subroutines for applying boundary conditions at system planes, e.g. solid walls.
- `ewald_module`
  Contains subroutines for calculating forces due to electrostatic interactions using Ewald summation-based methods.
- `manybody_module`
  Contains subroutines for calculating density-dependent forces between particles, including the calculation of localized densities.
- `bond_module`
  Contains book-keeping and force calculation routines for bonds, angles and dihedrals.
- `domain_module`
  Contains subroutines to construct parallel link cells, import and deport particles in and export particle data to domain boundary halos.

- `start_module`
  Contains subroutines to initialize and restart DPD calculations.
- `config_module`
  Contains subroutines to read in input files with system and molecule/bond data, to zero all parameters and accumulators for statistical data and (for parallel version only) determine 3D domain decomposition.
- `field_module`
  Contains subroutines to calculate pairwise forces between particles.
- `integrate_module`
  Contains subroutines to integrate the equations of motion using the Velocity Verlet scheme and apply various thermostats and barostats.
- `statistics_module`
  Contains subroutines to calculate and write out statistical and trajectory data.
- `run_module`
  Contains program loops for different integrators and barostats.

Most of the above modules and the main program file `dlmesodpd.f90` are identical for both serial and parallel versions of DL_MESO_DPD. The filenames for the serial versions of `comms_module` and `domain_module` end with `_ser`, but are referred to in the code by their standard names.

The module for many-body DPD interactions, `manybody_module`, may be modified by users to incorporate alternative schemes. Additional bond, angle and dihedral models can be added to `bond_module` by the user, although modifications to `config_module` are also required to allow DL_MESO to recognise new types of bond interaction. For anything else, however, we recommend that DL_MESO users put any self-defined subroutines and functions into a module file called `user_module.f90` so future upgrades of DL_MESO will not interfere with their contributions.

13.2 DL_MESO_DPD Subroutines and Functions

13.2.1 Main program: dlmesodpd

Both the serial and parallel versions of the program operate in a similar way. Before DPD calculations start, the following tasks are carried out:

- For parallel running only, MPI is started up (`initcomms`) and the node properties are determined.
- An I/O channel for the general `OUTPUT` file is opened.
- The system clock is consulted for a start time (`timchk`).
- The starting banner for DL_MESO_DPD is printed.
- System and bond data are read in and initialized (`sysdef`).
- Initial values are set (`zero`).
- The starting configuration is set up (`start`), either from scratch or specified by the user using a `CONFIG` file.
- The system clock is consulted again for the start of the DPD calculation cycle (`timchk`).

A loop for DPD calculations is then called from `run_module` depending on the integrator and barostat selected by the user. Each step of DPD calculations involves the following:

- The step counter `nstep` is increased.
- The system clock is checked (`timchk`).
- The first stage of the required integrator is used to update the motion of the particles and their positions.
- Masses and particle/molecule names are reassigned to the particles.
- The parallel link-cell structure is set up and the pairwise forces calculated (`plcfor_*`).
- The time taken to calculate the forces is determined (`timchk`).
- The second stage of the integrator is applied to calculate the velocities at the end of the time step. This may include e.g. recalculation of dissipative forces or resizing of the system for a barostat.
- Statistical properties for the system are calculated and, during equilibration, the particle velocities are rescaled for the specified temperature (`statis`).
- After every `nsbpo` time steps, the system clock is consulted (`timchk`) and statistical data is printed to the `OUTPUT` file (`printout`). If equilibration has come to an end, i.e. `nstep = nseql`, this is also reported (`equilout`).
- If requested by the user, after every `iscorr` time steps statistical data is written to the `CORREL` file (`corout`) and after every `ntraj` time steps trajectory data is saved to the `HISTORY` or `HISTORY*` file(s).
- The step time is calculated (`timchk`) and if the allocated time has expired, the job is closed down; otherwise restart data is saved (`revive`).

After all the time steps have been calculated or the allocated time has elapsed:

- The final calculation summary is printed to the `OUTPUT` file (`result`).
- The duration of the calculation run is determined and printed (`timchk`).
- All remaining output channels (for `OUTPUT`, `HISTORY*` files) are closed.
- For parallel running only, MPI communications are closed down (`exitcomms`).

13.2.2 numeric_container

This package contains general purpose functions which may be replaced with any suitable functions in Fortran90 standard libraries, as well as bookkeeping subroutines for the global/local particle number list.

duni

- **Header records**
  `REAL(KIND=dp) FUNCTION duni (idnode)`
- **Function**
  Creates a double precision random number between 0 and 1.
- **Dependencies**
  None

- **Arguments**

`idnode`	input	integer
`duni`	output	real(KIND=dp)

- **Comments**
  The random number generator is an implementation of the Universal Random Number Generator[39]. The processor name `idnode` is used as a seed, which is activated the first time the function is called.

mtrnd

- **Header records**
  `REAL(KIND=dp) FUNCTION mtrnd (idnode)`
- **Function**
  Creates a double precision random number between 0 and 1.
- **Dependencies**
  None
- **Arguments**

`idnode`	input	integer
`mtrnd`	output	real(KIND=dp)

- **Comments**
  The random number generator is an implementation of the Mersenne Twister random number generator[42]. The processor name `idnode` is used as a seed, which is activated the first time the function is called.

gaussmp

- **Header records**
  `REAL(KIND=dp) FUNCTION gaussmp (idnode)`
- **Function**
  Creates a Gaussian random number.
- **Dependencies**
  `mtrnd`
- **Arguments**

`idnode`	input	integer
`gaussmp`	output	real(KIND=dp)

- **Comments**
  This is an implementation of the Marsaglia polar method[38] to convert linear random numbers (generated by the Mersenne Twister method) to Gaussian random variables with zero mean and unity variance.

sclsum

- **Header records**
  `REAL(KIND=dp) FUNCTION sclsum (n, a, i)`
- **Function**
  Calculates the scalar sum of an array.
- **Dependencies**
  None

- **Arguments**

`n`	input	integer
`a`	input	array of real(KIND=dp)
`i`	input	integer
`sclsum`	output	real(KIND=dp)

erfcdp

- **Header records**
  `REAL(KIND=dp) FUNCTION erfcdp (x)`
- **Function**
  Calculates the complementary error function for `x`, erfc(x).
- **Dependencies**
  None
- **Arguments**

`x`	input	real(KIND=dp)
`erfcdp`	output	real(KIND=dp)

- **Comments**
  This approximation for the function is based on a Chebyshev polynomial fitting[22].

erfdp

- **Header records**
  `REAL(KIND=dp) FUNCTION erfdp (x)`
- **Function**
  Calculates the error function for `x`, erf(x).
- **Dependencies**
  None
- **Arguments**

`x`	input	real(KIND=dp)
`erfdp`	output	real(KIND=dp)

- **Comments**
  This approximation for the function is based on a Chebyshev polynomial fitting[22].

images

- **Header records**
  `SUBROUTINE images (dx, dy, dz, lx, ly, lz, shearx, sheary, shearz, sldx, sldy, sldz)`
- **Function**
  Calculates the minimum distance between two particles in a periodic orthogonal box, adjusting for Lees-Edwards shear if necessary.
- **Dependencies**
  None

- **Arguments**

`dx`	input/output	real(KIND=dp)
`dy`	input/output	real(KIND=dp)
`dz`	input/output	real(KIND=dp)
`lx`	input	real(KIND=dp)
`ly`	input	real(KIND=dp)
`lz`	input	real(KIND=dp)
`shearx`	input	integer
`sheary`	input	integer
`shearz`	input	integer
`sldx`	input	real(KIND=dp)
`sldy`	input	real(KIND=dp)
`sldz`	input	real(KIND=dp)

13.2.3 comms_module

This module is essential for parallel running and does not require detailed knowledge for its use: depending on the version and implementation of MPI available, the user may wish to select between the lines `USE MPI` and `INCLUDE "mpif.h"` for loading the necessary routines. The serial version of the `comms_module` primarily consists of dummy routines to satisfy the required calls in the rest of the code.

initcomms

- **Header records**
  `SUBROUTINE initcomms ()`
- **Function**
  Starts Message Passing Interface (MPI).
- **Dependencies**
  None

exitcomms

- **Header records**
  `SUBROUTINE exitcomms ()`
- **Function**
  Closes Message Passing Interface (MPI) in a controlled manner.
- **Dependencies**
  None

abortcomms

- **Header records**
  `SUBROUTINE abortcomms ()`
- **Function**
  Terminates Message Passing Interface (MPI).
- **Dependencies**
  None

gsync

- **Header records**
  `SUBROUTINE gsync ()`
- **Function**
  Pauses running until all processes are synchronized.
- **Dependencies**
  None

global_and

- **Header records**
  `SUBROUTINE global_and (iii, nnn, nod, idnode)`
- **Function**
  Finds global logical AND from a Boolean array `iii()` of size `nnn`, placing the result on processor `nod`.
- **Dependencies**
  None
- **Arguments**

`iii`	input/output	array of logical
`nnn`	input	integer
`nod`	input	integer
`idnode`	input	integer

- **Comments**
  For Boolean scalars, the alternative `SUBROUTINE global_sca_and (iii, nod, idnode)` is available.

global_and_all

- **Header records**
  `SUBROUTINE global_and_all (iii, nnn)`
- **Function**
  Finds global logical AND from a Boolean array `iii()` of size `nnn` and broadcasts result to all processors.
- **Dependencies**
  None
- **Arguments**

`iii`	input/output	array of logical
`nnn`	input	integer

- **Comments**
  For Boolean scalars, the alternative `SUBROUTINE global_sca_and_all (iii)` is available.

global_sum_dble

- **Header records**
  `SUBROUTINE global_sum_dble (aaa, nnn)`
- **Function**
  Globally sums double precision array `aaa()` of size `nnn`.

- **Dependencies**
  None
- **Arguments**

`aaa`	input/output	array of real(KIND=dp)
`nnn`	input	integer

- **Comments**
  For double precision scalars, the alternative `SUBROUTINE global_sca_sum_dble (aaa)` is available.

global_sum_int

- **Header records**
  `SUBROUTINE global_sum_int (iii, nnn)`
- **Function**
  Globally sums integer array `iii()` of size `nnn`.
- **Dependencies**
  None
- **Arguments**

`iii`	input/output	array of integers
`nnn`	input	integer

- **Comments**
  For integer scalars, the alternative `SUBROUTINE global_sca_sum_int (iii)` is available.

global_sca_max_dble

- **Header records**
  `SUBROUTINE global_sca_max_dble (aaa)`
- **Function**
  Finds global maximum value of double precision number `aaa`.
- **Dependencies**
  None
- **Arguments**

`aaa`	input/output	array of real(KIND=dp)

global_sca_max_int

- **Header records**
  `SUBROUTINE global_sca_max_int (iii)`
- **Function**
  Finds global maximum value of integer `iii`.
- **Dependencies**
  None
- **Arguments**

`iii`	input/output	array of integers

global_sca_min_dble

- **Header records**
  `SUBROUTINE global_sca_min_dble (aaa)`
- **Function**
  Finds global minimum value of double precision number `aaa`.
- **Dependencies**
  None
- **Arguments**

`aaa`	input/output	array of real(KIND=dp)

global_sca_min_int

- **Header records**
  `SUBROUTINE global_sca_min_int (iii)`
- **Function**
  Finds global minimum value of integer `iii`.
- **Dependencies**
  None
- **Arguments**

`iii`	input/output	array of integers

msg_receive_blocked

- **Header records**
  `SUBROUTINE msg_receive_blocked (msgtag, buf, length)`
- **Function**
  In a blocking call, receives data in the form of a double precision array `buf()`.
- **Dependencies**
  None
- **Arguments**

`buf`	output	array of real(KIND=dp)
`msgtag`	input	integer
`length`	input	integer

- **Comments**
  For a double precision scalar, the alternative `SUBROUTINE msg_receive_sca_blocked (msgtag, buf, length)` is available.

msg_receive_unblocked

- **Header records**
  `INTEGER FUNCTION msg_receive_unblocked (msgtag, buf, length)`
- **Function**
  In a non-blocking call, receives data in the form of a double precision array `buf()`.

- **Dependencies**
  None

- **Arguments**

`buf`	output	array of real(KIND=dp)
`msgtag`	input	integer
`length`	input	integer
`msg_receive_unblocked`	output	integer

- **Comments**
  Since no scalars are received in non-blocking calls, no scalar version of this function exists.

msg_send_blocked

- **Header records**
  `SUBROUTINE msg_send_blocked (msgtag, buf, length, pe)`

- **Function**
  In a blocking call, send data from a double precision array `buf()`.

- **Dependencies**
  None

- **Arguments**

`buf`	input	array of real(KIND=dp)
`msgtag`	input	integer
`length`	input	integer
`pe`	input	integer

- **Comments**
  For a double precision scalar, the alternative `SUBROUTINE msg_send_sca_blocked (msgtag, buf, length, pe)` is available.

msg_wait

- **Header records**
  `SUBROUTINE msg_wait (request)`

- **Function**
  Causes process to wait for an unblocked message.

- **Dependencies**
  None

- **Arguments**

`request`	input	integer

mynode

- **Header records**
  `INTEGER FUNCTION mynode ()`

- **Function**
  Returns name of process.

- **Dependencies**
  None

- **Arguments**

`mynode`	output	integer

- **Comments**
  The serial version of this function returns 0.

numnodes

- **Header records**
  `INTEGER FUNCTION numnodes ()`

- **Function**
  Returns total number of processes available.

- **Dependencies**
  None

- **Arguments**

`numnodes`	output	integer

- **Comments**
  The serial version of this function returns 1.

timchk

- **Header records**
  `SUBROUTINE timchk(ktim, time)`

- **Function**
  Determines the time elapsed since the start of the calculation run and, if `ktim` > 0, prints the time to the `OUTPUT` file.

- **Dependencies**
  None

- **Arguments**

`ktim`	input	integer
`time`	output	real(KIND=dp)

- **Comments**
  The serial version of DL_MESO_DPD uses the generic `SYSTEM_CLOCK` call, while the parallel version uses `MPI_wtime`.

13.2.4 error_module

This module is used to print error messages and shut down DL_MESO_DPD in a controlled manner. It requires the modules `constants` and `comms_module` to be loaded beforehand.

error

- **Header records**
  `SUBROUTINE error (idnode, iode, value)`

- **Function**
  Prints user-friendly error message in OUTPUT file and closes down DL_MESO_DPD.

- **Dependencies**
  `abortcomms`

- **Arguments**

`idnode`	input	integer
`iode`	input	integer
`value`	input	integer

- **Comments**
  The error code `iode` closes down DL_MESO_DPD when positive; negative values can be used to print warning messages for non-fatal problems.

13.2.5 parse_utils

getword

- **Header records**
  `CHARACTER(LEN=mxword) FUNCTION getword (txt, n)`

- **Function**
  Obtains the `n`th word from a line of text `txt` separated by spaces or commas.

- **Dependencies**
  None

- **Arguments**

`txt`	input	character(LEN=*)
`n`	input	integer
`getword`	output	character(LEN=mxword)

parseint

- **Header records**
  `INTEGER(KIND=li) FUNCTION parseint (word)`

- **Function**
  Reads the integer contained in the string `word`.

- **Dependencies**
  None

- **Arguments**

`word`	input	character(LEN=*)
`parseint`	output	integer(KIND=li)

parsedble

- **Header records**
  `REAL(KIND=dp) FUNCTION parsedble (word)`
- **Function**
  Reads the double precision number contained in the string `word`.
- **Dependencies**
  None
- **Arguments**

`word`	input	character(LEN=*)
`parsedble`	output	real(KIND=dp)

getint

- **Header records**
  `INTEGER(KIND=li) FUNCTION getint (txt, n)`
- **Function**
  Reads the `n`th 'word' of the string `txt` to obtain an integer.
- **Dependencies**
  `parseint`
- **Arguments**

`txt`	input	character(LEN=*)
`n`	input	integer
`getint`	output	integer(KIND=li)

getdble

- **Header records**
  `REAL(KIND=dp) FUNCTION getdble (txt, n)`
- **Function**
  Reads the `n`th 'word' of the string `txt` to obtain a double precision number.
- **Dependencies**
  `parsedble`
- **Arguments**

`txt`	input	character(LEN=*)
`n`	input	integer
`getdble`	output	real(KIND=dp)

lowercase

- **Header records**
  `SUBROUTINE lowercase (word)`
- **Function**
  Changes all upper case letters in the string `word` to lower case.
- **Dependencies**
  None

- **Arguments**

`word`	input/output	character(LEN=*)
`n`	input	integer
`getdble`	output	real(KIND=dp)

13.2.6 surface_module

This module requires the `variables` module to be loaded beforehand.

surfacenodes

- **Header records**
  `SUBROUTINE surfacenodes`
- **Function**
  Identifies nodes containing surfaces or other boundary conditions.
- **Dependencies**
  None
- **Comments**
  If using Lees-Edwards boundary conditions, these are only applied after equilibration.

hardreflect

- **Header records**
  `SUBROUTINE hardreflect (k)`
- **Function**
  Applies boundary condition for hard reflecting walls: calculates short-range forces on particles when `k=1`, applies bounce-back condition when `k=2` for particles about to pass through boundary.
- **Dependencies**
  `surfacebounce`
- **Arguments**

`k`	input	integer

- **Comments**
  The boundary condition is given by [50]: the applied wall potential on each particle is given as

$$U_{wall,i}(z) = \frac{1}{2} A_{wall,i} \left(\frac{1-z}{z_c} \right)^2$$

  for $z < z_c$.

surfacebounce

- **Header records**
  `SUBROUTINE surfacebounce (ddd, vdd, sided)`
- **Function**
  Applies boundary condition on a leaving particle by means of specular reflection: move particle back into system and invert velocity component normal to wall.

- **Dependencies**
  None

- **Arguments**

`ddd`	input/output	real(KIND=dp)
`vdd`	input/output	real(KIND=dp)
`sided`	input	real(KIND=dp)

- **Comments**
  Momentum tangential to the boundary is preserved, i.e. this applies a free-slip boundary condition.

frozenbead

- **Header records**
  `SUBROUTINE frozenbead`

- **Function**
  Determines number of frozen particles required for boundary walls, given wall thickness and bead density, and adjusts system dimensions and particle counts to accommodate them.

- **Dependencies**
  None

- **Comments**
  This routine is only called for new simulations (with or without a `CONFIG` file).

shearslide

- **Header records**
  `SUBROUTINE shearslide`

- **Function**
  Determines displacement of shearing boundary for Lees-Edwards boundary conditions.

- **Dependencies**
  None

- **Comments**
  The boundary condition is given by [34]. Displacement of boundaries only takes place after equilibration.

13.2.7 ewald_module

This module requires the `constant`, `variables`, `numeric_container` and `comms_module` modules to be loaded beforehand.

ewald_real_slater

- **Header records**
  `SUBROUTINE ewald_real_slater (nlimit)`

- **Function**
  Calculates real-space terms for Ewald summation with Slater-type (exponential decay) charge distributions.

- **Dependencies**
  `erfcdp`
- **Arguments**

`nlimit`	input	integer

- **Comments**
  Calculates short-range Coulombic forces and potential energies for Ewald summation using the following smeared (non-point) charge distribution[14]:

$$f(r) = \frac{q\beta^3}{\pi r_c^3} \exp\left(-\frac{2r\beta}{r_c}\right)$$

ewald_reciprocal

- **Header records**
  `SUBROUTINE ewald_reciprocal`
- **Function**
  Calculates reciprocal-space terms for Ewald summation.
- **Dependencies**
  None
- **Comments**
  Calculates long-range Coulombic forces and potential energies using standard Ewald summation, including self-energy corrections for charged systems.

ewald_frozen_slater

- **Header records**
  `SUBROUTINE ewald_frozen_slater`
- **Function**
  Calculates corrective forces, potential energies, virials and stress tensors to remove electrostatic interactions between charged frozen particles in the Ewald summation.
- **Dependencies**
  `erfdp`
  `images`
- **Comments**
  This routine uses a replicated data strategy to determine all electrostatic interactions between charged frozen particles. If a simulation does not use a barostat, only one call prior to force calculations is required; it otherwise has to be called whenever the simulation volume changes.

13.2.8 manybody_module

This module requires pre-loading of the `constants` and `variables` modules.

local_density

- **Header records**
  `SUBROUTINE local_density`

- **Function**
  Calculates local densities for many-body DPD interactions.
- **Dependencies**
  `weight_rho`
- **Comments**
  Uses the parallel link-cell structure to calculate local densities for each component,
  $$\rho_i = \sum_{j \neq i} w^{\rho}(r_{ij})$$
  omitting self-contributions for each particle[66].

weight_rho

- **Header records**
  `REAL(KIND=dp) FUNCTION weight_rho (rrr)`
- **Function**
  Calculates normalized weight function for local densities.
- **Dependencies**
  None
- **Comments**
  The default weight function[68] is
  $$w^{\rho}(r) = \frac{15}{2\pi r_d^3}\left(1 - \frac{r}{r_d}\right)^2,$$
  which may be changed by the user.

manybody_force

- **Header records**
  `SUBROUTINE manybody_force (i, j, k, rrr, mbforce)`
- **Function**
  Calculates many-body DPD interaction force and non-density-dependent potential energies between particles `i` and `j` using parameter set `k` and inter-particle distance `rrr`.
- **Dependencies**
  None
- **Arguments**

`i`	input	integer
`j`	input	integer
`k`	input	integer
`rrr`	input	real(KIND=dp)
`mbforce`	output	real(KIND=dp)

- **Comments**
  The default form for the many-body DPD interaction force is a two-term style suitable for modelling vapour-liquid mixtures[68]:
  $$\vec{F}_{ij}^{C} = \left[A_{ij}\left(1 - \frac{r_{ij}}{r_c}\right) + B_{ij}(\rho_i + \rho_j)\left(1 - \frac{r_{ij}}{r_d}\right)\right]\frac{\vec{r}_{ij}}{r_{ij}}.$$
  This subroutine may be changed by users who wish to use different many-body interaction functional forms.

manybody_potential

- **Header records**
  `SUBROUTINE manybody_potential`
- **Function**
  Calculates the self-energy for every particle resulting from density-dependent (many-body) interactions.
- **Dependencies**
  None
- **Comments**
  The default form is based on the two-term vapour-liquid interactions used for many-body forces[68], although only the density-dependent potential energies are calculated using this routine (the rest are calculated in `manybody_force`). Users may wish to modify this routine to use their own many-body interaction models.

13.2.9 bond_module

This module requires the modules `constants`, `variables`, `error_module`, `comms_module` and `numeric_container` to be loaded beforehand.

shellsort_list

- **Header records**
  `SUBROUTINE shellsort_list`
- **Function**
  Reorders the global/local particle number list (in terms of global particle number) using a Shell sort.
- **Dependencies**
  None

search_list

- **Header records**
  `INTEGER FUNCTION search_list (aim)`
- **Function**
  Determines the index for the global/local particle number list for a specified global particle number `aim` using a binary search.
- **Dependencies**
  None
- **Arguments**

`aim`	input	integer
`search_list`	output	integer

- **Comments**
  This function returns a negative value if the global particle number cannot be found in the list. If it is not the only entry for the global particle number, the function returns the index plus the value of `nlist` (number of list items) to flag up duplicate entries.

duplicate

- **Header records**
  `SUBROUTINE duplicate (global1, global2, ind1, ind2)`
- **Function**
  Determines the indices out of duplicate entries from the global/local particle number list for a pair of specified global particle numbers, `global1` and `global2`, that produce the shortest distance between the particles.
- **Dependencies**
  None
- **Arguments**

`global1`	input	integer
`global2`	input	integer
`ind1`	input/output	integer
`ind2`	input/output	integer

contract_bndtbl

- **Header records**
  `SUBROUTINE contract_bndtbl`
- **Function**
  Strips out all bond pairs from bond table that have been reassigned to neighbouring processors.
- **Dependencies**
  None
- **Comments**
  Only called for parallel version of DL_MESO_DPD when bond tables include only local bonds in each processor.

contract_angtbl

- **Header records**
  `SUBROUTINE contract_angtbl`
- **Function**
  Strips out all bond angle triples from angle table that have been reassigned to neighbouring processors.
- **Dependencies**
  None
- **Comments**
  Only called for parallel version of DL_MESO_DPD when angle tables include only local angles in each processor.

contract_dhdtbl

- **Header records**
  `SUBROUTINE contract_dhdtbl`

- **Function**
  Strips out all bond dihedral quadruples from dihedral table that have been reassigned to neighbouring processors.

- **Dependencies**
  None

- **Comments**
  Only called for parallel version of DL_MESO_DPD when dihedral tables include only local dihedrals in each processor.

bond_force

- **Header records**
  `SUBROUTINE bond_force (bondtype, r, a, b, c, d, force, potential)`

- **Function**
  Determines the stretching force and potential energy between a pair of bonded particles.

- **Dependencies**
  None

- **Arguments**

`bondtype`	input	integer
`r`	input	real(KIND=dp)
`a`	input	real(KIND=dp)
`b`	input	real(KIND=dp)
`c`	input	real(KIND=dp)
`d`	input	real(KIND=dp)
`force`	output	real(KIND=dp)
`potential`	output	real(KIND=dp)

- **Comments**
  If required, the user can add extra stretching bond interaction types in this subroutine as additional cases: this would also require changes to `read_control` in `config_module`.

angle_force

- **Header records**
  `SUBROUTINE angle_force (angtype, theta, rab, rcb, a, b, c, d, force, potential, virial, dfab, dfcb)`

- **Function**
  Determines the bond angle force, potential energy and virial across a triple of bonded particles.

- **Dependencies**
  None

- **Arguments**

`angtype`	input	integer
`theta`	input	real(KIND=dp)
`rab`	input	real(KIND=dp)
`rcb`	input	real(KIND=dp)
`a`	input	real(KIND=dp)
`b`	input	real(KIND=dp)
`c`	input	real(KIND=dp)
`d`	input	real(KIND=dp)
`force`	output	real(KIND=dp)
`potential`	output	real(KIND=dp)
`virial`	output	real(KIND=dp)
`dfab`	output	real(KIND=dp)
`dfcb`	output	real(KIND=dp)

- **Comments**
  If required, the user can add extra bond angle interaction types in this subroutine as additional cases: this would also require changes to `read_control` in `config_module`.

dihedral_force

- **Header records**
  `SUBROUTINE dihedral_force (dhdtype, phi, pb, pc, a, b, c, d, force, potential)`
- **Function**
  Determines the bond dihedral force and potential energy across a quadruple of bonded particles.
- **Dependencies**
  None
- **Arguments**

`dhdtype`	input	integer
`phi`	input	real(KIND=dp)
`pb`	input	real(KIND=dp)
`pc`	input	real(KIND=dp)
`a`	input	real(KIND=dp)
`b`	input	real(KIND=dp)
`c`	input	real(KIND=dp)
`d`	input	real(KIND=dp)
`force`	output	real(KIND=dp)
`potential`	output	real(KIND=dp)

- **Comments**
  If required, the user can add extra bond dihedral (and improper dihedral) interaction types in this subroutine as additional cases: this would also require changes to `read_control` in `config_module`.

bondforceslocal

- **Header records**
  `SUBROUTINE bondforceslocal`
- **Function**
  Calculates all bond (stretching, angle, dihedral) forces between particles in system using locally-defined bond lists.

- **Dependencies**
  `shellsort_list`
  `search_list`
  `duplicate`
  `error`
  `global_sca_and`
  `bond_force`
  `angle_force`
  `dihedral_force`
- **Comments**
  Assumes that bond lists contain only those bonds in the current node. This is the most efficient method for parallel running but runs the risk of losing track of bonded pairs if bond lengths get longer than the subdomain halo size `rhalo`.

bondforcesglobal

- **Header records**
  `SUBROUTINE bondforcesglobal`
- **Function**
  Calculates all bond (stretching, angle, dihedral) forces between particles in system using globally-defined bond lists.
- **Dependencies**
  `shellsort_list`
  `search_list`
  `images`
  `error`
  `global_sum_dble`
  `global_sca_and`
  `bond_force`
  `angle_force`
  `dihedral_force`
- **Comments**
  Assumes that bond lists contain all bonds in the entire system. This is less efficient for parallel running than only calculating the bonds in each node but ensures longer bond lengths can be accommodated.

13.2.10 domain_module

This module requires the modules `constants`, `variables`, `error_module`, `comms_module` (if running parallel version) and `bond_module` (if running parallel version) to be loaded beforehand.

domain_decompose

- **Header records**
  `SUBROUTINE domain_decompose`
- **Function**
  Determines 3D domain decomposition for system: number of nodes in each direction, location for each node in system, nearest neighbouring nodes.

- **Dependencies** (parallel version only)
  `msg_receive_blocked`
  `msg_receive_sca_blocked`
  `msg_send_blocked`
  `msg_send_sca_blocked`
- **Comments**
  Essential for parallel version of DL_MESO_DPD; the serial version of this subroutine sets values for a single processor.

domain_dimensions

- **Header records**
  `SUBROUTINE domain_dimensions`
- **Function**
  Determines dimensions of domain and link cells
- **Dependencies**
  None
- **Comments**
  This subroutine is called during setup and, if a barostat is used, when the volume of the system changes.

parlnk

- **Header records**
  `SUBROUTINE parlnk (num1, num2)`
- **Function**
  Constructs the parallel link cells for calculations of pairwise forces between particles.
- **Dependencies**
  None
- **Arguments**

`num1`	input	integer
`num2`	input	integer

- **Comments**
  Pairwise electrostatic forces typically act over longer lengthscales between charged particles only and are not considered using this subroutine: a similar routine for the real-space part of Ewald summation is included in `ewald_real`.

deport

- **Header records**
  `SUBROUTINE deport (nlimit, mdir, mp, begin, final, shove, skip)`
- **Function**
  Deports particles from boundary halo to neighbouring domain.
- **Dependencies**
  `contract_bndtbl`
  `contract_angtbl`

`contract_dhdtbl`
`error`
`global_sca_and_all`
`msg_receive_unblocked`
`msg_receive_sca_blocked`
`msg_send_blocked`
`msg_send_sca_blocked`
`msg_wait`

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`mp`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)
`skip`	input	logical

- **Comments**
  Only exists in parallel version of DL_MESO_DPD. The switch `skip` prevents particles in the boundary halo of the current domain from being transferred: this is useful for applying non-periodic boundary conditions (e.g. hard surfaces, Lees-Edwards shearing).

deport_shear

- **Header records**
  `SUBROUTINE deport_shear (nlimit, mdir, begin, final, shove, shove1, shove2, vshove1, vshove2, side1, side2)`

- **Function**
  Deports particles from boundary halo to appropriate domains for Lees-Edwards shearing.

- **Dependencies**
  `contract_bndtbl`
  `contract_angtbl`
  `contract_dhdtbl`
  `error`
  `msg_receive_unblocked`
  `msg_receive_sca_blocked`
  `msg_send_blocked`
  `msg_send_sca_blocked`
  `msg_wait`

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)
`shove1`	input	real(KIND=dp)
`shove2`	input	real(KIND=dp)
`vshove1`	input	real(KIND=dp)
`vshove2`	input	real(KIND=dp)
`side1`	input	real(KIND=dp)
`side2`	input	real(KIND=dp)

- **Comments**
  Only exists in parallel version of DL_MESO_DPD. This routine requires information for the two dimensions along the surface of the shearing boundary: the variables `shove1` and `shove2` represent the displacement due to shear, `side1` and `side2` are the domain dimensions and `vshove1` and `vshove2` are the velocity corrections to particles passing through the shearing boundary.

import

- **Header records**
  `SUBROUTINE import (nlimit, mdir, mp, begin, final, shove, skip)`

- **Function**
  Imports particle forces from boundary halo for integration schemes and thermostats that only require one set of forces to be calculated.

- **Dependencies**
  `error`
  `global_sca_and_all`
  `msg_receive_unblocked`
  `msg_receive_sca_blocked`
  `msg_send_blocked`
  `msg_send_sca_blocked`
  `msg_wait`

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`mp`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)
`skip`	input	logical

- **Comments**
  Only exists in parallel version of DL_MESO_DPD. This routine is suitable for thermostats using the standard Velocity Verlet algorithm for force integration and one set of forces: this includes the DPD thermostat with MD integration (MD-VV), the Lowe-Andersen and Peters thermostats. The switch `skip` prevents the importing of forces from the boundary halo of the current domain: this is useful for applying non-periodic boundary conditions (e.g. hard surfaces, Lees-Edwards shearing).

import_shear

- **Header records**
  `SUBROUTINE import_shear (nlimit, mdir, begin, final, shove, shove1, shove2, side1, side2)`
- **Function**
  Imports particle forces from boundary halos for integration schemes and thermostats that only require one set of forces to be calculated and Lees-Edwards shearing.
- **Dependencies**
  `error`
  `msg_receive_unblocked`
  `msg_receive_sca_blocked`
  `msg_send_blocked`
  `msg_send_sca_blocked`
  `msg_wait`
- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)
`shove1`	input	real(KIND=dp)
`shove2`	input	real(KIND=dp)
`side1`	input	real(KIND=dp)
`side2`	input	real(KIND=dp)

- **Comments**
  Only exists in parallel version of DL_MESO_DPD. This routine is suitable for thermostats using the standard Velocity Verlet algorithm for force integration and one set of forces: this includes the DPD thermostat with MD integration (MD-VV), the Lowe-Andersen and Peters thermostats. This routine requires information for the two dimensions along the surface of the shearing boundary: the variables `shove1` and `shove2` represent the displacement due to shear, while `side1` and `side2` are the domain dimensions.

importvariable

- **Header records**
  `SUBROUTINE importvariable (nlimit, mdir, mp, begin, final, shove, first, skip)`
- **Function**
  Imports particle forces from boundary halo for integration schemes and thermostats that require two separate sets of forces to be calculated.
- **Dependencies**
  `error`
  `global_sca_and_all`
  `msg_receive_unblocked`
  `msg_receive_sca_blocked`
  `msg_send_blocked`
  `msg_send_sca_blocked`
  `msg_wait`

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`mp`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)
`first`	input	logical
`skip`	input	logical

- **Comments**
Only exists in parallel version of DL_MESO_DPD. Setting `first` to `.true.` imports both sets of forces (constant and variable), while setting to `.false.` imports just the variable forces. This routine is suitable for thermostats where two types of forces need to be kept separate and/or recalculated, i.e. the DPD thermostat with DPD Velocity Verlet integration (DPD-VV) and Stoyanov-Groot thermostat. The switch `skip` ignores any particles in halos for non-periodic boundary conditions (e.g. hard surfaces, Lees-Edwards shearing).

importvariable_shear

- **Header records**
`SUBROUTINE importvariable_shear (nlimit, mdir, begin, final, shove, shove1, shove2, side1, side2)`

- **Function**
Imports particle forces from boundary halos for integration schemes and thermostats that require two separate sets of forces to be calculated and Lees-Edwards shearing.

- **Dependencies**
`error`
`msg_receive_unblocked`
`msg_receive_sca_blocked`
`msg_send_blocked`
`msg_send_sca_blocked`
`msg_wait`

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)
`shove1`	input	real(KIND=dp)
`shove2`	input	real(KIND=dp)
`side1`	input	real(KIND=dp)
`side2`	input	real(KIND=dp)

- **Comments**
Only exists in parallel version of DL_MESO_DPD. Setting `first` to `.true.` imports both sets of forces (constant and variable), while setting to `.false.` imports just the variable forces. This routine is suitable for thermostats where two types of forces need to be kept separate and/or recalculated, i.e. the DPD thermostat with DPD Velocity Verlet integration (DPD-VV) and Stoyanov-Groot thermostat. This routine requires information for the two dimensions along the surface of the shearing boundary: the variables

`shove1` and `shove2` represent the displacement due to shear, while `side1` and `side2` are the domain dimensions.

importdensity

- **Header records**
  `SUBROUTINE importdensity (nlimit, mdir, mp, begin, final)`

- **Function**
  Imports local densities from boundary halo.

- **Dependencies**
  `error`
  `global_sca_and_all`
  `msg_receive_unblocked`
  `msg_receive_sca_blocked`
  `msg_send_blocked`
  `msg_send_sca_blocked`
  `msg_wait`

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`mp`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)

- **Comments**
  Only exists in parallel version of DL_MESO_DPD.

importdensity_shear

- **Header records**
  `SUBROUTINE importdensity_shear (nlimit, mdir, begin, final, shove, shove1, shove2, side1, side2)`

- **Function**
  Imports local densities from boundary halos for Lees-Edwards shearing.

- **Dependencies**
  `error`
  `msg_receive_unblocked`
  `msg_receive_sca_blocked`
  `msg_send_blocked`
  `msg_send_sca_blocked`
  `msg_wait`

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)
`shove1`	input	real(KIND=dp)
`shove2`	input	real(KIND=dp)
`side1`	input	real(KIND=dp)
`side2`	input	real(KIND=dp)

- **Comments**
  Only exists in parallel version of DL_MESO_DPD. This routine requires information for the two dimensions along the surface of the shearing boundary: the variables `shove1` and `shove2` represent the displacement due to shear, while `side1` and `side2` are the domain dimensions.

export

- **Header records**
  `SUBROUTINE export (nlimit, mdir, mp, begin, final, shove, skip)` (Parallel version)
  `SUBROUTINE export (nlimit, mdir, begin, final, shove)` (Serial version)

- **Function**
  Exports particle data (positions, velocity) to neighbouring domain as boundary halo.

- **Dependencies**
  `error`
  `global_sca_and_all` (Parallel version only)
  `msg_receive_unblocked` (Parallel version only)
  `msg_receive_sca_blocked` (Parallel version only)
  `msg_send_blocked` (Parallel version only)
  `msg_send_sca_blocked` (Parallel version only)
  `msg_wait` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`mp`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)
`skip`	input	logical

- **Comments**
  The switch `skip` (in the parallel version only) prevents particles in the boundary halo of the current domain from being transferred for non-periodic boundary conditions (e.g. hard surfaces, Lees-Edwards shearing).

export_shear

- **Header records**
  `SUBROUTINE export_shear (nlimit, mdir, begin, final, shove, shove1, shove2, side1, side2)`

- **Function**
  Exports particle data (positions, velocity) to appropriate domains as boundary halos for Lees-Edwards shearing.

- **Dependencies**
  `error`
  `msg_receive_unblocked` (Parallel version only)
  `msg_receive_sca_blocked` (Parallel version only)
  `msg_send_blocked` (Parallel version only)
  `msg_send_sca_blocked` (Parallel version only)
  `msg_wait` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)
`shove1`	input	real(KIND=dp)
`shove2`	input	real(KIND=dp)
`side1`	input	real(KIND=dp)
`side2`	input	real(KIND=dp)

- **Comments**
  This routine requires information for the two dimensions along the surface of the shearing boundary: the variables `shove1` and `shove2` represent the displacement due to shear, while `side1` and `side2` are the domain dimensions.

exportvelocity

- **Header records**
  `SUBROUTINE exportvelocity (nlimit, mdir, mp, begin, final, shove, skip)` (Parallel version)
  `SUBROUTINE exportvelocity (nlimit, mdir, begin, final)` (Serial version)

- **Function**
  Exports particle velocities to neighbouring domain as boundary halo for recalculation of dissipative forces (as required for DPD Velocity Verlet algorithm).

- **Dependencies**
  `error`
  `global_sca_and_all` (Parallel version only)
  `msg_receive_unblocked` (Parallel version only)
  `msg_receive_sca_blocked` (Parallel version only)
  `msg_send_blocked` (Parallel version only)
  `msg_send_sca_blocked` (Parallel version only)
  `msg_wait` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`mp`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)
`skip`	input	logical

- **Comments**
  In serial running, this routine can automatically deal with non-periodic boundary conditions. The switch `skip` (in the parallel version only) prevents particles in the boundary halo of the current domain from being transferred for non-periodic boundary conditions (e.g. hard surfaces, Lees-Edwards shearing).

exportvelocity_shear

- **Header records**
  `SUBROUTINE exportvelocity_shear (nlimit, mdir, begin, final, shove, shove1, shove2, side1, side2)`
- **Function**
  Exports particle velocities to appropriate domains as boundary halo for recalculation of dissipative forces (as required for DPD Velocity Verlet algorithm) with Lees-Edwards shearing.
- **Dependencies**
  `error`
  `msg_receive_unblocked`
  `msg_receive_sca_blocked`
  `msg_send_blocked`
  `msg_send_sca_blocked`
  `msg_wait`
- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)
`shove1`	input	real(KIND=dp)
`shove2`	input	real(KIND=dp)
`side1`	input	real(KIND=dp)
`side2`	input	real(KIND=dp)

- **Comments**
  Only exists in parallel version of DL_MESO_DPD. This routine requires information for the two dimensions along the surface of the shearing boundary: the variables `shove1` and `shove2` represent the displacement due to shear, while `side1` and `side2` are the domain dimensions.

exportdensity

- **Header records**
  `SUBROUTINE exportdensity (nlimit, mdir, mp, begin, final, shove)` (Parallel version)
  `SUBROUTINE exportdensity (nlimit, mdir, begin, final)` (Serial version)

- **Function**
  Exports particle data (local densities) to neighbouring domain as boundary halo for calculation of many-body DPD interaction forces.

- **Dependencies**
  `error`
  `global_sca_and_all` (Parallel version only)
  `msg_receive_unblocked` (Parallel version only)
  `msg_receive_sca_blocked` (Parallel version only)
  `msg_send_blocked` (Parallel version only)
  `msg_send_sca_blocked` (Parallel version only)
  `msg_wait` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`mp`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)
`skip`	input	logical

- **Comments**
  In serial running, this routine can automatically deal with non-periodic boundary conditions. The switch `skip` (in the parallel version only) prevents particles in the boundary halo of the current domain from being transferred for non-periodic boundary conditions (e.g. hard surfaces, Lees-Edwards shearing).

exportdensity_shear

- **Header records**
  `SUBROUTINE exportdensity_shear (nlimit, mdir, mp, begin, final, shove, skip)` (Parallel version)
  `SUBROUTINE exportdensity (nlimit, mdir, begin, final)` (Serial version)

- **Function**
  Exports particle data (local densities) to neighbouring domain as boundary halo for calculation of many-body DPD interaction forces.

- **Dependencies**
  `error`
  `msg_receive_unblocked`
  `msg_receive_sca_blocked`
  `msg_send_blocked`
  `msg_send_sca_blocked`
  `msg_wait`

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)
`shove1`	input	real(KIND=dp)
`shove2`	input	real(KIND=dp)
`side1`	input	real(KIND=dp)
`side2`	input	real(KIND=dp)

- **Comments**
  Only exists in parallel version of DL_MESO_DPD. This routine requires information for the two dimensions along the surface of the shearing boundary: the variables `shove1` and `shove2` represent the displacement due to shear, while `side1` and `side2` are the domain dimensions.

deportdata

- **Header records**
  `SUBROUTINE deportdata (nlimit)`
- **Function**
  Applies deport of particles from boundary halos to neighbouring domains and/or periodic boundary conditions.
- **Dependencies**
  `deport` (Parallel version only)
- **Arguments**

`nlimit`	input/output	integer

deportdata_shear

- **Header records**
  `SUBROUTINE deportdata_shear (nlimit)`
- **Function**
  Applies deport of particles from boundary halos to neighbouring domains and/or periodic boundary conditions with Lees-Edwards shearing.
- **Dependencies**
  `deport` (Parallel version only)
  `deport_shear` (Parallel version only)
- **Arguments**

`nlimit`	input/output	integer

importdata

- **Header records**
  `SUBROUTINE importdata (nlimit)`
- **Function**
  Applies import of particle forces from boundary halos of neighbouring domains and/or across periodic boundary conditions.

- **Dependencies**
  `import` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer

- **Comments**
  This subroutine is applicable for integrators/thermostats using single sets of particle forces (i.e. standard Velocity Verlet, Lowe-Andersen, Peters).

importdata_shear

- **Header records**
  `SUBROUTINE importdata_shear (nlimit)`

- **Function**
  Applies import of particle forces from boundary halos of neighbouring domains and/or across periodic boundary conditions with Lees-Edwards shearing.

- **Dependencies**
  `import` (Parallel version only)
  `import_shear` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer

- **Comments**
  This subroutine is applicable for integrators/thermostats using single sets of particle forces (i.e. standard Velocity Verlet, Lowe-Andersen, Peters).

importdata_dpdvv1

- **Header records**
  `SUBROUTINE importdata_dpdvv1 (nlimit)`

- **Function**
  Applies import of particle forces (constant and variable) from boundary halos of neighbouring domains and/or across periodic boundary conditions for the DPD Velocity Verlet algorithm.

- **Dependencies**
  `importvariable` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer

importdata_dpdvv1_shear

- **Header records**
  `SUBROUTINE importdata_dpdvv1_shear (nlimit)`

- **Function**
  Applies import of particle forces (constant and variable) from boundary halos of neighbouring domains and/or across periodic boundary conditions for the DPD Velocity Verlet algorithm with Lees-Edwards shearing.

- **Dependencies**
  `importvariable` (Parallel version only)
  `importvariable_shear` (Parallel version only)
- **Arguments**

`nlimit`	input/output	integer

importdata_dpdvv2

- **Header records**
  `SUBROUTINE importdata_dpdvv2 (nlimit)`
- **Function**
  Applies import of variable particle forces from boundary halos of neighbouring domains and/or across periodic boundary conditions for the DPD Velocity Verlet algorithm.
- **Dependencies**
  `importvariable` (Parallel version only)
- **Arguments**

`nlimit`	input/output	integer

importdata_dpdvv2_shear

- **Header records**
  `SUBROUTINE importdata_dpdvv2_shear (nlimit)`
- **Function**
  Applies import of variable particle forces from boundary halos of neighbouring domains and/or across periodic boundary conditions for the DPD Velocity Verlet algorithm with Lees-Edwards shearing.
- **Dependencies**
  `importvariable` (Parallel version only)
  `importvariable_shear` (Parallel version only)
- **Arguments**

`nlimit`	input/output	integer

importdata_stoyanov

- **Header records**
  `SUBROUTINE importdata_stoyanov (nlimit)`
- **Function**
  Applies import of particle forces (constant and variable) from boundary halos of neighbouring domains and/or across periodic boundary conditions for the Stoyanov-Groot thermostat.
- **Dependencies**
  `importvariable` (Parallel version only)
- **Arguments**

`nlimit`	input/output	integer

importdata_stoyanov_shear

- **Header records**
  `SUBROUTINE importdata_stoyanov_shear (nlimit)`

- **Function**
  Applies import of particle forces (constant and variable) from boundary halos of neighbouring domains and/or across periodic boundary conditions for the Stoyanov-Groot thermostat with Lees-Edwards shearing.

- **Dependencies**
  `importvariable` (Parallel version only)
  `importvariable_shear` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer

importdensitydata

- **Header records**
  `SUBROUTINE importdensitydata (nlimit)`

- **Function**
  Applies import of local densities for particles from boundary halos of neighbouring domains and/or across periodic boundary conditions.

- **Dependencies**
  `importdensity` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer

- **Comments**
  This subroutine is essential for DPD calculations involving many-body (density dependent) interactions.

importdensitydata_shear

- **Header records**
  `SUBROUTINE importdensitydata_shear (nlimit)`

- **Function**
  Applies import of local densities for particles from boundary halos of neighbouring domains and/or across periodic boundary conditions with Lees-Edwards shearing.

- **Dependencies**
  `importdensity` (Parallel version only)
  `importdensity_shear` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer

- **Comments**
  This subroutine is essential for DPD calculations involving many-body (density dependent) interactions.

exportdata

- **Header records**
  `SUBROUTINE exportdata (nlimit)`
- **Function**
  Applies export of particle properties (positions, velocities) to boundary halos of neighbouring domains and/or across periodic boundary conditions.
- **Dependencies**
  `export`
- **Arguments**

`nlimit`	input/output	integer

- **Comments**
  Non-periodic boundaries either do not require boundary halos or require them to be constructed differently: these are thus excluded from the export of particle properties.

exportdata_shear

- **Header records**
  `SUBROUTINE exportdata_shear (nlimit)`
- **Function**
  Applies export of particle properties (positions, velocities) to boundary halos of neighbouring domains and/or across periodic boundary conditions with Lees-Edwards shearing.
- **Dependencies**
  `export`
  `export_shear`
- **Arguments**

`nlimit`	input/output	integer

exportvelocitydata

- **Header records**
  `SUBROUTINE exportvelocitydata (nlimit)`
- **Function**
  Applies export of particle velocities to boundary halos of neighbouring domains and/or across periodic boundary conditions for recalculation of dissipative forces.
- **Dependencies**
  `exportvelocity`
- **Arguments**

`nlimit`	input/output	integer

- **Comments**
  Non-periodic boundaries either do not require boundary halos or require them to be constructed differently: these are thus excluded from the export of particle velocities.

exportvelocitydata_shear

- **Header records**
  `SUBROUTINE exportvelocitydata_shear (nlimit)`
- **Function**
  Applies export of particle velocities to boundary halos of neighbouring domains and/or across periodic boundary conditions with Lees-Edwards shearing for recalculation of dissipative forces.
- **Dependencies**
  `exportvelocity`
  `exportvelocity_shear` (Parallel version only)
- **Arguments**

`nlimit`	input/output	integer

exportdensitydata

- **Header records**
  `SUBROUTINE exportdensitydata (nlimit)`
- **Function**
  Applies export of local densities for particles to boundary halos of neighbouring domains and/or across periodic boundary conditions for calculation of many-body DPD interaction forces.
- **Dependencies**
  `exportdensity`
- **Arguments**

`nlimit`	input/output	integer

- **Comments**
  Non-periodic boundaries do not require boundary halos: these are excluded from the export of particle local densities.

exportdensitydata_shear

- **Header records**
  `SUBROUTINE exportdensitydata_shear (nlimit)`
- **Function**
  Applies export of local densities for particles to boundary halos of neighbouring domains and/or across periodic boundary conditions with Lees-Edwards shearing for calculation of many-body DPD interaction forces.
- **Dependencies**
  `exportdensity`
  `exportdensity_shear` (Parallel version only)
- **Arguments**

`nlimit`	input/output	integer

13.2.11 start_module

This module requires the modules `constants`, `variables`, `error_module`, `comms_module`, `numeric_container`, `parse_utils`, `ewald_module` and `surface_module` to be loaded beforehand.

start

- **Header records**
  `SUBROUTINE start`

- **Function**
  Sets up starting configuration for DPD calculations, depending on availablity of `CONFIG` or restart files.

- **Dependencies**
  `initialize`
  `revive`
  `read_config`
  `initialvelocity`
  `ewald_frozen_slater`

initialize

- **Header records**
  `SUBROUTINE initialize`

- **Function**
  Sets up starting configuration for DPD calculations without given initial or restart configuration.

- **Dependencies**
  `global_sca_and`
  `error`
  `duni`
  `mtrnd`
  `global_sum_int`

- **Comments**
  This routine assigns frozen bead walls and other unbonded beads as cubic lattices: the latter may be incomplete and species of unbonded beads are randomly assigned according to the numbers specified in the `FIELD` file. No duplication of system using `nfold` is assumed.

revive

- **Header records**
  `SUBROUTINE revive (key)`

- **Function**
  Saves and reads restart configurations.

- **Dependencies**
  `export*` files (if reading restart configuration)

- **Arguments**

`key`	input	integer

- **Comments**
  No duplication of system using `nfold` is assumed when reading restart files: if this is required, a `CONFIG` file should be created and used instead.

read_config

- **Header records**
  `SUBROUTINE read_config`
- **Function**
  Reads initial system configuration (positions, velocities, forces) from DL_POLY-style `CONFIG` file and assigns particles, bonds etc. to system, accounting for system duplication using `nfold`.
- **Dependencies**
  `CONFIG` file
  `getword`
  `getint`
  `getdble`
  `initialvelocity`
  `global_sca_and`
  `global_sum_sca_int`
  `error`
- **Comments**
  The number of particles in the CONFIG file must match up with the number described in the corresponding `FIELD` file, as should any molecule and bond information. The periodic boundary key `imcon`[65] is ignored, since all DL_MESO systems are orthorhombic. This routine also adds any required frozen bead walls as cubic lattices. No checks are made to ensure molecules do not cross non-periodic boundaries!

initialvelocity

- **Header records**
  `SUBROUTINE initialvelocity`
- **Function**
  Initializes particle velocities randomly in system to give required system temperature.
- **Dependencies**
  `duni`
  `global_sum_dble`

sort_beads

- **Header records**
  `SUBROUTINE sort_beads`
- **Function**
  Re-orders local identity numbers of beads in current processor to place frozen beads at beginning of list.
- **Comments**
  Frozen beads are moved to local bead numbers between 1 and `nfbeads`, while non-frozen beads are moved to local bead numbers between `nfbeads+1` and `nbeads`.

13.2.12 config_module

This module requires the modules `constants`, `variables`, `comms_module`, `numeric_container`, `parse_utils`, `domain_module` and `surface_module` to be loaded beforehand.

sysdef

- **Header records**
  `SUBROUTINE sysdef`
- **Function**
  Reads in system data, determines simulation properties, prints to `OUTPUT` file and allocates main arrays for calculations.
- **Dependencies**
  `scan_config`
  `read_control`
  `scan_field`
  `read_field`
  `frozenbead`
  `domain_decompose`
  `domain_dimensions`
  `surfacenodes`
  `elecgen`
  `error`
- **Arguments**
  None
- **Comments**
  The routine calculates a maximum size for the main arrays (`maxdim`) and a maximum number of pairwise interactions (`maxpair`) based on the total number of particles, the number of link cells and number of nodes.

scan_config

- **Header records**
  `SUBROUTINE scan_config`
- **Function**
  Scans `CONFIG` file for dimensions of system unit cell.
- **Dependencies**
  `CONFIG` file
  `getint`
  `getdble`
  `global_sca_and`
  `error`
- **Arguments**
  None

read_control

- **Header records**
  `SUBROUTINE read_control`
- **Function**
  Reads in system data from `CONTROL` file.

- **Dependencies**
  `CONTROL` file
  `getdble`
  `getint`
  `getword`
  `lowercase`
  `error`
  `global_sca_and`
- **Arguments**
  None

scan_field

- **Header records**
  `SUBROUTINE scan_field`
- **Function**
  Scans `FIELD` file for numbers of interactions, species etc. and sets up arrays for all interaction data (including bonds)
- **Dependencies**
  `FIELD` file
  `error`
  `global_sca_and`
  `getword`
  `lowercase`
  `getint`
  `getdble`
- **Arguments**
  None

read_field

- **Header records**
  `SUBROUTINE read_field`
- **Function**
  Reads in all species and interactions from `FIELD` file, including bonded interactions and molecule configurations
- **Dependencies**
  `FIELD` file
  `getdble`
  `getint`
  `getword`
  `lowercase`
  `error`
  `global_sca_and`
- **Arguments**
  None

- **Comments**
  This routine will determine interaction parameters for species pairs not specified in `FIELD` file *unless* any interactions are many-body DPD (this requires *all* interaction species pairs to be specified).

elecgen

- **Header records**
  `SUBROUTINE elecgen`
- **Function**
  Sets up electrostatic parameters for self-interaction and charged system corrections.
- **Dependencies**
  `error`
  `global_sca_and`

zero

- **Header records**
  `SUBROUTINE zero`
- **Function**
  Sets step counters, initial time parameters, system parameters, accumulators for statistical properties and long-range potential corrections to zero, initializes random number generators.
- **Dependencies**
  `duni`
  `mtrnd`

13.2.13 field_module

This module requires the modules `constants`, `variables`, `bond_module`, `manybody_module`, `ewald_module`, `surface_module`, `numeric_container` and `domain_module` to be loaded beforehand.

Different versions of each subroutine are available in this module for different integrators and/or thermostats:

- DPD thermostat with standard (molecular dynamics) Velocity Verlet integration (MD-VV)[67] (`mdvv`)
- DPD thermostat with DPD Velocity Verlet integration (DPD-VV)[3] (`dpdvv`)
- Lowe-Andersen thermostat[36] (`lowe`)
- Peters thermostat[47] (`peters`)
- Stoyanov-Groot thermostat[62] (`stoyanov`)

forces_*

- **Header records**
  `SUBROUTINE forces_* (nlimit)`
- **Function**
  Calculates all forces between particles, particularly pairwise forces within cut-off radius.

- **Dependencies**
  `mtrnd` (`duni`, `gaussmp`)
  `manybody_force`
  `manybody_potential`
  `ewald_real`
  `ewald_reciprocal`
  `bondforcesglobal`
  `bondforceslocal`
  `hardreflect`

- **Arguments**

`nlimit`	input	integer

- **Comments**
  DPD random forces are calculated using a uniform random number generator[16] (by default the Mersenne twister generator `mtrnd`), i.e.
  $$\zeta_{ij} \approx \sqrt{12}\,(u - 0.5)\,,$$
  which produces statistically similar results[10] and is computationally more efficient than using the Gaussian random number subroutine `gaussmp`, although this or the `duni` random number generator may be substituted if required. For systems involving Lees-Edwards boundary conditions, thermostats are not applied between particle pairs that cross shearing boundaries[5].

dragforces_dpdvv

- **Header records**
  `SUBROUTINE dragforces_dpdvv (nlimit)`

- **Function**
  Recalculates dissipative forces between particles within cut-off radius.

- **Dependencies**
  None.

- **Arguments**

`nlimit`	input	integer
`idnode`	input	integer

- **Comments**
  Only required for DPD Velocity Verlet (DPD-VV) integration. Particle pairs that cross Lees-Edwards shearing boundaries are omitted[5].

plcfor_*

- **Header records**
  `SUBROUTINE plcfor_*`

- **Function**
  Sets up parallel link-cells structure and calculates forces (including those involving particles in boundary halo).

- **Dependencies**
  `exportdata`
  `parlnk`
  `local_density`

`importdensitydata`
`exportdensitydata`
`forces_*`
`importdata` (`importdata_dpdvv1`, `importdata_dpdvv2`, `importdata_stoyanov`)
`freeze_beads`

freeze_beads

- **Header records**
  `SUBROUTINE freeze_beads`
- **Function**
  Quenches forces and velocities of frozen particles by resetting them to zero.
- **Dependencies**
  None.

13.2.14 integrate_module

This module requires preloading of the `constants`, `variables`, `domain_module`, `surface_module`, `comms_module`, `error_module`, `numeric_container` and `field_module` modules. Like the `field_module`, different versions of the subroutine (`mdvv`, `dpdvv`, `lowe`, `peters`, `stoyanov`) are available for each thermostat/integrator.

verlet_*

- **Header records**
  `SUBROUTINE verlet_* (stage)`
- **Function**
  Solves the equations of motion using the Velocity Verlet scheme[67].
- **Dependencies**
  `hardreflect`
  `deportdata`
  `error`
  `exportvelocitydata` (DPD-VV only)
  `dragforces_dpdvv` (DPD-VV only)
  `importforcedata` (DPD-VV only)
  `global_sca_sca_int`
  `global_sca_sca_dble`
  `global_sum_int` (Lowe, Peters, Stoyanov)
- **Arguments**

`stage`	input	integer

- **Comments**
  All versions of this subroutine use the standard Velocity Verlet algorithm to integrate the equations

  $$\frac{d\vec{v}_i}{dt} = \frac{\vec{F}_i}{m_i}$$

  $$\frac{d\vec{r}_i}{dt} = \vec{v}_i$$

  Both the MD-VV and DPD-VV algorithms use dissipative and random forces as the thermostat, with the DPD-VV algorithm repeating the calculation of dissipative forces at the end of the time step. The Lowe-Andersen, Peters and Stoyanov-Groot thermostats are applied after all other forces are integrated.

verlet_*_lang

- **Header records**
  `SUBROUTINE verlet_*_lang (stage)`
- **Function**
  Solves the equations of motion using the Velocity Verlet scheme coupled with a Langevin barostat[29].
- **Dependencies**
  `hardreflect`
  `deportdata`
  `frozenbead`
  `error`
  `exportvelocitydata` (DPD-VV only)
  `dragforces_dpdvv` (DPD-VV only)
  `importforcedata` (DPD-VV only)
  `global_sca_sca_int`
  `global_sca_sca_dble`
  `global_sum_dble`
  `global_sum_int` (Lowe, Peters, Stoyanov)
- **Arguments**

`stage`	input	integer

- **Comments**
  The Langevin barostat is configured to apply the following piston force (rate of change of momentum) in dimension α
  $$\dot{p}_{g,\alpha} = V(P_\alpha - P_0) + \frac{1}{N_f}\sum_i m_i v_i^2 - \gamma_p p_{g,\alpha} + \sigma_p \zeta_{p,\alpha} \Delta t^{-\frac{1}{2}}$$
  with the option of keeping the system isotropic by using equal values of P_α and $\zeta_{p,\alpha}$ for all dimensions. (This may be switched off if imposing constant surface tensions at given planar surfaces.) Rescaling of the simulation volume takes place in the first stage coupled with the integration of particle forces, while iteration of the barostat force to achieve convergence of particle velocities (to machine precision) takes place in the second stage.

verlet_*_berend

- **Header records**
  `SUBROUTINE verlet_*_berend (stage)`
- **Function**
  Solves the equations of motion using the Velocity Verlet scheme coupled with the Berendsen barostat[2].
- **Dependencies**
  `hardreflect`
  `deportdata`
  `frozenbead`
  `error`
  `exportvelocitydata` (DPD-VV only)
  `dragforces_dpdvv` (DPD-VV only)
  `importforcedata` (DPD-VV only)
  `global_sca_sca_int`
  `global_sum_dble`
  `global_sum_int` (Lowe, Peters, Stoyanov)

- **Arguments**

`stage`	input	integer

- **Comments**
  The Berendsen barostat rescales particle positions and system sizes in dimension α by the factor

$$\mu_\alpha = 1 + \frac{\beta}{\tau_p}\Delta t(P_\alpha - P_0)$$

  with the option of keeping the system isotropic by using equal values of P_α for all dimensions. (This may be switched off if imposing constant surface tensions at given planar surfaces.) Rescaling of the simulation volume takes place in the first stage, while calculation of the rescaling factor μ_α takes place in the second stage after force integration.

13.2.15 statistics_module

This module requires the modules `constants`, `variables`, `start_module`, `comms_module` and `numeric_container` to be loaded beforehand.

printout

- **Header records**
  `SUBROUTINE printout (time, lbegin)`

- **Function**
  Writes summary of simulation at the current time step to `OUTPUT` file.

- **Dependencies**
  None

- **Arguments**

`time`	input	real(KIND=dp)
`lbegin`	input	logical

- **Comments**
  The logical `lbegin` indicates whether or not the column titles should be printed before the data.

corout

- **Header records**
  `SUBROUTINE corout (time)`

- **Function**
  Writes statistical data to `CORREL` data file.

- **Dependencies**
  None

- **Arguments**

`time`	input	real(KIND=dp)

histout

- **Header records**
  `SUBROUTINE histout (time)`
- **Function**
  Writes trajectory data (particle positions and velocities) to `HISTORY*` data files.
- **Dependencies**
  None
- **Arguments**

`time`	input	real(KIND=dp)

result

- **Header records**
  `SUBROUTINE result`
- **Function**
  Writes final summary of simulation to `OUTPUT` file.
- **Dependencies**
  `revive`
- **Arguments**

`idnode`	input	integer
`nodes`	input	integer

statis

- **Header records**
  `SUBROUTINE statis`
- **Function**
  Calculates statistical properties of system, including rolling averages and fluctuations.
- **Dependencies**
  `sclsum`
  `global_sum_dble`
  `global_sca_max_dble`
  `global_sca_min_dble`

Chapter 14

DL_MESO DPD Examples

Test cases for Dissipative Particle Dynamics simulations using DL_MESO – including the required input and sample output files – can be found in the `DEMO/DPD` subdirectory. All of the following examples can be run using either the serial or parallel versions of DL_MESO_DPD; 96 processing units were used to test them in parallel although smaller and larger numbers should also work. The smaller problems (i.e. with up to 20 000 particles) are best suited to running in serial or on a small number of processor cores (e.g. 16 or less) to limit the times required for interprocess communication, while larger problems are better suited to running in parallel to reduce the memory requirements per processor core.

Images of all test cases and videos for some can be found in the Example Simulations page of the DL_MESO website: a link to it can be found at `www.ccp5.ac.uk/DL_MESO`

14.1 Mixture_Small

This simulation consists of 1000 particles with 3 species with populations of 333, 333 and 334 respectively. All particle types have identical sizes and masses but different energy parameters, using the default mixing rules for unlike particle parameters. Figure 14.1 gives a snapshot of the system at the end of the simulation, demonstrating mixing between the three particle types (represented by different colours).

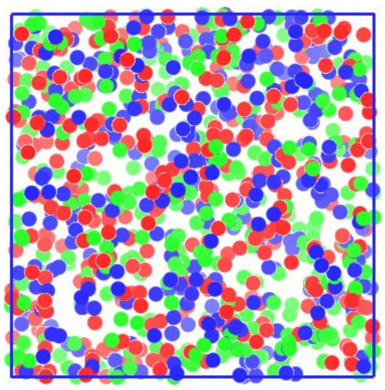


Figure 14.1: Visualization of system at final time step from DPD `Mixture_Ser` test case

14.2 Mixture_Large

This simulation example is similar to `Mixture_Small` but larger: it consists of 512 000 particles with 2 species, each with a population of 256 000 particles. The particle types have identical sizes and masses but different energy parameters, using the default mixing rules for unlike particle parameters. Figure 14.2 gives a snapshot of the system at the end of the simulation, demonstrating good mixing between the two particle types.

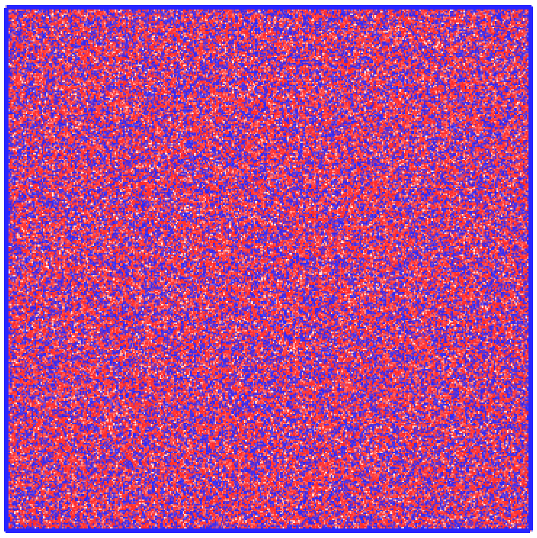


Figure 14.2: Visualization of system at final time step from DPD `Mixture_Par` test case

14.3 PhaseSeparation

This simulation example consists of 3000 particles with 2 species, each with a population of 1500. Both particle types have identical sizes, masses and like-like energy parameters, but the unlike energy parameter has been set to a larger value to produce phase separation, which can clearly be seen in Figure 14.3. The initial state of this simulation has been provided in a `CONFIG` file. An `.AVI` video file of the first half of the simulation can be found in the Example Simulations page of the DL_MESO website.

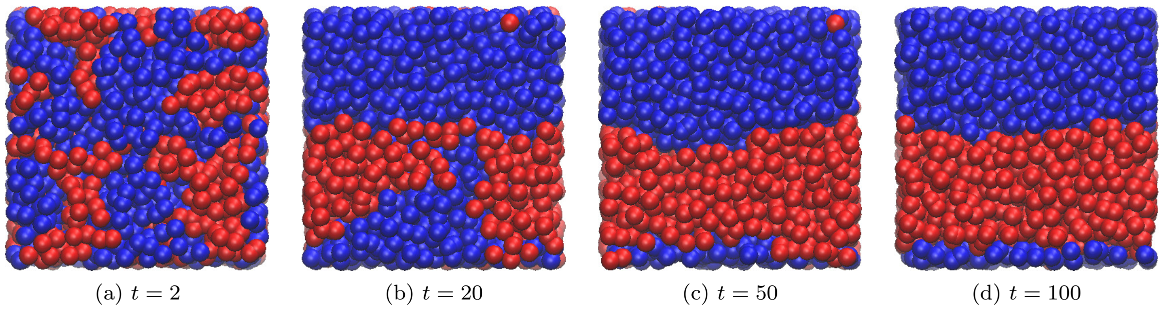

(a) $t = 2$ (b) $t = 20$ (c) $t = 50$ (d) $t = 100$

Figure 14.3: Visualizations of DPD `PhaseSeparation` test case (red for particle type 1, blue for type 2)

14.4 Aggregate

This simulation consists of 3000 unbonded particles and 30 molecules of 10 particles each with harmonic bonds between them. The unbonded particles and molecules are made up of different species with a higher energy parameter for unlike particle interactions. This causes the molecules to aggregate, which can be seen in Figure 14.4, a snapshot of the simulation.

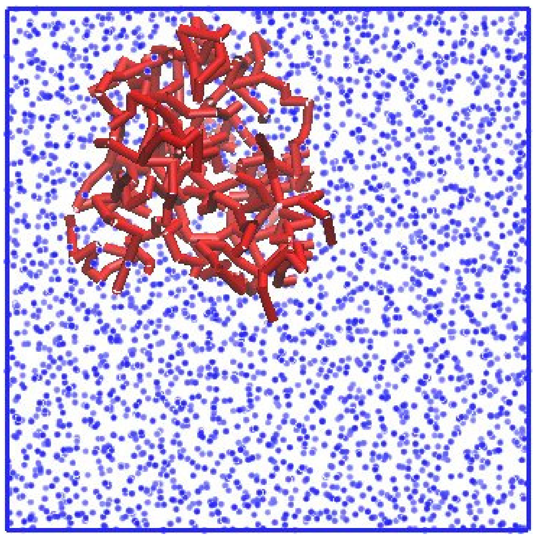
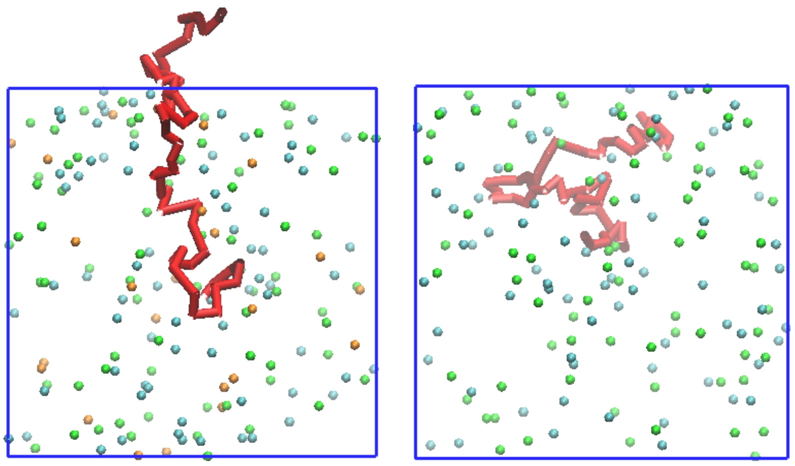


Figure 14.4: Visualization of system at $t = 2220$ from DPD `Aggregate` test case

14.5 Polyelectrolyte

This simulation example consists of a slightly hydrophobic polyelectrolyte molecule of 50 particles, each with a relative charge of $+0.5$, immersed in a salt solution of concentration $0.14M$[14]. The salt solution consists of 9900 neutral water particles, 75 cationic salt particles (net charge $+1$), 75 anions of charge -1 and 25 counterions of charge -1 to keep the system neutral. A similar simulation is included with the polyelectrolyte replaced with a neutral polymer of the same number of particles and the counterions replaced with water (`FIELD-neutral`: this should be renamed to `FIELD` to run the simulation and used with the same `CONTROL` file). Figure 14.5 gives a comparison between the polyelectrolyte and neutral polymer at the final time step, which have measured radii of gyration of 4.5 and 2.7 respectively.



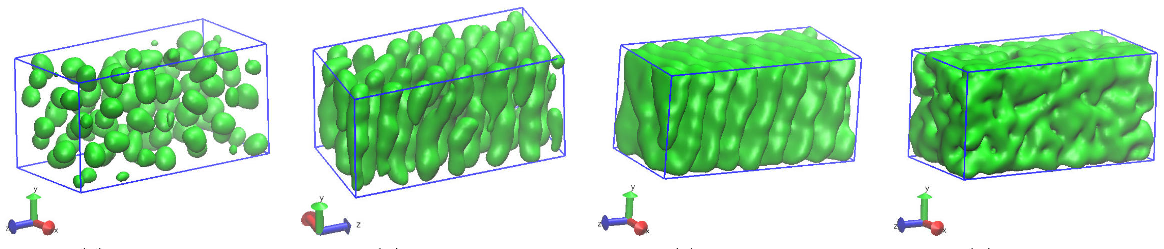
(a) Polyelectrolyte (b) Neutral polymer

Figure 14.5: Visualizations of DPD `Polyelectrolyte` test case: red for polyelectrolyte/polymer, green for salt cations, cyan for anions, orange for counterions (water omitted for clarity)

14.6 AmphiphileMesophases

This example consists of four separate simulations, each with 12 000 particles consisting of dimers (molecules consisting of two particles, one hydrophilic and the other hydrophobic, with harmonic bonds of equilibrium length 0.5 between them) and unbonded monomers[31]. Defining the composition ϕ as the ratio of DPD particles within dimers to the total number of particles in the system, the interaction data for simulations with dimer compositions of 30%, 55%, 75% and 90% are provided with filenames `FIELD-30`, `FIELD-55`, `FIELD-75` and `FIELD-90` respectively. (Each of these files should be renamed to `FIELD` to run the simulation, while the `CONTROL` defining the simulation properties can be used for all four simulations.)

These systems provide four points on a phase diagram corresponding to isotropic dimer, hexagonal, lamellar and isotropic monomer phases respectively. The final configurations obtained for each phase can be seen in Figure 14.6, shown as isosurfaces of the hydrophobic particles to highlight the distinctions between the phases.

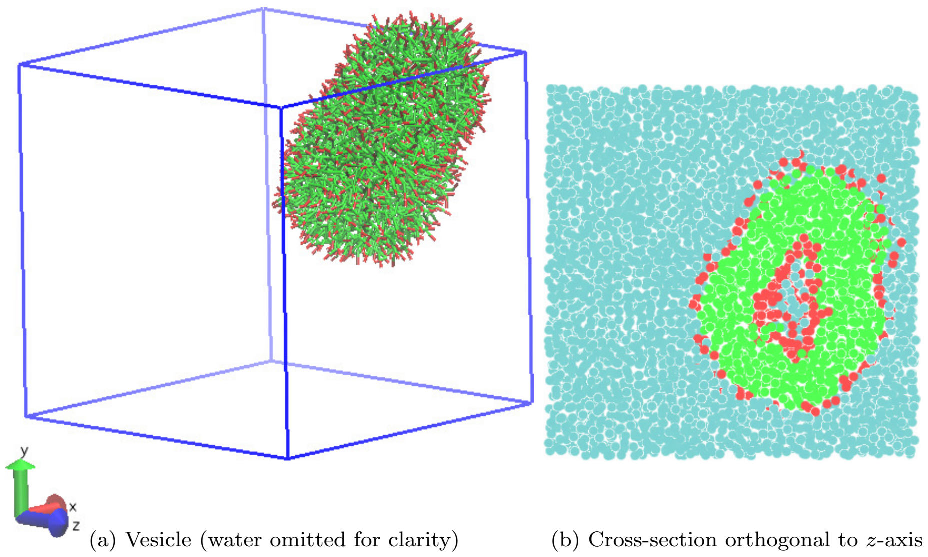


(a) $\phi = 0.30$ (b) $\phi = 0.55$ (c) $\phi = 0.75$ (d) $\phi = 0.90$

Figure 14.6: Visualizations of DPD `AmphiphileMesophases` test case at final time step (isosurfaces of hydrophobic particles)

14.7 VesicleFormation

This simulation example consists of 37 440 unbonded water particles and 1008 molecules, each consisting of one hydrophilic head particle and three hydrophobic tail particles bonded together with stiff harmonic bonds of equilibrium length 1.0 between them[71]. The molecules represent amphiphiles and during the course of the simulation self-assemble into a vesicle and encapsulate a number of water particles. Figure 14.7 shows the self-assembled vesicle, both in three dimensions and in a cross-section to show the encapsulated water. An `.AVI` video file of the simulation can be found in the Example Simulations page of the DL_MESO website.

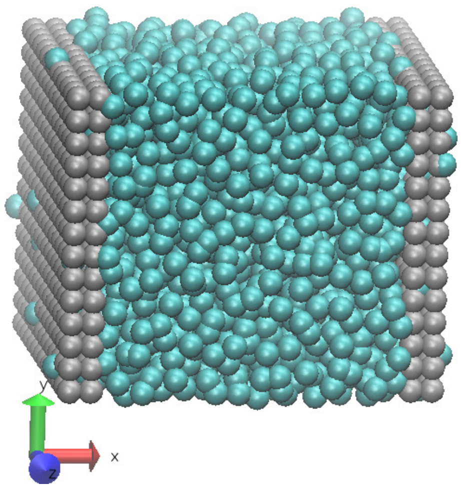


(a) Vesicle (water omitted for clarity) (b) Cross-section orthogonal to z-axis

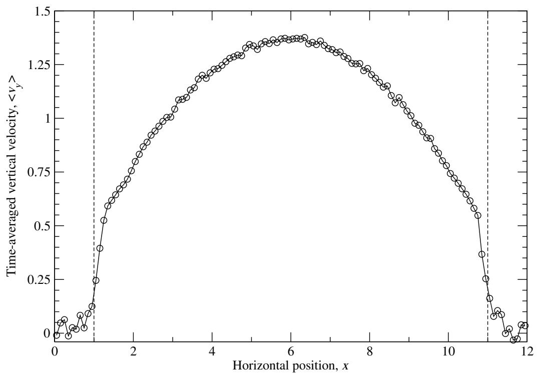
Figure 14.7: Visualizations of DPD `VesicleFormation` test case at $t = 50\ 000$ (red for hydrophile, green for hydrophobe, blue for water)

14.8 PoiseuilleFlow

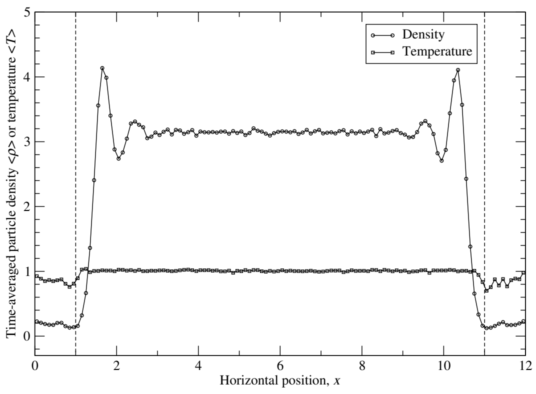
This simulation example consists of 3000 unbonded particles in a box of $10 \times 10 \times 10$ DPD length units with walls of frozen particles added to the surfaces orthogonal to the x-axis of thickness 1 DPD length unit and particle density of 3. A constant body force in the direction of the y-axis is added to each non-frozen particle which, in combination with the frozen particle walls approximating no-slip boundaries, gives Poiseuille flow of the DPD fluid. Figure 14.8 gives a snapshot of the system at the final time step, as well as plots of y-component velocity, density of the fluid particles and temperature (defined only by x- and z-components of velocity in this case). The snapshot demonstrates a slight porosity of the frozen particle wall due to soft DPD interactions, which could be alleviated by a higher frozen particle density[48], while the emergent velocity profile is similar to that expected for Poiseuille flow. The temperature and density profiles are mainly flat across the entire spacing between the walls, apart from significant density fluctuations close to the walls.



(a) Visualization of system at $t = 2\ 000$



(b) Time-averaged y-component of velocity



(c) Time-averaged fluid density and temperature

Figure 14.8: Visualization and plots from DPD `PoiseuilleFlow` test case: broken lines denote positions of no-slip boundaries due to frozen particle walls

14.9 ShearFlow

This simulation example consists of 3000 unbonded particles in a box of $10 \times 10 \times 10$ DPD length units with Lees-Edwards shearing boundaries orthogonal to the y-axis. The Stoyanov-Groot thermostat is used for this system to control both the fluid viscosity and system temperature. Figure 14.9 gives the time-averaged emergent velocity profile, yielding a shear rate of 0.1941 (in DPD units, close to the applied shear rate of 0.2) against a measured stress component $\langle \sigma_{yx} \rangle = -0.2063$.

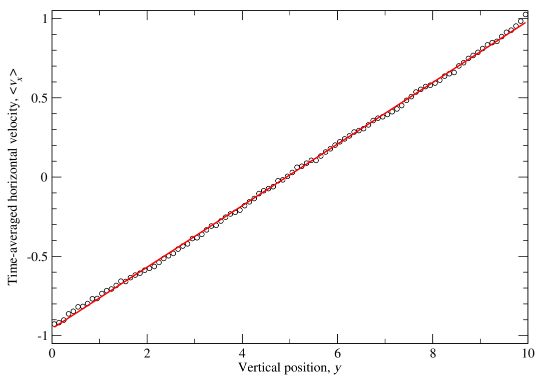


Figure 14.9: Plot of x-component velocity from DPD `ShearFlow` test case: red line denotes best-fit for determining shear rate

14.10 VapourLiquid

This simulation example consists of 1000 unbonded water particles initially distributed uniformly in a box of $5 \times 5 \times 22$ DPD length units, using the default many-body DPD interactions with $A_{ij} = -50$ and $B_{ij} = 25$ to apply vapour-liquid interactions and surface tension[68, 12]. Figure 14.10 shows the system at the final timestep after the water particles have coalesced into a single body surrounded by empty space.

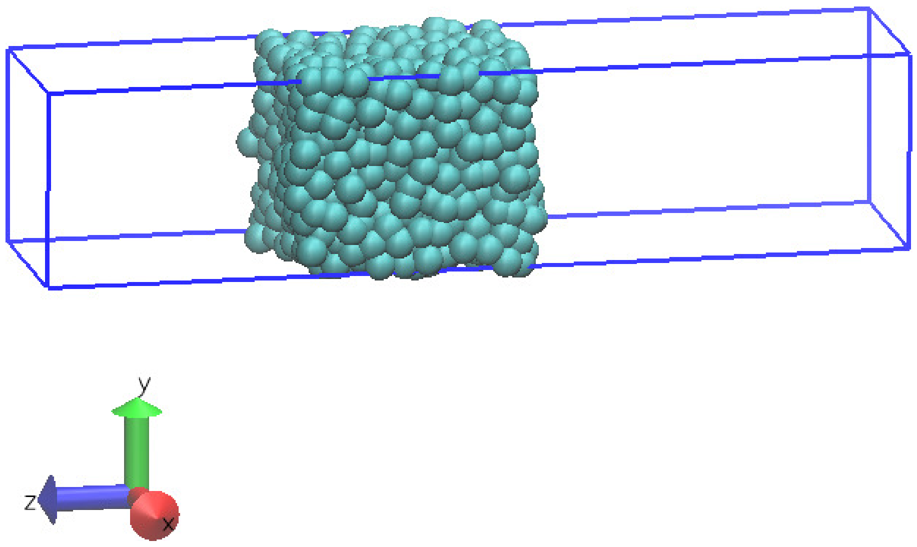


Figure 14.10: Visualization of system at final time step from DPD `VapourLiquid` test case

Appendix A

Manual compiliation and running of DL_MESO

A.1 DL_MESO_LBE

DL_MESO_LBE has been written in C++ in a modular form and the main program codes – `slbe.cpp` for serial running, `plbe.cpp` for parallel running – are designed to allow the user to change algorithms for collision, mesophases etc. by specifying them in input files. Customised codes (`slbecustom.cpp`[1] and `plbecustom.cpp`) are also available to allow users to 'hardwire' the algorithms for collisions, mesophases etc. into the code, which might improve computational efficiency. (Using incompressible fluids would require setting the parameter `incompress` equal to 1: this can either be hard-wired into the customised codes or specified in the `lbin.sys` file.)

To compile the code and produce an executable in the working directory, at a command line type:

- `c++ ../LBE/slbe.cpp -o lbe.exe`[2]

assuming that `c++` is the name of the available C++ compiler, `slbe.cpp` is the version of the code being compiled and `lbe.exe` is the name of the executable required. If compiling the parallel version of the code, the command for the C++ compiler 'wrapped' with MPI is required, which is commonly `mpiCC`. Additional compiler flags may be used between the compiler name and the reference to the code to improve computation speed or assist in debugging.

Before running the executable, the required input files (`lbin.sys`, `lbin.spa` and optionally `lbin.init`) need to be copied or moved into the working directory. If running in serial, the executable can just be run using the command:

- `lbe.exe` if running in Windows, or
- `/lbe.exe` if running in Unix-like operating systems,

while the parallel version requires a command to run N identical copies of the program on N processors, e.g.

- `mpirun -np` N `./lbe.exe`

and may need to be launched via a batch job script: please consult your machine administrator or documentation for further details.

Modifications may be made to the customised versions of the code to select routines for e.g. specific collision, propagation and mesophase interaction algorithms. For example, the Guo forcing scheme can be applied by

[1]If preferred, an alternative version that uses a boundary layer, `slbecombine.cpp`, is available.
[2]`/` may need to be replaced by \ on computers running Windows.

selecting collision routines with `Guo`, e.g. `fCollisionBGKGuo`, while multiple-relaxation-time (MRT) schemes can be used with calls to `fCollisionMRT*`. Other possible modifications include modifying tuneable parameters for MRT schemes, which are given in `lbpMODEL.cpp` for the required lattice model (D2Q9, D3Q15 or D3Q19). Additional boundary conditions should be added to the `fPostCollBoundary*` or `fPostPropBoundary*` routines, depending on whether they are applied after collisions (and before propagation) or after propagation respectively. Calls to alternative and/or additional output routines can be included in the main loops of the customisable codes, provided they are called before the output file number (`qVersion`) is increased. If no fluid-fluid interaction forces are required, the calls to routines to zero interaction forces (`fInteractionForceZero`) and calculate them (e.g. `fInteractionForceShanChen`) can be commented out to increase calculational efficiency.

A.2 DL_MESO_DPD

The Fortran90 modules for DL_MESO_DPD must be compiled in a particular order to satisfy dependencies of shared variables and arrays:

- `constants`
- `variables`
- `numeric_container`
- `comms_module`
- `error_module`
- `parse_utils`
- `bond_module`
- `surface_module`
- `ewald_module`
- `manybody_module`
- `domain_module`
- `start_module`
- `config_module`
- `field_module`
- `integrate_module`
- `statistics_module`
- `run_module`
- `dlmesodpd`

If running DL_MESO_DPD in serial, the modules `comms_module` and `domain_module` should be replaced by `comms_module_ser` and `domain_module_ser` respectively.

To simplify the process, a makefile may be created either in the `DPD` directory or in the working directory to automatically compile the modules and build the executable. Examples of these for running in the `DPD` directory may be found in the `DPD/makefiles` directory and modified by the user. The compiler (after `FC=`) and flags (after `FFLAGS=`) may need changing depending on the Fortran90 compiler available: if MPI is available, the

Fortran90 compiler 'wrapped' with MPI (most commonly `mpif90`) is required. If invoking from the working directory, the modules for DL_MESO_DPD should either be preceded by the path, i.e. `../DPD/`, in the list of compile sources or the directive `VPATH=../DPD/` can be used before the source list; the latter strategy is used by the DL_MESO GUI when creating makefiles.

DL_MESO_DPD can be compiled using the command `make` if the makefile is called `Makefile`, or if it has a custom name (e.g. `Makefile-custom`) by the command

- `make -f Makefile-custom`

The example makefiles will produce an executable with the name `dpd.exe`, which can be copied to the working directory (if necessary). The required input files (`CONTROL` and `FIELD`) will also need to be created in or copied into the same directory, as well as an optional `CONFIG` file to specify an initial configuration for a new simulation. `export*` files from a previous run can be used for restarting a previous simulation, providing the number of processing units remains the same; if the number of processing units changes, the utility `exportconfig` can be used to convert the restart files into a `CONFIG` file.

If running in serial, the executable can just be run using the command:

- `dpd.exe` if running in Windows, or
- `/dpd.exe` if running in Unix-like operating systems,

while the parallel version requires a command to run N identical copies of the program on N processors, e.g.

- `mpirun -np` N `./dpd.exe`

and may need to be launched via a batch job script: please consult your machine administrator or documentation for further details.

The maximum numbers of particles per process (`maxdim`), pairs of unbonded interactions (`maxpair`) and the maximum sizes of transfer buffers `maxbuf` are automatically set according to the total number of particles in the system and the number of processing units to be used for simulations. The value of `maxdim` can be increased by the user to allow for non-evenly distributed systems by setting a value for **densvar** in the `CONTROL` file.

If using alternative many-body DPD interactions to the vapour-liquid example provided, the routines `manybody_force` and `manybody_potential` in `manybody_module.f90` should be modified by the user as necessary; the routine `local_density` should not be altered by the user but the function `weight_rho` can be changed if an alternative weighting function for calculating local densities is required. Additional bond, angle and dihedral types can also be added to the subroutines `bond_force`, `angle_force` and `dihedral_force` respectively in `bond_module.f90`, but this will also require changes to `read_field` in `config_module.f90` to include a four-letter code for the bond/angle/dihedral type that can be read from the `FIELD` file.

Appendix B

DL_MESO Utilities

DL_MESO includes a number of utility programs which are not directly needed for Lattice Boltzmann or DPD simulations but are useful both for producing files required as inputs for those calculations and to process output files for visualization. These may be found in the `/LBE/utility` and `/DPD/utility` directories.

Compilation can either be carried out individually or collectively using makefiles: each utility directory includes a makefile to compile all the utilities therein and the working directory `/WORK` includes one to compile both sets for use with the GUI[1]. The latter can be invoked using the command

- `make -f Makefile-utils`

Some further details on these utilities can be found in the `README` files in the source directories.

B.1 DL_MESO_LBE

lbeinitcreate

`lbeinitcreate` is a utility written in C++ to create initialisation files (`lbin.init`) to override the default initial conditions. This utility can add fluid drops to the system (either circular in 2D or spherical in 3D) and rectangular 'sources' of specified solute concentrations or temperature to a system.

If `c++` is the command for the available C++ compiler, the executable `init.exe` can be produced by typing

- `c++ -o init.exe lbeinitcreate.cpp`

and run at the command line (`init.exe` or `./init.exe`).

A pre-existing `lbin.sys` file needs to exist in the directory where the utility is run, as this provides information on the dimensions and size of the simulation system, the numbers of fluids and initial and constant densities for each fluid, the number of solutes, whether or not a thermal lattice is included and the default initial velocity. This information is displayed on the screen when the utility is run: if no `lbin.sys` file can be found, an error message will be displayed and the utility will terminate.

If more than one fluid is specified in the `lbin.sys` file, the utility will then attempt to determine the continuous fluid for the system from the initial densities and, if necessary, ask the user to identify it. The utility will then ask for the number of drops to be added to the system: for each drop, the user will need to specify the fluid, its radius and where its centre is located on the lattice grid. (Note that it is possible for a drop to extend beyond the grid boundaries if periodic boundaries are in use.)

[1]If using the GUI and the utilties are to be compiled manually or in their source directories, copies of the executables are required in the directory from which the GUI is to be launched, e.g. `/WORK`.

If any solutes are to be included, the utility will ask for the number of solute ‘sources’ (i.e. regions of constant solute concentration): for each source, it will then ask for the solute number, the required concentration, the location of one corner of the rectangular source and its extent in each dimension (which can extend beyond periodic boundaries). Similarly, if a temperature grid is included in the system, the utility will ask for the number of temperature 'sources', followed by the required temperature and the location of the corner and the extent of the source.

Once all of the above information is obtained, the utility will then create the `lbin.init` file, which specifies the grid points, velocities, fluid densities, solute concentrations and temperatures for any locations in the system that require non-default initial conditions.

lbeplot3dgather

`lbeplot3dgather` is a utility written in C++ to gather Plot3D output files produced by the parallel version of DL_MESO_LBE and produce a single structure file (`lbtout.xyz` or `lbtout.xy`) and a single set of solution files (`lbtout*.q`) for visualization of the entire system.

If `c++` is the command for the available C++ compiler, the executable `plot3d.exe` can be produced by typing

- `c++ -o plot3d.exe lbeplot3dgather.cpp`

and either run at the command line or via the GUI under **Gather LBE Data**.

All `lbout*.xyz` and `lbout*.q` files should be copied to the directory including the executable (if necessary) before running, as well as the `lbout.info` file to give information on the sizes of integers and floating point numbers. No user input is required, although the utility will stop with an error message if no `lbout.info` file is available. No other error messages are produced, so care should be taken to ensure no solution files are missing.

This utility can be run with a command line argument to give the scalar property required, e.g. for Windows and Unix/Linux computers respectively

- `plot3d.exe 1`
- `./plot3d.exe 1`

where 0 is used for all properties, 1 for fluid density, 2 for mass fraction, 3 for solute concentration and 4 for temperature. (If the GUI is used, this can be selected using the pulldown list in the **Gather LBE Data** panel.) If the argument is omitted, the utility will ask the user to enter the required property. No other user input is required, but error messages will be produced if either of the files `lbout.info` and `lbout.ext` are missing. No other error messages are produced, so care should be taken to ensure no solution files for the pieces are missing before running the utility. Since the data is copied into the combined structure and solution files, the original output files can be deleted after this utility is run.

lbevtkgather

`lbevtkgather` is a utility written in C++ to gather Structured Grid XML-formatted VTK output files produced by the parallel version of DL_MESO_LBE (`lbout*.vts`) and produce a set of linking files (`lbtout*.pvts`) for visualization of the entire system.

If `c++` is the command for the available C++ compiler, the executable `vtk.exe` can be produced by typing

- `c++ -o vtk.exe lbevtkgather.cpp`

and either run at the command line or via the GUI under **Gather LBE Data**.

All `lbout*.vts` files should be copied to the directory including the executable (if necessary) before running, as well as the `lbout.info` and `lbout.ext` files to give information about the number of processors used for the simulation and the extents of each piece.

This utility can be run with a command line argument to give the scalar property required, e.g. for Windows and Unix/Linux computers respectively

- `vtk.exe 1`
- `./vtk.exe 1`

where 0 is used for all properties, 1 for fluid density, 2 for mass fraction, 3 for solute concentration and 4 for temperature. (If the GUI is used, this can be selected using the pulldown list in the **Gather LBE Data** panel.) If the argument is omitted, the utility will ask the user to enter the required property. No other user input is required, but error messages will be produced if either of the files `lbout.info` and `lbout.ext` are missing. No other error messages are produced, so care should be taken to ensure no VTK files for the pieces are missing, particularly since these files are required for plotting as the linking files do not include the data.

B.2 DL_MESO_DPD

convert-input

`convert-input` is a utility written in C++ to read DPD input files created for earlier versions of DL_MESO (up to version 2.4) and create `CONTROL` and `FIELD` files formatted in the style for versions 2.5 and later.

This utility can be compiled to produce the executable `convert.exe` with the command

- `c++ -o convert.exe convert-input.cpp`

if `c++` is the command for the available C++ compiler.

This utility can be run with up to three optional command line arguments specifying the names of the `CONTROL`, `FIELD` and `MOLECULE` files in that order if they have alternative names. If the standard names are used, the old `CONTROL` and `FIELD` are renamed after being read to prevent them being overwritten with the new versions of those files. (The `MOLECULE` file is no longer required and therefore does not need to be renamed.)

molecule-generate

`molecule-generate` is a utility written in C++ to generate the input files required for modelling particles in DPD simulations that are bonded together, i.e. molecules. A random flight generation system is used to generate the coordinates of bonded beads – which can form branched molecule chains – a constant distance apart within a cube of a size specified by the user, which will be used by DL_MESO_DPD to insert the molecule into the system.

This utility can be compiled to produce the executable `molecule.exe` with the command

- `c++ -o molecule.exe molecule-generate.cpp`

if `c++` is the command for the available C++ compiler. This utility can be run from the command line or via the GUI in **Set DPD Molecules** (which runs the utility in a new command line/shell window).

When running the utility, if a `FIELD` file exists in the same directory as the executable, the number of species and their names will be read from it; otherwise the user will be asked to enter this information and this will be written to a new `FIELD` file. The user will then be asked for the number of molecules required, the numbers of bond, angles and dihedrals and their types and parameters.

For each molecule, the user is asked for its name, the number to be included in the system and whether or not isomers of the molecule can be included. The side length for the cube inside which the molecule will fit is then required, followed by the bond length, the number of molecule chains and the number of particles for each chain. If the chain in question is not the first (primary) chain, the user will also be asked for a pre-existing bead number as the starting point for the chain.

After this, the default species for the beads in the molecule will be requested: the user will then be asked enter the bead numbers for each of the other species (0 can be entered to finish specifying bead numbers). If more than one bond type is to be included, the user will be asked to select the default bond type and then select the bonds that are of different types by typing the index bead number (and optionally the destination bead if more than one is available). Bond angles and/or dihedrals can also be selected by typing in the index bead number and then selecting the required bead triple or quadruple if more than one is available.

The molecules will be appended to the `FIELD` file in the correct format (see Section 12.1 for more details) with positions for the beads relative to each molecule's centre of mass. Note that this file will not be quite complete after running this utility: data for unbonded interactions and external force fields may be required (if the `FIELD` file is created using the utility) and a **close** directive will be required at the end.

exportconfig

`exportconfig` is a utility written in Fortran90 to produce a configuration file in DL_POLY format (`CONFIG`) from DL_MESO_DPD restart files (`export*`), which can be used as a starting point for new simulations (including restarting simulations on different numbers of processes). Since a limited amount of data is included in restart files, the `FIELD` file for the simulation is needed to provide some additional information.

The source code for this utility, `exportconfig.f90`, can be used for `export` or `export*` files created by both the serial and parallel versions of DL_MESO_DPD: their endianness is automatically detected, so the utility can be run on a different machine to the one used for DPD calculations. If the available Fortran90 compiler is invoked by the command `f90`, the executable `config.exe` can be produced by typing

- `f90 -o config.exe exportconfig.f90`

and either run at the command line or by using the GUI in **Process DPD Data** after entering the number of processes used in the required field and selecting the required `CONFIG` file key in the pulldown list.

This utility can be run with two command line arguments, the first indicating the number of processes used to generate the restart data and the second denoting the `CONFIG` file key (`levcfg`: 0 = positions only, 1 = positions and velocities, 2 = positions, velocities and force), e.g. if 16 processes were used and the particle positions and velocities are required, either of the following commands can be used:

- `export.exe 16 1`
- `./export.exe 16 1`

If no command line argument is given, the utility will ask the user to type in the number of processes and the `CONFIG` file key.

exportimage

`exportimage` is a utility written in Fortran90 to produce a VTF format trajectory file (`export.vtf`) from DL_MESO_DPD restart files (`export*`) that can be visualized to give a snapshot of the last simulation timestep.

Since a limited amount of data is included in restart files, the `FIELD` file for the simulation is needed to provide some additional information.

The source code for this utility, `exportimage.f90`, can be used for `export` or `export*` files created by both the serial and parallel versions of DL_MESO_DPD: their endianness is automatically detected, so the utility can be run on a different machine to the one used for DPD calculations. If the available Fortran90 compiler is invoked by the command `f90`, the executable `export.exe` can be produced by typing

- `f90 -o export.exe exportimage.f90`

and either run at the command line or by using the GUI in **Process DPD Data**.

This utility can be run with a command line argument indicating the number of processes used to generate the restart data, e.g. if 16 processes were used, either of the following commands can be used:

- `export.exe 16`
- `./export.exe 16`

If no command line argument is given, the utility will ask the user to type in the number of processes. (The number of processes can be specified in the GUI before running the utility.)

traject

The Fortran90 utility `traject` reads in `HISTORY*` output data files generated by DL_MESO_DPD and produces a VTF format trajectory file (`traject.vtf`) that can visualize the simulation after equilibration, such that snapshots at the recorded timesteps and animations can be produced.

Two versions of the utility are provided: `traject.f90` outputs every particle for all recorded timesteps, and `trajectselected.f90` allows the user to select the number of particles and the number of timesteps to output to the trajectory file. Both versions of the utility can be used with both `HISTORY` and `HISTORY*` files produced by the serial and parallel versions of DL_MESO_DPD respectively: their endianness is automatically detected, so the utility can be run on a different machine to the one used for DPD calculations. If `f90` is the command for the available Fortran90 compiler, the executable `traject.exe` can be produced by typing

- `f90 -o traject.exe traject.f90`

and either run at the command line or via the GUI in **Process DPD Data**. A command line argument indicating the number of processes (and therefore the number of `HISTORY` or `HISTORY*` files) can be included in a similar way to the `exportimage` utility.

The alternative version can be compiled in the same way: by default the makefiles create executables named `trajects.exe`. This version takes the same command line argument as `traject.f90`. After the number of processes is typed in, the utility displays the total number of particles, the number of unbonded particles in the simulation and the total number of timesteps available, before asking for the particle number range and number of timesteps (including a starting timestep) to be written in the trajectory file. This version of the utility cannot be invoked using the GUI.

local

`local` is a utility written in Fortran90 that can read in `HISTORY*` output data files generated by DL_MESO_DPD and produce series of VTK format files containing statistical properties – number of beads, density, compositions per particle and molecule types, temperature and mean velocity – in cuboidal subdivisions of the simulation volume for plotting and/or visualization.

The source code for this utility, `local.f90`, can read both `HISTORY` and `HISTORY*` files generated by the serial and parallel versions of DL_MESO_DPD respectively: their endianness is automatically detected, so the utility can be run on a different machine to the one used for DPD calculations. If `f90` is the command for the available Fortran90 compiler, the executable `local.exe` can be produced by typing

- `f90 -o local.exe local.f90`

and either run at the command line or via the GUI in **Process DPD Data** after entering both the number of processes used in simulations and the number of divisions required in each dimension.

This utility can be run with four command line arguments: the first denoting the number of processes and the others giving the number of divisions required in x-, y- and z-directions respectively. If these arguments are omitted, the user will be asked to enter these values.

Files named `local_*.vtk` are produced for all the specified time steps after equilibration containing the following data for each cuboidal cell:

- the mean velocity for all unfrozen beads
- the number of unfrozen beads
- overall temperature
- partial temperatures for each dimension (i.e. for dimension α, $T_\alpha = \sum_i m_i v_{i,\alpha}^2$)
- densities for each bead species
- volume fractions for bead species (starting from type 01)
- volume fractions for molecule types (starting from type 00 for all unbonded beads)

An additional file, `averages.vtk`, is also produced with time-averaged values for the velocities, species densities, overall and partial temperatures in each cuboidal cell.

The scalar properties (including compositions) may be considered to act across the entire volumes of the cells, while the velocities are representative for the cell centres.

Appendix C

DL_MESO_DPD Error Messages

This appendix documents the error and warning messages currently available in the DPD code in DL_MESO, DL_MESO_DPD, and recommendations for users to try and overcome the errors. Users may contact the authors of DL_MESO *after* attempting the recommended actions.

Message 1: cutoff radius value not set

A valid cutoff radius (r_c) for all interactions cannot be found in the `CONTROL` file: this is a compulsory parameter for DPD simulations.

Action: Look in the `CONTROL` file and make sure the **cut**off directive is included with a non-zero value.

Message 2: temperature not set

A valid system temperature ($k_B T$) cannot be found in the `CONTROL` file: this is a compulsory parameter for DPD simulations.

Action: Look in the `CONTROL` file and make sure the **temp**erature directive is included with a non-zero value.

Message 3: time step size not set

A valid simulation timestep (Δt) cannot be found in the `CONTROL` file: this is a compulsory parameter for DPD simulations.

Action: Look in the `CONTROL` file and make sure the **timestep** directive is included with a non-zero value.

Message 4: boundary halo size larger than half subdomain size

The size of the boundary halo (either specified by the user or determined from required interaction and bond lengths) exceeds half the length of at least one dimension of the subdomain volume assigned to each processor. The DPD simulation may therefore run less efficiently.

Action: None required to ensure the simulation runs as this is a warning message, but the user may wish to reduce the specified boundary halo size or use global bond calculations for future calculations.

Message 5: too many beads per node

The number of particles likely to be assigned to each processor is greater than the calculated maximum value `maxdim`.

Action: This error is unlikely to happen as `maxdim` is calculated according to the numbers of particles and processors available, but the user may wish to use the **densvar** directive in the `CONTROL` file to increase this value.

Message 10: cannot read CONFIG file

The supplied `CONFIG` file cannot be read by DL_MESO_DPD: it might have been corrupted.

Action: Check the `CONFIG` file to ensure it is complete and in ANSI (text) format.

Message 20: missing CONTROL file

No input file named `CONTROL` can be found.

Action: Make sure there is a `CONTROL` file in the same directory as the DL_MESO_DPD executable.

Message 21: cannot read CONTROL file

The supplied `CONTROL` file cannot be read by DL_MESO_DPD: it might have been corrupted.

Action: Check the `CONTROL` file to ensure it is complete and in ANSI (text) format.

Message 30: missing FIELD file

No input file named `FIELD` can be found.

Action: Make sure there is a `FIELD` file in the same directory as the DL_MESO_DPD executable.

Message 31: cannot read FIELD file

The supplied `FIELD` file cannot be read by DL_MESO_DPD: it might have been corrupted.

Action: Check the `FIELD` file to ensure it is complete and in ANSI (text) format.

Message 32: unrecognised bond type defined in FIELD file

A bond type not included in Table 12.3 has been found in the `FIELD` file.

Action: Check the `FIELD` file to ensure all bond types are valid; if adding a new bond type to DL_MESO_DPD, the `scan_field` and `read_field` routines in `config_module` need to be modified.

Message 33: unrecognised bond angle type defined in FIELD file

A bond angle type not included in Table 12.4 has been found in the `FIELD` file.

Action: Check the `FIELD` file to ensure all bond angle types are valid; if adding a new bond angle type to DL_MESO_DPD, the `scan_field` and `read_field` routines in `config_module` need to be modified.

Message 34: unrecognised bond dihedral type defined in FIELD file

A bond dihedral type not included in Table 12.5 has been found in the `FIELD` file.

Action: Check the `FIELD` file to ensure all bond dihedral types are valid; if adding a new bond dihedral type to DL_MESO_DPD, the `scan_field` and `read_field` routines in `config_module` need to be modified.

Message 35: non-existent species given in FIELD file for molecule i

An undefined species has been found in the definition for the i-th molecule type in the `FIELD` file.

Action: Check the `FIELD` file, particularly the i-th molecule type and the species definitions, to ensure the species in the molecule are defined.

Message 36: unrecognised bond definition in FIELD file for molecule i

A bond definition has been found in the `FIELD` file for the i-th molecule that was not detected during the initial scan of the input file.

Action: This error should never occur! If it does, please contact the authors of DL_MESO.

Message 37: unrecognised bond angle definition in FIELD file for molecule i

A bond angle definition has been found in the `FIELD` file for the i-th molecule that was not detected during the initial scan of the input file.

Action: This error should never occur! If it does, please contact the authors of DL_MESO.

Message 38: unrecognised dihedral angle definition in FIELD file for molecule i

A bond dihedral definition has been found in the `FIELD` file for the i-th molecule that was not detected during the initial scan of the input file.

Action: This error should never occur! If it does, please contact the authors of DL_MESO.

Message 40: non-existent species given in FIELD file for unbonded interaction i

An undefined species has been found in the i-th (unbonded) interaction definition in the `FIELD` file.

Action: Check the `FIELD` file, particularly the i-th interaction type and the species definitions, to ensure the species in the interaction are defined.

Message 41: non-existent species given in FIELD file for surface interaction i

An undefined species has been found in the i-th (hard) surface interaction definition in the `FIELD` file.

Action: Check the `FIELD` file, particularly the i-th surface interaction and species definitions, to ensure the species in the interaction are defined.

Message 42: non-existent species given in FIELD file for frozen wall interaction

An undefined species has been found in the definition for frozen particle walls in the `FIELD` file.

Action: Check the `FIELD` file, particularly the frozen particle wall and species definitions, to ensure the required frozen particle species is defined.

Message 43: incomplete many-body DPD interaction data in FIELD file

Not all species pairs have defined interaction parameters in the `FIELD` file: this is vital for systems with any many-body DPD interactions as universal mixing rules are unavailable for many-body DPD parameters.

Action: Check the `FIELD` file to ensure that unbonded interactions between every possible species pair is defined.

Message 44: no interaction data in FIELD file for single species i

Unbonded interaction data between particle pairs of the same species i are unavailable in the `FIELD` file: mixing rules to determine any missing interaction data thus cannot be applied.

Action: Check the `FIELD` file to ensure that unbonded interactions exist for same-species pairs.

Message 45: zero reciprocal vector range for ewald sum

The maximum reciprocal vector, $\vec{k}_{max}$, has not been defined for systems requiring Ewald sum electrostatics.

Action: Look in the `CONTROL` file and make sure the **ewald** directive includes the convergence parameter α and the extents of the maximum reciprocal vector, k_1, k_2 and k_3.

Message 50: insufficient number of beads per node allocated for required initialization

The value of `maxdim` is not large enough to include the unbonded particles assigned to each processor for a new simulation (without a `CONFIG` file).

Action: This error is unlikely to happen as `maxdim` is calculated according to the numbers of particles and processors available, but the user may wish to use the **densvar** directive in the `CONTROL` file to increase this value.

Message 51: discrepency in total number of starting beads - i too many/few

The total number of particles assigned to all processors for a new simulation does not match up with the numbers specified in the `FIELD` file (taking `nfold` duplication into account if a `CONFIG` file is used).

Action: For simulations without `CONFIG` files, this error should never occur and the authors of DL_MESO should be contacted if it does. If using a `CONFIG` file, check the `FIELD` file to ensure that the number of particles for each species and numbers of molecules match up with those in the `CONFIG` file.

Message 52: cube for molecule i bigger than domain

The maximum extent of molecule i, which is represented as a cube, is larger than the defined size of the system. This is particularly important for systems with hard surfaces or frozen walls as molecules cannot cross these boundaries.

Action: If running a simulation with hard surfaces or frozen walls, either the system size must be increased to accomodate the defined molecule or the molecule needs to be made smaller. If running a simulation without hard surfaces or frozen walls, this is only a warning message: no action is thus required but the user may wish to consider modifying either the system or molecule sizes in future.

Message 53: insufficient number of beads per node allocated for required CONFIG file

The value of `maxdim` is not large enough to include the particles assigned to each processor for a new simulation with a `CONFIG` file.

Action: This error is unlikely to happen as `maxdim` is calculated according to the numbers of particles and processors available (taking into account any `nfold` duplications), but the user may wish to use the **densvar** directive in the `CONTROL` file to increase this value.

Message 54: non-existent species given in CONFIG file for bead i

An undefined species has been found in the definition for the i-th particle in the `CONFIG` file.

Action: Check the species definitions in the `FIELD` file and the i-th particle in the `CONFIG` file to ensure that species is defined.

Message 61: deport coordinate buffers exceeded

The amount of particle data received during deport is greater than the current processor can accommodate.

Action: This error message suggests non-constant particle densities across the system and poor load-balancing. The user may wish to use the **densvar** directive in the `CONTROL` file to increase the value of `maxdim` and thus accommodate larger numbers of particles.

Message 62: deport coordinate buffers exceeded for lees-edwards shear

The amount of particle data received during deport with Lees-Edwards shearing is greater than the current processor can accommodate.

Action: This error message suggests non-constant particle densities across the system and poor load-balancing. The user may wish to use the **densvar** directive in the `CONTROL` file to increase the value of `maxdim` and thus accommodate larger numbers of particles.

Message 71: import coordinate buffers exceeded

The number of additional particles created during import of particle forces is greater than the current processor can accommodate.

Action: This error message suggests non-constant particle densities across the system and poor load-balancing. The user may wish to use the **densvar** directive in the `CONTROL` file to increase the value of `maxdim` and thus accommodate larger numbers of particles.

Message 72: import coordinate buffers exceeded for lees-edwards shear

The number of additional particles created during import of particle forces with Lees-Edwards shearing is greater than the current processor can accommodate.

Action: This error message suggests non-constant particle densities across the system and poor load-balancing. The user may wish to use the **densvar** directive in the `CONTROL` file to increase the value of `maxdim` and thus accommodate larger numbers of particles.

Message 81: export coordinate buffers exceeded

The number of additional particles created during export of particles into boundary halos is greater than the current processor can accommodate.

Action: This error message suggests non-constant particle densities across the system and poor load-balancing. The user may wish to use the **densvar** directive in the `CONTROL` file to increase the value of `maxdim` and thus accommodate larger numbers of particles.

Message 82: export coordinate buffers exceeded for lees-edwards shear

The number of additional particles created during export of particles into boundary halos with Lees-Edwards shearing is greater than the current processor can accommodate.

Action: This error message suggests non-constant particle densities across the system and poor load-balancing. The user may wish to use the **densvar** directive in the `CONTROL` file to increase the value of `maxdim` and thus accommodate larger numbers of particles.

Message 83: cannot correctly export velocities to boundary halos

Particle velocities (for DPD Velocity Verlet integration) cannot be exported correctly to particles already in the boundary halos.

Action: This error should never occur! If it does, please contact the authors of DL_MESO.

Message 84: cannot correctly export densities to boundary halos

Particle densities (for many-body DPD) cannot be exported correctly to particles already in the boundary halos.

Action: This error should never occur! If it does, please contact the authors of DL_MESO.

Message 100: wrong bead total after compression - i too many/few

The total number of particles after the first Velocity Verlet integration stage (including dealing with boundary conditions etc.) does not equal the specified total number of particles for the system.

Action: This error should never occur! If it does, please contact the authors of DL_MESO.

Message 200: bond too long or cannot be found

At least one bond between specified particles is too long (e.g. longer than the maximum specified length for the potential) or cannot be calculated due to lack of available information for both particles. The bond(s) identified as overly long or lost is/are printed either in the `OUTPUT` file or in the standard output (e.g. screen).

Action: If calculating bonds locally, increasing the size of boundary halos may reduce the likelihood of bonds being ‘broken’; alternatively global bond calculations can ensure all data is available at the cost of replication over all processors. Adjusting the parameters for the bond potential may also help ensure bonds do not get too long.

Message 201: too many interacting pairs

The number of interacting pairs for non-DPD thermostats (Lowe-Andersen, Peters, Stoyanov-Groot) exceeds the maximum number calculated from the number of particles in the system, `maxpair`.

Action: The user may wish to use the **densvar** directive in the `CONTROL` file to increase the values of `maxdim` and `maxpair`, thus accommodating larger numbers of interacting pairs.

Messages 1001–1096: allocation/deallocation errors

Allocation or deallocation of arrays for DPD calculations (including reading of input data, transfer buffers for communications between processors, global arrays of particle velocities for Lowe-Andersen/Peters/Stoyanov-

Groot thermostats etc.) has failed. This may be due to a lack of addressable memory required for the DPD calculations. These messages identify which allocation/deallocation has failed by module and routine names.

<u>Action</u>: Increase the amount of memory available for running DL_MESO_DPD by closing any other running applications, running the simulation on a larger number of processors (to reduce the memory required per processor), underpopulating multicore processors (i.e. using fewer cores per processor than the maximum available) or upgrading your machine. Alternatively, try running a smaller simulation.

Appendix D

DL_MESO Licence Agreement (Academic Purposes)

1. DEFINITIONS AND INTERPRETATION

1.1 In this Licence Agreement the following expressions have the meanings set opposite:

Academic Purposes	fundamental or basic research or academic teaching, including any fundamental research that is funded by any public or charitable body, but not any purpose that generates revenue (as opposed to grant income) for the Licensee or any third party. Any research that is wholly or partially sponsored by any profit making organisation or that is carried out for the benefit of any profit-making organisation is not an Academic Purpose;
a Derived Work	any modification of, or enhancement or improvement to, any of the DL_MESO Software and any software or other work developed or derived from any of the DL_MESO Software;
the DL_MESO Software	the release and version of the DL_MESO Software downloaded by the Licensee from the DL_MESO Website immediately after the Licensee agrees to the terms and conditions of this Licence Agreement;
the DL_MESO Website	the website with the URL http://www.ccp5.ac.uk/DL_MESO, and any website that from time to time replaces that website;
a Harmful Element	any virus, worm, time bomb, time lock, drop dead device, trap and access code or anything else that might disrupt, disable, harm or

	impede the operation of any information system, or that might corrupt, damage, destroy or render inaccessible any software, data or file on, or that may allow any unauthorised person to gain access to, any information system or any software, data or file on it;
Intellectual Property	patents, trade marks, service marks, registered designs, copyrights, database rights, design rights, know-how, confidential information, applications for any of the above, and any similar right recognised from time to time in any jurisdiction, together with all rights of action in relation to the infringement of any of the above;
the Licence Period	the period beginning when the Licensee agrees to the terms and conditions of this Licence Agreement and downloads the DL_MESO Software from the DL_MESO Website and ending on the termination of this Licence Agreement under clause 5.2.

2. LICENCE

2.1 STFC grants the Licensee an indefinite, non-exclusive, non-transferable, royalty free licence to use, copy, modify, and enhance the DL_MESO Software during the Licence Period on the terms and conditions of this Licence Agreement provided that:

2.1.1 the Licensee may not distribute any of the DL_MESO Software or any Derived Work to any third party, or share their use with any third party (whether free of charge or otherwise), and the Licensee may not sub-license the use of any of the DL_MESO Software to any third party unless, in each case, that third party has complied with clause 2.3 below;

2.1.2 the Licensee may not use the DL_MESO Software on behalf of, or for the benefit of, any third party (including, without limitation, using it to provide bureau, outsourcing or application services or facilities management services) party unless that third party has complied with clause 2.3 below; and

2.1.3 the DL_MESO Software and any Derived Work may be used by the Licensee and its employees and registered students for Academic Purposes only.

2.2 If the Licensee wishes to use the DL_MESO Software or any Derived Work in any way except for Academic Purposes, or wishes to distribute or make the DL_MESO Software or any Derived Work available to any third party for non-Academic Purposes, it must obtain a commercial licence from STFC. STFC may refuse to grant the Licensee a commercial licence. If STFC agrees to grant a commercial licence, that licence will be on such terms and conditions as STFC sees fit.

2.3 If the Licensee wishes to carry out any collaboration for Academic Purposes with any third party and that third party needs to use the DL_MESO Software in connection with that collaboration, or if the Licensee wishes to make the DL_MESO Software available online to any third party for Academic Purposes, the Licensee must direct that third party to the DL_MESO Website. That third party may use the DL_MESO Software and any Derived Work (whether obtained from STFC or from the Licensee) only if it has completed the registration process on the DL_MESO Website and agreed to the terms and conditions of the Licence Agreement for the use of the DL_MESO Software for Academic Purposes that then appear on the DL_MESO Website.

2.4 This Licence Agreement allows the Licensee to use only the release or version of the DL_MESO Software downloaded by the Licensee from the DL_MESO Website immediately after the Licensee agrees to the terms and conditions of this Licence Agreement; the Licensee must acquire a new licence for any future release or version of the DL_MESO Software that it wishes to use.

2.5 The Licensee will not tamper with, or remove, any copyright or other proprietary notice or any disclaimer that appears on or in any part of the DL_MESO Software, and will reproduce the same in all copies of any of the DL_MESO Software and in all Derived Works.

3. WARRANTIES AND LIABILITY

3.1 The DL_MESO Software is provided for Academic Purposes free of charge. Therefore STFC gives no warranty and makes no representation in relation to the DL_MESO Software or any assistance or advice that STFC may give in connection with the DL_MESO Software. The Licensee will indemnify STFC against any and all claims arising as a result of the Licensee having made any of the DL_MESO Software or any Derived Work available to any third party.

3.2 Before using any of the DL_MESO Software, the Licensee will check that the DL_MESO Software does not contain any Harmful Element. STFC does not warrant that the DL_MESO Software will run without interruption or be error free, or be free from any Harmful Element. STFC is not obliged to provide any support or error correction service, assistance or advice in relation to the DL_MESO Software, but the Licensee may access any error corrections and online assistance that STFC chooses to make available on the DL_MESO Website from time to time. If STFC does provide that sort of service, assistance or advice, subject to clause 3.7, STFC will not be liable for any loss or damage suffered by the Licensee as a result.

3.3 STFC will not be liable to the Licensee to the extent that any loss or damage is caused: by the Licensee's failure to implement, or the Licensee's delay in implementing, any correction or advice in relation to the DL_MESO Software that STFC has made available on the DL_MESO Website; or by the Licensee's failure to acquire a licence of and to implement any new release or version of the DL_MESO Software that would have remedied or mitigated the effects of any error, defect, bug or deficiency in the DL_MESO Software.

3.4 The Licensee acknowledges that proper use of the DL_MESO Software and any Derived Work is dependent on the Licensee, its employees and students exercising proper skill and care in inputting data and interpreting the output provided by the DL_MESO Software or that Derived Work. STFC will not be liable for the consequences of decisions taken by the Licensee or any other person on the basis of that output. STFC does not accept any responsibility for any use which may be made by the Licensee of that output, nor for any reliance which may be placed on that output, nor for advice or information given in connection with that output.

3.5 Subject to clause 3.7, STFC's liability or any breach of this Licence Agreement, any negligence or arising in any other way out of the subject matter of this Licence Agreement or the use of the DL_MESO Software, will not extend to any incidental or consequential damages or losses, or any loss of profits, loss of revenue, loss of data, loss of contracts or opportunity, whether direct or indirect, even if the Licensee has advised STFC of the possibility of those losses arising or if they were or are within STFC's contemplation.

3.6 Subject to clause 3.7, the aggregate liability of STFC for any and all breaches of this Licence Agreement, any negligence or arising in any other way out of the subject matter of this Licence Agreement or the use of the DL_MESO Software will not exceed in total 5000.

3.7 Nothing in this Licence Agreement limits or excludes STFC's liability for death or personal injury caused by its negligence or for any fraud, or for any sort of liability that, by law, cannot be limited or excluded.

3.8 The express undertakings and given by STFC in this Licence Agreement and the terms of this Licence Agreement are in lieu of all warranties, conditions, terms, undertakings and obligations on the part of STFC, whether express or implied by statute,

```
    common law, custom, trade usage, course of dealing or in any
    other way. All of these are excluded to the fullest extent
    permitted by law.

4. INTELLECTUAL PROPERTY RIGHTS AND ACKNOWLEDGEMENTS

4.1 Nothing in this Licence Agreement assigns or transfers any
    Intellectual Property Rights in any of the DL_MESO Software.
    Those rights are reserved to STFC.

4.2 The Licensee will ensure that, if any of its employees or
    students publishes any article or other material resulting from,
    or relating to, a project or work undertaken with the assistance
    of any part of the DL_MESO Software, that publication will
    contain the following acknowledgement:

    "DL_MESO is a mesoscale simulation package written by R. Qin,
    W. Smith and M. A. Seaton and has been obtained from STFC's
    Daresbury Laboratory via the website
    http://www.ccp5.ac.uk/DL_MESO".

5.  TERMINATION

5.1 This Licence Agreement will take effect and the Licence Period
    will start when the Licensee has agreed to the terms and
    conditions of this Licence Agreement and downloaded the DL_MESO
    Software from the DL_MESO Website.

5.2 This Licence Agreement will terminate immediately and
    automatically if:

    5.2.1 the Licensee is in breach of this Licence Agreement; or

    5.2.2 the Licensee becomes insolvent, or if an order is made or a
          resolution is passed for its winding up (except voluntarily
          for the purpose of solvent amalgamation or reconstruction),
          or if an administrator, administrative receiver or receiver
          is appointed over the whole or any part of its assets, or
          if it makes any arrangement with its creditors.

5.3 The Licensee's right to use the DL_MESO Software will cease
    immediately on the termination of this Licence Agreement, and the
    Licensee will destroy all copies of the DL_MESO Software that it
    or any of its employees or students then holds.

5.4 Clauses 1, 2.2, 3, 4, 5.3, 5.4, 5.5 and 6 will survive the expiry
    of the Licence Period and the termination of this Licence
    Agreement, and will continue indefinitely.

5.5 STFC may stop providing any assistance or advice in relation to,
    or any corrections, new releases or versions of the DL_MESO, and
    may stop updating or publishing the DL_MESO Website at any time.

6. GENERAL

6.1 Headings: The headings in this Licence Agreement are for ease of
    reference only; they do not affect its construction or
    interpretation.

6.2 Assignment etc: The Licensee may not assign or transfer this
    Licence Agreement as a whole, or any of its rights or obligations
    under it, without first obtaining the written consent of STFC.

6.3 Illegal/unenforceable provisions: If the whole or any part of any
    provision of this Licence Agreement is void or unenforceable in
    any jurisdiction, the other provisions of this Licence Agreement,
    and the rest of the void or unenforceable provision, will
    continue in force in that jurisdiction, and the validity and
    enforceability of that provision in any other jurisdiction will
    not be affected.

6.4 Waiver of rights: If STFC fails to enforce, or delays in
    enforcing, an obligation of the Licensee, or fails to exercise,
    or delays in exercising, a right under this Licence Agreement,
    that failure or delay will not affect its right to enforce that
    obligation or constitute a waiver of that right.  Any waiver by
    STFC of any provision of this Licence Agreement will not, unless
    expressly stated to the contrary, constitute a waiver of that
    provision on a future occasion.

6.5 Entire agreement: This Licence Agreement constitutes the entire
    agreement between the parties relating to its subject matter.
    The Licensee acknowledges that it has not entered into this
    Licence Agreement on the basis of any warranty, representation,
    statement, agreement or undertaking except those expressly set
    out in this Licence Agreement.  The Licensee waives any claim for
    breach of, or any right to rescind this Licence Agreement in
    respect of, any representation which is not an express provision
    of this Licence Agreement.  However, this clause does not exclude
    any liability which STFC may have to the Licensee (or any right
    which the Licensee may have to rescind this Licence Agreement) in
    respect of any fraudulent misrepresentation or fraudulent
    concealment before the signing of this Licence Agreement.

6.6 Amendments: No variation of, or amendment to, this Licence
    Agreement will be effective unless it is made in writing and
    signed by each party's representative.

6.7 Third parties: No one who is not a party to this Licence
    Agreement has any right to prevent the amendment of this Licence
    Agreement or its termination, and no one except a party to this
    Licence Agreement may enforce any benefit conferred by this
```

Licence Agreement, unless this Licence Agreement expressly provides otherwise.

6.8 Governing law: This Licence Agreement is governed by, and is to be construed in accordance with, English law. The English Courts will have exclusive jurisdiction to deal with any dispute which has arisen or may arise out of or in connection with this Licence Agreement, except that STFC may bring proceedings against the Licensee or for an injunction in any jurisdiction.

Bibliography

[1] Hans C. Andersen. Molecular dynamics simulations at constant pressure and/or temperature. *Journal of Chemical Physics*, 72(4):2384–2393, 1980.

[2] H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, A. DiNola, and J. R. Haak. Molecular dynamics with coupling to an external bath. *Journal of Chemical Physics*, 81(8):3684–3690, 1984.

[3] Gerhard Besold, Ilpo Vattulainen, Mikko Karttunen, and James M. Polson. Towards better integrators for dissipative particle dynamics simulations. *Physical Review E*, 62(6):R7611–R7614, December 2000.

[4] P. L. Bhatnagar, E. R. Gross, and M. Krook. A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems. *Physical Review*, 94(3):511–525, May 1954.

[5] A. Chatterjee. Modification to Lees-Edwards periodic boundary condition for dissipative particle dynamics simulation with high dissipation rates. *Molecular Simulation*, 33(15):1233–1236, 2007.

[6] Shiyi Chen and Gary D. Doolen. Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30(1):329–364, 1998.

[7] Paul J. Dellar. Bulk and shear viscosities in lattice Boltzmann equations. *Physical Review E*, 64(3):031203, August 2001.

[8] Dominique d'Humières, Irina Ginzburg, Manfred Krafczyk, Pierre Lallemand, and Li-Shi Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 360(1792):437–451, 2002.

[9] U. D'Ortona, D. Salin, Marek Cieplak, Renata B. Rybka, and Jayanth R. Banavar. Two-color nonlinear Boltzmann cellular automata: Surface tension and wetting. *Physical Review E: Statistical Physics, Plasmas, Fluids, and related interdisciplinary topics*, 51(4):3718–3728, April 1995.

[10] Burkhard Dünweg and Wolfgang Paul. Brownian dynamics simulations without Gaussian random numbers. *International Journal of Modern Physics C*, 2(3):817–827, 1991.

[11] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the Navier-Stokes equation. *Physical Review Letters*, 56(14):1505–1508, April 1986.

[12] A. Ghoufi and P. Malfreyt. Mesoscale modeling of the water liquid-vapor interface: A surface tension calculation. *Physical Review E*, 83:051601, May 2011.

[13] J. B. Gibson, K. Chen, and S. Chynoweth. The equilibrium of a velocity-Verlet type algorithm for DPD with finite time steps. *International Journal of Modern Physics C*, 10(1):241–261, February 1999.

[14] Minerva González-Melchor, Estela Mayoral, María Eugenia Velázquez, and José Alejandre. Electrostatic interactions in dissipative particle dynamics using the Ewald sums. *Journal of Chemical Physics*, 125(22):224107, 2006.

[15] R. D. Groot. Electrostatic interactions in dissipative particle dynamics—simulation of polyelectrolytes and anionic surfactants. *Journal of Chemical Physics*, 118(24):11265–11277, 2003.

[16] Robert D. Groot and Patrick B. Warren. Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation. *Journal of Chemical Physics*, 107(11):4423–4435, 1997.

[17] Andrew J. Gunstensen, Daniel H. Rothman, Stéphane Zaleski, and Gianluigi Zanetti. Lattice Boltzmann model of immiscible fluids. *Physical Review A*, 43(8):4320–4327, April 1991.

[18] Zhaoli Guo, Baochang Shi, and Chuguang Zheng. A coupled lattice BGK model for the Boussinesq equations. *International Journal for Numerical Methods in Fluids*, 39(4):325–342, 2002.

[19] Zhaoli Guo, Chuguang Zheng, and Baochang Shi. Discrete lattice effects on the forcing term in the lattice Boltzmann method. *Physical Review E: Statistical, Nonlinear and Soft Matter Physics*, 65(4):046308, April 2002.

[20] I. Halliday, A. P. Hollis, and C. M. Care. Lattice Boltzmann algorithm for continuum multicomponent flow. *Physical Review E: Statistical, Nonlinear and Soft Matter Physics*, 76(2):026708, 2007.

[21] I. Halliday, R. Law, C. M. Care, and A. Hollis. Improved simulation of drop dynamics in a shear flow at low Reynolds and capillary number. *Physical Review E: Statistical, Nonlinear and Soft Matter Physics*, 73(5):056708, 2006.

[22] Cecil Hastings. *Approximations for digital computers.* Princeton University Press, Princeton, NJ, 1955.

[23] Xiaoyi He, Shiyi Chen, and Gary D. Doolen. A novel thermal model for the lattice Boltzmann method in incompressible limit. *Journal of Computational Physics*, 146(1):282–300, 1998.

[24] Xiaoyi He and Li-Shi Luo. Lattice Boltzmann model for the incompressible Navier-Stokes equation. *Journal of Statistical Physics*, 88(3–4):927–944, 1997.

[25] F. J. Higuera and J. Jiménez. Boltzmann approach to lattice gas simulations. *EPL (Europhysics Letters)*, 9(7):663–668, 1989.

[26] R. W. Hockney and J. W. Eastwood. *Computer simulation using particles.* McGraw-Hill International, 1981.

[27] Takaji Inamuro, Masato Yoshino, Hiroshi Inoue, Riki Mizuno, and Fumimaru Ogino. A lattice Boltzmann method for a binary miscible fluid mixture and its application to a heat-transfer problem. *Journal of Computational Physics*, 179(1):201–215, 2002.

[28] Takaji Inamuro, Masato Yoshino, and Fumimaru Ogino. A non-slip boundary condition for lattice Boltzmann simulations. *Physics of Fluids*, 7(12):2928–2930, 1995.

[29] Ask F. Jakobsen. Constant-pressure and constant-surface tension simulations in dissipative particle dynamics. *Journal of Chemical Physics*, 122(12):124901, 2005.

[30] J. E. Jones. On the determination of molecular fields. II. From the equation of state of a gas. *Proceedings of the Royal Society of London. Series A*, 106(738):463–477, October 1924.

[31] Simon Jury, Peter Bladon, Mike Cates, Sujata Krishna, Maarten Hagen, Noel Ruddock, and Patrick Warren. Simulation of amphiphilic mesophases using dissipative particle dynamics. *Physical Chemistry Chemical Physics*, 1(9):2051–2056, 1999.

[32] J. M. V. A. Koelman and P. J. Hoogerbrugge. Dynamic simulations of hard-sphere suspensions under steady shear. *EPL (Europhysics Letters)*, 21(3):363–368, 1993.

[33] Pierre Lallemand and Li-Shi Luo. Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invariance, and stability. *Physical Review E*, 61(6):6546–6562, June 2000.

[34] A. W. Lees and S. F. Edwards. The computer study of transport processes under extreme conditions. *Journal of Physics C: Solid State Physics*, 5(15):1921–1928, 1972.

[35] S. V. Lishchuk, C. M. Care, and I. Halliday. Lattice Boltzmann algorithm for surface tension with greatly reduced microcurrents. *Physical Review E: Statistical, Nonlinear and Soft Matter Physics*, 67(3):036701, March 2003.

[36] C. P. Lowe. An alternative approach to dissipative particle dynamics. *EPL (Europhysics Letters)*, 47(2):145–151, July 1999.

[37] John F. Marko and Eric D. Siggia. Stretching DNA. *Macromolecules*, 28(26):8759–8770, 1995.

[38] G. Marsaglia and T. A. Bray. A convenient method for generating normal variables. *SIAM Review*, 6(3):260–264, July 1964.

[39] George Marsaglia, Arif Zaman, and Wai Wan Tsang. Toward a universal random number generator. *Statistics and Probability Letters*, 9(1):35–39, January 1990.

[40] C. A. Marsh, G. Backx, and M. H. Ernst. Static and dynamic properties of dissipative particle dynamics. *Physical Review E*, 56(2):1676–1691, August 1997.

[41] Nicos S. Martys and Hudong Chen. Simulation of multicomponent fluids in complex three-dimensional geometries by the lattice Boltzmann method. *Physical Review E*, 53(1):743–750, January 1996.

[42] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[43] Keijo Mattila, Jari Hyväluoma, Tuomo Rossi, Mats Aspnäs, and Jan Westerholm. An efficient swap algorithm for the lattice Boltzmann method. *Computer Physics Communications*, 176(3):200–210, February 2007.

[44] Philip M. Morse. Diatomic molecules according to the wave mechanics. II. Vibrational levels. *Physical Review*, 34(1):57–64, July 1929.

[45] David R. Noble, Shiyi Chen, John G. Georgiadis, and Richard O. Buckius. A consistent hydrodynamic boundary condition for the lattice Boltzmann method. *Physics of Fluids*, 7(1):203–209, 1995.

[46] I. Pagonabarraga and D. Frenkel. Dissipative particle dynamics for interacting systems. *Journal of Chemical Physics*, 115(11):5015–5026, 2001.

[47] E. A. J. F. Peters. Elimination of time step effects in DPD. *EPL (Europhysics Letters)*, 66(3):311–317, May 2004.

[48] Igor V. Pivkin and George Em Karniadakis. A new method to impose no-slip boundary conditions in dissipative particle dynamics. *Journal of Computational Physics*, 207(1):114–128, 2005.

[49] Kannan N. Premnath and John Abraham. Three-dimensional multi-relaxation time (MRT) lattice-Boltzmann models for multiphase flow. *Journal of Computational Physics*, 224(2):539–559, 2007.

[50] P. Prinsen, P. B. Warren, and M. A. J. Michels. Mesoscale simulations of surfactant dissolution and mesophase formation. *Physical Review Letters*, 89(14):148302, September 2002.

[51] M. Revenga, I. Zúñiga, and P. Español. Boundary conditions in dissipative particle dynamics. *Computer Physics Communications*, 121–122:309–311, 1999. Proceedings of the Europhysics Conference on Computational Physics CCP 1998.

[52] A. G. Schlijper, P. J. Hoogerbrugge, and C. W. Manke. Computer simulation of dilute polymer solutions with the dissipative particle dynamics method. *Journal of Rheology*, 39(3):567–579, 1995.

[53] Xiaowen Shan. Analysis and reduction of the spurious current in a class of multiphase lattice Boltzmann models. *Physical Review E*, 73(4):047701, 2006.

[54] Xiaowen Shan. Pressure tensor calculation in a class of nonideal gas lattice Boltzmann models. *Physical Review E*, 77(6):066702, June 2008.

[55] Xiaowen Shan and Hudong Chen. Lattice Boltzmann model for simulating flows with multiple phases and components. *Physical Review E: Statistical Physics, Plasmas, Fluids, and related interdisciplinary topics*, 47(3):1815–1819, March 1993.

[56] Xiaowen Shan and Hudong Chen. Simulation of nonideal gases and liquid-gas phase transitions by the lattice Boltzmann equation. *Physical Review E: Statistical Physics, Plasmas, Fluids, and related interdisciplinary topics*, 49(4):2941–2948, April 1994.

[57] W. Smith. Molecular dynamics on hypercube parallel computers. *Computer Physics Communications*, 62(2-3):229–248, March 1991.

[58] W. Smith. A replicated data molecular dynamics strategy for the parallel Ewald sum. *Computer Physics Communications*, 67(3):392–406, January 1992.

[59] W. Smith. Calculating the pressure. *CCP5 Information Quarterly*, 39:14–20, October 1993.

[60] W. Smith, T. R. Forester, and I.T. Todorov. *The DL_POLY Classic user manual.* STFC, STFC Daresbury Laboratory, Daresbury, Warrington, Cheshire, WA4 4AD, United Kingdom, version 1.0 edition, December 2010.

[61] T. J. Spencer, I. Halliday, and C. M. Care. Lattice Boltzmann equation method for multiple immiscible continuum fluids. *Physical Review E*, 82(6):066701, December 2010.

[62] Simeon D. Stoyanov and Robert D. Groot. From molecular dynamics to hydrodynamics: A novel Galilean invariant thermostat. *Journal of Chemical Physics*, 122(11):114112, 2005.

[63] Sauro Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond.* Clarendon Press, Oxford, 2001.

[64] Michael R. Swift, W. R. Osborn, and J. M. Yeomans. Lattice Boltzmann simulation of nonideal fluids. *Physical Review Letters*, 75(5):830–833, July 1995.

[65] I. T. Todorov and W. Smith. *The DL_POLY_4 user manual.* STFC, STFC Daresbury Laboratory, Daresbury, Warrington, Cheshire, WA4 4AD, United Kingdom, version 4.01.0 edition, October 2010.

[66] S. Y. Trofimov, E. L. F. Nies, and M. A. J. Michels. Thermodynamic consistency in dissipative particle dynamics simulations of strongly nonideal liquids and liquid mixtures. *Journal of Chemical Physics*, 117(20):9383–9394, 2002.

[67] Loup Verlet. Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Physical Review*, 159(1):98–103, July 1967.

[68] P. B. Warren. Vapor-liquid coexistence in many-body dissipative particle dynamics. *Physical Review E*, 68(6):066702, December 2003.

[69] John D. Weeks, David Chandler, and Hans C. Andersen. Role of repulsive forces in determining the equilibrium structure of simple liquids. *Journal of Chemical Physics*, 54(12):5237–5247, 1971.

[70] Dean R. Wheeler, Norman G. Fuller Rowley, and Richard L. Non-equilibrium molecular dynamics simulation of the shear viscosity of liquid methanol: adaptation of the Ewald sum to Lees-Edwards boundary conditions. *Molecular Physics*, 92(1):55–62, 1997.

[71] Satoru Yamamoto, Yutaka Maruyama, and Shi-aki Hyodo. Dissipative particle dynamics study of spontaneous vesicle formation of amphiphilic molecules. *Journal of Chemical Physics*, 116(13):5842–5849, 2002.

[72] M. Yoshino and T. Inamuro. Lattice Boltzmann simulations for flow and heat/mass transfer problems in a three-dimensional porous structure. *International Journal for Numerical Methods in Fluids*, 43(2):183–198, 2003.

[73] Qisu Zou and Xiaoyi He. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Physics of Fluids*, 9(6):1591–1598, June 1997.